ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

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

Evaluation von Hybrid Mobile App Frameworks im Kontext von medizinischen Applikationen

Bachelorarbeitan der Universität Ulm

Vorgelegt von:
Julius Friedrich
julius.friedrich@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2017

Fassung 17. März 2017

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Medizinisch unterstützende Applikationen sind Anwendungen, die einen Teil der ärztlichen Behandlung von Patienten sowie deren Betreuung unterstützen oder sogar abnehmen sollen. Neben den schon existierenden Anwendungsbereichen von mobilen Applikationen, bietet sich auch dieser Anwendungsbereich durch die steigende Anzahl an Nutzern von Mobilgeräten an. Diese Applikationen bergen jedoch auch Gefahren und Risiken für den Patienten. Um diese Gefahren und Risiken einschätzen und minimieren zu können, muss in diesem Gebiet geforscht werden. Dafür werden Applikationen benötigt, mit denen Tests und Studien durchgeführt werden können. Für die Entwicklung von solchen Test-Applikationen, können hybride Frameworks durch ihren schnellen Entwicklungsvorgang vorteilhaft sein. Durch das Aufkommen neuer Mobile App Frameworks wird die Entscheidung für oder gegen ein Framework zunehmend schwieriger.

Ziel dieser Arbeit ist es zu untersuchen, welches Framework sich für das Erstellen von medizischen, beziehungsweise im erweiterten Sinne Patienten betreuende Applikationen, am besten eignet. Dazu werden grundlegende und repräsentative Funktionen der Applikation mit den Frameworks Meteor und Ionic 2. Anschließend werden die entstandenen Applikationen analysiert und evaluiert. Weiterhin enthält die Arbeit eine begründete Empfehlung für das Framework Ionic 2 und eine Diskussion darüber, welche weiteren Aspekte bei der Entwicklung von medizinischen Applikationen beachtet werden sollten.

Inhaltsverzeichnis

1

Einleitung

In der Medizin gibt es schon seit langer Zeit Software, die Ärzte bei ihrer Arbeit unterstützt. Beispielsweise braucht ein Ultraschallgerät entsprechende Software. Das Interesse an technischer Unterstützung ist groß, da sie arbeitserleichternd sein kann. Mit der steigenden Anzahl von Smartphone-Nutzern bietet sich die Möglichkeit, medizinische Behandlung in Form von mobilen Applikationen zu unterstützen. Mit dieser Medizinische Applikationen können verschiedene Vorteile mit sich bringen. Ein Arzt hätte durch Zeitersparnisse größere Kapazitäten für mehr Patienten, Kommunikationswege könnten kürzer sein und bürokratischer Aufwand verringert werden. Auf der anderen Seiten bergen solche Applikationen auch Gefahren, wie zum Beispiel Fehler bei der Datenübertragung oder Fehler in der Software. Das Institut für Datenbanken und Informationssysteme der Universität Ulm beschäftigt sich mit der Entwicklung von medizinisch unterstützenden Applikationen und möchte eine solche Applikation implementieren, um dieses Gebiet weiter zu erforschen. Beispielsweise wurde bereits daran geforscht, wie *Wearables*, wie zum Beispiel Smartwatches, im medizinischen Kontext verwendet werden können (vergleiche [1]). Medizinische Applikationen können aber auch direkt behandelnd sein und nicht nur unterstützend. Zum Beispiel wurden Applikationen entwickelt, die Tinnitus-Patienten helfen, ihre Lokalisierungsfähigkeit zurück zu erlangen oder zu verbessern. Im Kontext *Serious Games* wurde eine solche Applikation bereits entwickelt (siehe [2]). Der größte Aufwand in der Entwicklung vieler Applikationen entsteht bei der Unterstützung von mehreren Plattformen. Meist wird eine Applikation für eine Plattform erstellt und dann für andere Plattformen nachgebaut. Dieses Vorgehen kann speziell in der Wartung sehr zeitaufwändig sein.

Einen anderen Ansatz bieten Hybride Frameworks, die mit nur einem Quellcode arbeiten, der dann für auf verschiedenen Plattformen funktioniert. Dahinter steckt die Idee, nur eine Applikation zu implementieren und auch nur einen Code zu warten. Diese Technologie ist jedoch relativ neu und es sind im Zeitraum der letzten Jahre viele neue Frameworks mit Fokus auf unterschiedliche Funktionen und Ziele entstanden.

1.1 Problemstellung

Beim Entwurf einer mobilen Applikation gilt es im Vorhinein verschiedene Vorkehrungen und Entscheidungen zu treffen. Neben der genauen Definition der funktionalen und nicht funktionalen Anforderungen, ist vor allem die Frage nach dem richtigen Werkzeug zu beantworten. Es muss also anhand der Anforderungen festgelegt werden, welches Framework, oder im weiteren Sinne, welche Programmiersprache gewählt werden sollte. Möglich sind zum einen native Applikationen, die speziell für ein Betriebssystem programmiert werden. Zum Beispiel eine native Android-Applikation für Mobilgeräte mit einem Android-Betriebssystem. Da der Fokus dieser Forschung auf dem medizinischen Aspekt liegen soll, ist es vorteilhaft, den technischen, beziehungsweise den die Programmierung betreffenden Aufwand zu reduzieren. Dieser Aufwand kann durch die Wahl eines hybriden Frameworks reduziert werden. Für eine solche medizinische Applikation gilt es nun ein Framework zu finden, das die Anforderungen bestmöglich erfüllen kann.

1.2 Zielsetzung

In dieser Arbeit sollen zunächst die beiden Frameworks Meteor und Ionic 2 vorgestellt werden. Anschließend sollen die genauen Anforderungen an die Applikation definiert werden. Aus diesen Anforderungen sollen Funktionen abgeleitet werden, die repräsentativ für die Anforderungen sind. Diese abgeleiteten Funktionen sollen mit Hilfe der verschiedenen Frameworks umgesetzt werden.
Darauf werden die verschiedenen Implementierungen der Funktionen in Form einer mobilen Applikation analysiert und verglichen. Anhand dieser Erkenntnisse soll evaluiert

werden, welches Framework sich am besten eignet, um abschließend eine fundierte Empfehlung für die reelle Umsetzung einer Applikation zur Patientenbetreuung abzugeben.

1.3 Struktur der Arbeit

Zunächst werden einige Grundlagen, die für das Verständnis von hybriden Frameworks nötig sind, in Kapitel 2 geklärt. Danach folgt in Kapitel 3 eine umfangreiche Einführung in die Funktionsweisen der Frameworks Meteor und Ionic 2. Anschließend werden die funktionalen und nicht-funktionalen Anforderungen an die Applikation in Kapitel 4 spezifiziert. In Kapitel 5 werden die letzten Vorkehrungen für die Implementierung in Form von einer Konzeption und von Mock-ups. Nach der Implementierung werden in diesem Kapitel dann die Applikationen vorgestellt. Nach der Realisierung folgt in Kapitel 6 nun die Evaluation. Dabei werden aus den Kenntnissen, die bei der Realisierung der Applikationen gewonnen wurden, dazu verwendet, um die Frameworks zu bewerten. Abschließend folgt in Kapitel 7 ein Ausblick.

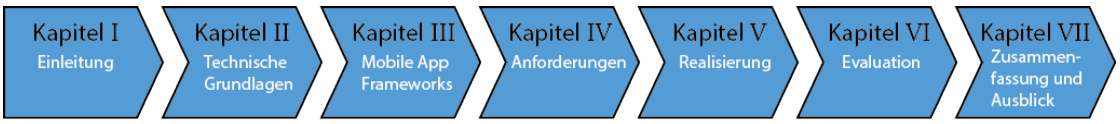

Abbildung 1.1: Struktur der Arbeit

2

Technische Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die für das Verständnis dieser Arbeit erforderlich sind. Zum einen wird darauf eingegangen, wie eine REST-API und eine mobile Applikation untereinander kommunizieren können und was Cordova ist, beziehungsweise wie Cordova genutzt werden kann.

2.1 REST-Services

Representational **S**tate **T**ransfer wurde im Jahr 2000 von Roy Fielding in seiner Dissertation [3] als Architekturstil eines Webservices definiert. Webservices benutzen die REST-Architektur für die Datenübertragung zwischen Client und Webserver und für die Darstellung von Programmzuständen. Die Datenübertragung beim REST basiert auf dem HTTP-Protokoll. Die Architektur eines REST-Services ist in Abbildung 2.1 dargestellt.

Abbildung 2.1: Architektur eines REST-Services. Entnommen aus [4]

2.1.1 Funktionsweise

Webservices die eine REST-Architektur verwenden, bestehen aus einer Web-API und einem Backend, das von der API angesprochen werden kann. Ein simples Anwendungsszenario ist ein Client, der Ressourcen vom Webservice anfragt. Jede Ressource hat dabei einen eigenen, eindeutigen Identifier, in Form eines URI (**U**niform **R**essource **I**dentifier). Im Fall eines Webservices haben Ressourcen eine URL (**U**niform **R**essource **L**ocator).
Ein einfaches Beispiel für eine URL ist in Listing 2.1 gegeben. Die Web-API stellt dabei dem Client eine Route zur Verfügung, in diesem Beispiel eine statische Route. Der Webservice kann nun die Anfrage (Request) des Clients mit einer Antwort (Response), in der die Ressource enthalten ist, beantworten.

```
1 example.com/ressource1
```

Listing 2.1: Beispiel für einen Uniform Ressource Locator

Bei der Kommunikation mit dem Server kann der Client zwischen den verschiedenen HTTP-Methoden wählen. Bei der realen Verwendung kommen zumeist die GET-Methode, beispielsweise zum Anfragen von Ressourcen, und die POST-Methode, beispielsweise zum Übertragen von Formulardaten, zum Einsatz. Außerdem existieren auch die PUT- und die DELETE-Methode, die zum Hochladen, Ersetzen und Löschen von Ressourcen verwendet werden können. In den meisten Webservices werden PUT und DELETE aber nur selten verwendet, da das Verwalten von Ressourcen als sicherheitskritisch eingestuft wird und deshalb oftmals vom Backend übernommen wird.

Eine REST-API kann auch als Instrument verwendet werden, das komplexere Vorgänge im Backend des Webservices steuert. Zum Beispiel kann beim Senden von Daten an eine bestimmte URL der REST-API, ein Vorgang im Backend gestartet werden, der dann eine Datenbankoperation mit den erhaltenen Daten durchführt. Bei der Übertragung von Daten ist es wichtig, dass Client und Server ein einheitliches Datenformat verwenden.

2.1.2 JSON-Format

Ein mögliches Format, um einen einheitlichen und konsistenten Datenaustausch zu gewährleisten, ist das JSON-Format. JSON ist eine Abkürzung für **J**ava**S**cript **O**bject **N**otation und wurde in RFC 7195 [5] von Douglas Crockford spezifiert. Das Format wird hauptsächlich bei der Kommunikation zwischen Anwendungen verwendet.

Die Datensätze werden in Textform abgespeichert und sind so auch für den Menschen leicht analysierbar. Ein wichtiger Vorteil ergibt sich daraus, dass JSON-Datensätze sehr kompakt sind und dadurch nur einen geringen Overhead an zusätzlichen Informationen haben, beziehungsweise brauchen. Dieser Vorteil ergibt sich nicht zuletzt aus der Strategie, *Key-Value*-Paare zu verwenden. Dabei ist es neben einfachen Datentypen, wie Booleans oder Integern, auch möglich, Arrays und komplexe Javascript Objekte im JSON-Format zu serialisieren. Ein Beispiel für ein JSON-Datensatz ist in Listing 2.2 gegeben, die genaue Syntax ist in RFC 7195 [5] nachzulesen.

```
1  {"house": {
2    "id": "1",
3    "height": "2.8",
4    "floorArea": "120",
5    "rooms": [
6        {"type": "kitchen", "floorArea": "30"},
7        {"type": "bathroom", "floorArea": "20"},
8        {"type": "livingroom", "floorArea": "70"}
9    ]
10 }}
```

Listing 2.2: Ein Beispiel für einen JSON-Datensatz

2.2 Cordova

Apache Cordova (früher PhoneGap) ist eine Open-Source Software, die es ermöglicht, mobile Applikationen auf der Basis von moderner Webtechnologie wie HTML5, CSS3 und Javascript zu programmieren. Cordova ist ein *Cross-Plattform-Framework*. Das bedeutet, dass eine Cordova Applikation auf mehreren Betriebssystemen genutzt werden kann. Unterstützt werden hauptsächlich die Betriebssysteme Android, iOS und Windows Phone.
Die resultierenden Applikationen sind sogenannte Hybride. Es sind auf der einen Seite keine Web-Applikationen, da sie nicht im gewöhnlichen mobilen Browser, sondern als eigene Applikation installiert sind. Außerdem haben Cordova Applikationen die Möglichkeit auf Hardware, wie beispielsweise auf die Kamera oder die GPS-Position, zuzugreifen. Auf der anderen Seite sind es offensichtlich auch keine nativen Applikationen, da das gesamte Rendering in einem Webview (eine Art Browser-Fenster ohne erweiterte Bedienoberfläche) geschieht.

Eine Cordova Applikation besteht, wie in Abbildung 2.2 gezeigt wird aus mehreren Modulen: der Web-Applikation selbst, einem WebView und aus verschiedenen Plugins. Die Web-Applikation, bestehend aus HTML, CSS und Javascript, enthält die Logik der Anwendung. Dieser Teil der Applikation funktioniert auch eigenständig in einem modernen Browser und gleicht der Form einer normalen Web-Applikation.

Die Web-Applikation läuft aber unter Cordova in einem modifizierten WebView, der im Gegensatz zu einem normalen mobilen Browser, zusätzliche Schnittstellen bietet, um so auf spezielle Hardware des Mobilgerätes zuzugreifen. Für diese erweiterte Kommunikation mit dem WebView stellt Cordova eine eigene API zur Verfügung.

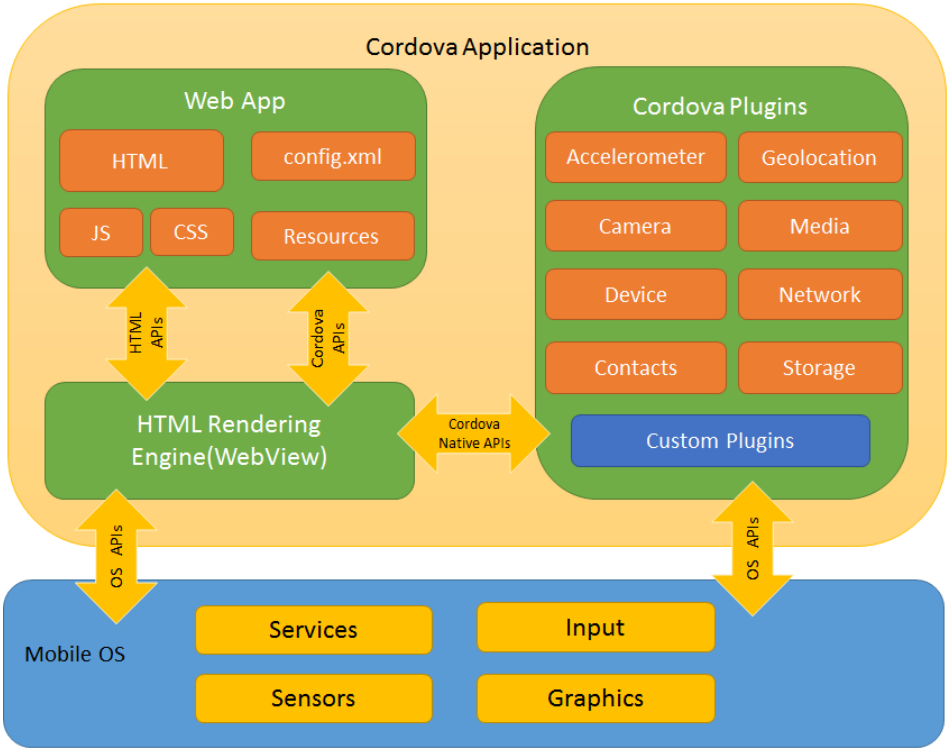

Abbildung 2.2: Funktionsweise von Cordova in einem Schaubild. [6]

Die Kommunikation mit der Hardware funktioniert aber nicht über den direkten Weg mit der Hardware-Komponente, sondern über die Cordova Plugins. Jede Hardware-Komponente wird über ein passendes Plugin angesprochen. Diese Plugins enthalten nativen Code für alle unterstützen Betriebssysteme und können mit entsprechenden Berechtigungen auf die Hardware-Komponenten zugreifen. Der WebView stellt also der Web-Applikation eine API zur Verfügung, um Plugins zu bedienen. Die Plugins selbst kommunizieren dann letztendlich mit den Hardware-Komponenten.
Eine weitere Möglichkeit zur Kommunikation mit Hardware-Komponenten bieten die HTML5-APIs, die auch in mobilen Browsern funktionieren. Dabei kann der WebView, wie auch ein Browser, auf einige Hardware-Komponenten zugreifen ohne ein Plugin zu verwenden. Als Beispiel kann auf das Mikrofon über die HTML5 Web-Audio-API zugegriffen werden, nachdem eine Berechtigung erteilt wurde. Jedoch ist ein Zugriff auf das Filesystem nicht standardmäßig mit einer der HTML5-APIs möglich. Diese

Kommunikation muss über ein entsprechendes Plugin, in diesem Fall das Cordova *File Plugin*, erfolgen. Die längerfristige Entwicklung deutet aber darauf hin, dass sich die HTML5-APIs weiterentwickeln und in Zukunft mehr Möglichkeiten bieten, wie zum Beispiel auch den Zugriff auf das Filesystem eines mobilen Gerätes.

3

Mobile App Frameworks

Die Applikation soll mit Hilfe von verschiedenen hybriden Mobile App Frameworks umgesetzt werden. Hybride Apps sind Web-Applikationen die auf mobilen Geräten laufen, mit der Möglichkeit auf Hardwarekomponenten des Endgerätes zuzugreifen. Dabei setzen hybride Applikationen auf Webtechnologien wie HTML5, CSS3 und Javascript und laufen dann in einer Art Browser, einem sogenannten Webview. Die Architektur der Applikationen verfolgt das MVVC-Pattern (Model-View-ViewModel). Abhängig von der jeweiligen Plattform des Endgerätes laufen die Applikation über unterschiedliche Webviews. Zum Beispiel nutzen Geräte mit Android als Betriebssystem den einem Chromium Webview. Hybride Mobile Apps funktionieren dann auf mehreren Plattformen, sind aber genau genommen nicht *cross plattform*. *Cross plattform* Applikationen teilen sich zwar über mehrere Plattformen einen Großteil des geschriebenen Codes, aber die Benutzeroberfläche wird für jede Plattform durch nativen Code zusammengebaut. Einige Cross-Plattform-Frameworks bieten auch Möglichkeiten, den Code, der die Benutzeroberfläche betrifft, zwischen den Plattformen zu teilen. Ein solches Cross-Plattform-Framework ist beispielsweise Xamarin. Es bietet mit seinem Feature *Xamarin.Forms* eine spezielle XAML-Syntax für graphische Elemente, die dann für jede Plattform zur Laufzeit das XAML in nativen Code umwandelt. Ein Beispiel für ein Cross-Plattform-Framework ohne zusätzliche XAML-Syntax ist React Native. In dieser Arbeit werden die hybriden Mobile App Frameworks **Meteor** und **Ionic 2** untersucht.

3.1 Meteor

Meteor ist ein Javascript Web-Framework für Webanwendungen und mobile Applikationen. 2011 wurde Meteor unter dem Namen *Skybreak* veröffentlicht und bekam in 2012 seinen jetzigen Namen. Im Gegensatz zu vielen anderen hybriden Frameworks, bietet Meteor die Möglichkeit zu jeder Applikation einen eng verknüpften Server zu programmieren. Zu den wichtigsten Bestandteilen von Meteor gehören *MongoDB* als Datenbank und das DDP (**D**istributed **D**ata **P**rotocol), welches zur Kommunikation zwischen der Client-Applikation und dem Server dient. Mobile Anwendungen werden durch die Unterstützung von Cordova (siehe 2.2) erstellt.

Meteor verfolgt bei Anwendungen vier Grundprinzipien [7]. Meteor Applikationen benutzen nur **eine Programmiersprache** in allen Meteor Umgebungen. Das bedeutet sowohl in der mobilen und webbasierten Client-Applikation, als auch auf Seiten des Servers, wird Javascript verwendet. Weiterhin verfolgt Meteor das **data on the wire**-Konzept, was mit Hilfe des DDPs umgesetzt wird. Das bedeutet, dass zwischen Client-Applikation und Server nur Daten gesendet werden und keine vollständigen HTML-Seiten.
Mit dem eigenen *Package*-System, für das alle Benutzer eigene autonome Programmteile erstellen und der Allgemeinheit zur Verfügung stellen können, verfolgt Meteor die Strategie **embrace the ecosystem** und bindet damit aktiv die Community in die Plattform ein. Das letzte Prinzip besteht in **full stack reactivity**. Dadurch ist es mit Meteor möglich, den aktuellen Zustand der Applikation mit minimalem Aufwand im User Interface darzustellen.

3.1.1 Architektur

Meteor Applikationen bestehen aus mehreren Ebenen und Komponenten, die miteinander kommunizieren. Sie teilen sich in die Client-Ebene und in die Server-Ebene auf. Von der Server-Ebene können Datenbanken und externe REST-Services angesprochen werden. In Abbildung 3.1 ist die Architektur einer Meteor-Applikation veranschaulicht.

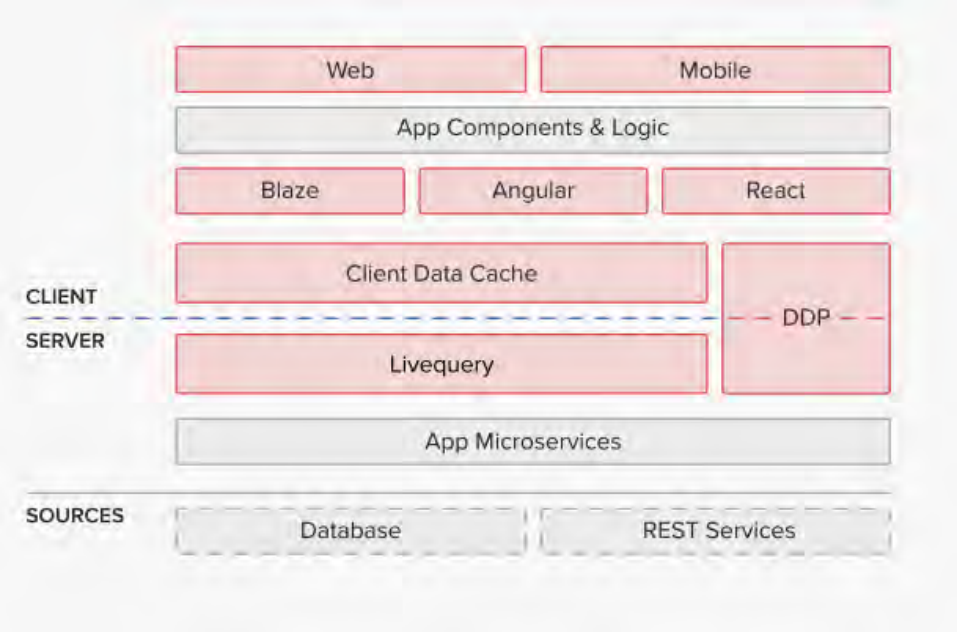

Abbildung 3.1: Architektur einer Meteor-Applikation. Entnommen aus [8]

User Interface

Meteor unterstützt unabhängig von der Zielplattform verschiedene *UI-Engines*. Während in frühen Versionen *Blaze* als einzige *UI-Engine* zur Auswahl stand, sind mittlerweile auch das von Google entwickelte *Angular* und das von Facebook entwickelte*React* eine Option.

Blaze

BlazeJS ist eine *Rendering Library* für User Interfaces und wurde 2011 als Teil von Meteor entwickelt. Blaze besteht zu einem großen Teil aus der Template-Engine *Spacebars*, einer Weiterentwicklung von *Handlebars* (siehe [9]). Spacebars erweitert einfaches HTML um reaktive Elemente durch spezielle Syntax. Im Beispiel 3.1 ist ein Template mit dem Namen **example** definiert. Dabei wurden bekannte HTML-Tags wie **<h1>** durch Elemente mit doppelt geschweiften Klammern erweitert.

```
<template name="example">
  <h1>{{title}}</h1>
  <div>
      {{#if birthday}}
        Happy Birthday!
      {{/if}}
  </div>
</template>
```

Listing 3.1: Beispiel für durch Spacebars erweitertes HTML

Dieses Template wird dann von der Template-Engine auf die erweiterte Syntax durchsucht und ersetzt die Spacebars-Elemente durch entsprechendes valides HTML. Die Logik für die Spacebars-Elemente wird in Javascript in Form von sogenannten *Helpern* definiert (siehe Beispiel 3.2). Für das Template **example** werden zwei *Helper* definiert, die von der Template-Engine dann zum Ersetzen der Spacebars-Elemente verwendet werden. Dabei kann ein *Helper* nicht nur durch einfache Variablen, sondern auch durch eine komplexe Funktion definiert werden. Zur Laufzeit wird so das Template mit Hilfe der Helper kompiliert.

```
Template.example.helpers({
  title : "Welcome!",
  birthday : function(){
    return userHasBirthday(); //diese Methode sei an anderer
  }                           //Stelle definiert.
});
```

Listing 3.2: Beispiel für Spacebars-Helper

Die Template-Engine kompiliert aber nicht nur einmalig, sondern jedes mal wenn sich die Abhängigkeit eines Helpers ändert. Man nennt diese Strategie **data-binding**. Dabei rendert die Template-Engine Spacebars-Elemente neu, wenn sich die Daten, von denen der Helper abhängig ist, ändern.

Angular

AngularJS ist ein Framework zum Erstellen von Client-Applikationen und wurde 2009 von Google veröffentlicht. 2016 wurde dann Angular (auch bekannt als Angular 2) veröffentlich, als neue Version mit überarbeiteten Konzepten und einer neuen Architektur. Meteor integriert sowohl AngularJS, als auch das aktuelle Angular. Die genaue Funktionsweise wird bei der Vorstellung von Ionic 2 in Kapitel 3.2 erklärt.

React

React ist ebenfalls eine Javascript-Bibliothek zum Erstellen von reaktiven User Interfaces und wurde 2013 von Facebook veröffentlicht. Ähnlich zu Angular verfolgt React den Ansatz, Applikationen in modulare *Components* einzuteilen. React wird in Javascript oder in JSX programmiert. In JSX wird Javascript mit eingebetteter XML-Syntax kombiniert. Das bedeutet, im Javascript-Code können XML-Tags verwendet werden, ohne diese in einen String zu schreiben. Für JSX-Files gibt es momentan noch keinen weitreichenden Browser-Support, weshalb es verschiedene Kompilatoren gibt, die JSX in pures Javascript umwandeln.

React-*Components* bestehen aus einem Konstruktor, einem State und einer Render-Funktion. Der State spiegelt den momentanen Zustand des *Components* wieder in Form von Variablen. Der Konstruktor kann verwendet werden, um den *Component* in einen initialen State zu versetzen. Die Render-Funktion ist dem Template in Angular ähnlich, denn sie enthält HTML und kann durch spezielle Syntax auch Variablen des States darstellen. Wenn sich der State verändert, wird abhängiges HTML aktualisiert.

```
class Message extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: "Hello World!"};
  }
  updateMessage(){
    this.setState((prevState) => ({
      message: "You just clicked!"
    }));
  }
  render() {
    return (
      <h1 onClick={this.updateMessage} >{this.state.message}</h1>
    );
  }
}
```

Listing 3.3: Beispiel für einen React-*Component* in JSX

Das Codebeispiel 3.3 enthält einen React-*Component*, der in seinem Konstruktor seinen State mit einem String initialisiert, dieser wird in der Render-Funktion in einen Headline-Tag eingebettet. Außerdem wird dem Tag ein Event-Listener zugewiesen, der bei Klick die Methode *updateMessage* des *Components* ausführt. Diese Methode aktualisiert den State des *Components*, was dazu führt, dass sich die Überschrift ändert.
Components können in React auch ineinander geschachtelt werden, um so Hierarchien zu erschaffen. Codebeispiel 3.4 zeigt die Render-Funktion eines *Component*, die den *Component* aus Beispiel 3.3 verwendet, indem ein XML-Tag mit dem Namen dieses *Component* eingefügt wird. Hierbei entsteht eine Hierarchie mit dem *Message-Component* als Kind-*Component*. Dabei können wie in dem Beispiel mit Hilfe von Attributen auch Abhängigkeiten übergeben werden, auf die dann vom Kind-*Component* über *this.props* zugegriffen werden kann.

```
render() {
  return (
    <div>
      <Message example="true" />
    </div>
  );
}
```

Listing 3.4: Beispiel für Hierarchien zwischen *Components* in der *render*-Methode

Aufteilung in Client und Server

Meteor-Applikationen teilen sich einen Client- und in einen Serverteil auf. Weiterhin teilen sich Client und Server auch große Teile des Codes. Wie in Codebeispiel 3.5 gezeigt wird, kann mit *Meteor.isClient* und *Meteor.isServer* festgelegt werden, wo welcher Code laufen soll. Codebereiche, für die kein Bereich festgelegt wurde, werden sowohl auf dem Client, als auch auf dem Server ausgeführt.
Die gesamte Applikation wird, bevor sie ausgeführt werden kann, kompiliert. Bei der Kompilation teilt Meteor den Code entsprechend in die Client- und in die Serverapplikation auf. Außerdem ist es auch möglich durch die Ordnerstruktur festzulegen, wo welcher Code laufen soll. Dafür kann ein /**client**- und ein /**server**-Ordner in die Projektstruktur eingefügt werden.

```
if(Meteor.isClient){
  //client only
}
if(Meteor.isServer){
  //server only
}
//shared code
```

Listing 3.5: Verwendung von Client- und Server-Code

Datenbanken

Neben der MongoDB als Teil der Serverapplikation, gibt es auch auf der Seite des Clients eine Datenbank. Diese Client-Datenbank nennt sich Minimongo und gleicht, bezogen auf ihre Funktionalität, einer MongoDB. Minimongo ist in Javascript geschrieben und kann JSON-Objekte abspeichern. Das primäre Ziel dieses Konzeptes ist es, Datensätze der Server-Datenbank in der Client-Applikation zwischen zu speichern.

Meteor nutzt zum Datenaustausch zwischen Client und Server das **publish-subscribe pattern**. Das *publishing* findet auf dem Server statt, wobei ausgewählte Datensätze zur Verfügung gestellt werden. Zu diesen Datensätzen können Clients *subscriben*, das heißt Datensätze werden angefordert. Falls der Client sich mit den nötigen Rechten ausweisen kann, werden nun alle angeforderten Daten in die Minimongo des Clients eingefügt. Nach diesem Vorgang können die Daten beispielsweise zur Darstellung in einem Template verwendet werden. Wenn sich Datensätze auf dem Server ändern, werden durch Subscriptions betroffene Clients informiert. Dadurch wird die Minimongo Datenbank des Clients stetig auf einem aktuellen Stand gehalten.

Da Meteor als Datenbank MongoDB benutzt, werden Datensätze in *Collections* abgespeichert. Beispielsweise kann eine Collection mit dem Namen **Store** mit dem Befehl `Store = new Mongo.Collection('store');` erschaffen werden. Auf diesem Objekt können dann Operationen zum Suchen, Ändern und Löschen von Datensätzen ausgeführt werden. Auf dem Client sind Collections inital immer leer, auch wenn eigentlich Datensätze auf dem Server liegen. Mit Hilfe der Methoden *publish* auf dem Server und *subscribe* wird die Collection auf dem Client mit Daten befüllt.

Server und Kommunikation mittels DDP

Die Client-Applikation und der Server sind in Meteor-Applikationen eng miteinander verbunden, da sie eine beständige Kommunikation miteinander führen. Zur Kommunikation benutzt das **D**istributed **D**ata **P**rotocol [11], ein auf Basis von Websockets eigens entwi-

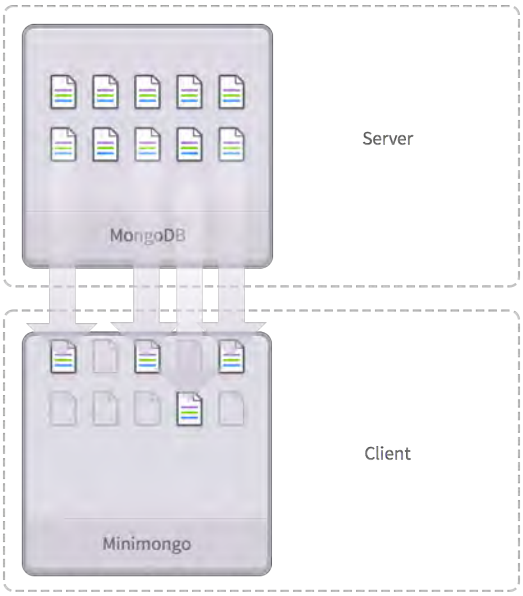

Abbildung 3.2: Das **publish-subscribe pattern** in Meteor. Entnommen aus [10]

ckeltes Protokoll. DDP-Nachrichten sind JSON-Objekte und werden über Websockets zwischen Client und Server ausgetauscht. Dabei sind im DDP bestimmte Nachrichtentypen festgelegt. Ein Beispiel hierfür ist die *ping*-Nachricht, die der Server nutzt, um festzustellen, ob die Verbindung zum Client noch besteht. Der Client würde dann mit einer *pong*-Nachricht antworten, um zu bestätigen.

Das Protokoll soll in erster Linie zwei Aufgaben erfüllen. Zum einen soll es vom Client ausgehende Aufrufe an den Server, sogenannte *remote procedure calls*, ermöglichen. Zum anderen soll es das *publish-subscribe pattern* umzusetzen. Das heißt, wie in Abschnitt 3.1.1 erklärt, dem Client Datensätze zukommen zu lassen und diese aktuell zu halten.

Meteor Methods (Remote Procedure Calls)

Meteor Methods sind Prozeduren, die auf dem Server definiert werden. Eine solche Methode ist ein *key-value*-Paar, wobei der *key* ein eindeutiger Name und die *value* eine

Funktion ist. Diese Prozeduren können vom Client mittels **Meteor.call("method_X");** aufgerufen werden, wobei *method_X* ein Platzhalter für den Namen der Prozedur meint. Sie dienen also als Mittel zur Kommunikation vom Client zum Server zu einem beliebigen Zeitpunkt. Für diese sogenannten *Remote Procedure Calls* wird das DDP benutzt.

Package System

Packages sind modular programmierte Einheiten, die von Nutzern für andere Nutzer veröffentlicht werden. Sie bringen Meteor als Framework mehr Funktionalität und verhindern, dass immer gleiche Problemstellungen von verschiedenen Leuten für jeden Einsatz neu programmiert werden müssen. Bei vielen größeren Packages sind auch mehrere Nutzer am Erstellen und Instandhalten beteiligt, was dann oft zu einer hohen Code-Qualität und Code-Effizienz führt. Meteor Packages lassen sich mit dem Befehl **meteor add package_x** in der Kommandozeile hinzufügen.

3.1.2 Mobile Anwendungen

Da Meteor ein hybrides Framework ist, können nicht nur Webapplikationen, sondern auch mobile Applikationen erstellt werden. Dabei wird mit Hilfe von Cordova (siehe 2.2) für Android oder iOS eine Applikation exportiert. Es ist mit Meteor durchaus möglich, eine Webapplikation und die dazugehörige mobile Applikation in einem gemeinsamen Projekt zu verwalten. Um im Code feststellen zu können, ob es sich bei der aktuellen Zielplattform um einen Browser oder Android/iOS handelt, bietet Meteor neben den schon erwähnten Überprüfungen **Meteor.isClient** und **Meteor.isServer** auch **Meteor.isCordova** an. So kann je nach Plattform unterschiedlicher Code ausgeführt werden. Mit der Cordova-Integration kommen eine Menge an Plugins hinzu, die in Meteor ebenfalls zur Verfügung stehen. In der Kommandozeile können Cordova Plugins zu einem Meteor Projekt hinzugefügt werden. Mit Hilfe des Kommandozeilen-Befehls `meteor add cordova:cordova-plugin-camera@1.2.0` wird ein Cordova Plugin für die Kamera hinzugefügt.

Die Plugins werden dann in der globalen *Scope* der Applikation eingeordnet und sind damit an allen Stellen der Applikation verfügbar. Die genauen Spezifikationen der Cordova-Plugins und deren Unterstützung für die verschiedenen Plattformen ist auf der offiziellen Cordova Website nachzulesen (siehe [12]). In Codebeispiel 3.6 wird nun die Kamera verwendet. Dazu wird zum einen eine Funktion für den Fall, dass ein Foto gemacht wurde und zum anderen eine Funktion, falls ein Fehler dabei auftritt übergeben. Zusätzlich kann ein Objekt mit Optionen übergeben werden.

```
1  Meteor.startup(function() {
2    //plugins now initialized
3    function onSuccess(imagedata){
4      //use imagedata here
5    }
6    function onFail(error){
7      alert("error! +" error);
8    }
9
10   var destination = navigator.camera.DestinationType.DATA_URL;
11   var options = {
12     quality: 60,
13     destinationType: destination
14   }
15   navigator.camera.getPicture(onSuccess, onFail, options);
16 });
```

Listing 3.6: Hinzufügen des Camera-Cordova-Plugins

3.2 Ionic 2

Ionic 2 ist ein hybrides Framework für mobile Applikationen, die auf Basis von HTML5, CSS und Javascript, beziehungsweise Angular mit Hilfe von Cordova erstellt werden.

Ionic erschien 2013 auf Basis von AngularJS, Ionic 2 wurde 2016 veröffentlicht und unterstützt nun Angular (auch als Angular 2 bekannt). Diese neue Version von Ionic verfolgt das gleiche Konzept wie die Vorgängerversionen, hat aber viele neue Features und durch das neue Angular eine bessere Performance. Diese Performance wird unter anderem durch ein neues Konzept von *Change Detection* erreicht, was in diesem Blog-Eintrag von Victor Savkin [13] genauer erklärt wird ist.

Durch die neue Version von Angular hat sich Ionic als Framework grundlegend verändert. Im Vordergrund stehen dabei TypeScript als neue Programmiersprache und der modulare Aufbau von Applikationen durch *Components*. Da TypeScript momentan noch nicht von modernen Browsern genutzt werden kann, wird der TypeScript-Code in valides Javascript kompiliert. Das bedeutet für Ionic-Applikationen, dass sie ebenfalls übersetzt, beziehungsweise kompiliert werden müssen.

3.2.1 Architektur

Das *Graphical User Interface* und die Logik von Ionic-Applikationen basieren auf Angular, weshalb ein Grundverständis von Angular notwendig ist, um die Architektur von Ionic zu verstehen. Im Folgenden wird Angular als Basis des Ionic Frameworks vorgestellt.

ngModules

ngModules sind Module, die zur Strukturierung von Applikationen und als externe Bibliotheken in andere Applikationen eingebunden werden können. Jede Applikation besteht zumindest aus einem Modul, dem *root module*. Dies ist der Einstiegspunkt der Applikation, wo unter anderem das *root component* definiert wird. Unter Ionic selbst liegt ein *ngModule*, das als Einstiegspunkt aller Ionic-Applikationen dient.
Module können andere Module einbinden. Viele Bibliotheken stellen externe Module mit verschiedenen Features zur Verfügung, die leicht eingebunden werden können.

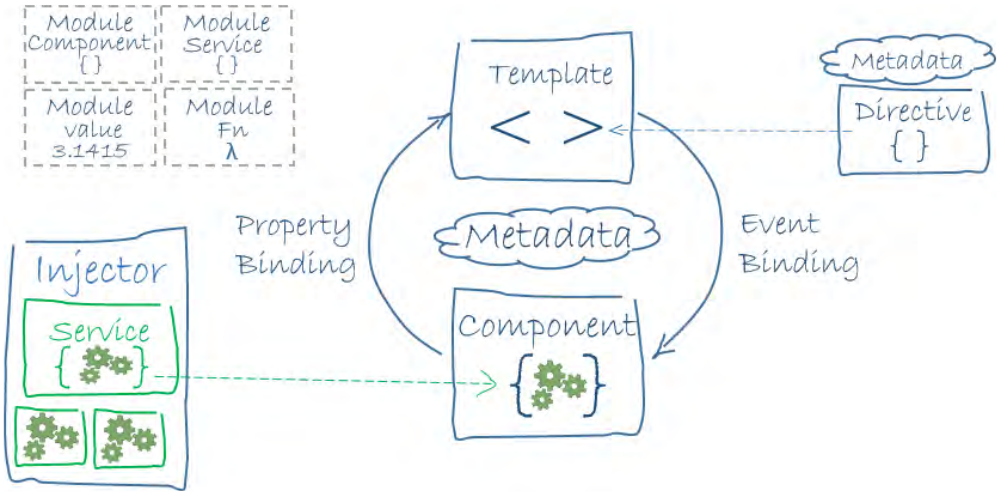

Abbildung 3.3: Architektur einer Angular-Applikation. Entnommen aus [14]

Beispielsweise importiert das *IonicModule* das *HttpModule*, sodass Applikationen HTTP-Aufrufe durchführen können.

Components

Ein *Component* kontrolliert einen Teil des Views, beziehungsweise sein Template. Jede Applikation hat neben einem *root module* ein *root component*, der als Einstiegspunkt für die Applikation dient. Jeder *Component* besteht aus einem *decorator* und einer *Klasse*. Der *decorator* enthält *metadata-properties*, wobei die essentiellen Eigenschaften-Felder das *selector*-Feld und das *template*-Feld sind (weiter Felder sind hier [15] nachzulesen). Damit andere *Components* in ihren Templates diesen *Component* verwenden können, wird im Template ein XML-Tag mit dem Namen des *selectors* eingefügt. Mit dem Aufbau und der Funktionsweise von Templates beschäftigt sich das Unterkapitel 3.2.1.

In Codebeispiel 3.7 ist ein *Component* durch einen *decorator* mittels *@Component* und einer Klasse definiert. Das Importieren zu Beginn wird für das Erstellen von Angular-*Components* benötigt. Die Klasse **Welcome** enthält die Member-Variable **name**. Klassen haben einen Konstruktor und können zusätzliche Member-Variablen und Funktionen

besitzen. Außerdem wird die Klasse exportiert, sodass der *Component* an anderer Stelle importiert werden kann.

```
1 import {Component} from '@angular/core';
2 @Component({
3   selector: 'welcome',
4   template: 'Welcome, {{name}}!'
5 })
6 export class Welcome {
7   name: String = 'Julius';
8 }
```

Listing 3.7: Beispiel eines Angular-/Ionic-*Components*

Ionic bietet eine Auswahl von Standard-*Components* als Teil des Frameworks an. Diese Auswahl besteht aus speziellen *Components*, die in vielen mobilen Applikationen zum Einsatz kommen, wie zum Beispiel Menüs, Suchleisten oder *Toasts* (Pop-up-Benachrichtigungen). In der offiziellen Ionic-Dokumentation [16] befindet sich eine vollständige Liste der Ionic-*Components* an.

Templates & Directives

Templates bestehen in erster Linie aus gewöhnlichem HTML und zusätzlich aus XML-Tags zum Einfügen anderer *Components*. Zusätzlich können sogenannte Directives verwendet werden, wobei es zwei Arten von Directives gibt: *attribute directives* und *structural directives*. Ionic bietet einige *attribute directives* und nennt sie **Utility Attributes**, in Codebeispiel 3.8 wurde durch **text-right** eine solche *attribute directive* verwendet. Weiterhin ist **<ion-content>** kein valides HTML, sondern ein von Ionic bereitgestellter *Component*, der an dieser Stelle eingefügt wird. Zusätzlich zu den von Ionic bereitgestellten *attribute directives*, können auch eigene Directives erstellt werden.

```
<ion-content text-right>
  <ul *ngFor="let student of students">
    <li>{{student.name}}</li>
  </ul>
</ion-content>
```

Listing 3.8: Beispiel-Template in Ionic

Die zweite Art von Directives sind die *structural directives*. Diese sind für die Dynamik des Templates verantwortlich. Dabei gibt es drei von Angular bereitgestellten Directives: *ngIf*, *ngFor* und *ngSwitch*. In Codebeispiel 3.8 wurde eine **ngFor**-Directive verwendet. Damit wird über den Array **students**, eine Membervariable des zugehörigen *Components*, iteriert. Im gleichen Schritt wird für jeden Eintrag, beziehungsweise für jeden Student, der Name des Studenten in einem List-Item-Tag gerendert. Die **ngIf**-Directive kann dazu benutzt werden um Inhalte ein oder aus zu blenden. Außerdem gilt wie bei den *attribute directives* auch hier, dass eigene Directives definiert werden können.

Data- & Event-Binding

Data-Binding in Ionic gibt es in mehreren Ausführungen. In Codebeispiel 3.7 haben wir bereits Data-Binding im Template an der Stelle **{{name}}** verwendet. Man nennt diese Form *1-way binding*. Dabei wird eine Klassenvariable im Template dargestellt und falls sich die Variable ändert, wird auch die Darstellung aktualisiert.
Ein *2-way binding* kann verwendet werden um Teile des Views an Variablen zu koppeln. Das heißt zum *1-way binding* gibt es zusätzlich eine Rückrichtung, die immer dann ausgelöst wird, wenn ein verbundenes UI-Element verändert wird. In Codebeispiel 3.9 wurde ein *2-way binding* mittels `[(ngModel)]="username"` umgesetzt. Wenn nun in das Eingabefeld geschrieben wird, ändert sich die Variable **username** in der Klasse. Da sich aber zusätzlich ein *1-way binding* im **button**-Tag befindet, wird beim Schreiben in das Eingabefeld auch der Inhalt des Buttons geändert.

Neben Data-Bindings gibt es auch Event-Bindings. Diese sind dazu da, bei einem bestimmten Ereignis eine Funktion auszuführen. In Ionic 2, beziehungsweise Angular, wird dies umgesetzt, indem UI-Elementen ein spezielles Attribut zugewiesen wird. Im Beispiel 3.9 geschieht das an der Stelle `(click)="buttonPressed()"`. Eine solche Bindung von Funktionen an Events besteht immer aus einer Klammerung und einem Event-Typ. In diesem Fall geht es um ein Klick-Event. Als Wert des Attributes wird eine Funktion eingetragen, die Teil der Klasse ist und ausgeführt werden soll. In diesem Beispiel wird die Methode `buttonPressed()` ausgeführt.

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'page-about',
5    template: `
6      <h4>Enter Username:</h4>
7      <ion-input [(ngModel)]="username" ></ion-input>
8      <button (click)="buttonPressed()">
9        Is your name {{username}}?
10     </button>
11   `
12 })
13
14 export class ExampleBindings {
15   username:string="";
16   buttonPressed(){
17     alert("A user just verified his name.");
18   }
19 }
```

Listing 3.9: Beispiele für Data- und Event-Bindings

Services

Services sind ebenfalls Teil von Angular und somit auch in Ionic verfügbar. Ähnlich wie *Components* sind auch Services modulare Einheiten. Ihr Unterschied zu *Components* ist, dass sie losgelöst vom *View* sind. Zum Beispiel werden Services benutzt um komplexe Berechnungen durchzuführen oder wie in Codebeispiel 3.10 HTTP-Requests zu verwalten.

```
1  import {Injectable} from "@angular/core";
2  import {Http} from "@angular/http";
3  import 'rxjs/add/operator/toPromise';
4
5  @Injectable()
6  export class ExampleService {
7    constructor(http: Http) {
8      this.http = http;
9      this.data = null;
10   }
11
12   makeCall() {
13     this.http.get('./example/route')
14     .toPromise()
15     .then(response =>
16       this.data = response.json().data
17     );
18   }
19
20   getData() {
21     return this.data;
22   }
23 }
```

Listing 3.10: Beispiele für einen HTTP-Service

In Codebeispiel 3.10 wurde ein Service implementiert. Services werden mit **@Injectable()** eingeleitet und bestehen ansonsten aus einer Klasse, in der die eigentliche Aufgabe des Services implementiert wird. In unserem Beispiel besteht der Service aus einer Methode zum HTTP-Request absenden und einer Methode zum Abfragen der empfangenen Daten.

Pagestack

Ionic Applikationen arbeiten gewöhnlich mit Pagestack. Eine Page ist ein *Component*, der die komplette Bildschirmoberfläche einnimmt. Zum Verwalten der Pages stellt Ionic einen **NavController** bereit.
Für den Start der Applikation muss eine **root page** festgelegt werden, diese nimmt den untersten Platz des Stapels ein. Die oberste Page auf dem Stapel ist auch die Page, die gerade angezeigt wird. Um eine Page auf den Stapel zu legen, wird die **push**-Methode des *NavControllers* verwendet. Diese Page wird dann automatisch angezeigt. Um wieder zur vorherigen Page zurückzukehren wird die **pop**-Methode verwendet.

Ionic bietet im Kommandozeilen-Interface einen Befehl zum Generieren von Pages an. Mit dem Befehl `ionic -g page examplePage` werden in der Ordnerstruktur des Ionic-Projekts entsprechende Dateien für die Page angelegt.

Storage

Zum dauerhaften Abspeichern von Daten verwendet Ionic ein eigenes *Storage*-Module. Dieses Modul kommuniziert mit einer SQLite-Datenbank, die als Cordova-Plugin hinzugefügt werden muss. Mit dem Kommandozeilen-Befehl `cordova plugin add cordova-sqlite-storage --save` wird das Plugin hinzugefügt.

Der *Storage* kann wie in Codebeispiel 3.11 importiert werden und dann für eine Klasse im Konstruktor verfügbar gemacht werden. Der *Storage* kann *Key-Value*-Paare abspeichern. Zum Einfügen oder Aktualiseren eines Paares wird auf dem *Storage*-Objekt die Methode `set(key,value)` benutzt. Zum Auslesen eines Datensatzes wird die Methode `get(key)` verwendet. Da die Zugriffe auf den *Storage* asynchron ablaufen, gibt die Methode zum Auslesen ein sogenanntes *Promise* zurück, auf das die Methode `then()` wartet. In der **then()**-Methode ist der Datensatz dann verfügbar und kann verwendet werden.

```
1  import { Storage } from '@ionic/storage';
2  export class MyApp {
3    constructor(storage: Storage) {
4       // set a key/value
5       storage.set('name', 'Max');
6       // Or to get a key/value pair
7       storage.get('name').then((val) => {
8         console.log('Your name is', val);
9       })
10   }
11 }
```

Listing 3.11: Beispiel für die Verwendung des *Storage* in Ionic. Entnommen aus [17]

3.2.2 Command Line Interface

Ionic 2 lässt sich über ein CLI (**C**ommand **L**ine **I**nterface) steuern. Im Folgenden werden die wichtigsten Befehle vorgestellt.

Eine neue Applikation kann mittels `ionic start exampleApp --v2` erstellt werden. Es wird ein neuer Projektordner mit entsprechnder Ionic-Ordnerstruktur erstellt. Ionic 2 enthält Generierungs-Befehle mit denen die Grundgerüste von Pages und Providern für Services hinzugefügt werden. Um zum Beispiel Pages hinzuzufügen gibt

es den Befehl `ionic g page PAGE-NAME`. Mit Hilfe von `ionic serve` wird die Applikation standardmäßig lokal auf Port 8100 gestartet und im Browser geöffnet. Der Entwicklungsprozess, beziehungsweise das Austesten von Funktionen kann im Browser stattfinden, jedoch muss beim Testen von Cordova-Plugins ein mobiles Gerät oder ein Emulator verwendet werden.
Cordova-Plugins können mit dem Befehl `cordova plugin add PLUGIN-NAME` hinzugefügt werden. Mit dem Befehl `ionic emulate` kann ein Android- oder iOS-Emulator, auf dem die Ionic-Applikation läuft, gestartet werden. Es ist aber auch möglich, ein eigenes mobiles Gerät mit dem Computer zu verbinden und dann mittels `ionic run` auf dem Gerät zu testen.
Um letztendlich eine kompilierte *.apk*- oder *.ipa*-Version der Applikation zur Installation auf Endgeräten zu erschaffen, verwendet man `ionic build PLATFORM-NAME`. Diese können dann auch zum Upload in verschiedene App-Stores verwendet werden.

4

Anforderungen

Vor der Umsetzung einer Applikation, müssen die Anforderungen festgelegt werden. Anforderungen kann man in zwei Kategorien unterteilen. Funktionale Anforderungen betreffen Leistungen, die die Anwendung erbringen soll. Nicht-funktionale Anforderungen beschreiben, wie gut die Leistung erbracht werden soll.

4.1 Funktionsweise der Applikation

Bevor die genauen Anforderungen an die Applikation definiert werden, soll im Folgenden der Kontext in dem die Applikation steht und die Funktionsweise der Applikation erklärt werden. Nutzer dieser Applikation sind Patienten, die in psychologischer Betreuung eines Therapeuten sind. Die Applikation ist zum einen Teil Kommunikationsmittel zwischen Patient und Therapeut und zum anderen Teil ein aktives Hilfsmittel der Behandlung. Das heißt ein Teil der Behandlung spielt sich in der Applikation selbst ab. Um den Patienten zu behandeln, werden vom Therapeuten über die Applikation Aufgaben gestellt, die der Patient in der Applikation selbst oder in seiner Umgebung mit Hilfe der Applikation absolvieren soll.

Wie eine solche Aufgabe, beziehungsweise Hausaufgabe, abläuft, ist in Schaubild 4.1 zu sehen. Der Therapeut kann dem Patienten Hausaufgaben stellen. Dabei gibt es zwei mögliche Abläufe. Der Patient sieht in der Applikation eine Liste von Hausaufgaben, die er zu einer beliebigen Zeit abarbeiten kann. Es gibt aber auch Aufgaben, die durch bestimmte Kontexte ausgelöst werden, um direkt erledigt zu werden. Ein Kontext kann

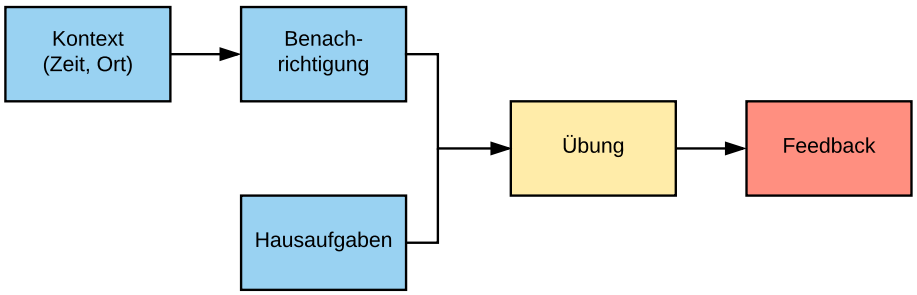

Abbildung 4.1: Der konzeptionelle Ablauf einer Patienten-Hausaufgabe

entweder zeitlich oder ortsgebunden sein, also beispielsweise eine Uhrzeit oder ein Aufenthaltsort. Wenn ein solcher Kontext ausgelöst wird, bekommt der Patient eine Benachrichtigung. Danach folgt die Übung selbst, die dann in der Applikation oder mit Unterstützung durch selbige stattfindet. Nach jeder Übung folgt ein Feedback, in welchem der Patient Rückmeldung an den Therapeuten verfassen kann. Das Feedback besteht aus einer Bewertung der Übung anhand einer Skala, sowie einem optionalen Textfeld für komplexere Rückmeldung.

4.2 Funktionale Anforderungen

Abkürzung	Anforderung
FA01	Fragebögen mittels Formularen
FA02	Darstellung von Multimedia-Elementen
FA03	Benachrichtigungen erstellen
FA04	Auf Benachrichtigungen reagieren
FA05	Mittels Kontext einen Vorgang auslösen
FA06	Einen Anruf starten

Tabelle 4.1: Tabelle der funktionalen Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen an die Applikation definiert.

FA01 Fragebögen mittels Formularen

Die Applikation soll Fragebögen anzeigen können, die der Patient ausfüllen und abschicken kann.

FA02 Darstellung von Multimedia-Elementen

Die Applikation soll PDFs, Bilder und Texte, sowie formatiertes HTML anzeigen können. Weiterhin soll sie Audio-Dateien abspielen und Videos darstellen können.

FA03 Benachrichtigungen erstellen

Die Applikation soll Benachrichtigungen an das Betriebssystem senden können, um den Patienten beispielsweise an Hausaufgaben zu erinnern oder eine Übungsaufgabe einzuleiten.

FA04 Auf Benachrichtigungen reagieren

Wenn der Patient eine Benachrichtigung erhält, soll die Applikation bei Aktivierung der Benachrichtigung entsprechend reagieren. Beispielsweise soll bei einer Benachrichtigung, die eine Erinnerung an einen Fragebogen beinhaltet, der entsprechende Fragebogen angezeigt werden.

FA05 Mittels Kontext einen Vorgang auslösen

Die Applikation soll mittels verschiedener Kontexte einen Vorgang ausführen. Kontext meint hier externe Gegebenheiten, zum Beispiel ein Datum, eine Uhrzeit oder einen Aufenthaltsort (GPS). Ein Vorgang meint hierbei eine Benachrichtigung.

FA06 Einen Anruf starten

Damit der Patient im Notfall seinen Therapeuten anrufen kann, soll die Applikation eine eingespeicherte Rufnummer per Knopfdruck anrufen können.

Abkürzung	Anforderung	Priorität
NFA01	Kompatibilität	9
NFA02	Zuverlässigkeit	8
NFA03	Wartbarkeit	9
NFA04	Benutzbarkeit	7
NFA05	Design ("Look & Feel")	5

Tabelle 4.2: Tabelle der funktionalen Anforderungen mit Priorität auf einer Skala von 1 (unwichtig) bis 10 (unerlässlich)

4.3 Nicht-Funktionale Anforderungen

Neben Anforderungen funktionaler Art, gibt es auch gewisse Qualitätseigenschaften und Rahmenbedingungen, die die Applikation erfüllen soll. Diese nicht-funktionalen Anforderungen werden in der folgenden Auflistung definiert.

NFA01 Kompatibilität
Die Applikation soll auf möglichst vielen Ziel-Plattformen laufen.

NFA02 Zuverlässigkeit
Die Applikation soll keine fehlerhaften Zustände annehmen oder sogar abstürzen.

NFA03 Wartbarkeit
Es soll nachträglich möglich sein, die Applikation zu verändern und um Funktionen zu erweitern.

NFA04 Benutzbarkeit
Der Benutzer soll durch ein verständliches User-Interface schnell die Funktionen der Applikation kennenlernen und mit diesen umgehen können.

NFA05 Design ("Look & Feel")
Durch ansprechende Gestaltung der Applikation, soll sich der Benutzer im Umgang mit der Applikation wohl fühlen.

5

Realisierung

Auf Basis der in Kapitel 3 vorgestellten Frameworks und den in Kapitel 4 definierten Anforderungen soll nun je eine Applikation pro Framework entwickelt werden. Zur strukturierten Realisierung der Applikationen, wurden zunächst Mock-ups der einzelnen Screens aufbauend auf die Anforderungen erstellt. Dabei sollen sich der Aufbau der Graphical User Interfaces der Applikationen möglichst gleichen, um gute Bedingungen für eine fundierte Evaluation zu schaffen.

5.1 Konzeption & Mock-ups

Mock-ups sind Modelle für das Aussehen und auch für die Funktionalität der Applikation. Sie können verwendet werden, um die Anforderungen an das User Interface festzulegen und die Anordnung der Bedienelemente zu definieren. In diesem Kapitel werden die einzelnen Screens und die Funktionalität der Applikation anhand von Mock-ups vorgestellt. Die Applikationen sollen sich in der Anordnung der Bedienelemente gleichen, sollen aber nicht zwangsweise das gleiche Design haben. Die folgenden Mock-ups wurden mit der Software *Balsamiq 3.5.7* erstellt.

Startscreen

Nach dem Start der Applikation befindet sich der Patient auf dem Startscreen (siehe Abbildung 5.1). Unter dem Schriftzug oder dem Logo der Applikation befinden sich drei Buttons in der Mitte des Screens. Mit diesen Buttons gelangt der Patient zu seinen noch

nicht erledigten Hausaufgaben und zu den Einstellungen. In dem Button "Hausaufgabenïst auch zu sehen, wie viele Hausaufgaben noch zu erledigen sind. Ein weiterer Button soll einen Anruf an den Therapeuten starten.

Hausaufgaben

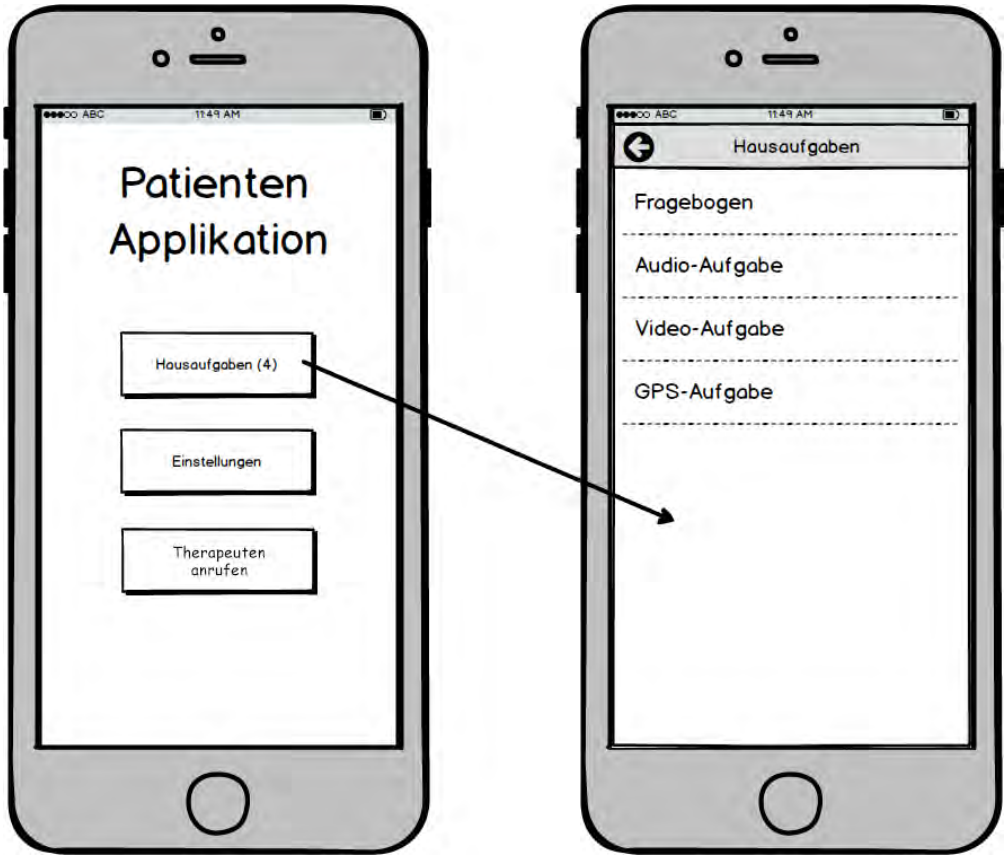

Abbildung 5.1: Mock-up des Hausaufgaben-Screens

In Abbildung 5.1 ist außerdem dargestellt, wie man vom Startscreen über den Button zu den noch offenen Hausaufgaben gelangt. Der Patient sieht dort eine Liste der offenen Hausaufgaben. Über der Liste befindet sich eine Navigation, die auch in anderen Screens der Applikation Verwendung findet. Die Navigation besteht aus einem Titel und einem Button, über den man zum vorherigen Screen gelangt.

Fragebögen

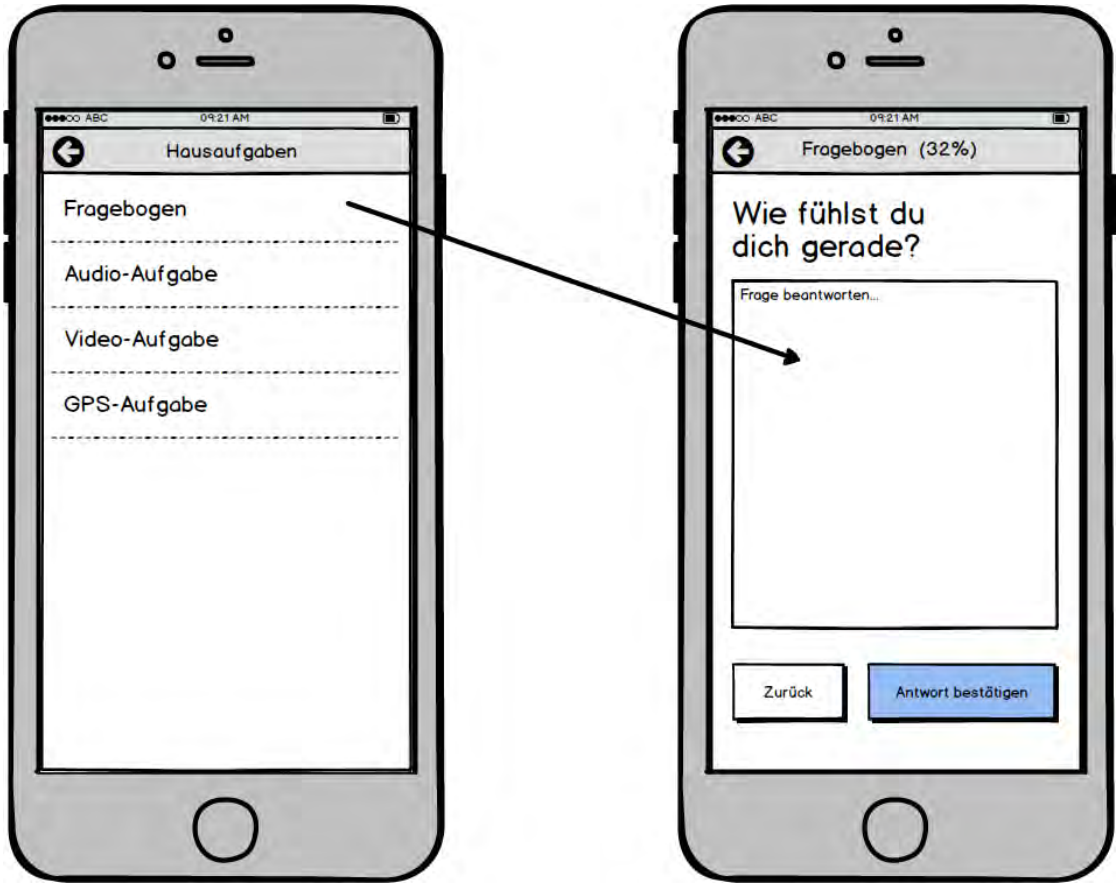

Abbildung 5.2: Mockup des Fragebogen-Screens

Eine Art von Hausaufgaben ist das Ausfüllen eines Fragebogens. Wie in Abbildung 5.2 dargestellt ist, kann der Benutzer durch Anwählen des entsprechenden Listenelements zu dem Screen des Fragebogens gelangen. Der Fragebogen selbst besteht aus mehreren Fragen, wobei die aktuelle Frage den gesamten Screen einnimmt.
Die jeweilige Frage steht unter der Navigation und kann in einer Text-Box beantwortet werden. Mit dem Button Äntwort bestätigen"kann zur nächsten Frage fortgefahren werden oder mit dem SZurückButton zur vorherigen Frage gewechselt werden. In der Navigation wird zusätzlich der Fortschritt der beantworteten Fragen als Prozentzahl angezeigt. Sind alle Fragen beantwortet soll ein Pop-up erscheinen, dass dem Patienten

mitteilt, dass der Fragebogen fertig beantwortet ist. Danach kehrt der Benutzer zurück zum Hausaufgaben-Screen.

Medien-Aufgaben

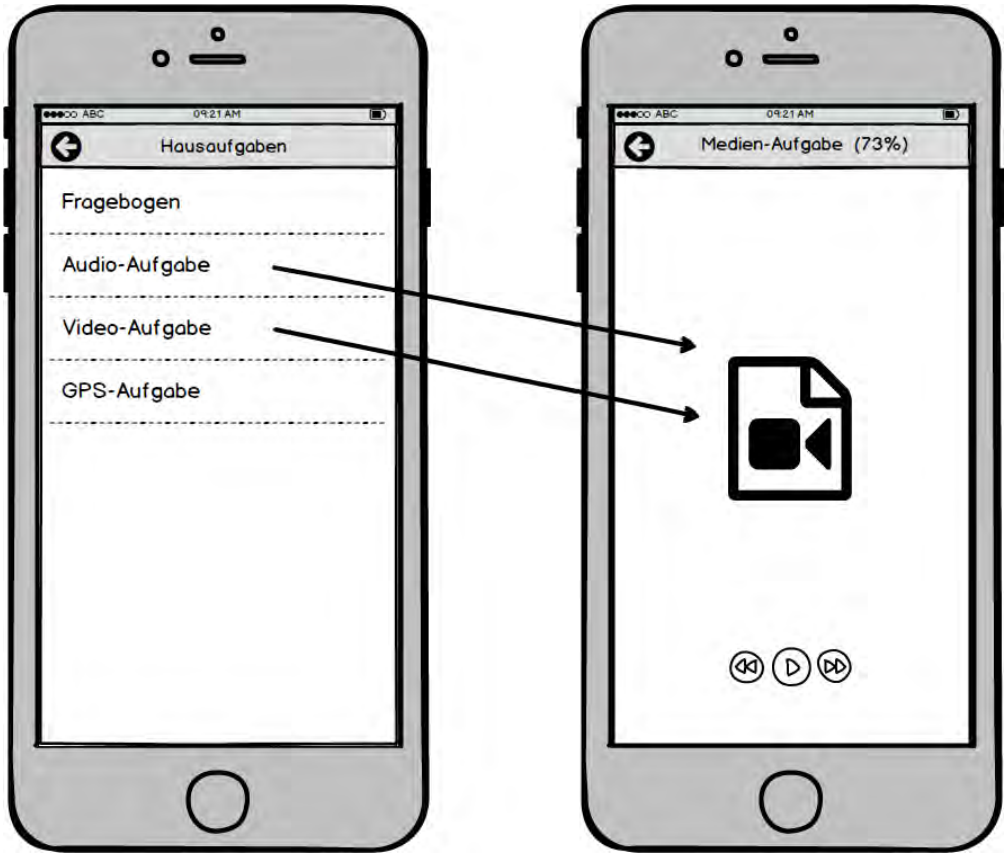

Abbildung 5.3: Mockup der Medien-Screens

Eine weitere Art von Hausaufgabe ist der Konsum von diversen Medien. Möglich sind Audio-, Video-, Bild-, PDF- und HTML-Dateien. Das Mock-up für den Screen der Medien-Aufgaben ist in Abbildung 5.3 dargestellt. In der Navigation soll die Medien-Art stehen und wie auch bei dem Fragebogen-Screen eine Prozentanzeige des aktuellen Fortschritts. Den Rest des Screens nimmt das Medium selbst ein. Zusätzlich sollen falls notwendig Bedienelemente zur Verfügung gestellt werden. Beispielsweise soll bei Video- oder Audio-Dateien eine Steuerung zum Pausieren oder zur Regelung der Lautstärke erscheinen.

GPS-Aufgaben

Abbildung 5.4: Mockup des GPS-Screens

Neben den bisher vorgestellten Übungen, beziehungsweise Hausaufgaben, die am mobilen Gerät selbst erledigt werden, gibt es GPS-Aufgaben. Bei dieser Art von Aufgaben, geht es darum, dass der Patient einen bestimmten Ort aufsucht. Dazu wird der Screen aus Abbildung 5.4 genutzt. Die Navigation enthält eine Anzeige, die die noch zu absolvierende Distanz angibt. Der restliche Screen soll von einer Karte gefüllt sein, die den Benutzer zum Ziel leiten soll. Die Navigation zum Zielort und die Interaktivität der Karte soll von einer eingebetteten externen Software übernommen werden.

Feedback

Nach jeder abgeschlossenen Übung wird der Patient, wie in Abbildung 5.5 zu sehen ist, um ein Feedback gebeten. Hierbei kann der Patient durch die Vergabe von einem bis zu maximal fünf Sternen die vorangegange Übung bewerten. Zusätzlich gibt es ein Textfeld um komplexere Rückmeldung geben zu können. Das können zum Beispiel Probleme sein, die bei der Übung aufgetreten sind. Nach dem Feedback ist die Hausaufgabe beendet.

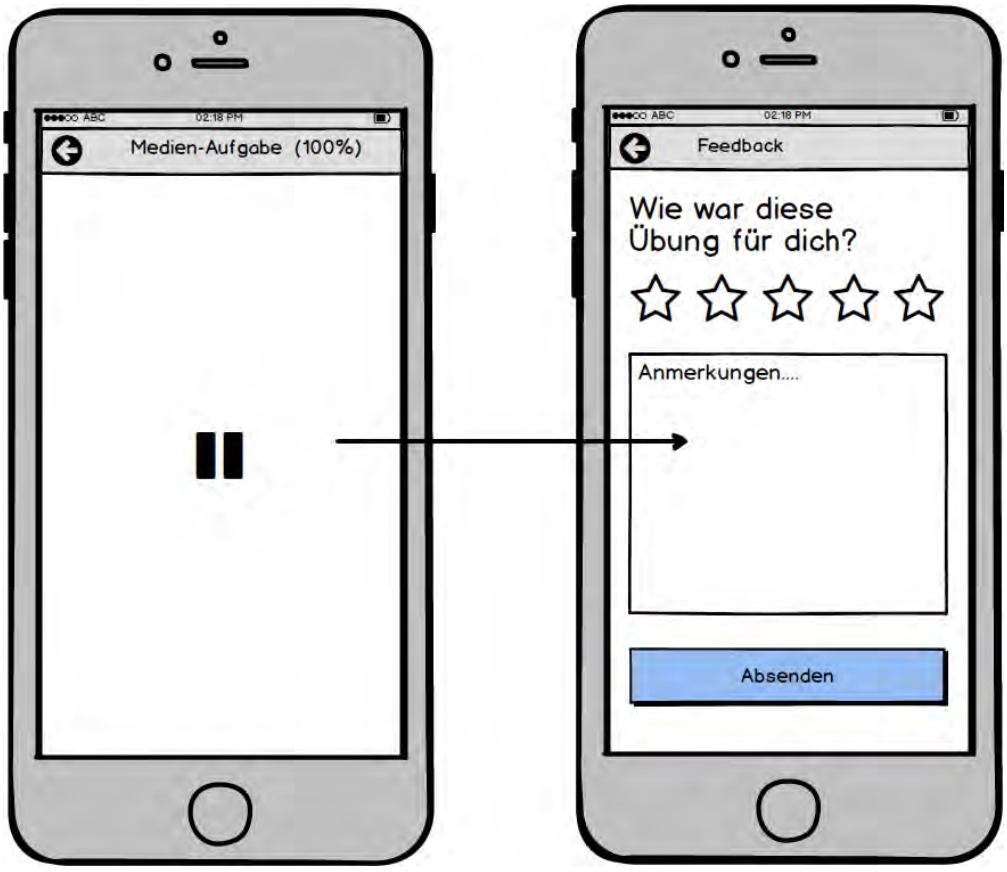

Abbildung 5.5: Mockup des Feedback-Screens

Einstellungen

Übungen, die durch Kontexte ausgelöst werden sollen, brauchen teilweise Referenzen. Zum Beispiel kann ein Kontext für eine Übung das morgendlichen Aufstehen sein. Da Menschen zu unterschiedlichen Zeiten aufstehen, ist es hierbei sinnvoll, den Patienten dies festlegen zu lassen. In Abbildung 5.6 ist das Mock-up des Einstellungs-Screen zu sehen. Dieser ist vom Startscreen aus erreichbar und lässt den Patienten verschiedene Kontexte festlegen. Denkbar wäre auch eine Möglichkeit zum Festlegen von Orten, wie beispielsweise der Arbeitsplatz.

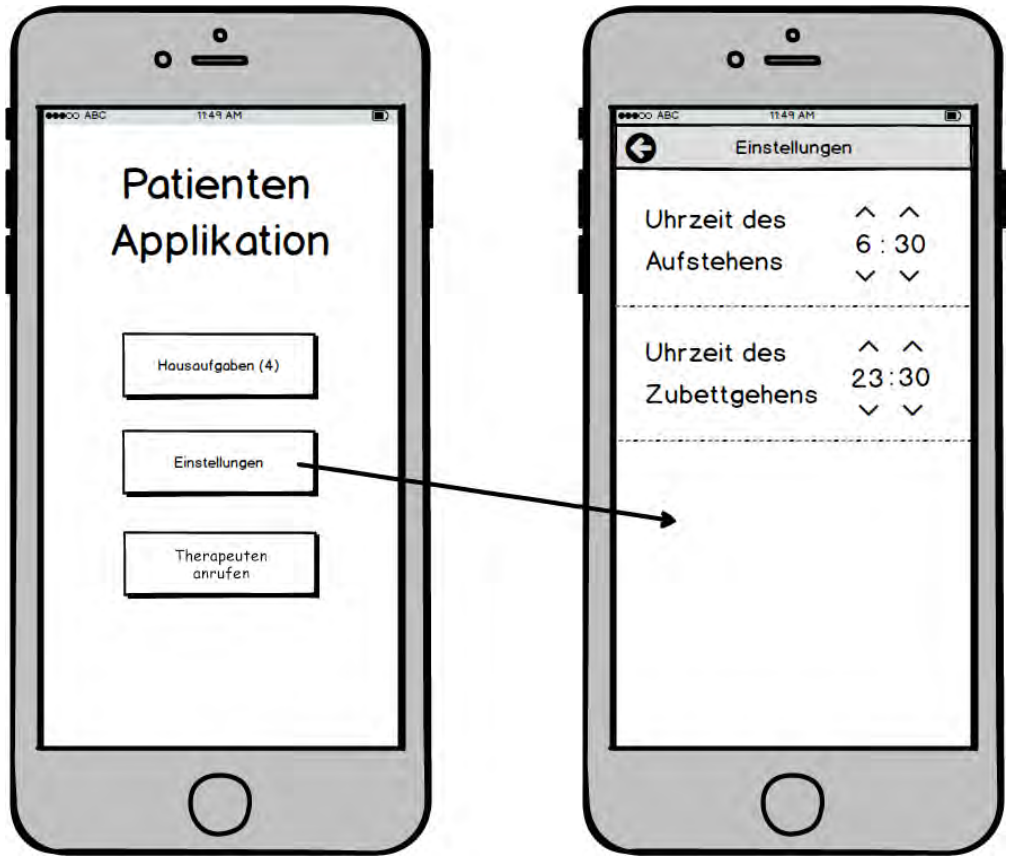

Abbildung 5.6: Der konzeptionelle Ablauf einer Patienten-Hausaufgabe

5.2 Meteor

Anhand der Mock-ups und Anforderungen wurden nun die Applikationen erstellt. Dieser Abschnitt dokumentiert die Realisierung der Meteor-Applikation dar und erklärt deren Aufbau und Funktionsweise. Diese Applikation verwendet das Meteor Framework in der Version 1.4.3.1.

5.2.1 Architektur

Das Grundgerüst der Applikation besteht aus dem Router, den Controllern und Templates. Die Bestandteile und ihre Relation zueinander ist in Schaubild 5.7 zu sehen. Meteor gibt dem Entwickler die Option, sich bezüglich der *UI-Engine* zwischen Angular, React und Blaze zu entscheiden. Die Wahl für diese Applikation fiel auf Blaze, da die Komplexität und Größe der Applikation verhältnismäßig niedrig sind. Warum sich Blaze gut für diese Applikation eignet, wird in Kapitel 6.3 diskutiert.

Das Routing, beziehungsweise die Navigation, wird mit Hilfe des Meteor-Package **Iron Router** [18] umgesetzt. Dieses Package ermöglicht Routing in der Client-Applikation. Das bedeutet, dass das Meteor-Package eigenständig den Pfad der URL manipulieren und entsprechende Inhalte dazu rendern kann. In dieser Applikation wurden in der Datei *Router-Definitions* verschiedene Definitionen für Pfade festgelegt, auf die der Router reagieren soll. Je nach Pfad werden auf diese Weise verschiedene Templates gerendert. Um sich wiederholende Elemente, zum Beispiel Navigations-Elemente, nicht mehrfach zu implementieren, kann ein Layout-Template festgelegt werden, welches für mehrere Pfade benutzt wird. In der Applikation besteht dieses aus einer Navigationsleiste, in der einen Zurück-Knopf und den Titel der aktuellen Seite darstellt. Der Einstiegspunkt der Applikation ist die URL ohne Pfad oder auch ¨ \ ¨, in den *Router-Definitions* ist zu diesem Pfad definiert, dass das Template **Main** gerendert wird. Die Navigation durch die Applikation erfolgt über HTML Link-Tags, wie in Codebeispiel 5.1 zu sehen ist, oder Javascript-Funktionen, die auf dem globalen Routerobjekt definiert sind. Die Syntax der Blaze-Templates wurde bereits In Kapitel 3.1.1 erklärt. In Code-Auszug 5.2 ist das

Template für die Liste noch offener Hausaufgaben zu sehen. Das Template besteht aus einer ungeordneten Liste, in die dynamisch Listen-Elemente eingefügt werden. Für das Template wurde der Helper **hausaufgabenArray** definiert, wie in Code-Auszug 5.3 gezeigt wird. Falls der Array leer ist, was mittels des *{{#unless}}*-Blocks überprüft wird, wird angezeigt, dass aktuell keine Hausaufgaben offen sind. Mit Hilfe des *{{#each}}*-Blocks wird für jedes Array-Element ein Listen-Element eingefügt. Der Array enthält alle Dokumente der *HausaufgabenCollection*, einer *Mini-MongoDB-Collection*, die vom *Storage-Controller* verwaltet wird.

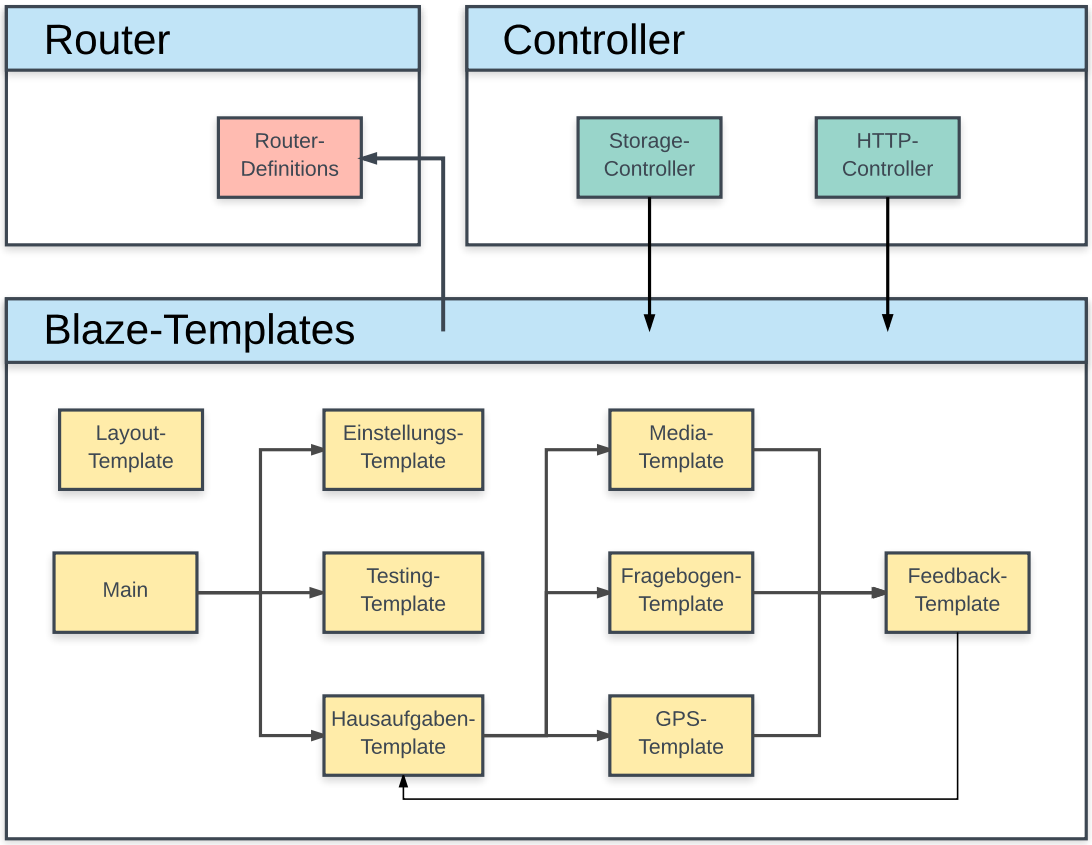

Abbildung 5.7: Architektur-Schaubild der Meteor-Applikation

```
<a href="/hausaufgaben">
  <button type="button">
    Hausaufgaben
  </button>
</a>
```

Listing 5.1: Navigation von der Startseite zur Liste der offenen Hausaufgaben mittels HTML-Tag (Meteor-Applikation)

```
<template name="hausaufgaben">
  <ul class="list-group">
    {{#unless hausaufgabenArray}}
      <div class="container mtop">
        <h4>Es stehen keine Hausaufgaben zur Verfuegung.</h4>
      </div>
    {{/unless}}
    {{#each hausaufgabenArray}}
      <a class="link" href="/hausaufgabe/{{id}}">
        <li class="list-group-item hausaufgabe">
          {{title}}
        </li>
      </a>
    {{/each}}
  </ul>
</template>
```

Listing 5.2: Template der Liste noch offenen Hausaufgaben

Neben dem *Storage-Controller*, der für die lokale Verwaltung und Bereitstellung von Daten verantwortlich ist, gibt es den *HTTP-Controller*, welcher die Kommunikation zu einem Server und zur Rest-API ermöglicht. Controller sind Javascript-Klassen, die bestimmte Aufgaben übernehmen, die logisch unabhängig vom aktuellen Zustand der Applikation

sind. Diese Auslagerung von Funktionen vermeidet zum einen sich wiederholenden Programmier-Code und schafft zum anderen bessere Übersicht. Die Controller-Klassen werden hauptsächlich in Templates genutzt, wo sie dann zum Beispiel zum Abspeichern von Daten verwendet werden.

```
1 Template.hausaufgaben.helpers({
2   hausaufgabenArray(){
3     return HausaufgabenCollection.find().fetch();
4   }
5 });
```

Listing 5.3: Definition des Helpers für das Hausaufgaben-Template

5.2.2 User Interface

Das User Interface der Applikation hat sich im Aufbau an den Mock-ups aus Kapitel 5.1 orientiert. Für das CSS-Design des User Interfaces wurde das Framework Bootstrap 4 verwendet, was sich zum Zeitpunkt der Implementierung noch in einer Alpha-Phase befand. Bootstrap 4 bietet vorgefertigte Designs für Navigationsleisten, Buttons und weitere Bedienelemente.

Beim Design wurde besonders darauf geachtet, die nicht-funktionalen Anforderungen, die in Kapitel 4.3 festgelegt wurden, zu erfüllen. Für eine gute Benutzbarkeit (NFA04) und gute Bedienbarkeit wurde darauf geachtet, große Schaltflächen und nur wenige Informationen pro Dialog, beziehungsweise Seite zu verwenden. Weiterhin liegt ein Schwerpunkt beim Design des User Interfaces auf Schlichtheit und Klarheit. Daraus soll ein möglichst gutes *Look & Feel* (NFA05) und eine Fokussierung auf das Wesentliche für den Benutzer resultieren.

Im Folgenden werden einige Screenshots der Meteor-Applikation gezeigt, um einen Einblick in das implementierte Design zu geben. In Abbildung 5.8 ist auf der linken Seite

der Startscreen der Applikation zu sehen. Im Vergleich zum Mock-up des Startscreens aus Kapitel 5.1 wurde die Fläche des Bildschirms komplett ausgenutzt, die Schaltflächen sind breiter gestaltet und sind damit einfacher anwählbar. Außerdem wurde die Schaltfläche *"Therapeut anrufen"* grün eingefärbt, um sich von den andern Bedienelementen abzuheben. Zuletzt wurde ein schlichter Hintergrund hinzugefügt für ein besseres *Look & Feel*. Auf der Rechten Seite ist der Einstellungs-Screen zu sehen. Dieser Screen ist seinem Mock-up aus Kapitel 5.1 sehr ähnlich. Die Kontext-Elemente sind hier als Schaltflächen implementiert worden, um die Möglichkeit des Anwählens zu suggerieren. Das Ändern der Uhrzeit findet dann in einem Pop-up-Dialog statt.

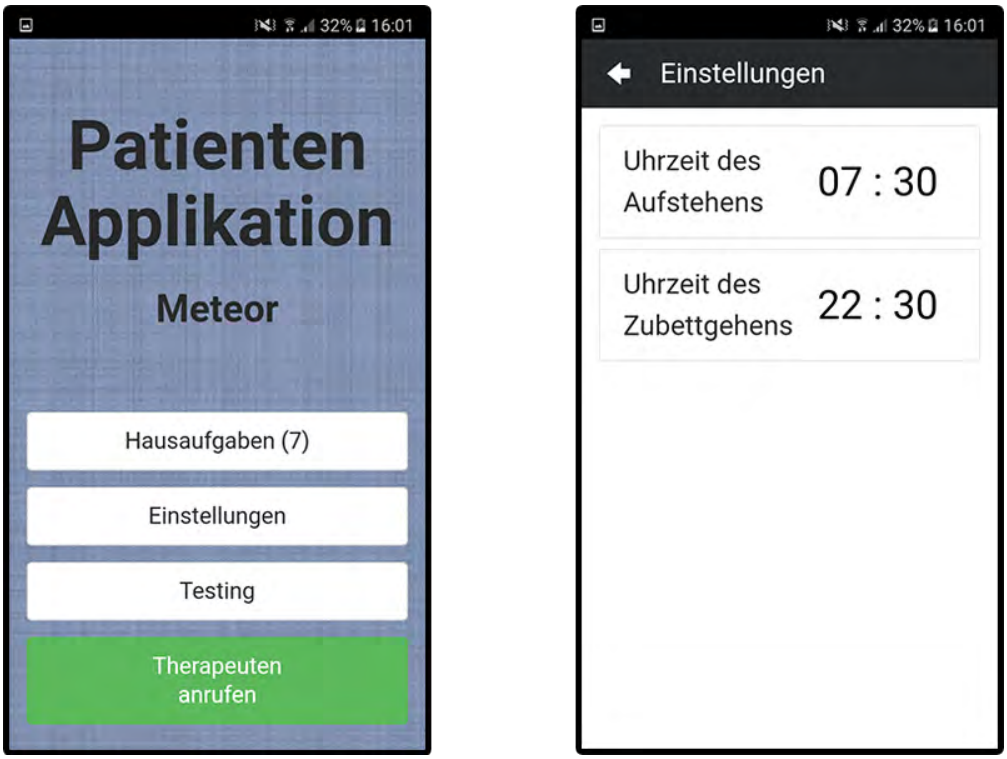

Abbildung 5.8: Implementierung des Startscreens und der Einstellungs-Seite für Kontexte in der Meteor-Applikation

Zwei weitere Screenshots der Meteor-Applikation sind in Abbildung 5.9 zu sehen. Auf der linken Seite der Abbildung ist die Liste offener Hausaufgaben dargestellt. Diese Seite ist dem Mock-up aus Kapitel 5.1 sehr ähnlich. Die Listen-Elemente enthalten weiterhin

nur den Titel und sind in einer schlichten Weise untereinander angeordnet. Auf der rechten Seite der Abbildung ist eine GPS-Übung zu sehen. Die Karte wurde mit Hilfe der Javascript-API von Google Maps implementiert.

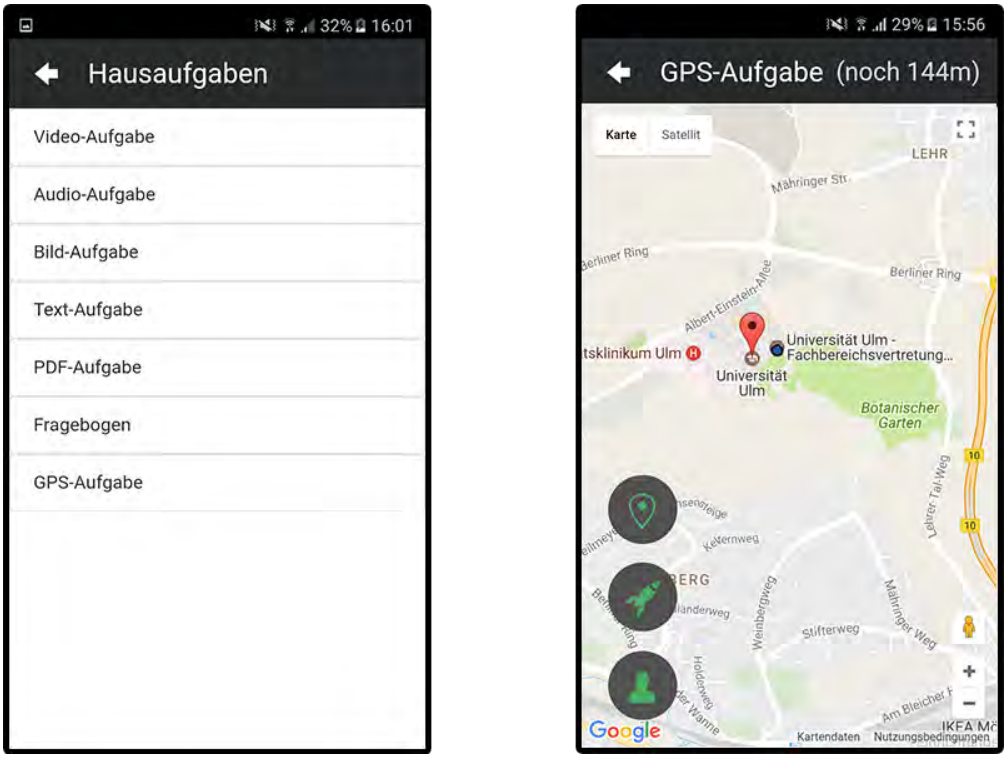

Abbildung 5.9: Implementierung der Liste offener Hausaufgaben und der GPS-Übungen (Karte ist Teil von Google Maps [19] und Eigentum von Google) in der Meteor-Applikation

Auf der Karte selbst ist zum einen der Standort des Übungsziels in Form eines roter Markers und zum anderen der Standort des Patienten in Form eines blauen Punktes. Über der Karte schwebend wurden unten links drei *Floating-Action-Buttons* angeordnet. Diese sind zur Navigation auf der Karte für den Benutzer geeignet. Der obere Button zentriert die Karte auf der Zielposition, der untere Button zentriert die Karte auf der Positon des Patienten. Da bei Präsentation der Applikation effektiv keine Strecke zurückgelegt werden kann, wurde der mittlere Button hinzugefügt, um die Zielposition auf die Position des Patienten zu setzen.

5.2.3 Verzicht auf Meteor-Server

In Meteor werden Client und Server im gleichen Projekt implementiert. Da in diesem Anwendungsfall bereits ein Server und eine REST-API existieren, entfällt die Nützlichkeit eines Meteor-Servers. Es existiert zwar die Möglichkeit den Meteor-Server zu implementieren und ihn zwischen die Client-Applikation und die REST-API zu stellen. Dies würde dazu führen, dass die Kommunikation mehr Zeit in Anspruch nimmt und das System fehleranfälliger machen. Wenn beispielsweise der Meteor-Server ausfallen sollte, ist keine Kommunikation zwischen mobiler Applikation und der REST-API möglich, obwohl beide Kommunikationspartner dazu fähig werden. Aufgrund der angeführten Argumentation, wurde die Applikation ohne entsprechenden Server implementiert. Dieses Vorgehen sorgt zwar für effizientere Kommunikation, bringt aber gleichzeit neue Herausforderungen mit sich.
Das Konzept für die Datenverwaltung in Meteor mittels *Collections* wurde bereits in Kapitel 3.1.1 erklärt. Eine *Collection* existiert gewöhnlich auf dem Meteor-Server und die verschiedenen Clients bekommen einen Teil davon zugesendet. Es gibt eine Möglichkeit, *Collections* ohne Verbindung zum Server zu definieren. Dies kann mit Hilfe des Befehls `HausaufgabenCollection = new Mongo.Collection(null);` erreicht werden. Diese *Collection* bietet volle Funktionalität bezüglich der MongoDB-Befehle, die darauf ausgeführt werden können. Wird die Applikation jedoch geschlossen, ist nicht sichergestellt, dass die Daten bei erneutem öffnen noch vorhanden sind.

Collections auf Seite des Clients sind nicht dazu konzipiert, Daten langfristig abzuspeichern. Eine Möglichkeit wäre, auf Collections zu verzichten und den *Local Storage* oder eine Datenbank mit Hilfe eines Cordova Plugins zu verwenden. *Collections* gehören aber zu den Schlüsselfunktionen von Meteor und bieten gute Synnergie mit reaktiven Meteor-Funktionen, wie zum Beispiel den Helpern (siehe Code-Auszug 5.3). Weiterhin können die Datensätze von *Collections*, im Vergleich zu einem reinen *Local Storage*, mit dem **find**-Operator durchsucht und sortiert werden.

Um nicht auf diese Funktionalität verzichten zu müssen, können die Datensätze der Collection seperat abgespeichert werden und synchronisiert werden. Dieser Ansatz wurde

in dieser Meteor Applikation verfolgt. In Code-Auszug 5.4 ist die Klasse **StorageController** implementiert worden. Diese ist dafür zuständig, Datensätze im *Local Storage* abzuspeichern und anschließend mit der *Collection* zu synchronisieren. Die Daten der *Collection* werden in serialisierter Form im *Local Storage* abgespeichert. Bei einem Neustart der Applikation werden dann die Daten des *Local Storage* deserialisiert und in die *Collection* eingefügt. Im erweiterten Sinne ist dieser Prozess eine Synchronisation.

```
1  class StorageController{
2    constructor(){}
3    initStorage(){
4      this.initHausaufgaben();
5      this.initContext();
6      this.syncStorageWithCollections();
7    }
8    initHausaufgaben(){
9      hausaufgaben = localStorage.getItem(this.keys.hausaufgaben);
10     if(!hausaufgaben){
11       hausaufgaben = [];
12       localStorage.setItem(this.keys.hausaufgaben,hausaufgaben);
13     }
14   }
15 /*
16  weiter Funktionen und Variablen der Klasse sind implementiert,
17  werden aber hier nicht aufgelistet
18 */
19   syncStorageWithCollections() {
20     HausaufgabenCollection.remove({});
21     hausaufgaben = localStorage.getItem(this.keys.hausaufgaben);
22     if(!hausaufgaben){
23       hausaufgaben = [];
24     }
```

```
25      else{
26        hausaufgaben = JSON.parse(hausaufgaben);
27      }
28      if (hausaufgaben) {
29        for (let i = 0; i < hausaufgaben.length;i++){
30          HausaufgabenCollection.insert(hausaufgaben[i]);
31        }
32      }
33    }
34
35  }
```

Listing 5.4: Teil der StorageController-Klasse der Meteor-Applikation

In Zeile 3 des *Storage Controllers* wird die Methode `initStorage()` definiert. In dieser Methode werden zuerst die Datensätze für Hausaufgaben im *Local Storage* initialisiert. Dabei wird der Local Storage an der Stelle `this.keys.hausaufgaben` ausgelesen. Falls noch nichts an dieser Stelle abgespeichert ist, wird ein leerer Array in den *Local Storage* geschrieben. Danach werden die Kontext-Daten, beispielsweise Uhrzeiten zu denen der Benutzer aufsteht, initialisiert. In Zeile 7 wird letztendlich die Methode `syncStorageWithCollections()` aufgerufen. Dabei werden zuerst alle Datensätze der *Collection* gelöscht, anschließend werden die Daten der Hausaufgaben aus dem *Local Storage* gelesen und mit Hilfe des Befehls `JSON.parse()` in ein *JSON*-Objekt konvertiert. Über dieses Objekt wird dann in einer Schleife iteriert und die *Colletion* befüllt.

5.2.4 Lesen und Abspeichern von Dateien

Viele Hausaufgaben enthalten Multimedia-Elemente, die theoretisch vom Therapeuten an den Patienten gesendet werden. Da diese Applikation nur als Unterstützung zur nachfolgenden Evaluation herangezogen werden soll, ist es mit dieser Applikation nicht möglich Hausaufgaben herunterzuladen. Um trotzdem mit Hilfe der Applikation testen und evaluieren zu können, können Hausaufgaben beispielhaft generiert werden. Für die Generierung von Hausaufgaben mit einem Multimedia-Element müssen entsprechende Multimedia-Daten im Speicher hinterlegt werden.

```
1  HTTP.call("GET", "/data/audio.json",
2    (error, result)=>{
3      if(error)
4        alert("Failed to load Data:"+error);
5      let base64;
6      if(result.data){
7         base64 = result.data.data;
8      }
9      else{
10       base64 = JSON.parse(result.content).data;
11     });
12 });
```

Listing 5.5: Auslesen einer Datei in der Meteor-Applikation

Dateien können in Meteor-Applikationen im *public*-Ordner abgespeichert werden. Beim *Build*-Prozess von mobilen Meteor-Applikationen, wird der *public*-Ordner in die Applikation eingebaut. Bei Meteor-Applikationen für den Browser ist das nicht der Fall, für mobile Applikationen jedoch wird der Ordner direkt heruntergeladen um den Datenverkehr, der beim Nachladen von Bildern und anderen Dateien entstehen würde, zu minimieren. Beim Lesen von Dateien aus dem *public*-Ordner kann auf eine *File-Reader*-Bibliothek verzichtet werden. In Code-Auszug 5.5 aus der Meteor-Applikation ist der Zugriff auf eine JSON-Datei, die eine Base64-codierte MP3-Datei enthält, zu sehen. Auf dem mobilen

Gerät wird der *HTTP-GET*-Befehl umgeleitet und Meteor liest eigenständig die Datei aus dem Filesystems des mobilen Gerätes aus.
Bei der Generierung von Hausaufgaben wird der Code-Auszug 5.5 verwendet, um an die Multimedia-Daten für die Hausaufgabe zu kommen. Anschließend wird der Base64-codierte String im *Local Storage* mit einem Vermerk, zu welcher Hausaufgabe er gehört, abgespeichert. Wird diese Hausaufgabe anschließend vom Nutzer gestartet, wird dieser String wieder aus dem *Local Storage* ausgelesen und für die Darstellung des Multimedia-Elements verwendet.

5.2.5 Cordova Plugins & Kommunikation mit Hardware

Für die Erstellung von Benachrichtigungen auf dem Betriebssystem oder auch das Auslesen der GPS-Position, muss die Applikation mit dem Betriebssystem und teilweise auch mit dessen Hardware kommunizieren. Die Applikation verwendet zwei Cordova Plugins: das *HTTP*-Plugin [20] der Version 1.1.0 und das *Cordova Local-Notification Plugin* [21] der Version 0.8.4. Das *HTTP-Plugin* ist notwendig, weil der Webview selbst keine Verbindung mit einem REST-Server aufbauen darf, da dies gegen die sogenannte *Same Origin Policy* verstößt. Das Cordova-Plugin ist nicht Teil des Webviews und unterliegt nicht dieser Richtlinie. Dadurch kann die Appliaktion *Cross Origin Requests* ausführen. Das *Local-Notification Plugin* wird benötigt um Benachrichtigungen auf dem Mobilgerät zu erstellen, da dies zum momentanen Zeitpunkt außerhalb der Möglichkeiten von Webviews liegt.

Es gibt aber auch Funktionen die der Webview bereitstellt um mit dem Betriebssystem direkt ohne über ein Plugin zu kommunizieren. Diese Möglichkeiten ergeben sich zum einen durch HTML5, aber auch durch globale Variablen des Webviews in Javascript, wie zum Beispiel dem *navigator*-Objekt. Ein Feature, was mit HTML5 eingeführt wurde, ist die Verwendung von Telefonnummern in einem Link. Beispielsweise kann mit dem *HTML-Tag* `<a href="tel:3-231-331-5421">Click to call!</a>` die Telefon-Applikation des Gerätes mit der schon ausgefüllten Telefonnummer ge-

öffnet werden. Im Fall einer Patienten-Applikation könnte das die Telefonnummer des Therapeuten sein.

Die GPS-Position eines Gerätes kann ebenfalls ohne Cordova-Plugins ausgelesen werden. Das globale *navigator*-Objekt enthält ein *geolocation*-Objekt und dieses stellt die Methode `getCurrentPosition()` bereit, über welche die GPS-Position des Benutzer ausgelesen werden kann. Eine weitere Funktion des *navigator*-Objektes ist die Überprüfung, ob das Gerät online oder offline ist. `navigator.onLine` enthält einen Boolean, der *true* ist, falls das Gerät online ist.

5.3 Ionic 2

Nach der Vorstellung der Meteor-Applikation, beschäftigt sich dieses Unterkapitel mit der Realisierung der Ionic-Applikation und erklärt deren Aufbau und Funktionsweise.

5.3.1 Architektur

Wie in Abbildung 5.7 dargestellt ist, besteht die Ionic-Applikation aus Services, einer Model-Klasse für Hausaufgaben, Pages und Elementen zum Navigieren zwischen den Pages.

Die Ionic-Applikation startet auf der *Home*-Page. Wie in Kapitel 3.2 erklärt sind Pages *Component*-Klassen, die aus einem HTML-Template und der Klasse selbst bestehen. Um eine übersichtliche Projekt-Struktur zu erreichen, wurden die HTML-Templates für alle *Components* der Applikation in separate HTML-Dateien ausgelagert. Ein essenzieller Teil der Applikation ist die Liste der noch offenen Hausaufgaben. In Code-Auszug A.1 ist die *Component*-Klasse der Hausaufgaben-Page zu sehen. Das dazugehörige Template ist im gleichen Ordner, wie die *Component*-Klasse und wird im *decorator* der Klasse angegeben. In Code-Auszug A.2 ist das Template der Hausaufgaben-Page zu sehen. Viele der verwendeten HTML-Tags, wie zum Beispiel der *ion-list*, gehören nicht zur ursprünglichen HTML-Syntax, sondern sind von Ionic bereitgestellte Components.

Für diese Components wurden diese HTML-Tags als *selector* festgelegt und werden von der Angular-Engine dann an entsprechender Stelle eingefügt.

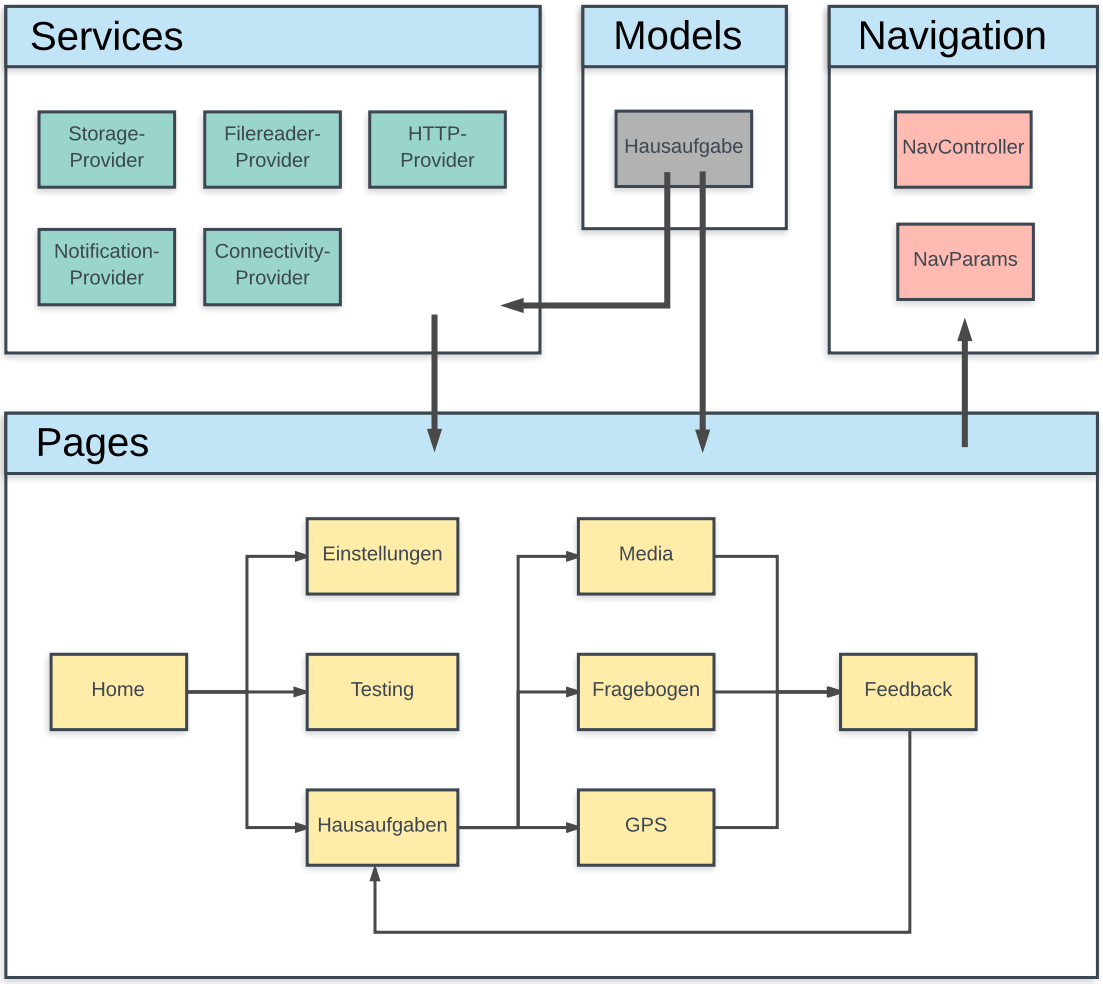

Abbildung 5.10: Architektur-Schaubild der Ionic-Applikation

Das Template der Hausaufgaben-Page enthält zum einen einen **ion-header**. Dieser wird in vielen Templates verwendet und fügt eine Navgiationsleiste ein, die einen Titel und einen Zurück-Knopf enthält. Der **ion-content** beginnt unter der Navigationsleiste und enthält die Liste der Hausaufgaben. Diese Liste wird mit Hilfe der *ngFor-directive* dynamisch befüllt und nutzt die Variable **hausaufgaben** der *Component*-Klasse. In Code-Auszug A.1 sieht man, dass die Variable einmal zu Beginn im Konstruktor der Klasse befüllt wird und danach beliebig oft aktualisiert werden kann, durch den *ion-*

refresher und die Methode `refreshHausaufgaben()`. Der *ion-refresher* ist ein Ionic-*Component*, mit dem der Benutzer beim Herunterziehen der Seite mit einer Fingerbewegung einen Ladevorgang starten kann. Jedes Listen-Element, beziehungsweise jede Hausaufgabe wird mit einem *Click-Listener* ausgestattet, in welchem die Methode `openHausaufgabe(id, type)` ausgeführt wird. Wobei jede Hausaufgabe eine spezifische ID und einen Typ hat, die der Methode übergeben werden. Beim Ausführen der Methode wird überprüft um welchen Typ es sich bei der angewählten Hausaufgabe handelt. Je nach Typ wird dann der mit Hilfe des *NavControllers* zu einer anderen Page navigiert.

Wenn es sich beispielsweise um eine Audio-Übung handelt, wird mit dem Befehl `this.navCtrl.push(MediaPage,{hausaufgabenId:id});` zur *Media*-Page navigiert. Der *NavController* ist eine von Ionic bereitgestellte Klasse zum Navigieren zwischen Pages. Wird jedoch nur zur *Media*-Page navigiert, ist aber an dieser Stelle nicht klar, welche Medien für welche Hausaufgabe dargestellte werden sollen. Deshalb wird der Methode *push* des NavControllers zusätzlich ein JSON übergeben, in welchem die ID der Hausaufgabe enthalten ist. In der Media-Page kann dann die ID der Hausaufgabe mittels der Klasse *NavParams* ausgelesen werden, um dann den Inhalt der Hausaufgabe darzustellen.
Da Ionic 2 in TypeScript geschrieben wird, ist es möglich *Model*-Klassen zu implementieren. Dies wurde in der Applikation in Form einer *Model*-Klasse für Hausaufgaben umgesetzt, wie in Code-Auszug A.3 gezeigt ist. Hausaufgaben bestehen aus einer ID, einem Titel, einem Text und einem Typ. Für jede dieser Variablen gibt es sogenannte *Getter*-Methoden, die den Wert der Variable zurückgeben. Zur Ausgabe der Hausaufgabe in der Konsole wurde die Methode `toString()` und zum Serialisieren des Hausaufgaben-Objektes wurde die Methode `getAsJSON()` implementiert. Eine Hausaufgaben-Objekt kann mit dem Befehl `new Hausaufgabe(id,title,text,type)` erstellt werden. Wie in anderen objektorientierten Programmiersprachen ist die Hausaufgaben-Klasse wie ein neuer Datentyp, der in Arrays auftauchen und in Methoden übergeben werden kann.

5.3.2 User Interface

Ionic 2 bringt mit den vielen nativen *Components* ein eigenes CSS-Design mit. Dieses Design muss nicht zwangsweise verwendet werden, es ist möglich CSS-Frameworks einzubinden. Jedoch ist das Design auf mobile Applikationen optimiert und bietet damit eine hohe Qualität im Hinblick auf gute Benutzbarkeit, beziehungsweise Bedienbarkeit (NFA04) und im Hinblick auf ein gutes *Look & Feel* (NFA05). Diese nicht-funktionalen Anforderungen sind in Kapitel 4.3 definiert.

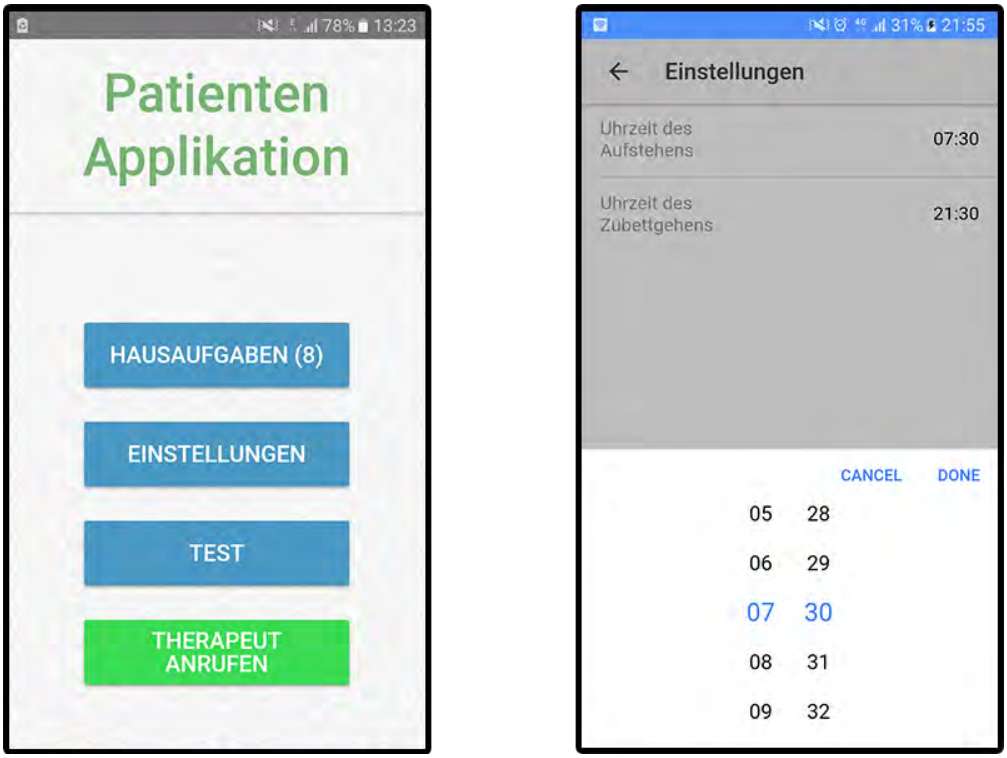

Abbildung 5.11: Implementierung des Startscreens und der Einstellungs-Seite für Kontexte in der Ionic-Applikation

Um einen Einblick in das Design des User-Interface zu geben, werden im Folgenden einige Screenshots der Ionic-Applikation vorgestellt. In Abbildung 5.11 ist auf der linken Seite der Startbildschirm der Applikation zu sehen. Dieser Screen beinhaltet drei Navigations-Buttons, die zur Liste offener Hausaufgaben, zu den Einstellungen der Kontexte und

zu einer Test-Page, die zur Demonstration verschiedener Funktionen verwendet wird, führen. Weiterhin gibt es einen Button, mit dem der Patient einen Anruf an den Therapeuten initialisieren kann. Auf der rechten Seite der Abbildung ist die Einstellungs-Page zu sehen. Hier kann der Patient Kontexte für Übungen genauer bestimmen, wie zum Beispiel das Festlegen einer Uhrzeit für einen bestimmten Tagesabschnitt. Weiterhin ist in der unteren Hälfte des Screns der Ionic-*Component ion-datetime* zu sehen, mit dem eine Uhrzeit ausgewählt werden kann.

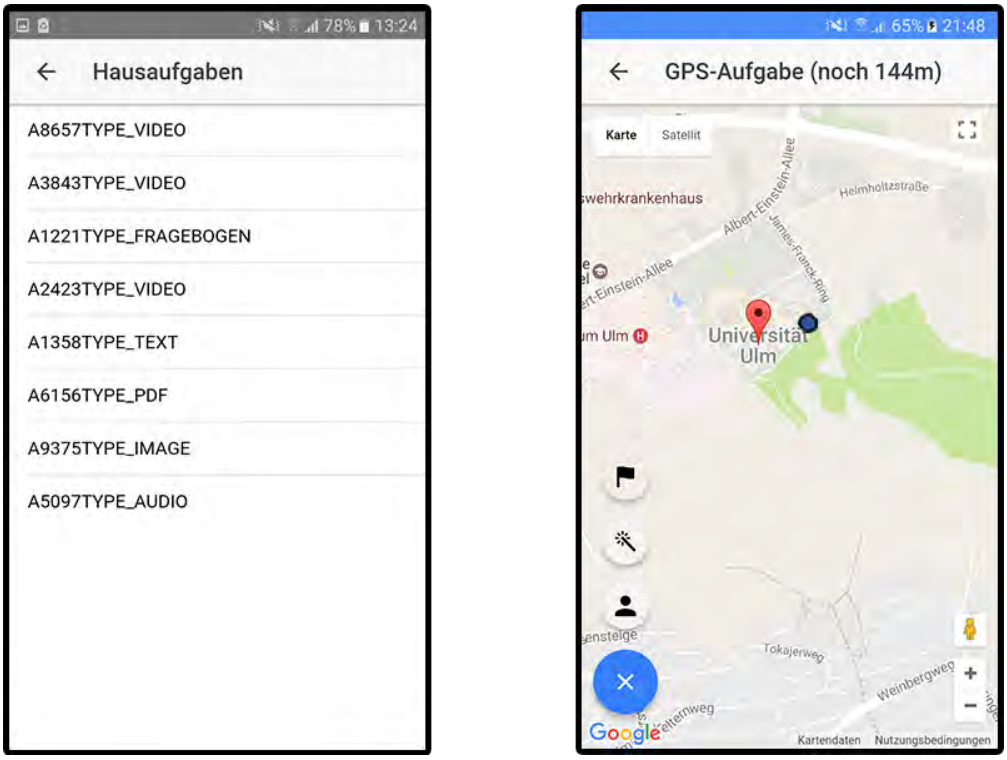

Abbildung 5.12: Implementierung der Liste offener Hausaufgaben und der GPS-Übung (Karte ist Teil von Google Maps [19] und Eigentum von Google) in der Ionic-Applikation

Auf der linken Seite der Abbildung 5.12 ist die Liste offener Hausaufgaben zu sehen. Die Listen-Elemente sind untereinander angeordnet und enthalten den Titel der jeweiligen Hausaufgabe. Auf der rechten Seite der Abbildung ist die Implementierung der GPS-Page zu sehen. In der Navigationsleiste wird die Distanz bis zum Ziel in Metern

angezeigt. Den Rest des Screens nimmt die Google-Maps-Karte ein, diese wurde mit Hilfe der Google Maps Javascript-API eingebettet. Der rote Marker auf der Karte stellt die Ziel-Position dar. Der blaue Punkt zeigt die aktuelle Position des Patienten. Unten Links wurde eine Liste von *Floating-Action-Buttons* eingefügt. Mit dem untersten Button kann die Liste versteckt werden.

Mit dem Button, der eine Flagge enthält, wird die Ziel-Position auf der Karte zentriert. Mit dem Button, der ein Benutzer-Icon enthält, wird die Karte auf die Position des Benutzer zentriert. Um die Applikation im Rahmen einer Präsentation vorstellen zu können, wurde der Button, der ein Zauberstab-Icon enthält, hinzugefügt. Wenn dieser aktiviert wird, wird die Ziel-Position auf die Position des Benutzers gesetzt, um das Erreichen des Ziels zu simulieren.

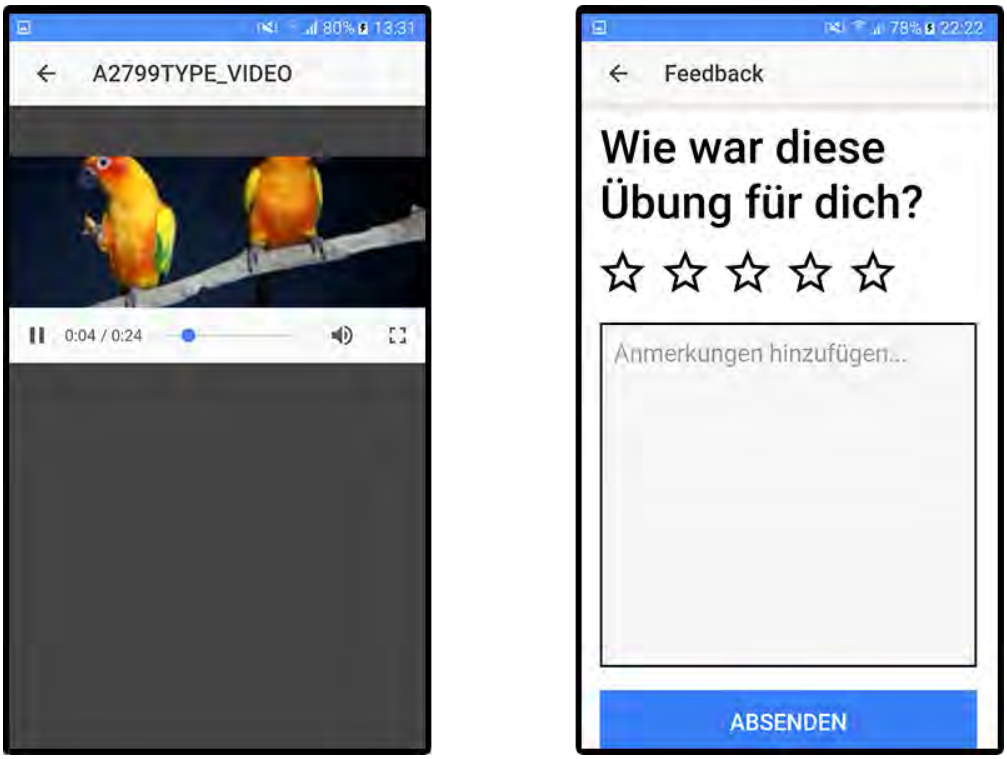

Abbildung 5.13: Implementierung der Media-Page, in dem eine Video-Übung abgespielt wird und der Feedback-Page in der Ionic-Applikation

In Abbildung 5.13 ist auf der linken Seite die Media-Page zu sehen. Die oben angeordnete Navigationsleiste enthält den Titel der Hausaufgabe. Unter der Navigationsleiste wird das Video der Übung, welches mit dem HTML5-Video-Player dargestellt wird, dargestellt. Der HTML5-Video-Player bietet Kontrollelemente zum Stoppen, zum Ändern der Lautstärke und zum Aktivieren des Vollbildmodus. Wenn das Video vollständig geschaut wurde, wird automatisch zur Feedback-Page gewechselt, welche in Abbildung 5.13 auf der rechten Seite zu sehen ist.

Die Feedback-Page wird nach jeder Hausaufgabe geöffnet, um dem Patienten die Möglichkeit zu geben, die Übung zu bewerten und Anmerkungen hinzuzufügen. Zur schnellen Bewertung der Übung kann der Patient mindestens einen und maximal fünf Sterne vergeben. Bei der Auswahl von Sternen werden diese schwarz ausgefüllt. Neben den Sternen kann der Patient mit der Textbox die Übung mittels Text genauer bewerten oder beispielsweise Fragen zur Übung stellen. In diesem Screen sieht man beim Button zum Absenden des Feedbacks zusätzlich, dass alle Pages unabhängig von der Bildschrimgröße des Gerätes dargestellt werden können, da alle Inhalte von einem *Scrollcontainers* umgeben sind.

5.3.3 Services

Mit Ionic 2 können Code-Blöcke, die logisch losgelöst von anderem Code sind, in modularen Service-Klassen definiert werden. Diese Service-Klassen können mit Hilfe eines *Providers* anderen Klassen, wie *Page-Components* zur Verfügung gestellt werden. In der Ionic-Applikation wurden alle Service-Klassen im *Root-Module* der Applikation als *Provider* eingetragen. Dies führt dazu, dass beim Starten der Applikation für jede Service-Klasse eine neue Instanz, auf die alle *Page-Components* zugreifen können, erstellt wird.

Mit dem *Connectivity-Provider* (siehe Code-Auszug A.5) kann in der Applikation zu jedem Zeitpunkt festgestellt werden, ob ein Internetzugriff existiert oder nicht. Der

FileReader-Provider kann Dateien aus dem *asset*-Ordner lesen, dies ist im nächsten Unterkapitel genauer erklärt. Eine Server-Verbindung kann mit Hilfe des HTTP-Providers getestet werden. Dieser Service wäre in einer Implementierung für den realen Gebrauch verantwortlich für die Kommunikation mit einer REST-API. In dieser Applikation wird zu Demonstrationszwecken jedoch nur beispielhaft mit einem Server kommuniziert. Der *Notification-Provider* ist für das Erstellen von Benachrichtigungen zuständig und wird in Unterkapitel 5.3.5 genauer behandelt.

Über den *Storage-Provider* können Components Daten permanent abspeichern und lesen. In Code-Auszug 5.6 ist die Methode `addHausaufgabe()` des *Storage-Providers*, als Beispiel für seine generelle Funktionsweise, zu sehen. Diese Methode wird aufgrufen, um eine Hausaufgabe abzuspeichern. Für den Zugriff auf den *Local Storage* der Applikation wird die von Ionic bereitgestellte *Storage*-Klasse mit dem Befehl `import {Storage} '@ionic/storage';` importiert. Die Methode bekommt eine Hausaufgabe und dazugehörige Daten, wie Base64-codierte Mediendaten oder eine GPS-Position, als Parameter übergeben. Direkt zu Beginn wird ein **Promise** zurückgegeben, da in der Methode asynchroner Code ausgeführt wird. Zunächst werden schon existierende Hausaufgaben mit Hilfe der Methode `getHausaufgaben()` gelesen. Danach werden die schon existierenden Hausaufgaben und die neue Hausaufgabe in einem Array zusammengeführt. Da im Storage nur *Key-Value*-Paare abgespeichert werden können, müssen die Hausaufgaben in das JSON-Format konvertiert werden.

Nun werden zwei asynchrone Aufrufe getätigt. Zum einen wird mit Hilfe des Befehls `this.storage.set(...)` der Hausaufgaben-Array abgespeichert. Zum anderen werden die zur Hausaufgabe gehörenden Daten separat abgespeichert. Dafür wird eine weitere Methode des *Storage-Providers* genutzt. Um sicherzustellen, dass beide asynchronen Aufrufe ausgeführt wurden, kann der Methode `Promise.all(...)` ein Array aus mehreren *Promises* übergeben werden, um diese in einem *Promise* zu vereinen. Wenn die beiden asynchronen Aufrufe ausgeführt wurden, kann letztlich das zu Beginn erstellte *Promise* mit der Methode `resolve()` aufgelöst werden.

```
addHausaufgabe(_hausaufgabe: Hausaufgabe, _data: any) {
  return new Promise((resolve) => {
    let hausaufgabenArr: Hausaufgabe[] = [_hausaufgabe];
    this.getHausaufgaben().then((val: Hausaufgabe[]) => {
      if (val && val.length && val.length > 0) {
       hausaufgabenArr = hausaufgabenArr.concat(val);
      }
      let hausaufgabenArrJSON = [];
      for (let i = 0; i < hausaufgabenArr.length; i++) {
       hausaufgabenArrJSON.push(hausaufgabenArr[i].getAsJSON());
      }
      let promise1 = this.storage.set(
       this.keys.hausaufgaben,
       hausaufgabenArrJSON
      );
      let id= _hausaufgabe.id;
      let promise2 = this.setHausaufgabenData(id,_data);
      Promise.all([promise1,promise2]).then(()=>{
        resolve(true);
      })
    });
  });
}
```

Listing 5.6: Methode *addHausaufgabe()* als Ausschnitt des *Storage-Providers* der Ionic-Applikation

5.3.4 Lesen und Abspeichern von Dateien

Zum Hinterlegen von Dateien gibt es in Ionic den Projektordner **assets**. Die Ionic-Applikation nutzt diesen Ordner hauptsächlich für das Laden von JSON-Dateien, die

Base64-codierte Audio-, Video-, Bild- und PDF-Daten enthalten. Diese Daten werden verwendet, um Hausaufgaben zu Testzwecken zu generieren. Bei der Generierung der Hausaufgaben, müssen Dateien eingelesen werden. In der Ionic-Applikation wurde für diese Aufgabe ein Service erstellt, der in Code-Auszug A.4 zu sehen ist.

Der *FileReader-Provider* benutzt die HTTP-Klasse von Angular und stellt die Methode `getData(type)` bereit. Diese Methode bekommt einen Hausaufgaben-Typ als Parameter übergeben, um innerhalb der Methode feststellen zu können, welche Beispiel-Hausaufgaben geladen werden sollen. Je nach Typ wird die entsprechende URL der JSON-Datei für die darauf folgende HTTP-Anfrage festgelegt. Ionic erkennt anhand der URL, dass es sich nicht um eine echte HTTP-Anfrage handelt, sondern aus dem *assets*-Ordner gelesen werden soll. Somit sind in Ionic keine *FileReader*-Bibliotheken und *Array-Buffer* zum Lesen von Dateien notwendig.

5.3.5 Cordova Plugins & Kommunikation mit Hardware

Einige Funktionen der Ionic-Applikation bauen auf Ionic- und Cordova-Plugins. Ionic-Plugins sind von Ionic herausgegebene Plugins die größtenteils andere Cordova-Plugins enthalten, aber eigene Methoden zur Benutzung bereitstellen. Im Folgenden sind alle verwendeten Plugins aufgelistet:

- BackgroundMode-Plugin
- Geolocation-Plugin
- HTTP-Plugin
- SQLite-Storage-Plugin
- Crosswalk-Webview-Plugin
- Local-Notifications-Plugin

Das *BackgroundMode*-Plugin wird benötigt, um die Position des Nutzers zu verfolgen und anhand dieser Übungen zu starten, auch wenn der Patient die Applikation nicht

geöffnet hat. Um die Position des Nutzers auslesen zu können, wurde das *Geolocation*-Plugin verwendet. Bei der Kommunikation mit der REST-API können aus dem Webview selbst keine Anfragen aufgrund der *Same-Origin-Policy*, deshalb wird das *HTTP*-Plugin verwendet. Mit diesem Plugin werden Anfragen aus dem Webview ausgelagert und vom Plugin übernommen. Somit wird die *Same-Origin-Policy* des Webviews umgangen und es kann mit einer REST-API kommuniziert werden.

Der *Storage-Provider* nutzt das *SQLite-Storage*-Plugin, um Daten zu verwalten. Dieses Plugin stellt Methoden zum Lesen und Abspeichern von *Key-Value*-Paaren bereit. Diese werden je nach Verfügbarkeit in einer *IndexedDB*, in einer *WebSQL-Database* oder im *LocalStorage* des Webviews gespeichert. Um auf der Zielplattform den bestmöglichen Webview zur Verfügung zu haben und damit auch auf Funktionen wie eine *IndexedDB* Zugriff zu haben, wurde das *Crosswalk-Webview*-Plugin hinzugefügt. Statt den Standard-Webview des Betriebssystems zu verwenden, wird ein eigener Webview installiert. Damit wird eine höhere Kompatibilität erreicht.

Das letzte essentielle Plugin ist das *Local-Notifications*-Plugin von Ionic. Um dem Patienten Benachrichtigungen anzuzeigen, wird ein solches Plugin benötigt, da Webviews nicht die nötigen Berechtigungen dazu haben. In Code-Auszug 5.7 wird eine Methode des *Notification-Providers* gezeigt. Die Methode `setGPSNotification(...)` wird beim Verfolgen der Position des Patienten genutzt. Falls der Patient einen bestimmten Ort erreicht hat, wird er benachrichtigt, dass eine Übung für ihn bereit ist. Der *Notification-Provider* nutzt das *Local-Notifications*-Plugin, um Benachrichtigungen zu erstellen. Dazu stellt das Plugin die Methode `schedule()` bereit. Diese kann eine oder mehrere Benachrichtigung entgegennehmen und diese dem Betriebssystem mitteilen. Dabei wird eine ID angegeben, um die Benachrichtigung beim Klicken referenzieren zu können. Mit dem Text- und Titel-Feld werden die Texte festgelegt, die der Benutzer sieht, wenn die Benachrichtigung aktiv ist. Im *data*-Feld können diverse Daten mitgegeben werden, die ebenfalls beim Klicken der Benachrichtigung ausgelesen werden können. In diesem Fall sind das die ID und der Typ der Hausaufgabe.

```
setGPSNotification(_hausaufgabenId, _type){
    LocalNotifications.schedule([{
      id:2,
      title: "Uebung bereit (GPS)!",
      text: "Uebung wurde durch deinen Standort ausgeloest.",
      data:{
        "id": _hausaufgabenId,
        "type": _type
      }
    }]);
  }
```

Listing 5.7: Methode *setGPSNotification(...)* als Ausschnitt des *Notification-Providers* der Ionic-Applikation

Um reagieren zu können, wenn eine Benachrichtigung geklickt wird, kann ein spezieller *Listener* implementiert werden. Wichtig ist, dass dieser *Listener* erst aktiviert wird, wenn die Applikation betriebsbereit ist. Dies ist der Fall im *device ready callback*. Der *Listener* wird mittels `LocalNotifications.on("click",(notification)=>{...})` implementiert. Im Fall der Patienten-Applikation werden nun die Daten der Benachrichtigung ausgelesen und die entsprechende Übung gestartet.

6

Evaluation

Im folgenden Kapitel wird evaluiert, welches Framework sich für die Entwicklung einer Patient-Applikation eignet. Dazu werden die Kenntnisse, die bei der Realisierung gewonnen worden, herangezogen. Dabei werden Vergleiche zwischen den Frameworks und den dazugehörigen Applikationen gezogen. Zusätzlich werden auch generelle Vorteile und Nachteile der Frameworks aufgezeigt. Dazu werden zunächst Kriterien zur Evaluation festgelegt. Darauf folgt ein Anforderungsabgleich, in dem festgestellt wird, mit welchen Applikationen, welche Anforderungen erfüllt werden konnten. Die dann folgenden Unterkapitel behandeln die Kriterien, die vorher definiert wurden. Abschließend folgt ein Fazit, welches alle Kriterien abwiegt und die Frameworks final bewertet.

6.1 Definition der Evaluations-Kriterien

In diesem Abschnitt werden Kriterien zur Evaluation festgelegt. Anhand dieser Kriterien werden Argumente vorgebracht, die entweder für oder gegen die Verwendung eines Frameworks zur Entwicklung einer Patienten betreuenden Applikation sprechen.

Ein wichtiger Vergleichspunkt sind die *UI-Engines* der Frameworks. Zum einen sind damit die Möglichkeiten bei der Gestaltung des User Interface unter Verwendung einer bestimmten *UI-Engine* gemeint, zum anderen aber auch Vor- und Nachteile der Architektur, die hinter dem User Interface steht. Je nach *UI-Engine* variieren Entwicklungsgeschwindigkeit, Komplexität und Erscheinungsbild. Dies gilt es in diesem Kriterium zu untersuchen.

Bei der Realisierung war die Entwicklung und Umsetzung eines Konzeptes für die Datenspeicherung eine der anspruchvollen Aufgaben. Die Patienten-Applikation muss in erster Linie Hausaufgaben und Informationen zu Kontexten, bei welchen Übungen ausgelöst werden, abspeichern können. Dabei hat sich gezeigt, dass beide Frameworks unterschiedliche Ansätze zur Abspeicherung von Daten haben. In Kapitel 6.4 soll untersucht werden, welches Konzept zur Datenspeicherung effizienter und der Aufgabe angemessener ist.

Neben vielen architektonischen Gemeinsamkeiten der Frameworks, unterscheiden sich die Frameworks durch unterschiedliche Versionen von Javascript und haben damit auch unterschiedliche die Programmierung betreffende Möglichkeiten. Ein wichtiger Aspekt ist die Behandlung von asynchronen Aufgaben. Während Ionic 2 mit TypeScript das Verwenden von *Promises* nahelegt, arbeitet Meteor mit herkömmlichen *Callbacks*. Welche Vorteile *Promises* gegenüber *Callbacks* haben und ob *Promises* überhaupt notwendig sind, wird in Kapitel 6.5 behandelt. Als letztes Kriterium wird der Entwicklungsaufwand, sowie die Code-Qualität der Applikationen bewertet. Entscheidend beim Entwicklungsaufwand sind die Einarbeitungszeit, die Zeitdauer, die mit Programmieren verbracht wurde und die Komplexität der Programmiersprache. Die Code-Qualität der Applikationen wird anhand der Übersichtlichkeit und der Struktur des Codes bewertet. Aus guter Code-Qualität folgt meist auch gute Wartbarkeit.

6.2 Anforderungsabgleich

In diesem Kapitel werden die funktionalen Anforderungen, die in Kapitel 4.2 definiert wurden, mit den realisierten Applikationen abgeglichen.

6.2.1 Meteor-Applikation

FA01 Fragebögen mittels Formularen
Die Meteor-Applikation kann Fragebögen mittels Formularen darstellen. Ein Fragebogen besteht dabei aus einem *JSON-Object-Array*, wobei jedes *JSON-Object* in diesem Array

aus einer Frage mit einem Titel und einer Beschreibung besteht, wie in Code-Auszug A.6 zu sehen ist. Beim Ausfüllen des Fragebogens wird jeweils eine Frage auf dem Bildschirm angezeigt, durch die der Nutzer navigieren kann, um den Fragebogen letztlich abzusenden.

FA02 Darstellung von Multimedia-Elementen

Die Meteor-Applikation kann Multimedia-Elemente im Media-Template darstellen. Dabei können mit Hilfe von HTML-Tags Bilder, Audio, Video, sowie formatiertes HTML dargstellt werden. Die Multimedia-Daten sind Base64-codiert und werden den HTML-Tags als *src*-Attribut übergeben. Für das Darstellen von PDFs wurde die Javascript-Bibliothek *PDF.js* (siehe [22]) verwendet.

FA03 Benachrichtigungen erstellen

Die Meteor-Applikation kann Benachrichtigungen an das Betriebssystem senden. Jedoch können Benachrichtigungen nicht für einen späteren Zeitpunkt festgelegt werden, sie werden immer sofort erstellt und angezeigt. Dies hängt mit einem Fehler bei der Übertragung von Uhrzeiten zwischen Meteor und dem Plugin zusammen.

FA04 Auf Benachrichtigungen reagieren

Wenn der Patient eine Benachrichtigung erhält, kann die Applikation eine Übung starten. Dies wurde in der Meteor-Applikation bei Benachrichtigungen, die durch eine GPS-Position ausgelöst werden, demonstriert. Bei Aktivierung einer solchen Benachrichtigung wird eine Video-Übung gestartet.

FA05 Mittels Kontext einen Vorgang auslösen

Die Meteor-Applikation kann eingeschränkt mittels verschiedener Kontexte einen Vorgang auslösen. Durch die eingeschränkte Möglichkeit beim Erstellen von verzögerten Benachrichtigungen, ist es nicht möglich auf Uhrzeiten oder Daten zu reagieren. Die GPS-Position kann jedoch überprüft und abgeglichen werden, um so eine Benachrichtigung zu erstellen.

FA06 Einen Anruf starten
Mit einem Button des Startscreens kann über einen HTML-Link mit einer Telefonnummer die Telefon-Applikation des Mobilgerätes gestartet werden.

6.2.2 Ionic-Applikation

FA01 Fragebögen mittels Formularen
In der Ionic-Applikation können Fragebögen als Formulare ausgefüllt und abgeschickt werden. Fragebögen bestehen aus einem *JSON-Object-Array* in welchem jede Frage einen Titel und eine Beschreibung hat. Diese Struktur ist in Code-Auszug A.6 zu sehen. Für jede Frage wird dem Patienten die Frage und ihre Beschreibung und darunter ein Textfeld zur Beantwortung angezeigt. Der Patient kann durch die Fragen navigieren und den Fragebogen letztlich absenden.

FA02 Darstellung von Multimedia-Elementen
Die Ionic-Applikation kann Multimedia-Elemente in der Media-Page darstellen. Dabei können mit Hilfe von HTML-Tags Bilder, Audio, Video, sowie formatiertes HTML dargstellt werden. Alle Multimedia-Daten sind Base64-codiert. Zum Darstellen von PDFs wurde ein Angular-Package genutzt. Dieses Package stellt einen *PDFViewerComponent* (siehe [23]) bereit.

FA03 Benachrichtigungen erstellen
Die Meteor-Applikation kann Benachrichtigungen an das Betriebssystem senden. Zusätzlich können Benachrichtigungen auch für spätere Zeitpunkte festgelegt werden.

FA04 Auf Benachrichtigungen reagieren
Wenn der Patient eine Benachrichtigung erhält, kann die Applikation eine Übung starten. Dies wurde in der Ionic-Applikation ebenfalls bei Benachrichtigungen, die durch eine GPS-Position ausgelöst werden, demonstriert. In diesem Beispiel wird dann gleichfalls eine Video-Übung gestartet.

FA05 Mittels Kontext einen Vorgang auslösen
Die Ionic-Applikation kann mittels verschiedener Kontexte einen Vorgang auslösen. Dies ist sowohl durch eine GPS-Position, an der sich der Patient aufhält, als auch durch eine Uhrzeit, die der Patient selbst festgelegt hat, möglich. Zum Beispiel kann der Patient für den Kontext Äufstehenëine Uhrzeit festlegen. Zu dieser Uhrzeit wird dann eine Benachrichtigung ausgelöst.

FA06 Einen Anruf starten
Mit einem Button des Startscreens kann über einen HTML-Link mit einer Telefonnummer die Telefon-Applikation des Mobilgerätes gestartet werden.

6.3 UI-Engines

In diesem Kapitel werden die *UI-Engines* Blaze und Angular im Hinblick auf ihren Einsatz in der Patienten-Applikation diskutiert. Sowohl Blaze als auch Angular nutzen HTML-Templates mit einer erweiterten Syntax. Durch die erweiterten Syntaxen können mit beiden Frameworks Variablen im Template eingefügt werden. Weiterhin können HTML-Teile mehrfach oder nur bedingt gerendert werden. In ihren Möglichkeiten beim Darstellen von Inhalten unterscheiden sich die Templates also nicht.

Blaze-Templates werden an einer beliebigen Stelle im Projekt durch den HTML-Tag `<template>` erstellt. Ab diesem Zeitpunkt kann es vollständig verwendet werden und zum Beispiel von einem anderen Template eingebunden werden. In Ionic 2 reicht das pure HTML-Template nicht aus, denn ein Template ist Teil eines *Component*. Dieser *Component* wird in TypeScript als Klasse implementiert und hat damit durch seine Deklaration einen gewissen Overhead. In der Patienten-Applikation gibt es einige Templates, die wenige komplexe Aufgaben haben. Für diese ist das Erstellen von *Components* in Ionic eher zusätzlicher Aufwand.

Ein wichtiger Aspekt von UI-Engines ist das Einbinden von *Event Listenern*. Wie in Kapitel 3.9 gezeigt wird, werden in Angular *Event Listener* im Template deklariert. In Blaze findet das Einbinden von *Event Listenern* unabhängig vom Template-Code in Javascript statt. Das *Event Binding* geschieht zur Laufzeit. Die Referenzierung, welches DOM-Element, welchen *Event Listener* zugewiesen bekommt, wird über ID- und Klassen-Attribute gelöst. Durch die vielen unnötigen Klassen und IDs für das *Event Binding* und zusätzlich durch Klassen und IDs für CSS-Zuweisungen führt dieses Vorgehen führt dazu, dass Templates schnell unübersichtlich werden.

In Blaze wird im Javascript-Code strikt getrennt in *Helper*, *Event Listener* und *Template-Callbacks* (*onCreated, onRendered, ...)*. Diese haben jeweils ihren eigenen Gültigkeitsbereich für Variablen die darin verwendet werden. Dadurch ist es aufwändig Daten zwischen diesen Bereichen auszutauschen. Ein Standardbeispiel dafür ist das deklarieren einer *Reactive Var* im *onCreated-Callback* des Templates und das Verwenden dieser *Reactive Var* in einem *Helper*. Im *Helper* kann dann mittels `Template.instance()` auf die aktuelle Instanz des Templates und damit auf Variablen zugegriffen werden. Im Vergleich dazu bietet Ionic 2 mit seinen *Components* ein einheitliches System. Wenn Variablen als Klassenvariablen mit einem `this.` deklariert werden, sind sie auf Blaze übertragen automatisch *Helper* und können im Code verwendet werden. *Event Listener* rufen eine Methode des *Components* auf, die Zugriff auf alle Klassenvariablen hat. Falls in der Methode eine Klassenvariable manipuliert wird, werden auch die Teile des Templates aktualisiert, die von der Klassenvariable abhängig sind. Ebenfalls stehen in Angular ein Konstruktor und durch Ionic 2 zusätzliche Methoden, wie beispielsweise `ionViewDidEnter()` bereit. Hierbei ist wieder ein direkter Zugriff auf die Klassenvariablen gegeben.

Abschließend lässt sich sagen, dass beide *UI-Engines* ähnlich gute Möglichkeiten haben und für eine Patienten-Applikation ausreichend viele Funktionen besitzen. Einfache Templates lassen sich in Meteor durch den geringeren Overhead schneller implementieren. Komplexe Templates sind in Ionic 2 effizienter und übersichtlicher umsetzbar. Letztendlich ist diese Entscheidung auch abhängig von der Erfahrung und den Präferenzen des Entwicklers.

6.4 Datenspeicherung

Meteor nutzt, wie in Kapitel 3.1.1 beschrieben, *Collections* zur Datenspeicherung. Diese *Collections* sind Teil der *MiniMongoDB* in Meteor-Client-Applikationen. In den beiden Realisierungs-Kapiteln wurde für Meteor in Kapitel 5.2.3 und für Ionic 2 in Kapitel 5.3.3 erklärt, wie Daten in den jeweiligen Applikationen abgespeichert werden.
Zusammenfassend werden in Meteor Daten in Collections bereitgestellt und im *Local Storage* serialisiert. Während in Ionic 2 nur der *Local Storage* zur Verfügung stand. *Collections* sind global verfügbare, reaktive Variablen. Sie werden zum Start der Applikation mit den schon vorhandenen Daten aus dem *Local Storage* gefüllt. *Meteor-Helper* können dann diese Daten aus der Collection verwenden und auf deren Basis UI-Elemente rendern. Wird nun ein Datensatz der *Collection* hinzugefügt, wird der Helper, unabhängig von seinem Template, aktualisiert. Durch diese Reaktivität werden UI-Elemente also automatisch aktualisert.
In Ionic existiert eine solche Reaktivität nicht. Wenn neue Datensätze erstellt werden, muss das User Interface proaktiv aktualisiert werden. Wenn beispielsweise eine Klassenvariable eines *Components* mit Daten aus dem *Local Storage* belegt wird und zu einem späteren Zeitpunkt ein Datensatz im *Local Storage* abgespeichert wird, muss die Klassenvariable manuell aktualisiert werden. Diese Synchronisation der Klassenvariable mit dem *Local Storage* ist möglich, solang der Initiator, der den neuen Datensatz eingefügt hat, der *Component* selbst ist. Falls aber ein Hintergrund-Service, der unabhängig vom *Component* existiert, neue Daten vom Server erhält und diese im *Local Storage* abspeichert, wird der Component davon nicht informiert und kann nicht aktualisiert werden. Die neuen Daten wären erst bei einer neuen Erstellung des Components sichtbar.

Bei der Patienten-Applikation ist eine Reaktivität des Datenspeichers aber nicht unbedingt notwendig. Das herangeführte Beispiel eines Hintergrund-Services tritt in einer Patienten-Applikation, die mit einer REST-API kommuniziert, nicht auf. Bei einer Patienten-Applikation, in welcher der Therapeut aktiv mit dem Patienten über *Web-Sockets* kommunizieren kann, wäre ein solcher Hintergrund-Service aber notwendig. Unabhängig von der Reaktivität bieten Collections weitere Vorteile. In *Collections* kann

nach Datensätzen gesucht werden, Datensätze können sortiert werden und über Datensätze kann iteriert werden. Der zustätzliche Aufwand in Meteor, der sich durch die Serialisierung, Deserialisierung und Synchronisation der Datensätze ergibt, wirkt sich im Vergleich zur reinen Verwendung des *Local Storage* nachteilig aus. Es ist aber auch möglich in Meteor auf *Collections* zu verzichten und sich wie Ionic 2 auf den *Local Storage* zu beschränken. Abschließend bietet Meteor Entwicklern mehr Optionen bei der Datenspeicherung.

6.5 Behandlung von asynchronen Aufgaben

Aus der Anbindung an eine REST-API und die daraus resultierenden HTTP-Anfragen folgt eine Menge an asynchronem Code. Auch Methoden des *Storage* in Ionic 2 sind asynchron, sowie auch viele Plugins asynchrone Methoden bereitstellen. Um festzustellen, wann asynchroner Code abgearbeitet wurde, bieten Meteor und Ionic zwei Unterschiedliche Konzepte. Meteor setzt dabei auf bewährte *Callbacks*, die schon immer essentieller Bestandteil von Javascript sind. In Ionic 2 können ebenfalls *Callbacks* verwendet werden. Da Ionic 2 aber in TypeScript geschrieben wird, werden hauptsächlich *Promises* verwendet. Die generelle Funktionsweise von *Promises* wurde bereits in Kapitel 5.3.3 erklärt.

Promises können als eine Weiterentwicklung von *Callbacks* bezeichnet werden und liefern viele komfortable Verbesserungen für die Behandlung von asynchronen Aufgaben. In Codebeispiel 6.1 wird eine Hintereinanderausführung von asynchronen Arbeitsschritten mit der Nutzung von *Promises* gezeigt. Als Vergleich dazu werden in Codebeispiel 6.2 die gleichen Arbeitsschritte ausgeführt. Anhand dieser Beispiel werden die Vorteile von *Promises* ersichtlich. Um mit *Callbacks* asynchronen Code hintereinander auszuführen, müssen Funktionen geschachtelt werden. Dies ist mit *Promises* nicht mehr nötig und führt zu einer besseren Übersichtlichkeit des Codes. Mit der zu den *Promises* gehörenden Methode `catch()` können Fehler abgefangen werden. Die Methode ist der Ersatz für einen herkömmlichen *try-catch*-Block, wie er im *Callback*-Beispiel zu sehen

ist. Nach Aufruf der `catch()`-Methode kann ein weiteres `then()` platziert werden. In Beispiel 6.2 muss die Methode `methode3()` zwei mal aufgerufen werden, um sicher zu gehen, dass sie mindestens ein mal aufgerufen wird.

```
method1().then((data)=>{
  //do something
  return method2();
}).then((data)=>{
  //do something
  return method3();
}).catch((error)=>{
  //handle error
}).then(()=>{
  //do something no matter if an error occurred or not
});
```

Listing 6.1: Beispiel für eine Hintereinanderausführung von asynchronen Funktionen mit *Promises*

In der *Storag-Provider*-Klasse der Ionic-Applikation (siehe Code-Auszug 5.6) wurde noch eine weitere Besonderheit von Promises gezeigt. In diesem Code können mehrere asynchrone Aufrufe simultan gestartet werden. Die *Promises*, die dabei entstehen können dann mit der Methode `Promise.all(...)` zu einem *Promise* zusammengeführt werden. Mit *Callbacks* bräuchte man einen einen *Counter*, der für jeden abgeschlossenen asynchronen Aufruf hochgezählt wird oder externe Bibliotheken, die eine solche Funktionalität mitbringen.

```
method1(function(data){
  //do something
  method2(data,function(){
    try{
      method3(function(){
```

```
6           //do something if no error occurred
7         });
8         //do something
9       }
10      catch(error){
11        method3(function(){
12          //do something if an error occurred
13        });
14      }
15    });
16  });
```

Listing 6.2: Beispiel für eine Hintereinanderausführung von asynchronen Funktionen mit *Callbacks*

6.6 Entwicklungsaufwand & Code-Qualität

Der Entwicklungsaufwand bei der Implementierung der Applikationen war unterschiedlich groß. Mit Meteor lag die Entwicklunsgdauer deutlich unter der Entwicklungsdauer, die für die Implementierung der Ionic-Applikation benötigt wurde. Die Konzepte von Meteor sind teilweise intuitiver und weniger komplex. Um eine Ionic-Applikation erstellen zu können, benötigt man gute Kenntnisse in Javascript ECMA-Script 6. Dies beinhaltet Kenntnisse über objektorientierte Programmierung. Selbst für triviale Aufgaben müssen Klassen erstellt werden, was die Entwicklung lähmen und die eigentliche Aufgabe verkomplizieren kann. Auch das Erstellen und die korrekte Verwendung von *Services* verlangt eine hohe Einarbeitungsdauer.

Der Vorteil des modularen Ansatzes von Ionic 2 ist ein übersichtlicher und auch im Nachhinein verständlicher Code. Dadurch, dass alle Aufgaben auf unabhängige Klassen verteilt werden, ist es einfacher Strukturen nachvollziehen zu können. Anhand der *import*-Befehle zu Beginn jeder Klasse, kann direkt eingesehen werden können, welche

anderen Klassen verwendet werden. Bei exakter Einhaltung dieses modularen Ansatzes ergibt sich eine hohe Code-Qualität und eine hohe Wartbarkeit.

Meteor gibt keinen solchen Rahmen vor. Der Entwickler muss selbst darauf achten, Code übersichtlich und nachvollziehbar zu gestalten. Eine wichtige Voraussetzung für eine gute Code-Qualität ist eine geeignete Projektstruktur. Diese Projektstruktur gibt Meteor aber nur teilweise vor, Templates können theoretisch alle in einer HTML-Datei definiert werden. Genause können auch *Helper* und *Event-Bindings* in einer Javascript-Datei implementiert werden. Durch diese geringen Vorgaben konnte die Meteor-Applikation im Vergleich zur Ionic-Applikation schneller entwickelt werden. Im Hinblick auf Code-Qualität und daraus folgende Wartbarkeit ist Ionic 2 durch seine hohen Vorgaben tendenziell vorteilhafter.

6.7 Kompatibilität & Zuverlässigkeit

Das Hauptkriterium für eine hohe Qualität ist der Webview des mobilen Gerätes und die API-Version des Betriebssystems, auf dem die Applikation später laufen soll. Verschiedene HTML-Features, sowie auch Javascript-Features benötigen sehr aktuelle Webviews. Die API-Version des Gerätes ist vor allem bei Cordova-Plugins wichtig. Da Cordova-Plugins nativen Code ausführen, kann es passieren, dass Methoden veraltet sind und nicht mehr unterstützt werden oder Methoden in einer API-Version noch nicht zur Verfügung stehen. Das resultiert darin, dass Cordova-Plugins nicht funktionieren.

Um dieses Problem zu beseitigen kann sowohl der Ionic-, als auch der Meteor-Applikation das **Crosswalk**-Plugin hinzugefügt werden. Dieses Plugin installiert einen eigenen Webview auf dem mobilen Gerät, in dem dann die Applikation ausgeführt werden kann. So kann sichergestellt werden, dass alle verwendeten HTML- und Javascript-Features funktionieren. Der Nachteil des *Crosswalk*-Plugins ist, dass der Speicherplatz, den die Applikation auf dem Gerät verbraucht deutlich erhöht wird. Bei der Auswahl von

Cordova-Plugins sollte im Hinblick auf hohe Kompatibilität darauf geachtet werden, dass sie möglichst auch alte API-Versionen der Betriebssysteme unterstützen.

Nachdem beide Applikationen vollständige getestet wurden, ist vor allem auffällig geworden, dass die initiale Ladezeit der Applikationen sehr hoch ist. Die Ladezeit der Meteor-Applikation (circa 4 Sekunden) ist zwar deutlich niedriger als die der Ionic-Applikation (circa 8-10 Sekunden), dennoch ist dies bei der dauerhaften Nutzung ein störender Faktor. Ansonsten ist die Zuverlässigkeit, beziehungsweise die Stabilität der Applikation bei beiden Applikationen gegeben.

6.8 Fazit

Nach der Bewertung der Frameworks anhand der Kriterien, die in Kapitel 6.1 definiert und im Folgenden abgearbeitet wurden, folgt nun ein Fazit. Bevor auf die Eigenschaften der Frameworks eingangen wird, ist zu sagen, dass mit beiden Frameworks der funktionale Anforderungskatalog erfüllt werden konnte. Die einzige Ausnahme ist das Erstellen von zeitlich verzögerten Benachrichtigungen in der Meteor-Applikation.

Im Bezug auf die *UI-Engine* kann man verschiedene Standpunkte vertreten. Beide *UI-Engines* bieten eine breite Palette an Möglichkeiten bei der Gestaltung und Umsetzung des User Interfaces. Wie in Abschnitt 6.3 erklärt wurde, bringt Ionic von sich aus einen gewissen Overhead durch die strikte Modularisierung von logischen Einheiten mit sich. Dafür besitzt Ionic ein wohl strukturiertes Projekt und übersichtlichen Code mit sich. Falls mit der Erstellung der Patienten-Applikation mehrere Leute beschäftigt sind, ist dadurch Ionic 2 besser als Framework geeignet. Bei der Datenspeicherung ist Meteor im Vorteil durch den *Local Storage* in Kombination mit *Collections*. Der Entwickler hat dabei einfach mehr Möglichkeiten und wird mit zusätzlicher Reaktivität im User Interface belohnt.

Im Vergleich zu Meteor ist Ionic 2 mit TypeScript auf dem aktuelleren Stand, durch Typ-Sicherheit und die Verfügbarkeit von Klassen ist Ionic 2 ein mächtigeres Instrument

zum Erstellen von mobilen Applikationen. Durch die vielen *Ionic-Components* wird der Entwicklungsaufwand enorm gesenkt. Man hat nicht das Gefühl, Dinge programmieren zu müssen, die sicher schon jemand anderes effizienter programmiert hat. Ein weiteres Argument, warum Ionic 2 das bessere Framework ist, sind die Ionic-Plugins. Der Vorteil von diesen Plugins ist, dass sich Ionic aktiv darum kümmert, dass diese auch funktionieren. Es gibt sehr viele Cordova-Plugins im Internet und viele davon sind fehlerhaft oder veraltet. Bei Ionic-Plugins kann man sich relativ sicher sein, dass diese auf dem aktuellsten Stand sind. Durch die vielen Features, die Ionic 2 mitbringt, ist die Applikation auch entsprechend schwerfälliger, was sich teilweise in langsameren Antwortzeiten widerspiegelt. Abraten von Meteor als Framework kann man aber auch nicht ohne Weiteres. Die Entscheidung für oder gegen ein Framework ist auch stark abhängig von persönlichen Präferenzen. Letztlich fällt meine eigene Tendenz eher in Richtung des Ionic-Frameworks aus, da es dem Entwickler ein solides, gut funktionierendes und sehr aktuelles Gesamtpaket bietet.

7

Zusammenfassung & Ausblick

In dieser Arbeit wurden die zwei hybriden Frameworks Meteor und Ionic 2 untersucht. Insbesondere im Hinblick auf eine Verwendung in einer Patienten betreuenden Applikation wurden die Architekturen und Funktionsweisen der Frameworks vorgestellt. Dabei wurde besonders darauf eingegangen, welche Besonderheiten und Alleinstellungsmerkmale sie besitzen. Nachdem dann genaue Anforderungen definiert wurden, wurde eine Patienten-Applikation mit beiden Frameworks realisiert. Bei der Realisierung stand nicht im Vordergrund, eine vollständig funktionsfähige Applikation zu bauen. Das Ziel war es, tiefere Erkenntnisse über die Frameworks und ihrer Eignung bei der Entwicklung einer solchen Applikation, zu sammeln.

Mit Hilfe des gesammelten Wissens wurden dann, in Form einer Evaluation, Vor- und Nachteile der Frameworks aufgezeigt. Dabei wurden vor allem betrachtet, welche *UI-Engine* sich besser eignet, welches Framework eine bessere Datenspeicherung aufzeigt, aber auch welche Vor- und Nachteile die Frameworks im Allgemeinen haben. Dazu zählt zum Beispiel die Entwicklungsgeschwindigkeit und die resultierende Code-Qualität, sowie Kompatibilität und Zuverlässigkeit. Letztlich wurde ein Fazit gezogen, in welchem alle Kriterien abgewogen wurden. Dabei ist klar geworden, dass sich beide Frameworks dazu eignen, eine Patienten betreuende Applikation zu bauen. Jedoch wurde eine Tendenz in Richtung des Ionic 2 Frameworks ausgesprochen.

Nach der Realisierung der beiden Applikationen und deren Bewertung, beziehungsweise Vergleich, wird an dieser Stelle noch darauf eingegangen, was an Funktionen zu einer Patienten-Applikation noch hinzukommen könnte und welche Fragen bei der Evaluation offen geblieben sind. Die Bewertung beruft sich auf die Funktionsweisen der Frameworks

und den Erfahrungen, die bei der Realisierung gesammelt wurden. Es wäre aber zusätzlich noch interessant, wie genaue Effizienz-Analysen ausfallen würden. Dabei könnte man feststellen, welche Applikation das bessere Antwortzeit-Verhalten abliefert oder wie schnell Daten gelesen oder abgespeichert werden können. Weiterhin ist Effizienz beim Verwenden von Hardware-Sensoren sehr wichtig, da der Stromverbrauch bei ineffizienten Zugriffen stark ansteigen kann. Dazu wurde bereits in einer Arbeit des Institutes für Datenbanken und Informationssysteme der Universität Ulm eine *Engine* zur effizienten Nutzung von Hardware-Sensoren geschrieben (siehe [24]). Eine weitere Funktion, die eine Patienten-Applikation gebrauchen könnte, ist ein Hintergrund-Service, wie auch viele Nachrichten-Applikationen ihn besitzen. Dieser Hintergrund-Service könnte genutzt werden, um nicht nur von der Applikation aus nach Daten zu fragen (*pulling*), sondern um dem Therapeut zu ermöglichen, auch aktiv der Applikation Daten zukommen zu lassen kann (*pushing*).

Da diese Arbeit eine technische Auseinandersetzung mit Patienten betreuende Applikationen ist, wäre es interessant diese Applikationen an Personen zu testen. Bei diesen Tests könnten Ideen für weitere Funktionen erschlossen werden und es gäbe eine Rückmeldung für die aktuelle Funktionsweise der Applikation, um Verbesserungen einzuleiten. Weiterhin wurde sich im Rahmen dieser Arbeit nicht mit Sicherheitsaspekten befasst. Gemeint sind dabei zum einen die Daten, die auf dem mobilen Gerät gespeichert werden, aber auch die Daten, die zwischen der Patienten-Applikation und der REST-API ausgetauscht werden. Für die Sicherheit bei der Kommunikation zwischen Applikation und Server sollte HTTPS als Kommunikationsprotokoll verwendet werden. Was für weitere Aspekte beachtet und Schritte eingeleitet werden sollten, könnte Teil einer hierauf aufbauenden Arbeit sein, die sich mit der Realisierung einer vollständigen und später einsatzfähigen Patienten-Applikation beschäftigt. Weitere Aspekte, die bei der Entwicklung von medizinischen Applikationen beachtet werden sollten, wurden beispielsweise im Buch *Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health* [25] behandelt.

Letztlich gibt es noch viele Fragen, die sich durch Patientenbetreuung in einer Applikation stellen. Dabei sind viele von diesen Fragen auch nicht-technischer Art. Viele dieser

Fragen sind zum rechtlicher Art, darf die Betreeung eines Patienten überhaupt über eine mobile Applikation stattfinden, beziehungsweise in welchem Ausmaß? Was spricht für oder gegen das Verschreiben von Rezepten über eine solche Patienten-Applikation? Andere offene Fragen sind medizinischer, beziehungsweise psychologischer Art. Wie wirkt sich eine solche Betreuung auf Patienten aus? Kann das Vertrauen zwischen Patient und Therapeut über eine Applikation aufrechtgehalten werden?

Bei der Formulierung dieser Fragen fällt auf, dass diese Aufgabe, eine Applikation zur Patientenbetreuung zu entwickeln, nicht nur eine Aufgabe für die Informatik oder nur eine Aufgabe für die Medizin ist. Diese Aufgabe ist interdisziplinär und benötigt die Mitarbeit von Experten in vielen verschiedenen Bereichen.

Literaturverzeichnis

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

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

[3] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. dissertation, UNIVERSITY OF CALIFORNIA, IRVINE (2000)

[4] Masse, M.: REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media, Inc. (2011)

[5] Crockford, D.: The JavaScript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/pdf/rfc7159.pdf` (2014) Zugriff: 2017-01-05.

[6] Foundation, A.S.: Overview and Architecture of Cordova Applictions. `https://cordova.apache.org/docs/en/6.x/guide/overview/index.html` (2012) Zugriff: 2017-01-04.

[7] Group, M.D.: What is Meteor? `https://guide.meteor.com/#what-is-meteor` (2017) Zugriff: 2017-01-10.

[8] Agüero, F.: 5 reasons why I choose React-Meteor for my projects. `http://fjaguero.com/blog/5-reasons-why-i-choose-react-meteor-for-my-projects/` (2016) Zugriff: 2017-01-11.

[9] Various Contributers: Handlebars. `http://handlebarsjs.com/` (2012) Zugriff: 2017-01-12.

[10] Greif, S.: Understanding meteor publications & subscriptions. https://www.discovermeteor.com/blog/understanding-meteor-publications-and-subscriptions/ (2014) Zugriff: 2017-01-18.

[11] Various Contributers: DDP Specification. https://github.com/meteor/meteor/blob/master/packages/ddp/DDP.md (2014) Zugriff: 2017-01-18.

[12] Foundation, A.S.: Platform Support. https://cordova.apache.org/docs/en/dev/guide/support/index.html#core-plugin-apis (2013) Zugriff: 2017-01-20.

[13] Savkin, V.: Change Detection in Angular 2. https://vsavkin.com/change-detection-in-angular-2-4f216b855d4c (2016) Zugriff: 2017-01-21.

[14] Google Inc.: ARCHITECTURE OVERVIEW. https://angular.io/docs/ts/latest/guide/architecture.html (2016) Zugriff: 2017-01-21.

[15] Google Inc.: Component Decorator. https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html (2016) Zugriff: 2017-01-21.

[16] Drifty Co.: Ionic Component Documentation. http://ionicframework.com/docs/v2/components/ (2016) Zugriff: 2017-01-24.

[17] Drifty Co.: Storage. https://ionicframework.com/docs/v2/storage/ (2016) Zugriff: 2017-01-25.

[18] Mather, C.: Iron Router. http://iron-meteor.github.io/iron-router/ (2013) Zugriff: 2017-02-25.

[19] Google Inc.: Google Maps. https://www.google.de/maps/ (2005)

[20] Various Contributers: Cordova HTTP Plugin. `https://github.com/afkpost/cordova-HTTP` (2014) Zugriff: 2017-02-25.

[21] Katzer, S.: Cordova Local-Notification Plugin. `https://github.com/katzer/cordova-plugin-local-notifications` (2015) Zugriff: 2017-02-25.

[22] Various Contributers: PDF.js. `https://github.com/mozilla/pdf.js/` (2015) Zugriff: 2017-02-26.

[23] Yatsyuk, V.: Angular2 PDF Viewer. `https://github.com/VadimDez/ng2-pdf-viewer` (2016) Zugriff: 2017-02-26.

[24] 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

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

A

Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt, die in anderen Kapiteln referenziert werden.

```
1  import { Component } from '@angular/core';
2  import { NavController, NavParams } from 'ionic-angular';
3  import { StorageProvider } from '../../providers/storage-provider';
4  import { Hausaufgabe } from "../../model/hausaufgabe";
5  import { MediaPage } from "../media/media";
6  import { FragebogenPage } from "../fragebogen/fragebogen";
7  import { GpsPage } from "../gps/gps";
8
9  @Component({
10   selector: 'page-hausaufgaben',
11   templateUrl: 'hausaufgaben.html'
12 })
13 export class HausaufgabenPage {
14   private hausaufgaben:Hausaufgabe[] =[];
15
16   constructor(public navCtrl: NavController, public navParams:
17     NavParams, public storageService: StorageProvider) {
18     this.storageService.getHausaufgaben()
19       .then((val:Hausaufgabe[])=>{
20       this.hausaufgaben = val||[];
```

```
21        console.log(this.hausaufgaben);
22      });
23    }
24    refreshHausaufgaben(refresher){
25      this.storageService.getHausaufgaben()
26        .then((val:Hausaufgabe[])=>{
27        this.hausaufgaben = val;
28        if(refresher){
29          refresher.complete();
30        }
31        console.log(this.hausaufgaben);
32      });
33    }
34    openHausaufgabe(id,type){
35      console.log(id+type);
36      switch(type){
37        case Hausaufgabe.TYPE_IMAGE:
38        case Hausaufgabe.TYPE_PDF:
39        case Hausaufgabe.TYPE_TEXT:
40        case Hausaufgabe.TYPE_AUDIO:
41        case Hausaufgabe.TYPE_VIDEO:
42          this.navCtrl.push(MediaPage,{hausaufgabenId:id});
43          break;
44        case Hausaufgabe.TYPE_FRAGEBOGEN:
45          this.navCtrl.push(FragebogenPage,{hausaufgabenId:id});
46          break;
47        case Hausaufgabe.TYPE_GPS:
48          this.navCtrl.push(GpsPage,{hausaufgabenId:id});
49          break;
50        default:
51          alert("Kein Typ verfuegbar.");
```

```
52          break;
53      }
54    }
55    ionViewDidEnter(){
56      this.refreshHausaufgaben(null);
57    }
58  }
```

Listing A.1: Die Component-Klasse der Hausaufgaben-Page der Ionic-Applikation

```
1  <ion-header>
2    <ion-navbar>
3      <ion-title>Hausaufgaben</ion-title>
4    </ion-navbar>
5  </ion-header>
6  <ion-content>
7    <ion-refresher (ionRefresh)="refreshHausaufgaben($event)">
8      <ion-spinner class="refreshSpinner" name="circles">
9      </ion-spinner>
10   </ion-refresher>
11   <ion-list>
12     <button ion-item *ngFor="let hausaufgabe of hausaufgaben"
13     (click)="openHausaufgabe(hausaufgabe.id, hausaufgabe.type)">
14       {{hausaufgabe.title}}
15     </button>
16     <button ion-item *ngIf="hausaufgaben.length==0">
17       Aktuell gibt es keine weiteren Hausaufgaben.
18     </button>
19   </ion-list>
20 </ion-content>
```

Listing A.2: Das HTML-Template der Ionic-Applikation der Liste offener Hausaufgaben

```
export class Hausaufgabe{

  public static TYPE_VIDEO = "TYPE_VIDEO";
  public static TYPE_AUDIO = "TYPE_AUDIO";
  public static TYPE_IMAGE = "TYPE_IMAGE";
  public static TYPE_TEXT = "TYPE_TEXT";
  public static TYPE_PDF = "TYPE_PDF";
  public static TYPE_FRAGEBOGEN = "TYPE_FRAGEBOGEN";
  public static TYPE_GPS = "TYPE_GPS";

  public get id(): String {
    return this._id;
  }

  public get title(): String {
    return this._title;
  }

  public get text(): String {
    return this._text;
  }

  public get type(): String {
    return this._type;
  }
  constructor(private _id:String, private _title:String,
  private _text:String, private _type:String){}
  getAsJSON(){
    return {
      id: this._id,
      title:this._title,
```

```
32        text:this.text,
33        type:this.type
34      }
35    }
36    toString(){
37      return `
38        Hausaufgabe:
39        id:\$\{this._id\}
40        title:\$\{this._title\}
41        type:\$\{this._type\}
42        text:\$\{this._text\}
43      `;
44    }
45  }
```

Listing A.3: Die Hausaufgaben-Klasse der Ionic-Applikation

```
1  import { Injectable } from '@angular/core';
2  import { Http } from '@angular/http';
3  import 'rxjs/add/operator/map';
4  import {Hausaufgabe} from "../model/hausaufgabe";
5
6  @Injectable()
7  export class FilereaderProvider {
8
9    constructor(private http: Http) {}
10
11   getData(type){
12     let url = "";
13     if(type===Hausaufgabe.TYPE_VIDEO){
14       url = 'assets/data/video.json';
15     }
```

```
16      else if(type===Hausaufgabe.TYPE_AUDIO){
17        url = 'assets/data/audio.json';
18      }
19      else if(type===Hausaufgabe.TYPE_IMAGE){
20        url = 'assets/data/img.json';
21      }
22      else if(type===Hausaufgabe.TYPE_PDF){
23        url = 'assets/data/pdf.json';
24      }
25      return new Promise((resolve)=>{
26        this.http.get(url)
27          .map((res) => res.json())
28          .subscribe((val) => {
29            resolve(val.data);
30          }, (rej) => {console.error("Could not load local data",rej)});
31      });
32    }
33  }
```

Listing A.4: Implementierung des *Filereader-Providers* in der Ionic-Applikation

```
1  import { Injectable } from '@angular/core';
2  import { Network } from 'ionic-native';
3
4  @Injectable()
5  export class ConnectivityProvider {
6
7    constructor(){}
8
9    isOnline(): boolean {
10     if(Network.type){
11       return Network.type !== "none";
```

```
12      }
13      return navigator.onLine;
14    }
15  }
```

Listing A.5: Implementierung des *Connectivity-Providers* in der Ionic-Applikation

```
[
  {
    title: "Beispielfrage 1?",
    description:"Dies ist der Beschreibungstext zu Beispielfrage 1."
  },
  {
    title: "Beispielfrage 2?",
    description:"Dies ist der Beschreibungstext zu Beispielfrage 2."
  },
  {
    title: "Beispielfrage 3?",
    description:"Dies ist der Beschreibungstext zu Beispielfrage 3."
  }
];
```

Listing A.6: Beispiel für einen *JSON-Object-Array* für Fragebögen

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Julius Friedrich Matrikelnummer: 841963

Erklärung

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

Ulm, den .....

Julius Friedrich