Ostfalia
Hochschule für angewandte Wissenschaften
Suderburg
Fakultät Bau-Wasser-Boden
Angewandte Informatik

Aufgabenstellung zur Bachelor-Arbeit

für Herrn Sören Schuba, Matrikel-Nr.: 7045 3010

Thema: Plattformunabhängige Programmierung: Progressive Web Apps

Erstprüfer: Prof. Dr.-Ing. Bernd-Uwe Rogalla

Zweitprüfer: Tristan Knies B.Sc.

Tag der Ausgabe: 14. April 2020

Tag der Abgabe: 26. Juni 2020 **bis 10.00 Uhr**

Bachelorarbeit: Herr Sören Schuba, Matrikel-Nr.: 7045 3010

Thema: Plattformunabhängige Programmierung: Progressive Web Apps

Mit der Weiterentwicklung der Technologien im Bereich der Webentwicklung bieten sich immer mehr Möglichkeiten. So können heute ganze komplexe Anwendungen im Web bereitgestellt werden. Webanwendungen können, soweit der Entwurf stimmt, plattformunabhängig von allen Geräten mit einem Browser aufgerufen werden. Seit etwa 2015 sind Progressive Web Apps verfügbar. Die Unterstützung von Progressive Web Apps in den Browsern ist seit 2015 stark gestiegen, was die Relevanz dieser Technologie für die Zukunft bestätigt.

Als Hauptprojekt dieser Arbeit soll eine Progressive Web App in Angular erstellt werden. Diese beinhaltet ein Authentifizierungsverfahren, in dem sich ein Benutzer registrieren, einloggen, und ausloggen können soll. Sofern der Benutzer eingeloggt ist, soll dieser Freunde hinzufügen und Treffen organisieren können. Um ein Treffen zu organisieren, werden der Ort, die Zeit und die Aktivität angegeben. Nach der Eingabe wird das Treffen für alle Freunde freigegeben. Für die Signalisierung, dass eine neue Anfrage für ein Treffen erstellt wurde, sollen alle Freunde eine Push-Nachricht bekommen. Je nach Belieben der Freunde kann dem Treffen zugestimmt werden.

Das Ziel dieser Arbeit ist es, anhand des Projektes die Umsetzung einer Progressive Web App zu zeigen. Anders als bei normalen Webanwendungen gibt es Merkmale, welche erfüllt werden müssen und Strategien, die in Bezug auf die Offlinefähigkeit sowie die Synchronisierung der Daten zu beachten sind. Ebenfalls soll ein Vergleich zu anderen plattformunabhängigen und -abhängigen Technologien gezogen werden, so dass die möglichen Anwendungsbereiche einer Progressive Web App deutlich werden.

Im Einzelnen sind folgende Punkte zu bearbeiten:

- Vergleich von Progressive Web Apps (PWA) und plattformabhängigen Apps (Android/IOS)
- Analyse und Dokumentation der Anforderungen, Use Cases, Projektplanung
- Architektur der Anwendung (App und Backend)
- Entwurf der App (Mockups), Service Worker, Datenbank, Push-Nachrichten
- Sicherheitsaspekte (Verbindung, Authentifizierung, Autorisierung)
- Implementierung incl. Unittests
- Integrationstest
- Fallbeispiel

Ostfalia
Hochschule für angewandte Wissenschaften
Suderburg

Plattformunabhängige Programmierung: Progressive Web Apps

Abschlussarbeit zur Erlangung des Hochschulgrades Bachelor of Science im Studiengang Angewandte Informatik

an der Ostfalia Hochschule für angewandte Wissenschaften - Hochschule Suderburg
im
Studiengang Angewandte Informatik
von
Sören Schuba, 70453010

Erstprüfer: Prof. Dr.-Ing Bernd-Uwe Rogalla
Zweitprüfer: Tristan Knies B.Sc.
Eingereicht am _____

Inhaltsverzeichnis

Abkürzungsverzeichnis

PWA Progressive Web App

UI User Interface

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

SSL Secure Sockets Layer

TLS Transport Layer Security

CORS Cross-Origin Resource Sharing

CSS Cascading Style Sheets

SCSS Sassy Cascading Style Sheets

SASS Syntactically Awesome Style Sheets

DOM Document Object Model

URL Uniform Resource Locator

HTML Hypertext Markup Language

API Application Programming Interfaceg

JSON JavaScript Object Notation

SQL Structured Query Language

NoSQL Not only Structured Query Language

CLI Command Line Interface

SDK Software Development Kit

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Mit dem Erscheinen der Progressive Web App, hat diese in der plattformunabhängigen Welt einen großen Platz eingenommen. Moderne Webschnittstellen und die starke Weiterentwicklung der Browser ermöglichen Funktionalitäten, auf welche vorher nur die native Anwendungsentwicklung Zugriff hatte. Beispielsweise die Installierbarkeit der Anwendung, Push-Benachrichtigungen, die Fähigkeit auch Offline ausgeführt zu werden oder In-App-Käufe.

Mit der Definition der Merkmale einer Progressive Web App entstand eine Sonderform der Webanwendung. Diese ist von einer nativen Anwendung kaum noch zu unterscheiden und löst das Problem der Plattformabhängigkeit. Daraus entstehen große Vorteile, unter anderem deckt eine Progressive Web App eine Breite von Plattformen ab, die einen relativ modernen Browser auf dem Gerät installiert haben. Die potenzielle Benutzerreichweite steigt somit enorm, was sich als große Bereicherung im Marketing auswirken kann.

Der Gedanke einer Progressive Web App geht sogar so weit in der Zukunft den App Store abzuschaffen (Saborowski 2018). Dieser Gedanke resultiert aus der Fähigkeit der Progressive Web App sich über den Browser zu installieren.

Die Browser-Unterstützung von Progressive Web Apps ist in den vergangenen Jahren stark gestiegen. Die meistgenutzten Browser (Chrome, Firefox, Safari, Edge (Tenzer 2020)) unterstützen die wichtigsten Schnittstellen, welche die Voraussetzung einer Progressive Web App sind (Petereit 2020a).
Die Progressive Web App steht somit für Funktionalität und plattformunabhängige Ausführbarkeit. Doch in welchen Einsatzgebieten ist die Progressive Web App anzuwenden? Welche Funktionen bietet sie? Und wo sind die Grenzen?

1.1. Zielsetzung der Arbeit

Das Ziel dieser Arbeit besteht darin die vorab gestellten Fragen zu beantworten. Hierfür wird das Thema zunächst theoretisch ergründet. Mit Hilfe moderner Ansätze und Frameworks wird ein Beispielprojekt erstellt und somit das Thema in der Praxis erläutert. Innerhalb einer Diskussion wird die Progressive Web App mit anderen Formen der plattformunabhängigen Programmierung und der nativen Anwendungsentwicklung verglichen. Die Vergleiche sollen Aufschluss über die Einordnung innerhalb der plattformunabhängigen Programmierung geben. Die Sicht auf Progressive Web Apps wird hierbei zum einen allgemein und zum anderen innerhalb eines speziellen Falles betrachtet.

Als Projekt wird eine Organisator App entwickelt, in welcher Treffen mit Freunden organisiert werden

können. Diese wird verwendet, um die Umsetzung von den Merkmalen einer Progressive Web App aufzuzeigen. Hierbei ist die Entwicklung einer Single-Page Webanwendung in Angular ein weitergehendes Thema, welches hinzu kommt. Weiterführend soll das Projekt getestet und auf Qualität validiert werden. Daraus ergibt sich ein Leitfaden zur Erstellung einer Progressive Web App.

1.2. Struktur der Arbeit

Als Einführung in das Thema wird die plattformunabhängige Programmierung behandelt. Hierbei werden die Themen „Plattform, Formen der plattformunabhängigen Programmierung und Problematiken“ aufgezeigt. Das nächste Kapitel behandelt das Hauptthema „Progressive Web Apps“ als Teilbereich der plattformunabhängigen Programmierung detailliert. Es werden die einzelnen Merkmale aufgezählt und theoretisch beschrieben, sowie mit kleinen Code-Beispielen dargelegt. Im vierten Kapitel ist die Architektur Angulars erklärt, um die Grundlagen der Webentwicklung in diesem Framework aufzuzeigen. Anhand eines praktischen Beispiels wird ab Kapitel fünf eine vollständige PWA entwickelt. Dieses setzt das theoretisch behandelte Wissen in die Praxis um. Innerhalb einer Anforderungsanalyse sind die Logik, Dynamik und die Statik sowie nicht funktionale Anforderungen der Progressive Web App definiert. Im Kapitel sechs „Projektplanung“ werden die Architektur, das Frontend und das Backend der Organisator App geplant. Die Umsetzung der Progressive Web App wird dabei anschließend mit Hilfe von Code-Teilen des Back- und Frontends beschrieben. Daraufhin werden Tests auf die Anwendung ausgeführt, diese enthalten das Testen des Back-, Frontend und ein Integrationstest. Für das Sicherstellen der Qualität werden Tools zur Validierung vorgestellt. Im Rahmen einer Diskussion werden Vergleiche zur plattformunabhängigen Programmierung sowie der nativen Anwendungsentwicklung gezogen. Nach einem Ausblick und dem Fazit endet diese Arbeit.

2. Plattformunabhängige Programmierung

Zur Erläuterung dieses Themas soll zunächst die Bezeichnung „Plattformunabhängige Programmierung“ geklärt werden. Hierzu werden die Wörter getrennt voneinander betrachtet. Der Begriff „plattformunabhängig“ wird im Duden mit „auf verschiedenen Plattformen lauffähig“(Dudenredaktion o.D.(b)) definiert. Eine „Plattform“ wird sinnhaft als „Basis“(Dudenredaktion o.D.(a)) bezeichnet. Somit ist der Begriff „Plattform“ nicht spezifisch festgelegt. Was genau ist in der Informatik also eine Plattform und wo muss unterschieden werden? Für diese Frage macht es Sinn den Blick auf eine Plattform in mehrere Schichten einzuteilen. Zunächst gibt es das Gerät selbst, welches eine Schicht darstellt. Mit Gerät ist ein Smartphone, Tablet, Computer et cetera gemeint, welches aus physischen elektronischen Bauteilen besteht. Hierbei wird auch von der Hardwareschicht gesprochen. Eine weitere Schicht ist das Betriebssystem. Unter den Geräten selbst gibt es verschiedene Betriebssysteme, wie iOS und Android für Smartphones und Tablets oder Windows und MacOS für Computer. „Ein Betriebssystem umfasst alle Programme eines Computersystems, die den Betrieb des Systems steuern und überwachen“(Schmidt 2005). Folglich dieser Information können Anwendungsprogramme, welche auf einem Betriebssystem lauffähig sind, schon jetzt auf verschiedenen Geräten mit diesem Betriebssystem laufen. Dies kann schon als ein gewisser Grad der Plattformunabhängigkeit angesehen werden. Die letzte Schicht, welche für dieses Thema bekannt sein muss, ist der Programmcode, mit dessen Hilfe Anwendungsprogramme geschrieben werden können. Bei dem Betriebssystem und dem Quellcode wird auch von einer Software-Plattform gesprochen. Betrachtet man die Abhängigkeiten der Plattformen, findet eine immer größer werdende Begrenzung der Geräte statt.

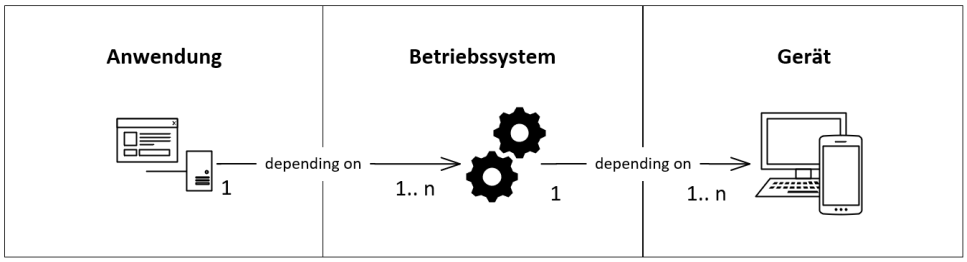

Abbildung 2.1.: Plattformabhängigkeiten

In der gezeigten Abbildung 2.1, Abbildung 2.2, Abbildung 2.3, Abbildung 2.4 steht „n“ für eine begrenzte Anzahl und „*“ für annähernd alle. Es ergibt sich: Nicht jede Programmiersprache kann für jedes Betriebssystem übersetzt werden. Folglich funktioniert nicht jede Anwendung auf allen Betriebssystemen. Nicht jedes Betriebssystem läuft auf jedem Gerät. Schlussendlich kann nicht auf jedem Gerät ein Betriebssystem installiert werden. Dies ist auch der Grund, warum die plattformunabhängige Programmierung häufig als plattformübergreifende Programmierung bezeichnet wird. Es ist praktisch unmöglich ein Programm zu schreiben, welches auf allen Plattformen funktioniert. Ziel der plattformunabhängigen Programmierung ist demnach einen Großteil der Plattformen mit nur ei-

nem Code abzudecken. Aus dieser Eigenschaft heraus ergeben sich aber noch weitere Vorteile, welche im Folgenden benannt werden sollen.

- **Weniger Personal:** Die Programmierung in den einzelnen Plattformen erfordert viel Wissen über die Programmiersprache, bis hin zum Projektaufbau in der Entwicklung der jeweiligen Plattform. In den meisten Fällen wird das Personal explizit für eine Plattform geschult. Bei der mobilen Entwicklung könnte die Unterteilung in iOS- und Android-Entwickler geschehen. Aus diesem Grund wird letztendlich auch mehr spezialisiertes Personal benötigt. Mehr Personal bedeutet auch mehr Kosten.
- **Weniger Zeitaufwand:** In der nativen Anwendungsentwicklung müsste für jede Plattform ein eigenes Projekt aufgesetzt werden. Dieses müsste jeweils einzeln geplant, umgesetzt und gemanagt werden. Das kostet viel Zeit und viel Geld.
- **Größere Reichweite:** Die Möglichkeit mehrere Plattformen abzudecken impliziert auch die Möglichkeit mehrere Benutzer zu erreichen.
- **Konsistenz zwischen den einzelnen Plattformen:** Da der Anwendungscode und somit auch die Benutzeroberfläche auf allen Plattformen gleich ist, sieht die Anwendung auch auf den unterschiedlichen Plattformen gleich aus.
- **Wiederverwendbarer Anwendungscode:** Ein großer Vorteil der plattformunabhängigen Programmierung ist die Wiederverwendbarkeit des Anwendungscodes. Geschriebene Komponenten stehen für mehrere Plattformen zur Verfügung.

Zusammenfassend ist der Hauptgrund also der geringe Kostenfaktor mit einer hohen Lauffähigkeit unter den Geräten (Nitze und Schmietendorf 2014). Die Möglichkeit Code für mehrere Plattformen gleichzeitig zu schreiben bringt jedoch auch Nachteile mit sich. Dadurch, dass nicht für eine spezifische Plattform programmiert wird, stehen teilweise einige Funktionen und Möglichkeiten der Umsetzung nicht zur Verfügung. Diese Nachteile werden in Abschnitt 2.4 näher definiert. Der zweite Begriff „Programmierung“ steht in keinem weiteren Zusammenhang und somit für sich selbst (Dudenredaktion o.D.(c)). Eine Plattformunabhängigkeit kann in verschiedenen Formen umgesetzt werden.

2.1. Webanwendungen

Eine Webanwendung war ursprünglich nicht mehr als ein statisches Dokument, welches von einem Browser interpretiert und dargestellt wurde. Mit Hilfe von Hypertext sollten Informationen über größere Distanzen ausgetauscht und aktualisiert werden(*CERN - European Organization for Nuclear Research - Copyright* o.D.). Der Browser ist ein Anwendungsprogramm auf dem Gerät. Um zu verstehen wodurch eine Plattformunabhängigkeit entsteht, muss zunächst Aufschluss gegeben werden, wie ein Browser grundlegend funktioniert.

Die Darstellung einer Webseite wird in HTML- und CSS-Dateien definiert. Wie die Tags innerhalb einer solchen Datei auszusehen haben, wird von der Normenorganisation W3C festgelegt. Der Browser selbst besteht standardmäßig aus sieben Hauptkomponenten (Garsiel und Irish 2011):

- Einer Benutzeroberfläche
- Einem Browser-Modul, welches als Bindeglied zwischen Benutzeroberfläche und dem Rendering-Modul dient
- Dem Netzwerk-Modul, welches für die Netzwerk-Aufrufe zuständig ist
- Einen JavaScript-Interpreter, welcher JavaScript-Code parsed und ausführt
- Einem UI-Backend, welches für die Darstellung grundlegender Widgets und Fenster benötigt wird
- Dem Datenspeicher, welcher für die persistente Speicherung zuständig ist. Beispiele sind hier Webdatenbanken und der LocalStorage zu denen im Unterabschnitt 3.3.5 nähere Informationen angegeben werden
- Und einem Rendering-Modul. In diesem geschieht die wohl größte Arbeit. Das Rendering-Modul parsed die erstellten HTML- und CSS-Dokumente. Auf diesem Wege wird aus den HTML-Tags ein Inhaltsbaum erstellt. Gemeinsam mit dem Inhaltsbaum und den Style-Informationen (CSS) eine Rendering-Struktur erzeugt. In einem Layout-Prozess werden alle Elemente ihrer Position zugeteilt. Zuletzt wird die Rendering-Struktur vom Painting-Prozess durchlaufen und mit Hilfe des UI-Backend dargestellt

Sobald Interaktionen mit dem Benutzer zustande kommen sollen, werden in der Regel Skriptsprachen verwendet. Skriptsprachen bringen dem eigentlichen Dokument die Dynamik. Durch diese kann von einer Anwendung gesprochen werden. Im Gegensatz zu einer Programmiersprache müssen Skriptsprachen nicht kompiliert werden. So kann über den JavaScript-Interpreter der Skript-Code geparsed und ausgeführt werden, ohne in „Computer-Sprache“ übersetzt zu werden. Die Ausführung geschieht innerhalb einer Laufzeitumgebung im Browser (Avci, Trittmann und Mellis 2013).

In der Webentwicklung wird zwischen zwei Formen einer Webanwendung unterschieden: Der Multi-Page-Webanwendung und der Single-Page-Webanwendung. Die Multi-Page-Webanwendung besteht aus mehreren verlinkten HTML-Dokumenten. Die Single-Page Webanwendung hingegen besteht lediglich aus einem HTML-Dokument. Das Laden der Inhalte geschieht hier dynamisch (*Single-page application vs. multiple-page application* 2016). Der Trend geht eindeutig in Richtung Single-Page-Webanwendungen (Varty 2019, Lozhko 2019). Zwei Gründe dafür sind:

- **Eine reduzierte Serverlast:** Durch das Auslagern der Aufgaben wie zum Beispiel das Laden von Webseiten auf den Client (Einmaliges Laden).
- **Eine Umsetzung selbstständiger Webclients:** Durch zum Beispiel das Verwenden von Service Worker, kann eine Offline-Fähigkeit hergestellt werden. Unabhängig vom Webserver, können nun Interaktionen stattfinden.

Um den Bezug zur plattformunabhängigen Programmierung wiederherzustellen: Mit dem Erstellen einer Webanwendung befindet sich diese noch vor der Schicht der Anwendung selbst. Das heißt, dass eine Betriebssystemunabhängigkeit mit dem Browser selbst geschaffen wurde. Streng betrachtet wird also kein Anwendungsprogramm entwickelt, sondern lediglich eine Definition von Aufbau, Aussehen und Dynamik, welche der Browser umsetzen soll. Es gibt aber auch Nachteile bei der Entwicklung einer Webanwendung. Anders als bei der nativen Anwendungsentwicklung gibt es hierbei nicht die Möglichkeit dessen Schnittstellen zu nutzen. Für die Webentwicklung muss unter anderem für den Zugriff der Kamera oder das Ausführen von HTTP-Requests auf die Webschnittstellen des Browsers zugegriffen werden. Je nach Aktualität des Browsers und dem Browser selbst kann die Unterstützung dieser Webschnittstelle nicht gegeben sein.

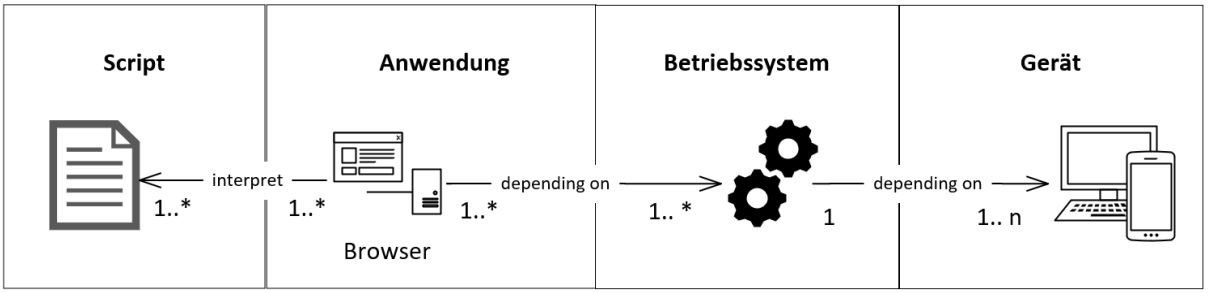

Abbildung 2.2.: Plattformabhängigkeiten - Lösung Webanwendung

2.2. Hybrid-Apps

Wie bereits erläutert gibt es Einschränkungen bei der plattformunabhängigen Entwicklung. Was die hybriden Apps ausmacht, ist der Vorteil einer plattformabhängigen Anwendung vereint mit den Vorteilen einer plattformunabhängigen Anwendung. Es wird also eine Webanwendung mit einer nativen Anwendung vermischt. Mit diesem Ansatz können die zuvor genannten Einschränkungen stark reduziert werden. So können sowohl plattformspezifische Hard- und Software Funktionen genutzt werden, als auch die Lauffähigkeit auf unterschiedlichen Geräten und Betriebssystemen mit geringem Aufwand umgesetzt werden. Normalerweise werden für die Umsetzung Hybrider-Apps Frameworks wie Cordova, Ionic oder PhoneGap als Wrapper verwendet. Diese blenden die Bedienelemente des Browsers aus, sodass der Nutzer der App die User Experience eines Anwendungsprogramms

bekommt (Brockschmidt 2015). Die Vorteile einer hybriden App im Gegensatz zu einer Webanwendung werden ersichtlich, wenn die Rechte und Möglichkeiten miteinander verglichen werden: Tabelle 2.1. Umklammerte Kreuze entsprechen einer geringen Browserunterstützung. Dies sind nur

Zugriff/Möglichkeit	Webanwendung	Hybrid-App
Kontakte		X
Kamera	X	X
Standort	X	X
Bewegungssensoren	X	X
Dateien	X	X
Installierbarkeit	X	X
Batterie Status		X
Network Information	(X)	X
Splash Screen	(X)	X

Tabelle 2.1.: Vergleich von Webanwendungen und Hybrid-Apps am Beispiel PhoneGap (*Plugin APIs* o.D., *Can I use* o.D.)

einige Beispiele. Vor allem mit Blick auf das Marketing bietet eine Hybrid-App mehrere Möglichkeiten den Kunden zu werben und/oder zu binden. Zusätzlich bietet diese eine große Reichweite mit vergleichbar geringem Aufwand.

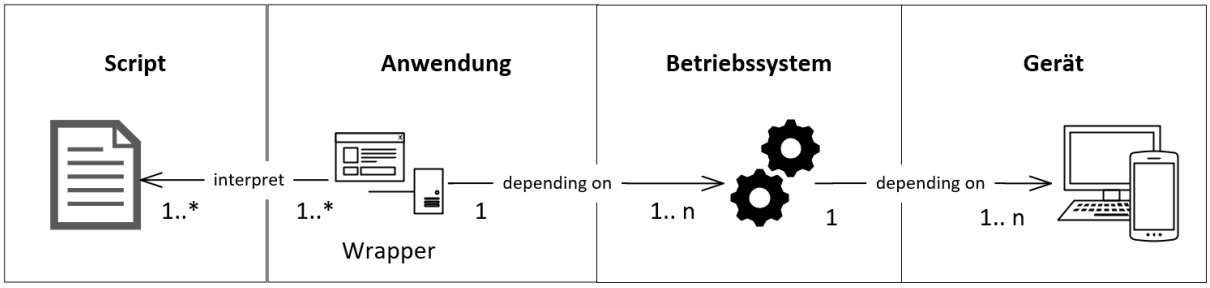

Abbildung 2.3.: Plattformabhängigkeiten - Lösung Hybrid-App

2.3. Cross-Plattform-Apps

Cross-Plattform-Apps verfolgen im Unterschied zu den Web- und Hybrid-Apps einen von Grund auf anderen Ansatz. In diesem Fall wird der Anwendungscode nämlich via Bridge in die plattformspezifischen UI-Elemente/API's übersetzt. Wohingegen der hybride Ansatz seinen Anwendungscode in einer Web-View anzeigt und somit die einzelnen UI-Elemente lediglich nachbildet. Dies führt zu Einbußen bei der Performance und des gesamten Erscheinungsbildes. In der Regel werden für die Entwicklung einer Cross-Plattform-App Programmiersprachen wie C#, C++ oder JavaScript benutzt.

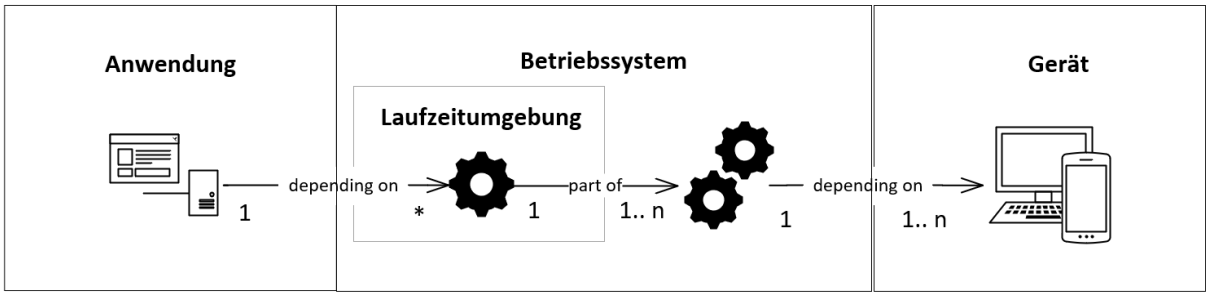

Abbildung 2.4.: Plattformabhängigkeiten - Lösung Cross-Plattform-App

Auch hier können zur Umsetzung Frameworks wie Xamarin oder React Native eingesetzt werden. Um im späteren Verlauf dieser Arbeit einen Vergleich ziehen zu können, folgt nun ein kleines Beispiel mit React Native:

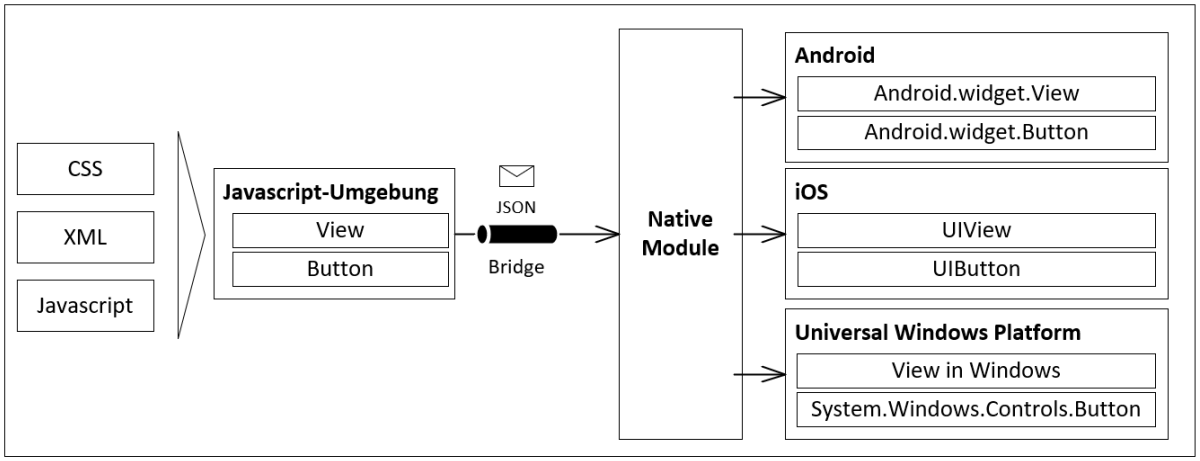

Abbildung 2.5.: Beispiel - React Native

Wie bereits erwähnt ist React Native erst einmal nicht mehr als ein Framework für JavaScript. Die Struktur und das Aussehen der Benutzeroberfläche werden mit einer XML- und CSS-artigen Sprache beschrieben. Diese dienen als plattformübergreifende Abstraktion. Das in JavaScript geschriebene Anwendungsprogramm hat zur Laufzeit der App den vollständigen Zugriff auf die einzelnen nativen UI-Elementen/API's der entsprechenden Plattform. Dies funktioniert über eine Laufzeitumgebung für JavaScript auf der jeweiligen Plattform und ist somit die Voraussetzung für die Funktionalität des Anwendungsprogramms. Der JavaScript Code selbst wird in einem eigenen Thread ausgeführt. Nachrichten, welche über die Bridge laufen, werden im JSON-Format asynchron an oder von dem nativen Modul kommuniziert. Sie werden asynchron übermittelt, um eine gegenseitige Blockade der einzelnen Threads zu verhindern. Das native Modul selbst ordnet die Anweisungen, die über die Bridge kommuniziert werden, dem zuständigem nativen Modul zu. Während der Entwicklung sorgt der sogenannte Packager dafür, dass der JavaScript Code bei Änderungen aktualisiert wird.

Dies erspart die Kompilierzeit, welche in der nativen Anwendungsentwicklung anfallen würde (Behrends 2018). Diese Form der plattformunabhängigen Programmierung hat durch ihre Komplexität aber auch Nachteile. Oft müssen gewisse Voraussetzungen vorliegen und erfüllt sein. Als Beispiel benötigt React Native die Laufzeitumgebung für JavaScript auf der entsprechenden Plattform und Xamarin eine Apple-Hardware zum Entwickeln von iOS Anwendungen (*Erste Schritte mit Xamarin.iOS* o.D.).

2.4. Problematiken

Einige Problematiken insbesondere in Bezug auf die Formen der plattformunabhängigen Programmierung wurden in dieser Arbeit bereits erläutert. Genauso wichtig ist es, die grundlegenden Problematiken der plattformunabhängigen Programmierung zu kennen. Gerade in der modernen Softwareentwicklung werden hohe Ansprüche an ein Anwendungsprogramm gestellt. Einige Problematiken sollten bereits im Entwurf bedacht werden, sodass es zu keinen Problemen in der Programmierphase kommt.

Fehlende native Funktionalitäten: Einige Funktionalitäten und Schnittstellen sind von der Plattform abhängig. Dies hat zur Folge, dass sie in der plattformunabhängigen Programmierung nicht zur Verfügung stehen.

Eingeschränkte Wiederverwendbarkeit des Anwendungscodes: Frameworks benutzen oft ihre eigene Sammlung an Befehlen. So kann es bei einem Umstieg des Frameworks dazu kommen, dass einige Teile des geschriebenen Codes nicht weiterverwendet werden können.

Integrationsprobleme: Auf Grund der hardwarefernen Programmierung kann es schwer werden lokale Einstellungen zu speichern oder Notifikationen zu handhaben.

Update Problematik: Vor allem mit dem Blick auf mobile Anwendungen werden des Öfteren Features und Funktionen für das entsprechende Betriebssystem zur Verfügung gestellt. In der nativen Anwendungsentwicklung können diese direkt genutzt werden. Bei der plattformunabhängigen Programmierung muss gewartet werden, bis diese in dem entsprechenden Framework zur Verfügung gestellt werden.

Performanceeinbußen: Abgesehen von den Performance-Werten der einzelnen Formen müssen Entwickler während der Programmierung vieles beachten. Ein Computer hat viel mehr Potenzial als ein Smartphone. Zu betrachten ist hier zum Beispiel der Arbeitsspeicher, die Prozessorleistung oder die Größe an persistentem Speicher. Ein Anwendungsprogramm, welches auf einem Computer mit einer geringen Auslastung ausgeführt werden kann, könnte ein Smartphone schon an

seine Grenzen bringen. Andersherum kann ein Anwendungsprogramm, dessen Performance extra für Smartphones optimiert wurde, nicht das volle Potenzial auf einem Computer zeigen.

Eine Benutzeroberfläche für alle Geräte: Was vom Aufwand her als großer Vorteil gesehen wird, stellt aber einen Nachteil bei der User Experience dar. Es kann Funktionen geben, welche zwar auf einem Computer zur Verfügung stehen sollen, beim Aufrufen über ein Smartphone jedoch nicht. Dazu kommt, dass die Vorteile der Plattformen nicht optimal genutzt werden können. Beispielsweise kann ein Seitenwechsel mit einer Swipe-Bewegung nach links oder rechts auf einem Smartphone von Vorteil sein. Am Computer wirkt sich eine solche Geste hingegen negativ aus, da diese mit der Maus als nicht natürlich wahrgenommen wird.

3. Progressive Web Apps

Der Begriff „progressive“ wird in die deutsche Sprache mit „fortschrittlich“, und „web“ mit „Netz“ übersetzt. Der Begriff App ist eine Abkürzung des Wortes Applikation, welches für ein Anwendungsprogramm steht. Sinngemäß übersetzt ist eine Progressive Web App also ein fortschrittliches Anwendungsprogramm, welches über das Netz zur Ausführung kommt. Um eine Webanwendung als Progressive Web App (PWA) zu bezeichnen müssen einige Voraussetzungen erfüllt und bestimmte Merkmale gegeben sein. Diese Voraussetzungen sind von Browser zu Browser unterschiedlich. Ein gutes Leitbild gibt jedoch Alex Russell, ein Projektmanager bei Google, welcher 2015 in einem Blogpost die Progressive Web Apps ins Leben gerufen hat. Die Idee hinter einer PWA ist es auf einer Webseite oder genauer gesagt, in einer Registerkarte des Browsers bei Interesse des Benutzers die Möglichkeit zu geben diese Anwendung als Verknüpfung über einen Button zur Programmliste hinzuzufügen.

Mit der Umsetzung einer Progressiv Web App entstehen einige Vorteile. So ist allein das Laden einer PWA um ein vielfaches schneller, als die Ladezeiten einer einfachen Webanwendung (Taylor 2019). Auch die Wartezeiten beim Installieren fallen nahezu weg, da lediglich eine Verknüpfung zu der entsprechenden Website erstellt werden muss. Das Laden der Assets und Seiten geschieht im Hintergrund über den Service Worker.

Bezüglich der Installation hat der Betreiber die Möglichkeit direkt auf der Website die Installation der PWA anzubieten. Eine Weiterleitung zu einem entsprechenden App-Store ist nicht mehr nötig. Dies bedeutet nicht, dass eine PWA nicht in den jeweiligen App-Stores der Plattform angeboten werden kann. PWAs können in App-Stores, wie dem iOS App Store, Google-Play-Store oder Microsoft Store (parallel) zur Verfügung gestellt werden (Petereit 2020b). Dafür ist lediglich eine Codierung nötig.

Auch beim Updaten der App wird dem Entwickler Arbeit abgenommen: Denn durch die Ressourcenverwaltung des Service Worker können Änderungen an der App direkt übernommen werden und müssen nicht erst im jeweiligen App-Store hochgeladen werden. Ein weiterer großer Vorteil gegenüber nativen Apps ist die Ausführbarkeit der Anwendung. Eine native Anwendung kann in der Regel gar nicht erst installiert werden, wenn eine Plattform gewisse Funktionalitäten nicht besitzt. Eine PWA kann hingegen auch ohne diese Funktionalitäten ausgeführt werden, denn diese bleiben dem Benutzer in einem solchen Fall enthalten. Auch die Möglichkeit von Push-Notifikationen ist bei PWAs gegeben. Entsprechende APIs stehen zwar auch den normalen Webanwendungen zur Verfügung, kommen jedoch so dem App-Gedanken näher. Auch bei der Konnektivität ist eine PWA weit vorne positioniert. Denn eine Offline-Fähigkeit der Anwendung ist ein Merkmal einer PWA. Gerade in der heutigen Zeit, wo Webseitenaufrufe mobiler gegenüber stationärer Geräte überwiegen, sollte

eine Anwendung eine gewisse Toleranz bei Verbindungsausfällen und geringer Übertragungsraten haben. Um die Vorzüge einer PWA nutzen zu können, muss der genutzte Browser mehrere Funktionen unterstützen, wie in Tabelle 3.1 zu sehen ist.

Der Trend von PWAs geht aufwärts. Mobile Sitzungen auf PWAs steigen im Durchschnitt um 80% (Uzun 2018, *PWA Stats* o.D.). Das sehen auch große Unternehmen wie Netflix, Google, Twitter, Instagram und Telegram, die selber PWAs entwickeln und/oder bereits publiziert haben (*Der PWA Showroom* o.D., *Netflix arbeitet an PWA für Windows 10* 2020).

	Browser							
Funktionen		Firefox	Chrome	Opera	Android B.	Safari	Samsung	Edge
	Web Manifest	(X)	X		X	(X)		X
	Service Worker	X	X	X		X	X	X
	InstallEvent API	X	X	X	X	X	X	X
	Push API	X	X	X		(X)	X	X
	Fetch API	X	X	X	X		X	X
	Cache API	X	X	X	X	X	X	X

Tabelle 3.1.: Vergleich der Browserunterstützung von Webschnittstellen und Webtechnologien(*Can I use* o.D.)

3.1. Eigenschaften und Merkmale

Die Eigenschaften und Merkmale machen eine Webanwendung zu einer PWA. Es ist wichtig alle Merkmale zu erfüllen, andernfalls kann es zu Problemen mit der Auffindbarkeit oder Anerkennung einer PWA kommen. Folgende Merkmale ergeben sich aus den Spezifikationen von Chrome (LePage o.D.), Firefox (*Progressive Web-Apps* 2019) und dem Blogbeitrag von Alex Russell (Russell 2015):

App-ähnlich: Orientiert an der Struktur einer nativen App, soll auch die PWA entwickelt werden. So soll eine PWA vor der Ausführung die nötigen Dateien herunterladen. „only load once“ ist hier das Paradigma. In diesem Zustand soll die PWA selbstständig auf dem Client ausgeführt werden können, ohne zur Ausführung erneut wichtige Dateien herunterladen zu müssen. Single-Page-Webanwendung setzen eine solche Anwendungsarchitektur um und sind somit optimal für die Umsetzung einer PWA geeignet. Zudem sollen Funktionalitäten, die von nativen Anwendungsprogrammen bekannt sind, über moderne Webschnittstellen, in einer PWA umgesetzt werden. Funktionalitäten wie Push-Benachrichtigungen, Sensor-, Standort- oder Kamera-Zugriff sollen dem Benutzer das Gefühl eines nativen Anwendungsprogramms geben. Des Weiteren soll die Benutzeroberfläche einer PWA dem Vorbild eines nativen Anwendungsprogramms folgen. Bestandteil davon ist das

Verwenden von Animationen und Effekten sowie der Aufbau der Grundelemente einer Anwendung. Das Konzept der App Shell ist hierbei der Inbegriff eines solchen Vorbildes. Nach diesem Konzept werden die Grundelemente, die oft bis immer dargestellt werden müssen, offline bestehen bleiben. Grundelemente können hierbei Steuerelemente, Kopf- und Fußzeilen sowie Seitenheader sein. Beispiele eines Layouts mit dem Konzept der App Shell könnten wie in Abbildung 3.1 demonstriert aussehen.

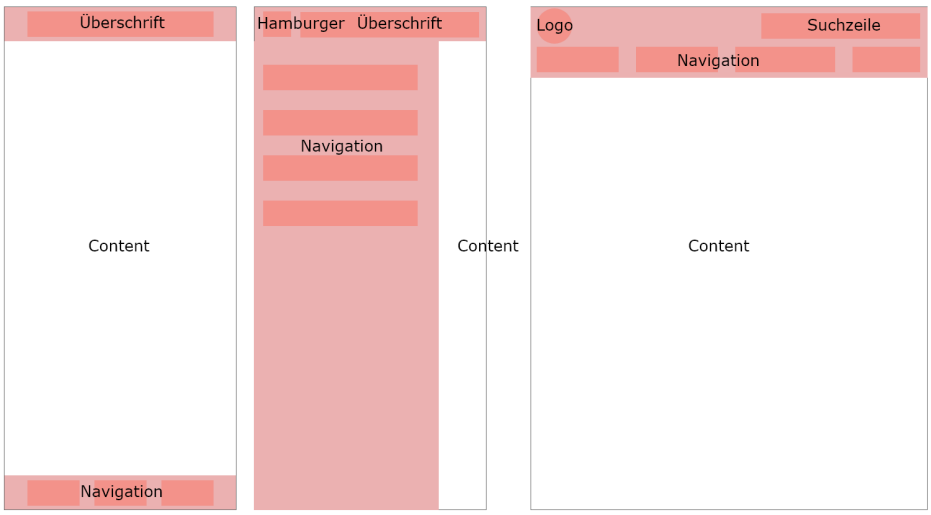

Abbildung 3.1.: App-Shell

Die rot eingezeichneten Bereiche in Abbildung 3.1 würden demnach in den Cache geladen werden. Durch das persistente Speichern der Grundelemente der Benutzeroberfläche, wird nicht nur Datenvolumen gespart, sondern auch eine Offline-Fähigkeit hergestellt.

Verbindungsunabhängigkeit: Angeknüpft an der App-Ähnlichkeit ist eine Offline-Fähigkeit eine Voraussetzung einer PWA, denn ohne diese Offline-Fähigkeit könnte die Anwendung ohne eine Internetverbindung nicht ausgeführt werden. Dies entspreche nicht der Vorstellung eines selbstständigen Webclients. So ist eine Internetverbindung ja auch keine Voraussetzung für ein natives Anwendungsprogramm. Mit dem Anwenden des Konzeptes der App-Shell kann das Merkmal „verbindungs-unabhängig“ erfüllt werden. Die Idee kann an diesem Punkt jedoch noch weiter ausgeführt werden. So gibt es Konzepte und Ansätze, um eine Anwendung mit der Basis einer kompletten Offline-Fähigkeit zu entwerfen - inklusive aller Datenbankzugriffe. Anschließend wird eine gewisse Online-Fähigkeit hinzugefügt. Grundlegend wird die Verbindungsunabhängigkeit mit Hilfe eines Service Worker umgesetzt. Der Service Worker ist ein Hintergrundprozess im Browser, vergleichbar mit einem Thread, welcher unabhängig vom Server agieren kann. Dieser wird in Webanwendungen als zentraler Proxy genutzt und kann Daten synchronisieren und zum Beispiel Push-Benachrichtigungen entgegennehmen und verarbeiten. Das Verwenden eines Service Workers setzt

somit die Offline-Fähigkeit einer Webanwendung um.

Aktualität des Service Worker: Natürlich soll die Version einer PWA immer auf dem neusten Stand sein. Mit dem Umsetzen einer Verbindungsunabhängigkeit kommt somit auch die Frage nach der Konsistenz zwischen den bestehenden Website-Versionen auf. Bei dem Registrieren eines Service Workers wird dieser für den Fall, dass bereits ein Service Worker registriert ist eingereiht.

Sicherheit: Die Ansprüche der Sicherheit an eine PWA sind grundlegend dieselben, die auch an eine Webanwendung gestellt werden. Ein größerer Aspekt im Gegensatz zu einer „einfachen“ Webanwendung ist hingegen das Verwenden von HTTPS. Mit der Nutzung moderner Webschnittstellen und der Übertragung der Anwenderdaten ist von einer HTTPS-Verbindung nicht abzusehen. HTTPS ist ein Sicherheitsprotokoll, das die Daten verschlüsselt, die zwischen dem Server und dem Client übertragen werden. Zusätzlich wird mit HTTPS eine Authentifizierung umgesetzt, in der die Identität des Servers, aber auch des Webclients überprüft werden kann. Mit dieser Methode wird das Abfangen und Lesen von Daten, die sich in der Übertragung befinden verhindert. Angriffe wie Man-in-the-Middle oder Phishing sind mit einer HTTPS-Verbindung nicht möglich.

Responsive Web Design: Das Erstellen eines Responsive Web Designs ermöglicht es, eine Benutzeroberfläche für mehrere Plattformen zu erstellen. Hierbei liegt der Fokus vor allem auf der Anzeige der Seitenelemente innerhalb des verfügbaren Größenbereiches. In dem Entwurf einer Benutzeroberfläche müssen mehrere Aspekte beachtet werden:

- Größe des Browserfensters
- Größe des Bildschirms eines Gerätes
- Auflösung des Bildschirms
- Orientierung des Formates: Quer- und Hochformat

Unter Beachtung der aufgezählten Aspekte sollen die Elemente mit Hilfe aktueller Webstandards wie HTML5, CSS3 und JavaScript positioniert und skaliert werden. Ausgangsbasis ist hierbei ein einziger Anwendungscode. Abgesehen von der Anzeige der Seitenelemente müssen auch die jeweiligen Eingabemethoden des Gerätes in die Anwendung einfließen. Eingabemethoden sind hierbei der Mausklick am Computer oder das Tippen mit dem Finger auf einem Touchscreen.

Auffindbarkeit: Damit eine PWA von einer Webanwendung unterschieden werden kann, muss in einer PWA eine Manifest-Datei erstellt werden. Diese beinhaltet alle nötigen Metadaten der Webanwendung und wird im JSON-Format angegeben.

Installierbarkeit: Da eine PWA den Gedanken verfolgt nativen Anwendungsprogrammen näher zu

kommen, ist auch die Installierbarkeit ein Merkmal einer PWA. Mit dem Erstellen des Manifestes ist eine PWA von einer nativen App zumindest auf mobilen Geräten kaum noch zu unterscheiden.

Verlinkbarkeit: Anders als bei nativen Anwendungsprogrammen benötigt eine PWA keinen App-Store oder Ähnliches um über einen Link aufgerufen zu werden. Es ist eine direkte Verlinkung der PWA möglich.

Nutzerbindung: Mit Hilfe der Notification API des Service Worker können Push-Benachrichtigungen an den Benutzer gesendet werden. Push-Benachrichtigungen können über den Service Worker auch dann entgegen genommen werden, wenn das Anwendungsprogramm selbst gar nicht geöffnet ist. Ziel ist es den Benutzer mit Erinnerungen, Aufforderungen oder Impressionen zurück zum Anwendungsprogramm zu führen.

3.2. Manifest

Das Manifest ist ein wichtiger Bestandteil einer PWA und setzt mehrere Merkmale um. Soll einer Webanwendung ein Manifest hinzugefügt werden, muss zunächst im HTML-Teil eine Referenzierung zu der Manifest-Datei hinzugefügt werden.

```
1 <link rel="manifest" href="/manifest.webmanifest">
```

Listing 3.1: Manifest - Tag

Innerhalb der Manifest-Datei sollten folgende Eigenschaften definiert werden (*Web App Manifest* o.D.):

Name: Um der PWA einen Namen zu geben, welcher später in der Programmliste oder auf dem Startbildschirm angezeigt werden soll, muss die Eigenschaft „name“ angegeben werden. Da es bei langen Namen einer PWA zu Platzproblemen kommen kann, sollte ebenfalls auch die Eigenschaft „short_name“ definiert werden. Falls der Name aufgrund seiner Länge abgeschnitten werden würde, wird die Eigenschaft „short_name“ als Name der PWA verwendet.

App-Start: Mit dem angeben der „start_url“ wird der Bereich einer Webanwendung definiert, welcher als Startseite der PWA benutzt werden soll. Beim Klicken auf die Anwendung würde bei einer „start_url“ mit „/“ beispielsweise die Startseite der Webanwendung geöffnet werden.

Anzeigemodus: Es gibt verschiedene Arten eine Webanwendung anzuzeigen. Diese Eigenschaft wird mit der Erkennung „display“ definiert. Der Anzeigemodus „fullscreen“ zeigt die Webanwendung auf dem gesamten Bildschirm an. Steuerelemente des Browsers und des Betriebssystems werden ausgeblendet. Bei fehlender Unterstützung des Browsers wird der Anzeigemodus auf „standalone“

zurückfallen. Dieser blendet alle Browser-Elemente aus und wird wie ein natives Anwendungsprogramm angezeigt, wodurch dieser sich perfekt für PWAs eignet. Beim Anzeigemodus „minimal-ui“ wird die Anwendung ähnlich wie bei einem nativen Anwendungsprogramm angezeigt. Der Unterschied zu „standalone“ ist hierbei, dass eine minimale Anzahl an UI-Elementen zur Steuerung/Navigation des Browsers hinzugefügt werden. „minimal-ui“ ist auch der Fallback-Anzeigemodus von „standalone“. Mit dem Modus „browser“ wird die gesamte Anwendung im Browser dargestellt, wie es bei einer „einfachen“ Webanwendung auch der Fall ist. Sollte also auch die „minimal-ui“ als Anzeigemodus fehlschlagen wird stattdessen „browser“ verwendet.

Farben: Die Eigenschaft „background_color“ beschreibt die Hintergrundfarbe der PWA und wird unter anderem repräsentativ als eingefärbter Sichtbereich der Webanwendung während des Ladevorgangs angezeigt. Durch das Angeben der „theme_color“ hat der Browser die Möglichkeit sich optisch an die Webanwendung anzupassen.

Beschreibung: Mit der Eigenschaft „description“ kann eine Beschreibung der Webanwendung hinzugefügt werden.

Icons: Das Angeben der Icons wird für die Anzeige der Anwendung auf dem Startbildschirm oder in der Programmliste gefordert. Es ist darauf zu achten, dass das Icon in mehreren Formaten zur Verfügung steht, damit es auf den verschiedensten Bildschirmauflösungen scharf angezeigt werden kann. Die Formate unterscheiden sich hierbei von Browser und Gerät. Ein Icon wird im Manifest mit

- „src“, die URL zum entsprechenden Icon
- „size“, die Abmessung des Icons z.B „1024x1024“
- „type“, der MIME-Type des Icons

grundlegend angeben. Zusätzlich kann mit „plattform“ die Zielplattform und mit „purpose“ das Betriebssystem für die Verwendung des Icons definiert werden.

App-Bereiche: Mit der Eigenschaft „scope“ werden Bereiche der Webanwendung definiert, die in der installierten Anwendung ausgeführt werden dürfen. Der Scope wird mit einer URL angegeben. Alle Pfade innerhalb dieser URL stehen in der Benutzung der installierten Anwendung zur Verfügung.

Die erläuterten Eigenschaften sind die wichtigsten Eigenschaften zum Erstellen eines Manifestes einer PWA. Weitere sind in den Spezifikationen unter *Web App Manifest* o.D. zu finden, um den Rahmen dieser Arbeit nicht zu sprengen, sollen sie an dieser Stelle allerdings nicht thematisiert werden. Abschließend ist zu bemerken, dass es abhängig vom Browser und Betriebssystem stets

Ausnahmen gibt. So müssen für iOS und Internet Explorer zusätzliche Tags im HTML-Teil hinzugefügt werden. Bezüglich der Ausnahmen macht es Sinn einen Generator zu verwenden, um das Manifest zu erstellen und nachträglich eventuelle Anpassungen durchführen zu können. Vorausgesetzt, dass alle notwendigen Eigenschaften angegeben sind, sollte das Manifest mindestens folgende Angaben enthalten:

```
1  {
2    "name": "example",
3    "short_name": "example",
4    "theme_color": "#1976d2",
5    "background_color": "#fafafa",
6    "display": "standalone",
7    "scope": "./",
8    "start_url": "./",
9    "icons": [
10     {
11       "src": "assets/icons/icon-72x72.png",
12       "sizes": "72x72",
13       "type": "image/png",
14       "purpose": "maskable any"
15     },
16   ...
17   ]
18 }
```

Listing 3.2: Manifest-Beispiel

3.3. Netzwerk-Verhalten

Bei der Umsetzung einer Offline-Fähigkeit sollten mehrere Aspekte betrachtet werden. Die Fähigkeit die Webanwendung selbst im Offline-Zustand bereitzustellen und die Daten, welche dynamisch während der Laufzeit geladen werden müssen.

3.3.1. Caching-Strategien

Bei einer Caching-Strategie handelt es sich um ein definiertes Verhalten des Service Worker zum Bearbeiten eingehender Anfragen von Ressourcen und Daten. Je nach Caching-Strategie werden geforderte Ressourcen/Daten über den Cache selbst oder das Netzwerk bedient. Die Auswahl der richtigen Caching-Strategie ist wichtig für die Performance, Benutzererfahrung und die Aktualität der Daten, die dem Benutzer angezeigten werden.

Cache Only:

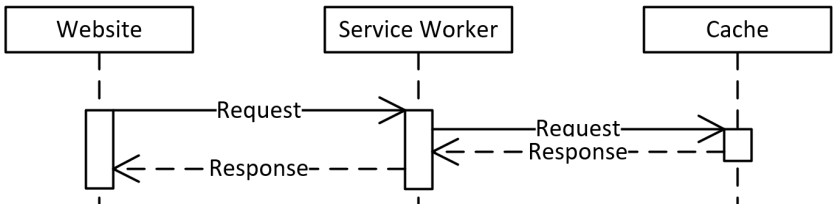

Abbildung 3.2.: Caching-Strategien - Cache Only

In dieser Caching-Strategie werden alle Anfragen ausschließlich an den Cache gestellt. Das Prinzip der App Shell wird mit dieser Strategie umgesetzt, um wichtige Ressourcen zu laden.

Network Only:

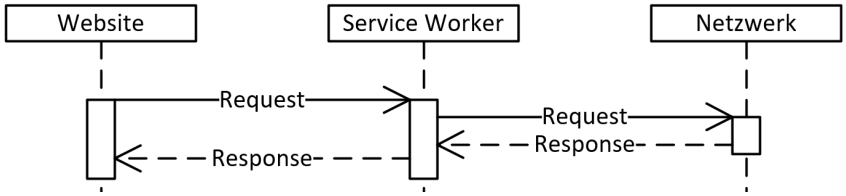

Abbildung 3.3.: Caching-Strategien - Network Only

Das Gegenstück dazu ist eine „Network Only“-Strategie. Diese findet jedoch nur selten Anwendung, da sie eine Netzwerkverbindung voraussetzt. Anfragen werden hier ausschließlich über das Netzwerk bedient. Ein Anwendungsbereich wäre beispielsweise die Zurverfügungstellung sensibler Live-Daten, da diese stets aktuell sein müssen. Grundsätzlich ist jedoch von dieser Strategie abzuraten.

Cache falling back to Network:

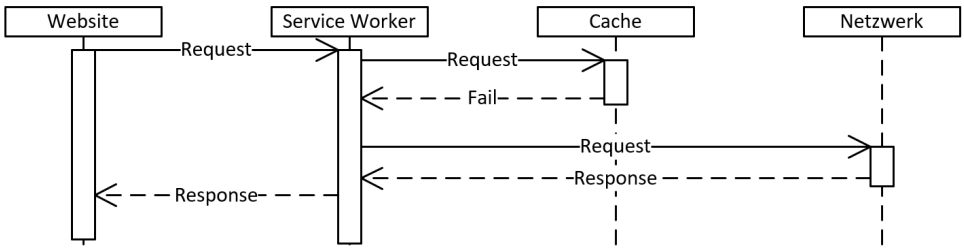

Abbildung 3.4.: Caching-Strategien - Cache falling back to Network

Für die Umsetzung von „Offline-first“ ist die Implementierung der Caching-Strategie „Cache falling back to Network“ ideal. Hierbei wird zunächst der Cache angefragt, sofern dieser die Anfrage nicht beantworten kann, wird eine Anfrage über das Netzwerk gestellt.

Network falling back to Cache:

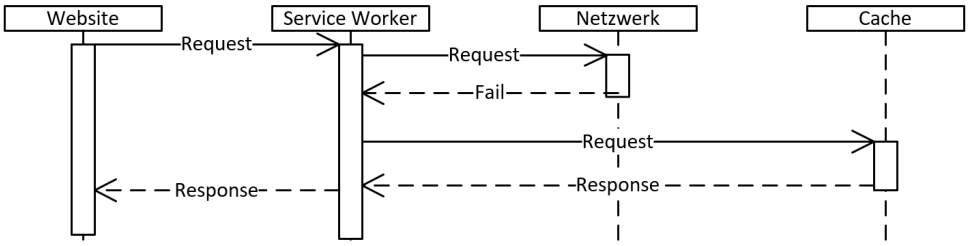

Abbildung 3.5.: Caching-Strategien - Network falling back to Cache

Innerhalb einer Anwendung wird des Öfteren dynamisch erzeugter Inhalt benutzt. Bei einer Offline-Unterstützung bieten sich entsprechend zwei Caching-Strategien optimal an: Zunächst wird eine Anfrage über das Netz gestellt. Kann diese nicht beantwortet werden, wird die Anfrage an den Cache weitergeleitet. Hierbei steht die Aktualität der Ressourcen im Vordergrund.

Cache then Network:

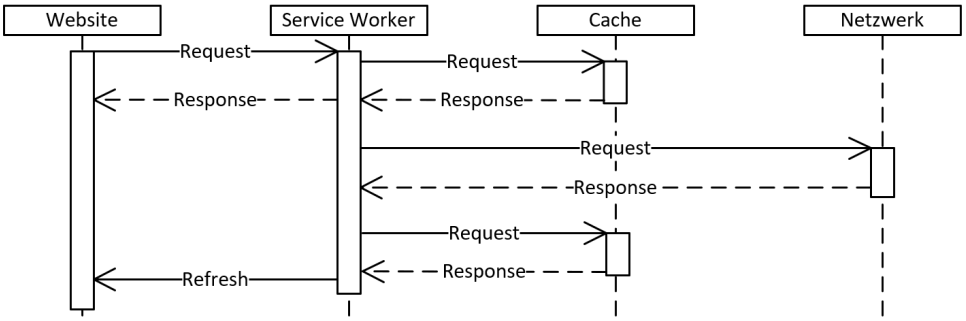

Abbildung 3.6.: Caching-Strategien - Cache then Network

Sofern die Aktualität zweitrangig ist, kann die Anfrage über den Cache gestellt und anschließend über das Netzwerk aktualisiert werden.

Cache and Network race:

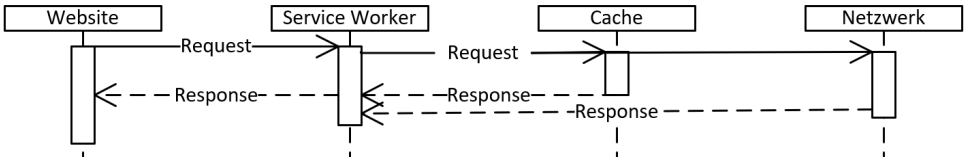

Abbildung 3.7.: Caching-Strategien - Cache and Network race

Beim Laden von Assets ist der Ursprung (Cache/Netzwerk) dieser in den meisten Fällen irrelevant. Ein wichtiger Aspekt kann hierbei jedoch die Geschwindigkeit der Verfügbarkeit sein. Mit der

Caching-Strategie „Cache and Network race“ wird parallel eine Anfrage an den Cache und das Netzwerk gestellt. Im Anschluss wird die schnellste Antwort verwendet.

Generic Fallback:

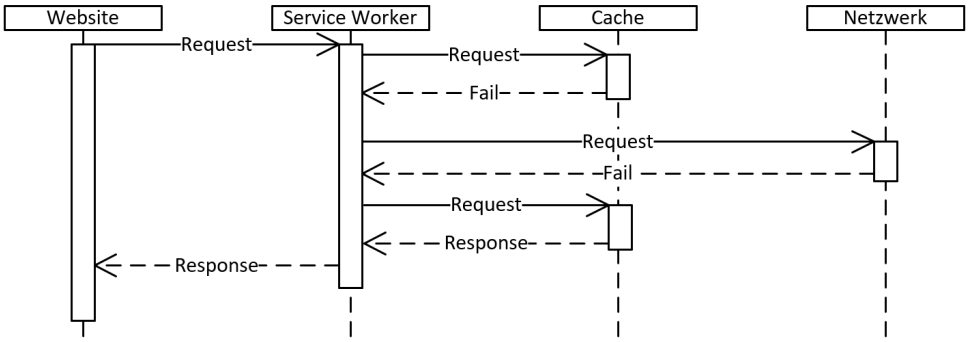

Abbildung 3.8.: Caching-Strategien - Generic Fallback

In manchen Fällen kommt es in einer Anwendung dazu, dass die Verfügbarkeit einer Ressource nicht garantiert ist. Beispielsweise das Profilbild eines Benutzers. Für einen solchen Fall gibt es die Caching-Strategie „Generic Fallback“. Kommt es zu keiner Antwort vom Cache oder Netzwerk wird die Anfrage generisch beantwortet. Im Falle eines nicht vorhandenen Profilbildes würde entsprechend eine vordefinierte Grafik angezeigt werden (Archibald 2014).

3.3.2. Service Worker

Aus dem bereits erläuterten Merkmal „Verbindungsunabhängigkeit“ geht hervor, dass der Service Worker die Offline-Fähigkeit der Webanwendung umsetzt. Der Service Worker selbst ist, wie auch der Shared Worker, eine besondere Art des Web Worker und entsprang ursprünglich der Single-Thread-Problematik. Diese Problematik entsteht im Browser, da sich JavaScript einige Ressourcen mit dem Render- und Painting-Prozess des Browsers teilt. Als Folge von großen Rechenzeiten oder Endlosschleifen im JavaScript-Code kommt die Webanwendung ins Stocken oder friert komplett ein. Das Verwenden eines Web Worker löst das Problem, da dieser in einem eigenen Thread ausgeführt wird. Der Zugriff auf den globalen Kontext wird mit einem eigenen isolierten globalen Kontext ersetzt, um Race Conditions zu verhindern. Auch der Zugriff auf das DOM des Elterndokuments wird blockiert, um Thread-Exception zu verhindern. Alternativ kann aber eine Schnittstelle zwischen dem UI-Thread und dem Web Worker benutzt werden. Über diese Schnittstelle können im Anschluss Stil- oder Strukturinformationen übermittelt werden. Der Web Worker hat lediglich Bezug auf eine Registerkarte. Dies kann zum Problem werden, wenn Ressourcen oder Informationen untereinander geteilt werden sollen. An diesem Punkt kommt der Shared Worker zum Einsatz. Dieser kann auf alle geöffneten Browser-Kontexte unterhalb der Origin zugreifen, unterscheidet sich aber ansonsten nicht von einem Web Worker. Die Instanziierung eines Web oder Shared Workers geschieht über

die Website selbst. Folglich hängt sein Lebenszyklus vom Lebenszyklus der Webanwendung ab. Der Service Worker ist hingegen unabhängig von der Webanwendung, denn dieser wird im Browser registriert und ausgeführt. Durch diese Eigenschaft ist der Service Worker optimal für die Umsetzung eines zentralen Proxys geeignet (Liebel 2018a).

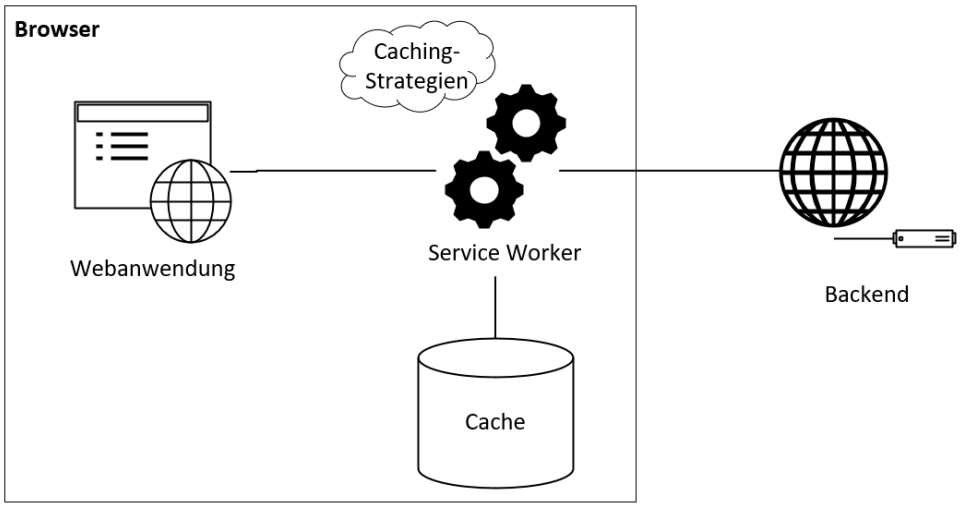

Abbildung 3.9.: Service Worker

Wie in Abbildung 3.9 zu sehen, ruft ein Benutzer eine PWA auf, registriert die Webanwendung den Service Worker beim Browser. Mit der Registrierung und Aufleben des Service Workers wird über den Browser eine Anfrage der Installation gestellt. Unabhängig davon kann der Service Worker nun mit der Webanwendung kommunizieren. Stellt diese eine Ressourcen-Anfrage, wird diese an den Service Worker gestellt. Der Service Worker bedient die Anfrage je nach Implementierung der Caching-Strategie über den Cache oder das Netzwerk. Das Cachen und Anfragen der Ressourcen wird im Service Worker mit Hilfe moderner Webschnittstellen realisiert, welche im nächsten Unterabschnitt 3.3.4 genauer beschrieben werden. Der Service Worker durchläuft von der Registrierung bis zum Ausloggen einen Lebenszyklus. Während seiner Lebenszyklen durchläuft der Service Worker folgende Status:

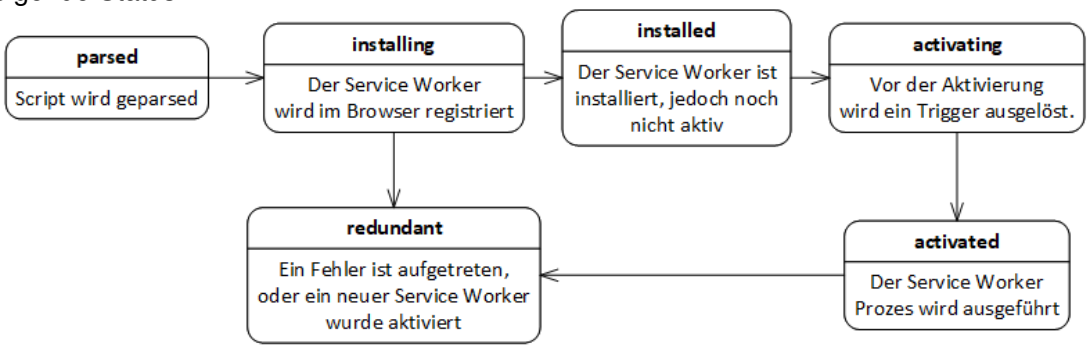

Abbildung 3.10.: Service Worker - Lebenszyklus

Während des Status „installing“ bis „activated“ können Schnittstellen des Service Workers genutzt werden. So kann das Service Worker Skript mit der „update“-Funktion aktualisiert oder mit der „unregister“-Funktion ausgeloggt werden. Der Status des Service Workers kann mit den Eigenschaften „installing, waiting, active“ abgefragt werden. Die Kommunikation zwischen der Website und dem Service Worker wird mit der „postMessage“-Funktion umgesetzt. Diese Funktion wird jeweils über den Status des Service Workers ausgeführt. (Russell u. a. 2019, Liebel 2018b)

In folgendem Beispiel soll der Service Worker registriert werden und Anfragen mit der Caching-Strategie „Cache falling back to Network“ bedienen.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Example</title>
7  </head>
8  <body>
9      <h1>Service Worker</h1>
10     <script src="client.js"></script>
11 </body>
12 </html>
```

Listing 3.3: index.html - Benutzeroberfläche

In diesem Beispiel soll lediglich eine Überschrift als Benutzeroberfläche ausreichen. Wichtig ist hierbei nur das Einbinden des Scriptes in Zeile zehn.

```
1  if('serviceWorker' in navigator) {
2      register().catch(err => console.err(err));
3  }
```

Listing 3.4: client.js - Browserunterstützung prüfen

In diesem Teil wird die Browserunterstützung vom Service Worker geprüft. Sollte die Registrierung des Service Workers fehlschlagen, würde eine Fehlermeldung in der Konsole angezeigt werden. „navigator“ ist der Kontext der Webanwendung, in dem nach der Registrierung des Service Workers angefragt wird. Bei der Funktion „register()“ handelt es sich um eine im nächsten Schritt definierte Funktion.

```
1  async function register() {
2      navigator.serviceWorker.register('/sw.js', {
3          scope: '/'
4      });
5  }
```

Listing 3.5: client.js - Service Worker registrieren prüfen

Die Registrierung des Service Workers ist asynchron, so wird der Benutzer in seiner Anwendung nicht behindert. Die Registrierung des Service Worker „sw.js“ geschieht hierbei wieder über den Kontext „navigator“. Über die Einstellung „scope“ wird der Bereich, in dem der Service Worker tätig werden soll, definiert.

```
1  const ressourcen = [
2    'index.html',
3    'client.js'
4  ]
5  self.addEventListener('install', event => {
6    event.waitUntil(
7      caches.open('cache').then(cache => {
8        cache.addAll(ressourcen);
9      })
10   )
11 });
```

Listing 3.6: sw.js - Service Worker installieren und cachen

Im Service Worker wird zunächst ein Array definiert, welcher die zu cachenden Ressourcen benennt. Über ein Eventlistener wird das Cachen der Ressourcen beim Installieren des Service Workers ausgelöst. Da dieser Vorgang etwas dauern kann, können währenddessen keine Anfragen bedient werden. Mit Zeile sechs wird dem Browser deswegen kommuniziert, dass das Event noch nicht verarbeitet ist und bis dahin gewartet werden muss. Der Cache wird mit dem Befehl „caches.open()“ mit einem entsprechenden statischen Namen geöffnet/initialisiert. In Zeile acht werden im Anschluss alle genannten Ressourcen über die Cache API gecached.

```
1  self.addEventListener('fetch', event => {
2    event.respondWith(
3      caches.match(event.request).then(response => {
4        return response || fetch(event.request);
5      })
6    )
7  });
```

Listing 3.7: sw.js - Service Worker Ressourcen bereitstellen

Für das Laden der Ressourcen werden die Fetch-Events abgefangen und mit der Caching-Strategie „Cache falling back to Network“ bedient. Hierbei wird mit Hilfe der Cache API der Request über den Cache beantwortet. Ist die angefragte Ressource nicht im Cache vorhanden, wird sie normalerweise „undefined“ zurückgegeben. In dieser Implementierung wird stattdessen eine Netzanfrage über die Fetch API ausgeführt. Das Ergebnis wird mit der Methode der Zeile zwei als Response zurückgegeben.

3.3.3. Eine Sichere Verbindung - Hypertext Transfer Protocol Secure

Eine HTTPS-Verbindung ist für eine PWA unabdingbar, denn ohne eine solche Verbindung ist es nicht möglich einen Service Worker zu registrieren. Da dieser Zugriff auf alle Ressourcen sowie auf ein- und ausgehende Daten hat, ist das Verbot begründet. Die einzige Ausnahme für das Registrieren eines Service Workers ohne HTTPS-Verbindung ist die Ausführung auf dem eigenen Computer. Bei HTTPS werden die Datenübertragungen auf der Anwendungsschicht über das HTTP umgesetzt und in der Transportschicht mit der Transport Layer Security (TLS) verschlüsselt übertragen.

Bei HTTP handelt es sich um ein zustandsloses Protokoll zum Übertragen von Daten in einem Netzwerk (Fielding u. a. 1999). Die Daten sollen also die Funktionalitäten für die Anwendung zur Verfügung stellen. Eine HTTP-Netzanfrage besteht immer aus einem Request und einer Response. Der Request leitet hierbei die Netzanfrage ein. In dieser stellt der Client eine zielgerichtete Anfrage einer Ressource (über die URL) an den Webserver. Der Webserver antwortet folglich mit der Response. Der Request sowie die Response wird in Form einer Nachricht mit Header und Body kommuniziert. Der Header der Nachricht enthält jegliche Metadaten des Bodys und der Ressource. Im Body enthalten ist die Nutzlast (Payload) in dem definierten Format, Codierung des Headers. Für das Abrufen oder Manipulieren der Daten stehen mehrere Methoden für die Umsetzung zur Verfügung.

HTTP-Methode	**Beschreibung**
GET	Die GET-Methode realisiert das Abfragen von Daten, die über eine Request-URI identifiziert werden.
HEAD	Die HEAD-Methode realisiert ebenfalls eine solche Anfrage, jedoch wird hier nur der Statuscode und der Header zurückgegeben.
Beide Methoden werden als „Safe“ betitelt, da sie keine Änderungen der Ressourcen auf dem Server vornehmen. Um Daten zu manipulieren, können die folgenden drei Methoden benutzt werden.	
PUT	Für das Ersetzen von Ressourcen steht die PUT-Methode zur Verfügung.
DELETE	Ressourcen können mit der DELETE-Methode entfernt werden.
POST	Ressourcen können mit der POST-Methode hinzugefügt werden.
TRACE	Gibt die Anfrage an den Client so zurück, wie diese an den Server gestellt wurde.
CONNECT	Die CONNECT-Methode realisiert einen SSL-Tunnel, indem diese Methode von dem Proxy-Server implementiert wird.
OPTIONS	Die OPTIONS-Methode listet unterstützte Methoden und Merkmale des Servers.

Tabelle 3.2.: HTTP-Methoden

Als Antwort eines Requests wird über die Response immer ein Statuscode zurückgegeben. Eine Tabelle der Statuscodes ist in der Spezifikation unter Goland u. a. 1999 zu finden.

HTTPS stellt mit Hilfe von der TLS eine sichere Internetverbindung her. Sie wird mit mehreren Verfahren umgesetzt: Zum einen findet eine Authentifizierung über das Vergeben von digitalen TLS-Zertifikaten statt. Ein digitales TLS-Zertifikat wird bei einer Zertifizierungsstelle mit Angabe des Domainnamens ausgestellt. Diese führt eine Domainvalidierung durch, in der die Existenz, die Identität des Betreibers und die Domänenzugehörigkeit geprüft wird. Bei einem TLS-Zertifikat gilt jedoch, dass dieses nur als sichere Verbindung akzeptiert wird, wenn der Browser und/oder das Betriebssystem die Zertifizierungsstelle als vertrauenswürdig eingestuft hat. Eine erweiterte Sicherheit des TLS-Zertifikats bietet zum anderen das Extended-Validation-Zertifikat. Um dieses Zertifikat zu erhalten müssen bei einer vorab geprüften Zertifizierungsstelle mehrere verschärfte Vergabekriterien erfüllt sein (*Guidelines For The Issuance And Management Of Extended Validation Certificates* 2007). Für die Gewährleistung der Gültigkeit des TLS-Zertifikats besitzt dieses einen Ablauf-Zeitpunkt. Aus diesem Grund muss es stets rechtzeitig aktualisiert werden (Liebel 2018c).

3.3.4. Web-Schnittstellen

Wie im vorherigen Kapitel aufgezeigt, werden alle Anfragen der Ressourcen von dem Service Worker bedient. Dieser holt die Ressourcen wiederum über den Cache oder das Netzwerk ein. Um für die Umsetzung Daten zu cachen oder Netzanfragen zu stellen, werden in der Regel Webschnittstellen verwendet.

Cache API: Ursprünglich war die Cache API lediglich für die Verwendung durch den Service Worker gedacht. Dieser sollte angefragte Ressourcen über Schlüssel-Paare speichern können. Das Abrufen der Ressourcen über den lokalen Zwischenspeicher ist hierbei deutlich schneller, als Ressourcen über das Netzwerk zu laden. Mit der Caching API kann heute der Zugriff auf die Ressourcen über den Cache von jeglichen Scripten aus verwendet werden. So kann das Update-Verhalten der angefragten Ressourcen im Cache je nach Anwendungsfall spezifisch implementiert werden. Für die Aktualisierung der Ressourcen muss diese vorerst explizit angefordert werden. Die Speicherkapazität des Caches wird allerdings über den Browser begrenzt. Es ist daher wichtig den Cache selber zu bereinigen und nicht mehr benötigte Ressourcen zu entfernen, denn gespeicherte Ressourcen besitzen im Cache über kein Verfallsdatum. Sollte das Speicherlimit des Browsers überschritten werden, wird dieser einzelne Ressourcen aus dem Cache entfernen, bis der Speicher des Caches wieder unterhalb der Speichergrenze liegt(*Cache* o.D., *CacheStorage* o.D., LePage 2020). In der

Umsetzung werden Event-Listener benutzt, welche zum Beispiel beim Installieren der Webanwendung oder bei Netzwerkantworten getriggert werden. Über eine Callback-Funktion kann über die Cache API das weitere Verarbeiten der Ressourcen definiert werden. Wie bereits erläutert, sollte für eine optimale Performance anwendungsspezifisch die richtige Caching-Strategie gewählt werden.

Fetch API: Die Fetch API ist eine Schnittstelle mit der asynchrone Netzanfragen über HTTP/HTTPS innerhalb der Webanwendungen realisiert werden kann. Mit der Fetch API steht eine Schnittstelle zur Verfügung, in der auf die Request- und Response-Objekte zugegriffen werden können. Für Netzwerkanfragen einer PWA ist die Verwendung der Fetch API deswegen optimal, da diese die Spezifikation des Service Workers bezüglich des Abfangens und Manipulierens von Netzanfragen erfüllt (Russell u. a. 2019). Zusätzlich können bei einem Request über das „RequestInit“-Objekt verschiedene Einstellungen getätigt werden. Im Vergleich zu der älteren Schnittstelle „XMLHttpRequest“, werden bei der Fetch API nur asynchrone Netzanfragen gestellt. Der Grund hierfür liegt dabei an der Gefahr, dass die Webanwendung einfriert oder ins Stocken kommt. Anders als bei dem XMLHttpRequest werden die Antworten einer Netzwerkanfragen über die Fetch API nicht über eine Callback-Funktionen umgesetzt, sondern mit Promises. Die Verwendung ist somit deutlich einfacher. Zum Erstellen einer Netzanfrage über die Fetch API kann die Methode „WindowOrWorkerGlobalScope.fetch()“ verwendet werden. Diese ist eine erwähnenswerte Methode, da diese Netzanfragen in fast jedem Kontext zur Verfügung steht. Zudem können Erweiterungen für HTTP oder CORS mit der Fetch API verwendet werden (*Using Fetch* o.D.).

3.3.5. Datenspeicher

Für eine Offline-Unterstützung der Zugriffe persistenter Daten ist ein lokaler Speicher auf dem Gerät nötig. Ist dieser nicht vorhanden, können auch offline keine Daten gelesen oder geschrieben werden, sofern diese nicht im Zwischenspeicher liegen. In einer Webanwendung gibt es mehrere Möglichkeiten Daten lokal abzuspeichern.

IndexedDB: IndexedDB ist eine NoSQL Datenbank im Browser. Sie ermöglicht eine clientseitige persistente Speicherung von strukturierten Daten. Die IndexedDB ist vor allem für die Speicherung großer Mengen von Daten geeignet. Dateien können in ihr abgespeichert werden (*IndexedDB* o.D.). Das Konzept der IndexedDB wird mit JavaScript umgesetzt. Mit Hilfe von Schlüsselwerten können Objekte abgespeichert und aufgefunden werden. Die Voraussetzung eines solchen Objektes ist die Unterstützung eines strukturierten Klon Algorithmus (*The structured clone algorithm* o.D.). Für eine optimale Performance macht es Sinn für einige Eigenschaften der Objekte Indizes zu erstellen. Mit Hilfe von Indizes können Objekte schneller sortiert und gefunden werden. Gerade bei komplexen Strukturen der Objekte sind Indizes unabdingbar. Um die Konsistenz der Daten zu sichern arbeitet IndexedDB mit einem Transaktions-Datenmodell. Eine Transaktion ist ein Ablauf von Operationen, welche in diesem Fall auf Datenbestände der IndexedDB angewandt wird. Eine Transaktion wird als

eine Einheit gesehen und hat den Anspruch fehlerfrei oder nicht durchzulaufen und sorgt somit für einen konsistenten Datenbestand. Kommt es bei einer Transaktion zu Fehlern, werden bisher getätigte Änderungen des Ablaufes rückgängig gemacht. Das Committen einer Transaktion geschieht in der IndexedDB automatisch und kann nicht manuell durchgeführt werden. Das Durchführen von Transaktionen ist gerade bei Webanwendungen, bei denen mehrere Tabs gleichzeitig geöffnet sein können, sehr wichtig. Operationen geschehen in der IndexedDB meistens asynchron und mit Hilfe von Callback Funktionen. Der Callback enthält Informationen bei einer erfolgreichen Anfrage „onsuccess“ oder bei einer fehlgeschlagenen Anfrage „onerror“. Mit Hilfe des „errorCode“ kann der Fehler der Anfrage identifiziert werden. Ergebnisse einer Operation werden mit einem DOM-Ereignis kommuniziert. Die Möglichkeit einer promisebasierten Umsetzung ist mit Dexie.js jedoch auch gegeben. Zur Identifikation benutzt IndexedDB eine Same-Origin-Policy. Das bedeutet, dass die Herkunft über die Domain, das Anwendungsschichtenprotokoll und den Port bestimmt wird. Außerdem werden Datenbanken einer Herkunft mit einem eindeutigen Namen belegt. Diese Identifikation stellt zusätzlich auch eine Sicherheitsbegrenzung dar, da es eine Anwendung daran hindert auf Daten einer anderen Herkunft zuzugreifen. Vergleichbar mit einer serverseitigen Datenbank steht durch die definierte Speichergrenze des Browsers natürlich nicht annähernd so viel Speicher zur Verfügung (*IndexedDB* o.D.). Der Speicher reicht aber in der Regel mehr als aus, um benutzerbezogene Daten abzuspeichern. Die Speichergrenze ist vom Browser abhängig. Bei sensiblen Daten ist jedoch Vorsicht geboten. Denn anders als eine serverseitige Datenbank unterliegt die IndexedDB keinerlei Sicherheitsregeln. Denkbar ist, dass sich ein Benutzer auf einem fremden Gerät einloggt. Daten, welche zu diesem Zeitpunkt in die IndexedDB geschrieben worden sind, können sogar direkt vom Browser ausgelesen werden.

sessionStorage und localStorage: Für die clientseitige Speicherung kleiner Datenmengen bietet sich der session- oder localStorage an. Er wird über die Web Storage API umgesetzt. Der Zugriff auf die Daten kann von jedem Script der entsprechenden Domain geschehen. Auch hier ist die Speicherung sensibler Daten ein potenziell kritischer Sicherheitsaspekt, welcher beachtet werden muss. Der Unterschied zwischen session- und localStorage ist hierbei der Lebenszyklus. Beim Schließen vom Register Tab des Browsers, wird der sessionStorage gelöscht. Der localStorage bleibt hingegen erhalten und hat somit auch keinen Ablaufzeitpunkt. Die Umsetzung dieses Speichers geschieht über Storage-Objects. Innerhalb eines Storage-Objects können die Daten in Form von Key-Value Paaren abgespeichert werden. Wobei sowohl der Key als auch der Wert ein String sein müssen. Wird ein anderer Datentyp hineingegeben, wird dieser automatisch mit der toString-Methode konvertiert. Um den ursprünglichen Datentyp gelesener session- oder localStorage-Daten wiederherzustellen, müssen dementsprechende Parse-Methoden angewandt werden. Da die spätestens beim Verwenden komplexer Daten zu einem Problem werden können, wird das Serialisieren mit JSON empfohlen (Hickson 2016).

Eine Anwendung ohne Datenzugriff im Offline-Zustand kann die User Experience stark reduzieren. Bei einer schwachen oder stockigen Internetverbindung könnte die Anwendung im schlimmsten Fall abstürzen oder Fehler produzieren. Ist eine Offline-Unterstützung der Daten hingegen gefordert, so sollte die Synchronisation der Daten strategisch gut geplant und durchdacht sein.

3.4. Synchronisation von Daten

Angelehnt an die Offline-Verfügbarkeit von Daten, ist auch die Synchronisation der client- und serverseitigen Daten von großer Wichtigkeit. Angenommen ein Benutzer bearbeitet zunächst offline einen Eintrag mit seinem Smartphone. Etwas später bearbeitet er diesen erneut online an seinem Computer. Nach der zweiten Bearbeitung bekommt sein Smartphone wieder eine Verbindung und ist somit wieder online. Die Bearbeitungen von diesem Benutzer können nun in Konflikt miteinander stehen. In diesem Fall besteht der Konflikt in der Aktualität der Bearbeitungen. Ein solcher Konflikt kann jedoch in vielen verschiedenen Formen ent- und bestehen und existieren. Folgen solcher Konflikte können zum Beispiel Datenverlust, veraltete Daten oder falsche Daten sein. Es ist somit wichtig, dass die Synchronisierung eine Strategie verfolgt, welche der Wichtigkeit der Daten gerecht wird.

Synchronisationsprobleme können auch dann auftreten, wenn mehrere Benutzer an einer Ressource arbeiten. Am Beispiel von Git wird die Entscheidung, welche Bearbeitung Vorrang hat, dem jeweiligen Benutzer überlassen. Bei der Synchronisation von Offline- und Online-Daten funktioniert diese Konfliktlösung hingegen nicht, da der Konflikt zum Zeitpunkt der Entstehung noch nicht bekannt ist. Denn ohne die Online-Daten kann offline kein Vergleich der Daten gezogen werden. Doch auch hier gibt es Lösungsansätze, um einen solchen Konflikt zu lösen. Der einfachste, aber wohl auch kritischste Lösungsansatz wäre eine „last one wins“-Strategie. In dieser werden die Operationen auf die Daten in der Reihenfolge akzeptiert in der diese hinein kommen. Kritisch ist es deswegen, weil die Aktualität und Priorität sowie andere möglicherweise wichtige Aspekte der Operation nicht berücksichtigt werden. Außerdem können sich Operationen bei der Synchronisierung vermischen und zu inkonsistenten Datenbeständen führen, sofern diese nicht als Transaktion durchgeführt werden. Ein besserer Lösungsansatz ist das Speichern des Zeitpunktes oder auch Timestamp genannt. Jegliche Daten, die geschrieben werden, bekommen einen „Zeitlichen Stempel“. Die Synchronisierung der Daten geschieht somit in chronologischer Reihenfolge. Mit diesem Vorgehen kann die Aktualität der Daten bestimmt werden. Je nach Anwendung können natürlich noch weitere spezifische Aspekte in die Synchronisation hineinfließen, wie zum Beispiel die Priorität des Operanten oder die Rollen der Benutzer und so weiter. Bei einer Strategie zur Synchronisation sollte grundlegend die Frage nach einer einseitigen(One-Way) oder zweiseitigen(Two-Way) Synchronisation gestellt werden. Eine einseitige Synchronisation geschieht in diesem Fall entweder vom Server zum Client oder vom Client zum Server. Ist eine Anwendung nach dem Konzept “Offline-first“ entwickelt und besitzt zum

Beispiel lediglich eine Rolle mit einem Schreibrecht, könnte eine einseitige Synchronisation (vom Cache zum Server) sinnvoll sein. Bei der zweiseitigen Synchronisation findet die Synchronisation der Daten client- sowie server-seitig statt. Eine Strategie der zweiseitigen Synchronisation ist beispielsweise die „Server wins“- oder auch „Client wins“-Strategie. Mit dem Zuspruch der Gültigkeit auf Server- oder Client-Seite ist diese Strategie ebenfalls eine sehr einfache, jedoch kritische Synchronisation.

Ein weiteres Problem bei der Synchronisation ist das Erkennen gelöschter Daten. Denn bei der Synchronisation ist ohne weitere Maßnahmen nicht zu erkennen, ob Daten serverseitig hinzugefügt oder clientseitig entfernt wurden sind. Die einfachste Lösung ist es Löschungen zu vermeiden. Diese Lösung ist sehr anwendungsspezifisch und sicher nicht für alle Anwendungen umzusetzen. Soll eine Löschung möglich sein, so kann eine Verfolgbarkeit der Löschvorgänge hergestellt werden. Die Implementierung geschieht hierbei über einen Mechanismus, welcher eine Liste aller gelöschten Daten führt. Als letzte Alternative können auch Flags jedem einzelnen Datensatz hinzugefügt werden. Diese Möglichkeit, sollte jedoch als letztes gewählt werden, da mit Ausnahme in jeder Operation der Flag berücksichtigt werden muss. Ein Flag könnte beispielhaft ein Boolean sein, welcher bei einer Löschung auf „true“ gesetzt wird (Whitehorn o.D.).

3.5. Push-Benachrichtigungen

Push-Benachrichtigungen sind der zentrale Punkt des Merkmals „Nutzer bindend“. Dieser wurde bereits fundamental in Abschnitt 3.1 erklärt. In der Webentwicklung werden Push-Benachrichtigungen technisch über die „Push API“ realisiert. Die Umsetzung dieser Schnittstelle ist von der Normenorganisation W3C definiert worden. Das Konzept der API setzt hierbei die Verwendung eines Service Worker voraus. Der Ablauf beginnt zunächst mit der Registration des Service Workers. Die Anfrage wird von der Webseite an den Benutzer Agenten („user agent“) gestellt. Als Rückgabe wird eine „ServiceWorkerRegistration“ zurückgegeben. Nun kann die Webseite das Anmelden („subscribe“) beim Push Service über den Benutzer Agenten („user agent“) anfragen. Dieser fordert die Erlaubnis Push-Benachrichtigungen an den Benutzer zu stellen. Ohne Erlaubnis ist das Senden von Push-Benachrichtigungen nicht möglich. Der Benutzer Agent würde demnach die Anmeldung nicht an den Push Service weitergeben. Willigt der Nutzer ein und ist die Anfrage erfolgreich, wird eine „subscription“ zurückgegeben. Diese enthält eine eindeutige Zeichenkette, welche als Adresse dient. Nun muss dem Applikations-Server die „push subscription“ mitgeteilt werden, damit dieser einen Eintrag als Benachrichtigungs-Ziel tätigen kann. Standardmäßig wird die Subscription in eine Datenbank geschrieben. Der Applikation-Server kann nun Benachrichtigungen an den Push-Service senden. Dieser gibt die Benachrichtigung an den Benutzer Agenten weiter, welcher die Erlaubnis prüft. Wurde der Erlaubnis Push-Benachrichtigungen zu erhalten bereits zugestimmt, wird diese an den Service-Worker weitergeleitet und von diesem verarbeitet. Sollen keine Benachrichtigun-

gen mehr erhalten werden, wird dem Applikations-Server dies zunächst mitgeteilt. Auf eine Antwort muss nicht gewartet werden, da die Subscription im nächsten Schritt von der Webseite über den Benutzer Agenten beim Push Service abgemeldet wird. Der Push Service sendet nach erfolgreichem Abmelden eine Antwort über den Benutzer Agenten zurück an die Webseite (Beverloo u. a. 2020).

Zur Veranschaulichung folgt ein Beispiel eines Push-Services. Dieser soll einen Benutzer im Backend registrieren und als Test-Nachricht eine Push-Benachrichtigung zusenden. Für dieses Beispiel wird eine aktuelle Version von Node.js vorausgesetzt.

Backend Code: Als erster Schritt muss zunächst ein Ordner erstellt werden, indem das Projekt liegen soll. Mit „npm init“ wird dieser als Projekt-Ordner für Node.js initialisiert. Daraufhin müssen die benötigten Bibliotheken heruntergeladen werden. Hierfür wird der Befehl „npm install web-push express body-parser“ in der Konsole verwendet. Die Verwendung dieser werden in folgender Erläuterung ersichtlich.

```
1 const express = require("express");
2 const webpush = require("web-push");
3 const bodyParser = require("body-parser");
4 const path = require("path");
5 const cors = require('cors');
6 const app = express();
```

Listing 3.8: Backend - index.js Bibliotheken importieren

Vorerst müssen alle geladenen Bibliotheken importiert werden.

```
1 // Konfigurationen
2 app.use(cors());
3 app.use(express.static(path.join(__dirname, "client")));
4 app.use(bodyParser.json());
```

Listing 3.9: Backend - index.js Konfigurationen

Für das Zugreifen des Backends wird der Mechanismus CORS ausgewählt. Dieser erlaubt serverseitig den Zugriff über einen Cross-Origin-Request. Bei diesem handelt es sich um den internen Zugriff einer Webseite einer gegebenenfalls anderen Domain auf diesen Server. Wäre der Mechanismus CORS in diesem Beispiel hingegen nicht ausgewählt, würde die Same-Origin-Policy den Zugriff verweigern. In Zeile drei wird über Express eine neue Middleware Funktion erstellt. Dies ist eine Funktion, welche über den angegebenen statischen Pfad reinkommende Ressourcenanfragen bedienen kann (**cors**). Die letzte Zeile gibt an, dass die Nachrichten im JSON-Format kommuniziert werden.

```
1 const publicKey = "BIcYx8hh6XlDoR3z1RQNTzHVEvYDBUN79UhXBQTGyNvl-
    pHVW9ehEeXKkHNIHwYFdtaoHPk5aGMstnZMuW-7AvA"
```

```
2 const privateKey = "i-zhyVP6OHPeOJ10-MWFsM1YLevPpoUfPrPR4rcVOhA"
```

Listing 3.10: Backend - index.js Private- und Public-Key

Als Authentifizierungsmethode wird die Public-Key-Authentifizierung verwendet. Hierzu wird mit der Hilfe des privaten Schlüssels eine Signatur auf dem Server erzeugt. Diese kann wiederum mit dem öffentlichen Schlüssel (Public Key) des Webclients verifiziert werden. Das Schlüsselpaar kann über das Terminal im Projektordner über den Befehl „./node_modules/.bin/web-push generate-vapid-keys“ erzeugt werden. Anmerkung: Die Schlüssel sollten in einem realen Projekt als Environment Variable abgespeichert und nicht innerhalb des Anwendungscodes definiert werden.

```
1 webpush.setVapidDetails(
2   "mailto:example@example.com",
3   publicKey,
4   privateKey
5 );
```

Listing 3.11: Backend - index.js Vapid-Details

Zum Versenden von Push-Benachrichtigungen wird in diesem Bereich die Autorisierung des Senders durchgeführt. Dies geschieht über das Schlüsselpaar und eine E-Mail Adresse.

```
1 // Registrieren
2 app.post("/register", (req, res) => {
3   // pushSubscription Object entgegennehmen
4   const subscription = req.body.notification;
5   // Response-Status senden
6   res.status(201).json({});
```

Listing 3.12: Backend - index.js Registrations-Methode

Innerhalb des statischen Pfades soll sich der Webclient im Pfad „/register“ für die Push-Benachrichtigungen eintragen können. Nach dem Eintragen wird ein Response mit dem Status 201 zurückgesendet.

```
1   // Payload konfigurieren
2   const payload = JSON.stringify({
3     notification: {
4       title: 'Push-Benachrichtigung',
5       body: 'Sie koennen nun Push Notifications erhalten',
6     }
7   });
```

Listing 3.13: Backend - index.js Payload

In diesen Zeilen wird eine Push-Benachrichtigung definiert. Diese beinhaltet einen Titel und einen Body. Natürlich können noch weitere Konfigurationen einer Notification getätigt werden.

```
1   // Push-Benachrichtigung senden
2   webpush
```

```
3       .sendNotification(subscription, payload)
4       .catch(err => console.error(err));
5   });
```

Listing 3.14: Backend - index.js Push-Benachrichtigung senden

Zum Versenden einer Push-Benachrichtigung wird die Subscription und die Payload benötigt und als Parameter angegeben.

```
1   app.listen(5000, () => console.log(`Serverport ${port}`));
```

Listing 3.15: Backend - index.js Push-Benachrichtigung senden

Als letztes muss der Port des Servers angegeben werden, auf dem die Anfragen entgegengenommen werden sollen.

Clientseitiger Code: Für den Webclient muss zunächst ein Ordner „client“ erstellt. In diesem wird eine

- „index.html“ für die Benutzeroberfläche
- „client.js“ zum Registrieren des Service Workers
- „sw.js“ welche die Implementierung des Service Worker enthält

erstellt.

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>Example</title>
7   </head>
8   <body>
9       <h1>Push-Benachrichtigung</h1>
10      <script src="client.js"></script>
11  </body>
12  </html>
```

Listing 3.16: Client - index.html Benutzeroberfläche

Für dieses Beispiel soll eine Überschrift als Benutzeroberfläche ausreichen. Wichtig ist hierbei, wie in Zeile zehn zu sehen, dass das Script „client.js“ eingebunden wird.

```
1   const publicKey = "BIcYx8hh6XlDoR3z1RQNTzHVEvYDBUN79UhXBQTGyNvl-
        pHVW9ehEeXKkHNIHwYFdtaoHPk5aGMstnZMuW-7AvA"
```

Listing 3.17: Client - client.js Öffentlicher Schlüssel

Wie im Backend-Teil erklärt, benötigt der Client den öffentlichen Schlüssel zur Authentifizieren am Server. Und auch hier ist der öffentliche Schlüssel nur für Veranschaulichungszwecke als statische Variable innerhalb des Scriptes angegeben.

```
1 if('serviceWorker' in navigator) {
2     example().catch(err => console.log(err));
3 }
4 async function example() {
5     const register = await navigator.serviceWorker.register('/sw.js', {
6         scope: '/'
7     });
```

Listing 3.18: Client - client.js Service Worker Registration

Noch vor der Registration der Push-Benachrichtigungen muss der Service Worker registriert werden.

```
1     const subscription = await register.pushManager.subscribe({
2         userVisibleOnly: true,
3         applicationServerKey: urlBase64ToUint8Array(publicKey)
4     })
```

Listing 3.19: Client - client.js Push-Benachrichtigung Registration Browser

Für die Registration beim Browser der Push-Benachrichtigungen wird „userVisibleOnly“ auf true gesetzt. Dies bewirkt, dass dem Benutzer mit seiner Erlaubnis Push-Benachrichtigungen angezeigt werden dürfen. Der „applicationServerKey“ wird für die Authentifizierung beim Server benötigt. Hierbei handelt es sich um den öffentlichen Schlüssel. Bevor dieser jedoch definiert werden kann, muss dieser jedoch vorher in einen Uint8Array konvertiert werden. Als Response wird die Subscription zurückgegeben. Da es sich um eine asynchrone Funktion handelt muss der Befehl „await“ zum Warten des Response hinzugefügt werden.

```
1     //Registrieren
2     await fetch("/register", {
3         method: "POST",
4         body: JSON.stringify(subscription),
5         headers: {
6           "content-type": "application/json"
7         }
8     });
9 }
```

Listing 3.20: Client - client.js Push-Benachrichtigung Registration Backend

In Zeile zwei wird über die Fetch API ein asynchroner Request an das Backend gestellt. Da der Client sich für die Push-Benachrichtigungen eintragen soll, wird die Methode „POST“ verwendet. Im Body des Requests wird die Subscription in das JSON-Format geparsed und angegeben. Die Subscription dient dem Server später als Zieladresse der Push-Benachrichtigungen. In Zeile sechs

wird im Header noch einmal die Information angegeben, dass der Body im JSON-Format angegeben wird.

```
1 self.addEventListener('push', event => {
2   const notification = event.data.json();
3   self.registration.showNotification(notification.title, notification);
4 });
```

Listing 3.21: Client - sw.js Push-Benachrichtigung empfangen

Innerhalb des Service Worker wird nun ein Event Listener hinzugefügt, welcher bei eingehende Push-Benachrichtigungen eine Notifikation anzeigt. Die Informationen der Push-Benachrichtigung werden in Zeile zwei als JSON über das Event entgegengenommen. Als Information ist wie in Listing 3.13 zu sehen nur der Titel und die Nachricht angegeben.

4. Architektur von Angular

Da es sich bei einer PWA um eine Webanwendung handelt, liegt dementsprechend auch ein großer Fokus auf der Entwicklung solcher. Der Standard moderner Webentwicklung ist die komponentenbasierte Entwicklung und Programmierung von Single-Page Webanwendungen. Diese werden mit Frameworks wie beispielsweise React, Vue oder Angular umgesetzt. Jedoch muss vor der Erklärung des Angular-Frameworks zunächst zwischen einer Bibliothek und einem Framework unterschieden werden.

Bibliotheken: Bei einer Bibliothek handelt es sich um eine Ansammlung von Programmcode zum Beispiel in Form von Funktionen, Objekten oder Makros, die für die Verwendung bereitgestellt wird. Hierbei wird zwischen statischen und dynamischen Bibliotheken unterschieden. Eine statische Bibliothek wird nach der Kompilierung durch den Linker mit dem ausführbaren Programm zusammengeführt (Terzibaschian o.D.). Eine dynamische Bibliothek hingegen wird erst zur Laufzeit eingebunden. Bibliotheken sind keine Neuheiten in der Webentwicklung, dennoch können diese unter anderem moderne Schnittstellen realisieren oder zu einer Problemlösung beitragen. Um ein Beispiel zu nennen setzt dexie.js eine Lösung promise-basierten Zugriffes der IndexedDB um.

Frameworks: Die objektorientierte sowie komponentenbasierte Softwareentwicklung ist in der nativen Anwendungsentwicklung schon lange etabliert. Für die komponentenbasierte Webentwicklung werden hingegen Frameworks zur Umsetzung benötigt. Bei diesen handelt es sich um ein Programmierkonstrukt, welches generische Funktionen zur Verfügung stellt. Der Entwickler kann diese generischen Funktionen mit selbst definierten Code ändern, sodass diese zu einer anwendungsspezifischen Software angepasst wird. Die Konzeptionierung eines Frameworks setzt in der Regel eine komponentenbasiertes Konstrukt um. Durch dieses kann modulare, universelle und wiederverwendbare Software geschrieben werden. Außerdem bietet ein Framework eine standardisierte Entwicklung der Webanwendungen. So werden bei der Implementierung bereits bewährte Entwurfsmuster und Anwendungsarchitekturen umgesetzt. Bei einem Framework handelt es sich aber nicht um eine Bibliothek (Lüecke 2005). Um eine Abgrenzung zu schaffen sind dies die unterscheidenden Merkmale:

- Der Framework-Code kann nicht verändert, sondern lediglich über die vorgegebenen generischen Funktionen erweitert werden (Open-closed principle (Goll 2019a)).
- Ein Framework kann erweitert werden (Extensibility principle).
- Der Steuerfluss der Anwendungsschicht wird vom Framework vorgegeben (Inversion of Control principle (Goll 2019b)).

Für die Entwicklung einer PWA bietet sich Angular als Framework besonders gut an, da es zusätz-

lich eine PWA-Unterstützung gibt, welche das Umsetzen erleichtert und gleichermaßen Zeit spart. Auch native OS APIs stehen in Angular zur Verfügung. Grundlegend ist das Framework zur Umsetzung einer PWA aber unbedeutend. Wichtig ist nur, dass die Webanwendung schlussendlich die Merkmale und Voraussetzungen einer PWA erfüllen. Für eine gute Qualität des Endproduktes (die PWA), sollten die Funktionsweisen und das Konzept des Frameworks verstanden sein.

4.1. Decoratoren

Decoratoren sind ein Teil des Hauptkonzeptes von Angular Core. Ein Decorator ist ein Strukturmuster, welches verwendet wird, um Funktionen und Klassen dynamisch mit zusätzlichen Funktionalitäten zu erweitern(Gamma u. a. 1996).

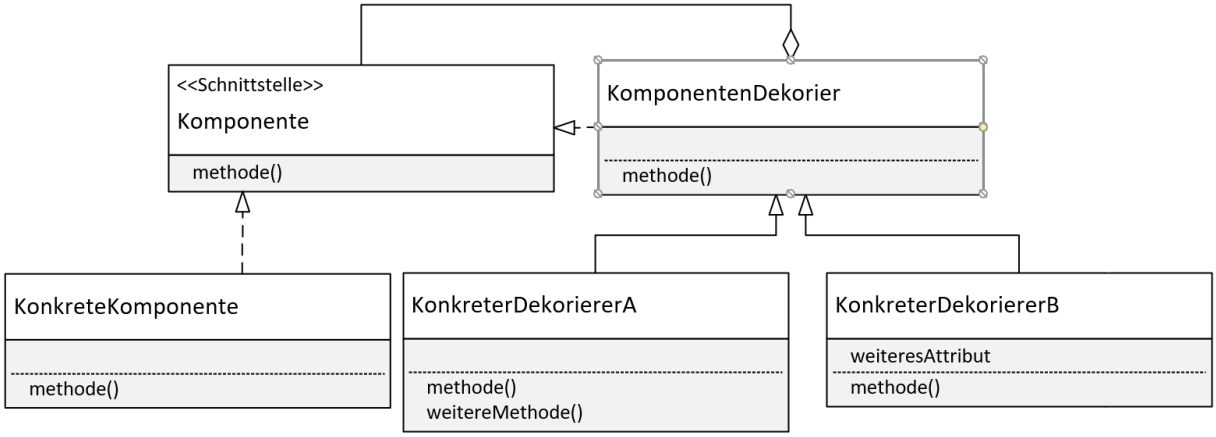

Abbildung 4.1.: Decorator Strukturmuster

Zunächst wird die zu dekorierende Funktion/Klasse über die Schnittstelle (Interface) kommuniziert. Die konkrete Komponente stellt in diesem Zusammenhang den Grundzustand der Klasse/Funktion dar. Mit Hilfe der Dekorierer-Klasse wird diese mit einer Klasse zusätzlicher Funktionalitäten spezialisiert. In Angular wird über das Zeichen „@“ dem Parser, die Verwendung eines Decorators mitgeteilt. Angebunden am Zeichen wird die zu verwendende Funktion/Klasse angegeben. Diese wird erweitert und zurückgegeben. In Angular wird bei der Anwendung von Decoratoren unterschieden (Motto 2017):

Class Decorators werden in Angular verwendet, um Metadaten für die Konfiguration der Kompilierung anzugeben. Diese werden auch als „top-level decorators“ bezeichnet, da sie die wichtige Kennzeichnung einer Klasse bestimmen also, ob es sich hierbei um eine Komponente oder ein Modul handelt. Ein Modul wird mit dem Decorator „@NgModule“ und eine Komponente mit dem Decorator „@Component“ identifiziert. Beim Erstellen eines neuen Projektes über die CLI werden

grundlegende Vorkonfigurationen mit dem Decorator „@NgModule“ getätigt:

- Die „declarations“ geben Bestandteile wie Komponenten, Anweisungen und Pipes an, die zu dem entsprechenden Modul gehören.
- Die „imports“ geben dem Netzwerkmodul, das für die Netzwerkaufrufe zuständig ist, an welche Module innerhalb dieses Moduls als Vorlage verfügbar sind.
- Die „providers“ geben die injizierbaren Objekte an, die in diesem Modul verwendet werden dürfen. Oft sind diese Objekte Services.
- „bootstrap“ gibt jene Komponenten an, die beim Start des Moduls gebootet werden sollen. In der Regel ist das die Root-Komponente „AppComponent“.

Auch beim Erstellen einer Komponente über die CLI werden standardmäßig drei Konfigurationsoptionen in dem Decorator „@Component“ angegeben:

- Der „selector“ wird hierbei für die Identifizierung der Komponente innerhalb eines Templates verwendet.
- Die „templateUrl“ gibt den Pfad zur HTML-Template Datei an.
- Die „styleUrls“ gibt die Pfade zu den Style Dateien an.

Eine Übersicht aller Konfigurationsoptionen ist in der Dokumentation von Angular unter dem Link „*API List* o.D.“ zu finden.

Property Decorators werden in Angular für die Bindung eines Wertes an die Eigenschaft einer Klasse verwendet. Explizit wird diese Umsetzung des Bindings auch „Property Binding“ genannt. Beispiele sind hierbei die Decoratoren „@Input“ oder „@Output“, die jeweils die Ein- oder Ausgabe einer Eigenschaft frei geben.

Method Decorators werden im Gegensatz zu den Property Decorators nicht für die Bindung von Werten, sondern von Methoden einer Klasse verwendet. So kann beispielsweise mit dem Decorator „@HostListener“ eine Funktion bei einem definierten Event ausgeführt werden.

Parameter Decorators werden in Angular für die Manipulation von Parametern einer Methode oder des Konstruktors verwendet. Genauer kann ein Objekt, welches als Parameter des Konstruktors übergeben wird beispielsweise mit dem Decorator „@Inject“ mit einer erstellten Klasse erweitert werden.

4.2. Components

Um in Angular eine Komponente zu erstellen, wird der Befehl „ng generate component «Name»“ verwendet. Eine Komponente ist ein wiederverwendbarer Bestandteil, der aus einer View (DOM) und einer logischen Einheit besteht. Die logische Einheit kann über eine API mit der View interagieren. Die Angaben der Metadaten werden über Class Decoratoren angegeben. Diese enthalten die Informationen über die Bestandteile der Komponente und ihren Pfad, mit denen die Komponente mit ihrer View erstellt werden kann. Die Logik der Komponente wird innerhalb einer Klasse implementiert. Über ein Template wird die Struktur der Komponente in einer Mischform der Marked-Up Language HTML und der Angular eigenen Template Syntax definiert. Innerhalb des Templates können also auch Data-Bindings, Directives und Pipes verwendet werden. Style-Angaben können über CSS-Sprachen definiert werden. Auch können Komponenten über den „selector“ innerhalb eines Templates verschachtelt werden, sodass eine Hierarchie entsteht (*Angular Components - The Fundamentals* 2020).

4.2.1. Directives

Directives sind ein großer Bestandteil von Angular. Mit ihnen können strukturelle, optische und wertgebundene Direktiven bestimmt werden. Eine Komponente in Angular ist beispielsweise auch ein Directive, da dieser das Verhalten in der View (DOM) mit dem Template manipuliert.

Attribute Directives werden mit dem Decorator „@Directive()“ markiert. In diesem ist der „selector“ angegeben, welcher in einem Template innerhalb eines Tags auf diesem Directive verweist. Das Verhalten dieses Elementes kann nun durch die Implementierung der Klasse des Attribut Directives manipuliert werden. Über den Konstruktor der Directive Klasse kann auf die Element-Referenz des View-Elements über den Injektor zugegriffen werden. Über diesen Weg können Animationen, optische Veränderungen zur Laufzeit oder auch Interaktionen mit dem Benutzer umgesetzt werden (*Attribute directives* o.D.).

Structural Directives sind im Gegensatz zu den Attribut Directives auf die Struktur der DOM spezialisiert. Mit diesen werden View-Elemente hinzugefügt, entfernt oder auch bearbeitet. Angular selbst stellt einige Structural Directives für den häufigen Gebrauch zur Verfügung. Außerdem können auch eigene Structural Directives implementiert werden. Bei der Erstellung eines Structural Directives stellt Angular eine Microsyntax zur Verfügung. Diese gibt die Möglichkeit ein implementiertes Structural Directive über einen String im Template zu konfigurieren. Structural Directives werden mit einem Stern vorab des „selectors“ angegeben. Der Zugriff der Klasse eines Structural Directives der DOM geschieht ebenfalls über den Konstruktor. Anders als beim Attribut Directive benutzt der Structural Directive Container, um mit Hilfe des Templatereferenten die View zu erstellen, zu entfernen oder zu bearbeiten (*Structural directives* o.D.).

4.2.2. Data-Bindings

In Angular wird die Kommunikation zwischen der View (DOM) und der Komponente mit Data-Bindings umgesetzt. Hierbei können unter anderem Templates mit Daten gefüllt oder Events in der View (DOM) der Komponente mitgeteilt werden. In Hinblick auf das Prinzip von Single-Page Webanwendungen soll zum Beispiel das Auswerten von Formen nicht auf einer anderen Seite stattfinden. Mit der Lösung des Data-Bindings profitiert auch die Performance der Webanwendung, da diese keine weiteren Seiten laden muss. Zusätzlich bekommt die Komponente Änderungen der Daten aus der Form direkt mit und kann bereits vor dem vollständigen Ausfüllen eine Validierung durchführen. Das Data-Binding kann in verschiedenen Formen umgesetzt werden (*HTML attribute vs. DOM property* o.D.).

Im One-way Data-Binding werden Daten exklusiv nur in eine Richtung gebunden. Sollen also Daten von der Komponente zur View (DOM) gebunden werden können Interpolations verwendet werden. Diese werden innerhalb des HTML-Templates mit der Template Expression „“ identifiziert. Innerhalb dieser Klammern ist es möglich Methoden, Eigenschaften oder JS Anweisungen, et cetera zu benutzen. Anders als beim Interpolation Binding, können Daten auch über die Eigenschaften eines View Elementes gebunden werden. Die Eigenschaften eines View Elementes in Angular werden innerhalb eckiger Klammern angegeben. Property Binding ermöglicht es beispielsweise über die Eigenschaft „value“ eines View Elementes den Wert dieses mit einer Eigenschaft der Komponente zu binden (*Data Binding in Angular* 2016). Auch über Attribut Binding können Daten von der Komponente an die View (DOM) gebunden werden. Diese Art des Bindings sollte jedoch erst zweitrangig verwendet werden, da sie nicht so performant und intuitiv wie das Property Binding ist. Ähnlich wie dieses Prinzip können auch Style-Anweisungen und Klassenattribute eines View Elementes festgelegt werden. Diese Bindings werden als „Style Binding“ und „Class Binding“ bezeichnet. Das Binden von Daten oder expliziter: „Events“ wird über Event Bindings realisiert. Innerhalb der runden Klammern wird das Event angegeben, welches die entsprechende Methode/Funktion auslöst. Das Two-way Data-Binding realisiert beide Richtungen. So können zum Beispiel Daten innerhalb einer Form verändert und bei eventuellen Aktualisierungen über die Webanwendung rückwirkend angepasst werden (*HTML attribute vs. DOM property* o.D.).

In Verbindung mit den Decoratoren und Directives können Komponenten optimal für die Visualisierung strukturierter dynamischer Daten verwendet werden. Mit Property Decoratoren werden die nötigen dynamischen Daten zur Visualisierung an die Eigenschaften der Komponente gebunden. Weiterführend können diese Eigenschaften über Interpolation oder Property Binding mit der View (DOM) gebunden werden. Für die Umsetzung kann das Directive „*ngFor“ im Template für das Iterieren der Daten verwendet werden.

4.2.3. Pipes

Mit Hilfe von Pipes kann die Ausgabe eines Wertes in der View verändert werden. So kann das Format von einem Datum oder die Zeitzone einer Uhrzeit in der View richtig dargestellt werden. Pipes werden innerhalb des Templates mit der Template Expression „<Information> | <Format>“ realisiert. Angular beinhaltet standardmäßig eine Reihe von Pipes, welche in der Entwicklung öfters benötigt werden. Enthalten sind hier: Datums-, Groß- und Kleinschreib- sowie Währungs-Pipes. Eine Pipe kann innerhalb der Template Expression mit einer Weiteren kombiniert werden (Chaining pipes). Je nach Anwendungsfall können auch eigene Pipes implementiert werden (Custom pipes). Mit Hilfe des Decorators wird zwischen einer „pure“ und „impure“ Pipe unterschieden. Diese Angabe bezieht sich auf den Eingabewert der Pipe. Die Sprache ist von „pure“, wenn eine Änderung eines einfachen Datentyps oder der Objektreferenz eines komplexen Datentyps vorgenommen wird. Bei einer „impure“ Pipe wird hingegen jegliche Änderungen erfasst. Eine „impure“ Pipe benötigt deswegen auch um einiges mehr Rechenleistung als eine „pure“ Pipe. So sollte bestenfalls vorerst eine Änderungserkennung einer „pure“ Pipe implementiert werden, statt eine „impure“ Pipe zu verwenden (*Pipes* o.D.).

4.3. Modules

Angular besitzt ein eigenes modulares System, welches es dem Nutzer ermöglicht je nach Anliegen einzelne Module dem Projekt hinzuzufügen. Dieses System heißt NgModules. Ein Angular-Projekt besitzt immer ein Root-Modul namens „AppModule“. Innerhalb eines Projektes können Funktions- oder Angular-Module hinzugefügt werden (*Introduction to Angular concepts* o.D.).

Funktions-Module können über die CLI von Angular erstellt werden. Diese bieten einem Entwicklerteam die Möglichkeit einzelne Funktionalitäten gekapselt vom Root-Modul zu entwickeln. Dies wirkt sich positiv auf die Arbeitsstrukturierung und das Verwalten der Speichergröße des Root-Modules aus. Für eine optimale Einteilung der Funktionsmodule, sollten diese sich in einer der folgenden fünf Kategorien einordnen lassen (*Types of feature modules* o.D.):

Domain Funktions-Modul: Hierbei handelt es sich um ein Modul, welches eine Applikation in seine logischen Einheiten einteilt. Dies wird über eine Hauptkomponente realisiert, welche jeweils einzelne Subkomponenten besitzt. So könnte zum Beispiel der Administrationsbereich von dem normalen Benutzerbereich getrennt werden. Auf Grund ihrer Funktion, sollte ein Domain Funktions-Modul nur wenige bis keine Provider beinhalten. Sollten Provider nötig sein, so muss der Lebenszyklus mit dem des Moduls gleich sein. Da es sich bei diesem Domain Funktions-Modul um eine Logikeinheit der Applikation handelt, wird dieses auch nur einmal importiert. Die Importierung kann hier entweder von einem weiteren größeren Funktions-Modul oder vom „AppModule“ der Applikation importiert werden.

Das **Routed Funktions-Modul** ist grundlegend auch ein Domain Funktions-Modul, mit dem Unterschied, dass die einzelnen Hauptkomponenten hier die Ziele der Navigationsrouten sind. Eine Form der Umsetzung ist das Lazy-loaded Modul. Dieses Modul wird mit dem Entwurfsmuster „Lazy Loading Design Pattern“ realisiert. In diesem wird ein verwendetes Objekt erst initialisiert, wenn dieses tatsächlich benötigt wird. Lazy Loading kommt dann zur Anwendung, wenn das Erstellen eines Objektes viele Systemressourcen in Anspruch nimmt, obwohl dieses nur selten verwendet wird. In Angular kann das Lazy-loaded Modul bei großen Applikationen mit vielen Routes verwendet werden, um die Bundle-Größe zu verringern und folglich die Ladezeit zu verkürzen.

Routing Modul: Dieses Modul separiert ausschließlich die Aufgabe des Routings eines anderen Moduls. Das Routing Modul kann Guards als Provider verwenden, um die Befugnisse eines Benutzers auf diesem Modul zu kontrollieren. Es fügt Router-Konfigurationen zum Modul hinzu und definiert die Routes des Begleiter-Modules. Für die Erkennung eines Routing Moduls trägt dieses den gleichen Namen wie das Begleit-Modul. Ergänzend wird das Suffix „Routing“ angehängt. Für die Funktionalität des Routings, kann das Begleit-Modul auf die Router-Anweisungen des Moduls zugreifen.

Das **Service Funktions-Modul** besitzt, anders als die anderen Module, keine Deklarationen von Komponenten und hat dementsprechend auch keine Exports. Mit diesem Modul sollen lediglich Services wie Datenzugriffe aus einer Datenbank oder Push-Benachrichtigungen zur Verfügung gestellt werden. Das Service Funktionsmodul wird über das „AppModule“ importiert.

Ein **Widget Funktions-Modul** stellt eine Sammlung von UI-Komponenten zur Verfügung. Vor allem immer wiederverwendete Komponenten, sollen in diesem Modul deklariert werden. Hierfür können wenn nötig Provider benutzt werden. Dieses Modul kann in allen anderen Funktions-Modulen importiert werden, sofern diese von den UI-Komponenten Gebrauch machen.

Angular-Modules stellen zusätzliche Funktionalitäten zur Verfügung und setzen das Konzept als Framework um. Unter anderem werden jegliche Bibliotheks-Module zur Verfügung gestellt, welche „Components“, „Directives“, „Services“ et cetera realisieren.

4.4. Injektor und Services

In der objektorientierten und auch der komponentenbasierten Entwicklung gilt: Je höher die Kohäsion, umso besser. Eine Komponente hat spezifisch nur die Aufgabe die View und ihre Logik umzusetzen. So könnte eine Komponente für die Darstellung eines Formulars und dem Auslesen der im Formular enthaltenen Werte zuständig sein. Das Verarbeiten dieser Daten ist nicht die Aufgabe der

Komponente. Stattdessen können Services genutzt werden. Bei Services spricht man von Diensten, welche Aufgaben von Komponenten übernehmen, um diese klein und effizient zu halten. Die Aufgabe der Verarbeitung von den Daten würde in dem Formular-Beispiel also in eine Service-Klasse ausgelagert werden. Um die Methoden eines Services aufzurufen, werden in Angular Dependency Injections verwendet. Angular stellt für einige Reglementierungen eigene Frameworks zur Verfügung. Diese kommen beim Verwenden von Services und Deklarationen eines Moduls zum Einsatz. Ziel dieser Injectors ist es grundlegend die Webanwendung effizienter und modularer zu gestalten. Szenario-abhängig wird zwischen den Injections unterschieden:

Die **Dependency Injection** wird über ein Entwurfsmuster dem „Dependency Injection pattern“ realisiert. Dieses setzt das Single-Responsibility-Prinzip um. Es soll also eine Komponente geben, welche die Objekte zur Laufzeit managed und als zentraler Vermittler agiert. In Angular wird die Dependency Injection für die Vermittlung von Services und Objekten an eine Klasse verwendet, welche von diesen Methodenaufrufe ausführen sollen. Somit muss keine neue Instanz der Klasse erzeugt werden. Damit ein Service via Dependency Injektion vermittelt werden kann, wird die Klasse mit dem Decorator „@Injectable()“ markiert. In diesem muss der Provider des Services angegeben werden. Ein Provider stattet das Objekt mit einem DI Token aus. Dieser DI Token wird von dem Injektor zur „Bestimmung der Laufzeitversion des Abhängigkeitswertes benutzt“ (übersetzt von *Dependency providers* o.D.). Das Objekt des Services wird über den Konstruktor übergeben.

Die **Hierarchical injectors** teilen sich in Angular in „ModuleInjector“ und „ElementInjector“ auf. Grundlegend verfügen diese über Regeln der Sichtbarkeit eines Injectables. Über diese können die Provider einem Modul, einer Komponente oder einer Directive zugewiesen werden. Bei „Mo-

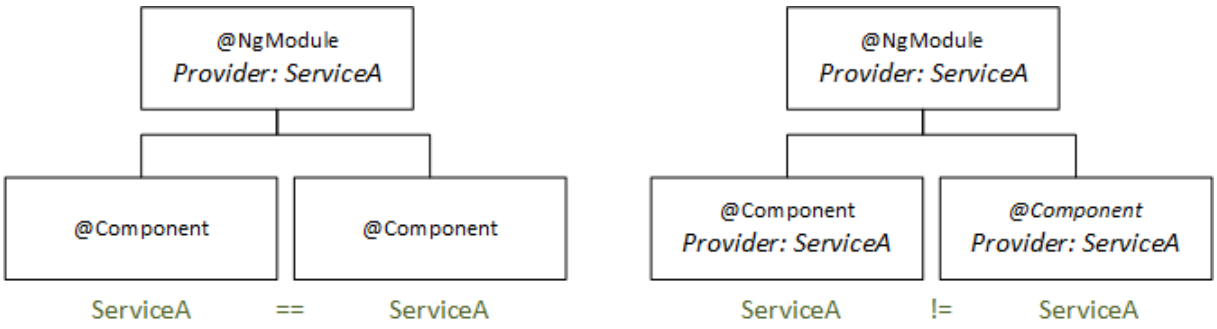

Abbildung 4.2.: Hierarchical injectors - Instanzen

duleInjector“ handelt es sich um Komponenten mit den „NgModule“ und „Injectable“ Decorator. Bei „ElementInjector“ handelt es sich um Komponenten mit den „Directive“ und „Component“ Decorator. In beiden kann ein Service über die „providers“-Angabe deklariert werden. Das Anfordern einer Instanz geschieht zunächst über den ElementInjector und erst danach über den ModuleInjector. Ist der Service innerhalb des Moduls instanziiert, können dessen Komponenten diesen Service über

das Singleton-Entwurfsmuster verwenden. Sollte dieser jedoch den Service selbst als Provider angegeben haben, würde eine neue Instanz erstellt werden. Komponenten, die sich gegenüberstehen, würden demnach unterschiedliche Instanzen des Services verwenden (*Hierarchical injectors* o.D.).

5. Anforderungsanalyse

In diesem Teil der Arbeit soll die Theorie in einem praktischen Beispiel angewendet werden. Bei der praktischen Umsetzung einer PWA gibt es mehrere Möglichkeiten, um die Merkmale und Eigenschaften zu erfüllen. Mit Hilfe der beispielhaft erstellten Organisator App sollen mehrere Umsetzungsmöglichkeiten präsentiert werden. Einige Code-Beispiele wurden in den vorherigen Kapiteln bereits vorgeführt. In diesem Beispiel soll nun das gesamte Vorgehen der technischen Umsetzung einer PWA erklärt und ein Vergleich der Herangehensweise zum Entwickeln einer PWA aufgezeigt werden. Bevor das Projekt geplant werden kann müssen zunächst die Anforderungen bekannt sein. Für die Ermittlung der Anforderungen, müssen diese zunächst ermittelt und definiert werden. Ist die Auflistung der Anforderungen vollständig, wird eine Struktur mit den Abhängigkeiten und Zusammengehörigkeiten erstellt. Zum Schluss müssen diese wiederum auf Korrektheit, Machbarkeit Notwendigkeit, Priorisierung und Nutzbarkeit geprüft werden. Hierzu wird zunächst ein Anwendungsfalldiagramm erstellt.

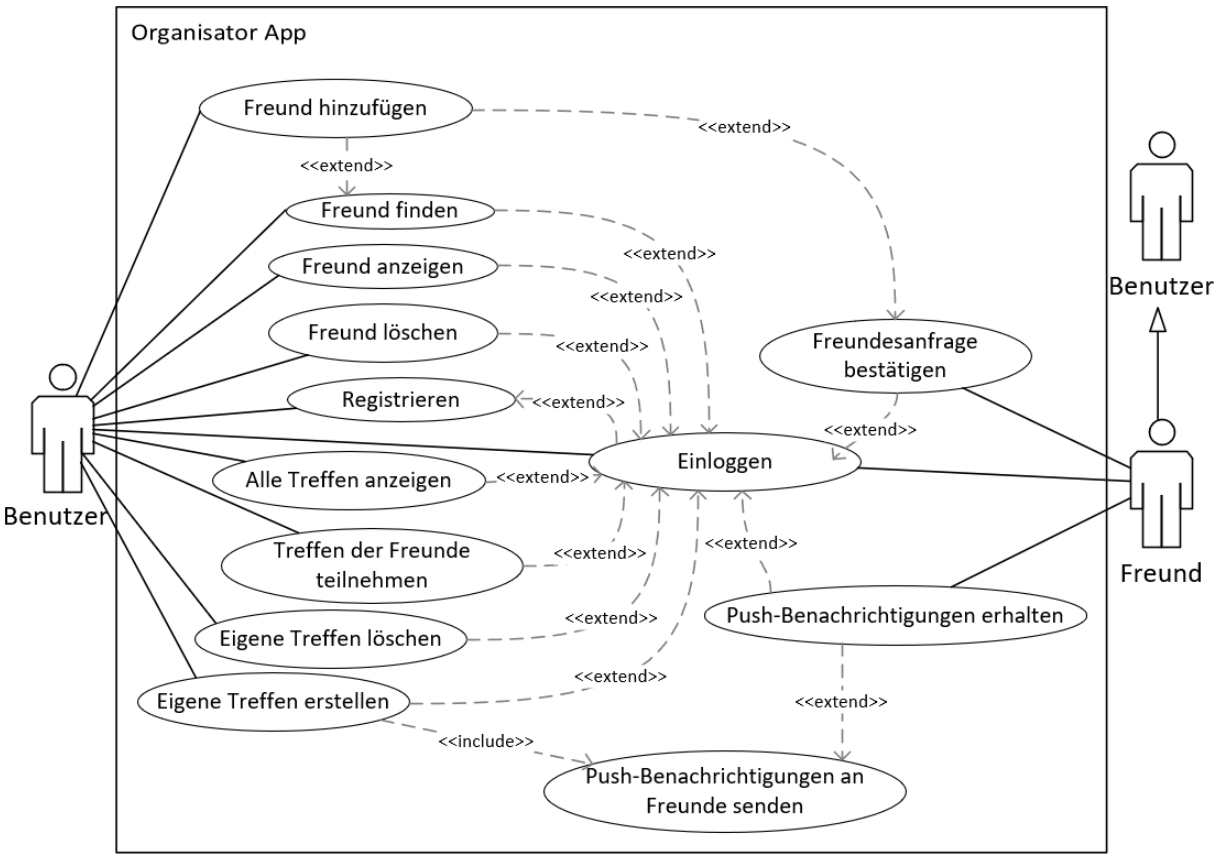

Abbildung 5.1.: Beispiel - Anwendungsfalldiagramm

Für eine detaillierte Beschreibung werden die **Funktionalen Anforderungen** mit präzisen Worten benannt.

Funktionale Anforderungen	
ID	**Anforderung**
Statik des Systems	
S0	Die PWA muss eine Seite mit folgende Anforderungen haben: S3, S4, S5, D1, D2, L1, L2, L3.
S1	Die PWA muss einen Bereich mit folgende Anforderungen besitzen: S6, S7, S8, S9, S5, D3, D4, L4, L5, L6, L7.
S2	Die PWA muss einen Bereich mit folgende Anforderungen besitzen: S10, S11, S12, S5, D5, D6, D7, L8, L9, L10, L11, L12.
S3	Die PWA muss die Möglichkeit bieten in einem Formular die E-Mail und das Passwort anzugeben.
S4	Die PWA muss die Möglichkeit bieten in einem Formular die E-Mail, das Passwort, eine Bestätigung des Passwortes, den Vor- und Nachnamen anzugeben.
S5	Die PWA muss die Möglichkeit geben ein Formular zu bestätigen.
S6	Die PWA muss die Möglichkeit geben selbst erstellte Treffen anzuzeigen.
S7	Die PWA muss die Möglichkeit geben Treffen von Freunden anzuzeigen.
S8	Die PWA muss die Möglichkeit geben an Treffen von Freunden teilzunehmen.
S9	Die PWA muss die Möglichkeit geben in einem Formular den Ort, die Aktivität, das Datum und die Uhrzeit anzugeben.
S10	Die PWA muss die Möglichkeit geben alle Freundschaftsanfragen anzuzeigen.
S11	Die PWA muss die Möglichkeit geben alle bestätigten Freunde anzuzeigen.
S12	Die PWA muss die Möglichkeit bieten in einem Formular den Vor- und Nachnamen anzugeben.
Dynamik des Systems	
D0	Die PWA muss fähig sein zu den Bereichen S1 und S2 zu gelangen.
D1	Die PWA muss die Möglichkeit geben zwischen den Formularen S3 und S4 zu wechseln.
D2	Die PWA muss fähig sein nach dem erfolgreichen einloggen auf den Bereich S1 weiterzuleiten.
D3	Die PWA muss fähig sein das Formular S9 in einem Overlay anzuzeigen.
D4	Die PWA muss fähig sein das Formular S9 mit dem Overlay auszublenden.
D5	Die PWA muss fähig sein das Formular S12 in einem Overlay anzuzeigen.
D6	Die PWA muss fähig sein das Formular S12 mit dem Overlay auszublenden.

D7	Die PWA muss fähig sein die Ergebnisse der Freunde-Suche durch Bestätigung des Formulars S12 anzuzeigen.
Logik des Systems	
L0	Die PWA muss einem nicht eingeloggten Benutzer folgende Funktionen verweigern: S1, S2.
L1	L1: Die PWA muss dem Benutzer die Möglichkeit geben sich mit den Angaben aus S3 einzuloggen.
L2	Die PWA muss einem eingeloggten Benutzer die Möglichkeit geben sich auszuloggen.
L3	Die PWA muss dem Benutzer die Möglichkeit geben sich mit den Angaben aus S4 zu registrieren.
L4	Die PWA muss fähig sein mit den Angaben aus S9 ein Treffen zu erstellen.
L5	Die PWA muss dem eingeloggten Benutzer die Möglichkeit geben selbst erstellte Treffen zu entfernen.
L6	Die PWA muss dem eingeloggten Benutzer die Möglichkeit geben an den Treffen der Freunde teilzunehmen.
L7	Die PWA muss fähig sein nach der Teilnahme eines Treffens eine weitere Teilnahme zu verwehren.
L8	Die PWA muss einem eingeloggten Benutzer die Möglichkeit geben andere Benutzer als Voraussetzung für L9 zu finden.
L9	Die PWA muss einem eingeloggten Benutzer die Möglichkeit geben andere Benutzer eine Freundschaftsanfrage zu senden.
L10	Die PWA muss einem eingeloggten Benutzer die Möglichkeit geben Freundschaftsanfragen zu bestätigen.
L11	Die PWA muss beim Bestätigen der Freundschaftsanfrage beide Benutzer als Freunde betrachten.
L12	Die PWA muss einem eingeloggten Benutzer die Möglichkeit geben eingetragene Freunde zu entfernen.

Tabelle 5.1.: Beispiel - Funktionale Anforderungen

Mit Hilfe der Ids wird hierbei die Strukturierung der Anforderungen durchgeführt. Die Tabelle zeigt jene Anforderungen, die auf die Anwendung bezogen sind. Darüber hinaus soll die Anwendung jedoch noch weitere Anforderungen erfüllen. Die **nicht funktionalen Anforderungen** sind wie folgt definiert:

- **Sicherheit:** Der Schutz der Daten und das Absichern der Anwendung gegen bekannte Angriffsmethoden soll gewährleistet sein.
- **Erweiterbarkeit:** Die Anwendung soll durch Services und Komponenten mit wenig Aufwand erweitert werden können.
- **Verständlichkeit:** Auch für die Erweiterbarkeit soll das Konzept verständlich konstruiert sein.
- **Fehlertolerant:** Die Anwendung soll bei Software-Fehlern nicht zum Absturz kommen.

6. Projektplanung und Architektur der Anwendung

Um die Merkmale einer PWA umsetzen zu können müssen diese bereits in der Planung integriert werden.

6.1. Systementwurf - Architektur

Der Systementwurf der Anwendung ist entscheidend für die Qualität des Endergebnisses. Deswegen ist es wichtig zu wissen, wie das Endergebnis aussehen soll. Bereits in der Planung können Probleme erkannt werden, welche in der Programmierphase Zeit kosten würden. Für einen guten Systementwurf sollten bekannte Prinzipien eingehalten werden. Moderne Frameworks setzen in der Regel bereits in ihrem Konzept Entwurfsmuster und Architekturschichten um, mit welchen die Webanwendung implementiert werden sollten. Da die Entscheidung des Frameworks bereits getroffen ist, muss diese auch im Entwurf bedacht werden.

Im Anschluss muss eine passende Architektur ausgewählt werden. Für dieses Beispiel reicht eine Drei-Schichten-Architektur aus. Diese besteht erstens aus einer Anwendungsschicht, indem der Webclient realisiert werden soll. Zweitens aus einer Logikschicht, in der die Anwendungslogik definiert ist. Drittens aus einer Datenschicht in der die Daten zur persistenten Speicherung abgelegt werden sollen. Grundlegend gilt bei einer Schichtenarchitektur, dass eine übergeordnete nicht von der untergeordneten Schicht abhängen darf (Dependency-Inversion-Prinzip). Eine Besonderheit einer PWA bei der Architektur stellt der Service Worker dar. Dieser überträgt nämlich mit seinem Aufleben die Funktion der Anwendungslogik auf den Webclient. Der Webserver ist an dieser Stelle nur noch für das Updaten des Service Worker zuständig. Der Webclient kann demzufolge selbstständig agieren und handeln.

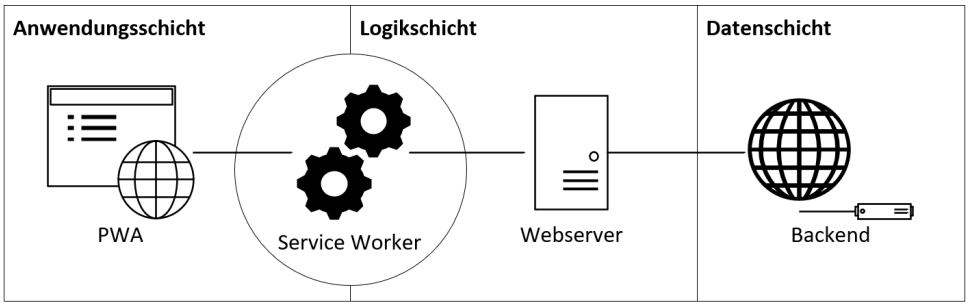

Abbildung 6.1.: Beispiel - Architektur

Die Anwendung lässt sich in weiteren Schichten aufteilen. Bereits bekannt ist, dass Angular als Framework verwendet werden soll. Vereint mit den Anforderungen ergeben sich also weitere Schichten.
Wie in der Grafik zu sehen ist realisiert das Framework einige Schichten auf der Anwendungsebene.

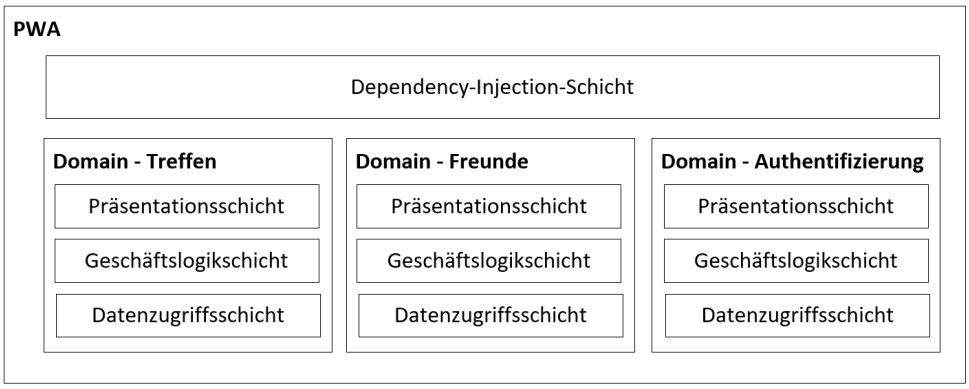

Abbildung 6.2.: Beispiel - Anwendungsschicht

6.2. Frontend

Für die Planung des Frontends soll ein Layout erstellt werden, welches ein Responsive Design besitzt. Dazu sollen Lösungen für die Barrierefreiheit in den Entwurf integriert werden. Für die Planung sollen ebenfalls die Komponenten, Services, et cetera der Module bestimmt werden.

6.2.1. Module

Nachdem die PWA in ihren Schichten eingeteilt ist, können die einzelnen Module der PWA geplant werden. Mit Blick auf Angular sollten die bereits vorgestellten Richtlinien von Abschnitt 4.3 zu den einzelnen Typen von Feature Modules eingehalten werden.

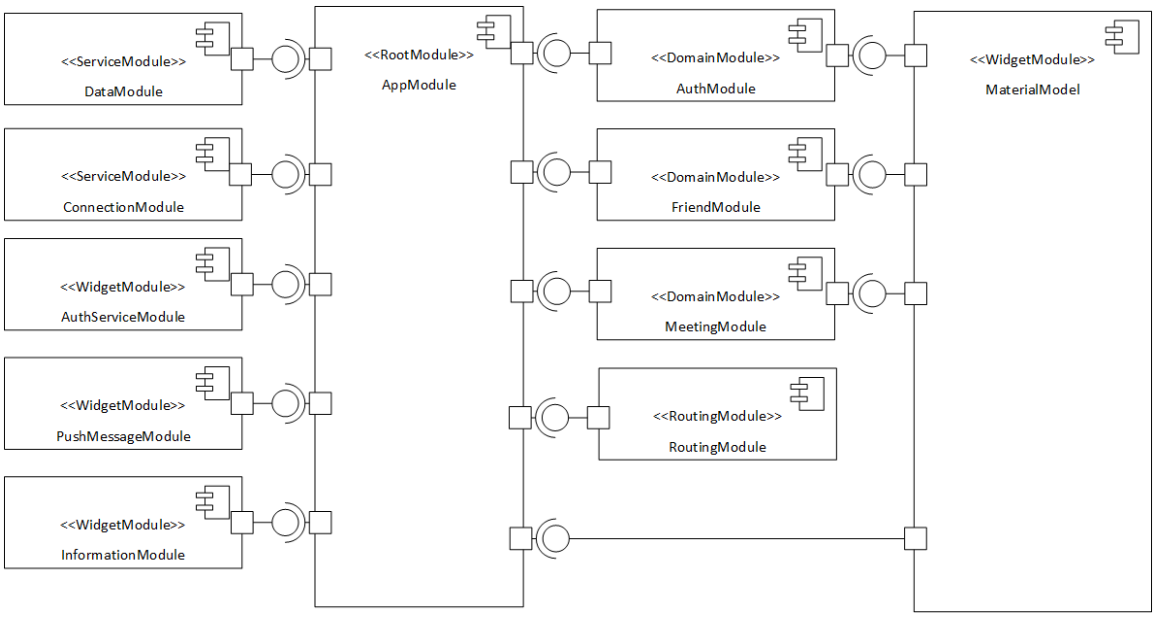

Abbildung 6.3.: Beispiel - Modulplan

Erläuterung der Module:

- **RoutingModule:** Navigiert zwischen den einzelnen Seiten in der PWA.
- **AppModule:** Ist das Hauptmodul der PWA in Angular.
- **AuthModule:** In diesem Modul soll eine Authentifizierung umgesetzt werden. Es soll das Ein- und Ausloggen sowie eine Registrierung realisieren.
- **AuthServiceModule:** Authentifizierung eines Benutzers. Das Modul kann Benutzer einloggen, ausloggen und registrieren.
- **DataModule:** Für das Zugreifen der Daten ist das DataModule zuständig. Dieses soll den lokalen und den externen persistenten Speicher organisieren.
- **FriendModule:** Das FriendModule ist für das Management der Freunde zuständig. In diesem sollen Freunde angezeigt, hinzugefügt und entfernt werden können.
- **MeetingModule:** Im MeetingModule sollen die Treffen der Freunde des Benutzers angezeigt werden. Es soll an den Treffen teilgenommen werden können. Über dieses Modul soll es die Möglichkeit geben eigene Treffen zu organisieren und zu entfernen.
- **InformationModule:** Zur Laufzeit soll es möglich sein dem Benutzer Meldungen anzuzeigen. Beispielsweise das Fehlschlagen eines Logins oder das erfolgreiche Registrieren. Die Meldung soll von jedem Kontext der Webanwendung erreichbar sein.
- **MaterialModule:** Innerhalb dieses Modules sollen individuelle UI-Elemente zur Verfügung gestellt werden. So zum Beispiel Textboxen, Buttons oder Overlays.
- **PushMessageModule:** Um das Merkmal „Nutzer bindend“ zu erfüllen sollen Push-Benachrichtigungen empfangen werden können. Dieses Modul ist für die Registrierung des Push-Services und dem verarbeiten von Push-Benachrichtigungen zuständig.
- **ConnectionModule:** Informationen über den Onlinestatus der Anwendung sollen über das ConnectionModule umgesetzt werden.

6.2.2. Entwurf der App (Mockups)

Bei der Erstellung eines Layouts, welches für alle Geräte und Bildschirmgrößen angepasst sein soll, macht es Sinn mit einem Mobile-Format zu beginnen. Hierbei spricht man von dem Mobile-First-Konzept. Dies liegt daran, dass mobile Endgeräte weniger Platz für die Strukturierung besitzen und ein Tippen mit dem Finger eine größere Fläche einnimmt, als der Cursor am Computer beim Aktivieren oder Fokussieren einer Schaltfläche. Nachdem das Layout für Smartphones erstellt ist können diese in ein Layout für Tablets und Computer umgesetzt werden.

Für die Authentifizierung soll nach den Anforderungen eine eigene Webseite existieren. Diese soll wie in der nachfolgenden Abbildung 6.4 aussehen (oder Desktop-Version: Abbildung B.1 im Anhang).

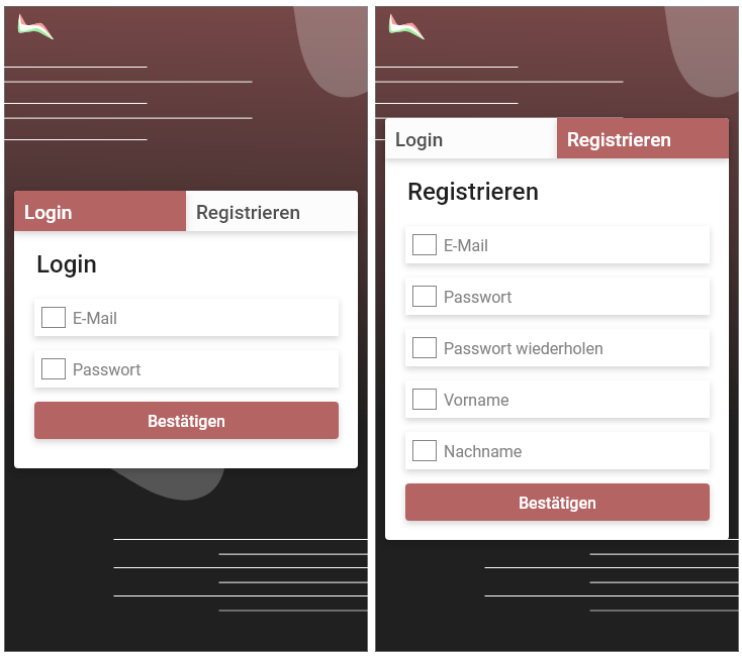

Abbildung 6.4.: Beispiel - Layout Authentifizierung

Innerhalb der Textboxen sind Vierecke eingezeichnet. Diese stehen stellvertretend für Icons, welche passend des Inputs ausgewählt werden sollen. Nach dem Login stehen dann folgende Bereiche, wie in Abbildung 6.5 zu sehen, in der PWA zur Verfügung. Für die Darstellung auf dem Computer fiel die Entscheidung beide Bereiche „Freunde“ und „Treffen“, auf einer Seite anzuzeigen (Abbildung B.4 im Anhang). Die Entscheidung fiel aus optischen Gründen und um die PWA übersichtlicher aufzubauen. In der Smartphone-Ansicht (Abbildung 6.5) sollen diese hingegen über eine Navigation einzeln angezeigt werden.

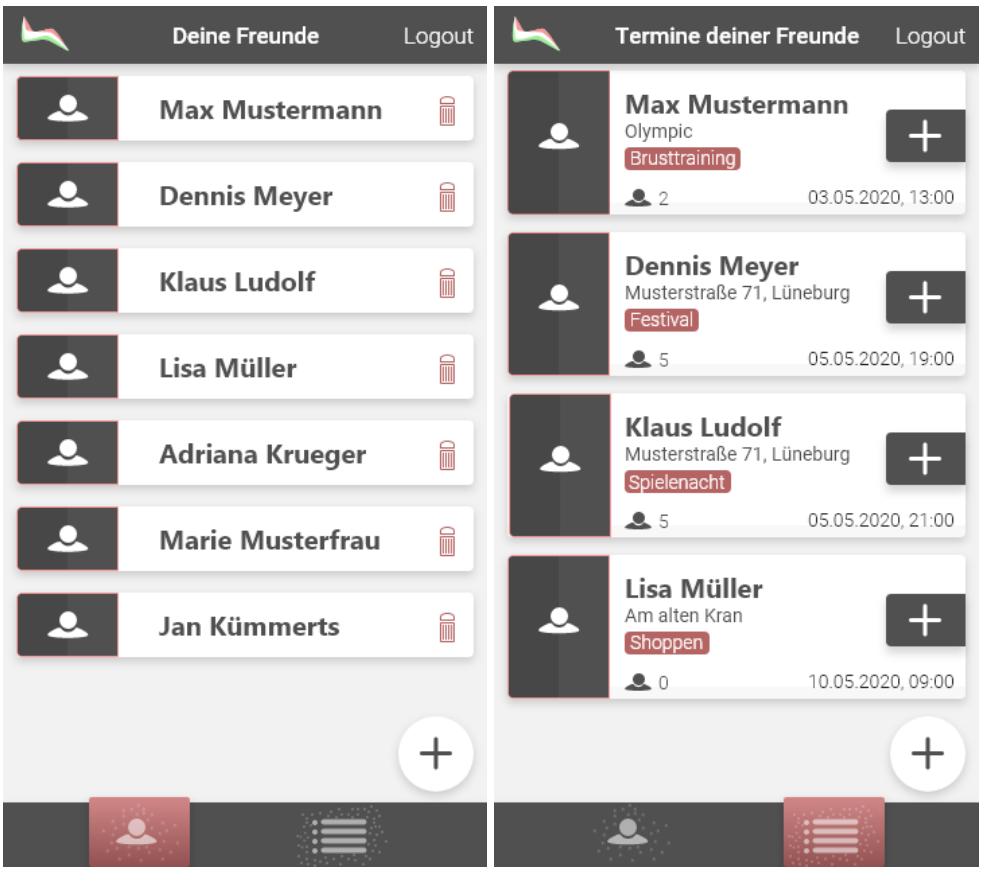

Abbildung 6.5.: Beispiel - Layout Freunde und Termine/Treffen

Für die Barrierefreiheit ist der Kontrast so gewählt, dass auch Menschen mit einer Sehschwäche oder ähnlichen Einschränkungen die Schriften erkennen können.

Layout Erläuterung: In Abbildung 6.4 kann sich ein Benutzer registrieren oder einloggen. Oberhalb der weißen Karte kann zwischen einloggen und registrieren gewechselt werden. Mit bestätigen des Buttons „Bestätigen“ findet die Authentifizierung statt. Ist diese erfolgreich verlaufen, wird nach dem Einloggen zu den Bereich in Abbildung B.4 weitergeleitet. In der Navigation der Smartphone-Ansicht kann zwischen den Bereichen „Termine deiner Freunde“ und „Deine Freunde“ gewechselt werden. Zum Ausloggen kann im Bereich der Überschrift der Button „Logout“ betätigt werden. Über den Mülleimer können Freunde und Termine/Treffen entfernt werden. Zum Hinzufügen eines Freundes wird auf den Hover-Button unten-rechts in dem Freundes-Bereich geklickt, woraufhin sich das Overlay aus Abbildung 6.6 öffnet (oder Desktop-Version: Abbildung B.2 im Anhang). In diesem können Benutzer über den Vor- und Nachnamen gefunden und über den Plus-Button hinzugefügt werden. Um ein Treffen/Termin zu erstellen wird der Hover-Button im Bereich der Termine geklickt. Es öffnet sich das Overlay aus Abbildung 6.6 (oder Desktop-Version: Abbildung B.3 im Anhang).

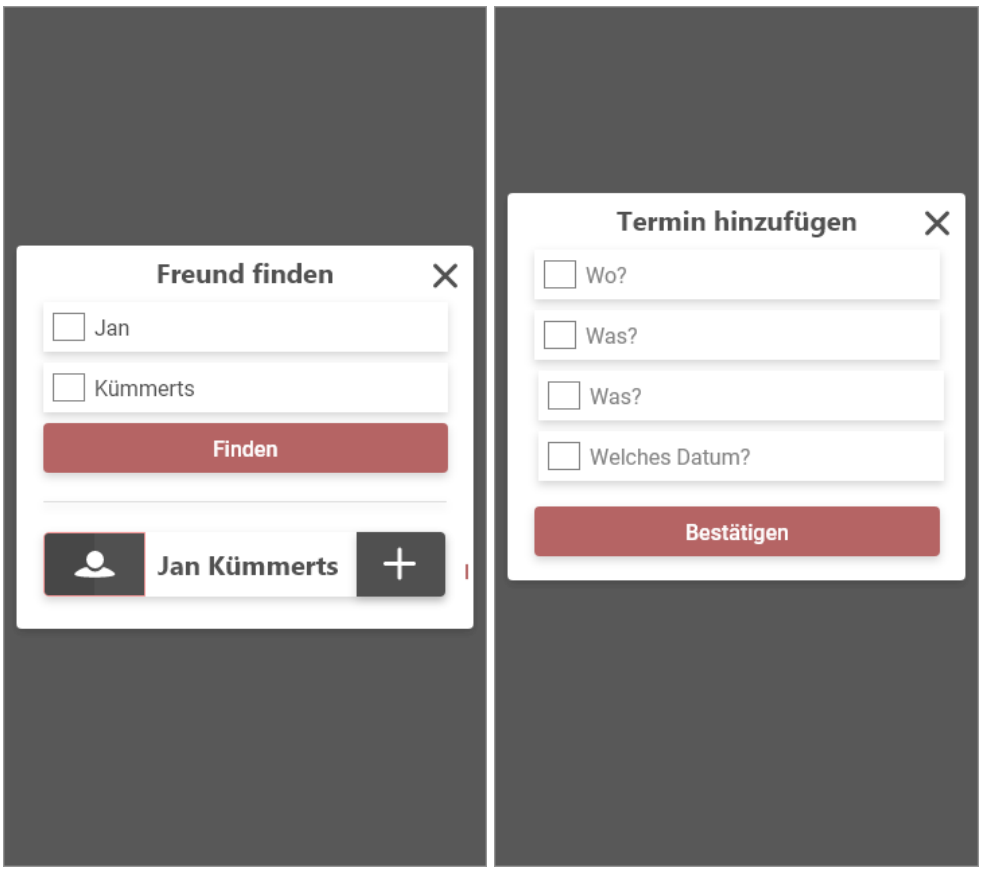

Abbildung 6.6.: Beispiel - Layout Overlay: Freund hinzufügen (links), Termin erstellen (rechts)

Hier müssen lediglich die nötigen Eingaben zum Erstellen eines Treffens/Termins eingetragen werden und über den Button bestätigt werden.
Mit Blick auf Prinzipien sollten bei der Erstellung des Layouts bereits wiederholende Elemente als Komponenten abgespeichert werden („Don‘trepeat yourself“). In Abbildung 6.7 sind nun die UI-Komponenten aufgezeigt.

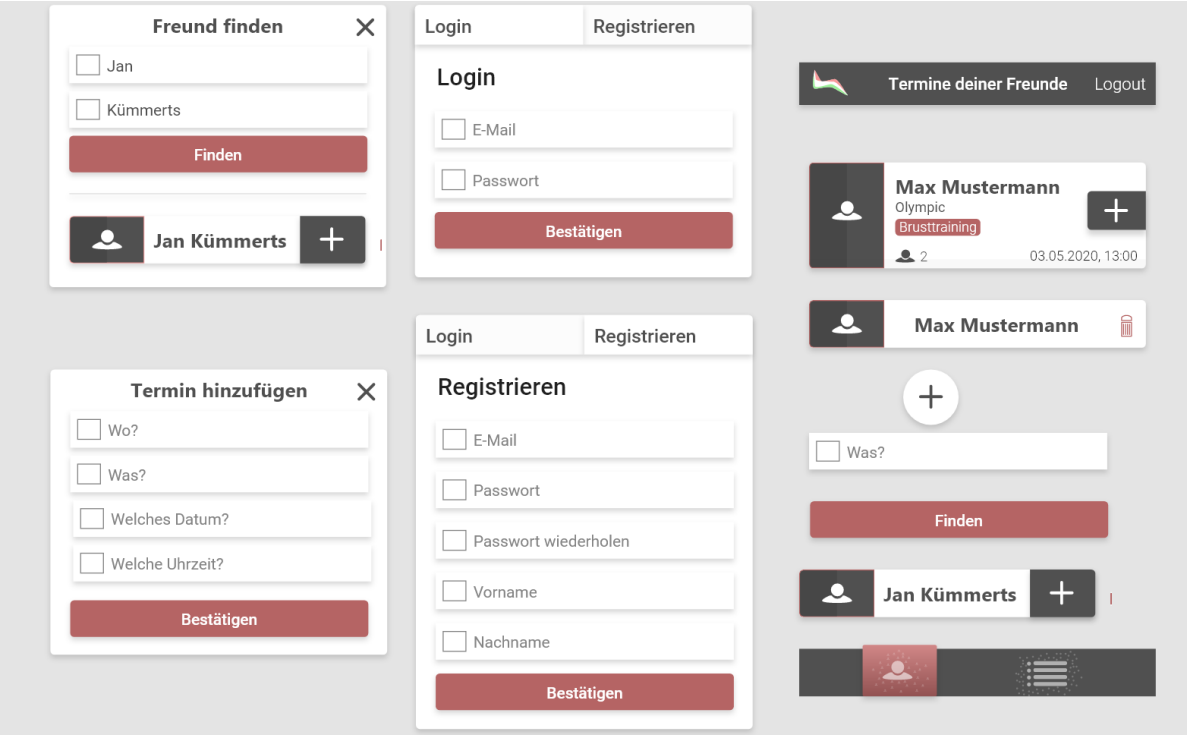

Abbildung 6.7.: Beispiel - UI-Komponenten

6.3. Backend - Datenzugriffe

Bei der Wahl des Datenbankmanagementsystems ist die Entscheidung auf „Firebase Cloud Firestor“ gefallen. Dies ist eine Dokumentenbasierte NoSQL Realtime Datenbank, welche als Cloud-Service angeboten wird.

Cloud: Eine Cloud ist ein Rechennetzwerk für die Bereitstellung von Rechenleistung, Speicher oder Software (Barabas o.D.). Auf diesen können unter anderem auch Webanwendungen ausgeführt werden. Hierbei wird von einer „Serverless Architecture“ gesprochen. Die Webanwendung ist also nicht auf einem definierbaren Server gespeichert, sondern befindet sich extern auf den Servern der Cloud. Dies hat den Vorteil, dass Wartungs- sowie Einrichtungsarbeiten am Server wegfallen und keine Hardware gekauft werden muss. Bei der Entwicklung können ebenfalls Cloud Services eingesetzt werden. Es gibt unterschiedliche Cloud Servicemodelle welche unter Barabas o.D. nachgelesen werden können.

NoSQL: NoSQL Datenbanken werden anders als bei relationalen SQL-basierten Datenbanken mit einer Dokumentenstruktur, Graphenstruktur oder Ähnlichem umgesetzt. Somit lassen sie sich unkomplizierter für die Anwendungsentwicklung nutzen. Das Problem SQL-basierter Datenbanken in der Anwendungsentwicklung ist häufig, dass die Daten zur Abspeicherung unstrukturiert oder in-

einander verschachtelt sind und dadurch über Tabellen mit Zeilen und Spalten schwer dargestellt werden können. So kann es bei der Entwicklung der Datenstruktur einer SQL-Datenbank zu einem Chaos von Abhängigkeiten und Referenztabellen kommen. Zusätzlich wird viel Zeit aufgewandt um die Objekte als Tabellen abzuspeichern oder umgekehrt. Mit NoSQL-Datenbanken wird das Verschachteln von Informationen zugelassen und bietet somit eine flexible Art Daten abzuspeichern. Besonders komplexere Datenstrukturen können übersichtlich umgesetzt werden. Performance-technisch sind NoSQL-Datenbanken ebenfalls den relationalen SQL-basierten Datenbanken in einigen Punkten voraus, da sie horizontal skalierbar sind. Das heißt, dass die Leistung einer NoSQL-Datenbank mit dem Hinzufügen von Hardware (zum Beispiel ein weiterer Server) erhöht werden kann. Infolgedessen können diese ideal für Big-Data-Anwendungen verwendet werden.

Realtime Datenbanken: Bei einer Realtime Database handelt es sich um eine Datenbank, welche in nahezu Echtzeit Änderungen der Daten in der Datenbank mit den Clients synchronisiert. Dies geschieht ohne eine erneute Anfrage des Clients. Wird also ein Eintrag getätigt, sendet die Datenbank diese Information an jene Webclients, welche momentan auf diese zugreifen. So bleiben diese stets aktuell.
Bei der Plattform handelt es sich hierbei um eine Plattform zum Entwickeln des Backends (Funktionen, Sicherheitsregeln, Trigger, et cetera). Zusätzlich bietet Firebase die Möglichkeit auf dem lokalen Rechner zu programmieren und den entsprechenden Code nachträglich hochzuladen. Firebase stellt für den Zugriff der Datenbank vom Client eine eigene Bibliothek für Angular zur Verfügung, die auch eine Offline-Unterstützung beinhaltet. Mit Hilfe von Emulatoren können Sicherheitsregeln und Zugriffe auf die Datenbank getestet werden. Zuletzt besitzt Firebase außerdem eine umfangreiche Dokumentation.
Für das Backend sollte eine Struktur der Daten definiert werden. Dies dient nicht nur der Übersicht, sondern ist auch nötig, um Sicherheitsregeln und Rollen festzulegen. Die Datenstruktur der Organisator App ist in Abbildung 6.8 zu sehen.

Erklärung von Abbildung 6.8: „Firebase Cloud Firestore“ organisiert Daten innerhalb der Datenbank mit Sammlungen von Dokumenten und Dokumenten. Um den Unterschied klar darzustellen sind Sammlungen in Grün eingezeichnet. Dokumente werden mit einem Kreis eingeleitet, wie in der Abbildung 6.8 definiert. Bei unmarkierten Texten handelt es sich um ein Feld des Dokuments. Jedes Dokument in „Firebase Cloud Firestore“ wird mit einer Id identifiziert. Diese kann manuell oder automatisch definiert werden. Bestenfalls sollten Datensätze hierbei in mehrere Sammlungen aufgeteilt werden, um die Datenabfragen von Datensätzen performant zu halten.
Als Hauptsammlung sollen alle User aufgelistet werden. Diese haben Informationen, welche für alle eingeloggten Benutzer zur Verfügung gestellt werden sollen („publicInfo“). Es gibt aber auch Informationen, wie die Subscription für Push-Benachrichtigungen, die nicht öffentlich einsehbar sein sollen („privateInfo“). Für die Speicherung der Freunde werden die Ids dieser in der Sammlung „friends“,

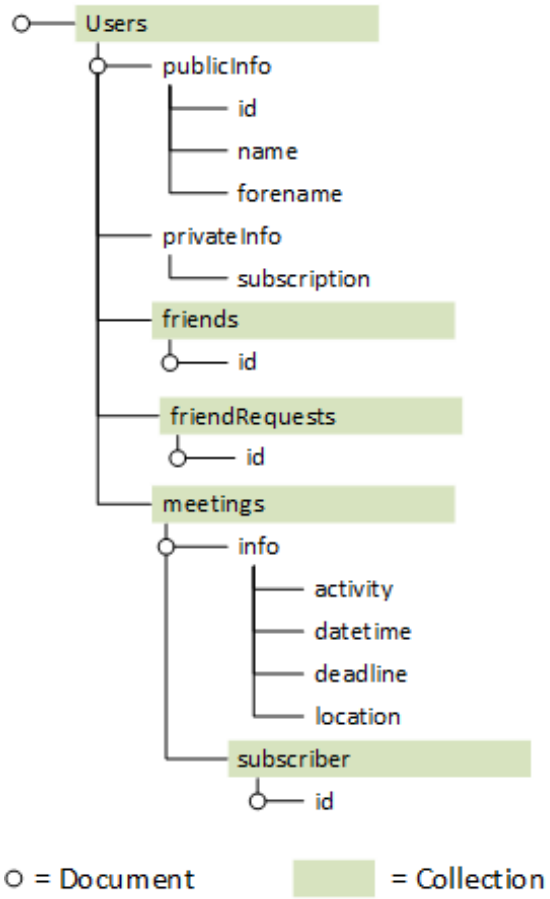

Abbildung 6.8.: Beispiel - Datenstruktur

und Freundesanfragen in der Sammlung „friendRequests“ aufgelistet. Da ein Benutzer auch eigene Termine/Treffen eintragen kann, muss es auch eine Sammlung für Meetings geben. In „Info“ können alle Informationen des Treffens/Termins angegeben werden. Sollten andere Benutzer an dem Meeting teilnehmen wollen, sollen diese sich in der Sammlung „subscriber“ eintragen können.
Nun müssen die Zugriffsrechte und Rollen festgelegt werden.

Rolle	**Definition**
Owner	Ist auch ein User, welcher auf seine eigenen Daten zugreift.
Friend	Ist auch ein User, welcher auf die Daten eines Users zugreift, der diesen in der Sammlung "friends"hinzugefügt hat.
User	Ist ein Benutzer, welcher in der Sammlung „users“ enthalten ist.

Tabelle 6.1.: Beispiel - Datenbank-Rollen

Ort	**Rolle**	**Zugriff**		
		lesen	**erstellen**	**löschen**
publicInfo	User	ja	nein	nein
friends	Owner	ja	Nur wenn die Id in der Sammlung friend-Requests enthalten ist	ja
	Friend	nein	nein	nein
	User	nein	Nur wenn die Id in der Sammlung friends enthalten ist	Nur die eigene Id
friendRequests	Owner	ja	nein	ja
	Friend	nein	nein	nein
	User	nein	Nur die eigene Id	Nur die eigene Id
meetings	Owner	ja	ja	ja
	Friend	ja	nein	nein
	User	nein	nein	nein
subscriber	Owner	ja	nein	nein
	Friend	ja	Nur die eigene Id	Nur die eigene Id
	User	nein	nein	nein

Tabelle 6.2.: Beispiel - Datenbank-Rechte

7. Fallbeispiel

7.1. Backend

Für die Implementierung des Backends müssen lediglich die definierten Sicherheitsregeln und das Versenden der Push-Benachrichtigungen umgesetzt werden.

7.1.1. Die Sicherheitsregeln definieren

Die Sicherheitsregeln können entweder über die Webseite von Firebase Cloud Firestore oder auf dem lokalen Computer entwickelt werden. Um später die Sicherheitsregeln und Backend-Funktionen testen zu können, ist das Einrichten der Emulator Suite auf dem lokalen Computer zu empfehlen. Danach können innerhalb der „firestore.rules“-Datei können nun die Sicherheitsregeln definiert werden. Für das Verständnis wird die Syntax von Firebase Cloud Firestore Security Rules und erstellte Funktionen erklärt.

```
1 rules_version = '2';
2 service cloud.firestore {
3
4   match /databases/{database}/documents {
```

Listing 7.1: firebase.rules - Documents

Syntax: Als Erstes muss die Version der Firebase Cloud Firestore Security Rules angegeben werden. Mit dem Befehl „match“ wird der Ort angegeben, für welchen die folgenden Sicherheitsregeln definiert werden sollen. „{database}“ wird in Firebase Cloud Firestore „wildcard“ genannt, wobei es sich um ein Dokument handelt. In diesem Fall wird hier die Projektidentifikation vom erstellten Google-Projekt eingefügt. Dokumente sind wie bereits erwähnt Teil einer Sammlung. In Listing 7.1 ist also „databases/“ eine Sammlung.

```
1     function isUser() {
2       return  request.auth.uid != null
3       &&      exists(/databases/$(database)/documents/users/$(request.auth.
            uid));
4     }
5
6     match /users/{userId} {
7
8       function isOwner() {
9         return isUser() && request.auth.uid == userId;
10      }
11
12      function isFriend() {
13        return isUser() && exists(/databases/$(database)/documents/users/$(
              userId)/friends/$(request.auth.uid));
```

```
14        }
```

Listing 7.2: firebase.rules - Rollen

Syntax: Mit „request.auth.uid“ wird die Benutzer-Id abgefragt. Hierbei handelt es sich um einen vertrauenswürdigen Wert, da dieser nicht vom Client, sondern von Firebase selbst definiert wird. Die Funktion „exist(«Dokument»)“ wird von Firebase zum Kontrollieren der Existenz eines Dokuments verwendet. Als Rückgabe wird ein Boolean zurückgegeben. Innerhalb des Pfades der „exist“-Funktion können mit „$(«Variable»)“ Variablen angegeben werden. Als Variablen können hier auch Wildcards verwendet werden. Für die Definition der Rollen werden in Firebase Cloud Firestore Funktionen verwendet.
Funktion: In diesem Code-Teil werden die einzelnen Rollen definiert. Ein User muss in der Liste der Users enthalten sein. Der Owner muss die selbe Id des Dokuments in der User-Sammlung besitzen, um sich als solcher zu identifizieren. Der „Friend“ muss in der Freundes-Sammlung des Dokumentes der User-Sammlung enthalten sein, um sich als dieser zu identifizieren.

```
1         allow read:
2           if isUser();
```

Listing 7.3: firebase.rules - User Regeln

Syntax: Eine Regel wird mit „allow «Zugriff» if «Bedingung»;“ definiert. Als Zugriff ist der lesende oder schreibende Zugriff gemeint. Lesende Zugriffe sind zum Beispiel:

- „get“: Dokument lesen
- „list“: Sammlung lesen
- „read“: Alle vorher genannten lesende Zugriffe

Schreibende Zugriffe hingegen:

- „create“: Erstellen eines Dokumentes
- „delete“: Entfernen eines Dokumentes
- „update“: Verändern eines Dokumentes
- „write“: Alle vorher genannten schreibenden Zugriffe

Funktion: Um auf die „publicInfos“ des Dokumentes zuzugreifen, muss dieser die Rolle „User“ erfüllen.

```
1         match /friends/{friendId} {
2
3           function isCurrentFriend() {
4             return request.auth.uid == friendId;
5           }
```

```
6
7           function requestedFriendDataValid() {
8             return  request.resource.data.keys().size() == 1
9             &&      request.resource.data.keys().hasAll(['id'])
10            &&      request.resource.data.id  is string
11            &&      request.resource.data.id == friendId;
12          }
13
14          function approvedFriendDataValid() {
15            return  request.resource.data.keys().size() == 1
16            &&      request.resource.data.keys().hasAll(['id'])
17            &&      request.resource.data.id  is string
18            &&      request.resource.data.id == request.auth.uid;
19          }
20
21          function isRequested() {
22            return  exists(/databases/$(database)/documents/users/$(userId)/
                  friendRequests/$(request.resource.data.id));
23          }
24
25          function isApproved() {
26            return  exists(/databases/$(database)/documents/users/$(request.
                  auth.uid)/friends/$(userId));
27          }
```

Listing 7.4: firebase.rules - Freunde Funktionen

Syntax: Daten, welche zum Beispiel in die Datenbank hinzugefügt werden sollen besitzen immer einen Key, der die Identifikation der Information ermöglichen soll sowie der Bereitstellung der Information selbst. Um eine Validierung und Prüfung der Daten zu ermöglichen, können Funktionen von Firebase Cloud Firestore Security Rules verwendet werden. Die Anzahl an Informationen kann über die Anzahl der Keys bestimmt werden („request.resource.data.keys().size()“). Um zu prüfen, ob die Key-Bezeichnungen korrekt sind und auch keine Informationen unter einem anderen Key abgespeichert werden, wird die Funktion „request.resource.data.keys().hasAll([...])“ verwendet. Innerhalb des Array-Parameters von hasAll können die Felder angegeben werden, welche als Informationen im Dokument abgespeichert werden müssen. Auch „hasAny()“ oder „hasOnly“ können verwendet werden. „hasAny()“ ist erfolgreich, sobald die zu prüfende Liste einen Wert der Liste als Parameter enthält. „hasOnly“ ist erfolgreich, wenn jeder Wert der zu Prüfenden Liste in der Liste als Parameter enthalten ist. Der Datentyp kann mit „request.resource.data.«Key» is «Datentyp»“ definiert werden.
Funktion: Um zu prüfen, dass ein „Friend“ auch nur auf sein eigenes Dokument zugreifen kann, muss dieser mit der Wildcard verglichen werden. Für die Einheitlichkeit werden in diesem Beispiel Funktionen mit dem Namensende „...Valid“ für das Prüfen der Daten auf Gültigkeit und Richtigkeit verwendet (Zeile 7 und Zeile 14). Das heißt, dass diese die Daten auf die richtige Anzahl an Informationen zum Eintragen, die richtige Key-Bezeichnung, die richtigen Datentypen und die korrekte

Angabe der Id prüfen werden. Eine Freundschaftsanfrage ist „Requested“, wenn ein Eintrag mit der Id des Users in der Sammlung „friendRequests“ des Vater-Dokumentes enthalten ist (Zeile 21). Eine Freundschaftsanfrage ist „Approved“, wenn ein Eintrag mit der Id des Users in der Sammlung „friends“ des zugreifenden Users enthalten ist (Zeile 25).

```
1           allow read:
2             if  isOwner();
3
4           allow create:
5             if  isOwner()
6                 &&  isRequested()
7                 &&  requestedFriendDataValid()
8             ||  !isOwner()
9                 &&  isUser()
10                &&  isApproved()
11                &&  approvedFriendDataValid()
12
13          allow delete:
14            if  isOwner()
15            ||  isCurrentFriend();
16        }
```

Listing 7.5: firebase.rules - Friend Regeln

Syntax: Auch logische Operatoren, wie „UND“ oder „ODER“ können in der Bedingung verwendet werden.

Funktion: Die Freunde sollen nur von dem Owner selbst eingesehen werden können. Der Owner soll nur Freunde hinzufügen können, welche ihm eine Anfrage gestellt haben. Sollte der Owner einem anderen User eine Freundschaftsanfrage gesendet haben, kann dieser sich unter der Voraussetzung, dass dieser ihn auch in der Freundesliste hinzugefügt hat ebenfalls als Freund eintragen. Zusätzlich müssen die angegebenen Daten auf Gültigkeit und Richtigkeit geprüft werden. Der Owner hat die Möglichkeit jeden Freund aus der Liste zu entfernen. Ein Freund kann aber lediglich seinen eigenen Eintrag entfernen.

```
1         match /friendRequests/{requestFriendId} {
2
3           function isCurrentRequestFriend() {
4             return request.auth.uid == requestFriendId;
5           }
6
7           function friendRequestDataValid() {
8             return  request.resource.data.keys().size() == 1
9             &&      request.resource.data.keys().hasAll(['id'])
10            &&      request.resource.data.id  is string
11            &&      request.resource.data.id == request.auth.uid;
12          }
13
```

```
14          allow read:
15            if  isOwner();
16
17          allow create:
18            if  isUser()
19            &&  isCurrentRequestFriend()
20            &&  friendRequestDataValid()
21            &&  !isOwner()
22            &&  !isFriend();
23
24          allow delete:
25            if  isCurrentRequestFriend()
26            ||  isOwner();
27        }
```

Listing 7.6: firebase.rules - FriendRequests

Funktion: Auch hier soll nur der Owner Freundschaftsanfragen einsehen und nur der Owner oder der entsprechende Freund seinen eigenen Eintrag entfernen können. Freundschaftsanfragen soll hier aber lediglich der User erstellen können, kein Freund und auch nicht der Owner. Andernfalls könnte ein Benutzer selber Freundschaftsanfragen von anderen Benutzern erstellen. Und auch hier müssen die angegebenen Information auf Gültigkeit und Richtigkeit geprüft werden.

```
1         match /meetings/{meetingId} {
2
3           function meetingDataValid() {
4               return  request.resource.data.keys().size() == 1
5               &&      request.resource.data.keys().hasAll(['info'])
6               &&      request.resource.data.info.keys().size() == 4
7               &&      request.resource.data.info.keys().hasAll(['datetime', '
                    location', 'activity', 'deadline'])
8               &&      request.resource.data.info.datetime    is string
9               &&      request.resource.data.info.location    is string
10              &&      request.resource.data.info.activity    is string
11              &&      request.resource.data.info.deadline is timestamp;
12            }
13
14          allow read:
15            if  isOwner()
16            ||  isFriend();
17
18          allow create:
19            if  meetingDataValid()
20            &&  isOwner();
21
22          allow delete:
```

```
23            if  isOwner();
```

Listing 7.7: firebase.rules - Meetings

Funktion: Der Owner und die Freunde sollen die erstellten Treffen/Termine einsehen können. Aber nur der Owner soll seine eigenen Termine/Treffen erstellen und löschen können. Die Daten müssen dazu zur Erstellung geprüft und validiert werden.

```
1          match /subscriber/{subscriberId} {
2
3            function isCurrentSubscriber() {
4              return request.auth.uid == subscriberId;
5            }
6
7            function subscriberDataValid() {
8              return  request.resource.data.keys().size() == 1
9              &&      request.resource.data.keys().hasAll(['id'])
10             &&      request.resource.data.id  is string
11             &&      request.resource.data.id == subscriberId;
12           }
13
14           allow read:
15             if  isOwner()
16             ||  isFriend();
17
18           allow create:
19             if  isFriend()
20             &&  isCurrentSubscriber()
21             &&  subscriberDataValid();
22
23           allow delete:
24             if  isCurrentSubscriber();
25         }
26       }
27     }
28   }
29 }
```

Listing 7.8: firebase.rules - Meetings Subscriber

Funktion: Damit die Freunde an dem Termin/Treffen teilnehmen können, können diese sich in der Sammlung „subscriber“ eintragen. Außerdem sollen diese sich auch wieder aus der Liste austragen können. Der Owner und seine Freunde haben die Möglichkeit die Teilnehmer einzusehen.

7.1.2. Das Versenden von Push-Benachrichtigungen

Zunächst muss im Backend Firebase initialisiert sowie alle nötigen Bibliotheken eingebunden werden.

```
1 import * as functions from 'firebase-functions';
2 import * as admin from 'firebase-admin';
3
4 admin.initializeApp(functions.config().app.env);
```

Listing 7.9: Firebase Cloud Functions - Initialisierung

Hierfür werden zum einen die „firebase-functions“ benötigt, um die Implementierung des Backends zu realisieren. Zum anderen muss die „firebase-admin“-Bibliothek importiert werden um oberhalb der Sicherheitsregeln Zugriffe implementieren zu können. Die Initialisierung des Admins ist in Zeile Vier zu sehen. Wichtig an dieser Stelle ist, dass die Konfigurationen von Firebase als Environment-Variablen abgespeichert werden. Hierbei handelt es sich um sensible Informationen, welche ausdrücklich geheim gehalten werden müssen. Mit dem Befehl:

```
1 firebase functions:config:set <<service>>.<<Key>>="<<Konfiguration>>"
```

Listing 7.10: Manifest - Tag

können die Environment-Variablen in der Konsole definiert werden. Um das Versenden von Push-Benachrichtigungen zu realisieren muss dem Client zunächst die Möglichkeit gegeben werden sich mit seiner Push-Subscription in eine Liste des Push-Backends einzutragen. Hierfür soll eine Schnittstelle via HTTPS implementiert werden. Firebase ermöglicht dies über Firebase Cloud Functions.

```
1      const uid = context.auth?.uid;
2      if (uid && data) {
3          return admin.firestore()
4          .doc(`/pushEndpoint/${uid}`)
5          .set({
6              subscription: String(data)
7           });
8      } else {
9          console.log(uid + ', ' + data?.id);
10         throw new functions.https.HttpsError('failed-precondition', 'missing
               params! ');
11     }
12 })
```

Listing 7.11: Firebase Cloud Functions - Subscribe via HTTPS

Zunächst wird der Name der Funktion festgelegt und definiert, dass es sich hierbei um eine HTTPS-Funktion handelt. Für die Identifizierung wird in Zeile Zwei die Id des Benutzers gelesen. Beim „context“ handelt es sich um Informationen, welche von Firebase selbst automatisch an die Anfrage angehängt werden. Aus diesem Grund werden sie auch als vertrauenswürdig betrachtet. Besitzt der Benutzer eine Id und sendet dazu die Subscription mit, wird diese explizit als String eingetragen. Da die Daten über den Admin eingetragen werden, ist es wichtig, dass diese vorab geprüft werden und in diesem Fall als String deklariert werden. Andernfalls stellen diese Schnittstellen ein hohes Sicherheitsrisiko dar (zum Beispiel durch Injection). Schlägt der Eintrag fehl wird ein Error zurückgegeben

und dieser im Backend geloggt. Zum Versenden von Push-Benachrichtigungen werden sogenannte Trigger verwendet. Diese führen eine Funktion bei einer definierten Gegebenheit aus, ähnlich wie Event-Listener.

```
1  // Trigger => neue Freundesanfrage
2  export const newFriendRequest = functions.firestore
3  .document('users/{userId}/friendRequests/{requestId}')
4  .onCreate((snapshot, context) => {
5      return new Promise<void>((resolve, reject) => {
6          const data = snapshot.data();
7          // Parameter muessen valide sein
8          if (data && data.id && snapshot.id) {
9              // ID der Person, welcher ein Freund werden soll
10             const friendId: string = context.params.userId;
11             // Name des Benutzers, welcher eine Anfrage stellt
12             getFullname(String(data.id)).then(fullname => {
13                 console.log('id: ', snapshot.id, 'friendid: ', friendId, '
                       name: ', fullname);
14                 // Sende eine Notifikation an die Person, welche als Freund
                       angefragt wurde
15                 sendAddFriendNotification(data.id, friendId, fullname);
16                 resolve();
17             })
18             .catch(err => {
19                 // Wichtig => Reproduzierbarkeit: Fehler muessen
                       reproduzierbar sein
20                 console.error(err);
21                 reject();
22             });
23         } else {
24             console.error('no data! data: ' + data ? JSON.stringify(data) :
                   'undefined');
25             reject();
26         }
27     });
28 });
```

Listing 7.12: Firebase Cloud Functions - Trigger-Funktion

In diesem Fall soll bei einer Freundschaftsanfrage eine Push-Benachrichtigung an den angefragten Benutzer gesendet werden. Mit der Funktion onCreate wird beim Erstellen eines Dokumentes eine Callback-Funktion ausgelöst. Als Parameter wird der Snapshot, also die Daten, welche erstellt werden sollen und der Context übergeben. Nachdem alle Daten geprüft worden sind kann die Push-Benachrichtigung gesendet werden.

```
1  function sendAddFriendNotification(id: string, friendId: string, fullname:
       string): void {
2      const payload = {
```

```
3           data: {
4               id,
5               title: 'Freundschaftsanfrage',
6               body: fullname + ' moechte Sie als Freund hinzufuegen'
7           },
8           notification: {
9               title: 'Freundschaftsanfrage',
10              body: fullname + ' moechte Sie als Freund hinzufuegen'
11          }
12       };
13      admin.firestore()
14      .doc(`/pushEndpoint/${friendId}`)
15      .get().then(endpoint => {
16          const subscription = endpoint?.data()?.subscription;
17          if (subscription) {
18              console.log('Notification to Endpoint: ', subscription);
19              admin.messaging().sendToDevice(subscription, payload)
20              .then(() => {
21                  console.log('Notification sended to Endpoint: ',
                        subscription);
22              })
23              .catch(err => console.error(err));
24          } else {
25              console.error('no Subscription!')
26          }
27      })
28      .catch(err => console.log(err));
29  }
```

Listing 7.13: Firebase Cloud Functions - Push-Benachrichtigung senden

Hierfür wird zunächst die Payload definiert. Unter Data können jegliche Informationen übersendet werden, welche beim Client zum Verarbeiten benötigt werden. Bei der Definition der Daten für die Benachrichtigung („notification“) müssen hingegen die vordefinierten Keys von Firebase verwendet werden, damit diese im Browser korrekt angezeigt werden. In dieser Push-Benachrichtigung soll lediglich der Titel der Benachrichtigung und der Text angegeben werden. Alle verfügbaren Konfigurationen der Push-Benachrichtigung sind auf der Firebase Website unter *Firebase Cloud Messaging HTTP protocol* 2020 zu finden. Im nächsten Schritt muss die Subscription des Benutzers aus der Liste gesucht werden (Zeile 16). Ist diese vorhanden, wird eine Nachricht versendet (Zeile 19).

7.2. Frontend

Für die Implementierung der PWA werden die geplanten Module umgesetzt.

7.2.1. Die UI-Elemente erstellen (MaterialModule)

Zur Umsetzung der PWA sollen vorerst alle benötigten UI-Elemente erstellt werden, sodass sie später in den einzelnen Seiten platziert werden können. Hierfür soll die Logik der einzelnen Komponenten keine Verarbeitung der Eingaben vornehmen. Ziel der UI-Komponenten ist es lediglich, die Struktur und das Aussehen eines Elementes zu definieren. Eventuelle Ein- und Ausgaben werden über die Property Decoratoren kommuniziert.

```
1    @Input()
2    userInfos: IUserInfo[] = [];
3    @Output()
4    isVisibleCallback: EventEmitter<boolean> = new EventEmitter<boolean>();
```

Listing 7.14: UI-Komponenten - „MaterialModule/addfriend“ Property Decoratoren

Eingaben können direkt mit dem entsprechenden Datentyp definiert werden. Bei Ausgaben wird der EventEmitter verwendet. Über diesen können die Nachrichten synchron oder asynchron übermittelt werden. Mit der Subscribe-Methode des EventEmitters können die ausgehenden Informationen abgehört werden.

```
1      this.onCreate.emit(meeting);
```

Listing 7.15: UI-Komponenten - „MaterialModule/AddMeeting“ Emit via EventEmitter

Das Erstellen einer Komponente wird mit dem Befehl „ng generate component «Komponentenname»“ eingeleitet. Um eine Komponente dem Modul zuzuordnen, wird diese innerhalb des Class Decorators als Deklaration angegeben. Da das „MaterialModule“ ein Widget Funktions-Modul sein soll, muss die Komponente ebenfalls für den Export freigegeben werden.

```
1  @NgModule({
2    declarations: [
3      AddmeetingComponent,
4      ...
5    ],
6    entryComponents: [
7      AddmeetingComponent,
8      ...
9    ],
10   imports: [
11     ReactiveFormsModule,
12     ...
13   ],
14   exports: [
15     AddmeetingComponent,
16     ...
17   ]
18 })
```

Listing 7.16: UI-Komponenten - „MaterialModule“ Decorator

In Listing 7.16 wird die Overlay-Komponente “Addmeeting“ deklariert. Dynamische Komponenten, welche zur Laufzeit geladen und ausgeblendet werden, müssen als „entryComponent“ angegeben werden. Zusätzlich wird das „ReactiveFormsModule“ aus Angular importiert. Mit Reactive Forms wird ein Input-Stream der Eingaben erzeugt. Mit diesem können Daten live aus den Textboxen und anderen Eingabefeldern gelesen werden. Um Reactive Forms umzusetzen, muss zunächst innerhalb der Komponente eine „FromGroup“ erstellt werden.

```
1    meetingform = new FormGroup({
2      activity: new FormControl(''),
3      location: new FormControl(''),
4      date: new FormControl(''),
5      time: new FormControl('')
6    });
```

Listing 7.17: UI-Komponenten - „MaterialModule/AddMeeting“ FormGroup

In dieser wird der Name der Form und der Name der Eingabe-Felder definiert. Diese müssen nun noch innerhalb der View (DOM) den entsprechenden Elementen zugeordnet werden.

```
1    <form class="form" [formGroup]="meetingform" (ngSubmit)="create()">
2      <app-textbox [icon]="'SVGs/location.svg'" [type]="'text'"
           formControlName="location">Wo?</app-textbox>
3      ...
```

Listing 7.18: UI-Komponenten - „MaterialModule/AddMeeting“ Struktur

Die Zuordnung geschieht über Data Binding. Wie in Listing 7.18 zu sehen ist, handelt es sich bei der Textbox um ein selbst erstelltes Input-Feld. Damit dieses innerhalb der Reactiv Form auch als solches funktioniert, müssen innerhalb der Textbox-Komponente einige Einstellungen und Implementierungen getätigt werden.

Zunächst muss innerhalb des Decorators der entsprechende Provider hinzugefügt werden.

```
1    providers: [
2      {
3        provide: NG_VALUE_ACCESSOR, multi: true,
4        useExisting: forwardRef(() => TextboxComponent),
5      }
6    ]
```

Listing 7.19: UI-Komponenten - „MaterialModule/Textbox“ Decorator

Der „NG_VALUE_ACCESSOR“ ermöglicht in Angular das Binding einer Komponente.

```
1  export class TextboxComponent implements OnInit, ControlValueAccessor {
```

Listing 7.20: UI-Komponenten - „MaterialModule/Textbox“ Implements

Ebenfalls wird über diese Klasse der Komponente der „ControlValueAccessor“ implementiert. Dieser ist die Schnittstelle zur Angular Forms API. Die Implementierung der Methoden bietet folgende Möglichkeiten:

```
1    writeValue(obj: any): void
```

Listing 7.21: UI-Komponenten - „MaterialModule/Textbox“ Implementierung von „ControlValueAccessor“ writeValue

Ermöglicht das Registrieren einer Callback-Funktion, welche beim Verändern eines Wertes ausgelöst wird. Diese Methode ist für das Aktualisieren der Form, wenn Eingaben von der View eingehen.

```
1    registerOnChange(fn: any): void
```

Listing 7.22: UI-Komponenten - „MaterialModule/Textbox“ Implementierung von „ControlValueAccessor“ registerOnChange

Ermöglicht das Registrieren einer Callback-Funktion, welche von der Forms API beim Verlassen des Fokus des DOM-Elementes ausgelöst wird.

```
1    registerOnTouched(fn: any): void
```

Listing 7.23: UI-Komponenten - „MaterialModule/Textbox“ Implementierung von „ControlValueAccessor“ registerOnTouched

Ermöglicht das Registrieren einer Callback-Funktion, welche von der Forms API beim Wechseln des Control-Status des DOM-Elements ausgelöst wird.

```
1    setDisabledState?(isDisabled: boolean): void
```

Listing 7.24: UI-Komponenten - „MaterialModule/Textbox“ Implementierung von „ControlValueAccessor“ setDisabledState

Diese Callback-Funktion ist optional und wird für das Aktivieren und Deaktivieren der Komponente verwendet.

Nun können die Input- und Touch-Events über das Data Binding an die Forms API weitergegeben werden.

Bei der Erstellung von UI-Elementen können Interaktionen mit dem Benutzer dynamisch gestaltet werden. Hierfür muss von der Komponente auf das DOM-Element zugegriffen werden, um Animationen oder stilistische Änderungen vorzunehmen.

```
1  <input #textbox autocomplete="{{autocomplete}}" type="{{type}}" class="
       textfield" (focus)="onClick()" (blur)="onBlur()" (input)="onInput()" [
       formControl]="input">
```

Listing 7.25: UI-Komponenten - „MaterialModule/Textbox“ Struktur

Mit „#«Name»“ kann ein Element innerhalb der Komponente als ViewChild identifiziert werden.

```
1    @ViewChild('label', {static: true})
2    label: ElementRef;
3    @ViewChild('textbox', {static: true})
4    textbox: ElementRef;
```

Listing 7.26: UI-Komponenten - „MaterialModule/Textbox“ ViewChild

Um innerhalb der Komponente auf die View-Elemente zugreifen zu können, werden die Variablen „textbox“ über den Decorator als Elementreferenz definiert und können beispielsweise über den Renderer zur Laufzeit animiert werden:

```
1        this.renderer.setStyle(this.label.nativeElement, 'transform', '
             translate(40px, 8px)');
```

Listing 7.27: UI-Komponenten - „MaterialModule/Textbox“ Renderer

7.2.2. Die Benutzer-Informationen umsetzen (InformationModule)

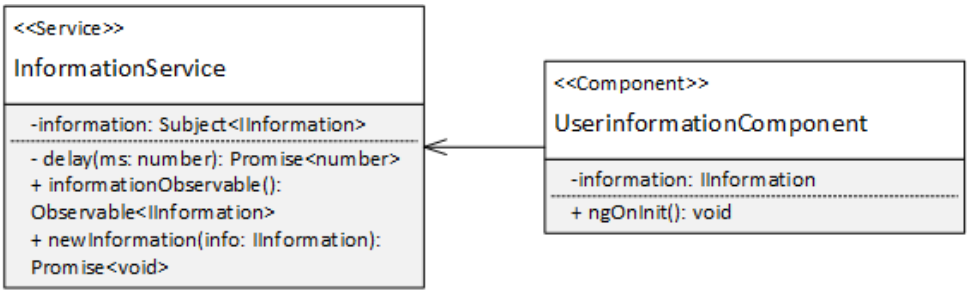

Abbildung 7.1.: InformationModule - Klassendiagramm

Damit von jedem Kontext der Webanwendung aus eine Information erstellt werden kann, wird ein Service verwendet. In diesem können Informationen über eine Observable übermittelt werden. Soll eine neue Information erstellt werden, kann diese mit der Methode „newInformation“ an das DOM-Element kommuniziert werden. Eine Information wird jeweils fünf Sekunden angezeigt.

```
1    ngOnInit(): void {
2      this.informationService.informationObservable.subscribe(async info => {
3        this.information = info;
4      });
5    }
```

Listing 7.28: Benutzerinformationen - „InformationModule/UserInformation“ Information anzeigen

Beim Initialisieren der Komponente werden die Informationen, welche über den Service erstellt werden, in einer Klassen-Variable „information“ definiert.

```
1  [style.background-color]="info.error ? '#B56464' : 'green' " >
```

Listing 7.29: Benutzerinformationen - „InformationModule/UserInformation“ Struktur

Je nachdem ob eine Information vorliegt, soll ein Banner erscheinen. Der Banner hat die Farbe Grün, wenn eine Aktion erfolgreich war und Rot, wenn ein Fehler aufgetreten ist.

7.2.3. Die Authentifizierung ermöglichen (AuthServiceModule)

Für die Authentifizierung der Benutzer stellt Firebase die entsprechenden Methoden über einen eigenen Authentifizierungs-Service zur Verfügung. Für eine Individualisierung der Informationen des Benutzers sollen eigene Funktionalitäten hinzugefügt werden.

```
1   register(email: string, repassword: string, password: string, forename:
        string, name: string): Promise<string> {
2     return new Promise<string> ((resolve, reject) => {
3       if (!this.hostConnectiontService.connection) {
4         return reject('Login fehlgeschlagen! Es konnte Keine Verbindung
              hergestellt werden.');
5       }
6       if (password !== repassword) {
7         return reject('Registrierung fehlgeschlagen! Die Passwörter müssen ü
              bereinstimmen.');
8       }
9       this.angularFireAuth.createUserWithEmailAndPassword(email, password)
10      .then(credential => {
11        this.dbauthService.createUser({publicInfo: {id: credential.user.uid,
              name, forename}})
12        .then(() => {
13          resolve('Registrierung war erfolgreich!');
14        })
15        .catch(err => {
16          return reject('Registrierung fehlgeschlagen!');
17        });
18      })
19      .catch(err => {
20        return reject('Registrierung fehlgeschlagen!');
21      });
22    });
23  }
```

Listing 7.30: Authentifizierung - „AuthServiceModule/AuthService“ Registrieren

Bei der Registrierung der Organisator App soll zunächst ein Profil bei Firebase erstellt werden. Anschließend sollen die angegebenen Informationen in der Datenbank abgespeichert werden. Die Registrierung kann nur durchgeführt werden, wenn der Benutzer eine Verbindung zum Backend hat. Ist diese nicht gegeben, wird die Registrierung abgebrochen. Andernfalls werden die beiden Passwort-Eingaben auf Gleichheit geprüft. Sind die angegeben Daten also valide, kann das Firebase-Profil erstellt werden(Zeile 9). Danach sollen die anderen Angaben innerhalb der Datenbank abgespeichert werden(Zeile 11). Sollten Probleme bei der Registrierung zustande kommen, bricht der Registrie-

rungsprozess ab. Zusätzlich wird eine Abbruchs-Nachricht für den Benutzer zurückgegeben, welche über das “InformationModule“ an den Benutzer kommuniziert wird. Innerhalb des „DataModule“ wird der Benutzer dann in der Datenbank erstellt.

```
1   login(email: string, password: string): Promise<string> {
2     return new Promise<string> ((resolve, reject) => {
3       if (!this.hostConnectiontService.connection) {
4         return reject('Login fehlgeschlagen! Es konnte Keine Verbindung
              hergestellt werden.');
5       }
6       this.angularFireAuth.signInWithEmailAndPassword(email, password)
7       .then(credential => {
8         resolve('Sie haben sich erfolgreich eingeloggt!');
9       })
10      .catch(() => return reject('Login fehlgeschlagen!'));
11    });
12  }
```

Listing 7.31: Authentifizierung - „AuthServiceModule/AuthService“ Einloggen

Um Fehlerausgaben von Firebase zu vermeiden, wird auch beim Login die Verbindung zum Backend geprüft. Ist diese gegeben und die Daten sind korrekt, ist der Login erfolgreich.

```
1   logout(): Promise<string> {
2     return new Promise<string> ((resolve, reject) => {
3       this.angularFireAuth.signOut().then(() => {
4         Promise.all([
5           this.dbauthService.logout(),
6           this.pushService.logout()
7         ]).then(() => resolve('Erfolgreich ausgeloggt!'))
8         .catch(err => reject(err));
9       })
10      .catch(() => {
11        return reject('Ausloggen fehlgeschlagen!');
12      });
13    });
14  }
```

Listing 7.32: Authentifizierung - „AuthServiceModule/AuthService“ Ausloggen

Möchte sich ein Benutzer ausloggen, soll auch der Datenbankservice ausgeloggt und der Service für die Push-Benachrichtigungen beim Backend abgemeldet werden.

7.2.4. Den Verbindungsstatus bestimmen (ConnectionModule)

Mithilfe des Connection-Moduls soll der Status der Verbindung zum Backend abgefragt werden können.

```
1   get connectionObservable(): Observable<boolean> {
2     return this.hostConnectionObserver;
3   }
4
5   get connection() {
6     return this.hostConnection;
7   }
```

Listing 7.33: Backendverbindung - „ConnectionModule/HostConnectionService“ Eigenschaften

Dabei gibt es die Möglichkeit, über den Service einen Live-Status mit der Observable zu erhalten oder eine einmalige Abfrage durchzuführen.

```
1   constructor(private readonly connectionService: ConnectionService,
2               private readonly informationService: InformationService) {
3     this.hostConnectionObserver = new Subject<boolean>();
4     this.hostConnection = navigator.onLine;
5     this.hostConnectionObserver.next(navigator.onLine);
6     this.observeInternetConnection();
7     this.userConnectionInformation();
8   }
```

Listing 7.34: Backendverbindung - „ConnectionModule/HostConnectionService“ Initial-Status

Grundlegend wird zur Bestimmung der Backendverbindung ein Initial-Status der Verbindung des Gerätes festgelegt (Zeile 4). Dieser Status ist jedoch nicht immer richtig, da die Implementierung dieser Funktion von Browser zu Browser unterschiedlich ist. Bei Chrome und Safari wird zum Beispiel ein Status von „false“ zurückgegeben, wenn keine Verbindung zum Router aufgebaut werden konnte (*NavigatorOnLine.onLine* o.D.). Nur impliziert die Verbindung zum Router nicht die Verbindung zum Backend. Deswegen müssen zusätzliche Methoden zu diesem Service hinzugefügt werden.

```
1   private observeInternetConnection(): void {
2     this.connectionService.monitor().subscribe(isConnected => {
3       if (!isConnected) {
4         this.hostConnection = false;
5         this.hostConnectionObserver.next(false);
6       } else {
7         this.hostConnection = true;
8         this.hostConnectionObserver.next(true);
9       }
10    });
11  }
```

Listing 7.35: Backendverbindung - „ConnectionModule/HostConnectionService“ Internetverbindung

Einer dieser Methoden ist das Verwenden des “ng-connection-service„. Mit diesem Service ist es möglich, die Verbindung zum Internet über eine Observable abzufragen.

```
1   public noConnection(): void {
```

```
2       if (this.hostConnection) {
3         this.hostConnection = false;
4         this.hostConnectionObserver.next(false);
5       }
6     }
7
8     public connectionOK(): void {
9       if (!this.hostConnection) {
10        this.hostConnection = true;
11        this.hostConnectionObserver.next(true);
12      }
```

Listing 7.36: Backendverbindung - „ConnectionModule/HostConnectionService“ Datenbankverbindung

Doch können immer noch Verbindungsprobleme zum Backend auftreten, zum Beispiel wenn dieser Offline ist. Sollte dies vorkommen, kann das Data-Modul dieses dem Connection-Modul über die Methoden mitteilen.

```
1     private userConnectionInformation() {
2       this.hostConnectionObserver.subscribe(res => {
3         if (res === false) {
4           this.informationService.newInformation({information: 'Verbindung
                wurde unterbrochen!', error: true});
5         } else {
6           this.informationService.newInformation({information: 'Verbindung ist
                 wieder hergestellt!', error: false});
7         }
8       });
9     }
```

Listing 7.37: Backendverbindung - „ConnectionModule/HostConnectionService“ Benutzer-Information

Sollte sich der Verbindungs-Status zur Laufzeit ändern, so wird dies dem Benutzer über den Information-Service mitgeteilt.

7.2.5. Die Datenspeicherung realisieren (DataModule)

Für die Umsetzung des Data-Modules wurde zunächst eine Vaterklasse erstellt, welche die grundlegenden Funktionalitäten der Services implementiert. Die einzelnen Subklassen implementieren die einzelnen Datenbank-Zugriffe.

7.2.5.1. Vaterklasse

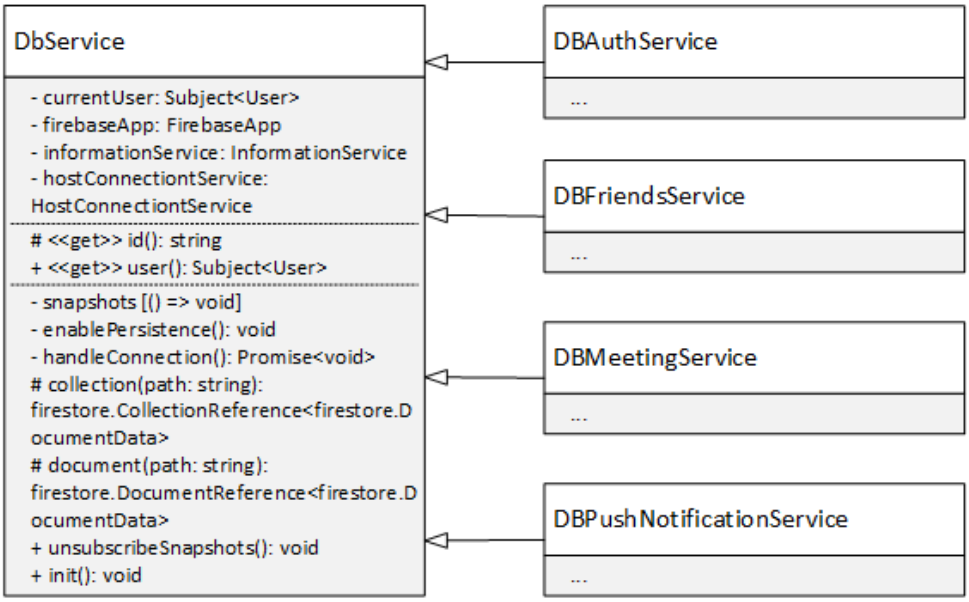

Abbildung 7.2.: DataModule - Klassendiagramm

Für die Initialisierung des Services wird die init-Methode zur Verfügung gestellt.

```
1    public init(): void {
2      this.enablePersistence();
3      this.handleConnection();
4    }
```

Listing 7.38: Datenbankzugriff - „DataModule/DBService“ Initialize

In dieser muss zum einen die Offline-Speicherung der Daten erlaubt werden. Diese funktioniert lediglich, wenn ein Browser-Tab geöffnet ist und die Browserunterstützung gegeben ist. Andernfalls wird der Benutzer über den Information-Service informiert. Zum anderen muss abhängig von dem Verbindungsstatus zum Backend bestimmt werden, ob die Daten in der IndexedDB oder der externen Firestore-Datenbank zugegriffen werden sollen. Da die Id und das User-Objekt des angemeldeten Benutzers häufig benötigt werden, können diese über die Getter „id, user“ abgefragt werden. Zusätzlich wird zur Leserlichkeit der Zugriff auf die Sammlungen und Dokumente innerhalb der Hilfsmethoden implementiert.

```
1    protected collection(path: string): firestore.CollectionReference<
         firestore.DocumentData>{
2      return this.firebaseApp.firestore().collection(path);
3    }
4
5    protected document(path: string): firestore.DocumentReference<firestore.
         DocumentData> {
6      return this.firebaseApp.firestore().doc(path);
```

```
7     }
```

Listing 7.39: Datenbankzugriff - „DataModule/DBService“ Methoden

7.2.5.2. Subklassen

Eine Subklasse des DB-Services ist jeweils einem Domain-Modul zugeteilt. Zusätzlich ist eine Subklasse für das Push-Module erstellt worden. In diese werden die Zugriffe wie folgt implementiert:

```
1     createUser(user: User): Promise<void> {
2       return new Promise<void>((resolve, reject) => {
3         if (user && user.publicInfo) {
4           this.document(`/users/${user.publicInfo.id}`).set(user)
5           .then(() => resolve())
6           .catch(() => return reject('User konnte nicht erstellt werden.'));
7         } else {
8           return reject('User konnte nicht erstellt werden.');
9         }
```

Listing 7.40: Datenbankzugriff - „DataModule/DBAuthService“ Benutzer erstellen

Um einen schreibenden Zugriff auf ein Dokument durchzuführen, wird in Firestore die „set“-Funktion verwendet. In diesem Code-Ausschnitt wird der Benutzer in der Registrierung erstellt. Hierbei soll das Dokument die Id des Benutzers annehmen. Die Id des Benutzers ist wie in Zeile 4 angegeben unter „user.publicInfo.id“ des übergebenen Parameters zu bekommen.

```
1     setPushSubscription(subscription: string): Promise<any> {
2       if (this.hostConnectiontService.connection) {
3         const setPushSubscription = this.firebaseApp.functions().httpsCallable
              ('setPushEndpoint');
4         return setPushSubscription(subscription);
5       }
6     }
```

Listing 7.41: Datenbankzugriff - „DataModule/DBPushNotificationService“ Firebase Cloud Functions Subscription setzen

Jegliche Zugriffe auf die Datenbank können auch über die Firestore Cloud Functions umgesetzt werden. In diesem Fall ist das Eintragen der Subskription realisiert worden. Firestore Cloud Functions haben den großen Vorteil, dass die angegebenen Daten innerhalb der Funktion verändert werden können. Der Nachteil ist hingegen eine Verbindung zum Backend des Clients als Voraussetzung. Da in diesem Fall das Erhalten der Push-Benachrichtigungen so oder so eine Verbindung voraussetzt, kann diese auch über die Firebase Cloud Functions umgesetzt werden.

```
1   deleteMeeting(meetingId: string): Promise<string> {
2     return new Promise<string>((resolve, reject) => {
3       const id = this.id;
4       if (id && meetingId) {
5         this.document(`/users/${id}/meetings/${meetingId}`).delete()
6         .then(() => resolve('Treffen wurde erfolgreich entfernt!'))
7         .catch(err => {
8           return reject('Treffen konnte nicht entfernt werden!');
9         });
10      } else {
11        return reject('Keine Authorisation!');
12      }
13    });
14  }
```

Listing 7.42: Datenbankzugriff - „DataModule/DBMeetingService“ Treffen entfernen

Das Löschen von Daten wird auf ein Dokument mit der Methode „delete()“ ausgeführt.

```
1   public async findNewFriends(name: string, forename: string): Promise<any
        []> {
2     return new Promise<any[]>((resolve, reject) => {
3       this.collection('users')
4       .where('publicInfo.forename', '==', forename)
5       .where('publicInfo.name', '==', name).get()
6       .then(res => {
7         const data = res.docs.map(doc => doc.get('publicInfo'));
8         if (data) {
9           resolve(data);
10        } else {
11          reject(null);
12        }
13      })
14      .catch(err => {
15        reject(null);
16      });
17    });
18  }
```

Listing 7.43: Datenbankzugriff - „DataModule/DBFriendsService“ Benutzer finden

Beim Lesen der Daten kann eine einmalige Abfrage durchgeführt werden. Hierbei können die Bedingungen als Where-Klausel an die Sammlung angehängt werden. Über das Promise kann das Result ausgelesen werden. In diesem Ausschnitt sollen lediglich die öffentlichen Informationen des Benutzers abgefragt werden.

```
1         ...
```

```
2         const usersRequestedSnapshot = this.collection('users').where('
              publicInfo.id', 'in', requestIds)
3         .onSnapshot(usersRequestedSnap => {
4           const usersRequestedData = usersRequestedSnap.docs.map(doc => doc.
                data());
5           if (usersRequestedData) {
6             this.requestedFriends.next(usersRequestedData as User[]);
7           }
8         }, err => {
9           this.informationService.newInformation({
10            information: 'Laden der Freundesanfragen fehlgeschlagen!',
11            error: true
12          });
13        });
14        ...
```

Listing 7.44: Datenbankzugriff - „DataModule/DBFriendsService“ Benutzer lesen

Bei diesem Lesezugriff handelt es sich um eine Aktion, welche die live-Daten einer Sammlung abfragen lässt. Über die Callback-Funktion aus Zeile 3 werden die Änderungen der Daten kommuniziert und können entsprechend verarbeitet werden.

```
1   public approveFriendRequest(friendId: string): Promise<string> {
2     console.log(friendId);
3     return new Promise<string>((resolve, reject) => {
4       const id = this.id;
5       if (id && friendId) {
6
7         const batch = this.firebaseApp.firestore().batch();
8
9         const ownerFriendListDoc = this.document(`/users/${id}/friends/${
              friendId}`);
10        const ownerFriendRequestsListDoc = this.document(`/users/${id}/
              friendRequests/${friendId}`);
11        const commingFriendFriendListDoc = this.document(`/users/${friendId
              }/friends/${id}`);
12
13        batch.set(ownerFriendListDoc, {
14          id: friendId
15        });
16        batch.set(commingFriendFriendListDoc, {id});
17        batch.delete(ownerFriendRequestsListDoc);
18
19        batch.commit()
20        .then(msg => resolve('Freundschaftsanfrage erfolgreich bestätigt!'))
21        .catch(err => {
22          return reject('Freundschaftsanfrage konnte nicht bestätigt werden!
              ');
```

```
23          });
24        } else {
25          return reject('Keine Authorisation!');
26        }
27      });
28    }
```

Listing 7.45: Datenbankzugriff - „DataModule/DBFriendsService“ Freundesanfrage bestätigen

Wird eine Freundschaftsanfrage bestätigt, werden mehrere Dokumente verändert. Damit diese konsistent bleiben, werden Batches verwendet. Diese stellen sicher, dass die Zugriffe auch alle ohne Fehler bis zum Commit ausgeführt werden. Ist dies nicht der Fall, werden die Änderungen rückgängig gemacht. Auch können Transaktionen mit Firebase Cloud Firestore ausgeführt werden. Sie werden in der Organisator App aber nicht benötigt.

7.2.6. Das Layout umsetzen (FriendModule, MeetingModule, AuthModule)

Das komplette Layout der Webanwendung wird innerhalb der Domain-Module definiert. In diesen werden die UI-Elemente mit der Logik vereint. Dazu werden die UI-Elemente zunächst innerhalb des Templates der Root-Komponente vom Domain-Modul definiert. Hier am Beispiel der Treffen der Freunde des Benutzers:

```
1   <ul *ngIf="meetings && (meetings.length > 0)"  [@listAnimation]="meetings.
        length" class="meetinglist" id="friend-meetings">
2     <app-meeting (onSubscribe)="subscribeToMeeting($event)" [userId]="id"
          class="meetingitem"  *ngFor="let meeting of meetings" [meeting] = "
          meeting"></app-meeting>
3   ...
```

Listing 7.46: Domain-Modul - „MeetingModule/MeetingRoot“ Struktur

Damit die Liste nur angezeigt wird, wenn sich auch Treffen in der Liste befinden, wird das Directive „ngIf“ verwendet, um diese entsprechend ein-, beziehungsweise auszublenden. Innerhalb der Liste wird die UI-Komponente „app-addmeeting“ über eine Liste von Objekten mit Hilfe des Directive „ngFor“ iteriert. Hierbei werden die Informationen über das Objekt über Data Binding an das jeweilige UI-Element weitergegeben. Betätigt der Benutzer den Button zum Teilnehmen eines Treffens, wird entsprechend das „onSubscribe“-Event ausgelöst und die entsprechenden Parameter an die „subscribeToMeeting“-Methode weitergegeben.

```
1   subscribeToMeeting(meeting: any): void {
2     this.dbMeetingService.subscribeToMeeting(meeting)
3     .then(msg => this.informationService.newInformation({information: msg,
          error: false}))
4     .catch(err => this.informationService.newInformation({information: err,
          error: true}));
```

```
5    }
```

Listing 7.47: Domain-Modul - „MeetingModule/MeetingRoot“ Logik

Diese muss lediglich bestimmen, mit welchem Service die Informationen verarbeitet werden soll. Anschließend wird der Erfolg oder der Fehlschlag dem Benutzer mitgeteilt.

7.2.7. Die Sicherheit und Navigation der Progressive Web App (RoutingModule)

```
1  const routes: Routes = [
2    { path: 'meetings', component: MeetingRootComponent, canActivate: [
         AuthGuardService, SmartphoneGuardService]},
3    { path: 'friends', component: FriendRootComponent, canActivate: [
         AuthGuardService, SmartphoneGuardService]},
4    { path: 'overview', component: OverviewRootComponent, canActivate: [
         AuthGuardService, ComputerGuardService]},
5    { path: '', component: AuthRootComponent, canActivate: [LoginGuardService
       ]},
6    { path: '**', redirectTo: ''}
7  ];
```

Listing 7.48: Routing - „RoutingModule“ Routes

Zum Erstellen der Routes wird jeweils der URL-Pfad und die entsprechende Root-Komponente angegeben, welche unter diesem Pfad angezeigt werden soll. Mit Hilfe der Guards in Angular können Pfade geschützt werden. Zum Umsetzen eines Guardes wir die canActivate Methode implementiert. In diesem Beispiel sollen die Bereiche „overview, meetings, friends“ nur nach einer erfolgreichen Authentifizierung aufgerufen werden. Damit sich ein Benutzer nicht zweimal anmeldet, wird der Authentifizierungsbereich ebenfalls geschützt. Sollte ein Benutzer einen Bereich aufrufen, welcher nicht in der App existiert, wird dieser zur Startseite zurückgeleitet.

```
1  export class AuthGuardService  implements CanActivate {
```

Listing 7.49: Routing - „RoutingModule/AuthGuard“ Guards-Klasse

Zum Erstellen eines Guards muss zunächst das Interface canActivate implementiert werden.

```
1    canActivate(
2      next: ActivatedRouteSnapshot,
3      state: RouterStateSnapshot) {
4      if (this.authService && this.authService.isLoggedIn) {
5         return true;
6      } else {
7        this.router.navigate(['']);
8        return false;
9      }
10   }
```

Listing 7.50: Routing - „RoutingModule/AuthGuard“ canActivate

Innerhalb der Methode wird der Sicherheitsalgorithmus implementiert. In diesem Guard wird eine Prüfung des Login-Status durchgeführt.

7.2.8. Das Verarbeiten von Push-Benachrichtigungen (PushMessageModule)

```
1   }
2
3   async registration() {
4     navigator.serviceWorker.getRegistration()
5     .then(swr => this.firebaseApp.messaging().useServiceWorker(swr));
6     this.getPermission();
```

Listing 7.51: Push-Benachrichtigungen - „PushMessageModule“ Registration

Um Push-Benachrichtigungen zu erhalten, wird zunächst Firebase der bereits existierende Service Worker von Angular übergeben.

```
1   private getPermission() {
2     this.messaging.requestPermission()
3     .then(() => {
4       this.firebaseAuth.onAuthStateChanged(user => {
5         if (user && user.uid) {
6           if (this.hostConnectiontService.connection) {
7             this.messaging.getToken()
8             .then(token => {
9               this.dbPushNotificationService.setPushSubscription(token);
10            })
11            ...
```

Listing 7.52: Push-Benachrichtigungen - „PushMessageModule“ Erlaubnis

Dieser kann nach der Erlaubnis des Benutzers über den SwPush-Service die Push-Benachrichtigungen entgegennehmen. Sobald sich ein Benutzer einloggt, wird ein neuer Push-Token beim Firebase Backend angefragt und in der Firestore Datenbank abgespeichert.

```
1   }
2
3   private receiveMessage() {
4     this.swPush.messages.subscribe(msg => console.log('push message', msg));
5     this.swPush.notificationClicks.subscribe(click => {
6       window.open('http://localhost:3000/friends', '_blank');
```

Listing 7.53: Push-Benachrichtigungen - „PushMessageModule“ Benachrichtigungen verarbeiten

Nun können Push-Benachrichtigungen erhalten werden. Über die Observable können diese verarbeitet werden.

```
1   logout(): Promise<void> {
2     return new Promise<void>((resolve, reject) => {
3       this.firebaseApp.messaging().getToken()
4       .then(token => {
```

```
5           this.firebaseApp.messaging().deleteToken(token)
6           .then(() => {
7               this.swPush.unsubscribe().then(() => {
8                 this.dbPushNotificationService.deletePushSubscription()
9                 .then(() => resolve())
10                ...
```

Listing 7.54: Push-Benachrichtigungen - „PushMessageModule“ Ausloggen

Wenn sich ein Benutzer ausloggt, muss der verwendete Token gelöscht, sich beim SwPush-Service abgemeldet und im Backend der Token entfernt werden.

7.2.9. Die Module zusammenführen (AppModule)

Innerhalb des App-Modules werden alle nötigen Komponenten der Struktur hinzugefügt:

```
1 <app-userinformation></app-userinformation>
2 <router-outlet></router-outlet>
3 <app-nav></app-nav>
```

Listing 7.55: App - „AppModule/App“ Struktur

Und die entsprechenden Module werden importiert:

```
1  @NgModule({
2    declarations: [
3      AppComponent
4    ],
5    imports: [
6      BrowserModule,
7      MatSnackBarModule,
8      InformationModule,
9      AngularFireModule.initializeApp(environment.firebase),
10     ServiceWorkerModule.register('ngsw-worker.js', { enabled: environment.
           production })
11     ...
12   ],
13   providers: [],
14   bootstrap: [AppComponent]
15 })
```

Listing 7.56: App - „AppModule“ Decorator

Wichtig an dieser Stelle ist, dass das AngularFireModule initialisiert wird. Die Environments enthalten den API-Key sowie wichtige Konfigurationsangaben, ohne die die Funktionalitäten der Bibliothek nicht gegeben wären.

```
1    constructor(private readonly dbService: DbService,
2                private readonly swUpdate: SwUpdate,
3                private readonly matSnackbar: MatSnackBar) {
4      dbService.init();
```

```
5       ...
```

Listing 7.57: App - „AppModule“ Konstruktor

Für die Initialisierung des DataModules muss ebenfalls auch die „init“-Funktion aufgerufen werden.

7.2.10. Den Service Workers implementieren

Mit dem Befehl „ng add @angular/pwa –project «Projektname»“ wird die PWA-Unterstützung zum Projekt hinzugefügt. Danach sollte die Datei „nsgw-config.json“ hinzugefügt worden sein. In dieser können die Einstellungen für den Service Worker vorgenommen werden. Beim Builden der Anwendung wird automatisch ein Service Worker mit den in der JSON vorab getätigten Konfigurationen erstellt.

```
1  {
2    "$schema": "./node_modules/@angular/service-worker/config/schema.json",
3    "index": "/index.html",
4    "assetGroups": [
5      {
6        "name": "app",
7        "installMode": "prefetch",
8        "resources": {
9          "files": [
10           "/favicon.ico",
11           "/index.html",
12           "/manifest.webmanifest",
13           "/*.css",
14           "/*.js"
15         ]
16       }
17     }, {
18       "name": "assets",
19       "installMode": "lazy",
20       "updateMode": "prefetch",
21       "resources": {
22         "files": [
23           "/assets/**",
24           "/*.(eot|svg|cur|jpg|png|webp|gif|otf|ttf|woff|woff2|ani)"
25         ]
26       }
27     }
28   ]
29 }
```

Listing 7.58: Service Worker - Service Worker Konfigurationen

Einstellungen:

- appData: Informationen zur App-Version.
- index: Pfad zur Index-Datei.
- assetGroups: Ressourcen, welche Teil der App sind.
  - name: Eindeutiger Name der assetGroup.
  - installMode: Kann entweder „prefetch“ oder „lazy“ sein. „prefetch“ cached alle einzelnen Ressourcen bei der Installation direkt. „lazy“ cached lediglich jene Ressourcen, welche angefragt werden.
  - updateMode: Kommt eine neue Version der App, kann diese ebenfalls nach der „lazy“-, oder „prefetch“-Methode aktualisiert werden.
  - resources
    - files: Pfad zu den App-Ressourcen (Dateien).
    - urls: Pfad zu den App-Ressourcen (URLs).
- dataGroups: Richtlinien für das Cachen der Daten von Datenaufrufen.
  - name: Eindeutiger Name der dataGroup.
  - urls: Daten dieser URLs werden gecached.
  - version: Versionsangabe zum Vermeiden von eventuell fehlender Rückwärtskompatibilität.
  - cacheConfig: Richtlinien für das Cachen der Daten
    - maxSize: Maximale Anzahl der Einträge im Cache.
    - maxAge: Maximales Alter eines Eintrages, bevor dieser entfernt wird.
    - timeout: Maximale Länge einer Netzanfrage, bevor ein Timeout ausgelöst wird.
    - strategy: Entweder „freshness“ oder „performance“. „freshness“ entspricht der Caching Strategie „Cache falling back to Network“. „performance“ entspricht der Caching Strategie „Network falling back to Cache“
- navigationUrls: Optionale Liste mit URLs, welche auf die Index umleiten.

Für eine detailierte Beschreibung des Schemas kann im angegebenen Pfad der Zeile 2 in Listing 7.58 nachgelesen werden.

```
1 {
2   "projects": {
3       "architect": {
4         "build": {
5           "configurations": {
6               "serviceWorker": true,
```

```
7                 "ngswConfigPath": "ngsw-config.json"
8               }
9             },
10            ...
```

Listing 7.59: Service Worker - Eintrag in Angular Konfigurationen

Damit das Builden des Service Workers ausgeführt wird, muss innerhalb der „angular.json“ dieser angegeben werden. Der Pfad der Konfigurations-Datei muss ebenfalls angegeben werden. Im Normalfall wird der Eintrag automatisch erzeugt, sollte vor der ersten Ausführung doch trotzdem geprüft werden. Damit der Service Worker auch immer auf dem neusten Stand bleibt, denn dieser ist ja im Browser registriert. Demnach muss ein neuer Service Worker dem Client auch kommuniziert werden. Damit dieser die neue Version beim Browser registriert, stellt Angular die Klasse SwUpdate zur Verfügung.

```
1 export class AppModule {
2   constructor(private readonly swUpdate: SwUpdate,
3               private readonly matSnackbar: MatSnackBar) {
4     swUpdate.available.subscribe(() => {
5       const snackbar = matSnackbar.open('Neue Version verfügbar.', 'Neu
            laden');
6       snackbar.onAction().subscribe(() => window.location.reload());
7     });
8   }
9 }
```

Listing 7.60: Service Worker - Service Worker updaten

Innerhalb des App-Moduls wird ein Listener gestartet, welcher bei der Verfügbarkeit eines neuen Service Workers dem Benutzer dies kommuniziert und ihm die Möglichkeit gibt, die Seite neu zu laden und somit den neuen Service Worker zu registrieren.

8. Testen

Für die Sicherstellung, dass die Funktionalitäten und Sicherheitsregeln so funktionieren, wie diese vorgesehen waren, müssen Unit-Tests durchgeführt werden. Grundlegend sollten Unit-Tests in Jest oder Jasmine so strukturiert sein, dass mit Describe-Blocks eine Verantwortung/Funktion getestet wird. Innerhalb des Describe-Blocks wird diese auf unterschiedlichen Erwartungen getestet. Dabei wird bei einem Test immer nur auf eine Erwartung getestet.

8.1. Die Sicherheitsregeln testen

Das Testen der Sicherheitsregeln wird in diesem Beispiel mit dem Framework JestJS umgesetzt. Zur Übersichtlichkeit werden zuerst einige Hilfsfunktionen implementiert.

```
1 const firebase = require('@firebase/testing');
2 const fs = require('fs');
```

Listing 8.1: Firebase Cloud Firestore Security Rules testen - „helper.spec.js“ Bibliotheken

Für das Testen werden zwei Bibliotheken benötigt.

```
1  module.exports.setup = async (auth, data) => {
2      const projectId = `rules-spec-${Date.now()}`;
3      const app = await firebase.initializeTestApp({
4          projectId,
5          auth
6      });
7
8      const adminAPP = firebase.initializeAdminApp({
9          projectId,
10         auth
11     });
```

Listing 8.2: Firebase Cloud Firestore Security Rules testen - „helper.spec.js“ Setup

Für das Initialisieren einer Firebase-App wird eine eindeutige Id benötigt. Um diese zu erstellen, wird das „Date.now()“ verwendet. Für das Testen der Regeln muss eine Test-App initialisiert werden. Ein beliebiges Objekt ist hierbei „auth“ mit E-Mail und UId. Um Szenarien testen zu können, müssen eventuell vorab Einträge erstellt werden. Diese benötigen Rechte, welche von den Sicherheitsregeln blockiert werden können. Deswegen wird ebenfalls eine Admin-App erstellt.

```
1      const db = app.firestore();
2      const adminDB = adminAPP.firestore();
3
4      if (data) {
5          for (const key in data) {
6              const ref = adminDB.doc(key);
7              await ref.set(data[key]);
```

```
8           }
9       }
```

Listing 8.3: Firebase Cloud Firestore Security Rules testen - „helper.spec.js“ Data

Nun wird Firestore in einer Variable definiert und die entsprechenden Gegebenheiten als Vorbereitung der Tests vorgenommen.

```
1       await firebase.loadFirestoreRules({
2           projectId,
3           rules: fs.readFileSync('firestore.rules', 'utf8')
4       });
5
6       return db;
7   }
```

Listing 8.4: Firebase Cloud Firestore Security Rules testen - „helper.spec.js“ Load Firestore Rules

Für das Testen der Sicherheitsregeln werden diese geladen und die Test-App zurückgegeben.

```
1   module.exports.teardown = async () => {
2       Promise.all(firebase.apps().map(app => app.delete()));
3   };
```

Listing 8.5: Firebase Cloud Firestore Security Rules testen - „helper.spec.js“ Teardown

Damit alle Daten nach dem Testen bereinigt werden, wird eine Teardown-Funktion implementiert.

```
1   expect.extend({
2       async toAllow(x) {
3           let pass = false;
4           try {
5               await firebase.assertSucceeds(x);
6               pass = true;
7           } catch (err) {}
8
9           return {
10              pass,
11              message: () => 'Expected Firebase operation to be allowed, but
                    it failed'
12          };
13      }
14  });
```

Listing 8.6: Firebase Cloud Firestore Security Rules testen - „helper.spec.js“ toAllow

Bei Zugriffen, welche eine Erfolgs-Erwartung besitzen, wird eine Funktion geschrieben, welche eine entsprechende Nachricht zurückgeben, sollte diese scheitern. Dasselbe gilt für eine Fehlschlags-Erwartung. Da es beim Zugriff auf die Datenbank nur zwei Status gibt („gewährt, nicht gewährt“), reichen diese für das Prüfen der Regeln aus. Danach können die Hilfsfunktionen zum Testen importiert werden.

Für das Testen der Sicherheits-Regeln sollen nach jedem Test die im und für den Test getätigten Einträge bereinigt werden. Hierfür wird die „afterEach“-Funktion genutzt, um die „teardown“-Funktion auszuführen.

```
1      test('failed reading when is friend', async () => {
2          const db = await setup(mockAuth, mockData);
3          const userRef = db.collection(`users/${mockfriendId}/friends`);
4          await expect(userRef.get()).toDeny();
5      });
```

Listing 8.7: Firebase Cloud Firestore Security Rules testen - „friends.spec.js“ Beispiel

Ein Test wird jeweils mit einer Beschreibung des erwarteten Verhaltens dargestellt. Innerhalb des Tests werden die Authentifizierungs-Daten übergeben und mit Hilfe des „mockData“ die entsprechenden Startgegebenheiten der Datenbank definiert. In der Referenz versucht ein Freund eines Users auf dessen Freunde zuzugreifen. Nach den Sicherheitsregeln wird natürlich erwartet, dass dieser nicht die nötigen Berechtigungen hat um die Sammlung lesen zu können.

```
1    "scripts": {
2      "test": "jest --watchAll"
3    },
4    "dependencies": {
5      "@firebase/testing": "^0.18.2"
6    },
7    "devDependencies": {
8      "jest": "^25.2.7"
9    }
10   ...
```

Listing 8.8: Firebase Cloud Firestore Security Rules testen - „package.json“ JestJS in package.json setzen

Für das Ausführen der Tests müssen innerhalb der „package.json“ die Konfigurationen für das verwenden von JestJS gesetzt werden.

8.2. Unittest der Backend-Funktionen

Für das Testen der Funktionen wird zwischen Online- und Offline-Tests unterschieden. Online-Tests kommen zum Testen der Trigger zum Einsatz. Trigger führen eine definierte Funktion aus, wenn eine definierte Aktion passiert. So können zum Beispiel das Hinzufügen, Verändern oder Löschen von Dokumenten einen Trigger auslösen. HTTPS-Funktionen, welche als Backend-Funktionen vom Client aufgerufen werden, können hingegen offline getestet werden.

8.2.1. Online Test

Für das Testen wird das Framework Jest verwendet. Dieses kann mit dem Befehl „npm install –save-dev jest ts-jest @types/jest“ installiert werden. Zunächst wird ein Online-Test durchgeführt. Hierzu muss die Testbibliothek von Firebase installiert und importiert werden. Diese erlaubt das erstellen von Mock-Daten für die Firebase Cloud Functions und hilft beim Erstellen von Snapshots der Dokumente in Firestore. Hierfür wird der Befehl „npm i firebase-functions-test“ verwendet. Für die spätere Kontrolle muss Firebase-Admin ebenfalls importiert werden.

```
1 const test = require('firebase-functions-test')({
2     databaseURL: 'https://bachelorarbeit-2e596.firebaseio.com',
3     storageBucket: 'bachelorarbeit-2e596.appspot.com',
4     projectId: 'bachelorarbeit-2e596',
5 }, './service-account.json');
6
7
8 test.mockConfig({someapi: {key: 'abs123'}});
```

Listing 8.9: Firebase Cloud Function testen - „online.test.ts“ Konfiguration

Alle wichtigen Projektvariablen werden für den Test übergeben und ein Mock-API-Key gesetzt. Danach wird eine Beschreibung für die kommenden Tests angegeben.

```
1 describe('createUser', () => {
```

Listing 8.10: Firebase Cloud Function testen - „online.test.ts“ Test Beschreibung

Zum Ausführen der Tests wird eine Wrapped-Funktion definiert. Diese stellt sicher, dass die Ausführung nur innerhalb des Describe-Blocks gültig ist. Innerhalb der Wrapped-Funktion wird also die zu testende Funktion angegeben.

```
1     let wrapped;
2     beforeAll(() => {
3         wrapped = test.wrap(createUser);
4     });
```

Listing 8.11: Firebase Cloud Function testen - „online.test.ts“ Test Wrapper

Im Test wird dann der entsprechende Pfad des Dokuments und die Daten, welche im Request übergeben werden sollen, definiert.

```
1     test('it creates a user', async () => {
2         const path = 'users/123';
3         const data = {
4             publicInfo: {
5                 id: "123",
6                 forename: "forename123",
7                 name: "name123"
8             }
```

```
9           };
```

Listing 8.12: Firebase Cloud Function testen - „online.test.ts“ Test Environments

Damit der Trigger ausgelöst wird, wird der entsprechende Dokumenten-Snapshot erstellt und über die Wrapped-Funktion ausgeführt

```
1           const snap = test.firestore.makeDocumentSnapshot(data, path);
2           await wrapped(snap);
```

Listing 8.13: Firebase Cloud Function testen - „online.test.ts“ Test Trigger

Danach wird geprüft, ob die entsprechende Funktion korrekt ausgeführt wurde.

```
1           const after = await admin.firestore().doc(path).get();
2           expect(after?.data()?.publicInfo?.id).toBe("123");
```

Listing 8.14: Firebase Cloud Function testen - „online.test.ts“ Test Soll-Zustand

8.2.2. Offline Test

Ein Offline-Test benötigt hingegen keine Wrapper-Funktion. In diesem wird lediglich der Describe-Block und der Test erstellt. In diesem kann ein Request und ein Response definiert werden.

```
1           const res = {
2               send: (payload) => {
3                   expect(payload).toBe(true)
4               },
5           };
6           const req = "123456789";
```

Listing 8.15: Firebase Cloud Function testen - „online.test.ts“ Test Request, Response

Innerhalb des Response wird die Prüfung durchgeführt. Im Request werden hingegen die zu übermittelnden Daten angegeben. Mit der Ausführung der Funktion wird automatisch die Expectation ausgelöst.

```
1           setPushEndpoint(req as any, res as any)
```

Listing 8.16: Firebase Cloud Function testen - „online.test.ts“ Test Funktionsaufruf

8.3. Unittest und Integrationstest der Progressive Web App

Angular verwendet Jasmine zum Testen. Grundlegend können Komponenten und Services auf die selbe Art getestet werden. Die Tests werden hierbei innerhalb der „«Name».spec.ts“ implementiert. Mit dem Erstellen einer Komponente oder eines Services wird automatisch eine Testdatei generiert. Vordefiniert ist darin bereits das Erstellen der Komponente oder des Services.

```
1 describe('HostConnectiontService', () => {
```

Listing 8.17: Angular Komponenten und Services testen Describe-Block

Zunächst wird wie auch in den vorherigen Tests beschrieben, was genau getestet wird.

```
1     beforeEach(() => TestBed.configureTestingModule({
2       imports: [
3         InformationModule
4       ]
5     }));
```

Listing 8.18: Angular Komponenten und Services testen Konfigurationen

Innerhalb des Testprozesses wird ein Test-Modul erstellt, indem der Service getestet werden soll. Hierfür können die nötigen Abhängigkeiten, wie andere Services und Imports für eine Integrationstest getätigt werden. Bei Unit-Tests wird die Komponente hingegen gekapselt und als eigene Einheit getestet.

```
1     it('should be created', () => {
2       const service: HostConnectiontService = TestBed.get(
            HostConnectiontService);
3       expect(service).toBeTruthy();
4     });
```

Listing 8.19: Angular Komponenten und Services testen Testszenario

Ein Test wird mit der Funktion it durchgeführt. Innerhalb der Beschreibung wird das erwartete Ergebnis definiert. In diesem Test wird das Erstellen der Instanz vom Service getestet.

Zum Ausführen der Tests wird der Befehl „ng test“ im Terminal ausgeführt. Innerhalb des Browsers wird über Karma das Ergebnis angezeigt. Wird der Code während des Tests verändert, werden auch die geschriebenen Tests automatisch ausgeführt und die Ergebnisse aktualisiert.

9. Validieren und Migrieren

Mit dem Testen wurde sichergestellt, dass die Funktionalitäten der Anwendung das machen, was sie nach den Anforderungen machen sollen. Ungewiss ist bisher aber noch, ob die Webanwendung bei den Browsern auch als PWA erkannt wird und als solche die Merkmale erfüllt. Deswegen sollte noch vor der Migration der PWA eine Validierung durchgeführt werden.

9.1. Progressive Web App validieren

Bei einer Validierung wird die PWA auf mehrere Prüfkriterien untersucht und daraus eine Bewertung erstellt. Eine Validierung sollte bestenfalls nicht erst am Ende geschehen, sondern auch während der Programmierphase. In Bezug auf Angular lassen sich die folgenden vorgestellten Tools über die „package.json“-Datei im Building-Prozess integrieren. Hierfür muss der entsprechende Befehl lediglich in der Script-Option angegeben werden. So wird sicher gestellt, dass nicht nur die Merkmale der PWA erfüllt werden, sondern auch eine bessere Gesamt-Qualität der Webanwendung gewährleistet wird. Validierungs-Tools gibt es in verschiedenen Formen. Für die Validierung der PWA sollen die Tools Lighthouse und Webhint verwendet werden. Beide sind spezialisiert auf die Validierung der User Experience von Webanwendungen und besitzen eine Prüfung von PWAs.

Lighthouse: Bei Lighthouse handelt es sich um ein von Google bereitgestelltes Tool. Im Google Chrome Browser ist Lighthouse in den Entwicklungstools integriert. Das Tool ist unter dem Tab „AUDIT“ oder „Lighthouse“ zu finden. Vor der Validierung können Einstellungen getätigt werden:

- Für das Testen der Webanwendung kann zwischen einem mobilen und einem Desktop-Gerät ausgewählt werden. Bei einem Mobil-Gerät wird unter anderem die Größe des Bildschirms inklusive der Pixeldichte simuliert.
- Unter Audits können die entsprechenden Kategorien der Prüfkriterien ausgewählt werden.
- Da die mobilen Aufrufe in den letzten fünf Jahren stark gestiegen sind, kann über die Throttling-Einstellung die Verbindung auf 3G gedrosselt oder simuliert werden.
- Vor Ausführung der Validierung kann auch eine automatische Speicherreinigung durchgeführt werden.

Nach dem Generieren des Reports wird ein Score erstellt (Abbildung 9.1).
Bei diesem Stand der PWA sollten noch Optimierungen an der Performance vorgenommen werden. Die Schwachpunkte und Verbesserungsvorschläge zu den einzelnen Kategorien werden im unteren Bereich angezeigt (Abbildung B.5 im Anhang). Bei diesem Ergebnis wurde vorab der Speicher bereinigt. Der Service Worker wird während des Testes also erst registriert. Für ein Vergleich ist hier das Ergebnis ohne eine Vorab-Speicherbereinigung (Abbildung 9.2).

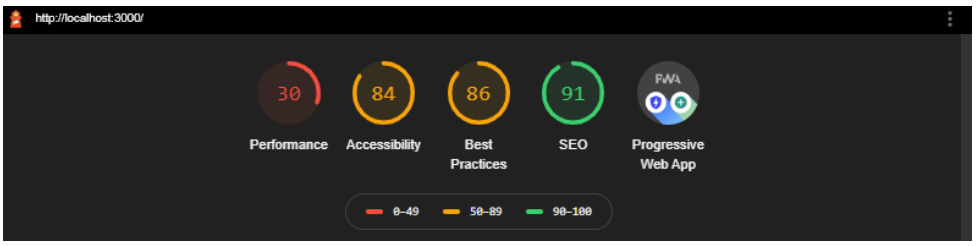

Abbildung 9.1.: Beispiel - Lighthouse Score mit Speicherbereinigung

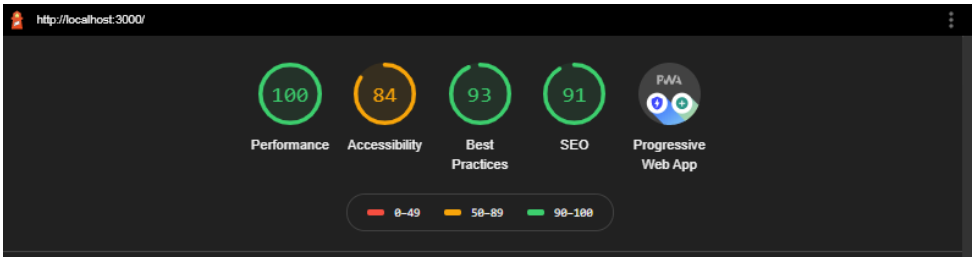

Abbildung 9.2.: Beispiel - Lighthouse Score ohne Speicherbereinigung

Wie im Ergebnis zu sehen ist, ist die Performance von 30% auf 100% gestiegen. Der Grund ist, dass die Ressourcen eben über den Service Worker bedient werden.

Sollte Chrome nicht als Browser installiert sein, so kann mit Hilfe vom Node Package Manage Lighthouse mit dem Befehl „npm i -g lighthouse@«Version»“ eigenständig installiert werden. Die Validierung wird mit dem Befehl „lighthouse «Domainname»“ ausgeführt. Das Resultat wird als HTML-Datei innerhalb des entsprechenden Pfades der Ausführung erstellt.

Webhint: Webhint ist ein Linter Tool, welches ursprünglich von Microsoft entwickelt wurde. Nun ist dieses ein Teil-Projekt von der OpenJS Foundation (*webhint charter* o.D.). Die Installation von Webhint wird über den Node Package Manager mit dem Befehl „npm i -g hint“ gestartet. Für das Validieren einer PWA kann mit dem Befehl „npm create hintrc“ eine Konfigurationsdatei erstellt werden. Diese kann entweder individuell oder in einer Auswahl vorgefertigter Vorlagen ausgewählt werden.

```
C:\WINDOWS\system32>npm create hintrc
npx: installed 184 in 17.624s
Welcome to hint configuration generator
? Do you want to use a predefined configuration or create your own based on your installed packages? predefined
? Choose the configuration you want to extend from progressive-web-apps
```

Abbildung 9.3.: Beispiel - Webhint Konfigurationen

Nach der Konfiguration kann mit dem Befehl „hint «Domainname»“ die Validation gestartet werden. Innerhalb der Konsole werden direkt Fehler und Warnungen je nach Konfiguration ausgegeben. Die Resultate werden ebenfalls als HTML-Datei innerhalb eines Ordners des Pfades der Ausführung

abgelegt. Der Pfad zum Ergebnis wird zusätzlich in der Konsole ausgegeben.

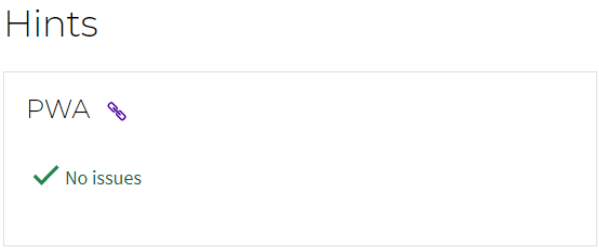

Abbildung 9.4.: Beispiel - Webhint Resultat

In diesem Ergebnis sind keine Fehler aufgetreten. Mit Webhint lassen sich auch mehrere Prüfkriterien der Webentwicklung validieren. Diese müssen innerhalb der erstellten „hintrc“-Datei angegeben werden.

```
{
  "connector": {
        "name": "puppeteer"
    },
    "formatters": [
        "html",
        "summary"
    ],
    "extends": [
        "web-recommended","progressive-web-apps"
    ]
}
```

Listing 9.1: Webhint - „.hintrc“ Konfiguration

„formatters“ beschreibt hierbei das Format, in dem das Ergebnis abgespeichert werden soll. Der Connector ist verantwortlich für das Rendering und Laden der Webseite und die Weitergabe der Daten und Ressourcen(*Connectors* o.D.). Innerhalb „extends“ können vorgefertigte Vorlagen verwendet werden.

9.2. Migration

Apache Cordova und GitHub Electron, PWABuilder, bieten die Möglichkeit, die PWA in einer ausführbaren Datei zu verpacken. Die Gründe für eine Migration können Problematiken(Abschnitt 2.4) der plattformunabhängigen Programmierung, fehlende Browserunterstützung oder Marketingaspekte sein. Die PWA kann somit als hybride Anwendung angeboten werden und hat Zugriff auf die nativen Schnittstellen. Die Migration der Organisator App soll in diesem Beispiel mit Apache Cordova

vorgenommen werden.

Apache Cordova kann Anwendungen für Android, iOS und Windows Store Apps erstellen. Entsprechend der Plattform müssen die jeweiligen Voraussetzungen, wie das Java Development Kit für Android oder Xcode für iOS erfüllt sein. Für die Installation über den Node Package Manager wird der Befehl „npm i -g cordova“ verwendet. Mit dem Befehl „cordova create «Ordnername» io.cordova.«Packagename» «Anzeigename»“ kann ein Apache Cordova Projekt erstellt werden. Innerhalb des Ordners werden die entsprechenden Dateien zur Entwicklung erstellt. Die Plattform, in welche Apache Cordova die Anwendung erstellen soll, wird mit dem Befehl „cordova platform add «Plattform»“ konfiguriert. Die Anwendung wird mit dem Befehl „cordova build «Plattform»“ erstellt. Das Umsetzen einer hybriden Anwendung in Apache Cordova soll in diesem Kapitel aber nicht weiter vertieft werden.

10. Vergleich von Progressive Web Apps

In dieser Diskussion soll eine Beurteilung des Ergebnisses stattfinden. Wie ist das Ergebnis ausgefallen und in welcher Hinsicht kann sich eine PWA von anderen nativen sowie plattformunabhängigen Apps abheben?

10.1. Vergleich von Progressive Web Apps mit Webanwendungen

Eine PWA hebt sich grundlegend mit seinen Merkmalen von einer „einfachen“ Webanwendung ab. Diese Merkmale sollen die sie zu einer fortschrittlichen Webanwendung machen. Dies hat eine indirekte Auswirkung auf ihre Entwicklung, weil keine konkreten Vorgaben angegeben werden, wie die Merkmale realisiert werden sollen. So wird die Verbindungsunabhängigkeit mindestens mit Hilfe der App-Shell verwirklicht. Die App-Shell wird grundlegend in eine Single-Page Webanwendungen implementiert. Das Verwenden von Single-Page Webanwendungen hat somit eine indirekte Auswirkung auf die Entwicklung von PWAs. Ohne oder mit welchem Framework diese erstellt wird, bleibt dem Entwickler an dieser Stelle überlassen.

Zum Thema der Sicherheit hebt sich eine Webanwendung nicht von der PWA ab. Das Verwenden von HTTPS ist bereits mit der Entscheidung vieler Browser, HTTP-Verbindungen als unsicher zu markieren, ein Standard unter den Webanwendungen. Zusätzlich wird die Sicherheit von HTTPS mittlerweile angezweifelt, da die Zertifizierungsstellen auf Grund der Massen an Anfragen teils mangelhafte Prüfungen durchführen und lediglich mit der Angabe des Domainnamens TLS-Zertifikate vergeben.

Auch wenn Schnelligkeit kein Merkmal einer PWA ist, wird diese über den Service Worker optimiert. Anders als der Zugriff die Ressourcen und Daten externer Speicherquellen ist der Zugriff auf den Cache deutlich schneller (siehe Abbildung 9.2). Es benötigt keine Verbindung zum Server und unterliegt keiner Sicherheitskontrolle. Dies ist zu einem ein Vorteil, kann aber auch zum Nachteil werden. Die lokale Speicherung sensibler Daten sollte nur mit Zustimmung des Benutzers geschehen, da andernfalls eine datenschutzrechtliche Verletzung entstehen kann.

Mit der Implementierung des Service Workers ist auch die Möglichkeit, Push-Benachrichtigungen zu empfangen, gegeben. Anders als bei „einfachen“ Webanwendungen können diese empfangen werden, wenn die Webanwendung im Browser gar nicht geöffnet ist.

Schlussendlich kann die PWA im Vergleich zur Webanwendung als vollwertige Version auf dem Gerät installiert werden. Diese ist kaum von einer nativen Anwendung zu unterscheiden und bietet mit den Push-Benachrichtigungen ein Werkzeug für das Marketing, den Benutzer zu binden.

10.2. Vergleich von Progressive Web Apps mit nativen Anwendungen

Eine PWA nutzt moderne Webschnittstellen des Browsers, um die Funktionalitäten der Webanwendung umzusetzen. Die native Anwendung setzt ihre Funktionalitäten hingegen über die nativen Schnittstellen der Plattform zur Verfügung. Mit diesen Schnittstellen können native Anwendungen teilweise direkt auf die Hardware-Funktionen zugreifen und werden dementsprechend schneller ausgeführt. Die Ausführung über den Browser hingegen benötigt zusätzlich den Browser-Prozess, um auf die Funktionalitäten zuzugreifen, was unter anderem den Batterieverbrauch erhöhen kann (*Architektur und die Vor- und Nachteile im Überblick* 2018).

Eine PWA hat den großen Vorteil, über die Webseite des Betreibers installiert zu werden. Hierfür wird lediglich eine Verknüpfung auf dem jeweiligen Gerät erstellt und die Anwendung kann auch wie eine native Anwendung benutzt werden. Das ist ein großer Vorteil, da das Interesse des Besuchers bereits geweckt sein kann, wenn diesem der Vorschlag einer Installation gegeben wird. Und auch, wenn die Anwendung nicht direkt installiert wurde, hat der Besucher trotzdem die Möglichkeit, den Content des Betreibers zu sehen. Mit der richtigen Marketingstrategie kann so ein Kunde an die Anwendung gebunden werden. Ein Nachteil dabei ist, dass sich, sofern die PWA nicht für eine Hybride App migriert wurde, der Betreiber auch ebenfalls um den Vertrieb kümmern muss. Ist die PWA hingegen migriert worden, kann diese parallel als PWA über das Web und innerhalb der App Stores als Hybride Anwendung angeboten werden.

Und auch in der Entwicklung ist eine PWA kostengünstig, im Verhältnis zur potentiellen Benutzerreichweite. Für eine native Anwendung müssten entsprechend mehrere Projekte erstellt werden, um mehrere Betriebssysteme zu erreichen, sofern die PWA die nötigen Schnittstellen für die Umsetzung zur Verfügung stellt.

Bei einer nativen Anwendung werden bei der Installation alle Ressourcen der Anwendung installiert. Bei der PWA übernimmt dies wie bereits geschrieben der Service Worker. Dieser übernimmt ebenfalls auch die Versionsverwaltung der PWA. Somit müssen diese nicht über den App Store gemanagt, sondern können über den Webserver automatisch aktualisiert werden.

Durch die Ressourcenverwaltung des Service Managers ist der erste Schritt zur Offline-Fähigkeit bereits getan. So können auch die Daten über den Service Worker offline zur Verfügung gestellt werden. Caching-Strategien können hierbei mit der Verfügbarkeit von Content die Benutzer-Experience erhöhen und die Daten zum Server einfacher synchronisieren. Bei der Verbindung wird hierbei eine HTTPS-Verbindung vorausgesetzt, welche innerhalb einer nativen Anwendungsentwicklung kein Muss sind. Somit haben PWAs einen größeren Sicherheitsstandard als eine native Anwendung.

10.3. Vergleich von Progressive Web Apps mit Hybrid und Cross-Platform Anwendungen

Die PWA lässt sich in der plattformunabhängigen Programmierung in die Form der Webanwendungen einordnen. Eine hybride Anwendung ist erstmal auch nur eine Webanwendung, welche innerhalb eines Wrappers zur Verfügung gestellt wird. Durch die zusätzliche native Schnittstelle ist eine Hybride Anwendung zunächst mächtiger als eine PWA, jedoch gibt es auch dort Einschränkungen. So ist das Verwenden der nativen Schnittstellen langsamer als bei einer nativen Anwendung. Und grundsätzlich müssen bei Anwendungen, welche im App Store zur Verfügung gestellt werden, die Voraussetzungen auch in Bezug auf die Inhalte erfüllt werden. Diese existieren im Web nicht.

Die Abgrenzung zu Cross-Platform Apps ist hingegen größer. Diese ist nicht von den Webschnittstellen abhängig. Wenn also neue native Funktionen für die Entwicklung zur Verfügung stehen, sind diese direkt verfügbar. Bei der Entwicklung einer PWA oder hybriden App müssen diese warten, bis die entsprechende Schnittstelle im Framework zur Verfügung steht. In Bezug auf die Entwicklung benötigt eine PWA keine gesonderte Codebasis und auch keine entsprechende SDK.

Die Vorteile, welche sich aus den Merkmalen einer Progressive Web App ergeben, sollen an dieser Stelle nicht erneut aufgegriffen werden.

11. Fazit und Ausblick

Für das Ergebnis dieser Arbeit soll aus den Kenntnissen ein Fazit gezogen werden. Schlussendlich soll das Thema mit einem Ausblick enden.

11.1. Ergebnis der Organisator App

Angular als Framework für die Entwicklung einer PWA: Angular bietet mit ihren Modulen und Komponenten eine optimale und einfache Art, Webanwendungen strukturiert zu entwickeln. Die Abgrenzung zwischen Struktur(HTML), Aussehen(CSS/SCSS/SASS) und Logik(TS) sorgt für eine gute Übersichtlichkeit und Separierung des Codes. Zusätzlich lassen diese sich optimal mit dem Data Binding verbinden, wodurch eine einfache Weitergabe der Informationen und Angaben ermöglicht wird. Die PWA-Unterstützung lässt sich über die CLI schnell und einfach integrieren. Der dem Angular eigene Service Worker, welcher über die „ngsw-config.json“ konfiguriert werden kann, gibt die Möglichkeit, Ressourcen und Daten einfach cachen zu lassen. Doch bringt die Umsetzung vorgegebener Konfigurationen des Service Workers auch Nachteile mit sich. So stehen mit „freshness, performance“ lediglich zwei Caching-Strategien zur Verfügung. Auch bei den Möglichkeiten der Installation der Daten und Ressourcen sind lediglich zwei Optionen gegeben („prefetch, lazy“). Die Umsetzung, einen Service Worker individuell zu implementieren, ist angesichts der verfügbaren Tools, wie zum Beispiel Workbox und der Verfügbarkeit moderner Webschnittstellen, welche in Unterabschnitt 3.3.4 vorgestellt wurden, ist auch keine große Hürde mehr. Für größere PWA-Projekte, welche mehrere Caching-Strategien und Installations-Einstellungen benötigen, ist die Implementierung eines Service Worker Skripts also trotzdem noch nötig. Diese können in Angular ebenfalls integriert werden. Mit der PWA-Unterstützung wird auch das Manifest erstellt und in der Webanwendung verlinkt, wodurch die Installierbarkeit der PWA gegeben ist. Insgesamt wird durch das Hinzufügen der PWA-Unterstützung der Grundbaustein der PWA gelegt. Dieser besteht zusammenfassend aus kleinen Einstellungen und hauptsächlich aus dem Service Worker von Angular.

Firebase als Backend: Für die Umsetzung der PWA wurde in diesem Beispiel-Projekt das Cachen der Daten über die Firebase Bibliothek im Service-Worker implementiert. In der Rede „*Firebase offline: What works, what doesn't, and what you need to know (Firebase Summit 2019)* 2019“ wird aufgezeigt, was in Firebase offline möglich ist und was nicht. Unter anderem ist das Umsetzen einer Offline-First Anwendung nicht alleine mit der Offline-Unterstützung von Firebase vorgesehen. Eher sollen zum Beispiel schwache und stockende Internetverbindungen ausgeglichen werden. Die Umsetzung dieser Offline-Fähigkeit kann mit Hilfe von Firebase Cloud Firestore einfach umgesetzt werden. Das Erstellen der Datenstruktur ist dank des Prinzips von Sammlungen und Dokumenten vergleichsweise einfach. Im Gegensatz zu reinen SQL-Datenbanken müssen keine Beziehungen oder Relationen durchdacht werden. Erst bei einem schreibenden Zugriff müssen in Verbindung

stehende Informationen eventuell über Transaktionen und Batches aktualisiert werden. Etwas komplexer hingegen gestaltet sich das Definieren der Sicherheitsregeln. Weder die Struktur noch die Rollen noch der Datentyp et cetera sind vorab definiert. Diese müssen über die Sicherheitsregeln bestimmt werden.

11.2. Fazit

Organisator App Mit Bezug auf das Beispielprojekt der Progressive Web App ist Angular als Framework zum Entwickeln Progressiver Web Apps geeignet. Die PWA-Unterstützung macht es leicht eine Progressive Web App zu entwickeln. Bei komplexen Progressive Web Apps sollte der Service Worker jedoch individuell implementiert werden, da der angular-eigene Service Worker nach dieser Erfahrung eher für einfachere Webanwendungen konzipiert ist. Komplexere Verhalten lassen sich auf Grund der Konfigurationsmöglichkeiten schwer bis gar nicht umsetzen.

Die Verwendung von Firebase hat gezeigt, dass sich mit dem dahinterstehenden Konzept einfach und unkompliziert Datenstrukturen umsetzen lassen. Über den Cloud-Service müssen keine Einrichtungs- und Netzwerkarbeiten getätigt werden, was viel Zeit erspart. Die Firebase Cloud Firestore Security Rules lassen schnell und einfach kleinere, aber auch komplexe Sicherheits-Regeln erstellen. Und auch Trigger und Backend-Funktionen lassen sich gut umsetzen. Bezogen auf das Umsetzen einer PWA bietet Firebase mit ihrem Firebase Messaging Service eine gute Wahl als Backend.

Progressive Web Apps Eine Progressive Web App bietet mit Hilfe moderner Webschnittstellen annähernd alle Funktionen, welche in der nativen Anwendungsentwicklung auch zur Verfügung stehen. Eingrenzungen gibt es nur in der Browserunterstützung älterer verwendeter Browser. Fehlende Funktionalitäten können jedoch mit einem minimalen Extraaufwand beseitigt werden. Hierfür kann die Progressive Web App in eine hybride Anwendung migriert werden. Schlussendlich kann eine Progressive Web App in jeglicher Form eingesetzt werden. Besonders attraktiv ist diese als Marketing-Werkzeug zum Gewinnen und halten der Benutzer. Um etwas spezifischer zu werden folgt eine Liste möglicher Anwendungsbereiche:

- **Anwendungen für Content:** Zeitung, Magazin, Blog, Nachrichten, Bilder und so weiter
- **Businessanwendungen:** Formulare, Informationen, Kundendaten und so weiter
- **Soziale Netzwerke und Messenger**

Die Funktionalität ist über die Browserunterstützung der momentan meistverwendeten Browser gegeben. Unabhängig von der Anzahl neuer moderne Webschnittstellen in der Zukunft, bietet die Webentwicklung schon jetzt enorm viele Möglichkeiten. Die präsentierten Schnittstellen dieser Arbeit sind lediglich die Basics.

Die Grenzen einer Progressive Web App hängen hierbei hauptsächlich von dem verwendeten Browser des Benutzers ab. Denn anders als bei der nativen Anwendungsentwicklung müssen Webanwendungen die Funktionalitäten des Browsers verwenden. Mit Blick auf die Statistik (Tenzer 2020) werden stark überwiegend moderne Browser verwendet.

11.3. Ausblick

Die Zukunft Progressiver Web Apps ist abzuwarten. Doch lässt sich rückblickend erkennen, dass sich dieser Ansatz im Aufwärtstrend befindet. Die Überlegung, dass die Progressive Web Apps in Zukunft den App Store abschaffen, ist angesichts der Benutzerakzeptanz wohl in naher Zukunft eher unrealistisch. Zu bedenken ist, dass die App Stores nicht nur als Vertriebsplattform gesehen werden kann. Aus der Sicht der App Store-Besucher, bietet der App Store eine große Auswahl an Apps, welche es ihm einfach macht, die richtige Anwendung zu finden. Gäbe es den App Store nicht, würde es für den Benutzer schwer werden die passende Anwendung zu finden.

Eine PWA hebt sich performance-technisch enorm von einer einfachen Webanwendung ab. Und auch in der Suchmaschinenoptimierung sind Erfolge zu verzeichnen (*PWA Stats* o.D.). Somit ist zu erwarten, dass in der Zukunft auch weiterhin von nativen Anwendungen und Webseiten auf PWAs umgestiegen wird.

12. Literaturverzeichnis

Gamma, Erich u. a. (1996). *Entwurfsmuster*. 5. Aufl. Addison-Wesley, S. 199. ISBN: 3-8273-1862-9.

Fielding, R. u. a. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. Abrufdatum: 05.06.2020. URL: https://www.w3.org/Protocols/rfc2616/rfc2616.html.

Goland, Y. u. a. (1999). *HTTP Extensions for Distributed Authoring – WEBDAV*. Abrufdatum: 05.06.2020. URL: https://tools.ietf.org/html/rfc2518.

Lüecke, Tim (2005). *Frameworks*. Abrufdatum: 20.06.2020. URL: http://www.se.uni-hannover.de/pub/File/kurz-und-gut/ws2004-seminar-entwurf/frameworks_tluecke.pdf.

Schmidt, M. (2005). *Betriebssysteme*. Abrufdatum: 02.06.2020. URL: https://www.informatik.uni-leipzig.de/~meiler/Schuelerseiten.dir/MSchmidt/allgemein.html.

Guidelines For The Issuance And Management Of Extended Validation Certificates (2007). Abrufdatum: 05.06.2020. URL: https://cabforum.org/wp-content/uploads/EV_Certificate_Guidelines.pdf.

Garsiel, Tali und Paul Irish (2011). *Funktionsweise von Browsern: Hinter den Kulissen moderner Web-Browser*. Abrufdatum: 04.06.2020. URL: https://www.html5rocks.com/de/tutorials/internals/howbrowserswork/.

Avci, Oral, Ralph Trittmann und Werner Mellis (2013). *Web-Programmierung - Softwareentwicklung mit Internet-Technologien — Grundlagen, Auswahl, Einsatz — XHTML & HTML, CSS, XML, JavaScript, VBScript, PHP, ASP, Java*. Berlin Heidelberg New York: Springer-Verlag, S. 72. ISBN: 978-3-322-96372-7.

Archibald, Jake (2014). *The offline cookbook*. Abrufdatum: 05.06.2020. URL: https://jakearchibald.com/2014/offline-cookbook/.

Nitze, Andre und Andreas Schmietendorf (2014). *Qualitative und quantitative Bewertungsaspekte bei der agilen Softwareentwicklung plattformübergreifender mobiler Applikationen* -. Berlin: Logos Verlag Berlin GmbH, S. 2. ISBN: 978-3-832-53774-6.

Brockschmidt, Kraig (2015). *Cross-Platform : Write Cross-Platform Hybrid Apps in Visual Studio with Apache Cordova*. Abrufdatum: 04.06.2020. URL: https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/special-issue/cross-platform-write-cross-platform-hybrid-apps-in-visual-studio-with-apache-cordova.

Russell, Alex (2015). *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. Abrufdatum: 18.06.2020. URL: https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955#.6czgj0myh.

Data Binding in Angular (2016). Abrufdatum: 20.06.2020. URL: https://alligator.io/angular/data-binding-angular/.

Hickson, Ian (2016). *Web Storage (Second Edition)*. Abrufdatum: 05.06.2020. URL: https://www.w3.org/TR/webstorage/.

Single-page application vs. multiple-page application (2016). Abrufdatum: 04.06.2020. URL: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.

Motto, Todd (2017). *A deep dive on Angular decorators*. Abrufdatum: 20.06.2020. URL: https://ultimatecourses.com/blog/angular-decorators#property-decorators.

Architektur und die Vor- und Nachteile im Überblick (2018). Abrufdatum: 11.06.2020. URL: https://easy-software.com/de/newsroom/progressive-web-apps/.

Behrends, Erik (2018). *React Native - Native Apps parallel für Android und iOS entwickeln*. Heidelberg: Dpunkt.Verlag GmbH, S. 3–7. ISBN: 978-3-960-09066-3.

Liebel, Christian (2018a). *Progressive Web Apps - Das Praxisbuch. Plattformübergreifende App-Entwicklung mit Angular und Workbox. Für Browser, Windows, macOS, iOS und Android*. Bonn: Rheinwerk Verlag GmbH, S. 171–175. ISBN: 978-3-836-26494-5.

– (2018b). *Progressive Web Apps - Das Praxisbuch. Plattformübergreifende App-Entwicklung mit Angular und Workbox. Für Browser, Windows, macOS, iOS und Android*. Bonn: Rheinwerk Verlag GmbH, S. 192–199. ISBN: 978-3-836-26494-5.

– (2018c). *Progressive Web Apps - Das Praxisbuch. Plattformübergreifende App-Entwicklung mit Angular und Workbox. Für Browser, Windows, macOS, iOS und Android*. Bonn: Rheinwerk Verlag GmbH, S. 112–114. ISBN: 978-3-836-26494-5.

Saborowski, Nico (2018). *Progressive Web App – Das Ende aller App Stores?* Abrufdatum: 20.06.2020. URL: https://www.handelskraft.de/2018/05/progressive-web-app-das-ende-aller-app-stores/.

Uzun, Halil (2018). *Progressive Web Apps (PWA): Die E-Commerce-Revolution*. Abrufdatum: 04.06.2020. URL: https://blog.comwrap.com/progressive-web-apps-pwa-revolutionieren-den-ecommerce.html.

Firebase offline: What works, what doesn't, and what you need to know (Firebase Summit 2019) (2019). Abrufdatum: 06.06.2020. URL: https://www.youtube.com/watch?v=XrltP8bOHT0.

Goll, Joachim (2019a). *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik - Strategien für schwach gekoppelte, korrekte und stabile Software*. Berlin Heidelberg New York: Springer-Verlag, S. 126–128. ISBN: 978-3-658-25975-4.

– (2019b). *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik - Strategien für schwach gekoppelte, korrekte und stabile Software*. Berlin Heidelberg New York: Springer-Verlag, S. 148. ISBN: 978-3-658-25975-4.

Lozhko, Mariia (2019). *15 Top web development trends in 2020*. Abrufdatum: 04.06.2020. URL: https://lanars.com/blog/top-web-development-trends.

Progressive Web-Apps (2019). Abrufdatum: 18.06.2020. URL: https://developer.mozilla.org/de/docs/Web/Progressive_web_apps.

Russell, Alex u. a. (2019). *Service Workers 1*. Abrufdatum: 05.06.2020. URL: https://www.w3.org/TR/service-workers/.

Taylor, Glenn (2019). *Study: PWAs Remain A Work In Progress, But Faster Load Times Show Promise*. Abrufdatum: 16.06.2020. URL: https://retailtouchpoints.com/topics/digital-marketing/mobile-marketing/study-pwas-remain-a-work-in-progress-but-faster-load-times-show-promise.

Varty, Joel (2019). *Top 10 Web Development Trends & Technologies For 2020*. Abrufdatum: 04.06.2020. URL: https://dev.to/agilitycms/top-10-web-development-trends-technologies-for-2020-11ii.

Angular Components - The Fundamentals (2020). Abrufdatum: 20.06.2020. URL: https://blog.angular-university.io/introduction-to-angular-2-fundamentals-of-components-events-properties-and-actions/.

Beverloo, Peter u. a. (2020). *Push API*. Abrufdatum: 05.06.2020. URL: https://www.w3.org/TR/push-api/#dfn-push-subscription.

Firebase Cloud Messaging HTTP protocol (2020). Abrufdatum: 06.06.2020. URL: https://firebase.google.com/docs/cloud-messaging/http-server-ref?hl=de.

LePage, Pete (2020). *The Cache API: A quick guide*. Abrufdatum: 05.06.2020. URL: https://web.dev/cache-api-quick-guide/.

Netflix arbeitet an PWA für Windows 10 (2020). Abrufdatum: 18.06.2020. URL: https://windowsunited.de/netflix-arbeitet-an-pwa-fuer-windows-10/.

Petereit, Dieter (2020a). *Webdesign: So steht es um Progressive-Web-Apps im Jahr 2020*. Abrufdatum: 20.06.2020. URL: https://t3n.de/news/webdesign-steht-um-jahr-2020-1242754/.

– (2020b). *Webdesign: So steht es um Progressive-Web-Apps im Jahr 2020*. Abrufdatum: 04.06.2020. URL: https://t3n.de/news/webdesign-steht-um-jahr-2020-1242754/.

Tenzer, F. (2020). *Marktanteile der meistgenutzten Browserversionen weltweit im April 2020*. Abrufdatum: 09.06.2020. URL: https://de.statista.com/statistik/daten/studie/158095/umfrage/meistgenutzte-browser-im-internet-weltweit/.

API List (o.D.). Abrufdatum: 19.06.2020. URL: https://angular.io/api.

Attribute directives (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/attribute-directives.

Barabas, Javier (o.D.). *Definition von IaaS, PaaS und SaaS*. Abrufdatum: 18.06.2020. URL: https://www.ibm.com/de-de/cloud/learn/iaas-paas-saas.

Cache (o.D.). Abrufdatum: 05.06.2020. URL: https://developer.mozilla.org/de/docs/Web/API/Cache.

CacheStorage (o.D.). Abrufdatum: 05.06.2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/CacheStorage.

Can I use (o.D.). Abrufdatum: 04.06.2020. URL: https://caniuse.com/.

CERN - European Organization for Nuclear Research - Copyright (o.D.). Abrufdatum: 04.06.2020. URL: http://info.cern.ch/Proposal.html.

Connectors (o.D.). Abrufdatum: 08.06.2020. URL: https://webhint.io/docs/user-guide/concepts/connectors/.

Dependency providers (o.D.). Abrufdatum: 18.06.2020. URL: https://angular.io/guide/dependency-injection-providers.

Der PWA Showroom (o.D.). Abrufdatum: 18.06.2020. URL: https://pwa.bar/.

Dudenredaktion (o.D.[a]). *Duden: Plattform: Rechtschreibung, Bedeutung, Definition, Herkunft*. Abrufdatum: 02.06.2020. URL: https://www.duden.de/node/112297/revision/112333.

Dudenredaktion (o.D.[b]). *Duden: plattformunabhängig: Rechtschreibung, Bedeutung, Definition, Herkunft*. Abrufdatum: 02.06.2020. URL: https://www.duden.de/node/156293/revision/156329.

– (o.D.[c]). *Duden: Programmierung: Rechtschreibung, Bedeutung, Definition, Herkunft*. Abrufdatum: 02.06.2020. URL: https://www.duden.de/node/115328/revision/115364.

Erste Schritte mit Xamarin.iOS (o.D.). Abrufdatum: 04.06.2020. URL: https://docs.microsoft.com/de-de/xamarin/ios/get-started/.

Hierarchical injectors (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/hierarchical-dependency-injection.

HTML attribute vs. DOM property (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/template-syntax#html-attribute-vs-dom-property.

IndexedDB (o.D.). Abrufdatum: 05.06.2020. URL: https://developer.mozilla.org/de/docs/Web/API/IndexedDB_API.

Introduction to Angular concepts (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/architecture.

LePage, Pete (o.D.). *Ihre erste Progressive Web App*. Abrufdatum: 18.06.2020. URL: https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp.

NavigatorOnLine.onLine (o.D.). Abrufdatum: 16.06.2020. URL: https://developer.mozilla.org/de/docs/Web/API/NavigatorOnLine/onLine.

Pipes (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/pipes.

Plugin APIs (o.D.). Abrufdatum: 04.06.2020. URL: http://docs.phonegap.com/references/plugin-apis/.

PWA Stats (o.D.). Abrufdatum: 04.06.2020. URL: https://www.pwastats.com/.

Structural directives (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/structural-directives.

Terzibaschian, Arvid (o.D.). *Einführung in die Programmierung*. Abrufdatum: 20.06.2020. URL: https://www.cs.uni-potsdam.de/ml/teaching/ws13/epr/VL07.pdf.

The structured clone algorithm (o.D.). Abrufdatum: 05.06.2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

Types of feature modules (o.D.). Abrufdatum: 20.06.2020. URL: https://angular.io/guide/module-types.

Using Fetch (o.D.). Abrufdatum: 05.06.2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.

Web App Manifest (o.D.). Abrufdatum: 04.06.2020. URL: https://developer.mozilla.org/de/docs/Web/Manifest.

webhint charter (o.D.). Abrufdatum: 08.06.2020. URL: https://webhint.io/about/project_charter/.

Whitehorn, Jason (o.D.). *Data Synchronization: Patterns, Tools, & Techniques*. Abrufdatum: 05.06.2020. URL: https://www.datasyncbook.com/.

A. Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Datum, Ort, Unterschrift

B. Abbildungen

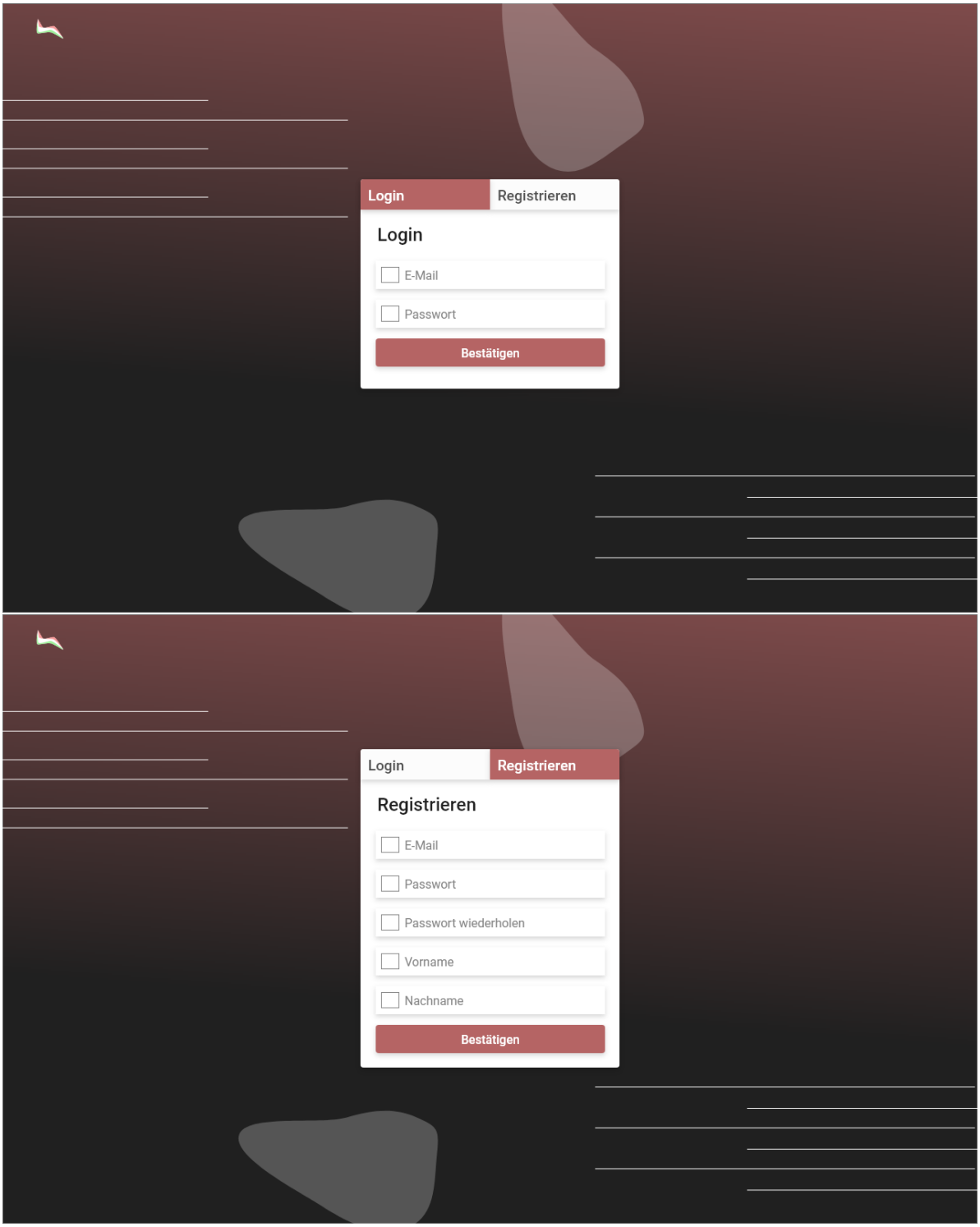

Abbildung B.1.: Beispiel - Layout Authentifizierung Desktop-Version

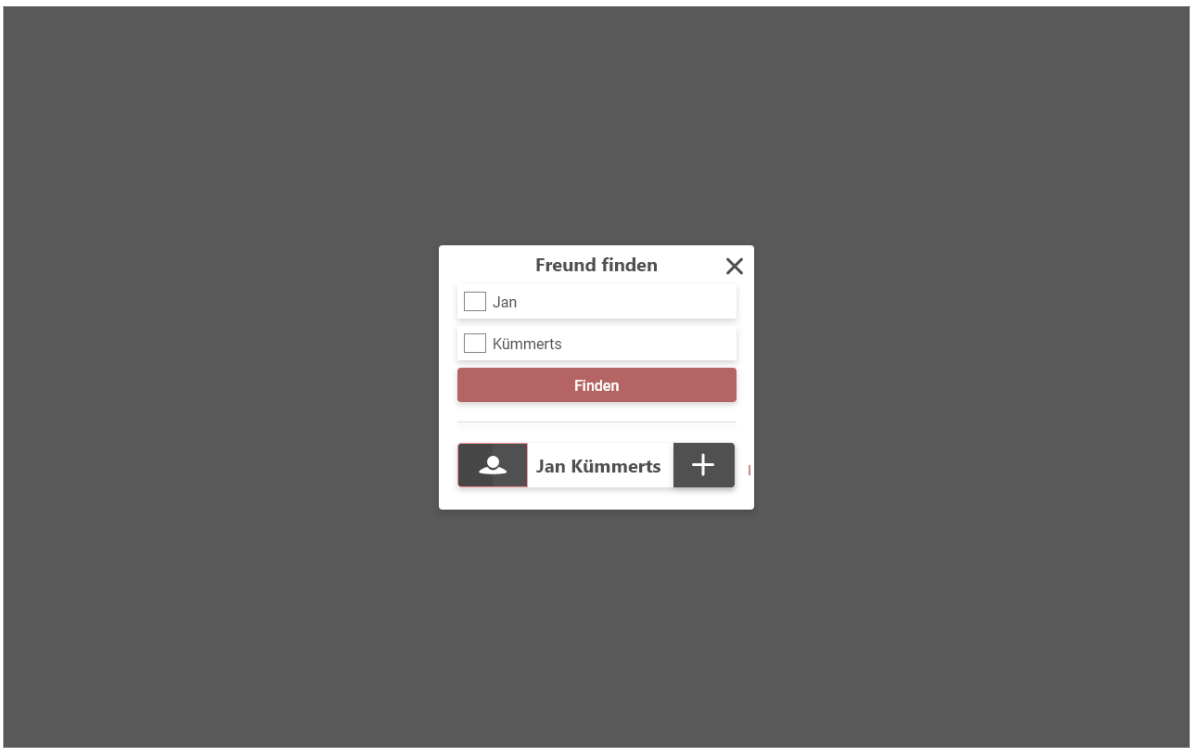

Abbildung B.2.: Beispiel - Layout Overlay Freund hinzufügen Desktop-Version

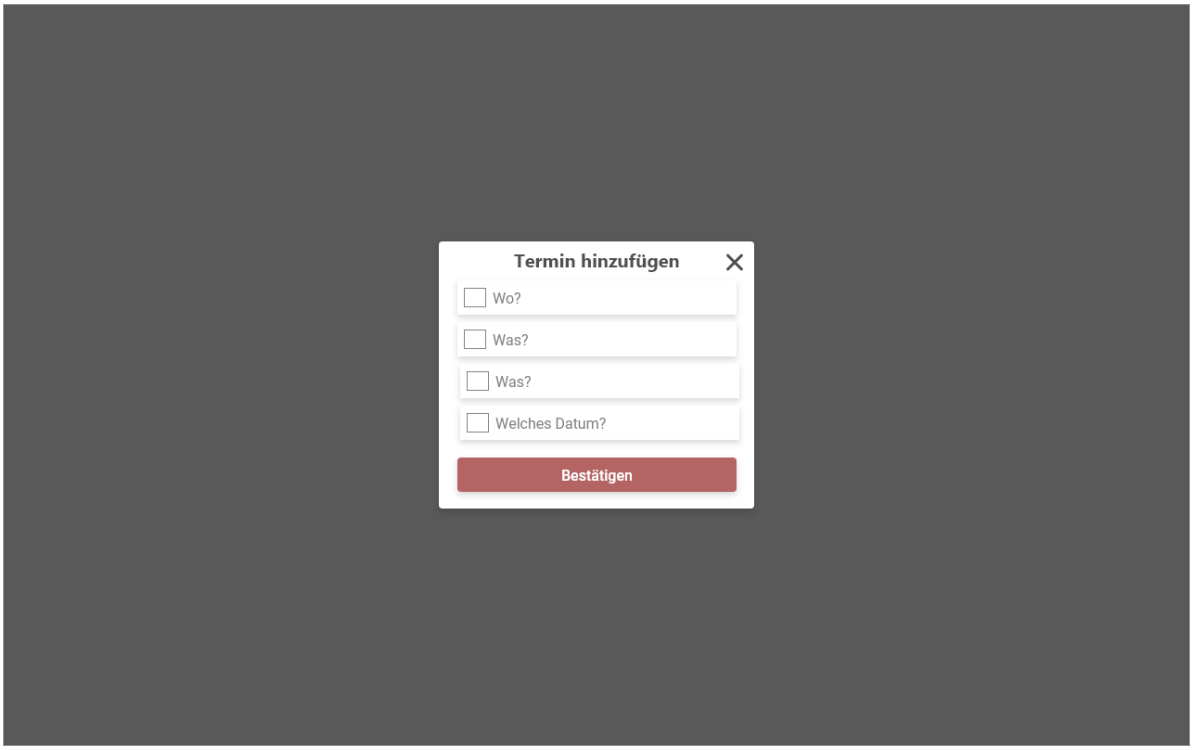

Abbildung B.3.: Beispiel - Layout Overlay Treffen/Termin hinzufügen Desktop-Version

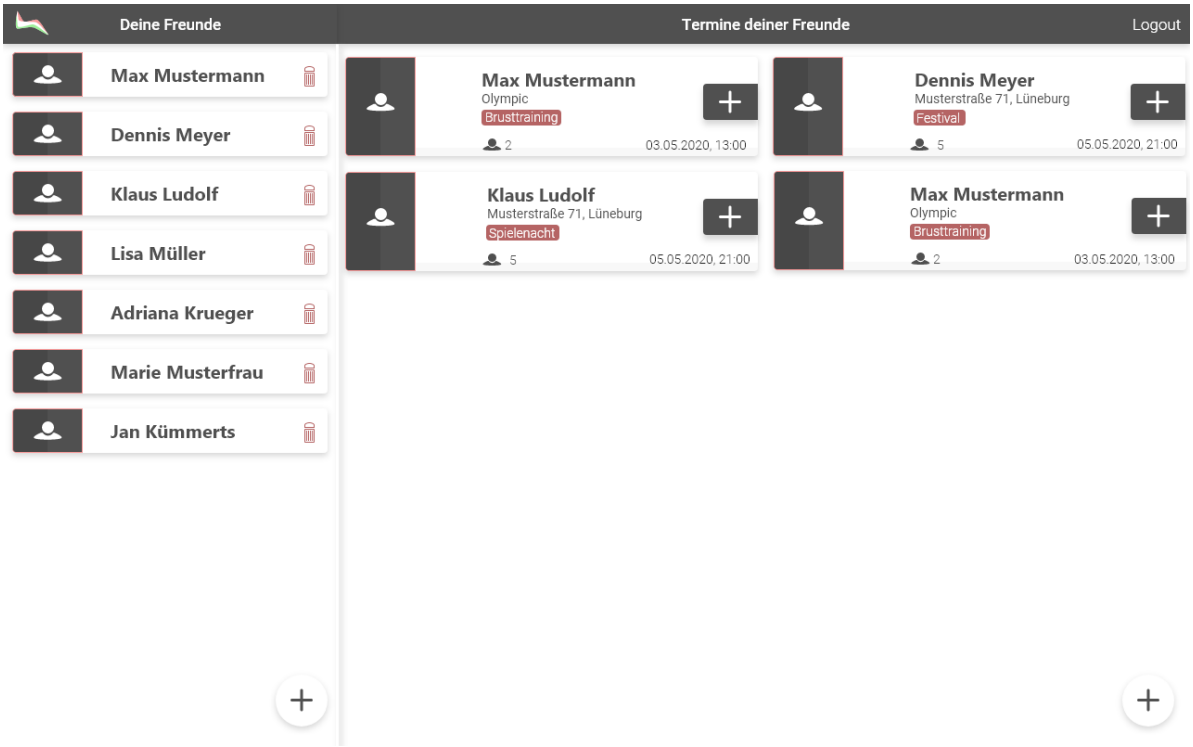

Abbildung B.4.: Beispiel - Layout Freunde und Termine/Treffen

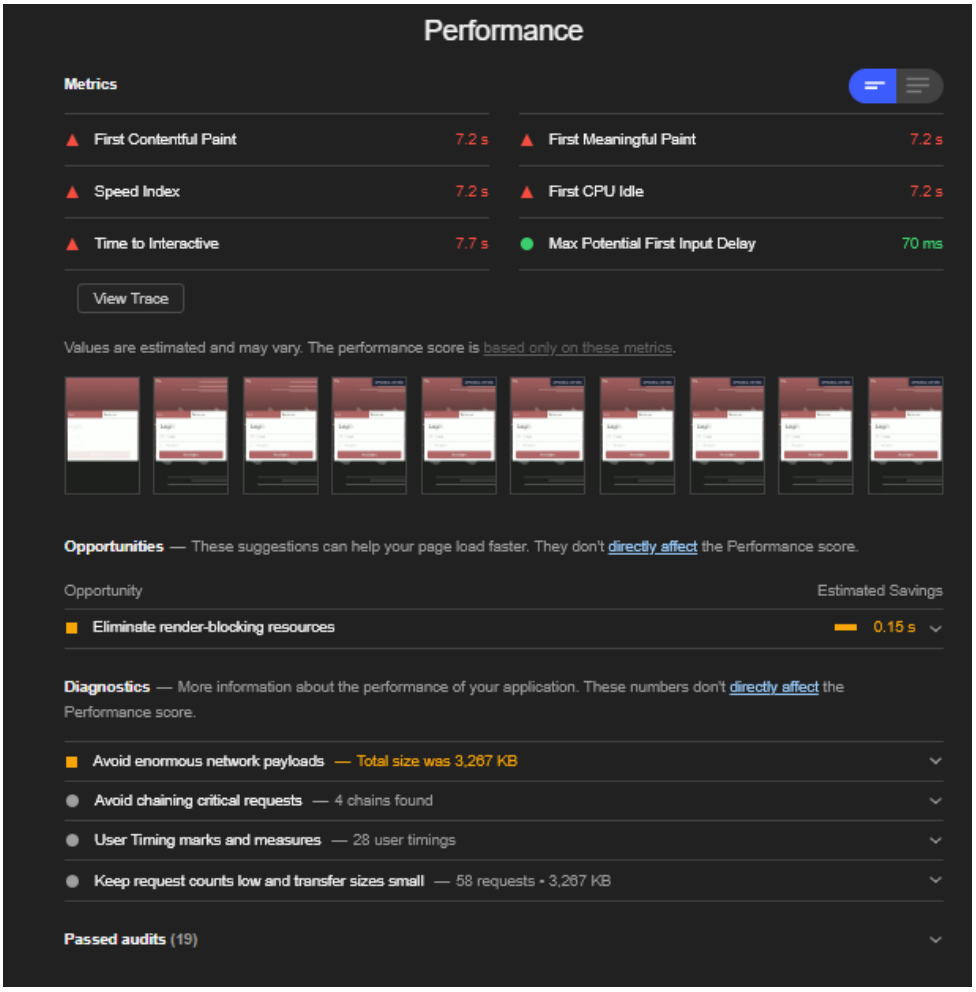

Abbildung B.5.: Beispiel - Lighthouse Schwachstellen der Performance