
Gegenüberstellung der Entwicklung nativer Mobilanwendungen und Progressive Web Apps (PWAs)

Studienarbeit (T3101)

für den Studiengang
Informatik

an der
**Dualen Hochschule
Baden-Württemberg
Stuttgart**

von
**STEFAN GOLDSCHMIDT
OLIVER RUDZINSKI**

Abgabedatum	08. Juni 2020
Matrikelnummer (Goldschmidt)	9760520
Matrikelnummer (Rudzinski)	5481330
Kurs	TINF17A
Hochschulbetreuer	Arne Heimeshoff
Studiengangsleitung	Prof. Dr. Dirk Reichardt Prof. Dr. Carmen Winter

Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema:

*Gegenüberstellung der Entwicklung
nativer Mobilanwendungen und
Progressive Web Apps (PWAs)*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

_____	_____	_____
Ort	Datum	Unterschrift Stefan Goldschmidt

_____	_____	_____
Ort	Datum	Unterschrift Oliver Rudzinski

Zusammenfassung

Apps, also Anwendungen für Smartphones, werden über einen Marktplatz des Betriebssystem-Herstellers vertrieben. Die Progressive Web App (**PWA**) bricht mit diesem Konzept und lässt Nutzer eine Webanwendung über den Browser auf dem Gerät installieren. Damit verspricht die **PWA** einige Vorteile gegenüber nativen Apps, wie Plattformunabhängigkeit und Verwendung von etablierten Webtechnologien.

Diese Arbeit vergleicht die Entwicklung nativer Apps mit derer **PWAs** und evaluiert die Technologien anhand mehrerer Kriterien. Dabei wird auf die technischen Grundlagen eingegangen und exemplarisch eine Todo-Anwendung für iOS und als **PWA** implementiert, welche anschließenden Evaluationsprozess stützt.

Abstract

Mobile apps are distributed via a marketplace managed by the OS vendor. The Progressive Web App (**PWA**) breaks with this concept and allows users to install web applications on the device using the browser. The **PWA** promises several advantages over native mobile apps, such as platform independence and use of established web technologies. This paper compares the development of native apps compared to PWAs and evaluates the technologies based on several criteria. The technical basics are discussed and an exemplary todo-application for iOS and as **PWA** is implemented, which supports the subsequent evaluation process.

Inhaltsverzeichnis

Abkürzungsverzeichnis	vi
Abbildungsverzeichnis	vii
Tabellenverzeichnis	viii
Quellcodeverzeichnis	ix
1 Einleitung	1
1.1 Kapitelübersicht	1
2 Grundlagen	3
2.1 Begriffsdefinitionen	3
2.2 Apps im Allgemeinen	4
2.3 Native Apps	5
2.3.1 Apple iOS	5
2.3.1.1 Entwicklung	5
2.3.1.2 Vertrieb von iOS-Applikationen	11
2.3.1.3 Nutzung von iOS-Applikationen	11
2.3.2 Android	12
2.3.2.1 Entwicklung	12
2.3.2.2 Vertrieb	12
2.4 Progressive Web Apps (PWAs)	13
2.4.1 Installation einer Progressive Web App (PWA)	14
2.4.2 Manifest-Datei für die Konfiguration der PWA	15
2.4.3 Service Worker für Offline-Funktionalität und Benachrichtigungen	16
2.4.4 PWA-unterstützende Plattformen	17
2.4.5 Grundlage der Webanwendung: JavaScript-Laufzeitumgebung <i>Node.js</i>	18
2.4.6 Front-End-Framework <i>Angular</i> für die Entwicklung von Webanwendungen	18
3 Wissenschaftliches Framework	20
3.1 Vorgehen bei der Bewertung	20
3.2 Betrachtete Aspekte der Entwicklung	21
3.3 Betrachtete Kriterien	22
4 Architektur	25
4.1 Anforderungsdefinition	26
4.1.1 Nicht-Funktionale Anforderungen	26

4.1.2	Funktionale Anforderungen	27
4.2	Speicherung der Daten	28
4.3	Benutzeroberfläche (UI)	29
4.4	Nutzungszyklus	31
5	Implementierung	33
5.1	Implementierung der nativen App	33
5.1.1	Gestaltung des User Interface	33
5.1.1.1	Einbinden von User Interface (UI)-Komponenten	33
5.1.1.2	Relative Positionierung der Komponenten	35
5.1.1.3	Dynamische Änderung der Komponentenposition	35
5.1.2	Entwicklung des Persistent Service	36
5.1.2.1	<code>createToDo(...)</code> : Anlegen von To-Do-Elementen	37
5.1.2.2	<code>loadToDos(...)</code> : Laden von To-Do-Elementen aus dem Speicher	37
5.1.2.3	Bearbeiten von Attributswerten von To-Do-Elementen	38
5.1.2.4	<code>deleteToDo(...)</code> : Löschen von To-Do-Elementen	38
5.1.3	Entwicklung der UI-Funktionalität	39
5.1.3.1	Verknüpfung der Create, Read, Update und Delete (CRUD)- Funktionen	39
5.1.3.2	Ergänzende UI-Operationen	40
5.1.4	Umsetzung von Benachrichtigungen über unerledigte To-Do-Einträge	41
5.1.4.1	Initialisierung und Genehmigung von Benachrichtigungen	41
5.1.4.2	Definition und Auslösung von Benachrichtigungen	42
5.2	Implementierung der PWA	43
5.2.1	Einrichten der Projektumgebung	43
5.2.2	Aufbau der Anwendung mit Angular	43
5.2.2.1	Bereitstellen der Klasse <code>TodoItem</code> zur Datenspeicherung	43
5.2.2.2	Implementierung des <code>todoService</code> für die Datenverwaltung	44
5.2.2.3	Beschreibung der Dateistruktur und Komponenten der Angular- Anwendung	45
5.2.2.4	Kommunikation der Komponenten mit dem <code>todoService</code>	47
5.2.2.5	Gestaltung des User Interface	48
5.2.2.6	Umsetzung der Benachrichtigung über unerledigte Aufgaben	49
5.2.3	Integration einer Manifestdatei	50
5.2.4	Hosting der App mit Firebase	50
5.2.5	Installation der Anwendung auf Smartphone und Desktop	51
5.2.6	Aktualisieren der Anwendung	52
6	Evaluation	53
7	Reflexion	54

Abkürzungsverzeichnis

API	Application Programming Interface
CLI	Command Line Interface
CRUD	Create, Read, Update und Delete
CSS	Cascading Style Sheets
CSV	Comma-separated Values
HTML	Hypertext Markup Language
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVC	Model-View-Controller
PWA	Progressive Web App
REST	Representational State Transfer
SCSS	Sassy CSS
SSL	Secure Sockets Layer
UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language

Abbildungsverzeichnis

2.1	Interface Builder in Xcode AppleIB	7
2.2	Zusammenspiel des Model-View-Controller (MVC)-Musters RayMVC	8
2.3	Dialogfenster zum Anlegen eines neuen Projektes (Xcode)	9
2.4	Browserdialog zur Installation einer PWA PWAAddToHomeScreenPrompt	15
2.5	Konzept des Service Workers ServiceWorkerDiagramm	17
2.6	MVC Konzept von Angular ProAngular	19
4.1	Wireframe der App	30
5.1	UI im Interface Builder von Xcode	34
5.2	Screenshot der iOS-App mit geöffneter Tastatur	42
5.3	Stylesheetformate beim Erstellen der Angularanwendung	43
5.4	Datenfluss bei Erstellung eines To-Do-Eintrages	45
5.5	Dateistruktur der Angularanwendung	46
5.6	Gerenderte todo-item -Komponente	47
5.7	Interaktion des Todo-service mit den Komponenten	48
5.8	Screenshot der PWA mit geöffneter Tastatur	49
5.9	Browser-Dialog zum Installieren der PWA als mobile App	51
5.10	Browser-Dialog zum Installieren der PWA als Desktop-App	51
5.11	Browser-Dialog für Benachrichtigungen	52

Tabellenverzeichnis

3.1	Verrechnungspunkte	21
3.2	Kriteriengewichtung	21
4.1	Attribute der ToDo-Entität	29
4.2	Farbtabelle	31

Quellcodeverzeichnis

1	Manifest-Datei einer PWA	16
2	Einbinden der Manifest-Datei	16
3	Funktion zur Erstellung von To-Do-Elementen (Swift)	37
4	Funktion zum Laden von To-Do-Elementen (Swift)	38
5	TodoItem-Klasse zur Datenspeicherung (TypeScript)	44
6	Klasse <code>TodoService</code> (TypeScript)	44
7	Hypertext Markup Language (HTML)-Template des <code>app-component</code> (Angular HTML)	46
8	HTML Template der <code>todo-item</code> (gekürzt)	47

1 Einleitung

Für das Jahr 2020 wird erwartet, dass 45 Prozent der Weltbevölkerung ein Smartphone nutzt **StatistaSmartphonesWorldwideStatistaWorldPopulation**. Unternehmen und Organisationen bedienen diesen Markt mit nativen Mobilanwendungen und mobil-optimierten Webseiten. Mit dem Konzept der **PWA** könnte diese zweigleisige Entwicklung bald auf die Webentwicklung reduziert werden: App und Webseite sind hier ein und dieselbe Anwendung.

Die **PWA** verspricht keine eingebettete Webseite, sondern Offline-Funktionalität, Entwicklung mit JavaScript mit Nutzung einer Vielzahl von Bibliotheken und Unabhängigkeit der Plattform und eine flüssige Nutzeroberfläche. Damit löst sie weitverbreitete Probleme von Webanwendungen, welche häufig die Geduld des Nutzers strapazieren. Hier müssen ständig Inhalte (Skripte, **CSS**-Stylesheets, Schriftarten, **HTML**, etc.) über eine langsame mobile Datenverbindung heruntergeladen werden.

Der hohe kombinierte Marktanteil von 80 Prozent des mobilen Browsers Google Chrome (60 Prozent in 2019) und Apple Safari für iOS (20 Prozent in 2019) inspiriert diese Arbeit, die Möglichkeiten der **PWA** mit dem Konzept der nativen App ausführlich zu vergleichen **StatistaMobileBrowserMarketShare**.

Mehrere Fallstudien von Google beschreiben einen positiven Effekt von **PWAs** auf Kunden **GoogleShowCases**. Diese Arbeit beschreibt nach ausführlicher Betrachtung der beiden Technologien **PWA** und native App die Vor- und Nachteile dieser Methoden, um ein grobes Bild der zukünftigen Entwicklung von Apps zu umreißen. Ferner behandelt sie die Forschungsfrage, ob native Apps durch **PWAs** langfristig ersetzt werden können.

1.1 Kapitelübersicht

In diesem Kapitel (**1**) wurden die aktuellen Marktentwicklungen kurz mit Zahlen benannt und die Motivation dieser Arbeit dargelegt. Das folgende Kapitel **2 (Grundlagen)** legt vorwiegend technischen Grundlagen für die spätere Implementierung einer Anwendung

als PWA und nativer App. Dabei wird auf die verwendeten Technologien und Frameworks eingegangen und speziell die PWA auf technischer Ebene erklärt.

Um den Entwicklungsprozess evaluieren zu können, wird in Kapitel 3 (**Wissenschaftliches Framework**) der Vergleichsprozess beschrieben und Kriterien mit ihrer Gewichtung aufgestellt und erläutert.

Kapitel 4 (**Architektur**) erläutert die Architektur der entwickelten Anwendung, die eine To-Do-Liste abbildet. In diesem Kapitel werden detaillierte Spezifikationen beschrieben, die den Entwicklungsprozess vergleichbar machen.

Anschließend werden im zweigeteilten Kapitel 5 (**Implementierung**) die Implementierungsprozesse der PWA und der nativen App detailliert dokumentiert.

Nach dem Sammeln von Erfahrungen bei der Implementierung werden in Kapitel 6 (**Evaluation**) beide Technologien mithilfe der Kriterien aus Kapitel 3 verglichen und evaluiert.

Abschließend gibt Kapitel 7 (**Reflexion**) ein Urteil über den Erfolg dieser Arbeit und gibt einen Ausblick auf die Zukunft der **PWA**.

2 Grundlagen

2.1 Begriffsdefinitionen

In der Praxis sind viele, in dieser Arbeit häufig verwendete, Begriffe unterschiedlich belegt. Aus diesem Grund soll im Folgenden eindeutig klargestellt werden, was mit den verwendeten Begriffen im Zuge dieser Arbeit gemeint ist.

App (plural: Apps) In dieser Arbeit werden Programme, die speziell für Smartphones entwickelt wurden, als *Apps* bezeichnet. Implizit wird hiermit auch die Plattform auf Android oder iOS eingrenzt. Der Begriff App zeichnet sich in dieser Arbeit dadurch aus, dass die damit gemeinte Anwendung für den Nutzer sehr einfach zu installieren ist. In der Regel beziehen Nutzer Apps aus einem zentralen Shop des Herstellers (bspw. Apples *App Store* oder Googles *Play Store*) und starten diese über ein Icon auf dem Startbildschirm des Betriebssystems.

Webseite, Webanwendung und Web App Der Begriff Webseite wird in dieser Arbeit für **HTML**-basierte Inhalte verwendet, die der Nutzer über den Browser abrufen.

Die Webanwendung unterscheidet sich dahingehend, dass sie dynamisch auf den Nutzer reagiert, seine Eingaben auswertet und gegebenenfalls den angezeigten Inhalt ändert oder nachlädt. Speziell werden JavaScript-basierte Anwendungen in dieser Arbeit als Webanwendung oder Web App bezeichnet. Web App und Webanwendung werden synonym verwendet.

Eine Webseite kann, aber muss keine Webanwendung oder Web App sein.

Progressive Web App (PWA) Eine Progressive Web App (**PWA**) ist eine Webseite und speziell eine Webanwendung oder Web App, welche dynamisch auf den Nutzer reagiert. In dieser Arbeit werden Webanwendungen, welche lokal auf einem Gerät installiert werden können als Progressive Web App (**PWA**) bezeichnet. Die **PWA** erfüllt die Kriterien des nachfolgenden Unterkapitels 2.4.

Im Unterschied zur nativen App kann dieselbe **PWA** sowohl auf Smartphones, als auch auf eine Desktopgerät (Notebook, Desktop-Computer, etc.) installiert werden.

Desktop PWA Mit Desktop PWA ist hier explizit eine Progressive Web App gemeint, welche auf einem Desktopgerät installiert wird.

native App Diese Arbeit beschäftigt sich mit einer modernen Methode, Mobilanwendungen zu programmieren, nämlich der PWA. Im Unterschied dazu ist eine native App in Java, Kotlin oder Swift geschrieben und ist damit stark plattformabhängig. Nativ implementierte Apps sind entweder für iOS oder Android entwickelt worden, nicht aber für mehrere Plattformen.

Container Da die Entwicklung von Apps stark am Front End orientiert ist, wird häufig der Begriff Container verwendet. Damit ist explizit *kein Container im Sinne von Virtualisierung*, wie z.B. ein Docker-Container, gemeint. Der Begriff wird im HTML-Kontext verwendet und bezeichnet in dieser Arbeit ein Element, das andere Elemente beinhaltet.

2.2 Apps im Allgemeinen

Anwendungen oder Programme, welche für Smartphones oder Tablets entwickelt wurden, werden gemeinhin als *Apps* bezeichnet. Der Name stammt hierbei aus dem Englischen: *App* ist die Kurzform von *Application*. Ins Deutsche könnte man *Application* wörtlich mit *Anwendung* übersetzen **BegriffApp**. Es handelt sich dabei um *High-Level*-Anwendungen mit umfangreicher Benutzeroberfläche. Im Gegensatz dazu stehen *Low-Level*-Anwendungen, wie bspw. Gerätetreiber.

Mit der Annäherung von Desktop-Computern über Notebooks an Tablets, welche eine Vielzahl mobiler Geräte, wie Touch-Notebooks, 2-in-1 Notebooks oder Tablets mit Tastatur und vollwertigem Betriebssystem auf den Markt brachte, ist der Begriff *App* nicht mehr scharf definiert. Längst bezeichnet Microsoft Windows Anwendungen, die über den Microsoft Store heruntergeladen werden können, als *Apps*. Es ist festzuhalten, dass im deutschen Sprachgebrauch *mobile Apps* für Smartphones und Tablets meist abgekürzt als *Apps* bezeichnet werden.

„Je nach Betriebssystem gibt es verschiedene *App-Stores*.“, schreibt das Gabler Banklexikon **BegriffAppGabler**. Sowohl unter Windows, als auch unter iOS und Android stellt der Betriebssystem-Hersteller Apps über einen Marktplatz zur Verfügung. Für dieses Konzept existieren mehrere Namen, im Umgangssprachlichen wird dieser Marktplatz aber meist als *App Store* bezeichnet, obwohl dies eigentlich der Eigenname des Anwendungsmarktplatzes von Apple ist. Ein solcher Marktplatz unterstützt Nutzer dabei, gezielt Apps zu suchen und zu installieren. Der Marktplatzbetreiber wickelt Zahlungen zwischen Nutzer und App-Entwickler ab und bietet Nutzern die Möglichkeit, Apps zu bewerten. Die

Progressive Web App (**PWA**) bricht mit diesem Konzept und lässt Nutzer eine Anwendung über den Browser installieren.

Zu jeder App existiert eine Verknüpfung bzw. eine Startschaltfläche auf dem Startbildschirm des Betriebssystems. Sie bildet einen der wichtigsten Bestandteile, welcher mit dem Wort *App* assoziiert wird: Ein einzigartiges, meist quadratisches Logo, dass die App per Fingertipp startet. Anschließend erwarten Nutzer einige Konventionen bezüglich der Nutzeroberfläche. Android Nutzer beispielsweise werden erwarten, dass der Hardwarebutton *zurück* zur vorherigen Ansicht zurückleitet. Apple iOS Nutzer werden hingegen eine sichtbare Menüführung mit kleinen Pfeilen zum Wechseln der Ansichten erwarten. Alle Nutzer werden meist annehmen, dass eine App nur dann *läuft* (also ausgeführt wird), solange er diese sieht und nicht geschlossen hat.

Zusammenfassend ist eine *App* eine Anwendung für Mobilgeräte, welche aus einem Marktplatz heruntergeladen wird und sich an herstellerspezifische Designrichtlinien hält.

2.3 Native Apps

Da im weiteren, v.a. praktischen Verlauf der Arbeit, die native App-Entwicklung für iOS-Geräte näher betrachtet wird, werden auch an dieser Stelle die Grundlagen der iOS-Entwicklung detaillierter betrachtet als jene der Android-Entwicklung.

2.3.1 Apple iOS

iOS ist ein Betriebssystem, welches vom US-amerikanischen Technologiekonzern *Apple, Inc.* im Rahmen des erstmalig vorgestellten *iPhones* im Jahre 2007, damals noch unter dem Namen *iPhone OS*, in Umlauf gebracht wurde. Stand heute läuft dieses Betriebssystem ausschließlich auf den Mobilgeräten des genannten Herstellers. Neben dem iPhone nutzt auch das Multimedia-Gerät *iPod touch* das Betriebssystem iOS. Der Tabletcomputer *iPad* wurde bis September 2019 ebenfalls über iOS betrieben, besitzt jedoch seit der Umstellung ein eigenes, an iOS stark angelehntes Betriebssystem *iPadOS*.

2.3.1.1 Entwicklung

In seiner ersten Version stellte iPhone OS noch keinerlei Möglichkeit bereit, Anwendungen von Drittanbietern bereitzustellen sowie zu nutzen. Dies änderte sich bei der Umbenennung des Betriebssystems in iOS im Jahre 2008, welche auch ein Software-Update zur Folge hatte, in welchem diese Eigenschaft nun geboten wurde.

Programmiersprache: Apple *Swift*

In der Anfangszeit der Anwendungsentwicklung für iOS wurde die bereits für andere Zwecke entwickelte und vorhandene Programmiersprache *Objective-C* als Standard gewählt. Dies änderte sich im Jahre 2014, als Apple bei seiner jährlichen Entwicklerkonferenz die hauseigene Programmiersprache *Swift* vorstellte, welche *Objective-C* in der ganzheitlichen Anwendungsentwicklung rund um Apple-Geräte ablösen sollte **AppleSwift**. In der Anfangszeit von *Swift* war diese immer noch stark an den Vorgänger *Objective-C* angelehnt. Über die Zeit sank der Einfluss, jedoch ist *Swift* weiterhin abwärtskompatibel zu *Objective-C* **AppleSwift**, welche wiederum abwärtskompatibel zu C ist.

Bei *Swift* handelt es sich um eine objekt- und protokoll-orientierte Programmiersprache, welche in Ihren verschiedenen Anwendungsbereichen ihre Zugehörigkeit zu verschiedenen Programmierparadigmen aufweist. Diese stützt sich vor allem auf ihrer Behauptung, möglichst leicht verständlich für einen Menschen zu sein **AppleSwift** und vermeidet bekannte Probleme anderer, populärer objektorientierter Programmiersprachen, bspw. Dereferenzierung von *null-Pointer-Exceptions*.

Neben der Entwicklung für alle Apple-Plattformen wird *Swift* unter anderem auch zur Back-End-Entwicklung genutzt. Laut einer Umfrage von *Stack Overflow* aus dem Jahr 2019 positioniert sich *Swift* auf Platz 15 der beliebtesten Programmiertechnologien, basierend auf 6,6 Prozent aller Stimmen **noauthor_stack_nodate**.

Entwicklungsumgebung: Apple *Xcode*

Xcode ist eine, ebenfalls von Apple entwickelte, sog. integrierte Entwicklungsumgebung (engl. Integrated Development Environment (**IDE**)) und wird primär für die Entwicklung von Anwendungen mit der Programmiersprache *Swift* eingesetzt. Grundsätzlich ist *Xcode*, und somit auch die Programmierung mit *Swift*, Apple *Mac*-Nutzern vorbehalten. Über die Zeit wurden weitere **IDEs** entwickelt, welche *Swift* unterstützen, denen jedoch grundlegende, nachfolgend beschriebene Funktionalitäten von *Xcode*, fehlen. Zu nennen ist hier bspw. die Lösung *AppCode* des Unternehmens *JetBrains*.

Xcode unterstützt neben *Swift* auch die abwärtskompatiblen Programmiersprachen *Objective-C*, C, aber auch C++, Python, Ruby, sowie andere. Darüber hinaus bietet *Xcode* einen sog. *Interface Builder*, mit welchem das Frontend über separate Ansichten (sog. *Views*) vorbereitet werden kann. Dabei werden verschiedenste Komponenten bereitgestellt, welche via *Drag'n'Drop* innerhalb der Ansichten platziert und optisch konfiguriert werden können. Auch Beziehungen zwischen den einzelnen *Views* können bereits hier angelegt werden. Ein Beispiel für den Aufbau des *Interface Builders* kann dem unten stehenden Screenshot entnommen werden.

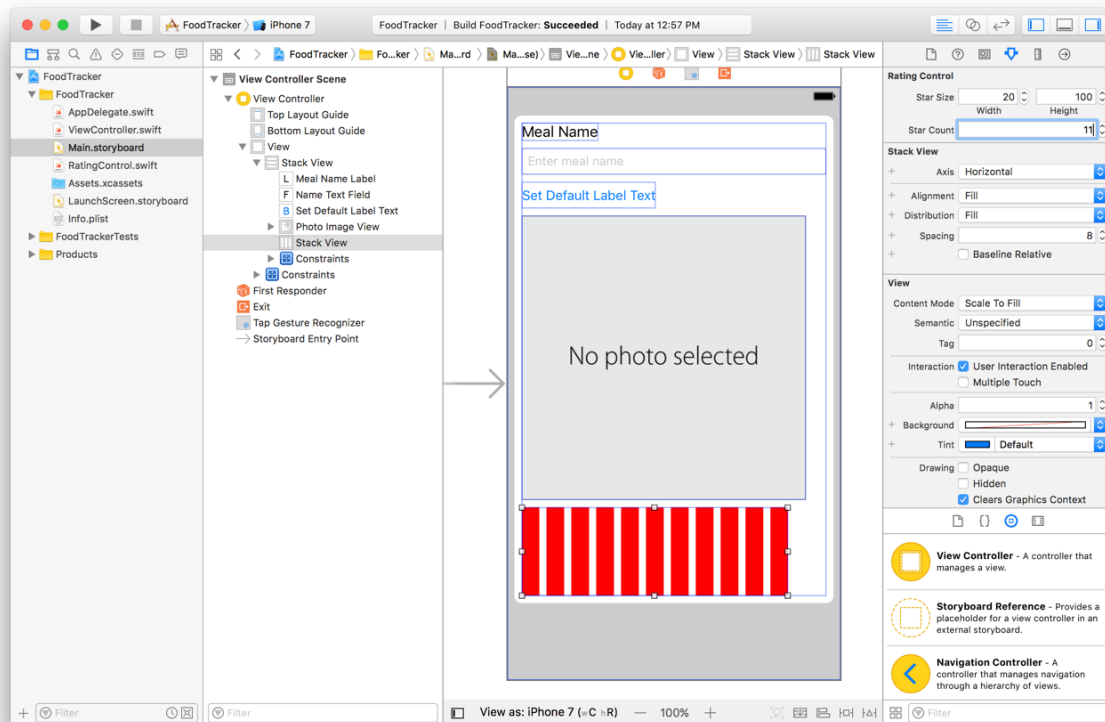
Abbildung 2.1: Interface Builder in Xcode **AppleIB**

Abb. 2.1 zeigt in der Mitte des Fensters den interaktiven Interface Builder. Über diesen kann die zu entwickelnde Anwendung bereits vor ihrer Laufzeit in ihren größten Zügen betrachtet werden. Durch die mehrschichtige Seitenleiste rechts ist es möglich, verschiedenste Konfigurationen an den entsprechenden Komponenten vorzunehmen, welche von unten rechts in die mittige Ansicht gezogen werden.

Die zusammengesetzten Komponenten können, ebenfalls über Drag-and-Drop in den Quellcode referenziert werden, um die Verhaltensweisen dieser programmatisch festzulegen. Ist die App in einem testreifen Zustand, kann diese direkt über Xcode emuliert werden. Es öffnet sich ein sog. *Simulator*, welcher das Zielgerät mit der geöffneten Anwendung darstellt. Das Verhalten und etwaige, daraus resultierende Probleme, können somit erkannt werden, ohne, dass es ein tatsächliches, physikalisches Zielgerät bedarf. Eigenschaften, welche auf die Hardware-Komponenten des Geräts zugreifen (bspw. die eingebauten Kameras, den Bluetooth-Sensor, etc.) können mit Ausnahme der Netzwerkkarte nicht simuliert werden.

Softwaredesign-Muster *Model-View-Controller (MVC)*

Apple empfiehlt als grundlegendes Prinzip für die App-Entwicklung mit Swift das Muster **MVC** **AppleMVC**, welches im Folgenden aufgeschlüsselt wird.

Model (dt. *Modell* bezieht sich auf das Datenmodell und die datenbedingte Kommunikation innerhalb der Anwendung. Innerhalb von der iOS-Anwendungsentwicklung finden sich hier Quellcode-Abschnitte für die Kommunikation der App mit einer potenziellen **API**, Code für die Definition und Bereitstellung von persistentem Speicher sowie die Handhabung der dadurch entstehenden Daten. Auch im Quellcode verwendete Konstanten sind dem Modell zuzuschreiben.

View (dt. *Präsentation*) bezieht sich auf die Frontend-Komponenten der Anwendung. Alle durch den Interface Builder spezifizierten Eigenschaften sind der Präsentation zuzuschreiben, also jegliche Komponenten, frontend-basierte Klassen sowie Animationen.

Controller (dt. *Steuerung*) beinhaltet die spezifische Verhaltenslogik der Anwendung bei Interaktion mit dieser durch den Nutzer. Vereinfacht formuliert bestimmt dieser, welche Funktionalität zu welcher Zeit auf Basis welches Verhaltens ausgeführt wird. Diese ist ebenfalls für die Kommunikation zwischen dem Modell und der Präsentation zuständig **LesevaAdams2016**.

Die Kooperation der einzelnen Komponenten dieses Musters lässt sich anhand der unten stehenden Abbildung erläutern.

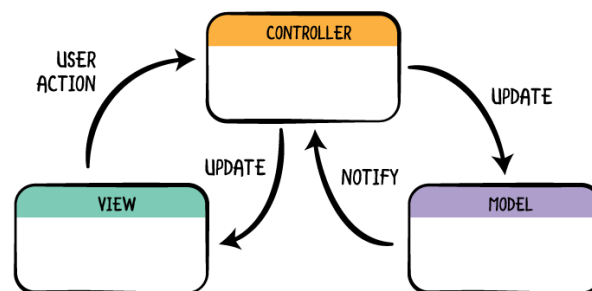


Abbildung 2.2: Zusammenspiel des **MVC**-Musters **RayMVC**

Technisch wird dieses Muster durch die Verknüpfung der einzelnen (durch den Interface Builder bereitgestellten oder erstellten) Views mit spezifischen, sog. *View Controllern* realisiert **AppleMVC**. Durch eine Aktion des Nutzers auf dem View wird die entsprechende Funktionalität über den View Controller bereitgestellt. Dieser kann dann ebenfalls Funktionen ausführen, welche durch das Modell bereitgestellt werden, falls die Aktion dies bedarf. Die entsprechenden Klassen der View Controller sind Unterklassen der einzelnen, spezifischen Views, welche deren vorgesetzte Basisfunktionalitäten um die gewünschten Funktionen des Entwicklers erweitern.

Vorbereitung der Projektumgebung

Bei der Erstellung eines neuen Projektes innerhalb von Xcode wählt der Entwickler bereits

zu Beginn bestimmte Spezifikationen für die zu entwickelnde Anwendung. Da Swift auch für andere Betriebssysteme von Apple zur Anwendungsentwicklung genutzt wird, gehört die Spezifizierung des Anwendungsbereiches (hier: iOS) sowie des grundlegenden Aufbaus der App zu den Vorbereitungsmaßnahmen der Projektumgebung. Dieses Dialogfenster kann aus dem unten stehenden Screenshot entnommen werden.

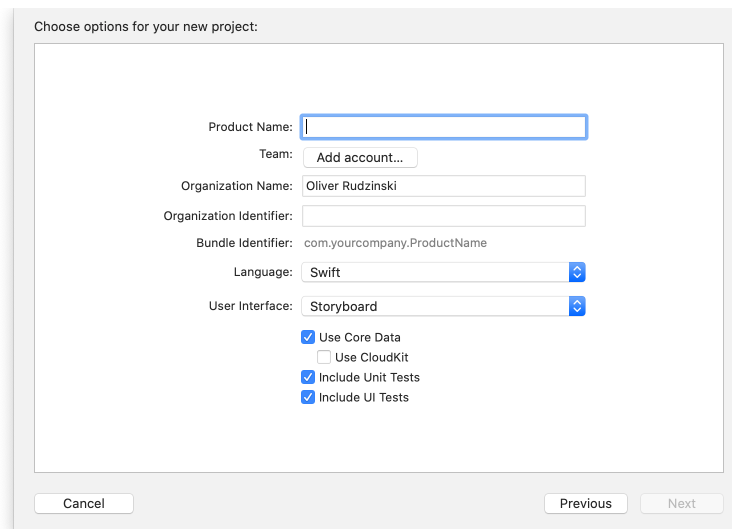


Abbildung 2.3: Dialogfenster zum Anlegen eines neuen Projektes (Xcode)

Im gezeigten Schritt der Projektvorbereitung trifft der Entwickler Entscheidungen über Projektverzeichnis, -namen und -sprache sowie über weitere Projektkomponenten, welche für die Anwendung von Bedarf wären.

Alle hier getroffenen Schritte können auch manuell bzw. im Nachgang der Projekterstellung angelegt werden. Grundsätzlich sind die Dateien und Quellcodezeilen, welche durch die genannten Vorbereitungsmaßnahmen automatisch generiert werden, unabdingbar für die Realisierung des Projekt und können somit für eine effizientere Arbeit des Entwicklers sorgen.

Struktur des initialen Entwicklungsverzeichnisses Bereits zu Anfang befinden sich bestimmte Dateien im Entwicklungsverzeichnis, welche sich grundsätzlich in jeder iOS-Anwendung benötigt werden. Diese werden im Folgenden beschrieben.

AppDelegate.swift ist der Eintrittspunkt der App. Dieser ist zuständig für das Verhalten der App, wenn diese (zum ersten Mal) geöffnet, geschlossen oder in den Hintergrund gerückt (d. h. inaktiv gesetzt) wird. Diese Datei ist vor allem dann von Interesse, wenn die Anwendung auch außerhalb ihrer Aktivität bestimmte Funktionen ausführt, bspw. also bei Musik-Anwendungen oder Stoppuhren. Aufgrund der

automatischen Bereitstellung von Core Data zu Anfang des Projektes ist hier auch das Grundgerüst des *Persistent Service* zu finden.

SceneDelegate.swift ähnelt dem **AppDelegate** sehr, bezieht sich jedoch auf das Verhalten der Ansicht(-en) der Anwendung. Diese wird benötigt, da seit iOS-Version 13 auch mehrere offene Instanzen derselben App möglich sind, welche jedoch immer den aktuellsten Zustand darstellen sollen, unabhängig von der gerade aktiv laufenden Instanz. In Form einer Analogie aus der Web-Entwicklung kann man den **SceneDelegate** als die oberste Hierarchie der *Frontend*-Steuerung bezeichnen, während das *Back End* über den **AppDelegate** orchestriert wird.

ViewController.swift ist einer von beliebig vielen möglichen Controllern, welche direkte Anbindung zu einer oder mehreren Ansichten hat. Dieser beinhaltet Konfigurationen der verschiedenen Komponenten der Ansicht und spezifiziert auch das Verhalten der Ansicht, wenn auf diese oder von dieser gewechselt wird.

Main.storyboard ist das Gegenstück zu den **ViewController**. In Form eines sog. *Interface Builders* können verschiedene Ansichten und deren Verknüpfungen via *Drag-and-Drop* erstellt werden. Jede Ansicht wird mit genau einem **ViewController** versehen, welcher die Steuerung der definierten Elemente übernimmt. Grundsätzlich ist es auch möglich, vollständig ohne Interface Builder auszukommen. Die initiale Konfiguration der Positionen und der weiteren optischen Eigenschaften wird dann ebenfalls über den entsprechenden **ViewController** gehandhabt, wobei dies einen deutlich höheren Programmieraufwand aufweisen könnte. Neben dem **Main.storyboard** existiert auch ein **LaunchScreen.storyboard**, welcher beim Laden der Anwendung ausgeführt wird.

Assets.xcassets beinhaltet das App-Logo, welches im Menü des Mobilgerätes angezeigt wird. Dieses muss in fest definierten Größen bereitgestellt werden, wird jedoch an dieser Stelle vernachlässigt. Darüber hinaus können hier auch alle weiteren Grafiken, etc., abgelegt werden, welche innerhalb der App genutzt werden, um für eine ordnungsgemäße Skalierung der Symbole je nach Bildschirmgröße des Endgeräts zu gewährleisten.

Info.plist ist eine allgemeine Konfigurationsdatei. Diese wird u.a. für Berechtigungen genutzt, welche die Anwendung außerhalb ihres eigenen Handlungsspielraums haben soll, bspw. die Nutzung der eingebauten Kameras, des GPS-Standortes des Geräts, die Berechtigung, Benachrichtigungen anzuzeigen, etc.

to_do.xcdatamodeld stellt die Konfigurationsdatei des Persistent Service dar, sofern dieser im Dialogfenster aktiviert wurde. In dieser werden Entitäten angelegt, mit möglichen Relationen versehen und für die Verwendung innerhalb der App exportiert.

2.3.1.2 Vertrieb von iOS-Applikationen

Der mit iOS 2.0 (Nachfolger von iPhone OS) ins Leben gerufene Apple *App Store* ist die einzige offizielle Bezugsstelle für iOS-Applikationen. Somit ist es auch der am meisten verwendete Marktplatz, in welchem Entwickler und Unternehmen ihre Anwendung kostenlos sowie kostenpflichtig zum Download anbieten.

Zunächst ist die Anmeldung des Entwicklers als *Apple Developer* im Vorhinein erforderlich. Ohne diese könnte die Anwendung zwar programmiert und innerhalb der Entwicklungsumgebung getestet, jedoch nicht auf tatsächlichen iOS-Geräten getestet und zum Download freigegeben werden. Dies hat sowohl sicherheitstechnische als auch wirtschaftliche Gründe, welche hier nicht weiter diskutiert, jedoch für die Evaluation der Entwicklungsfreiheit in Betracht gezogen werden. Letzteres erkennt man vor allem an der Tatsache, dass die Verbreitung von Anwendungen über den *App Store* kostenpflichtig ist **ApplePublish**.

Möchte ein Entwickler seine App im Apple App Store veröffentlichen, müssen bestimmte Vorkehrungen getroffen werden. Dazu gehört die Erstellung eines App-Icons, die Bereitstellung eines beschreibenden Textes für die App, wie dieser in der individuellen Seite im App Store dargestellt werden soll, eine definierte Altersbegrenzung, etc. Der Entwickler sendet die App via Xcode-Schnittstelle zur Evaluation aus. Diese wird von Apple getestet, bevor sie im App Store veröffentlicht wird. Auch die Beachtung der Veröffentlichungsrichtlinien von Apple wird an dieser Stelle überprüft **ApplePublish**.

Dieser Prozess wiederholt sich bei jedem Update, welches der Entwickler publiziert. Das Aktualisieren der App auf die neueste Version ist für die Nutzer meist optional, kann vom Entwickler jedoch vorausgesetzt werden; andernfalls kann die App nicht mehr vom Nutzer aufgerufen werden. Dies lässt sich jedoch nur gewährleisten, wenn eine aktive Internetverbindung seitens des Endgerätes des Nutzers vorhanden ist. Verschwindet eine App aus dem App Store, so verbleibt diese auf dem Gerät, bis sie gelöscht wird. Danach kann sie nicht erneut heruntergeladen werden.

Der genaue Prozess der Publikation von iOS-Applikationen kann im Rahmen dieser Arbeit nicht nachvollzogen werden, da keine Möglichkeit der tatsächlichen Bereitstellung der zu entwickelnden Beispielanwendung vorliegt.

2.3.1.3 Nutzung von iOS-Applikationen

Wie im vorigen Abschnitt erwähnt werden iOS-Anwendungen über den App Store bezogen. Auf weitere, inoffizielle und teils nonkonforme Praktiken wie das sog. *Jailbreaking* des

Gerätes, um auch nicht-authorisierte Anwendungen herunterladen und nutzen zu können, wird hier nicht eingegangen.

Der iOS-Nutzer sucht über den App Store die gewünschte App und lädt diese herunter. Für diesen Prozess ist eine sog. *Apple ID*, also ein Benutzerkonto bei Apple vonnöten. Sollte die gewünschte App kostenpflichtig sein, so müssen Kreditkarten- oder anderweitige, valide Zahlungsdaten dem Konto hinterlegt sein. Der Nutzer bestätigt den Kauf bzw. erstmaligen Download über das Benutzerkontenpasswort oder eine gerätespezifische Authentifizierungsmethode (bspw. Fingerabdruck-Erkennung via *Touch ID* oder Gesichtserkennung via *Face ID*, sofern vorhanden). Daraufhin startet der Download. Die App kann über ihr korrespondierendes App-Symbol, welches nun auf dem Menü-Bildschirm des Geräts erscheint, geöffnet werden.

Falls die App in einer nun aktuelleren Version vorliegt, wird diese entweder über den App Store automatisch im Hintergrund neu heruntergeladen. Andernfalls kann der Nutzer diesen Schritt auch manuell über den App Store in Kraft setzen.

2.3.2 Android

2.3.2.1 Entwicklung

Native Android-Anwendungen werden in Java entwickelt. Durch die Nutzung der zahlreichen Application Programming Interfaces (**APIs**) wird aus einem Java-Programm eine native Android App. **JavaForAndroid** Mittlerweile wird die teilweise veraltete Java-Syntax graduell durch die modernere Programmiersprache Kotlin abgelöst. **KotlinAndroid**

Meist werden Android Apps mithilfe der Entwicklungsumgebung *Android Studio* entwickelt, welches auf der IntelliJ-**IDE** von JetBrains aufbaut, jedoch von Google weiterentwickelt wird. Die **IDE** bietet Entwicklern unter anderem einen visuellen Layout-Editor (vergleichbar mit dem Interface Builder von Xcode) und eine Vielzahl von Android-Emulatoren zum Testen der Apps auf verschiedenen Android Versionen und unterschiedlicher Hardware. Dafür wird jedoch performante Hardware zum Entwickeln benötigt: Acht Gigabyte Arbeitsspeicher oder mehr ist die Empfehlung der Herausgeber. **AndroidStudio**

2.3.2.2 Vertrieb

Die meisten Apps beziehen Nutzer über den Google Play Store, einem Onlineshop für kostenlose und kostenpflichtige Android Anwendungen. Updates werden ebenfalls über den Play Store installiert. Alternativ kann ein Nutzer eine App in Form einer **.apk**-Datei

installieren. Dieser Weg bleibt jedoch aufgrund der Umständlichkeit und bedenklicher Sicherheit der App weitestgehend ungenutzt.

2.4 Progressive Web Apps (PWAs)

Dieses Unterkapitel erklärt die bereits erwähnte Progressive Web App (PWA) und nennt Kriterien, die von ihr erfüllt werden. Anschließend folgen die verwendeten Frameworks für die Entwicklung der PWA. Ein Überblick über diese ist essenziell für das Verständnis von Kapitel 5, in welchem die spezifischen Implementierungsschritte betrachtet werden.

Eine Progressive Web App (PWA) ist ein nächster Schritt nach der Dynamisierung statischer HTML-Seiten durch JavaScript und Front-End-Frameworks. Der Software-Entwickler und Autor MAJID HAJIAN charakterisiert PWAs mit acht Eigenschaften. Die wichtigsten dieser Charakteristiken werden im Folgenden zusammenfassend erläutert:

Installierbarkeit Der Nutzer einer Webanwendung kann diese lokal auf seinem Gerät installieren. Sie kann anschließend, wie eine native App, vom Startbildschirm aus gestartet werden. Um die Webanwendung zu nutzen muss ein Nutzer nun keinen Zwischenschritt mehr über den Browser tätigen.

Ähnlichkeit mit einer nativ implementierten App Klassischerweise werden Android-Apps in Java bzw. neuerdings Kotlin und iOS-Apps in Swift programmiert. Die PWA soll, wie eine native App, auf die Hardware des Mobilgerätes zugreifen können (beispielsweise für die Nutzung des Bluetooth-Chips). Außerdem unterscheidet sich das UI der PWA nicht maßgeblich von der nativen App.

Offline-Verwendung Die PWA soll unabhängig von einer Netzwerkverbindung funktionieren. Sie ist nach dem *Offline-First*-Design konzipiert. Die Google-Chrome Dokumentation für Cloud-Entwickler beschreibt Offline-First-Apps als Webanwendungen, deren Dateien (HTML, CSS, JavaScript etc.) bereits heruntergeladen sind. Daten werden temporär über eine Browser-Schnittstelle gespeichert und bei Bedarf synchronisiert. Außerdem kann die Anwendung auf eine unterbrochene Netzwerkverbindung reagieren **GoogleOfflineApps**. Die PWA ist demnach eine Webanwendung, die sowohl online, als auch offline nutzbar ist.

Optimierung für Mobilgeräte Die PWA ist für die (meist leistungsschwache) Mobilhardware konzipiert und funktioniert hierauf ohne Performanzprobleme. HAJIAN legt besonders auf das schnelle Laden beim Start der Anwendung wert.

Informierung des Nutzers Wie native Apps kann die PWA den Nutzer über Push-Nachrichten informieren oder zur Interaktion auffordern **Hajian2019**.

Diese Charakteristika decken sich mit der Beschreibung durch die Entwicklerdokumentation der **PWA** von Google. Im Vergleich zu HAJIAN ist diese etwas spezifischer und erwähnt bspw. die Kontrolle des Anwendungs-Caches durch einen JavaScript-Service-Worker (siehe Abs. 2.4.3), um die Abhängigkeit von einer Netzwerkverbindung aufzuheben. **GooglePWAOverview**

2.4.1 Installation einer Progressive Web App (**PWA**)

Diese Arbeit betrachtet die **PWA**, da sie (wie auch eine native Anwendung) lokal auf einem Gerät installiert werden kann. Es ist dafür kein zentraler Bezugspunkt nötig; die Installation wird über den Browser gestartet. Die Aufforderung zur Installation einer **PWA** kann entweder über den Browser (siehe Abb. 2.4) oder über ein Element der Website erfolgen, dass ein Event erzeugt, wie bspw. einen Button oder einen Dialog.



Abbildung 2.4: Browserdialog zur Installation einer **PWA**
PWAAddToHomeScreenPrompt

Zwar bezeichnen Browser die Installation meist nur mit der Aufschrift *Zum Startbildschirm hinzufügen*, tatsächlich generiert der Browser aber dann eine WebAPK, welche auf dem Gerät installiert wird. Auf Desktopgeräten startet die **PWA** in einem eigenen, stark verschlankten Browserfenster ohne Suchleiste und Bedienelemente. **GooglePWAInstallation**

2.4.2 Manifest-Datei für die Konfiguration der **PWA**

Um die **PWA** auf einem Gerät installieren zu können, muss ein sog. *Web-App-Manifest* zur Verfügung gestellt werden. Dieses ist eine JavaScript Object Notation (**JSON**)-Datei, welche Konfigurationsparameter für die Anwendung enthält. **GooglePWAManifest**

```
1 {
2   "short_name": "Maps",
3   "name": "Google Maps",
4   "icons": [
5     {
6       "src": "/images/icons-192.png",
7       "type": "image/png",
8       "sizes": "192x192"
9     },
10    {
11      "src": "/images/icons-512.png",
12      "type": "image/png",
13      "sizes": "512x512"
14    }
15  ],
16  "start_url": "/maps/?source=pwa",
17  "background_color": "#3367D6",
18  "display": "standalone",
19  "scope": "/maps/",
20  "theme_color": "#3367D6"
21 }
```

Quellcode-Ausschnitt 1: Manifest-Datei einer PWA

Quellcode-Ausschnitt 1 zeigt den Inhalt einer Manifest-Datei. Neben diversen Icons (Zz. 4-15) werden auch *Name* (Z. 3), *Farbschema* (Z. 20) und *Anzeigeeinstellungen* (Z. 18) festgelegt. Die Manifest-Datei wird im **HTML** der Webanwendung eingebunden, siehe Quellcode-Ausschnitt 2.

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

Quellcode-Ausschnitt 2: Einbinden der Manifest-Datei

Es ist die Einfachheit dieses Prozesses hervorzuheben: Das Hinzufügen einer (wenige Zeilen langer) **JSON**-Datei macht die gesamte Webanwendung installierbar. Es wird kein App-Store, manueller Dateidownload oder Installer benötigt.

2.4.3 Service Worker für Offline-Funktionalität und Benachrichtigungen

Damit die **PWA** trotz fehlender Netzwerkverbindung funktioniert, wird ein besonderer Mechanismus benötigt: Der sog. *Service-Worker*. Mit ihm können Abhängigkeiten der App lokal gecacht werden, sodass die Anwendung auch bei schlechter oder gar fehlender Netzwerkverbindung funktioniert **BeginningPWA**.

Ein Service-Worker ist ein vom **UI** separiert laufendes Hintergrundskript der Webanwendung (s. Abb. 2.5). Er wird genutzt, um Bilder, Skripts, Styles oder ganze Seiten zu cachen. Bei bestehender Netzwerkverbindung führt er nötige Synchronisierungen durch. Nicht zuletzt ist er auch für das Senden von Push-Benachrichtigungen zuständig **BeginningPWA**.

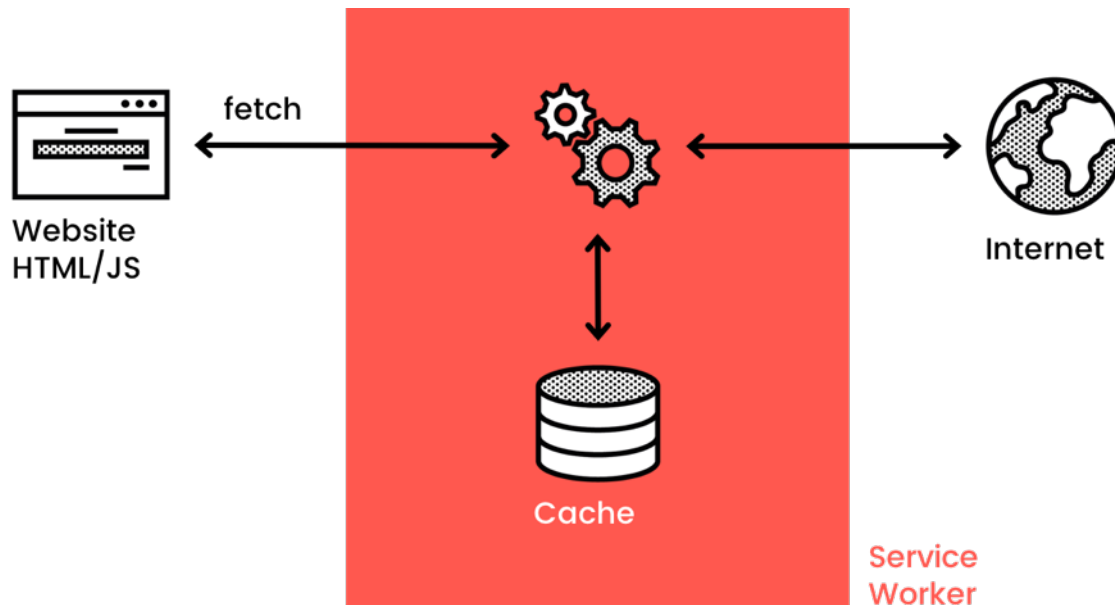


Abbildung 2.5: Konzept des Service Workers **ServiceWorkerDiagramm**

Alle verbreiteten Desktop-Browser wie Google Chrome, Mozilla Firefox, Opera, Microsoft Edge und mittlerweile auch Apple Safari unterstützen das Service-Worker-Konzept. Der mobile Chrome-Browser unter Android unterstützt Service-Worker bereits vollständig, während Safari unter iOS noch an diesem Feature arbeitet. **BeginningPWA**

2.4.4 **PWA**-unterstützende Plattformen

Das Projekt *CanIUse* aggregiert Daten zu Webstandards des W3-Konsortiums sowie zu Browser-Dokumentationen. Es wird als Quelle für die Unterstützung von Features durch aktuelle Browser herangezogen.

Apples mobiler Browser Safari unterstützt eines der wichtigsten Features der **PWA** noch nicht vollständig; es fehlen Teile des Web-App-Manifests. Allerdings wird der Service-Worker vollständig unterstützt **CanIUseWebManifest**. Wann und ob Safari die Unterstützung für das Manifest implementiert, ist unklar und bleibt abzuwarten. Die aktuelle Teilunterstützung lässt jedoch vermuten, dass sich Apple nicht grundsätzlich gegen die **PWA** weigert. Die Nutzung von **PWAs** ist nicht ausschließlich auf Smartphones begrenzt. Wie *normale* Desktop-Programme werden Desktop-**PWAs** in einem eige-

nen Fenster gestartet. Der Unterschied zwischen den Bedienelementen nativer Desktop-Anwendungen und Desktop-PWAs ist ausschließlich farblicher Natur. Stark vereinfacht beschrieben sind Desktop-PWAs Browserfenster ohne Tabs und Adressleiste. Durch die Nutzung von Service-Workern, welche die Webanwendung cachen, sind auch Desktop-PWAs nicht von einer Netzwerkverbindung abhängig.

Grundsätzlich können Desktop-PWAs auf jedem Betriebssystem installiert werden, auf welchem auch Google Chrome (Version größer 73) installiert werden kann: Windows, Mac, Linux und Chrome OS. **GooglePWADesktop**

2.4.5 Grundlage der Webanwendung:

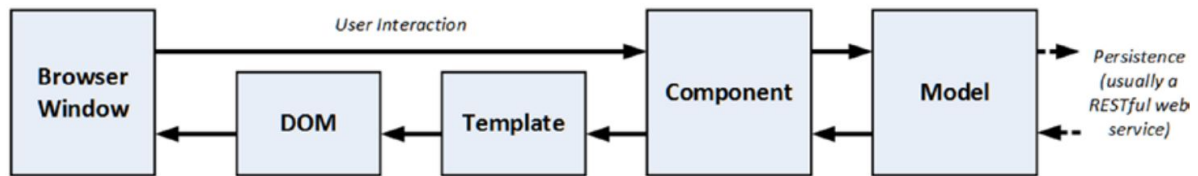
JavaScript-Laufzeitumgebung *Node.js*

Node.js ist eine open-source JavaScript-Laufzeitumgebung für die Entwicklung skalierbarer Webanwendungen **NodeJSWebsiteAbout**. Selbst baut Node.js auf der *V8-Engine* auf, einer Laufzeitumgebung, die auch von Google Chrome genutzt wird **NodeJSRecepies**. Wegen zeitsparenden Features, wie automatischem Typecasting oder der Tatsache, dass Node.js alle Daten als Objekt behandelt, erfreut sich Node.js großer Beliebtheit **PracitalNodeJS**. Die Kombination mit dem *Package Manager* **npm** ermöglicht die einfache Installation und Nutzung von Modulen, um die Funktionalität der Plattform zu erweitern. **NodeJSRecepies**.

2.4.6 Front-End-Framework *Angular* für die Entwicklung von Webanwendungen

Angular ist ein open-source TypeScript-basiertes Framework zur Entwicklung von Webanwendungen, welches Node.js nutzt. Mit über Achttausend bzw. über Siebentausend mitwirkenden Entwicklern belegen das Angular Command Line Interface (**CLI**) resp. das Angular-Framework die Plätze 4 und 6 der größten Projekte auf GitHub. **OctoverseGitHubStatistics**

Das Framework arbeitet auf Basis von Komponenten. Ein Eingabefeld, eine Seite oder eine Liste werden in Angular als separate Komponenten betrachtet. Auch in der Dateistruktur werden Komponenten stark getrennt. Jede Komponente besitzt bspw. ein eigenes **CSS-** (oder **SCSS**) und HTML-File. Eine Komponente für eine Seite kann so auch eine oder sogar mehrere Listenkomponenten einbinden. Durch die Wiederverwendung von Quellcode-Fragmenten in Komponenten wird der Programmquellcode sehr übersichtlich und strukturiert.

Abbildung 2.6: MVC Konzept von Angular **ProAngular**

Eine Angular-Anwendung ist wie folgt in drei Einheiten gegliedert:

Model enthält Logik für die Verwaltung von Daten, bspw. das Erstellen, Speichern oder Modifizieren. Dies kann über die Kommunikation mit einem Webserver via **REST-API** erfolgen. Das Model enthält keine Logik, um mit dem Nutzer zu interagieren.

Component enthält Logik für das Aktualisieren der Daten im Model aufgrund Nutzerinteraktion.

Template enthält Logik und Markup, um dem Nutzer Daten anzeigen zu können **ProAngular**.

3 Wissenschaftliches Framework

Die Grundlage für eine adäquate Bewertung der zu entwickelnden mobilen Anwendungen bildet die Definition beidseitig anwendbarer, quantifizierender Kriterien. Die Notwendigkeit hierin begründet sich darin, dass kein allgemein anerkannter Kriterienkatalog für die Pauschalisierung von Softwarequalität existiert. Viel eher benötigen unterschiedliche Projekte mit unterschiedlichen Schwerpunkten eine unterschiedlich definierte Definition der entsprechenden Gesichtspunkte.

Wie zuvor erwähnt wird die Forschungsfrage, ob **PWAs** native Apps langfristig ersetzen können, vor allem auf Basis der Umsetzbarkeit in der Entwicklung evaluiert. Umgekehrt könnten ähnliche Fragestellungen anhand wirtschaftlicher Aspekte begründet werden (bspw. Anzahl der benötigten Entwickler, preislicher Rahmen, Profitmöglichkeiten, etc.). Bei der Einfachheit der Beispielanwendung haben solche Aspekte jedoch kaum Bedeutung. Ebenfalls in Betracht gezogen werden muss die Tatsache, dass die Verfasser dieser Studienarbeit keine erfahrenen Entwickler ersetzen, weswegen auch quantifizierende Kriterien wie die Dauer der Entwicklungszeit, etc., keine Anwendung in der zugrunde liegenden Bewertung finden können. Gleiches gilt ebenfalls für die nur empirisch evaluierbaren Punkte der Nutzung seitens der Anwender (z.B. Intuitivität, Benutzerfreundlichkeit, Anspruch der Gestaltung, etc.), da eine solche Studie den Rahmen dieser Arbeit verlassen würde.

Bezüglich der Umsetzung in der Entwicklung lassen sich jedoch mehrere Kriterien aufstellen, die zur Beantwortung (oder zumindest Lenkung) der Forschungsfrage beitragen können. Die Definition der allgemeinen technischen Architektur in Kapitel 4 der zu entwickelnden Mobilanwendung geschieht unter Berücksichtigung der Kriterien, um eine neutrale Vergleichbarkeit zu ermöglichen.

3.1 Vorgehen bei der Bewertung

Um die Entwicklungen der Angular-**PWA** und der nativen iOS-App zu vergleichen, wird die im Folgenden beschriebene Kriteriengewichtung (siehe Tabelle 3.2) verwendet. Einzel-

ne Kriterien werden anhand Tabelle 3.1 bewertet und die Verrechnungspunkte anschließend über alle betrachteten Kriterien aufsummiert.

Bewertung	--	—	○	+	++
Beschreibung	schlecht	eher schlecht	neutral	eher gut	gut
Verrechnungspunkte	−2	−1	0	1	2

Tabelle 3.1: Verrechnungspunkte

Das Ergebnis ist eine Evaluationsmatrix, welche als Netzdiagramm dargestellt die bewerteten Kriterien einzeln visualisiert und eine Gesamtbewertung pro Technologie liefert. In diese fließen die Kriterien einzeln ein und bilden ein Gesamtbild. Die Gewichtung der Kriterien ist in Tabelle 3.2 dargestellt.

Kriterium	Gesamtanteil
Anwendung	
Plattformabhängigkeit	10%
Installation	5%
Speicherzugriff	5%
Speicherbedarf	5%
Aktualisierbarkeit	5%
Konsistenz des Designs	5%
Entwicklung	
Bibliotheken	10%
Umsetzung	20%
Testbarkeit	10%
Vorausgesetzte Entwicklungserfahrung	10%
Summe	100%

Tabelle 3.2: Kriteriengewichtung

3.2 Betrachtete Aspekte der Entwicklung

Bei der Evaluation soll auf verschiedene Aspekte beider Projekte eingegangen werden. Diese lauten wie folgt:

Anwendung Die Kriterien zur Anwendung (bzw. App) beziehen sich auf die Eigenheiten der Apps, darunter die Installation, die Aktualisierung und die Plattformabhängigkeit.

Entwicklung Kriterien der Entwicklung beziehen sich auf die Programmierung und Umsetzung der funktionalen und nicht-funktionalen Anforderungen.

Grundsätzlich gibt es einen Punktabzug, wenn der Nutzer grundlegenden Funktionen explizit zustimmen muss. Das Idealbild ist eine sofort nutzbare Anwendung, die ohne weitere Zwischenschritte den kompletten Funktionsumfang besitzt.

3.3 Betrachtete Kriterien

Die einzelnen Kriterien aus der Evaluationsmatrix werden nun genauer definiert:

Plattformabhängigkeit Unterstützt die Anwendung mehrere Plattformen, also beispielsweise Android und iOS wird dies als gut bewertet. Ist die Anwendung auch auf Desktop-Computern oder Tablets nutzbar, gibt dies ebenfalls eine positive Bewertung.

Installation Die App sollte ohne mehrere Zwischenschritte nutzbar sein. Ist die Installation zu kompliziert, führt dies zu Punktabzug.

Es fließt mit ein, wie viel Aufwand betrieben werden muss, um die App einem Publikum zur Verfügung zu stellen. Dieser Aufwand ist idealerweise gering und im Interesse des Entwicklers.

Speicherzugriff Anwendungen müssen Daten speichern können, um dem Nutzer einen Mehrwert zu bieten. Die Größe der zu speichernden Daten ist bei diesem Vergleich auf einige Kilobyte begrenzt. Gibt es die Möglichkeit Dateien abzulegen, ist dies positiv zu werten. Idealerweise können Daten in gängigen Formaten (beispielsweise **JSON**, **CSV**, **XML** oder Plaintext) gespeichert werden. Können Daten nur temporär und nicht persistent gespeichert werden, führt dies zu starkem Punktabzug. Muss der Nutzer dem Speichern von Daten aktiv zustimmen, stört dies die Nutzungserfahrung und ist daher negativ zu werten.

Speicherbedarf Der Speicherplatz eines Smartphones ist deutlich kleiner, als der eines Desktop-Computers. Bestenfalls ist die Anwendung nur einige Megabyte groß und kann so schnell über eine mobile Datenverbindung installiert und upgedatet werden **AppleMaxAppSize** **GoogleMaxAppSize**. Hoher Speicherverbrauch führt zu Punktabzug, wohingegen geringer Speicherverbrauch positiv bewertet wird. Der Verbrauch ist unter den einzelnen Plattformen relativ zu bewerten.

Aktualisierbarkeit Idealerweise kann die Installation von Updates vom Entwickler kontrolliert werden. Schnelle Updatezyklen sind wünschenswert, da in der Praxis dadurch schnell Sicherheitslücken und Bugs behoben werden können. Wenn ein Nutzer

sich aktiv gegen Updates weigern kann, oder diese manuell installieren muss, könnte dies für Kompatibilitätsprobleme mit Webschnittstellen oder Sicherheitsprobleme sorgen und führt deshalb zu Punktabzug.

Der Aufwand der betrieben werden muss, um Updates einzubringen, wird mitevaluiert. Je schneller Updates flächendeckend auf den Geräten der Nutzer landen, desto höher die Punktzahl.

Design Idealerweise sieht die Anwendung auf verschiedenen Geräten identisch aus. Grundsätzlich wird erwartet, dass Schriften, Farben und Größenverhältnisse auf unterschiedlichen Geräten und gegebenenfalls unterschiedlichen Browsern ein konsistentes Bild ergeben. Eine skalierende Nutzeroberfläche ist wünschenswert. Anzeigefehler, wie bspw. überlappende Objekte, fehlerhafte Elemente oder fehlende Schriftarten, führen zu Punktabzug. Auch der Aufwand, welcher betrieben werden muss, um das Nutzerinterface skalierbar zu gestalten, fließt in die Bewertung mit ein.

Bibliotheken Um den Programmier- und Wartungsaufwand zu minimieren, greifen Entwickler auf Bibliotheken zurück, welche Lösungen für verbreitete Probleme anbieten. In der Evaluation werden Bibliotheken von Open-Source-Organisationen und ggf. Bibliotheken des Plattformanbieters (bspw. Apple, Google, Mozilla, etc.) betrachtet. In die Bewertung fließt ein, wie komplex sich der Prozess für Entwickler darstellt, um Bibliotheken zu nutzen und zu installieren.

Umsetzbarkeit Das bewertete Kriterium der Umsetzbarkeit beinhaltet die Komplexität und die Herausforderungen bei der Implementierung der funktionalen und nicht-funktionalen Anforderungen. Sind Anforderungen aufgrund plattformspezifischer Einschränkungen nicht oder nur beschränkt umsetzbar, führt dies zu Punktabzug. Es wird angenommen, dass die Anforderungen mit allen etablierten Technologien für die App-Entwicklung vollständig umgesetzt werden können.

Bietet die Plattform im Umkehrschluss einfache und schnelle Mechanismen zur Umsetzung der Anforderungen, fließt dies positiv in die Wertung mit ein.

Testbarkeit Das Testen von Software gehört zu den Grundlagen der Qualitätssicherung. Es wird erwartet, dass es einfache und schnelle Methoden zum Testen der Apps nach einer Änderung im Quellcode gibt.

Vorausgesetzte Entwicklungserfahrung Es soll eingeschätzt werden, wie hoch die Einstiegshürde für die Entwicklung der jeweiligen Technologie ist. Die Anzahl und Komplexität der Tools fließt in die Bewertung mit ein. Es wird angenommen, dass die Entwicklung mit einem einzigen Werkzeug für den Entwickler einfacher ist, als das Bedienen mehrerer komplexerer Entwicklungstools. Gibt es grafische Oberflächen

für viele Entwicklungsschritte ist dies positiver zu bewerten, als das Arbeiten mit Skripten und der Kommandozeile.

4 Architektur

Die hier zu entwickelnde Anwendung dient zum Anlegen und Verwalten von Aufgaben der Nutzer. Somit löst sie die sogenannte, analoge *To-Do-Liste* ab. Dabei hat die Anwendung (und somit auch die Studienarbeit) keinen Anspruch auf Innovationsdarbietung. Die Begründung der dieses speziellen Entwicklungsbeispiels liegt darin, dass eine To-Do-Listen-Anwendung ein großes Spektrum von Funktionen abbilden kann. Dieses Spektrum reicht von grundlegenden Funktionen (bspw. dem bloßen Anlegen von Aufgaben) bis zu komplexeren Inhalten (bspw. automatischen Push-Notifications über unerledigte oder überfällige Aufgaben). Diese design- und architekturbedingenden Entscheidungen werden im Folgenden definiert und näher beschrieben.

Die Beschreibung der Architektur einer zu entwickelnden Applikation ist eine maßgebende Disziplin im Software-Engineering-Prozess. Dieser Prozess geschieht vor Beginn der Implementierung und ermöglicht, bezogen auf den Umfang dieser Studienarbeit, die Vergleichbarkeit der Applikation hinsichtlich der relevanten Entwicklungsplattformen. Der Umfang sämtlicher Software-Engineering-Prozesse wird grundsätzlich in großen Entwicklerteams praktiziert. Diese gehen der Entwicklung meist komplexer und skalierbarer Anwendungen nach. Im Vergleich dazu ist die hier zu entwickelnde Mobilanwendung lediglich Mittel zum Zweck für die Beantwortung der Forschungsfrage. Das Entwicklungsteam der To-Do-Anwendung besteht aus zwei Personen, welche sich im Rahmen dieser Arbeit autark mit unterschiedlichen Entwicklungsplattformen beschäftigen.

Dies sorgt dafür, dass sich lediglich eine abgespeckte Form des Software-Engineering auf dieses Projekt anwenden lässt. Konkret bedeutet dies, dass keine spezifischen Aussagen über den Software-Prozess bzw. über das Entwicklungsmodell (Wasserfall-Modell, iteratives Modell, etc.) gemacht werden. Dies würde sich hinsichtlich des verhältnismäßig geringen Entwicklungs- und Wartungsaufwands der App kontraproduktiv auf die Zielorientierung auswirken. Umso wichtiger ist die Definition funktionaler sowie nicht-funktionaler Anforderungen, welche daraufhin näher erläutert und spezifiziert werden. Auch die Interaktion zwischen Nutzer und Anwendung muss für eine vergleichende Entwicklung definiert werden. Die zeitlichen und komponentenabhängigen Abläufe innerhalb der App gehören ebenfalls zu den Bestandteilen der Architektur. Letztere beiden Punkte werden im Rah-

men des sog. *Systems Modelling* definiert. Darüber hinaus werden auch optische Aspekte und Verhaltensweisen des **UI** abgegrenzt.

4.1 Anforderungsdefinition

Die Funktionalität der Anwendung wird zunächst über die Anforderungsdefinition näher beschrieben. Diese kann auf allgemeine Funktionsweisen der App (nicht-funktionale Anforderungen), sowie auf spezifische, technische Charakteristika abgegrenzter Bereiche der Anwendung (funktionale Anforderungen). Grundsätzlich gilt es, nicht-funktionale Anforderungen im Laufe der Anforderungsdefinition in meist mehrere funktionale Anforderungen zu überführen **Garidis**, da diese qualifizierbarer sowie quantifizierbarer Natur sind und somit ebenfalls für eine bessere Vergleichbarkeit der zu entstehenden Anwendungen beitragen könnten.

4.1.1 Nicht-Funktionale Anforderungen

Die folgenden nicht-funktionalen Anforderungen beziehen sich auf Teile der Anwendung, welche jedoch abstrakter Natur sind, weswegen sie zunächst zu nicht-funktionalen Anforderungen gezählt werden müssen. Aufgrund der Formulierung werden diese Anforderungen auch als *Nutzeranforderungen* (engl. *User Requirements*) bezeichnet und stehen den spezifischeren, technisch versierteren *Systemanforderungen* (engl. *System Requirements*) gegenüber **Garidis**.

Anlegen, Auflisten, Bearbeiten und Löschen von Aufgaben Die Anwendung ermöglicht es, Aufgaben hinzuzufügen. Die hinzugefügten Aufgaben werden aufgelistet. Bei Bedarf soll der Inhalt der Aufgabe nachträglich abgeändert werden können. Ebenfalls ist es möglich, die Aufgabe aus der Ansicht innerhalb der Anwendung zu entfernen.

Aufgaben bestehen nach Neustart der Anwendung bei Wird die Applikation (gewollt und ungewollt) neu gestartet, bildet sie nach Neustart dieselben Aufgaben und Einstellungen wie zuvor ab.

Priorisierung der Aufgaben möglich Bei Bedarf ist es möglich, einer bestimmten Aufgabe einen gesonderten Stellenwert zuzuweisen.

Benachrichtigungen über nicht-erledigte und überfällige Aufgaben Der Nutzer wird unabhängig vom Status der Applikation oder des Smartphones (d.h. online/offline, geöffnet, im Hintergrund oder geschlossen bzw. gesperrt oder entsperrt) über nicht-erledigte und überfällige Aufgaben benachrichtigt.

Neben dieser Art der nicht-funktionalen Anforderungen koexistieren jene, welche zwar ebenfalls abstrakt und allgemein gehalten sind, jedoch keinen Bedarf resp. keine Möglichkeit zur weiteren Spezifizierung an dieser Stelle des Prozesses aufweisen.

Bereitstellung der Anwendung für mehrere Plattformen Die Anwendung ist nicht nur auf einer Plattform verfügbar, sondern kann auf Geräten unterschiedlicher Betriebssysteme installiert und verwendet werden.

Aussehen und Verhalten sind deckungsgleich Unabhängig davon, welche Plattform genutzt wird, ist die Interaktion zwischen dem Nutzer und der Anwendung annähernd identisch. Davon ausgenommen sind Aspekte, welche auf der entsprechenden Plattform nicht oder nur mit unverhältnismäßigem Aufwand erreicht werden können.

zeiteffizienter Entwicklungsprozess Um auch einen entwicklungstechnischen Vergleich ziehen zu können, soll die Anwendung in einer dem Projekt angemessenen Zeit vollständig entwickelt werden können.

Die Problematik nicht-funktionaler Anforderungen im Bezug auf realistisch zu betrachtende Entwicklungsprojekte kann hier interpretiert werden. Vor allem bei eher unerfahrenen Entwicklern (zu welchen sich das Entwicklerteam dieses Projektes zu zählen erlaubt) sind bestimmte Tendenzen unklar. Dazu gehören bspw. das Bewusstsein über die Realisierbarkeit bestimmter Komponenten sowie die zeitliche Aufwandseinschätzung. Diese Störfaktoren werden im Laufe der Arbeit versucht, entkräftet zu werden und sind in die Evaluation der Forschungsfrage kritisch einzubeziehen.

4.1.2 Funktionale Anforderungen

Nichtsdestotrotz ist die Spezifizierung der in Abs. 4.1.1 eingangs definierten nicht-funktionalen Anforderungen noch ausstehend. Zur Unterstützung der Lesbarkeit werden diese in Reihenfolge der nicht-funktionalen Anforderungen abgehandelt.

Bereitstellung klassischer **CRUD-Operationen** Sog. **CRUD**-Operationen greifen auf das Datenmodell der Anwendung zu. Diese erlauben die Manipulation der Daten auf Basis der gewünschten Operation. Diese Operationen sind unabhängig voneinander zu definieren und sinnvoll in den Verwendungsprozess der App einzubauen. Man spricht hier auch von sog. **CRUD-Endpoints**, welche vereinfacht als statische Funktionen beschrieben werden können.

Listendarstellung Die Aufgaben sollen grundsätzlich in einer sortierten Liste dargestellt werden. Die Liste besteht aus individuellen Elementen, welche jeweils eine Aufgabe darstellen. Jene **CRUD**-Operationen, welche speziell auf eine bestimmte Aufgabe

angewandt werden sollen, finden ihre Aktivierung ebenfalls über ihre entsprechenden Elemente.

Bereitstellung eines Persistent Services Bei Ausführen der zuvor definierten **CRUD**-Operationen werden die Daten nicht nur in den flüchtigen Arbeitsspeicher des Smartphones geschrieben, sondern zugleich auch auf einen der Applikation zugewiesenen Festspeicher. Diese idealisierte Datenbank gleicht dem Datenmodell für die **CRUD**-Operationen und wird somit bei jeder Ausführung dieser aktualisiert bzw. beansprucht.

Definition einer Hierarchie für Aufgaben Eine hierarchische Struktur der Daten soll ermöglichen, bestimmte Aufgaben seitens der Anwendung anders zu behandeln als andere. Durch das Setzen eines sog. *Flags* können die Aufgaben entsprechend der Hierarchiestruktur bestimmte Zustände übergeben bekommen, konkret eine hervorgehobene optische Darstellung innerhalb der **UI** sowie die Präsentation an Anfang der Liste (Eingriff in die Sortierung der Aufgaben)

Streng genommen sind funktionale Anforderungen sehr granular zu definieren **Garidis**. Da es sich hier jedoch um eine wissenschaftliche Arbeit handelt, und nicht um eine Entwicklerdokumentation, wird auf eine detaillierte Beschreibung verzichtet. Viel eher soll die nächste Sektion die genaue, weiterhin plattformunabhängige Umsetzung dieser Anforderungen erläutern, welche als Maßgabe für die spätere Entwicklung der Applikation dienen soll.

Unabhängig von der Plattform wird zunächst ein allgemeines, während der individuellen Entwicklungsphase zu spezifizierendes Grundgerüst der Funktionalität definiert. Bei einer To-Do-Applikation besteht dieses grundsätzlich aus zwei Komponenten. Die idealisierte *Datenbank* ermöglicht persistente Speicherung der angelegten To-Do-Einträge. Diese ist mit verschiedenen **CRUD**-Funktionen direkt an das **UI** angebunden und ermöglicht somit die Manipulation der Einträge.

4.2 Speicherung der Daten

Die Datenbank muss auf eine Weise angelegt werden, dass ihre Daten (d. h. To-Do-Einträge) einem bestimmten Schema folgen. Konkret werden folgende Attribute für die Entität *ToDo* festgelegt, welche in Tabelle 4.1 dargestellt sind.

Attribut	Beschreibung	
id	(alpha-)numerische Zeichenfolge, welche einen Eintrag eindeutig erkennbar macht	String
name	anzuzeigender Text, welcher den eigentlichen Eintrag darstellt und beschreibt	String
done	Status über die Erledigung des entsprechenden Eintrages	Boole
priority	Status über die Priorität des entsprechenden Eintrages	Boole

Tabelle 4.1: Attribute der ToDo-Entität

Unabhängig von der individuellen Architektur der jeweiligen Apps folgt dieses triviale Schema dem Konzept relationaler Datenbanken und könnte somit in einer einfachen Tabelle dargestellt werden.

Um auf die Datenbank zugreifen zu können, muss diese mit entsprechenden Funktionen ausgestattet werden. Neben dem bloßen Erstellen von Einträgen, müssen diese abgegriffen (engl. *fetch*) sowie bearbeitet und gelöscht werden können. Die beiden letztgenannten Funktionen haben bei Ausführung nur Einfluss auf einen durch den Nutzer ausgewählten Eintrag. Somit müssen diese Funktionen das *Objekt* des entsprechenden Eintrages übergeben bekommen. Weiterhin gliedert sich die Bearbeitung von Einträgen in drei Teilfunktionen auf, nämlich dem Ändern des **done**- oder **priority**-Attributes sowie dem Ändern des beschreibenden Textes des Eintrags.

Da die Ausführung sowie Umsetzung dieser Operationen mit der **UI** Hand in Hand geht, wird die Definition dieser vorgezogen.

4.3 Benutzeroberfläche (**UI**)

Die Benutzeroberfläche einer solch einfachen To-Do-Anwendung besteht aus einer Ansicht. Diese Ansicht lässt sich hierarchisch definieren. Die oberste Ebene dieser Hierarchie bildet der hier sog. *App-Container*. Anders als in aufwändigeren Applikationen kann dieser hier mit der Ansicht gleichgesetzt werden, da keine weiteren Ansichten existieren. Dieser Container beherbergt eine Listenansicht, welche für die Darstellung und Interaktion mit den bereits vorhandenen Einträgen zuständig ist. Für das Erstellen der Einträge steht ein separates Text(eingabe)feld zur Verfügung, sowie ein *Button* zur Bestätigung der Eingabe.

Die Listenansicht besteht nun aus mehreren Listeneinträgen (im Folgenden **Zellen** genannt). Eine Zelle ist für die Darstellung und Interaktion für genau einen To-Do-Eintrag zuständig. Um dies zu ermöglichen, besitzt jede Zelle, neben eines Textfeldes zum Anzeigen des To-Do-Textes, weitere Buttons zum Setzen der Priorität und des Status sowie zum Löschen des Eintrages. Während der Button, welcher für das Entfernen des Eintrages verwendet wird (dargestellt durch ein Kreuz, statischer optischer Natur ist (d. h., er ändert

nach einem Tippen sein Aussehen nicht), untermalen die Buttons der To-Do-Zustände die gewählten Einstellungen durch ihr Aussehen. Der Button für Priorisierung, welcher durch ein Sternsymbol dargestellt wird, ist bei aktiver Priorisierung gefüllt. Ist dies nicht der Fall, so ist lediglich der Umriss des Symbols zu erkennen. Gleiches gilt für den Button, welcher anzeigt, ob der Eintrag bereits erledigt ist, dargestellt durch ein Häkchen inmitten eines Kreises.

Die beschriebenen, visuellen Eigenschaften lassen sich nun in einem sog. *Wireframe* zusammenfassen, welches gleichzeitig die optische Grundlage der Entwicklung darstellen wird. Dies ist vor allem aufgrund der unterschiedlichen Entwicklungsplattformen von Relevanz, da das Einhalten bestimmter Standards der entsprechenden Plattformen dafür sorgen könnte, dass der letztendliche Vergleich beider Applikationen hohe Differenzen aufweist. Das Wireframe ist in Abb. 4.1 abgebildet.

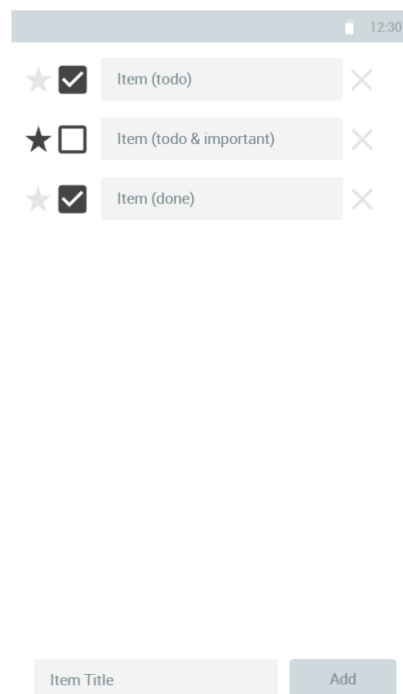


Abbildung 4.1: Wireframe der App

Eine Besonderheit in der Darstellung lässt sich innerhalb der priorisierten Elemente finden. Um diese weiter hervorzuheben, werden diese an den Anfang der Liste gesetzt. Es entstehen somit zwei Teillisten, welche sich jedoch in derselben Listenansicht befinden. Wird nun ein zuvor nicht-priorisierter Eintrag priorisiert, so wechselt dieser seine Position ans Ende der Liste mit den bereits priorisierten Einträgen (bzw. wird oberhalb der ersten nicht-priorisierten Elements platziert). Alle Einträge zwischen der alten und der neuen Position des gerade betrachteten Eintrags werden um eine Listenposition nach unten verschoben. Bei Entfernen der Priorisierung wird das entsprechende Element nun nicht an seine ursprüngliche Position vor der Priorisierung, sondern an den Anfang der

nicht-priorisierten Liste verschoben. Entfernt man also bspw. die Priorisierung des letzten Elements in der priorisierten Liste, ändert sich die Reihenfolge nicht. Dieses Verhalten kann vereinfacht in der untenstehenden Abbildung dargestellt werden.

Grundsätzlich werden alle Einträge in der Reihenfolge dargestellt, wie sie angelegt wurden, mit den eben beschriebenen Ausnahmen.

Um ebenfalls für farbliche Konsistenz zu sorgen, werden die beschriebenen Elemente auf Basis der folgenden Tabelle in ihrem Erscheinungsbild konfiguriert:

Bezeichnung	Hex-Code	Darstellung
Allgemeines		
Hintergrund	#F2F2F2	
Schriftfarbe	#8C8C8C	
Bedienelemente		
Hintergrund für inaktive Bedienelemente	#CECECE	
Hintergrund der Checkbox (angewählt)	#1A66FF	
Schriftfarbe der Checkbox (angewählt)	#FFFFFF	

Tabelle 4.2: Farbtabelle

4.4 Nutzungszyklus

Der Nutzer öffnet die Anwendung. Beim Laden der Ansicht wird eine Datenbank-Funktion ausgeführt, um bereits existierende Einträge zu laden und ihre entsprechenden Beschreibungen samt weiteren Attributen in die einzelnen Zellen der Listenansicht zu laden.

Der Nutzer möchte einen neuen To-Do-Eintrag erstellen. Dafür wird ein beschreibender Text in das Texteingabefeld am unteren Rand der Ansicht geschrieben. Um einen neuen Eintrag zu erstellen, wird ein beschreibender Text in das Texteingabefeld am unteren Rand der Ansicht geschrieben. Mit der Bestätigung über den **+**-Button wird ein neuer Eintrag in die Datenbank aufgefördert, wobei das **text**-Attribut mit dem zuvor gewählten Text aus dem Eingabefeld gefüllt wird. Die Zeichenfolge der **id** wird automatisch und zufallsbasiert generiert, **done** sowie **priority** standardmäßig auf **false** gesetzt. Der neue Eintrag wird unter den bereits vorhandenen Einträgen angefügt. Stern- sowie Häkchen-Symbol werden lediglich über ihre Kontur kenntlich gemacht.

Der Nutzer möchte den Text eines Eintrages ändern. Durch direktes Tippen auf den dargestellten Text in einer Zelle wird dies ermöglicht. Es erscheint ein Cursor, welcher ebenfalls Tastatur- und Berührungsten innerhalb dieses Feldes ermöglicht. Nach bestätigter Änderung über das Verlassen des Textfeldes (d. h. dem Tippen auf eine andere Stelle innerhalb der Ansicht) wird erneut eine Datenbank-Funktion ausgeführt. Diese bekommt

das `ToDo`-Objekt übergeben und ersetzt den bestehenden Inhalt des `text`-Attributes mit dem nun geänderten.

Der Nutzer möchte den zuvor erstellten Eintrag priorisieren. Tippt dieser auf das Stern-Symbol, wird dieses mit der zuvor definierten Farbe gefüllt. Gleichzeitig bewegt sich das priorisierte Element an das Ende der Teilliste mit priorisierten Einträgen. Eine Datenbank-Funktion wird aufgerufen, welche den `priority`-Wert des übergebenen Objekts von `false` auf `true` setzt.

Der Nutzer möchte den zuvor priorisierten Eintrag als abgeschlossen markieren. Tippt dieser auf das Häkchen-Symbol wird dieses mit der zuvor definierten Farbe gefüllt. Eine Datenbank-Funktion wird aufgerufen, welche den `done`-Wert des übergebenen Objekts von `false` auf `true` setzt.

Der Nutzer möchte den zuvor als abgeschlossenen Eintrag aus der Liste der priorisierten Einträge entfernen. Tippt dieser auf das Stern-Symbol, wird dessen Füllung entfernt. Gleichzeitig bewegt sich das Element an den Anfang der Teilliste mit nicht-priorisierten Einträgen. Die zuvor ausgeführte Datenbank-Funktion wird erneut aufgerufen, welche den `priority`-Wert des übergebenen Objekts von `true` nun wieder auf `false` setzt.

Der Nutzer möchte den Eintrag abschließend entfernen. Tippt dieser auf das Kreuz-Symbol, wird der Eintrag aus der Liste entfernt. Die Elemente unterhalb des gelöschten Eintrags verschieben sich um jeweils eine Position nach oben. Eine Datenbank-Funktion wird aufgerufen, welche das übergebene Objekt des Eintrages aus der Datenbank entfernt.

5 Implementierung

5.1 Implementierung der nativen App

Die Entwicklung der nativen Mobilanwendung wird im Folgenden exemplarisch anhand einer *iOS*-Applikation dargestellt. Diese werden ausschließlich auf Mobilgeräten vom U.S.-amerikanischen Hersteller *Apple, Inc.* ausgeführt, funktionieren also auf diversen *iPhone*-, *iPod touch*- und in gewisser Hinsicht auch auf *iPad*-Geräten. Letztere sind wegen der im September 2019 in Kraft getretenen Einführung des sog. *iPadOS* nicht mehr auf reine *iOS*-Anwendungen ausgelegt, sondern nutzen diese lediglich in Form einer Vergrößerung der Kleingeräteanwendung.

Dieses Kapitel stellt im Folgenden den gesamten Entwicklungsprozess der To-Do-Anwendung dar. Der in Abs. 2.3.1 angesprochene Standard heutiger *iOS*-Entwicklung wird auch in der Praxis dieser Untersuchung angewandt. Somit fällt die Wahl der Programmiersprache auf Apple Swift 5 unter der **IDE** Xcode 11. Im Hinblick auf die To-Do-Anwendung, welche einen persistenten Speicher benötigt, kann hier bereits die Integration der internen Bibliothek *Core Data* hinzugefügt werden.

Grundsätzlich ist es sinnvoll, mit der Gestaltung der visuellen Ansicht zu beginnen, da die erstellten Komponenten im Nachhinein explizit im Quellcode referenziert werden können, um ihnen Funktionalität zu verleihen.

5.1.1 Gestaltung des User Interface

5.1.1.1 Einbinden von **UI**-Komponenten

Initial beinhaltet das `Main.storyboard` eine Ansicht, welche bereits mit dem automatisch generierten `ViewController` verknüpft ist. Diese muss nun mit den in Kapitel 4 beschriebenen Komponenten gefüllt werden. Die Listenansicht wird über einen `UITableView` realisiert. Dieser füllt einen großen Teil der Ansicht aus. Der untere Rand der Ansicht wird mit einem `UITextField` sowie mit einem `UIButton` versehen, welcher als `+`-Symbol konfiguriert wird. Somit ist die hierarchisch höchste Stufe der Ansicht fertiggestellt.

Bislang existiert noch keine Konfiguration der unteren Hierarchiestufe, welche aus den einzelnen Zellen des `UITableView` besteht. Da diese dem gleichen Aufbau folgen sollen, ist die Definition *einer* `UITableViewCell` genügend, sodass diese dann für die gesamte Liste verwendet und repliziert werden kann. Diese Zelle beinhaltet erneut ein `UITextField`, sowie drei `UIButton`s, welche jeweils mit einem Kreuz-, Häkchen-, und Stern-Symbol konfiguriert werden.

Die Farbgebung der einzelnen Buttons wird nach Maßgabe der Farb-Definition aus [Sektion 4.3](#) als *Tint* der Buttons festgelegt.

Das finale **UI** der To-Do-Anwendung, wie es im Interface Builder dargestellt wird, kann dem folgenden Screenshot entnommen werden.

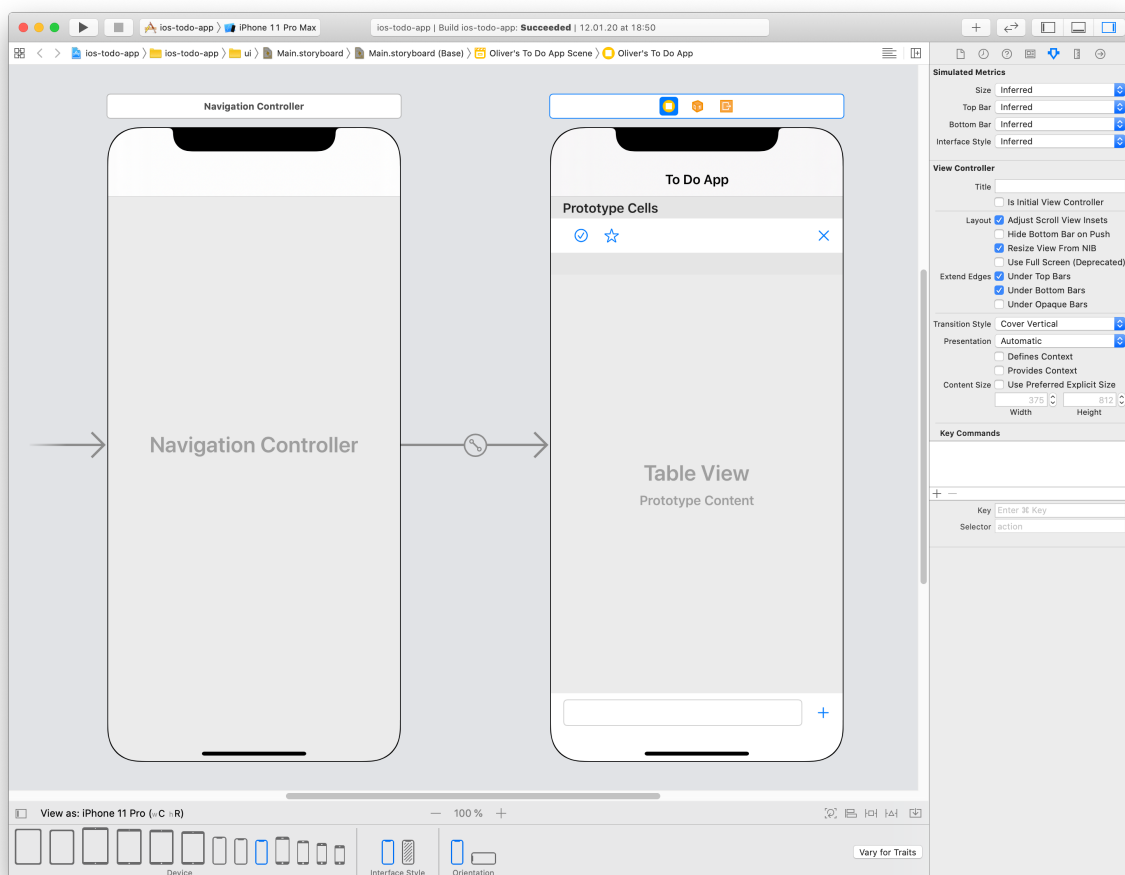


Abbildung 5.1: **UI** im Interface Builder von Xcode

Der auf der linken Seite sichtbare *Navigation Controller* kann an dieser Stelle vernachlässigt werden, da es nur das Hinzufügen der Kopfleiste mit der Aufschrift *To Do App* zur Folge hat.

5.1.1.2 Relative Positionierung der Komponenten

Apple bietet verschiedene Mobilgeräte in verschiedenen Größen an. Der Interface Builder von Xcode erlaubt die interaktive Positionierung der **UI**-Komponenten jedoch nur unter Betrachtung einer spezifischen Gerätegröße. Wird diese in der Ansicht gewechselt (s. Abb. 5.1 unten), fällt auf, dass die Positionierung der Komponenten absolut gesetzt wird und diese somit bei kleineren Geräten über den gedachten Bildschirmrand hinausgehen bzw. bei größeren Geräten nicht die gesamte Bildschirmbreite ausnutzen.

In vielen Fällen schaffen sog. *Auto-resizing Constraints* Abhilfe gegen dieses Problem. In diesen kann das Verhalten einzelner Komponenten bei bestimmten Bildschirmgrößen bestimmt werden. Dieses Verhalten beschreibt die Änderung der Position sowie der Skalierung der Komponente. So soll der `UITableView` den gesamten horizontalen Bereich des Bildschirms einnehmen. Gleiches gilt für den Bereich des Textfeldes und des `+`-Buttons. Da diese jedoch in Abhängigkeit voneinander stehen (d.h. das Textfeld soll immer einen bestimmten Abstand zum Button wahren; der Button soll immer einen bestimmten Abstand zum rechten Bildschirmrand wahren), müssen diese durch eine Unteransicht logisch verbunden werden. In dieser werden dann pixel- oder prozentgenaue Abstände und Änderungen definiert. Diese Unteransicht kann dann wieder über *Auto-resizing Constraints* die gesamte Breite unter der Listenansicht einnehmen. So kann die App nahtlos auf verschiedenen großen Geräten genutzt werden, ohne dass die Positionierung aller Komponenten für alle möglichen Bildschirmgrößen manuell gesetzt werden muss.

5.1.1.3 Dynamische Änderung der Komponentenposition

Die zuvor definierte Unteransicht, welche für das Erstellen neuer To-Do-Einträge verantwortlich ist, befindet sich zunächst noch jederzeit am unteren Bildschirmrand. Tippt der Nutzer nun in das Textfeld, um einen neuen Eintrag zu erstellen, so werden Textfeld und `+`-Button von der von unten einfliegenden Tastatur überdeckt. Somit kann der Nutzer seinen Eintrag ins Textfeld nicht sehen und diesen auch nicht über Tippen auf den Button erstellen. Diese Unteransicht muss also bei Aufruf der Tastatur zusammen *mit der Tastatur* verschoben werden.

Auf technischer Ebene muss eine Verschiebung der Unteransicht initiiert werden, sobald die Tastatur zu erscheinen beginnt. Gleichmaßen muss diese Verschiebung rückgängig gemacht werden, sobald die Tastatur wieder verschwindet. Es handelt sich hierbei um *Event Listening* bei einem allgemeinen **UI**-Komponenten, nämlich der Tastatur. Solches Event Listening ist in Swift (noch) nicht abgesehen, weswegen auf Objective-C zurückgegriffen werden muss. Die Listener, auch *Observer*, die auf das Verhalten der Tastatur achten, müssen bei Start der App initialisiert und bei Schließen der App deinitialisiert

werden. Sobald einer der Observer über die Änderung des Zustands der Tastatur benachrichtigt, wird die Positionsänderung aufgerufen.

Die absolute Position einer Komponente innerhalb der Ansicht kann anhand eines Koordinatensystems visualisiert werden, welches seinen Ursprung (0; 0) in der oberen linken Ecke hat. Nach rechts hin wächst die x -Koordinate, nach unten hin wächst die y -Koordinate. Somit muss bei Einfliegen der Tastatur die y -Position der Unteransicht um gerade die Höhe der zugrunde liegenden Tastatur reduziert werden, sodass die Unteransicht sich direkt über der Tastatur befindet. Dazu wird zunächst die je nach Sprache variable Tastaturhöhe ermittelt und die y -Position um genau diesen Wert reduziert. Um diese Aktion rückgängig zu machen, wird eine Erhöhung der Position um diesen Wert eingeleitet, sobald der entsprechende Observer die Funktion über das Verschwinden der Tastatur benachrichtigt. Die Unteransicht wird nun nicht mehr von der Tastatur verdeckt, sondern wirkt optisch wie ein Teil von ihr, sodass diese nun genutzt werden kann, um To-Do-Einträge zu erstellen.

An dieser Stelle scheint es intuitiv, die Ansicht nun zu testen. Da bislang jedoch noch keinerlei Funktionalität implementiert ist, ist es nun wichtig, Zellen erstellen zu können. Für Testzwecke könnte eine provisorische Funktionsweise entwickelt werden, welche die Zelleninformationen zunächst im Arbeitsspeicher der Anwendung speichert. Da dies jedoch im Nachhinein zu großen Abänderungen des Quellcode-Aufbaus führen kann, werden zunächst der Persistent Service und die **CRUD**-Funktionen definiert, welche dann direkt über den ViewController an das **UI** angebunden werden.

5.1.2 Entwicklung des Persistent Service

Wie in Abs. 2.3.1 erwähnt, geschieht die Konfiguration des Datenmodells des Persistent Service über die Konfigurationsdatei `to_do.xcdatamodeld`. Dort kann eine neue Entität `ToDo` angelegt werden, welche die in Sektion 4.2 beschriebenen Attribute (`id`, `text`, `done` und `priority`) in Form ihrer spezifizierten Datentypen beinhaltet. Da keine Beziehungen zu anderen Entitäten von Bedarf sind, kann diese Entität nun als sog. `NSManagedObject` exportiert werden. Dieses ermöglicht die Nutzung der Entität als Objekt im Quellcode und stellt notwendige Funktionen für die spätere **CRUD**-Funktionalität bereit.

Die notwendigen Komponenten des Persistent Service liegen im `AppDelegate` bereit. In diesem wird davon ausgegangen, dass mehrere Instanzen eines solchen Services genutzt werden. Da es sich im Fall der To-Do-Anwendung jedoch um einen global gleich genutzten Speicher handelt, spricht nichts dagegen, die Komponenten in eine statische Klasse `Storage` auszulagern. In dieser Klasse werden nun auch Funktionen für die verschiedenen

CRUD-Operationen angelegt. Auf Basis der bereits vorhandenen Funktion `saveContext()` wird die Nicht-Flüchtigkeit dieser **CRUD**-Operationen gewährleistet.

5.1.2.1 `createToDo(...)`: Anlegen von To-Do-Elementen

Der Aufbau der Funktion, welche für das Anlegen der To-Do-Elemente innerhalb des Persistent Service zuständig ist, wird hinsichtlich des zugrunde liegenden Quellcode-Ausschnitts beschrieben.

```

1  static func createToDo(toDoText: String, completion: (ToDo) -> Void) {
2      guard let toDoEntity = NSEntityDescription.entity(forEntityName: "ToDo", in: context)
        ↪ else { return }
3
4      let newToDo = ToDo.init(entity: toDoEntity, insertInto: context)
5
6      newToDo.id = UUID()
7      newToDo.text = toDoText
8      newToDo.done = false
9      newToDo.priority = false
10
11     saveContext()
12     print("[STORAGE] Created To-Do '\(toDoText)' with ID \(String(describing: newToDo.id))")
13     completion(newToDo)
14 }

```

Quellcode-Ausschnitt 3: Funktion zur Erstellung von To-Do-Elementen (Swift)

Es werden die Entitätsinformationen aus der Konfiguration der Datei `to_do.xcdatamodeld` entnommen (Z. 2), sodass ein neues Objekt nach dieser Entität erstellt und dem Speicher zugeordnet werden kann (Z. 3). Daraufhin werden die initialen Werte der einzelnen Attribute gesetzt (Zz. 6–9), wobei der beschreibende Text hier direkt aus dem Funktionsparameter entnommen wird. Zuletzt wird der Kontext (vereinfacht also der Speicher) aktualisiert und das neue Objekt wird in Form einer sog. *Completion* zurückgegeben (Z. 1, Z. 13). Dieses kann in einer sog. *Callback*-Funktion verwendet werden, welche nach erfolgreichem Durchlaufen der Ausgangsfunktion ausgeführt wird.

5.1.2.2 `loadToDos(...)`: Laden von To-Do-Elementen aus dem Speicher

Der Aufbau der Funktion, welche für das Laden der To-Do-Elemente aus dem Persistent Service zuständig ist, wird hinsichtlich des zugrunde liegenden Quellcode-Ausschnitts beschrieben.

```
1 static func loadTodos(completion: ([ToDo]) -> Void) {
2     let request: NSFetchedRequest<ToDo> = ToDo.fetchRequest()
3
4     do {
5         let todosFromDisk = try context.fetch(request)
6         completion(todosFromDisk)
7
8         print("[STORAGE] Loaded to-dos:")
9         for todo in todosFromDisk {
10             print("\(String(describing: todo.id)): '\(String(describing: todo.text))'")
11         }
12     } catch {
13         fatalError("[STORAGE/Error]: Could not load to-do entities from disk: \(error)")
14     }
15 }
```

Quellcode-Ausschnitt 4: Funktion zum Laden von To-Do-Elementen (Swift)

Zunächst wird der Datenabgriff vorbereitet, welcher auf die zuvor exportierte Klasse der ToDo-Entität zugreift (Z. 2). Auf Basis dieser Anfrage (Z. 5) wird ein Array vom Datentyp ToDo in gleicher Completion-Form wie zuvor zurückgegeben (Z. 6). Da dieser Vorgang *Exceptions* schmeißen kann, muss dieser Vorgang entsprechend kontrolliert (d. h. mit *Error Handling* versehen) ablaufen.

5.1.2.3 Bearbeiten von Attributswerten von To-Do-Elementen

Da das UI mehrere Bedienelemente für das Bearbeiten eines Eintrags aufweist (d. h. das Ändern des Textes, der Priorität sowie des Status über Abschluss der Aufgabe), müssen entsprechend viele Funktionen für das Bearbeiten dieser Eigenschaften angelegt werden. In jeder der Funktionen wird das zu bearbeitende To-Do-Objekt mitgegeben und das zu ändernde Attribut wird mit dem neuen, ebenfalls mitgegebenen, Wert überschrieben. Das abschließende Ausführen von `saveContext()` sorgt für die Konsistenz des Objekts.

5.1.2.4 `deleteToDo(...)`: Löschen von To-Do-Elementen

Das mitgegebene To-Do-Objekt wird über die Kontext-Löschfunktion entfernt und der Stand des Kontexts mit `saveContext()` gespeichert.

5.1.3 Entwicklung der UI-Funktionalität

In Abs. 5.1.1 sind die verschiedenen UIButton-Komponenten zum UI hinzugefügt worden, besitzen bislang jedoch keinerlei Funktionalität. Durch die in 5.1.2 implementierten CRUD-Funktionen ist es nun möglich, diese mit den UIButtons zu verknüpfen, um diese durch entsprechenden Knopfdruck auszulösen. Eine Ausnahme bildet die Funktion `loadToDos(...)` (vgl. Abs. 5.1.2.2), welche bei Initialisierung der App aufgerufen wird. Zusätzlich soll eine Folge von Abschlussoperationen durchgeführt werden, welche die Darstellung der Anwendung analog zum Zustand des Speichers korrigiert.

Im Quellcode wird die Verknüpfung über die Kombination aus sog. IBOutlet-Variablen und IBAction-Funktionen realisiert. Um die statischen Zustände der Komponenten erkennen, editieren und nutzen zu können, werden diese in Form von IBOutlet-Variablen den zuständigen Klassen hinzugefügt. Dies wird sich bei der Initialisierung der Anwendung zunutze gemacht, da so die Zustände der Zellenbuttons anhand der bereits vorhandenen und zu Start der Anwendung geladenen Einträge korrekt gesetzt werden können. Sollen Aktionen, wie vorhin beschrieben, durch die Komponenten ausgelöst werden (bspw., wenn ein Button angetippt wird), nutzt man IBAction-Funktionen, welche immer zu einer bestimmten Komponente gehören, und ihre Anweisungen ausführen, sobald der definierte Aktionszustand erreicht ist.

5.1.3.1 Verknüpfung der CRUD-Funktionen

Die Funktion `createToDo(...)` soll dem globalen +-Button zugewiesen werden. Das Bearbeiten (d.h. als erledigt markieren resp. priorisieren) und Löschen der bereits vorhandenen Einträge wird über die Verknüpfung mit den Buttons realisiert, welche pro Zelle jeweils einmal auftauchen. Das Bearbeiten der Eintragsbeschreibung nimmt an dieser Stelle eine Sonderrolle ein. Der Text nicht als statisches UILabel definiert ist, welcher sonst nur indirekt editierbar wäre. Es handelt sich, genau wie beim Textfeld für die Erstellung von To-Do-Einträgen, um ein UITextField, welches vom Nutzer direkt angetippt werden kann, um Änderungen vorzunehmen. Dieses kann ebenfalls sog. *Actions* ausführen, wenn ein bestimmter Zustand des Komponenten erreicht ist. Im Falle der individuellen Textfelder für die Eintragsbeschreibung wird die entsprechende CRUD-Funktion für die Änderung und Speicherung des Textes ausgeführt, sobald die aktive Nutzung des jeweiligen Textfeldes abgeschlossen ist, sich anschaulich also kein Cursor mehr in diesem Textfeld befindet.

Während `loadToDos(...)` und `createToDo(...)` auf globaler Anwendungsebene keine variablen Auswirkungen haben, stellen die restlichen CRUD-Funktionen eine interkommunikative Hürde dar. Die Ausführung der jeweiligen Funktion muss der gewünschten Zelle zugrunde liegen. Als Frage ließe sich formulieren: Wie weiß die in der Klasse

`TableViewCell` definierte **CRUD**-Funktion, auf welche der initialisierten Zellen im `ViewController` sich die Ausführung bezieht?

Konkret begründet sich die Problematik in der Tatsache, dass die Funktionen als Zellenfunktionen definiert sind, aber nur der übergeordnete `ViewController` Informationen über die Position der individuellen Zellen hält. Diese Disziplin lässt sich mit der Einführung sog. *Delegates* lösen. Bei diesen handelt es sich um Protokolle, welche Funktionen für eine Klasse deklarieren, welche stellvertretend von einer anderen Klasse definiert und ausgeführt werden. Für den Fall der zugrunde liegenden To-Do-App geschieht folgendes: Das Protokoll `ToDoCellDelegate` wird definiert und in der Klasse `ToDoCell` initialisiert. Das Protokoll deklariert lediglich spezifische Funktionsnamen, welche in der Klasse `ToDoCell` bei auslösen der zuvor beschriebenen `IBAction`-Funktionen ausgeführt werden. Gleichzeitig erbt die Stellvertreter-Klasse `ViewController` von `ToDoCellDelegate` und definiert die Funktionsabläufe. Da die Funktion per Definition immer die entsprechende Zelle als Parameter mitgegeben bekommt, ist nun bekannt, um welche Zelle es sich bei Auslösen der Zellenbuttons handelt, sodass die Operationen korrekt ausgeführt werden können.

5.1.3.2 Ergänzende **UI**-Operationen

Nun gilt es, das anderweitige Verhalten der App bei Auslösen der entsprechenden `IBAction`- bzw. der delegierten Funktionen zu definieren.

Bei Öffnen der App werden die Zellen mit der durch die Funktion `loadTodos(...)` zurückgegebenen To-Do-Einträge in Form eines Arrays gefüllt. Dabei werden zunächst alle priorisierten Einträge aufgeführt, bevor die übrigen Einträge folgen. Bei der Editierung der Textbeschreibung wird die alte Beschreibung des To-Do-Eintrages trivialerweise durch die neue ersetzt. Bei Erledigung der App wird der Zustand des Häkchens als Grundlage genommen und entsprechend gesetzt. Das Löschen eines Eintrages führt zusätzlich zur Entfernung aus dem Speicher ebenfalls zu einer animierten Verschiebung aller darunterliegenden Zellen um eine Position nach oben, sobald die gelöschte Zelle optisch entfällt. Die Priorisierung der Einträge erfordert ebenfalls eine Verschiebung von Einträgen. Sobald ein zuvor nicht-priorisierter Eintrag priorisiert wird, wird er an das Ende aller zuvor priorisierten (und sich somit bereits oben befindenden) Einträge geschoben. Da davon ausgegangen werden kann, dass zu Beginn der App-Nutzung die Einträge bereits korrekt sortiert wurden, wird das erste Element der Liste gesucht, welches nicht priorisiert ist. An diese Stelle gelangt nun der neu zu priorisierende Eintrag. Darüber hinaus werden alle Einträge, welche zwischen der neuen und der alten Position des nun priorisierten Eintrags liegen, um jeweils eine Zellenposition nach unten verschoben. Analog wird bei Depriorisierung verfahren: Das Element wird ans Ende der priorisierten Teilliste (resp. an

den Anfang der nicht-priorisierten Teilliste) geschoben. Dies bedeutet, dass die sequenzielle Priorisierung und Depriorisierung des gleichen Eintrages im zweiten Schritt keinen Positionswechsel mit sich zieht. Dies ist an der Stelle gewollt. Die Umsetzung, ein depriorisiertes Element ganz nach unten zu schieben, könnte auf gleiche Art und Weise realisiert werden.

5.1.4 Umsetzung von Benachrichtigungen über unerledigte To-Do-Einträge

Um zu verstehen, wie die Umsetzung von Benachrichtigungen (engl. *Notifications*) in einer iOS-App vonstattengeht, ist es notwendig, deren Nutzungszyklus nachzuvollziehen. Nach der Installation einer Anwendung, welche Benachrichtigungen unterstützt, wird der Nutzer vom System aufgefordert, zu entscheiden, ob Benachrichtigungen der App erwünscht sind. Erst nach expliziter Bejahung dieser Frage werden diese angezeigt. Somit muss zunächst die Systemanbindung der Benachrichtigungen gewährleistet (und somit implementiert) werden, bevor diese definiert werden können.

Das Verhalten der App gegenüber dem Betriebssystem iOS nach außen wird über den `AppDelegate` gesteuert (vgl. Abs. 2.3.1.1). Die Umsetzung von Benachrichtigungen geschieht über die zusätzliche Funktionalität der Bibliothek `UserNotifications`. Es handelt sich in diesem Fall um lokale Benachrichtigungen, da die Anzahl der unerledigten To-Do-Einträge direkt über die sich im Speicher befindlichen Informationen erfolgen kann.

5.1.4.1 Initialisierung und Genehmigung von Benachrichtigungen

Die Bibliothek `UserNotifications` wird im `AppDelegate` importiert. Das sog. *Notification Center* kann nun bei Start der Anwendung initialisiert werden. Es werden Standardeinstellungen für die Benachrichtigungen definiert, welche greifen, sobald der Nutzer diese genehmigt hat. Das Erscheinungsbild, der dazugehörige Ton und weitere Aspekte können dann im Nachhinein von Nutzer in den Systemeinstellungen angepasst werden. Bei diesen Standardeinstellungen handelt es sich zunächst um das Erscheinungsbild der Benachrichtigung, wenn die App im Vordergrund des Gerätes läuft, somit sichtbar ist. Als gewünschte Standardeinstellungen werden ein *Alert* (d.h. ein darüberliegendes Fenster) sowie ein Warnton gewählt. Im Sinne der zuvor definierten Architektur ist ebenfalls gewünscht, dass der Nutzer außerhalb der Anwendung Benachrichtigungen erhalten kann (bspw. im Hauptmenü, innerhalb anderer Apps, im Stand-By-Modus, im Sperrbildschirm, etc.). Diese werden ebenfalls im `AppDelegate` definiert. In diesem Fall sollen am oberen

Rand des Bildschirms sog. *Badges* erscheinen, welche kenntlich machen, um welche App es sich handelt und welche Benachrichtigung diese gerade liefert.

5.1.4.2 Definition und Auslösung von Benachrichtigungen

Die Definition und Auslösung von Benachrichtigungen geschieht nun wieder über den `ViewController`. Somit importiert dieser ebenfalls die Bibliothek `UserNotifications`. Da, wie im vorigen Abschnitt beschrieben, Benachrichtigungen nicht erzwungen werden können, müssen sog. *Notification Requests* gestellt werden. Diese überprüfen, ob Benachrichtigungen vom Nutzer gewünscht sind und lösen diese bei positiver Rückmeldung aus. Solche Requests benötigen zweierlei Parameter: Der *Content* gibt den Inhalt der Benachrichtigung an. Dieser kann aus einer Varietät von Komponenten bestehen; im Falle dieser Anwendung reicht eine Textzeile, welche angibt, wie viele To-Do-Elemente noch unerledigt sind. Der *Trigger* gibt an, unter welchen Umständen diese Benachrichtigung ausgeführt wird. Hier wird exemplarisch auf sich periodisch wiederholende Benachrichtigungen berufen, welche täglich um eine bestimmte Uhrzeit erscheinen, für den Fall, dass unerledigte Aufgaben existieren. Exemplarisch werden hier 09:00 Uhr, 12:00 Uhr und 15:00 Uhr gewählt. Im Sinne des *Triggers* zählt hier nur die Uhrzeit; die Tatsache, ob es unerledigte Aufgaben gibt, muss separat geprüft werden.

Zunächst wird der *Content* definiert, welcher einem String gleicht, der die Anzahl unerledigter Aufgaben aufzeigt. Da es sich um mehrere gleichartige Trigger handelt, können diese in einem Array gelagert und iterativ abgehandelt werden. Existieren nun unerledigte To-Do-Einträge, so werden die Trigger definiert. Diese Komponenten können dann genutzt werden, um Requests an das Notification Center zu stellen, die entsprechenden Benachrichtigungen anzuzeigen. Über das Antippen der Benachrichtigung kann dann in die To-Do-App gewechselt werden.

Abb. 5.2 zeigt abschließend einen Screenshot der iOS-App.

Abbildung 5.2: Screenshot der iOS-App mit geöffneter Tastatur

Da der Veröffentlichungsprozess der App im Rahmen dieser Studienarbeit nicht reell durchgeführt werden kann, beruft sie sich in diesem Fall auf die Erläuterungen aus Abs. 2.3.1.2. Es ist davon auszugehen, dass es bei diesem Anwendungsbeispiel keine Abweichungen ggü. der üblichen Prozedur gibt.

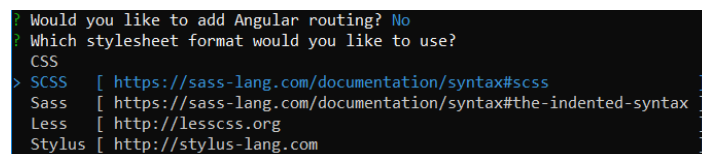
5.2 Implementierung der PWA

5.2.1 Einrichten der Projektumgebung

Für die Entwicklung der PWA wird die Entwicklungsumgebung *WebStorm* von JetBrains gewählt, da sie die gehäuften Routineaufgaben selbstständig bzw. mit geringem Aufwand ausführt.

Wie für die Entwicklung der meisten modernen Webanwendungen ist die Installation der *Node.js*-Laufzeitumgebung notwendig. Mit dem integrierten Paketmanager *npm* lassen sich Bibliotheken leicht zum Projekt hinzufügen. Um die *Angular*-Anwendung automatisiert zu erstellen, ist zuerst die Installation des *Angular Command Line Interface CLI* erforderlich. Mit dem Konsolenbefehl `npm install -g @angular/cli` wird *npm* aufgefordert, die neueste Version der *Angular CLI* global auf dem System zu installieren.

Mit dem Befehl `ng new todoapp` wird die *Angular CLI* (in der Konsole als `ng` abgekürzt) dazu gebracht, ein *Angular*-Projekt inklusive nötiger Dateistrukturen zu erstellen.



```
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use?
  CSS
> SCSS [ https://sass-lang.com/documentation/syntax#scss ]
  Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less [ http://lesscss.org ]
  Stylus [ http://stylus-lang.com ]
```

Abbildung 5.3: Stylesheetformate beim Erstellen der Angularanwendung

Die *CLI* bietet dem Nutzer einige Optionen bei der Erstellung an, die jedoch in diesem Projekt nicht zwangsläufig benötigt werden. So kann bspw. ein Routing oder Unit-Testing eingerichtet werden. Außerdem unterstützt die *CLI* verschiedene Stylesheet-Formate (siehe Abb. 5.3). Das ist für Entwickler sehr praktisch, wenn sie einen dieser *CSS*-Dialekte beherrschen. Die Arbeit mit Variablen in Stylesheets bevorzugt wird, werden in diesem Projekt *SCSS*-Dateien verwendet

5.2.2 Aufbau der Anwendung mit Angular

Im Folgenden werden die einzelnen Komponenten der *Angular*-Anwendung erläutert.

5.2.2.1 Bereitstellen der Klasse *TodoItem* zur Datenspeicherung

Die Komponenten tauschen untereinander Daten aus. Damit diese einer einheitlichen Struktur folgen, wird eine Klasse *TodoItem* erstellt, die einen To-Do-Eintrag repräsentiert.

Objekte dieser Klasse können nun zwischen Komponenten ausgetauscht und modifiziert werden.

```
1 export class TodoItem {
2     description: string;
3     id: string;
4     important: boolean;
5     done: boolean;
6     ...
7 }
```

Quellcode-Ausschnitt 5: `TodoItem`-Klasse zur Datenspeicherung (TypeScript)

Wie im Quellcode-Ausschnitt 5 zu sehen, hat ein To-Do-Element eine Aufgabenbeschreibung (Z. 2), eine eindeutige ID (Z. 3) sowie zwei boole'sche Werte, die speichern, ob das Element als priorisiert oder abgeschlossen markiert worden ist (Zz. 4–5). Die ID hilft später, ein bestimmtes To-Do-Element zu modifizieren.

5.2.2.2 Implementierung des `todoService` für die Datenverwaltung

Alle To-Do-Einträge sollen persistent auf dem Gerät gespeichert werden. Dafür wird ein Angular-Service erstellt; der `todoService`. Er ist für die **CRUD**-Operationen zuständig.

```
1 export class TodoService {
2     items: TodoItem[];
3
4     constructor() {
5         this.items = this.loadStorage() || [];
6     }
7
8     saveInStorage(object) {
9         localStorage.setItem("data", JSON.stringify(object));
10    }
11
12    loadStorage() {
13        return JSON.parse(localStorage.getItem("data"));
14    }
15
16    addTodoItem(description: string) {
17        this.items.push(new TodoItem(description));
18        this.saveInStorage(this.items);
19    }
20 }
```

Quellcode-Ausschnitt 6: Klasse `TodoService` (TypeScript)

In Quellcode-Ausschnitt 6 sind die Kernfunktionen des Services abgebildet. Der Service speichert ein Array von `TodoItem`-Objekten. Wenn der Nutzer ein Element hinzufügt, erstellt der Service ein neues Datenobjekt und speichert dieses im Array und dem Browserspeicher (Zz. 16-18). Die Funktionsweise der übrigen **CRUD**-Operationen ist analog dazu.

Mit `localStorage` (Zz. 8-14) kann auf den *Key-Value-Store* des Browsers zugegriffen werden. Beim Speichern werden die To-Do-Elemente als **JSON**-Objekt im Store abgelegt und analog dazu geladen. Der Key-Value-Store verliert seine Daten beim Schließen der Seite nicht und besitzt kein Ablaufdatum, wie bspw. ein Cookie **LocalStorage**.

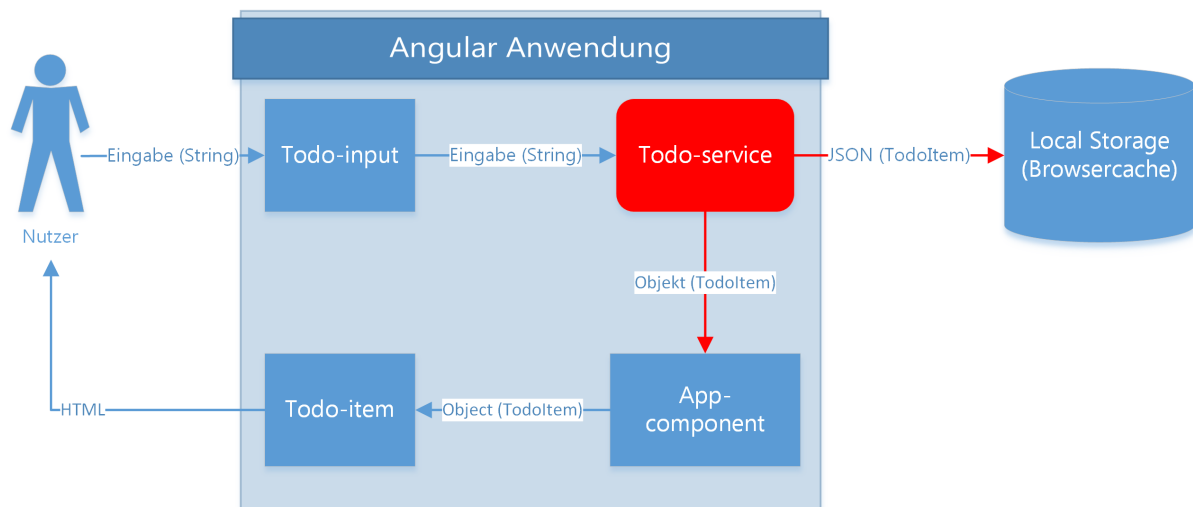


Abbildung 5.4: Datenfluss bei Erstellung eines To-Do-Eintrages

Abb. 5.4 zeigt den Datenfluss beim Erstellen eines To-Do-Eintrags. Die To-Do-Beschreibung wird vom Nutzer über ein Eingabefeld an den Service weitergereicht. Dieser erstellt ein entsprechendes Datenobjekt und reicht es an die Stammkomponente weiter, worüber schlussendlich **HTML**-Code erzeugt werden kann. Gleichzeitig speichert er das neue Element im Browserspeicher ab.

5.2.2.3 Beschreibung der Dateistruktur und Komponenten der Angular-Anwendung

Nach dem im Vorangegangenen beschrieben worden ist, wie das Projekt erstellt worden ist, wird im Folgenden detaillierter auf das erstellte Angular-Projekt eingegangen. Angular, welches stark komponentenorientiert aufgebaut ist, zeigt sich auch in seiner Dateistruktur stark komponentenbezogen. In Abb. 5.5 sind die wichtigsten Dateien in einem Diagramm dargestellt.

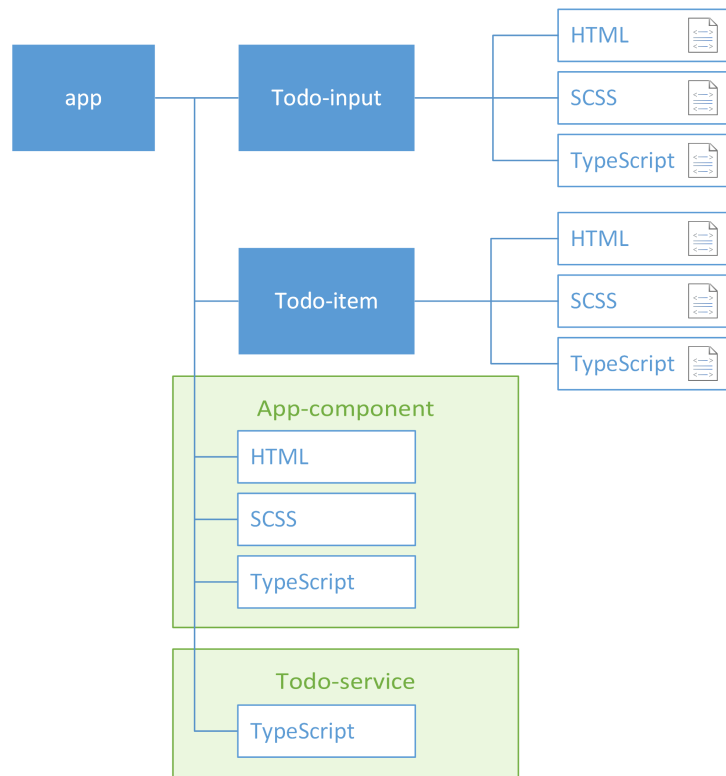


Abbildung 5.5: Dateistruktur der Angularanwendung

Stammkomponente Zunächst befinden sich im `app`-Verzeichnis eine **HTML**-, eine Stylesheet- und eine TypeScript-Datei des `app-component`, welcher die Stammkomponente darstellt. Sie dient als Container (Begriffserklärung s. Sektion 2.1) um die To-Do-Elemente und das Eingabefeld (siehe Ausschnitt 7, Z. 5).

```

1 <ng-container *ngFor="let item of todoService.items">
2   <app-todo-item [item]="item"></app-todo-item>
3 </ng-container>
4
5 <app-todo-input></app-todo-input>

```

Quellcode-Ausschnitt 7: **HTML**-Template des `app-component` (Angular **HTML**)

Ausschnitt 7 (Zz. 1–3), zeigt, wie die Daten aus dem `todoService` in **HTML**-Code dargestellt werden. Wie in einer `for`-Schleife wird jedes `item` des `todoService` einer neu erstellten `todo-item`-Komponente übergeben. Angular wird das **HTML**-Template automatisch neu rendern, wenn sich die Elemente im `todoService` ändern.

To-Do-Element Das `todo-item` ist ein einzelnes Element der To-Do-Liste. Es enthält einen Button zum priorisieren, eine Checkbox zum Abhaken des Elements, eine Textbox und einen Button zum Löschen (s. Abb. 5.6).

Abbildung 5.6: Gerenderte `todo-item`-Komponente

Es ist sinnvoll, jedes Listenelement in einer mehrfach instanziierten Komponente zu repräsentieren. Dies kapselt die Daten des Listenelements und die Kommunikation mit dem `todoService` von anderen Listenelementen ab. Wie bereits erwähnt, bekommt jede `todo-item`-Komponente genau ein Objekt der Klasse `TodoItem` übergeben.

```

1 <!-- PRIORITIZE BUTTON -->
2 <input type="checkbox" [(ngModel)]="item.important" (change)="updateTodoItem()" />
3
4 <!-- CHECKBOX -->
5 <input type="checkbox" [(ngModel)]="item.done" (change)="updateTodoItem()" />
6
7 <!-- TEXT INPUT -->
8 <input type="text" [(ngModel)]="item.description" (keyup)="updateTodoItem()" />
9
10 <!-- DELETE BUTTON -->
11 <button (click)="todoService.deleteTodoItem(item.id)">#9587;</button>

```

Quellcode-Ausschnitt 8: HTML Template der `todo-item` (gekürzt)

Quellcode-Ausschnitt 8 zeigt das **HTML**-Template des `todo-items`. Die ersten drei Input-Elemente werden durch `[(ngModel)]` mit einer *Property* (dt. *Eigenschaft*) des `TodoItems` verknüpft, welche in der `todo-item`-Komponente gespeichert wird. Dies bewirkt folgenden Mechanismus: Wird das Input-Element verändert, ändert sich auch das gespeicherte `TodoItem`. Wird das `TodoItem` im Code modifiziert, aktualisiert sich das Input-Element entsprechend.

Um modifizierte Daten auch in der Datenquelle zu speichern, wird das modifizierte `TodoItem` bei jeder Änderung dem `todoService` übergeben.

5.2.2.4 Kommunikation der Komponenten mit dem `todoService`

Abb. 5.7 fasst die Architektur der Angular-Anwendung zusammen. Den zentralen Punkt der Anwendung bildet der `TodoService`.

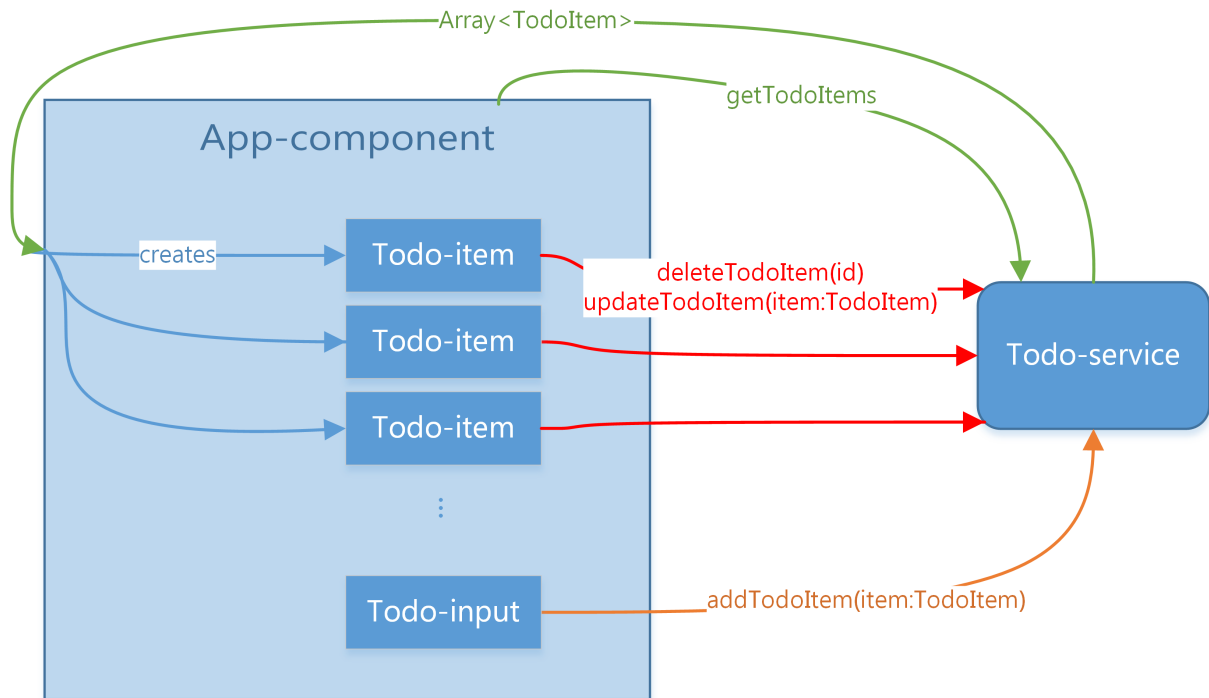


Abbildung 5.7: Interaktion des Todo-service mit den Komponenten

Grün dargestellt ist der Datenfluss der gespeicherten To-Do-Elemente zwischen der Stammkomponente und dem `todoService`. Die Stammkomponente erhält ein Array aus `TodoItems` und wird von Angular automatisch aktualisiert, wenn dieses sich ändert. Die Stammkomponente erstellt für jedes `TodoItem` eine `TodoItem`-Komponente (blau dargestellt), die so als Listeneintrag sichtbar wird.

Orange dargestellt ist das Einfügen von Daten durch den Nutzer im Input-Feld der App. Der Input stellt dem `todoService` ein neues `TodoItem`-Objekt bereit.

Rot dargestellt ist das Löschen oder Aktualisieren von Items. Wird der Button zum Entfernen gedrückt, oder ändert sich das gespeicherte `TodoItem`-Objekt der `TodoItem`-Komponente, werden die Änderungen dem `todoService` weitergegeben. Für das Löschen reicht allerdings die einzigartige ID des Objekts aus.

5.2.2.5 Gestaltung des User Interface

Das Styling des von Angular erzeugten **HTML**-Codes erfolgt mittels **CSS**. In den bereits erwähnten **CSS**-Dateien wird den einzelnen Elementen ein Style hinzugefügt. Erwähnenswert ist dabei, dass alle Anforderungen im Code einzeln beschrieben werden müssen, d.h. Schriftart, Position, Farbe, etc. sind einzeln zu deklarieren.

Mit der Nutzung der **CSS**-Eigenschaft *Flexbox* lassen sich Elemente responsiv gestalten. Die Nutzung eines **CSS** oder **UI**-Frameworks würde diese Aufgabe in einem größeren Pro-

jekt erleichtern. Mit bekannten Frameworks, wie bspw. dem *Material UI*-Framework von Google, lassen sich Elemente dem Material-Design-Prinzip anpassen (vgl. **MaterialUI**), dass auf ähnliche Weise auch bei Android genutzt wird. Hierzu existieren eine Vielzahl von Frameworks mit eigenen Designphilosophien, auf die in dieser Arbeit nicht weiter eingegangen wird.

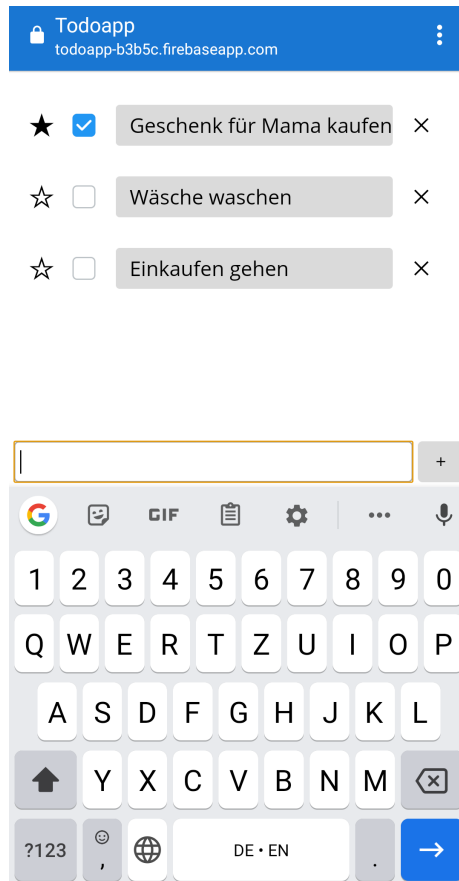


Abbildung 5.8: Screenshot der **PWA** mit geöffneter Tastatur

Abb. 5.8 zeigt einen Screenshot der **PWA**, nachdem diese auf dem Gerät installiert wurde. Tippt der Nutzer in das Eingabefeld, öffnet sich die Tastatur und das **UI** passt sich an die neuen Größenverhältnisse an.

5.2.2.6 Umsetzung der Benachrichtigung über unerledigte Aufgaben

Die definierten Anforderungen geben an, dass der Nutzer an seine Aufgaben erinnert werden soll. In der Praxis stellt sich jedoch heraus, dass die Web-Push-Benachrichtigungen über den Browser zwar sehr gut mit Webanwendungen funktionieren, aber nicht mit **PWAs** ohne Internetverbindung. Mit einem Server, der zu einem bestimmten Zeitpunkt eine Nachricht an die Webanwendung schickt, ist die Umsetzung möglich, allerdings nur bei bestehender Internetverbindung. Die Anforderung wird deshalb als *nicht umsetzbar* betrachtet, da das Kriterium der Offline-Verfügbarkeit nicht umgesetzt werden kann.

5.2.3 Integration einer Manifestdatei

Die Anwendung funktioniert zwar bereits im Browser, ist bislang aber noch keine **PWA**. Um dem Projekt ein *Manifest* und einen *Service Worker* hinzuzufügen, kann wieder das Angular **CLI** verwendet werden. `ng add @angular/pwa` erzeugt die entsprechenden Dateien automatisch, fügt eine `manifest.json`-Datei hinzu und bindet diese in die `index.html`-Datei des Projekts ein.

Es ist festzuhalten, dass das Hinzufügen der **PWA**-Funktionalität keinen nennenswerten Aufwand mit sich bringt, wenn das Ziel ausschließlich die Installation der Web-App ist.

5.2.4 Hosting der App mit Firebase

Damit die **PWA** als solche installiert werden kann, muss sie bestimmte Kriterien erfüllen (s. Sektion 2.4). Darunter ist die Notwendigkeit der Bereitstellung über Hypertext Transfer Protocol Secure (**HTTPS**). Dies gestaltet sich in der Praxis schwierig, da der Angular-Entwicklungsserver zwar auf **HTTPS** konfiguriert werden kann, jedoch selbst signierte **SSL**-Zertifikate nicht akzeptiert werden. Die **PWA** kann somit nicht installiert werden. Für die Entwicklung ist dies natürlich problematisch, sodass eine Lösung für dieses Problem gefunden werden muss.

Für die spätere Evaluation ist der hier verwendete Hosting-Anbieter nicht relevant. Es gibt zahlreiche Alternativen, die den Anforderungen an das Hosting der Anwendung gerecht werden. Auf diese soll hier aber nicht weiter eingegangen werden. Da das Projekt jedoch aus angeführten Gründen gehostet werden muss, wird das Hosting stellvertretend mit einem Google-Service erläutert.

Googles stellt eine schnelle und elegante Lösung zum Hosten einer Webanwendung bereit: *Firebase*. Über die *Firebase Console*, eine Webanwendung zur Verwaltung von Firebase-Projekten, kann innerhalb weniger Minuten ein Projekt inklusive Hosting erstellt werden.

Mit dem Firebase-**CLI** können automatisiert Konfigurationsdateien für das Firebase-Hosting angelegt werden. Die Schritte dazu sind ebenfalls trivial:

1. Anmeldung:

Das **CLI** muss mit einem Google-Konto verknüpft werden. Dazu beim Login über das **CLI** ein Browserfenster mit einem Login-Dialog von Google.

2. Initialisierung:

Die Initialisierung über das **CLI** legt u.a. eine **JSON**-Datei zur Konfiguration des Deployments an. Hier werden Dateipfade, wie bspw. die Start-**URL** oder Dateien, welche deployt werden sollen, gespeichert.

3. Deployment:

Über das Angular-**CLI** wird ein sog. *Production-Build* erstellt. Dies ist eine gepackte Version der Webanwendung für den Produktivbetrieb. Anschließend kann die gepackte Anwendung mit **firebase deploy** auf einem von Googles Servern bereitgestellt werden.

Die Anwendung läuft jetzt mit einer validen **HTTPS**-Verbindung und kann von Nutzern installiert werden.

Das Hosting mit Firebase löst gleichzeitig ein weitere Problem: Das Testen der Anwendung auf einem Smartphone. Die von Firebase bereitgestellte **URL** kann jetzt einfach im mobilen Chrome-Browser aufgerufen und installiert werden.

5.2.5 Installation der Anwendung auf Smartphone und Desktop

Smartphones: Nach dem Aufrufen der **URL** mit dem mobilen Browser erscheint eine Meldung zum Installieren der **PWA** (s. Abb. 5.9).

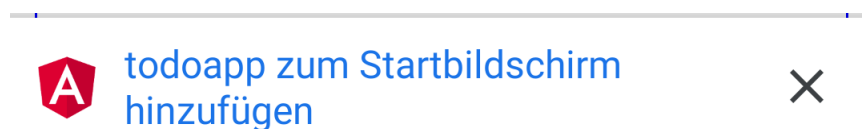


Abbildung 5.9: Browser-Dialog zum Installieren der **PWA** als mobile App

Desktop: In der Suchleiste von Chrome kann der Nutzer die **PWA** als Desktopanwendung installieren (s. Abb. 5.10).

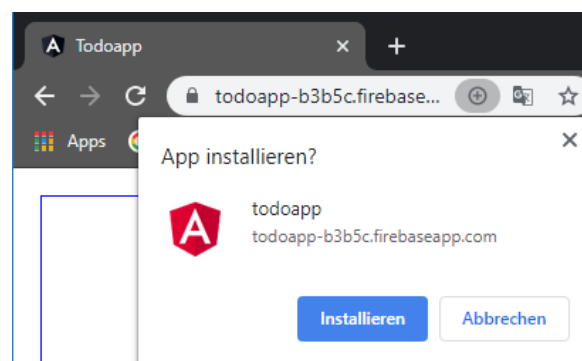


Abbildung 5.10: Browser-Dialog zum Installieren der **PWA** als Desktop-App

Damit dem Nutzer Benachrichtigungen tatsächlich angezeigt werden, muss er beim Erhalten der ersten Nachricht die Benachrichtigungen über einen Dialog aktivieren (s. Abb. 5.11).

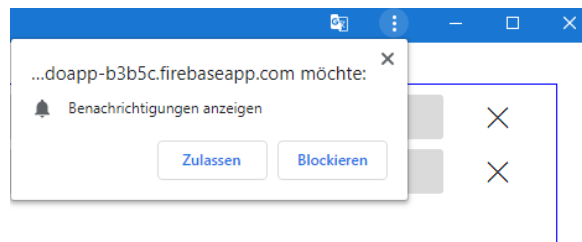


Abbildung 5.11: Browser-Dialog für Benachrichtigungen

5.2.6 Aktualisieren der Anwendung

Um das Update-Verhalten zu evaluieren, ist es interessant, den Update-Prozess einer Webanwendung bzw. einer **PWA** zu betrachten. Um Änderungen an die Nutzer zu verteilen, muss ein Entwickler einen neuen *Production Build* auf dem Server bereitstellen. Im Falle dieses Projekts werden Änderungen unter Nutzung des **Firebase-CLI** auf den Server übertragen.

In der Praxis werden die Änderungen in der installierten **PWA** nicht sofort sichtbar, wohingegen die Webanwendung beim nächsten Refresh der Seite aktualisiert wird. Dies hängt mit dem Caching der Ressourcen der **PWA** zusammen. Der Service Worker lädt cachebare Dateien entweder beim Start oder nachträglich, während die **PWA** läuft, in den lokalen Cache des Geräts.

Um dieses Verhalten zu Testen, wurde ein Update mit einer auffälligen Hintergrundfarbe auf dem Server bereitgestellt. In der Desktop-**PWA** war die Änderung erst nach einem System-Neustart zu sehen, wohingegen sich die **PWA** unter Android nach einigen Stunden automatisch aktualisiert hatte. Dabei gab es jeweils keine Benachrichtigung zur Aktualisierung für den Nutzer.

6 Evaluation

7 Reflexion