



Vergleich der drei Cross-Plattform Frameworks Flutter, React Native und Ionic anhand praxisbezogener Use-Cases

Bachelorarbeit an der Hochschule Ravensburg-Weingarten
Fakultät Elektrotechnik und Informatik
Studiengang Angewandte Informatik

Vorgelegt von Ruben Röhner
Matrikel-Nr.: 32889

19. April 2023

1. Gutachter: Prof. Thorsten Weiss
2. Gutachter: Prof. Jürgen Graef

Eidesstattliche Erklärung

Diese Abschlussarbeit wurde von mir selbständig verfasst. Es wurden nur die angegebenen Quellen und Hilfsmittel verwendet. Alle wörtlichen und sinngemäßen Zitate sind in dieser Arbeit als solche kenntlich gemacht.

Ort, Datum und Unterschrift

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die auf unterschiedlichste Art und Weise zum Gelingen dieser Arbeit beigetragen haben.

Ein besonderer Dank gilt Herrn Prof. Thorsten Weiss für die Betreuung und Ermöglichung dieser Bachelorarbeit. Für die stetige Erreichbarkeit bei Fragen, die konstruktive Kritik und die vielen hilfreichen Anregungen möchte ich mich bedanken.

Mein Dank gilt darüber hinaus auch all meinen Freunden und Familienmitgliedern, die mich in dieser Zeit begleitet und unterstützt haben.

Inhaltsverzeichnis

1	EINLEITUNG	7
2	STAND DER TECHNIK	8
2.1	Entwicklung nativer Applikationen	8
2.1.1	Android	8
2.1.2	iOS	8
2.2	Ansätze der Cross-Plattform Entwicklung	9
2.3	Flutter	11
2.3.1	Architektur	11
2.3.2	Kernkonzepte	12
2.3.3	Rendering	13
2.3.4	Tools	16
2.3.5	Interoperabilität	17
2.4	React Native	18
2.4.1	Architektur	18
2.4.2	Kernkonzepte	19
2.4.3	Rendering	19
2.4.4	Tools	22
2.4.5	Interoperabilität	23
2.5	Ionic	25
2.5.1	Architektur	25
2.5.2	Kernkonzepte	26
2.5.3	Rendering	27
2.5.4	Tools	28
2.5.5	Interoperabilität	29
3	USE-CASES	31
3.1	Listen	32
3.1.1	Implementierung mit Flutter	33
3.1.2	Implementierung mit Ionic	33
3.1.3	Implementierung mit React Native	34
3.1.4	Performance-Analyse während dem Scrollen	34

3.2	Cards	36
3.2.1	Implementierung mit Flutter	37
3.2.2	Implementierung mit Ionic	37
3.2.3	Implementierung mit React Native	38
3.3	Canvas	39
3.3.1	Implementierung mit Flutter	40
3.3.2	Implementierung mit Ionic	40
3.3.3	Implementierung mit React Native	41
3.4	REST API	43
3.4.1	Implementierung mit Flutter	44
3.4.2	Implementierung mit Ionic	45
3.4.3	Implementierung mit React Native	46
3.5	Persistenz	47
3.5.1	Implementierung mit Flutter	48
3.5.2	Implementierung mit Ionic	49
3.5.3	Implementierung mit React Native	50
3.6	Custom Design	51
3.6.1	Implementierung mit Flutter	52
3.6.2	Implementierung mit IONIC	53
3.6.3	Implementierung mit React Native	53
4	DISKUSSION	54
4.1	Zusammenfassung der Ergebnisse aus den Use-Cases	54
4.2	Vergleich der Frameworks	56
5	ZUSAMMENFASSUNG	60
5.1	Flutter	60
5.2	Ionic	60
5.3	React Native	61
6	AUSBLICK	62
	QUELLENVERZEICHNIS	63

Abkürzungsverzeichnis

HTML	Hypertext Markup Language	9
CSS	Cascading Style Sheets	9
API	Application Programming Interface	9
JIT	Just in Time Compiler	12
AOT	Ahead of Time Compiler	12
GPU	Graphics Processing Unit	13
CPU	Central Processing Unit	16
JSON	JavaScript Object Notation	17
JSX	JavaScript Syntax Extension	19
DOM	Document Object Model	28
CLI	Command Line Interface	29
SDK	Software Development Kit	23
SQL	Structured Query Language	49
FPS	Frames Per Second	34
URL	Uniform Resource Locator	43
FAB	Floating Action Button	43
REST	Representational State Transfer	31
HTTP	Hypertext Transfer Protocol	44

1 Einleitung

In der heutigen digitalen Welt sind mobile Anwendungen zu einem wichtigen Bestandteil unseres täglichen Lebens geworden. Sie ermöglichen es uns, mit Freunden und Familie in Kontakt zu bleiben, online einzukaufen, unsere Gesundheit zu überwachen und vieles mehr. Angesichts der steigenden Anzahl von Smartphone-Nutzern ist es für Unternehmen unerlässlich geworden, ihre Produkte und Dienstleistungen auch auf mobilen Geräten anzubieten.

Die Entwicklung mobiler Anwendungen für verschiedene Plattformen kann sehr zeitaufwändig und kostspielig sein. Eine Möglichkeit, diese Herausforderung zu bewältigen, ist die Entwicklung plattformübergreifender mobiler Anwendungen, die auf verschiedenen Betriebssystemen laufen können. Dies wird als Cross-Plattform Entwicklung bezeichnet.

Die beiden wichtigsten mobilen Betriebssysteme, Android und iOS, dominieren den Markt mit einem gemeinsamen Marktanteil von über 99% weltweit [22]. Aus diesem Grund konzentriert sich diese Arbeit auf Frameworks, die für die Cross-Plattform Entwicklung auf diesen beiden Plattformen entwickelt wurden. Dabei werden auf Grund des zeitlichen begrenzten Rahmens einer Bachelorarbeit nur Smartphone Use-Cases berücksichtigt.

Cross-Plattform Entwicklung bietet mehrere Vorteile gegenüber der plattformspezifischen Entwicklung. Eine Anwendung für mehrere Plattformen zu erstellen beschleunigt den Entwicklungsprozess und senkt die Kosten. Da Entwickler nur eine Codebasis schreiben müssen, können sie Zeit und Ressourcen sparen und die Anwendung schneller auf den Markt bringen.

Ein weiterer Vorteil der Cross-Plattform Entwicklung ist die Möglichkeit, eine größere Zielgruppe zu erreichen. Mit nur einer Codebasis kann eine Anwendung für verschiedene Plattformen bereitgestellt werden, wodurch die Reichweite der Anwendung erhöht wird. Dies kann dazu beitragen, dass Unternehmen mehr Nutzer erreichen und somit mehr Umsatz generieren können.

Ziel dieser Bachelorarbeit ist es, die Stärken der Frameworks React Native, Ionic und Flutter herauszuarbeiten und anhand dieser Informationen Entwickler oder Unternehmen bei der Entscheidung zu unterstützen, welche Technologie für ihre spezifischen Anforderungen am besten geeignet ist. Durch den Vergleich von Vor- und Nachteilen der Frameworks, anhand praxisbezogener Use-Cases, soll die Arbeit dabei helfen, eine fundierte Entscheidung zu treffen, welche plattformübergreifende Entwicklungslösung für die Entwicklung welcher mobiler Anwendungen am besten geeignet ist.

2 Stand der Technik

2.1 Entwicklung nativer Applikationen

Die Entwicklung nativer Anwendungen beschreibt die Programmierung mobiler Anwendungen, die nur auf der spezifischen Plattform lauffähig sind, für die sie entwickelt wurden.

In dieser Arbeit wird sich ausschließlich auf die beiden größten Plattformen - iOS und Android - im Smartphone-Bereich konzentriert. Alle anderen Betriebssysteme für Smartphones haben zusammen einen Marktanteil von weniger als einem Prozent und können daher für diesen Vergleich vernachlässigt werden [22]. Weitere Gerätetypen wie Tablets werden ebenfalls nicht berücksichtigt, um den zeitlichen Rahmen der Bachelorarbeit nicht zu überschreiten.

2.1.1 Android

Android ist das beliebteste Betriebssystem für Smartphones mit einem Marktanteil von 71,47% (August 2022 [22]). Die meisten Smartphone-Hersteller setzen auf dieses Betriebssystem. Native Android Anwendungen werden mit der Entwicklungsumgebung Android Studio entwickelt. Als Programmiersprache kann zwischen Kotlin und Java gewählt werden [14].

Jetpack Compose ist das neue UI-Toolkit von Google zur Erstellung von Benutzeroberflächen in Android. Jetpack Compose unterscheidet sich von der klassischen Programmierung mit XML dadurch, dass es einen deklarativen Ansatz zur Erstellung von UI-Elementen verwendet. Dadurch wird der Programmcode übersichtlicher und leichter verständlich [12].

2.1.2 iOS

iOS ist das Betriebssystem für Smartphones des Unternehmens Apple, welches auch von Apple selbst entwickelt wird. iOS ist, mit einem Marktanteil von 27,88% (August 2022 [22]), das zweitbeliebteste Betriebssystem für Smartphones. Zur Entwicklung von Anwendungen für iOS wird die hauseigene Entwicklungsumgebung Xcode verwendet. Als Programmiersprachen kommen Objective-C oder Swift zum Einsatz [2].

Apple hat außerdem ein neues deklaratives UI-Toolkit für die Entwicklung von Benutzeroberflächen für iOS-Anwendungen namens SwiftUI entwickelt. Auch SwiftUI soll den Entwicklungsprozess beschleunigen und den Programmcode übersichtlicher und verständlicher machen [3].

2.2 Ansätze der Cross-Plattform Entwicklung

Cross-Plattform Frameworks können auf verschiedenen Konzepten basieren. In dem Artikel von Xanthopoulos et al. aus dem Jahr 2013 [31] wurden vier verschiedene Ansätze für Cross-Plattform Frameworks definiert und miteinander verglichen. Diese vier Ansätze - Web, hybride, interpretierte und generierte Applikationen - werden in dem folgenden Kapitel vorgestellt [31].

Web-Applikationen

Der erste Ansatz zur Erstellung von mobilen Applikationen, welche auf mehreren Plattformen ausgeführt werden können, ist in Form einer Webanwendung. Diese werden mit den typischen Programmiersprachen der Webentwicklung – Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) und JavaScript – erstellt. Heutzutage werden immer häufiger Frontend-Frameworks wie React, Angular und vue.js verwendet. Diese vereinfachen und verkürzen den Entwicklungsprozess von Webanwendungen deutlich [31].

Auf mobilen Endgeräten werden diese Webanwendungen in einem Browser ausgeführt, das heißt diese Art von mobilen Applikationen werden nicht auf dem Gerät installiert. Sie werden nur aus dem Internet aufgerufen, wenn diese benötigt werden. Dadurch sind Webanwendungen nur mit einer bestehenden Internetverbindung nutzbar [31].

Webanwendungen haben nur eingeschränkten Zugriff auf Gerätehardware, wie zum Beispiel die Bluetooth-Funktion oder die Kamera. Dadurch sind die Einsatzmöglichkeiten deutlich eingeschränkt. Da Webanwendung auch Einbußen in der Performance mit sich bringen, sind sie für rechenintensive Applikationen, wie zum Beispiel 3D-Spiele, ungeeignet [31].

Hybride Applikationen

Hybride Applikationen sind eine Verbindung aus Webanwendungen und nativen Applikationen. Ausgeführt wird hier eine Webanwendung in einem nativen Container. Für iOS-Anwendungen typischerweise in einer `WKWebView` und für Android-Anwendungen in einer `android.webkit.WebView`. Dieser Container dient als Ausführungsumgebung der Webanwendung. Mit einem Application Programming Interface (API) stehen der Applikation gerätespezifische Funktionen, wie der Zugriff auf das Dateisystem oder die Kamera, zu Verfügung [31].

Auch bei hybriden Applikationen ist der Umstieg für einen Webentwickler deutlich einfacher, da hier ebenfalls eine Webanwendung entwickelt wird. Dadurch, dass die Anwendung in einem Container ausgeführt wird, gibt es Einschränkungen in der Performance gegenüber nativen Applikationen. Dies bedeutet auch für hybride Applika-

tionen, dass sie nicht für rechenintensive Aufgaben geeignet sind. Ein Vorteil gegenüber Webanwendungen ist die lokale Installation und Ausführung der hybriden Applikation, weshalb diese Anwendungen auch ohne Internetzugang nutzbar sind [31].

Ionic, beispielsweise, ist ein Framework für die Entwicklung von hybriden Applikationen. Die genaue Funktionsweise wird in Kapitel 2.5 erläutert, während in Kapitel 4 ein Vergleich mit anderen Cross-Plattform Frameworks durchgeführt wird [7].

Interpretierte Applikationen

Bei dieser Art von Applikationen wird der Programmcode zur Laufzeit in die jeweilige native Programmiersprache interpretiert. Dazu wird eine spezielle Laufzeitumgebung verwendet. Da im Endeffekt nativer Code ausgeführt wird, können native UI-Elemente verwendet werden, welche besonders performant sind und somit die Gesamtperformance der interpretierten Applikationen gegenüber hybriden oder Web-Applikationen deutlich erhöht. Neue Funktionen der nativen Plattformen müssen aber zunächst von den Frameworks entwickelt werden, um dann später in der Applikation verwendet werden zu können [31].

React Native verwendet diesen Ansatz. Mit dem Frontend Framework React wird eine Webanwendung entwickelt und der Programmcode wird zur Laufzeit in nativen Programmcode interpretiert und ausgeführt [24]. Die genaue Funktionsweise von React Native wird im Kapitel 2.4 erläutert.

Generierte Applikationen

Der letzte Cross-Plattform Ansatz sind generierte Applikationen, welche zu nativen Applikationen kompiliert werden. Bei diesen Applikationen wird ein Kompilierer verwendet, um den Programmcode aus der Programmiersprache des Cross-Plattform Frameworks in die native Programmiersprache der Plattform zu kompilieren. Dadurch wird auch bei diesem Ansatz eine native Applikation installiert. Diese müssen jedoch für jede Plattform separat kompiliert werden [31].

Das Cross-Plattform Framework Flutter verwendet diesen Ansatz [17]. Die Funktionsweise von Flutter Applikationen wird genauer im Kapitel 2.3 beschrieben.

2.3 Flutter

Flutter ist das von Google entwickelte Cross-Plattform Framework, welches auf der objektorientierten Programmiersprache Dart basiert. Der Dart-Code wird direkt in maschinenlesbaren Programmcode für x64- oder arm-basierte Systeme kompiliert. Dadurch gehören Flutter Applikationen zu den generierten Applikationen (vgl. Kapitel 2.2). Unterstützt werden die Plattformen iOS, Android, Windows, MacOS, Linux und Web [17].

2.3.1 Architektur

Das Framework ist in drei verschiedene Schichten aufgebaut, einem plattformspezifischen Embedder, die Engine und das Dart Framework. Der Embedder ist die unterste Schicht der Architektur. Er ist die Schnittstelle zwischen der Engine und dem Betriebssystem und stellt der Engine wichtige Funktionen, wie zum Beispiel für das Rendern von Inhalten, zu Verfügung [17].

Die Engine, welche in C und C++ geschrieben ist, ist die zweite Schicht der Architektur. Hier sind hardwarenahe Funktionen der Flutter API implementiert. Außerdem übernimmt die Engine das Rendern von zusammengesetzten Szenen. Dafür wird die Grafik-Bibliothek Skia verwendet, welche auch in anderen Softwareprodukten von Google, wie ChromeOS oder Android, eingesetzt wird [15]. Die Szenen einer Flutter Applikation werden aus Performance-Gründen zusammengesetzt gerendert. Dadurch kann die Engine entscheiden, welche Teile der Szene neu gerendert werden müssen und auch nur diese UI-Elemente neu rendern und den Rest der Szene beibehalten. Dadurch wird die Performance von Flutter Applikationen deutlich verbessert. Die `dart:ui` Bibliothek ist ein wichtiger Teil dieser Schicht, da diese die Verbindung zwischen der zweiten und dritten Schicht darstellt [17].

Die oberste Schicht der Flutter Architektur ist das, in Dart geschriebene, Framework. Es besitzt die API, mit der die Entwickler arbeiten. Es kommuniziert mit der darunter liegenden Engine. Das Framework beinhaltet die grundlegenden Klassen, zum Beispiel für Animationen oder Gesten. Ein weiterer wichtiger Teil des Frameworks ist die Rendering-Schicht, sie bietet die Möglichkeit den Rendering-Baum zu manipulieren. Eine eigene Schicht des Frameworks ist alleine für die UI-Elemente von Flutter zuständig [17].

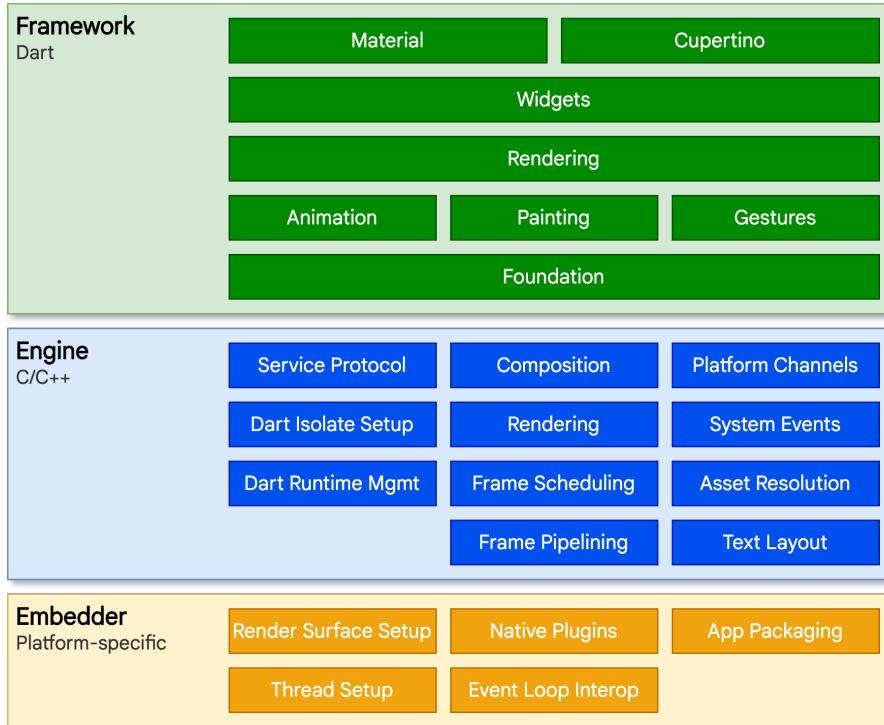


Abbildung 1: Architektur des Flutter Frameworks [17].

2.3.2 Kernkonzepte

Dart

Dart ist die Programmiersprache, mit welcher Flutter Applikationen geschrieben werden. Dart ist eine objektorientierte Programmiersprache, die sowohl typsicher als auch NULL-sicher ist [11].

Um die Typsicherheit des Programmcodes zu gewährleisten wird eine statische Analyse aller Variablen während dem Kompilieren durchgeführt und geprüft, ob der Wert der Variablen dem definierten Typ entspricht. Zusätzlich wird eine dynamische Analyse der Typen zur Laufzeit durchgeführt [11].

Die NULL-Sicherheit im Dart Programmcode wird durch den `?-Operator` gesteuert. Dieser kann an das Ende des Typs angehängt werden und somit kann der Wert dieser Variable NULL sein. Fehlt dieser `?-Operator` so darf der Wert nicht NULL sein und eine `NullPointerException` wird gegebenenfalls geworfen [11].

Es gibt mehrere Kompilierer die den Dart Programmcode in den gewünschten plattformspezifischen Programmcode kompilieren. Um Dart in x64 oder ARM Maschinencode zu kompilieren wird entweder der Just in Time Compiler (JIT) oder der Ahead of Time Compiler (AOT) verwendet. Der JIT wird wegen seiner Schnelligkeit während der Entwicklung verwendet und ermöglicht das sogenannte Fast Refresh. Der AOT dagegen wird für das Bauen von produktionsfähigen Applikation verwendet [11].

Um den Dart Programmcode in JavaScript zu übersetzen wird ein extra Kompilierer verwendet, welcher in zwei verschiedenen Modi ausgeführt werden kann. Er besitzt einen Modi für die schnelle Kompilierung zur Verwendung während der Entwicklung und einen optimierten Kompilierer, um eine produktionsfähige Webanwendung zu bauen [11].

UI-Elemente

UI-Elemente werden im Kontext von Flutter als Widgets bezeichnet. Diese Widgets werden in einer baumartigen Struktur ineinander verschachtelt, wodurch der gesamte Baum die Benutzeroberfläche der Applikation repräsentiert [17].

Flutter setzt ausschließlich auf eigen-implementierte UI-Elemente. Dies bedeutet, dass die UI-Elemente nicht durch native UI-Elemente ersetzt werden, sondern dass diese von einer eigenen Rendering Engine gezeichnet werden. Flutter begründet diese Entscheidung damit, dass dadurch die Performance und Erweiterbarkeit von Widgets besser gelöst sei [17]. Außerdem unterscheiden sich die UI-Elemente dadurch nicht, wenn sie auf verschiedenen Betriebssystemen und Betriebssystemversionen ausgeführt werden. Flutter bietet trotz dessen die Möglichkeit native UI-Elemente in eine Flutter Applikation zu integrieren [17].

Der Aufbau eines Widgets wird in der `build` Funktion festgelegt. Diese Funktion wird während dem Rendering Prozesses ausgeführt, um den `widget tree` zu erstellen [17]. Flutter unterscheidet grundsätzlich zwischen zwei Arten von Widgets, den stateful Widgets und den stateless Widgets. Stateful Widgets besitzen einen eigenen Zustand, welcher zur Laufzeit verändert werden kann. Dies führt dazu, dass Teile dieses Widgets mit dem neuen Zustand neu gezeichnet werden müssen. Mit der `setState` Funktion wird dem Framework mitgeteilt, dass der Zustand des Widgets verändert wurde. Daraufhin zeichnet Flutter den Teil des Widgets neu, der von dieser Änderung betroffen ist. Stateless Widgets dagegen besitzen keinen eigenen Zustand, der zur Laufzeit verändert werden kann. Ändern sich Eigenschaften eines stateless Widgets, so muss dieses komplett neu gezeichnet werden [17].

2.3.3 Rendering

Die Rendering Pipeline des Frameworks Flutter beschreibt den kompletten Ablauf von der Interaktion des Nutzers bis zur Ausführung der Instruktionen auf der Graphics Processing Unit (GPU). Diese Rendering Pipeline ist in sieben Schritte aufgeteilt, welche nacheinander ausgeführt werden [17]. Der Ablauf und die Funktionsweise dieser Rendering Pipeline wird im folgenden Abschnitt erläutert:

1. User Input

Der erste Schritt der Rendering Pipeline beinhaltet die Reaktion des Frameworks auf die Interaktion des Nutzers [17].

2. Animation

Im nächsten Schritt werden die Animationen ausgeführt, um dem Nutzer ein visuelles Feedback auf die Eingabe zu geben. Ein Beispiel dafür ist eine Druck-Animationen eines Knopfes, nach dem auf diesen getippt wurde [17].

3. Build

Die Build Phase ist ein essenzieller Bestandteil der Rendering Pipeline. Die `build` Funktion erstellt den sogenannten `widget tree`. Dieser `widget tree` ist eine Repräsentation der Benutzeroberfläche, in einer baumartigen Struktur [17].

Die Knoten dieses Baumes sind Widgets, welche im Dart Programmcode durch den Entwickler festgelegt wurden. Je nach Widget, können diese durch das Framework hinzugefügt worden sein. Die Elemente dieses Baumes können in zwei verschiedene Typen unterteilt werden `ComponentElement` und `RenderObjectElement`. Ein `ComponentElement` besteht aus weiteren Elementen, besitzt aber selbst keine visuelle Repräsentation auf dem Bildschirm. Ein `RenderObjectElement` dagegen wird in der layout und paint Phase berücksichtigt und ist dementsprechend auf dem Bildschirm sichtbar. Das `RenderObjectElement` dient als Vermittler, zwischen dem im Dart Programmcode beschriebenen Widget und dem `RenderObject` welches tatsächlich auf dem Bildschirm gezeichnet wird [17].

4. Layout

Die Layout Phase ist die erste Phase des eigentlichen Rendering Prozesses. In dieser Phase werden Position und Größe der Elemente ermittelt [17].

Um die Größe jedes Elementes zu ermitteln, durchläuft der Algorithmus den `element tree` nach dem Prinzip der Tiefensuche. Jeder Elternknoten übermittelt dabei `SizeConstraints` an seine Kindelemente. Diese `SizeConstraints` sind ein einfaches Modell, bestehend aus jeweils einer minimalen und maximalen Breite beziehungsweise Höhe. Ist der Algorithmus am untersten Ende des Baumes angekommen, ermittelt der unterste Knoten seine Größe mit Hilfe der übergebenen `SizeConstraints` und gibt seine tatsächliche Größe an seinen Elternknoten zurück. Dieser Elternknoten ermittelt nun seine Größe, mit den Größen seiner Kindelemente und der `SizeConstraints` seines Elternknotens. Nach diesem Prinzip wird der Baum nun bis zum Wurzelknoten nach oben rekursiv durchlaufen, bis jedes Element seine Größe ermittelt hat [17].

Die Position jedes Elementes wird durch den jeweiligen Elternknoten bestimmt, das Kindelement kennt dabei nur seine Größe und nicht seine Position. Jeder El-

ternknoten ordnet seine Kindelemente nach einer vorgegebenen Regel, dies kann zum Beispiel ein **FlexLayout** sein, in dem die Kindelemente horizontal nebeneinander angeordnet werden [17].

5. Paint

Im Paint Schritt werden aus den Elementen des `widget tree` visuelle Repräsentationen erstellt, den `RenderObjects`. Dafür wird auch hier der Baum mit einer Tiefensuche durchlaufen, wobei jeder Elternknoten eine Offset Variable an seine Kind Elemente weitergibt. Dieser Offset enthält die Position, an dem sich das Kind Element zeichnen soll [17].

Die Benutzeroberfläche in Flutter wird in mehreren Schichten gezeichnet, um komplexe Benutzeroberflächen erstellen zu können. Außerdem soll somit die Performance von Flutter Applikationen verbessert werden, in dem Schichten, die sich nicht verändert haben nicht immer wieder neu gezeichnet werden müssen. Die Kindelemente geben ihrem Elternknoten eine Information zurück, welche dem Elternknoten sagt, in welchem Layer sie sich zeichnen soll. Dies kann dieselbe oder eine höhere Schicht sein als die Schicht, in der sich das Kindelement gezeichnet hat [17].

6. Composite

Der nächste Schritt des Rendering Prozesses ist der Composite Schritt. In diesem werden die Schichten, die im vorherigen Schritt erstellt wurden, zusammengefasst. Das bedeutet, dass die zuvor erstellten Schichten in ihrer Reihenfolge übereinandergelegt werden, um die gewünschte Benutzeroberfläche zu erstellen [17].

7. Rasterize

Der letzte Schritt der Rendering-Pipeline besteht aus der Übersetzung der Anweisungen zum Zeichnen der Benutzeroberfläche, damit diese von der GPU ausgeführt werden können [17].

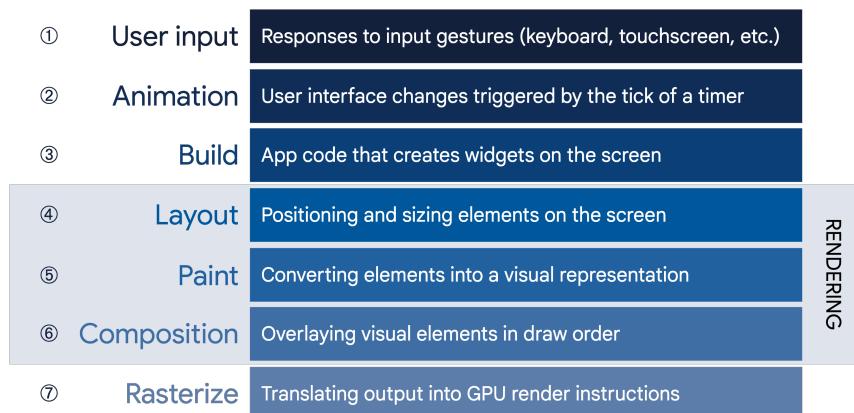


Abbildung 2: Flutter's Rendering Pipeline [17].

2.3.4 Tools

Entwicklungsumgebungen

Zur Erstellung von Flutter Applikationen werden die Entwicklungsumgebung Visual Studio Code, Android Studio, IntelliJ und Emacs empfohlen [17].

Dart DevTools

Die Dart DevTools sind eine Sammlung aus verschiedenen Tools, mit denen die Entwicklung von Flutter Applikationen unterstützt werden soll. Enthalten ist ein Debugging Tool zur Unterstützung bei der Fehlersuche mit Hilfe von typischen Funktionen, wie zum Beispiel Breakpoints und das Inspizieren von Variablen zur Laufzeit. Um das Verhalten der Applikation besser analysieren zu können gibt es eine Logging View, mit der Logs während der Ausführung der Applikation eingesehen werden können. Zur Analyse der Performance bieten die Flutter DevTools, Tools zur Überwachung der Auslastung der Central Processing Unit (CPU) und des Arbeitsspeichers. Außerdem kann der Netzwerk-Traffic mitgelesen und oder aufgezeichnet werden. Das Performance Tool kann zusätzlich dazu die Ressourcenauslastung der einzelnen Threads darstellen. Mit dem Flutter Inspector kann die Benutzeroberfläche der Applikation zur Laufzeit analysiert werden, wodurch die Fehlersuche deutlich vereinfacht wird [17].

Hot Reload

Um die Entwicklungszeit von Flutter Applikationen zu verringern, gibt es das Hot Reload Feature, mit welchem Änderungen am Programmcode in nur sehr kurzer Zeit angezeigt werden können [17]. Unterschieden wird dabei zwischen den folgenden Modi:

- Hot Reload: Bei einem Hot Reload werden die Änderungen des Programmcodes direkt in die Dart virtuelle Maschine geladen, wodurch der `widget tree` mit den neuen Änderungen erneut gebaut wird. Der Zustand der Applikation bleibt dabei erhalten. Der Hot Reload ist für kleine Änderungen der App, wie zum Beispiel das Ändern der Schriftgröße oder ähnliches [17].
- Hot Restart: Bei einem Hot Restart werden auch die Änderungen des Programmcodes direkt in die Dart virtuelle Maschine geladen. Danach wird die Applikation jedoch neu gestartet, was dazu führt, dass der Zustand der App verloren geht. Ein Hot Restart sollte deshalb dann durchgeführt werden, wenn die Änderungen am Programmcode den State der Applikation verändern [17].
- Full Restart: Bei einem Full Restart wird der Dart Code in die native Programmiersprache kompiliert und die Applikation neu gestartet. Dieser Vorgang dauert am längsten. Ein Full Restart muss immer dann ausgeführt werden, wenn Änderungen am nativen Programmcode der Applikation vorgenommen wurden [17].

2.3.5 Interoperabilität

Flutter zu einer nativen Applikation hinzufügen

Flutter kann als Bibliothek oder Modul in ein bestehendes Android oder iOS Projekt eingebunden werden. Dadurch können schrittweise Teile einer nativen Applikation mit Flutter implementiert werden. Es ist möglich sowohl UI-Elemente in Flutter zu implementieren und diese in eine native App einzubinden als auch Teile der Logik einer Applikation mit Flutter zu entwickeln. Dafür wird eine Flutter Instanz während der Laufzeit erstellt, die das Ausführen der Bibliothek beziehungsweise des Moduls übernimmt. Diese Funktion ist auf eine Bibliothek oder ein Modul beschränkt [17].

Plattformspezifischen Programmcode in einer Flutter App ausführen

Zur Kommunikation zwischen Flutter und der Host-Plattform wurden Plattform Channel erstellt, diese ermöglichen es Nachrichten zwischen Flutter und der Host-Plattform auszutauschen. Dadurch können Funktionen, die mit nativem Programmcode erstellt wurden, in Flutter aufgerufen werden. Die Daten dieser Nachrichten werden dabei ähnlich wie auch bei der JavaScript Object Notation (JSON) serialisiert. Dadurch ist es möglich, Parameter über diese Nachrichten auszutauschen. Dieser Vorgang wird asynchron ausgeführt, wodurch sichergestellt werden soll, dass die Benutzeroberfläche der Flutter Applikation weiterhin auf Eingaben reagieren kann [17].

Verwendung von nativen UI-Elementen in Flutter

Native iOS und Android UI-Elemente können direkt in die View Hierarchie der Flutter Applikation integriert werden. Dadurch können diese ohne Einschränkung in ihrer Funktionalität verwendet werden. Lediglich die Performance der Flutter Applikation wird dadurch beeinträchtigt. Android UI-Elemente können alternativ auch als sogenannte virtual displays eingebunden werden. Dafür erstellt Flutter eine Textur des gerenderten UI-Elementes, welche dann in die View Hierarchie aufgenommen wird. Diese Variante ermöglicht eine bessere Performance der Applikation, aber die UI-Elemente sind in ihren Interaktionsmöglichkeiten eingeschränkt [17].

2.4 React Native

2.4.1 Architektur

Das React Native Framework ist aufgebaut in drei verschiedene Schichten - JavaScript, C++ und die mobile Plattform. Die unterste Schicht ist eine Implementierung auf der mobilen Plattform. Sie dient als Schnittstelle zwischen der darunterliegenden Plattform und dem Renderer in der darüber liegenden Schicht. Durch diese werden dem Renderer plattformspezifische Funktionen zu Verfügung gestellt, um die React Native Applikation auf dem Bildschirm des mobilen Gerätes zu zeichnen und weitere Hardwarefunktionen nutzen zu können [24].

Der Renderer Fabric stellt die mittlere Schicht der Architektur von React Native dar. Er ist in der Programmiersprache C++ geschrieben, um diesen unabhängig von der Host Plattform verwenden zu können. Er ist das Bindeglied zwischen der Implementierung auf Plattform-Ebene und der React Implementierung auf der JavaScript-Ebene. Er hört auf Events, die die Plattform Implementierung an ihn sendet, prozessiert diese und leitet sie weiter an die JavaScript-Ebene. Außerdem nimmt der Renderer die render Anweisungen von React entgegen, verarbeitet diese und führt sie mit Hilfe der plattformspezifischen Implementierung aus [24].

Die oberste Schicht von React Native ist in JavaScript implementiert, sie beinhaltet alle Funktionen, die dem Entwickler bei der Entwicklung von React Native Applikationen zu Verfügung stehen. Render Anweisungen werden von hier aus direkt an den, in C++ geschriebenen, Renderer gesendet. React nimmt außerdem Events entgegen, die von dem Renderer gesendet werden [24].

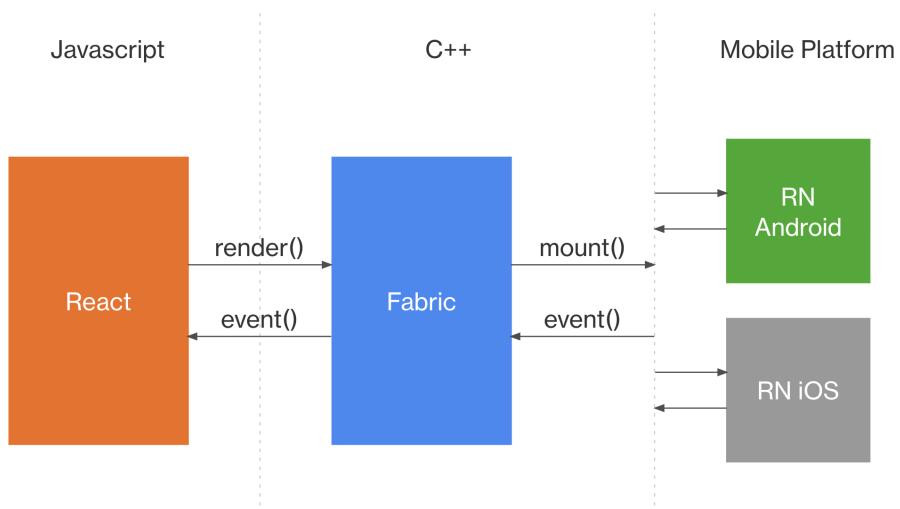


Abbildung 3: Übersicht der Architektur von React Native [24].

2.4.2 Kernkonzepte

React

React ist das JavaScript Framework, dass zur Erstellung von Benutzeroberflächen in React Native verwendet wird. React basiert auf dem Prinzip von Komponenten. Das heißt die UI-Elemente werden in wiederverwendbare Bausteine aufgeteilt. Dieses Konzept soll vor allem die Entwicklung von großen und komplexen Applikationen vereinfachen. Komponenten können mit ihren Eigenschaften statisch verändert werden. Sie besitzen außerdem einen State, was ein privater interner Speicher der Komponenten ist, dessen Daten zur Laufzeit verändert werden können. Es wird empfohlen bei der Entwicklung mit React die JavaScript Syntax Extension (JSX) zu verwenden. Diese ermöglicht es, HTML-ähnlichen Programmcode innerhalb des JavaScript Programm-codes zu verwenden [24].

UI-Elemente

Die Benutzeroberfläche von React Native Applikationen besteht aus React Komponenten, welche einen Teil der Benutzeroberfläche repräsentieren. React Komponenten können entweder durch eine Funktion oder eine Klasse definiert sein. Der zweite wichtige Baustein zur Erstellung von Benutzeroberflächen einer React Native Applikation sind React Elemente. Diese sind einfache Objekte, die beschreiben, was auf dem Bildschirm gezeichnet werden soll. React Elemente sind keine Instanzen von React Komponenten, da sie nur einfache beschreibende Objekte sind. Sie besitzen einen Komponententyp und dazugehörige Attribute. React Native Elemente können weitere React Native Elemente beinhalten, wodurch diese ineinander verschachtelt werden können und somit komplexe Benutzeroberflächen erzeugt werden können. React übernimmt das Erstellen und Verwalten der Instanzen der React Komponenten. Dies ist im nächsten Kapitel zum Rendering Prozess von React Native Applikationen genauer beschrieben [24].

2.4.3 Rendering

Der Rendering Prozess einer React Native Applikation wird von dem Renderer Fabric übernommen. Dieser ist zu einem großen Teil in C++ geschrieben, wodurch er einfach um weitere Host Plattformen erweitert werden kann [24].

Dieser Renderer besitzt eine Rendering-Pipeline, welche den kompletten Rendering Prozess vom Ausführen des React Programm-codes bis zur fertig dargestellten Applikation abbildet. Diese Rendering-Pipeline durchläuft drei Phasen - die Render-Phase, die Commit-Phase und die Mount-Phase [24].

Die Rendering-Pipeline wird durch zwei Fälle ausgelöst, zum einen bei dem initialen Rendern der Applikation und durch Änderungen des UI-State. Der Ablauf der Rendering-Pipeline unterscheidet sich durch ihren Auslöser geringfügig [24].

Wird die Benutzeroberfläche initial von der Rendering-Pipeline erstellt, sieht der Ablauf wie folgt aus:

Im ersten Schritt werden der `react element tree` und der `react shadow tree` erstellt. Dafür startet der Renderer bei dem Wurzelement der React Applikation und teilt dieses React Element so lange rekursiv auf, bis dieses nur noch aus nicht aufteilbaren Elementen besteht. Diese werden dann React Host Komponenten genannt. Der `react shadow tree` wird parallel dazu erstellt. Für jeden Knoten des `react element tree`, wird ein Knoten im `react shadow tree` erstellt. Diese Knoten werden React Shadow Knoten genannt und sind unveränderbar, was bedeutet, dass der `react shadow tree` neu erstellt werden muss, um einen der React Shadow Knoten zu ändern [24].

Als nächstes folgt die Commit-Phase, in der die beiden Operationen Layout Calculation und Tree Promotion asynchron im Hintergrund-Thread ausgeführt werden. Die Layout Calculation berechnet die Position und Größe jedes React Shadow Knotens. Dafür wird die Layout Engine Yoga verwendet. Die zweite Operation dieser Phase, die Tree Promotion, sorgt dafür, dass der neu erstellt Baum vom `new tree` zum `next tree` propagiert wird. Dadurch wird in der nächsten Mount Phase der Tree Diffing Algorithmus auf dem neuen Baum ausgeführt [24].

Die letzte Phase der Rendering Pipeline, die Mount-Phase, besteht aus drei Schritten, dem Tree Diffing, der Tree Promotion und dem View Mounting [24].

Das Tree Diffing ist ein Algorithmus, der die Differenz zwischen dem zuvor gerenderten Baum und dem nächsten Baum berechnet. Das Ergebnis dieses Algorithmus ist eine Liste aus atomaren Operationen, welche nötig sind, um die gewünschte Benutzeroberfläche zu zeichnen. Beispiele für solche Operationen sind `createView` zum Erstellen einer Host View oder `updateView` zum Ändern einer Host View. Host Views sind plattformspezifische UI-Elemente, wie zum Beispiel `android.widget.TextView` [24]. Ein weiterer Teil dieses Tree Diffing Schrittes ist das View Flattening, mit welchem versucht wird die Anzahl der Host Views zu verringern. Dafür werden Elemente zusammengefasst, die selbst keinen Inhalt anzeigen, sondern nur das Layout der Benutzeroberfläche beeinflussen. Der `react element tree` und der `react shadow tree` werden nicht verändert, lediglich die Anzahl der angezeigten Host Views wird verringert [24].

Im Tree Promotion Schritt wird der nächste Baum zum zuvor gerenderten Baum propagiert, dies dient als Vorbereitung für den nächsten Durchlauf der Pipeline. Im letzten Schritt der Rendering-Pipeline, dem View Mounting, werden die Operationen, die durch den Tree Diffing Algorithmus erstellt wurden, synchron ausgeführt, um die gewünschte Benutzeroberfläche zu zeichnen. Diese Operationen werden auf dem UI Thread der Host Plattform ausgeführt [24].

Die Rendering Pipeline unterscheidet sich in einigen wenigen Punkten, wenn diese durch ein Änderung des State der Applikation ausgeführt wird [24].

In der Rendering-Phase werden alle React Elemente des vorherigen `react element tree` in den neuen `react element tree` geklont, die von den Änderungen des State betroffen sind. Alle React Elemente, die nicht von den Änderungen betroffen sind, werden mit dem vorherigen Baum geteilt. Dasselbe gilt auch für den `react shadow tree` [24].

In der Commit-Phase wird die `Layout Calculation` nur auf den Elementen ausgeführt, die im vorherigen Schritt geklont wurden, also sich durch die Änderung des State geändert haben. Hierbei kann es dazu kommen, dass für weitere Elemente die Layoutinformationen neu berechnet werden müssen. Die `Tree Promotion` wird genau, wie bei dem initialen Rendering ausgeführt, genauso wie die gesamte Mount-Phase [24].

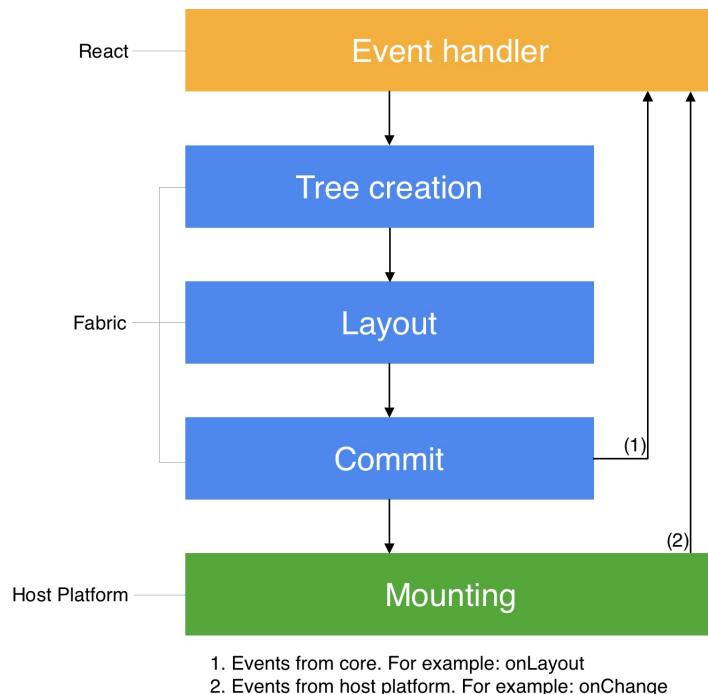


Abbildung 4: Rendering Prozess von React Native Applikationen [24].

2.4.4 Tools

Entwicklungsumgebung

Die empfohlene integrierte Entwicklungsumgebung, um React Native Applikationen zu erstellen ist Visual Studio Code. Für diese steht zusätzlich eine Erweiterung zu Verfügung, die bestimmte Prozesse, wie zum Beispiel das Erstellen einer React Native Applikation, vereinfachen soll [24].

React DevTools

Die React DevTools sind eine Sammlung an Werkzeugen, welche die Entwicklung von React Native Applikationen unterstützen sollen. Die React Developer Tools beinhalten den Inspektor, den Performance Monitor und die LogBox. Aufgerufen werden die React Native Developer Tools durch eine Tastenkombination [24].

Mit dem Inspektor können alle UI-Elemente untersucht werden, welche auf dem Gerät angezeigt werden. Dafür werden Layout Informationen zu den UI-Elementen mit Hilfe eines Overlays angezeigt [24].

Der Performance Monitor hilft dem Entwickler Performance Probleme in React Native Applikationen zu erkennen. Außerdem können die einzelnen Threads der React Native App analysiert werden und der systrace eingesehen und aufgezeichnet werden [24].

Die LogBox ist ein Tool für React Native, welches Fehler, Warnungen und Logs direkt in der laufenden App anzeigt. Zunächst werden diese als einfache Benachrichtigungen in der App angezeigt. Bei Fehlern und Warnungen kann auf diese Benachrichtigung getippt werden, um genauere Informationen dazu in einem separaten Screen anzuzeigen. Auf diesem separaten Screen wird die genaue Beschreibung des Fehlers oder der Warnung angezeigt. Der ursächliche Programmcode wird in einem Ausschnitt dargestellt. Zusätzlich dazu gibt es den Call Stack, welcher die zuvor aufgerufenen Funktionen auf-listet. Bei Fehlern wird auch der Component Stack, eine hierarchische Darstellung der betroffenen UI-Elemente, angezeigt [24].

Fast Refresh

Fast Refresh sorgt, während dem Debuggen dafür, dass Änderungen des Programm-codes innerhalb ein bis zwei Sekunden auf dem Gerät angezeigt werden. Dies soll die Entwicklung von React Native Applikationen beschleunigen. Der lokale State der Applikation wird bei einem Fast Refresh behalten. Es werden nur die Komponenten neu gerendert, die von den Änderungen betroffen sind. Alle Module die zu einer geänderten Komponente gehören werden ebenfalls neu geladen. Sollten Module geändert werden, die außerhalb des `rendering tree` verwendet werden, muss ein full reload durchgeführt werden, was zur Folge hat, dass die gesamte App neu gestartet wird und der State der Applikation verloren geht [24].

Expo

Expo ist ein Werkzeug, welches einen Schnellstart in die Entwicklung von React Native Applikationen ermöglichen soll. Es soll den Entwickler dabei unterstützen React Native Applikationen zu erstellen und im späteren Verlauf zu veröffentlichen. Expo wird vor allem für Einsteiger empfohlen, die nur einfache React Native Applikationen erstellen wollen. Expo schränkt jedoch die Möglichkeiten von React Native ein, wodurch das Tool für aufwändige Anwendungen nicht zu empfehlen ist. Beispielsweise können Applikationen mit Expo keinen plattformspezifischen Programmcode integrieren und keine JavaScript Pakete einbinden, die eine Verlinkung benötigen [24].

2.4.5 Interoperabilität

React Native zu einem existierenden nativen Projekt hinzufügen

React Native bietet die Möglichkeit React Native Komponenten zu einem iOS oder Android Projekt hinzuzufügen. Zunächst wird eine `package.json` Datei dem Projekt hinzugefügt, um im nächsten Schritt die Pakete und weitere Abhängigkeiten für React und React Native zu installieren. Diese Abhängigkeiten müssen auch dem nativen Projekt hinzugefügt werden, im Android Projekt mit Gradle und im iOS Projekt mit CocoaPods. Nun kann eine React Komponente erstellt werden, die später in einem Container, `ReactRootView` für Android und `RCTRootView` für iOS angezeigt wird [24].

Native UI-Elemente in React Native verwenden

React Native verwendet zum Erstellen von Benutzeroberflächen ausschließlich native UI-Elemente. Ein großer Teil der nativen UI-Elemente von Android und iOS wurden durch das Entwicklerteam von React Native bereits implementiert und in dem Software Development Kit (SDK) und weiteren Bibliotheken veröffentlicht. Weitere native UI-Elemente sind durch Bibliotheken von Drittanbietern erreichbar. Sollten native UI-Elemente noch nicht durch diese Quellen verfügbar sein, besteht die Möglichkeit eigene native UI-Elemente in React Native einzubinden [24].

Für die Implementierung eines nativen Android UI-Elementes wird eine Java oder Kotlin Klasse und eine Objective-C oder Swift Klasse für iOS UI-Elemente erstellt. In dieser nativen Klasse wird das native UI-Element initialisiert und Funktionen zum Setzen der Eigenschaften erstellt. Danach wird ein JavaScript Modul erstellt, dass als Schnittstelle zwischen der nativen Klasse und React Native fungiert. Es können Events erstellt werden, um diese zum React Native Programmcode weiterleiten zu können [24].

Kommunikation zwischen React Native und nativem Programmcode

Zur Kommunikation zwischen React Native und plattformspezifischen Implementierungen werden je nach Richtung der Kommunikation unterschiedliche Konzepte verwendet. Zur Kommunikation von React Native zum plattformspezifischen Programmcode werden Daten über Properties ausgetauscht und zum Aufrufen von Funktionen werden Events verwendet. Zur Kommunikation ausgehend von dem plattformspezifischen Programmcode zu React Native wird ein Modul in Java oder Kotlin beziehungsweise in Objektive-C oder Swift geschrieben. React Native erstellt für jedes dieser Module eine JavaScript Bridge, wodurch diese Module im JavaScript Programmcode zur Verfügung stehen und dessen Funktionen und Variablen aufgerufen werden können [24].

2.5 Ionic

Ionic ist ein Cross-Plattform Framework, mit welchem hybride Applikationen erstellt werden können. Ionic Applikationen benötigen eine Laufzeitumgebung, um auf den mobilen Betriebssystemen iOS und Android ausgeführt werden zu können. Außerdem kann aus einem Ionic Projekt eine Progressive Web App erstellt werden [7].

Die Laufzeitumgebung für Android und iOS kann gewählt werden, zu Verfügung stehen die beiden Projekte Capacitor und Cordova. Cordova ist das Open-Source Projekt zu Adobe PhoneGap. Cordova wurde bereits 2009 veröffentlicht und ist somit deutlich älter als das Capacitor Projekt, dessen erste Version im Jahr 2018 veröffentlicht wurde. Capacitor nutzt modernere APIs, die zum Zeitpunkt der Veröffentlichung von Cordova noch nicht zu Verfügung standen [7].

Capacitor wird von Ionic für die Entwicklung von Ionic Applikationen empfohlen. Aus diesem Grund wurde für diese Arbeit Capacitor als Laufzeitumgebung für alle Ionic Applikationen verwendet und wird auch im Folgenden weiter behandelt [23].

2.5.1 Architektur

Die Architektur von Ionic Applikationen, die mit der Laufzeitumgebung Capacitor entwickelt wurden, lässt sich in fünf wesentliche Bausteine aufteilen - die Ionic Applikation, die Webview, die Bridge, die Hardware API und das Betriebssystem [7].

App

In dieser Schicht der Architektur befindet sich die Ionic Applikation. Hier ist der Aufbau und die Funktionalität der Applikation mit Hilfe von Webtechnologien beschrieben. Typischerweise ist die Applikation mit einem der drei Frontend Webframeworks React, Angular oder vue.js entwickelt [7].

Webview

Die Ionic Applikation wird in einer Webview ausgeführt. Webviews sind voll funktionsfähige Browser, welcher in eine iOS oder Android Applikation eingebettet werden können. Für iOS kommt die `WKWebView` und für Android die `android.webkit.WebView` zum Einsatz. Webviews besitzen außerdem eine Schnittstelle, um mit der nativen Applikation kommunizieren zu können. Dadurch können Nachrichten zwischen dem JavaScript Programmcode und der nativen Applikation ausgetauscht werden [7].

Bridge

Die Bridge Schicht dient zur Kommunikation zwischen Ionic und der Hostplattform. Durch diese Schicht ist es für Ionic möglich plattformspezifische Funktionen zu verwenden. Diese Funktionen werden durch sogenannte Capacitor Plugins implementiert. Die

Bridge Schicht verwaltet dabei die Kommunikation zwischen Ionic und den Capacitor Plugins [7].

Capacitor Plugins sind Module, welche eine oder mehrere plattformspezifische Funktionen durch eine JavaScript Schnittstelle verfügbar machen. Capacitor Plugins werden fast ausschließlich nativ entwickelt. Das heißt für Android wird eine Kotlin/Java Klasse und für iOS ein Objective-C/Swift Klasse erstellt. In diesen Klassen wird die plattformspezifische Funktion implementiert. Damit diese auch von Ionic verwendet werden kann, wird die dazugehörige JavaScript API von Capacitor automatisch erstellt. Es besteht trotzdem die Möglichkeit JavaScript Programmcode zu einem Capacitor Plugin hinzuzufügen, um die JavaScript API zu erweitern [7].

Hardware API und das Betriebssystem

Die Hardware API und das Betriebssystem der Hostplattform bilden die unterste Schicht. Sie werden nur über die Bridge von der Ionic Applikation angesprochen [7].

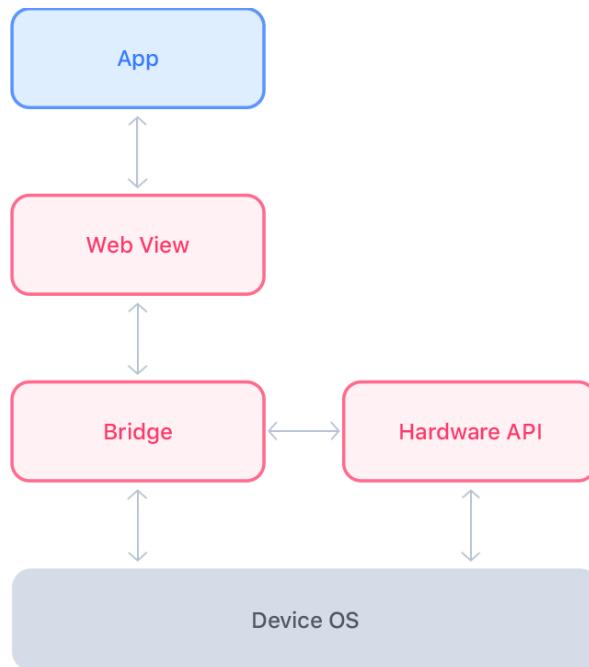


Abbildung 5: Architektur von Ionic Applikationen [7]

2.5.2 Kernkonzepte

Entwicklungstechnologien

Ionic bietet die Möglichkeit zwischen mehreren Technologien zur Entwicklung von Ionic Applikationen zu entscheiden. Empfohlen wird mit einem der drei Frontend Webframeworks React, Angular oder vue.js zu arbeiten [7].

React ist eines der populärsten Webframeworks, es zeichnet sich durch seine hohe Performance aus und eignet sich sowohl für kleine als auch große Applikationen. React wurde bereits genauer im Kapitel 2.4.2 beschrieben [24].

In den ersten drei Versionen von Ionic wurde nur das Webframework Angular unterstützt [7]. Angular verwendet Typescript, eine Skriptsprache, welche JavaScript um Konzepte wie zum Beispiel Typisierung oder Vererbung erweitert. Die Benutzeroberfläche von Angular setzt sich aus Komponenten zusammen, ähnlich wie auch bei React. Templates beschreiben dabei, wie diese Komponenten gerendert werden sollen. Angular zeichnet sich vor allem durch seine hohe Skalierbarkeit aus [13].

Seit Ionic 4 ist es möglich eine Ionic Applikation mit dem Webframework vue.js zu entwickeln. Wie auch React und Angular ist vue.js ein reaktives komponentenbasiertes Webframework. Die Benutzeroberfläche von vue.js wird mit Single-File Komponenten erstellt. Diese bestehen aus drei Teilen - der in JavaScript geschriebenen Logik, das Template in HTML und die styles in CSS. Vue.js bietet einen schnellen und einfachen Einstieg in die Entwicklung von Webanwendungen und ist vor allem deshalb bei unerfahrenen Entwicklern beliebt [32].

Alternativ dazu kann Ionic auch durch ein einfaches Skript in eine Webanwendung integriert werden, was die Möglichkeit bietet, eine Ionic Applikation mit einem anderen Webframework zu entwickeln oder aber auch ganz auf ein Webframework zu verzichten und die Ionic Applikation mit reinem HTML, CSS und JavaScript Programcode zu entwickeln [7].

UI-Elemente

UI-Elemente in Ionic werden Komponenten genannt. Diese werden mit Webtechnologien erstellt. Das bedeutet, dass keine nativen UI-Elemente in Ionic Applikationen verwendet werden. Die Ionic SDK beinhaltet eine Bibliothek, welche dem Entwickler eine Sammlung an Komponenten zur Erstellung von Ionic Applikationen bereitstellt. Auch Drittanbieter-Bibliotheken können verwendet werden, um weitere Komponenten zu einer Ionic Applikation hinzuzufügen. Außerdem können mit HTML, CSS und JavaScript weitere ganz individuelle UI-Elemente erstellt und in Ionic Applikationen verwendet werden [7].

2.5.3 Rendering

Ionic Applikationen werden von der nativen Webview gerendert. Die Webview, die für iOS Applikationen verwendet wird, ist `WKWebView`, diese basiert auf der Webkit Browser Engine [2]. Für Android Applikationen wird die `android.webkit.WebView` verwendet, welche auf der Browser Engine Blink basiert. Blink ist ein Fork von Webkit

und unterscheidet sich von Webkit nur in wenigen Punkten [14]. Der Rendering Prozess der Webviews kann grundsätzlich mit den folgenden Schritten beschrieben werden [7].

1. Parsing

Im ersten Schritt des Rendering Prozesses wird aus dem HTML Programmcode das Document Object Model (DOM) erstellt. Dieses ist eine Repräsentation der Benutzeroberfläche. Außerdem werden in diesem Schritt externe Inhalte, wie zum Beispiel Bilder, geladen [7].

2. Style calculation

Im zweiten Schritt des Rendering Prozesses werden die CSS Dateien geladen. Aus diesen werden die Styling Attribute für jedes DOM Element ermittelt [7].

3. Layout

Im Layout Schritt wird das zuvor erstellte DOM durchlaufen, um den `layout tree` zu erstellen. Der `layout tree` besteht aus allen DOM-Elementen, die später auf dem Bildschirm angezeigt werden sollen. DOM-Element, die auf dem Bildschirm nicht sichtbar sind, werden bei der Erstellung des `layout tree` nicht berücksichtigt. Für jedes Element des `layout tree` werden die Position und die Grenzen der Elementgröße berechnet [7].

4. Paint

Im nächsten Schritt wird der `layout tree` durchlaufen, um aus diesem die `paint records` zu erstellen. `paint records` beinhalten Information darüber, welche Elemente in welcher Reihenfolge gezeichnet werden sollen [7].

5. Compositing

Der letzte Schritt des Rendering Prozesses wird compositing genannt. In diesem Schritt wird der Inhalt der Webseite in mehrere Ebenen aufgeteilt. Diese Ebenen können später verschoben werden, wodurch die Webanwendung bei Animationen oder dem Scrollen nicht immer wieder komplett neu gerendert werden muss. Dafür wird zunächst der `layer tree` erstellt. In diesem `layer tree` werden die einzelnen Ebenen definiert und die Elemente aus dem `layout tree` werden den Ebenen zugeordnet. Danach wird jede Ebene gerastert und in Kacheln aufgeteilt. Diese Kacheln werden ebenfalls gerastert und an die GPU weitergegeben, damit diese auf dem Bildschirm gezeichnet werden können [7].

2.5.4 Tools

Entwicklungsumgebung

Ionic empfiehlt die integrierte Entwicklungsumgebung Visual Studio Code mit einer Ionic Erweiterung. Dies soll den Prozess der Entwicklung von Ionic Applikation verein-

fachen. Die Ionic Erweiterung bietet die meisten Funktionalitäten des Ionic Command Line Interface (CLI) in einer Benutzeroberfläche [7].

Ionic CLI

Das Ionic CLI ist ein Kommandozeilenprogramm, welches benötigt wird, um zum Beispiel eine Ionic Applikation zu erstellen oder auch zu bauen. Mit dessen Hilfe können die Plugins und Bibliotheken des Ionic Projektes verwaltet werden. Außerdem bietet es eine Integration von Appflow, einer cloudbasierten Plattform zur kontinuierlichen Integration und Auslieferung [7].

Debuggen

Zum Debuggen von Ionic Applikationen gibt es mehrere Werkzeuge. Zum einen können sowohl die iOS, als auch die Android Applikation einer Ionic Anwendung mit Hilfe von Visual Studio Code und der Ionic Erweiterung debuggt werden. Dazu wird die Ionic Applikation in Google Chrome ausgeführt. Ionic Android Applikationen können alternativ auch mit den Google Chrome Entwicklerwerkzeugen debuggt werden. Das gleiche gilt auch für Ionic iOS Applikationen und den Safari Entwicklerwerkzeugen [7].

Live Reload

Auch Ionic bietet die Möglichkeit während dem Debuggen, Änderungen am Programmcode innerhalb kürzester Zeit auf dem Bildschirm anzuzeigen. Während der Entwicklung wird die Ionic Applikation auf einem lokalen Server auf dem Computer, auf dem entwickelt wird, ausgeführt. Dadurch ist die Applikation nicht direkt auf dem Smartphone installiert. Dies bedeutet, dass die Applikation bei einem Live Reload auch nicht neu installiert werden muss [7].

2.5.5 Interoperabilität

Ionic zu einer existierenden nativen Applikation hinzufügen

Capacitor kann zu jeder Webview in einer nativen Applikation hinzugefügt werden. Dadurch ist es möglich Ionic in eine bestehende native Applikation einzubinden. Teile einer nativen Applikation können somit durch Webviews ersetzt werden, um die Applikation nach und nach mit Ionic zu entwickeln, ohne dabei die bestehende Applikation komplett durch eine neue ersetzen zu müssen [7].

Integration von plattformspezifischem Programmcode

Ionic bietet die Möglichkeit Capacitor Plugins einzubinden. Diese sind einfache Java/-Kotlin beziehungsweise Objective-C/Swift Klassen, die von einer generischen Capacitor

Plugin Klasse erben. Dadurch ist es möglich, nativen Programmcode in einer Ionic Applikation auszuführen. Ein Capacitor Plugin das hinzugefügt wurde, muss sowohl in dem Android/iOS Projekt registriert werden, als auch in dem Ionic Projekt. Das Ionic Team und auch viele Drittanbieter haben bereits sehr viele dieser Capacitor Plugins erstellt und veröffentlicht [7].

Verwendung von nativen UI-Elementen in Ionic

Ionic bietet keine Möglichkeit native UI-Elemente zu integrieren, da die gesamte Benutzeroberfläche der Applikation in einer Webview dargestellt wird. Dennoch besteht die Möglichkeit native UI-Elemente in Ionic nachzubauen, sodass diese die gleiche Funktionalität besitzen und auch gleich aussehen. Die einzige Möglichkeit native UI-Elemente in Verbindung mit Ionic zu verwenden ist, durch Hinzufügen einer oder mehrere Webviews zu einer nativen Applikationen [7].

3 Use-Cases

Im folgenden Kapitel werden sechs praxisnahe Use-Cases vorgestellt, um die Performance und Funktionalität der drei Cross-Plattform Frameworks React Native, Ionic und Flutter zu vergleichen. Jeder Use-Case wird einzeln betrachtet und es wird untersucht, wie gut jedes Framework die Anforderungen umsetzen kann. Die Use-Cases sind so ausgewählt, dass ein breites Spektrum der wichtigsten Funktionen mobiler Anwendungen untersucht wird.

Der erste Use-Case ist eine Anwendung, die eine Liste mit vielen Datensätzen anzeigt. Hier wird untersucht, wie die Frameworks mit großen Datenmengen umgehen und wie flüssig das Scrollen in der Liste ist.

Im zweiten Anwendungsfall geht es um das UI-Element Card. Dieses soll in verschachtelten Scrollviews sowohl als Liste als auch als Grid dargestellt werden. Dabei liegt der Fokus darauf, wie gut die Frameworks mit der Erstellung komplexer Layouts umgehen können.

Der dritte Use-Case ist die Implementierung eines Canvas UI-Elements, auf dem gezeichnet werden soll. Hier wird untersucht, wie gut die Frameworks mit der Implementierung dieses UI-Elements umgehen können und wie gut Animationen mit den drei Frameworks implementiert werden können.

Der vierte Anwendungsfall beschäftigt sich mit der Integration einer öffentlichen API basierend auf dem Representational State Transfer (REST) Protokoll. Hier wird untersucht, wie gut die Frameworks mit der Integration von Daten aus externen Quellen umgehen können und wie gut die Daten dargestellt werden können.

Der fünfte Use-Case ist die Integration einer SQLite-Datenbank und der SharedPreferences bzw. NSUserDefaults. Hier wird untersucht, wie gut die Frameworks mit der lokalen Speicherung von Daten umgehen können.

Im sechsten Use-Case soll eine Applikation mit ausgewählten UI-Elementen erstellt werden, die nach Vorgaben gestylt werden. Hier wird untersucht, wie gut die Frameworks mit der Umsetzung von benutzerdefinierten Designs umgehen können.

3.1 Listen

Mobile Anwendungen müssen oft mit großen Datenmengen umgehen können, daher beschäftigt sich der erste Use-Case mit dieser Problematik. Ziel ist es, eine große Anzahl von Datensätzen in einer Liste anzuzeigen. Diese Daten sollen über eine Suchleiste gefiltert werden können.

Die Testdaten wurden mit dem JavaScript-Tool Faker.js [8] erstellt und als JSON- oder Dart-Dateien in das Projekt der Applikation eingefügt. Dadurch müssen die angezeigten Daten zur Laufzeit nicht aus einer Datenbank oder einer anderen Quelle geladen werden. Dies hat zur Folge, dass die Anwendungen auf ihre Performance hin untersucht werden können, ohne dass die Ergebnisse durch unterschiedliche Ladezeiten der Daten verfälscht werden.

Die Benutzeroberfläche besteht aus einer Suchleiste am oberen Bildschirmrand und einer scrollbaren Liste direkt darunter. In dieser Liste werden die Mock-Daten in Listenelementen dargestellt. Diese Listenelemente bestehen aus einem Titel, einem Untertitel, einer Beschreibung und einem Chip.

Durch Eingabe eines Suchbegriffs in die Suchleiste können die Daten nach Titel, Untertitel und Beschreibung gefiltert werden. Wird auf eines der Listenelemente getippt, soll ein Dialog erscheinen, der den Titel und den Index des Listenelements enthält.

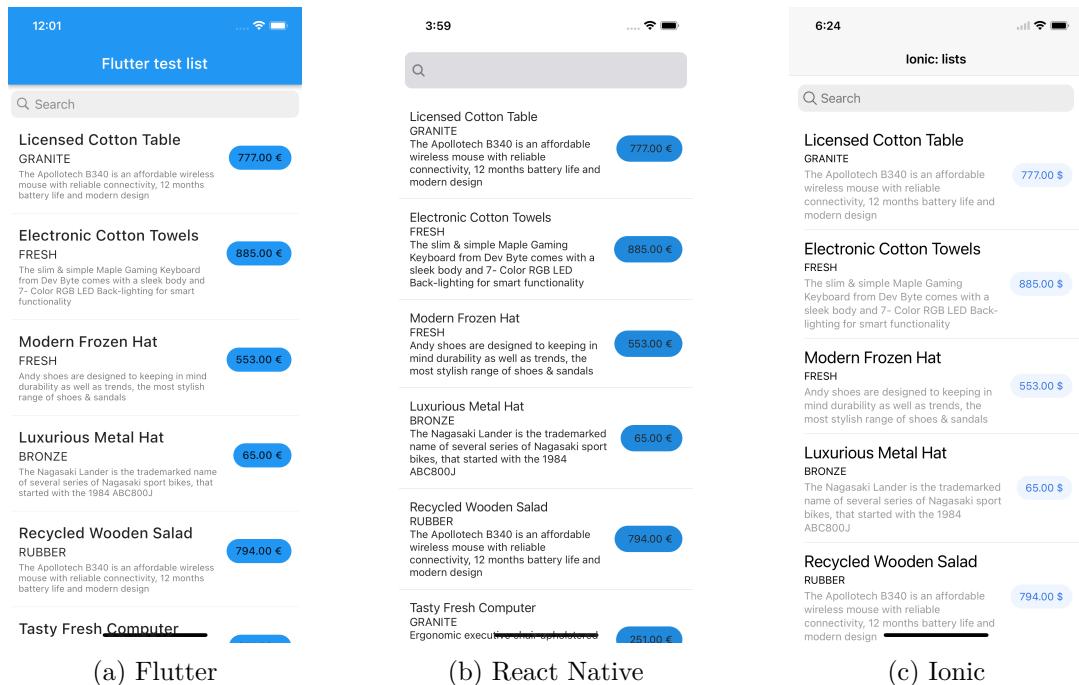


Abbildung 6: Screenshots des ersten Use-Cases.

3.1.1 Implementierung mit Flutter

Für die Implementierung dieses Use-Case mussten keine externen Bibliotheken eingebunden werden, lediglich vorimplementierte Widgets wurden verwendet, die Teil des Flutter SDK sind. Es wurden drei Klassen erstellt: `ListItemData` eine Datenklasse, die die anzuzeigenden Daten enthält, das Widget `ListItem`, in dem das Listenelement definiert ist und das Widget `MainPage`, in dem die Benutzeroberfläche bestehend aus der Suchleiste und der Liste definiert ist.

Zur Erstellung von horizontalen oder vertikalen Listen in Flutter wird das Widget `ListView` verwendet. Die Listenelemente werden dynamisch gerendert, so dass bei der Initialisierung nicht alle Listenelemente auf einmal gerendert werden. Die Klasse `ListView` besitzt mehrere Konstruktoren um eine Liste zu erstellen. Eine Liste kann beispielsweise durch ein Array von Widgets oder auch durch die Anzahl der Listenelemente in Verbindung mit einer Funktion zur dynamischen Erzeugung der Listenelemente definiert werden [17].

Alternativ zu einer `ListView` kann auch eine `SingleChildScrollView` in Verbindung mit einem `Column` Widget verwendet werden. In diesem Fall würden alle Listenelemente bei der Initialisierung gerendert werden, was bei sehr großen Listen zu Performance-Problemen führen kann [17].

Aus diesem Grund wurde für diesen Anwendungsfall das `ListView` Widget verwendet. Dieser `ListView` wird die statische Liste bestehend aus den Mock-Daten und eine Funktion zur dynamischen Erzeugung der Listenelemente übergeben.

3.1.2 Implementierung mit Ionic

Das `VirtualScroll` UI-Element wird in Ionic-Anwendungen verwendet, um Listen zu erstellen, die ihre Listenelemente dynamisch nachladen. Dieses UI-Element wird vom Ionic SDK nicht mehr unterstützt und es wird empfohlen, auf eine Lösung des gewählten JavaScript-Frameworks zurückzugreifen. Für React wird die `react-virtuoso` Bibliothek, für Angular das `Angular component developer kit` und für das vue.js Framework `vue-virtual-scroller` empfohlen [7].

Alternativ kann ein `IonContent`-Element verwendet werden, bei dem die Eigenschaft für das Scrollen gesetzt ist. Auch hier werden die Listenelemente bei der Initialisierung und nicht dynamisch zur Laufzeit geladen [7].

Um eine möglichst gute Performance zu erreichen, wurde die Liste für diesen Use-Case mit dem UI-Element `Virtuoso` aus der Bibliothek `react-virtuoso` realisiert [20].

3.1.3 Implementierung mit React Native

In der Implementierung mit dem React Native Framework musste eine externe Bibliothek verwendet werden, um weitere UI-Elemente verwenden zu können. Eine Suchleiste und ein Listenelement sind im React Native Framework nicht vorimplementiert und wurden daher über die externe Bibliothek `react-native-elements` hinzugefügt [28].

Für die Erstellung von Listen in React Native Anwendungen steht das UI-Element `FlatList` zur Verfügung. Damit können horizontale oder vertikale Listen erstellt werden, bei denen die Kindelemente dynamisch nachgeladen werden, sobald sie auf dem Bildschirm angezeigt werden. Kindelemente, die nicht mehr auf dem Bildschirm sichtbar sind, werden zerstört. Das `FlatList` Element bietet zusätzlich die Möglichkeit einen Header oder Footer einzubinden, einen Separator zwischen den Listenelementen anzuzeigen oder eine Pull-To-Refresh Funktion zu aktivieren [24].

Alternativ zu diesem UI-Element kann eine statische Liste auch mit dem UI-Element `ScrollView` erstellt werden. Die Performance ist jedoch deutlich schlechter [24].

Aus diesem Grund wurde für diesen Anwendungsfall das UI-Element `FlatList` gewählt. Das `FlatList` Element wird mit einem Array initialisiert, welches die Daten der Kindelemente enthält.

3.1.4 Performance-Analyse während dem Scrollen

Für die Performance-Analyse werden alle drei Implementierungen während des Scrollens durch die Liste untersucht, da hier viele UI-Elemente gerendert werden müssen. Dazu wird für alle Implementierungen eine produktionsfähige Applikation gebaut, die auf physischen Geräten installiert wird. Zum Einsatz kommen ein iPhone 13 mit der Version iOS 16.3.1 und ein OnePlus 7 mit Android 12. Untersucht werden soll die Bildwiederholrate in Frames Per Second (FPS) beim Scrollen durch die Liste. Jeder Testlauf dauert 30 Sekunden. In diesen 30 Sekunden wird einmal pro Sekunde manuell gescrollt. Diese Tests werden auf jedem Gerät für jedes Framework dreimal durchgeführt.

Die Framerate wird für die Flutter-Anwendung mit den Flutter DevTools aufgezeichnet [17]. Für das Ionic Framework werden die Chrome bzw. Safari Developer Tools zur Aufzeichnung der Framerate verwendet [7]. React Native Applikationen werden mit den Xcode Instruments für iOS und mit einem Systrace und dem Perfetto Tool für Android untersucht [24].

Ergebnisse

Die Ergebnisse aus dem Performance Tests sind als Durchschnittswerte in der Tabelle 1 zu finden. Die Werte zeigen die durchschnittliche Anzahl an Bildern, die pro Sekunde

während des Scrollens dargestellt wurden, sowie die minimalen Werte, die in Ausnahmefällen erreicht wurden. Die Ergebnisse zeigen, dass React Native auf beiden Plattformen die höchste durchschnittliche Bildwiederholrate erzielte, gefolgt von Flutter. Ionic hatte die niedrigste durchschnittliche Bildwiederholrate auf beiden Plattformen.

Auf Android erreichte React Native die höchste minimale Bildwiederholrate, während Ionic auf beiden Plattformen die niedrigste minimale Bildwiederholrate hatte. Auf iOS erreichte Flutter die höchste minimale Bildwiederholrate, während Ionic erneut die niedrigste hatte.

	Flutter		Ionic		React Native	
Plattform	Android	iOS	Android	iOS	Android	iOS
Min. FPS	16,89	23,36	5,06	14,01	41,66	46,67
Durch. FPS	59,33	59,57	50,73	32,32	56,70	55,00

Tabelle 1: Ergebnisse aus dem Performance Test

3.2 Cards

Im zweiten Use-Case wird eine Anwendung implementiert, die das UI-Element Card enthält. Eine Card ist ein UI-Element, das Inhalte in einem Box-ähnlichen Format darstellt, das an eine physische Karte erinnert. Cards enthalten in der Regel weitere UI-Elemente zur Darstellung von Text, Bildern oder ähnlichem. Ein Card UI-Element ist typischerweise ein rechteckiges Element mit abgerundeten Ecken und einem Schatten, um die Abgrenzung zum Rest der Benutzeroberfläche visuell zu verdeutlichen.

Die Benutzeroberfläche, die für diesen Anwendungsfall mit den drei Cross-Plattform Frameworks implementiert werden soll, besteht aus einer großen scrollbaren Liste. Diese Liste ist in drei Bereiche unterteilt, die jeweils durch ein Textelement getrennt sind. Der erste Bereich enthält Cards in einem Grid, wobei immer zwei Cards nebeneinander angezeigt werden. Darunter befindet sich eine horizontale ScrollView, die dieselben Cards enthält. Diese horizontale ScrollView ist auf eine Höhe von 300 Pixel begrenzt. Die Cards in dieser ScrollView passen sich dieser Größe an. Der letzte Bereich ist wieder ein Grid, das genauso zwei Cards nebeneinander anzeigt.

Die Cards, die für diesen Use-Case verwendet werden, expandieren ihre Größe in horizontaler und vertikaler Richtung. Das Seitenverhältnis ist auf zwei zu drei beschränkt. Die Cards enthalten ein Bild, das die maximal verfügbare Größe einnimmt. Darunter befindet sich ein einzeiliger Titel, eine auf zwei Zeilen beschränkte Beschreibung und ein Text, in dem der Preis angezeigt werden soll.

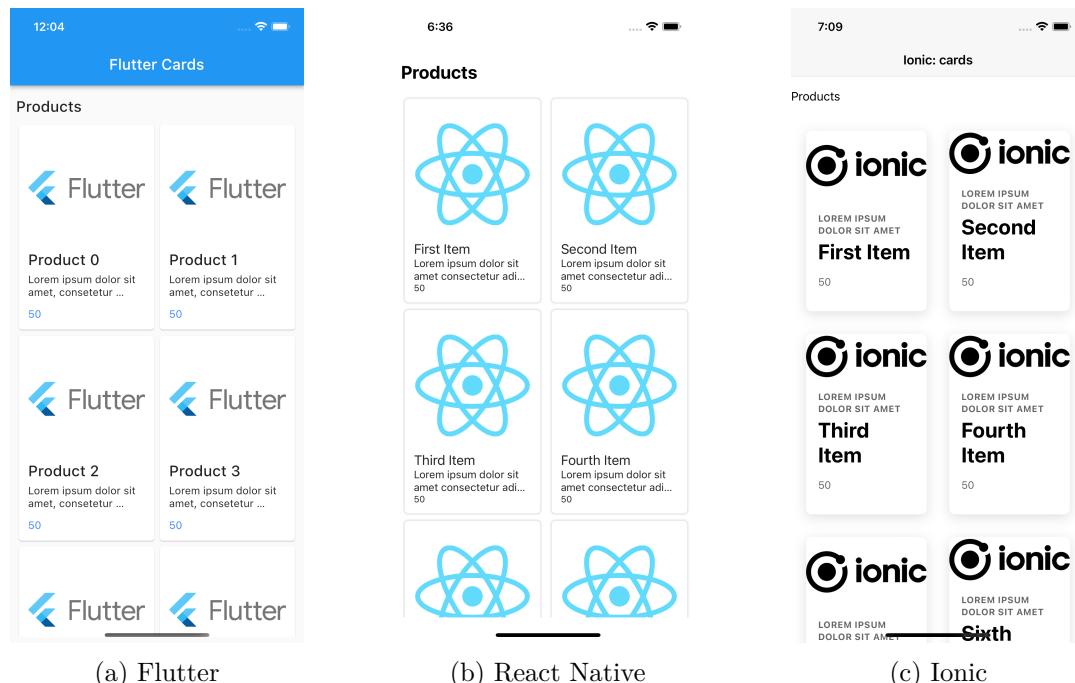


Abbildung 7: Screenshots des zweiten Use-Case.

3.2.1 Implementierung mit Flutter

Flutter bietet eine einfache Lösung zur Erstellung von Scrollviews mit komplexen Anforderungen. Das `CustomScrollView` Widget bietet die Möglichkeit, mehrere verschiedene scrollbare Abschnitte mit verschiedenen Inhalten zusammenzufügen. Diese scrollbaren Bereiche werden Sliver genannt. Flutter bietet mehrere Implementierungen dieser Sliver an, z.B. `SliverList` für einen Bereich bestehend aus einer Liste, `SliverGrid` um ein Grid der `CustomScrollView` hinzuzufügen oder `SliverToBoxAdapter` mit dem ein beliebiges Widget einer `CustomScrollView` hinzugefügt werden kann [17].

In diesem Anwendungsfall wurde die Benutzeroberfläche mit einer `CustomScrollView` realisiert. Für die trennenden Abschnittsüberschriften wird ein `SliverToBoxAdapter` verwendet, in dem ein Text Widget gerendert wird. Für das Grid mit zwei Spalten bestehend aus Cards wird ein `SliverGrid` verwendet. Diesem kann die Anzahl der Spalten und das Seitenverhältnis der Kindelemente, in diesem Fall das `CustomCard` Widget, als Eigenschaften übergeben werden. Damit wird die Größe der Kindelemente automatisch eingestellt. Für die horizontale Scrollview wird wiederrum ein `SliverToBoxAdapter` verwendet, in dem sich eine `ListView` befindet. Diese `ListView` rendert die `CustomCard` Widgets. Um einen UI-Fehler zu vermeiden muss die Größe der `ListView` statisch gesetzt werden.

3.2.2 Implementierung mit Ionic

Die Scrollview für diesen Use-Case basiert auf dem `IonContent` UI-Element, bei dem die Eigenschaft zum Scrollen gesetzt ist. Einfache Textelemente können verwendet werden, um die Überschriften für die drei Abschnitte anzuzeigen.

Für die Erstellung flexibler Grids bietet das Ionic Framework eine Implementierung, die auf dem CSS Flexbox-System basiert. Dabei handelt es sich um ein Layoutmodell zur Darstellung und Ausrichtung von Kindelementen in flexiblen Containern. Dadurch kann der Platz innerhalb eines UI-Elements optimal genutzt werden. Die Ionic Implementierung besteht aus den Komponenten `IonGrid`, zum Erstellen eines Grids, `IonRow`, welches eine Zeile in einem Grid repräsentiert und `IonCol`, welches eine Spalte innerhalb einer Zeile darstellt. Mit diesen Komponenten konnten die Grids zur Anzeige der Cards innerhalb der Scrollview sehr einfach implementiert werden [7].

Die Swiper.js Bibliothek stellt das UI-Element `Swiper` zu Verfügung, mit welchem horizontale Scrollviews erstellt werden können. Die scrollbaren Elemente sind `SwiperSlide`-Elemente. In einer `SwiperSlide` können dann beliebige UI-Elemente dargestellt werden. Standardmäßig füllt eine `SwiperSlide` die gesamte Breite aus und springt in seine Position. Diese Einstellungen können jedoch geändert werden, um den Anforderungen des Anwendungsfallen zu entsprechen [30].

Für diesen Use-Case wurde die `Swiper`-Komponente so konfiguriert, dass sie auf eine Höhe von 300 Pixel beschränkt ist, die `SwiperSlide`-Elemente auf eine Breite von 200 Pixel begrenzt sind und mehrere `SwiperSlide`-Komponenten direkt nebeneinander angezeigt werden können, ohne dass diese in ihre Position springen. Dies ermöglicht ein freies Scrollen durch die Cards.

3.2.3 Implementierung mit React Native

Für diesen Use-Case wurde das `ScrollView`-Element verwendet, welches mit CSS-Eigenschaften für das Flexbox-System so konfiguriert wurde, dass die Kindelement wenn möglich nebeneinander gerendert werden.

Es wurden drei React-Komponenten erstellt, die in dieser `ScrollView` angezeigt werden. Die `CustomCard`-Komponente, die das Card UI-Element darstellt. Die `SectionTitle` Komponente, um einen Titel zu erstellen, der die Abschnitte unterteilen soll. Und die `HorizontalScrollView`-Komponente, welche die horizontale ScrollView zur Anzeige der Cards implementiert.

Das `CustomCard`-Element ist auf die Hälfte der verfügbaren Breite beschränkt, damit das Flexbox-System immer zwei Cards nebeneinander anzeigt. Die Breite der beiden Komponenten `SectionTitle` und `HorizontalScrollView` wurde auf die volle Breite gesetzt, um dieses Verhalten zu verhindern.

Zum Rendern der Elemente der `ScrollView` werden die Daten der Kindelemente in einem Array an die `ScrollView` übergeben. Mit der `renderItem` Funktion wird dann abhängig von den Daten ein Kindelement erzeugt. Für diesen Use-Case wurde ein Array mit statischen Daten übergeben. Diese Daten können drei verschiedene Objekte sein, für die Komponente `SectionTitle`, `CustomCard` und `HorizontalScrollView`. Je nach Typ des Datensatzes wird die entsprechende Komponente von der Funktion `renderItem` übergeben. Die Anzahl, die Reihenfolge und der Typ der Komponenten kann somit über das Datenarray bestimmt werden.

3.3 Canvas

Der dritte Use-Case, der im Rahmen dieser Arbeit untersucht wurde, befasste sich mit der Implementierung eines Canvas UI-Elements.

Ein Canvas ist ein UI-Element, in dem durch den Aufruf von Funktionen gezeichnet wird. Das Zeichnen innerhalb eines Canvas erfolgt mit Hilfe von primitiven Zeichenoperationen, wie zum Beispiel das Zeichnen von Rechtecken, Kreisen oder Zeichen. Dadurch ist das UI-Element Canvas sehr flexibel in seiner Darstellung, da durch die vielseitigen Zeichenoperationen jede beliebige Grafik gezeichnet werden kann. Daher ist ein Canvas vor allem für die Implementierung spezieller UI-Elemente interessant, die nicht mit den vorgefertigten UI-Elementen des Frameworks realisiert werden können. Die Benutzeroberfläche für diesen Use-Case besteht nur aus dem Canvas-Element, das den gesamten Bildschirm ausfüllen soll. Im Canvas soll ein Kreis mit einem Text darunter gezeichnet werden. Dieser Kreis soll eine Größe von 100 Pixeln haben und genau in der Mitte des Bildschirms gezeichnet werden. Eine Animation soll den Kreis von unten nach oben füllen. Die Animation soll fünf Sekunden dauern und in einer Schleife wiederholt werden. Zur Umsetzung der Animationen sollen zunächst die Möglichkeiten zur Erstellung von Animationen mit den Frameworks untersucht werden.

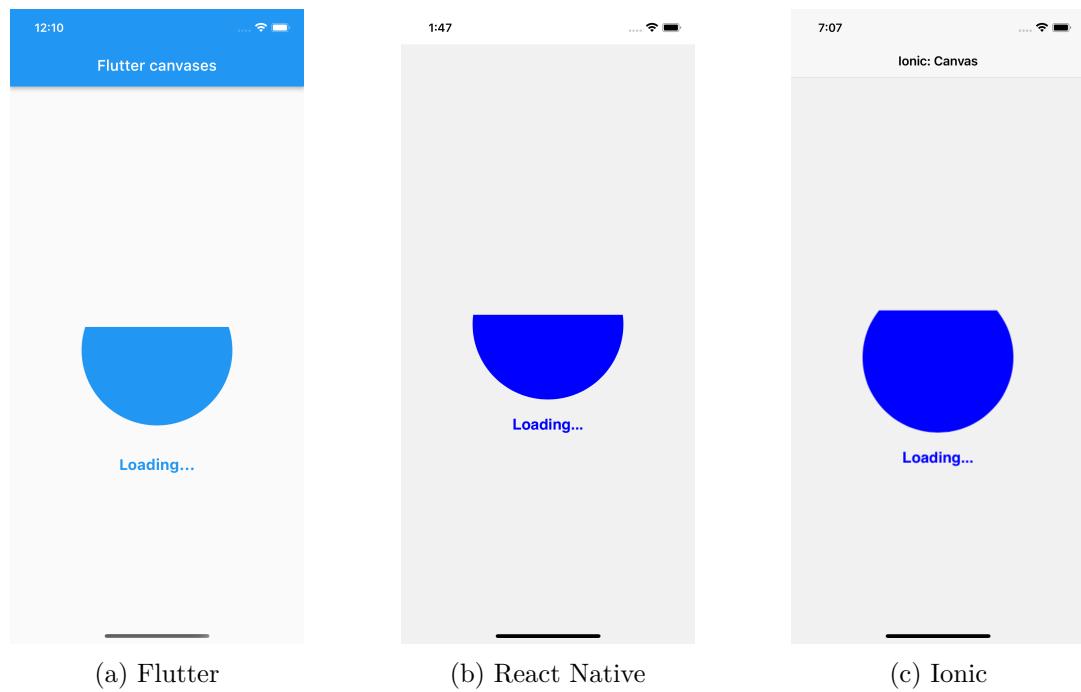


Abbildung 8: Screenshots des dritten Use-Case.

3.3.1 Implementierung mit Flutter

Das Canvas UI Element wird in dem Flutter SDK als `CustomPaint` bezeichnet. Um in diesem Element zeichnen zu können, wird eine Klasse erstellt, die von der Klasse `CustomPainter` erbt. In dieser Klasse können die beiden Funktionen `paint` und `shouldRepaint` überschrieben werden. Außerdem muss die Klasse alle notwendigen Variablen enthalten, die den State repräsentieren [17].

In der Funktion `paint` werden die Zeichenoperationen ausgeführt, um die gewünschte Grafik zu zeichnen. In der Funktion `shouldRepaint` wird definiert, bei welchen Änderungen des States das `CustomPaint` Element neu gezeichnet werden soll. In diesem Use-Case ist der Fortschritt der Animation die einzige Variable des State [17].

Das Flutter SDK enthält eine Bibliothek mit Funktionen zur Erstellung von Animationen. Aus diesem Grund musste keine zusätzliche Bibliothek installiert werden. Die wichtigste Klasse dieser Animations-Bibliothek ist die `Animation` Klasse, die die Basisklasse für alle Animationen darstellt. Mit ihr ist es möglich, Animationen einer Fließkommazahl zwischen zwei Grenzen in einer vorgegebenen Zeit auszuführen [17].

Ein `Tween` erweitert die `Animation` um die Interpolation der Animationswerte auf andere Datentypen, zum Beispiel kann mit einem `Tween` eine Variable vom Typ `Color` animiert werden. Flutter bietet auch vorgefertigte Widgets, deren Attribute animiert werden können [17].

Animationen werden über einen `AnimationController` gesteuert, der eine oder mehrere Animationen parallel und/oder seriell ausführen kann. Standardmäßig werden die Werte aller Animationen 60 mal pro Sekunde aktualisiert, die Animationsgeschwindigkeit kann jedoch angepasst werden [17].

Die Animation für diesen Use-Case wird von einem `AnimationController` gesteuert. Dieser wird beim Start der Anwendung initialisiert. In diesem Fall animiert der `AnimationController` eine Fließkommazahl linear zwischen den festgelegten Grenzen null und eins innerhalb der angegebenen Animationsdauer von fünf Sekunden.

3.3.2 Implementierung mit Ionic

Das Ionic Framework bietet keine eigene Implementierung eines Canvas UI-Elements, daher wurde auf das HTML Canvas Element zurückgegriffen. Mit einer Referenz auf dieses Canvas-Element können Zeichenoperationen ausgeführt werden, um auf der Fläche des Canvas zu zeichnen. Es erwies sich als schwierig, die Größe dieses Canvas-Elements so festzulegen, dass es die gesamte Benutzeroberfläche ausfüllt. Zunächst musste die Größe über die CSS-Eigenschaften festgelegt werden. Außerdem musste die

Größe des Canvas über eine Referenz auf das Canvas-Element festgelegt werden. Da die Größe des Canvas-Elements nur in Pixelwerten angegeben werden kann, musste die Größe zunächst in Pixel berechnet werden.

Das Ionic Framework beinhaltet eine Bibliothek zur Erstellung von einfachen und komplexeren Animationen, die CSS Eigenschaften der Komponenten animieren. Einfache Animationen können durch das Angeben von der zu animierenden Eigenschaft mit den dazugehörigen Grenzen, der Animationsdauer und der Anzahl an Iterationen erstellt werden. Komplexere Animationen können durch eine Keyframe Animation erstellt werden. Animationen in Ionic Applikationen können gruppiert werden, um parallel ausgeführt zu werden oder aber auch verkettet zu werden, um diese hintereinander auszuführen [7].

Alternativ kann auch die JavaScript Funktion `Window.requestAnimationFrame` zur Animation von Variablen verwendet werden. Dieser Funktion wird eine weitere Funktion übergeben, in der die Animation ausgeführt wird. Dadurch wird diese übergebene Funktion jedes mal aufgerufen, bevor die Benutzeroberfläche neu gezeichnet wird [7].

Für diesen Use-Case wurde die Funktion `Window.requestAnimationFrame` zur Animation des Kreises verwendet, da eine Variable der Canvas Komponente animiert werden soll und keine CSS Eigenschaft. Dafür wurde eine Funktion erstellt, die in Abhängigkeit des Fortschrittes den zu animierenden Kreis zeichnet.

3.3.3 Implementierung mit React Native

Das React Native Framework bietet kein UI-Element mit der Funktionalität eines Canvas. Dadurch musste eine drittanbieter Bibliothek verwendet werden, um ein solches UI-Element in der React Native Applikation zu verwenden. Diese externe Bibliothek `react-native-canvas` basiert auf dem HTML Canvas Element. Dieses wird in einer Webview gerendert. Mit einer API können alle Funktionen des Canvas HTML Elements verwendet werden. Auch bei der Implementierung der React Native Applikation war das Setzen der Größe des Canvas umständlich. Auch hier musste die Größe zusätzlich über eine Referenz in Pixeln gesetzt werden, damit die gesamte Benutzeroberfläche ausgefüllt wird [18].

React Native bietet gleich zwei Animationssysteme in dem SDK an. Das Layout Animation System zum Erstellen von globalen Animationen, die auf mehreren UI-Elementen ausgeführt werden, zum Beispiel zum Animieren von Übergängen zwischen zwei Screens. Und das Animated System zur Erstellung von lokalen Animationen auf einzelnen UI-

Elementen, welches relevant für diesen Use-Case ist. Es gibt drei Typen von Animationen. Abklingende Animationen, die mit einer Startgeschwindigkeit definiert werden, die zur Laufzeit abnimmt. Zeitlich festgelegte Animationen, bei denen die Dauer der Animation festgelegt ist. Und Animationen, die durch eine Federdynamik definiert sind [24].

Diese Animationen können verzögert, parallel mit oder ohne eine Verzögerung oder in einer Sequenz ausgeführt werden. Die generierten Animationswerte können interpoliert werden und auch mit Werten anderer Animationen durch mathematische Operatoren verknüpft werden [24].

Für diesen Use-Case wurde eine zeitlich festgelegte Animation verwendet, welche linear eine Fließkommazahl im Wertebereich von null bis einhundert animiert. Durch einen Listener auf dieser Animation, wird die Funktion zum Zeichnen des animierten Kreises mit dem neuen Fortschritt aufgerufen.

3.4 REST API

In diesem Use-Case soll die Implementierung einer REST API untersucht werden. Ziel ist es, eine Anwendung zu erstellen, die Daten aus einer öffentlichen REST API bezieht.

Die Daten für diesen Use-Case stammen von der öffentlichen REST API FakestoreAPI [25]. Diese ermöglicht den Zugriff auf Daten eines fiktiven Online-Shops. Sie bietet Endpunkte für Produkte, Produktkategorien, Benutzer und Warenkörbe [25].

- **Produkte:** Das Produkt-Objekt besteht aus einer Produkt-ID, dem Titel, der Beschreibung, dem Preis, der Kategorie und einem Uniform Resource Locator (URL) für das Bild. Über den zugehörigen Endpunkt können alle oder einzelne Produkte abgefragt, neue Produkte angelegt oder bestehende Produkte geändert und gelöscht werden [25].
- **Produktkategorie:** Jedes Produkt ist einer Kategorie zugeordnet. Diese Produktkategorien können über einen eigenen Endpunkt heruntergeladen werden. Außerdem können alle Produkte einer Kategorie ebenfalls über diesen Endpunkt abgerufen werden [25].
- **Benutzer:** Das Benutzer-Objekt besteht aus einer ID, der E-Mail-Adresse, dem Benutzernamen, dem Passwort, dem Namen, der Adresse und der Telefonnummer. Die Benutzer-Objekte können über den Endpunkt `/user` abgefragt werden. Das Anlegen, Ändern und Löschen von Benutzern erfolgt ebenfalls über diesen Endpunkt [25].
- **Warenkörbe:** Der Endpunkt `/carts` stellt Warenkorbdaten zu Verfügung. Ein Warenkorb-Objekt enthält eine ID, eine Benutzer-ID, das Datum und ein Array aus Produkt-IDs und der entsprechenden Anzahl von Produkten [25].

Die Benutzeroberfläche ist durch eine Navigationsleiste in drei Seiten unterteilt. Jeweils eine Seite für die Produkte, die Benutzer und die Warenkörbe.

Auf der Produktseite werden die Produkte in einer Liste angezeigt. Das Listenelement eines Produktes ist eine Card. Diese Card enthält den Titel, die Beschreibung, die Kategorie, den Preis und das Bild. Die Bilder der Produkte sind als URL angegeben und werden entsprechend aus dem Internet nachgeladen. Über einen Floating Action Button (FAB) kann ein Dialog geöffnet werden, um ein neues Produkt anzulegen. Dieser Dialog enthält jeweils ein Textfeld für den Titel, die Beschreibung und den Preis. Zur Auswahl der Kategorie werden alle Produktkategorien vom Endpunkt heruntergeladen und in einem Drop-Down-Menü angezeigt, um einer dieser Kategorien auszuwählen zu können. Durch langes Tippen auf einen Listeneintrag kann dieser gelöscht werden. Das Ergebnis des Anlegens und Löschens von Produkten wird in einer Snackbar angezeigt.

Im zweiten Tab für die Benutzer wird ebenfalls eine Liste angezeigt, in der alle Benutzer aufgelistet sind. Das zugehörige Listenelement stellt alle Informationen des Benutzers in einer tabellarischen Übersicht dar.

Im dritten Tab werden die Warenkörbe gleichermaßen in einer Liste angezeigt. Das Listenelement für die Warenkörbe enthält das Datum und die Anzahl der enthaltenen Produkte.

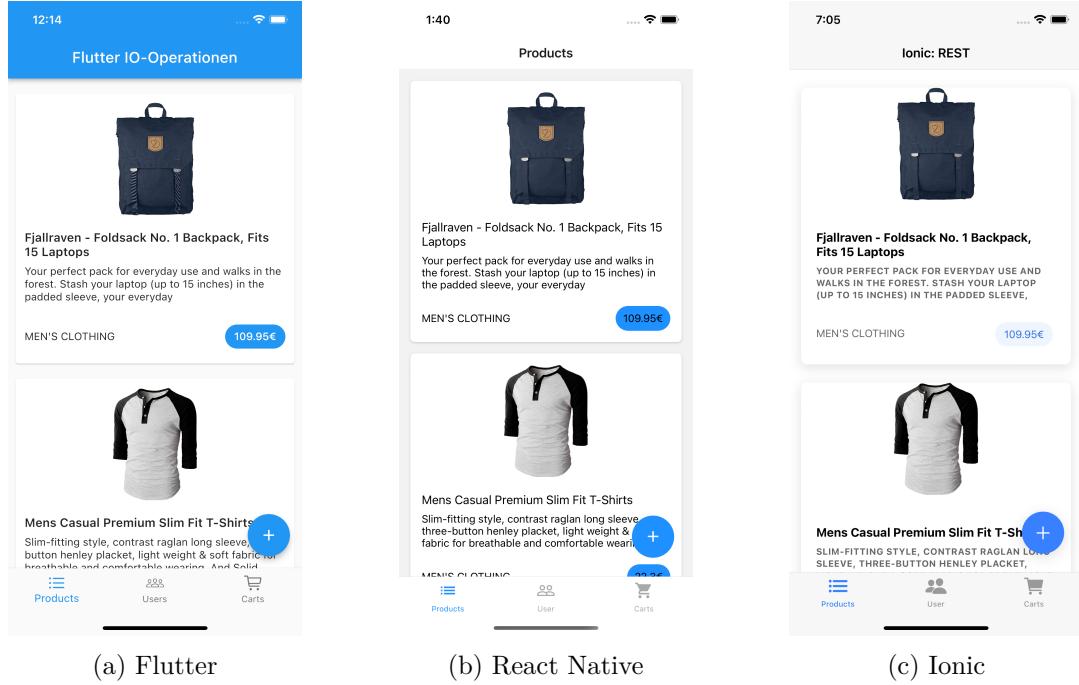


Abbildung 9: Screenshots des vierten Use-Case.

3.4.1 Implementierung mit Flutter

Die Integration der REST API erfolgt über die Bibliothek `http` von Flutter. Diese muss als Abhängigkeit zum Projekt hinzugefügt werden. Diese Bibliothek enthält einen HTTP-Client, mit dem Methoden des Hypertext Transfer Protocol (HTTP), wie zum Beispiel `GET` oder `POST` aufgerufen werden können. Diese Funktionen geben einen `Future` zurück, der die HTTP-Antwort enthält. Diese HTTP-Antwort enthält den Statuscode, die Header-Informationen und die Daten [6].

Ein `Future` ist ein Objekt, das einen zukünftigen Wert oder einen Fehler repräsentiert. Asynchrone Funktionen können einen `Future` zurückgeben, damit der Programmcode weiter ausgeführt werden kann, ohne auf das Ergebnis der Funktion warten zu müssen. Mit dem Schlüsselwort `await` oder der Funktion `Future.onComplete` kann auf das Ergebnis gewartet werden [17].

Die Daten der FakestoreAPI werden im JSON-Format übertragen [25]. Das bedeutet, dass die Daten, nachdem sie in der Anwendung angekommen sind, zunächst deserialisiert werden müssen. Und wenn Daten an die FakestoreAPI gesendet werden, müssen diese zuerst in das JSON Format konvertiert werden. Hierfür bietet Flutter zwei Möglichkeiten: manuelle Serialisierung durch die Erstellung von Hilfsfunktionen oder automatische Serialisierung mit Hilfe einer externen Bibliothek [17].

Die manuelle Serialisierung von Daten in einer Flutter-Anwendung beinhaltet die Konvertierung von Objekten in JSON-Strings und umgekehrt mit Hilfe von Hilfsfunktionen. Um ein Dart-Objekt in das JSON-Format zu konvertieren, wird zunächst eine Map aus Key-Value-Paaren aus dem Dart-Objekt erstellt. Mit der Funktion `jsonEncode` aus der `dart:convert` Bibliothek kann im nächsten Schritt aus dieser Map ein String mit den Daten im JSON Format erzeugt werden. Die Deserialisierung funktioniert genau umgekehrt. Die JSON-Daten als String werden mit der Hilfsfunktion `jsonDecode` in eine Map umgewandelt. Anschließend muss eine Funktion erstellt werden, die aus diesen Key-Value Paaren ein Dart Objekt erzeugt [17].

Bei der automatischen Serialisierung werden die Funktionen zur Konvertierung von Dart-Objekten in einen JSON-String und anders herum automatisch generiert. Flutter empfiehlt hierfür die Bibliothek `json_serializable` [17].

Für diesen Anwendungsfall wurden drei Services erstellt, das sind einfache Dart-Dateien, die Funktionen enthalten, um mit der FakestoreAPI zu kommunizieren. Jeweils ein Service für den Endpunkt Produkte und Produktkategorien, für den Endpunkt Warenkorb und für den Endpunkt Benutzer. Die Funktionen der Services rufen die entsprechenden HTTP-Methoden auf dem Endpunkt auf, validieren und konvertieren die Daten und geben diese als `Future` zurück. Die Funktionen der Services werden dann von den entsprechenden Widgets aufgerufen.

Ein wichtiges UI-Element zur Anzeige asynchroner Daten ist der `FutureBuilder`. Ein `FutureBuilder` ist in der Lage, den Fortschritt einer asynchronen Operation zu überwachen und die Benutzeroberfläche entsprechend anzupassen. Der `FutureBuilder` besteht aus einem `Future` und einer Funktion, die das Ergebnis des Futures in ein Widget umwandelt, das auf der Benutzeroberfläche angezeigt wird. In dieser Funktion kann auch festgelegt werden, welches Widget angezeigt werden soll, wenn der `Future` noch nicht abgeschlossen ist – typischerweise ist dies eine Ladeanzeige [17].

3.4.2 Implementierung mit Ionic

Die Integration einer REST-API in eine Ionic-Anwendung erfolgt mit Hilfe des Capacitor-Plugins `CapacitorHTTP` [5]. Dieses stellt Funktionen zum Senden von HTTP-Requests zur Verfügung. Für jede HTTP-Methode gibt es eine eigene Funktion. Diesen Funk-

tionen kann ein Objekt vom Typ `HttpOptions` übergeben werden, in dem Parameter für den HTTP-Request definiert werden. Zum Beispiel die URL oder Header-Informationen. Die Funktionen geben einen `Promise` zurück, der die HTTP-Antwort enthält [5].

Ein `Promise` gibt das Ergebnis zu einem späteren Zeitpunkt zurück, sobald es verfügbar ist. Dieses Ergebnis kann der angeforderte Wert sein, aber auch ein Fehler, wenn die asynchrone Funktion fehlschlägt. Sobald die HTTP-Antwort eingetroffen ist, kann der Body ausgelesen und die übergebenen Daten deserialisiert werden, um sie als TypeScript-Objekt weiter zu verarbeiten [5].

Zur Deserialisierung von JSON-Daten wird die Funktion `JSON.parse` und zur Konvertierung eines Objekts in einen JSON-String die Funktion `JSON.stringify` verwendet. Die Objekte müssen dann nur noch validiert werden, ob alle Werte vorhanden sind und der richtige Datentyp vorliegt. Diese Validierung kann manuell programmiert oder automatisch von einer Bibliothek durchgeführt werden. Ein Beispiel für eine solche Bibliothek ist die `JSONValidator`-Bibliothek [7].

3.4.3 Implementierung mit React Native

Für die Kommunikation mit der FakestoreAPI über das REST-Protokoll wurde die Fetch API verwendet. Diese ist Teil der Web API, so dass keine zusätzliche Bibliothek zu diesem Projekt hinzugefügt werden musste [24].

Diese Fetch API stellt die Funktion `fetch` zur Verfügung, über die HTTP-Requests gesendet werden können. Dieser Funktion werden Konfigurationsparameter wie die URL der Ressource, Header-Informationen und die zu sendenden Daten übergeben. Wie beim Ionic Framework wird ein Promise zurückgegeben, der eine Antwort auf den HTTP-Request enthält. Sobald die HTTP-Antwort empfangen wurde, müssen die übermittelten Daten, die im JSON-Format vorliegen, deserialisiert werden [24].

Die Serialisierung und Deserialisierung funktioniert genau wie beim Ionic Framework, da in beiden Anwendungen das React Frontend Framework verwendet wurde (siehe Kapitel 3.4.2) [24].

3.5 Persistenz

Dieser Use-Case beschäftigt sich mit der Implementierung einer SQLite Datenbank und der `SharedPreferences` auf der Android Plattform und der `NSUserDefaults` auf der iOS Plattform. In dieser App sollen Todos verwaltet werden können und eigene App-Einstellungen zur Verfügung stehen.

Die Todos sollen in der SQLite Datenbank gespeichert werden, damit sie auch nach einem Neustart der App noch vorhanden sind. Ein Todo-Datensatz enthält eine eindeutige ID, einen Titel, ein Fälligkeitsdatum und einen Boolean, der angibt, ob das Todo bearbeitet wurde oder nicht. Es sollte möglich sein, alle Todos aus der Datenbank auszulesen, bestehende Todos zu ändern und zu löschen und neue Todos zur Datenbank hinzuzufügen.

Die Einstellungen der App sollen unter Android in den `SharedPreferences` bzw. unter iOS in den `NSUserDefaults` gespeichert werden. In den App-Einstellungen sollen der Benutzername, die Optik der App und eine weitere Einstellung gespeichert werden.

Die Benutzeroberfläche der Anwendung ist in zwei Screens aufgeteilt. Ein Screen für die Todos und ein Screen für die Einstellungen. Die Navigation zwischen den Screens erfolgt über eine `BottomNavigationBar`. Die Todos werden in einer Liste dargestellt. Das Listenelement eines Todos besteht aus dem Titel, dem formatierten Fälligkeitsdatum und einer `CheckBox`. Durch Tippen auf die `Checkbox` kann markiert werden, ob ein Todo bereits bearbeitet wurde oder nicht. Durch Tippen auf den FAB am rechten unteren Bildschirmrand öffnet sich ein Dialog. In diesem Dialog kann ein neues Todo angelegt werden. Um ein neues Todo anzulegen, muss der Titel in ein Textfeld eingegeben und das Fälligkeitsdatum in einem Popup ausgewählt werden.

Der Einstellungs-Screen besteht aus mehreren Textfeldern und Schaltflächen, mit denen die Einstellungen der Anwendung angepasst werden können.

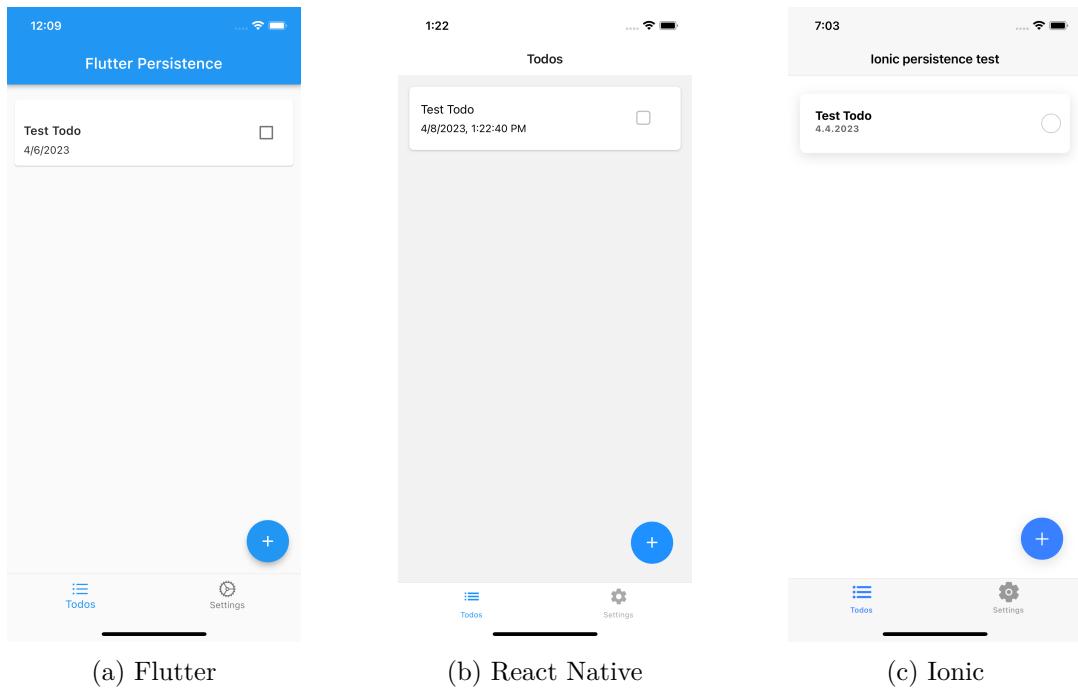


Abbildung 10: Screenshots des fünften Use-Case.

3.5.1 Implementierung mit Flutter

Für die Implementierung einer SQLite Datenbank in eine Flutter Anwendung wird das Paket `sqflite` benötigt [10]. Damit kann eine SQLite Datenbank erstellt und verwaltet werden [10].

In diesem Anwendungsfall wurde eine Helferklasse `DatabaseHelper` erstellt. Diese ist als Singleton gelöst, damit sie in der gesamten Anwendung verwendet werden kann. Um die Datenbank zu erstellen, muss der Funktion `openDatabase` der Pfad zum Speicherort und der Name der Datenbank übergeben werden. Der Rückgabewert ist ein Future, welcher das `SQLiteDatabase` Objekt enthält. Im zweiten Schritt der Initialisierung wird, falls noch nicht geschehen, die Tabelle zum Speichern der Todos erstellt. Dies geschieht mit der `execSQL` Funktion des `SQLiteDatabase` Objekts [17].

Für die Verwaltung einer `SQLiteDatabase` bietet das Paket mehrere Funktionen auf dem `SQLiteDatabase` Objekt an. Mit der Funktion `query` können Daten aus der Datenbank abgefragt werden. Mit der Funktion `insert` werden Objekte gespeichert, mit der Funktion `update` werden Objekte geändert und mit der Funktion `delete` werden Objekte aus der Datenbank entfernt. Die Objekte, die diesen Funktionen übergeben werden, müssen vom Typ `Map<String, Object>` sein. Dieser Typ enthält Key-Value-Paare. Das bedeutet, dass jedes Datenmodell, das in der Datenbank gespeichert werden soll, eine Funktionen `fromMap` und `toMap` implementieren sollte, um aus einem Objekt eine Map zu erzeugen und umgekehrt [17].

Für die Implementierung der `SharedPreferences` und der `NSUserDefaults` wird das Paket `shared_preferences` [9] verwendet. Dieses Paket enthält die `SharedPreferences` Klasse, die als Singleton-Objekt gelöst ist. Diese Klasse implementiert für die Datentypen `int`, `double`, `bool`, `String` und `List<String>` jeweils eine Funktion zum Setzen des Wertes und zum Lesen des Wertes. Der Schlüssel müssen als String übergeben werden [17].

3.5.2 Implementierung mit Ionic

Um die `SharedPreferences` und `NSUserDefaults` in einer Ionic Anwendung verwenden zu können, wird das Capacitor Plugin `@capacitor/preferences` [26] benötigt. Dieses wird in das Projekt installiert und kann dann als statische Klasse verwendet werden. Die wichtigsten Funktionen dieses Plugins sind `Preferences.get` und `Preferences.set`. Mit diesen beiden Funktionen kann der Wert zu einem Schlüssel aus dem Speicher gelesen bzw. in den Speicher geschrieben werden. Das Plugin unterstützt jedoch nur das Speichern von Strings, daher müssen alle anderen Datentypen in einen String umgewandelt werden [26].

Zur Implementierung einer SQLite Datenbank wird das `@capacitor-community/sqlite` [4] Capacitor Plugin verwendet. Dieses Plugin wird dem Projekt hinzugefügt und muss beim Start der Anwendung initialisiert werden. Zusätzlich muss in der Applikation ein React Hook erstellt werden, um auf die Funktionen des Plugins zugreifen zu können. Um eine Datenbank zu verwenden, muss ein Objekt der Klasse `SQLiteDBConnection` erstellt werden. Über diese Verbindung kann auf die SQLite Datenbank zugegriffen werden. Wichtig dabei ist, dass es nur eine Verbindung pro Datenbank geben darf, weshalb diese in der gesamten Applikation zur Verfügung stehen sollte [4].

Aus diesem Grund wurde für diese Implementierung auch eine `DatabaseHelper` Helferklasse erstellt. Diese Klasse dient als Schnittstelle zwischen der Datenbank und der Benutzeroberfläche. Die ausgetauschten Daten werden direkt in dieser Klasse validiert, um unerwartete Fehler zu vermeiden.

Abfragen können über die Funktion `execute` auf der `SQLiteDBConnection` ausgeführt werden. Dieser Funktion wird ein String übergeben, der das Structured Query Language (SQL)-Statement und ein Array mit Daten enthält. Um diesen SQL-Befehl mit den Daten zu füllen, werden Platzhalter in dem String mit einem `?` definiert. Diese werden vom Plugin durch die Daten des übergebenen Arrays ausgetauscht [4].

Die Bibliothek `ionic-long-press` [19] wurde für die Erstellung der Benutzeroberfläche verwendet, da die Ionic-Komponenten kein Ereignis für langes Tippen implementieren.

3.5.3 Implementierung mit React Native

Zur Erstellung der React Native Applikation für diesen Anwendungsfall wurde das Paket `react-native-sqlite-storage` [29] verwendet. Dieses ermöglicht es eine SQLite Datenbank zu erstellen, die dann in der React Native Anwendung verwendet werden kann.

Zur Erstellung einer SQLite Datenbank wird die Funktion `openDatabase` aus der Bibliothek `react-native-sqlite-storage` verwendet. Diese gibt ein Objekt vom Typ `SQLiteDatabase` zurück. Mit diesem Objekt werden zunächst die benötigten Tabellen erstellt, sofern dies noch nicht geschehen ist. Im Folgenden können mit diesem Objekt SQL Befehle über die Funktion `executeSQL` ausgeführt werden. Die SQL Befehle werden als String an die Funktion übergeben. Die Daten für die SQL Befehle, um zum Beispiel ein Todo zu erzeugen, werden direkt in den String interpoliert. Der Rückgabewert der Funktion `executeSQL` ist ein Promise, der ein Array aus `ResultSet`-Objekten enthält. Diese Objekte enthalten die gewünschten Daten aus dem Ergebnis des SQL-Befehls [29].

Für den Zugriff auf die `SharedPreferences` und die `NSUserDefaults` gibt es kein Paket, das eine API für beide Key-Value Speicherlösungen bietet. Aus diesem Grund wurde für diese Implementation das Paket `AsyncStorage` [27] verwendet. Dieses bietet einen eigenen Key-Value Speicher für React Native Anwendungen [24].

Über die statische Klasse `AsyncStorage` können die beiden asynchronen Funktionen `AsyncStorage.setItem` und `AsyncStorage.getItem` aufgerufen werden. Mit diesen kann ein String-Wert zu einem Schlüssel in den Speicher geschrieben bzw. aus dem Speicher gelesen werden. Unterstützt wird nur der Datentyp String, weshalb alle Daten von und zu einem String konvertiert werden müssen [24].

3.6 Custom Design

Im letzten Use-Case dieser Studie sollen die Frameworks auf ihre Anpassbarkeit hin überprüft werden. Dafür wurden zunächst Anforderungen an drei UI-Elemente definiert. Diese UI-Elemente sind ein Button, ein Bild und ein Textfeld. Wichtig bei der Untersuchung ist der Test nach Unterschieden zwischen den beiden Betriebssystemen Android und iOS.

Button-Styles

- Form: Der Button soll eine rechteckige Form mit abgerundeten Ecken, bei einem Radius von acht Pixel besitzen. Außerdem soll der Button einen Schatten besitzen, der einer Erhöhung von vier Pixel entspricht.
- Größe: Die Größe des Button soll auf eine minimale Breite von 180 Pixel und eine minimale Höhe von 56 Pixel angepasst sein. Außerdem soll der Button einen Padding haben mit den Werten horizontal 16 Pixel und vertikal 12 Pixel.
- Hintergrund: Der Hintergrund soll einen blauen Farbverlauf von der oberen linken Ecke zur unteren rechten Ecke haben (#373B44 zu #4286f4).
- Text: Der Text innerhalb des Buttons soll eine Größe von 16 Punkten haben und die Schriftart soll Montserrat Medium sein.
- Rand: Der Button soll einen blauen (#4286f4) zwei Pixel breiten Rand besitzen.

Bild-Styles

- Form: Das Bild soll eine rechteckige Form mit abgerundeten Ecken von acht Pixel Radius haben.
- Größe: Die Größe ist auf 300 Pixel beschränkt.
- Hintergrund: Hintergrundfarbe des Bildes soll hellgrau (#f1f1f1) sein.
- Bearbeitung: Das Bild soll das UI-Elemente soweit ausfüllen, sodass das Seitenverhältnis nicht geändert werden muss. Dem Bild soll ein Blur-Effekt hinzugefügt werden mit einem Radius von zwei Pixel.
- Rand: Der Rand soll zwei Pixel breit sein und eine graue Farbe (#b3b3b3) besitzen.

Textfeld-Styles

- Form: Der Button soll eine rechteckige Form mit abgerundeten Ecken, bei einem Radius von acht Pixel besitzen. Außerdem soll der Button einen Schatten besitzen, der einer Erhöhung von vier Pixel entspricht.
- Größe: Die Höhe des Textfeldes soll sich dem Inhalt anpassen und die Breite soll die gesamt mögliche Breite ausfüllen. Der Inhalt soll mit einem Padding von 16 Pixel horizontal und 12 Pixel vertikal umrandet sein.

- Text: Der Text des Textfeldes soll die Schriftart Montserrat Medium besitzen und eine Größe von 16 Pixel. Der Platzhalter des Textfeldes soll die gleiche Schriftgröße haben, jedoch eine Schriftart von Montserrat Light besitzen
- Rand: Der Rand des Textfeldes ist zwei Pixel breit und die Farbe passt sich dem Status des Textfeldes an. Standardmäßig ist diese grau (#b3b3b3), ist das Textfeld fokussiert, dann wechselt die Randfarbe zu blau (#4286f4).
- Validierung: Die Eingabe soll nach dem Format einer E-Mail Adresse validiert werden. Schlägt diese Validierung fehl, wechselt die Randfarbe zu rot (#ff3a30) und eine Fehlermeldung wird direkt unter dem Textfeld angezeigt.

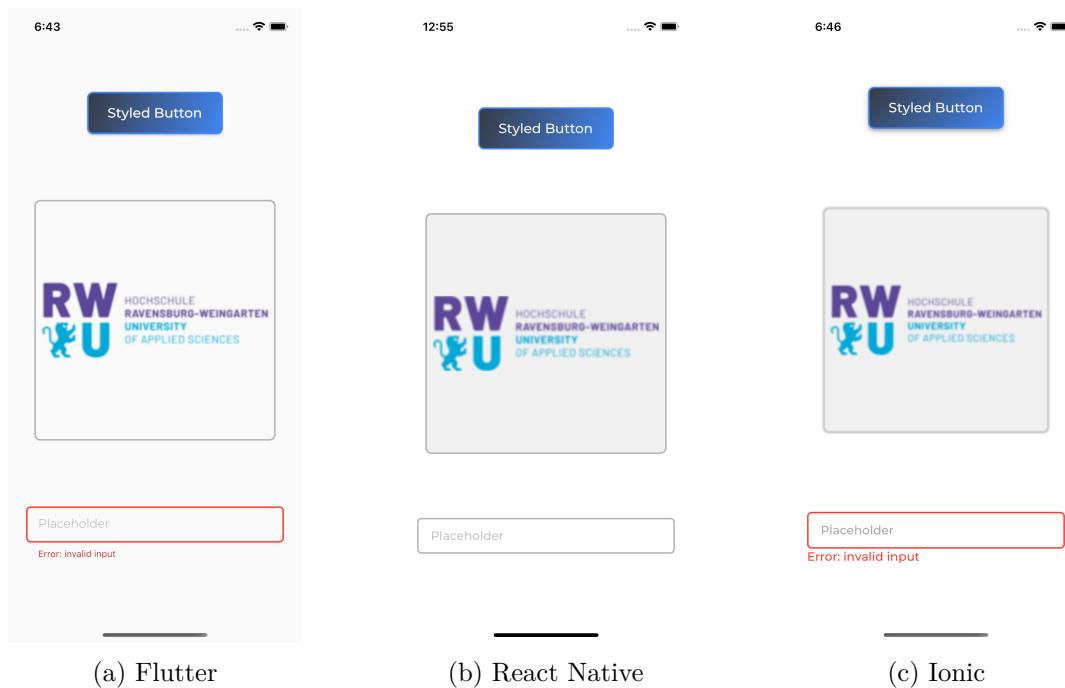


Abbildung 11: Screenshots des sechsten Use-Case.

3.6.1 Implementierung mit Flutter

Das Stylen von Widgets in Flutter Anwendungen erfolgt über die Eigenschaften der Widgets. Für wichtige Eigenschaften gibt es extra Klassen, zum Beispiel `TextStyle` für die Style-Eigenschaften eines Textes. Außerdem bietet Flutter eigene Widgets, welche ausschließlich zum Stylen von weiteren Widgets verwendet werden können, wie zum Beispiel `DecoratedBox` [17].

Für das globale Stylen von Flutter Anwendungen kann ein Theme verwendet werden. Dieses wird mit der Klasse `ThemeData` definiert. Mit dem Widget `Theme` kann das `ThemeData` Objekt in die Flutter Applikation integriert werden und wird von den darunterliegenden Widgets verwendet. Dadurch ist es möglich in einer Anwendung mehrere

Themes zu verwenden, da ein Widget immer das nächste Theme im `WidgetTree` verwendet. Des weiteren ist es möglich unterschiedliche Themes für einen light und dark Mode festzulegen. Wird ein Theme nicht automatisch von einem Widget verwendet, kann das nächste Theme mit `Theme.of(context)` in der Anwendung abgerufen werden [17].

3.6.2 Implementierung mit IONIC

Die UI-Komponenten einer Ionic Applikation lassen sich über CSS-Eigenschaften stylen. Dadurch besteht eine hohe Flexibilität bei der Erstellung von Benutzeroberflächen mit dem Ionic Framework [7].

Mit globalen CSS Variablen lässt sich ein Theme für die Ionic Anwendung erstellen. Diese Variablen sind in der gesamte Anwendung verfügbar, womit die UI-Elemente gleichermaßen gestyliert werden können. Ionic bietet außerdem vorimplementierte globale CSS-Variablen, welche in der `variables.scss` Datei geändert werden können. Ionic Komponenten verwenden diese Variablen als Standardwerte für einige der CSS-Eigenschaften. Mit CSS Media Queries lassen sich unterschiedliche Themes für einen light und dark Mode festlegen [7].

3.6.3 Implementierung mit React Native

Die Erscheinung von UI-Elementen in React Native Applikationen lässt sich durch das `style`-Attribut verändern. Je nach UI-Element ist der Typ dieses Attributes unterschiedlich und es lassen sich unterschiedliche Eigenschaften übergeben. Diesem `style`-Attribut wird ein JavaScript Objekt übergeben, in dem die Styling Eigenschaften definiert sind. Diese Eigenschaften erinnern sehr stark an CSS-Eigenschaften, sind aber in ihrer Vielzahl eingeschränkt [24].

Erstellen von Themes für eine React Native Anwendung kann entweder manuell oder mit der Hilfe von Bibliotheken vorgenommen werden. Manuelles Erstellen von Themes kann durch erstellen von globalen statischen JavaScript Objekte passieren, welche dann in der gesamten Anwendung verfügbar sind und nur noch den UI-Elementen über ihr `style`-Attribut übergeben werden müssen. Bibliotheken können diesen Prozess vereinfachen [24].

4 Diskussion

Im folgenden Kapitel werden zunächst die Ergebnisse aus den Implementierungen der Use-Cases zusammengefasst. Im Anschluss wird ein Vergleich zwischen den drei Cross-Plattform Frameworks gezogen, um zum Schluss alle Ergebnisse zu diskutieren.

4.1 Zusammenfassung der Ergebnisse aus den Use-Cases

In diesem Kapitel werden die Ergebnisse aus der Studie zur Entwicklung der Use-Cases zusammengetragen.

1. Use-Case: Listen

Die Umsetzung des ersten Use-Cases erwies sich mit allen drei Frameworks als unproblematisch. Alle drei Frameworks bieten ein UI-Element, das eine Liste erzeugt und die Listeneinträge beim Scrollen automatisch nach lädt.

Im Performancetest schnitt die Flutter Anwendung am besten ab, was durch das eigene Rendering System basierend auf der Skia Engine erklärt werden kann. Auch die React Native Anwendung erzielt gute Ergebnisse, die nur sehr geringe Unterschiede zu Flutter aufwies. Dies ist vor allem auf die Verwendung von performanten nativen UI-Elementen zurückzuführen. Ionic zeigte die schlechtesten Ergebnisse, die teilweise deutliche Unterschiede zu den beiden anderen Frameworks aufweisen. Dies liegt vor allem an der Architektur des Frameworks, da die App in einer Webview gerendert wird, was nicht so performant ist. Die niedrigere Bildwiederholrate der Ionic Anwendung fiel bereits bei der Durchführung der Testdurchläufe auf.

2. Use-Case: Cards

Auch der zweite Use-Case zur Umsetzung des Card UI-Elements in verschachtelten Scrollviews konnte mit allen drei Frameworks umgesetzt werden.

Hervorzuheben ist bei diesem Test das Flutter Framework, welches mit dem UI-Element `CustomScrollView` und den Sliver UI-Elementen die Entwicklung deutlich vereinfachte. Daher war die Entwicklungszeit für diesen Anwendungsfall mit dem Flutter Framework kürzer. Allerdings war die Entwicklung mit den beiden anderen Frameworks nicht viel komplizierter, da bekannte Webtechnologien verwendet wurden und somit viele Ressourcen für die Problemstellungen zur Verfügung standen.

3. Use-Case: Canvas

Die Implementierung eines Canvas war mit allen drei Frameworks problemlos möglich. Alle drei Frameworks bieten ein Animationssystem zur Ausführung und Verwaltung von Animationen in der Benutzeroberfläche. Flutter bietet ein sehr komplexes und

umfangreiches Animationssystem, um UI-Elemente oder Übergänge zwischen Seiten zu animieren. Das React Native Framework bietet ebenfalls ein umfangreiches Animationssystem zur Animation von UI-Komponenten. Das Ionic Framework bietet ein einfaches und schnell erlernbares System zur Animation der Benutzeroberfläche. Zusätzlich gibt es für alle drei Frameworks externe Bibliotheken, die weitere Animationssysteme implementieren.

4. Use-Case: REST API

Die Integration einer öffentlichen REST-API mit dem HTTP-Protokoll konnte mit allen drei Frameworks problemlos realisiert werden. Insbesondere die Anfragen über das HTTP Protokoll konnten von allen drei Frameworks ohne viel Programmcode umgesetzt werden.

Unterschiede ergaben sich bei der Serialisierung der JSON Daten.

Weitere Unterschiede ergaben sich bei der Erstellung der Benutzeroberfläche und der Anbindung der REST-API an diese.

Flutter mit dem `FutureBuilder` UI Element vereinfachte die Implementierung der entsprechenden Benutzeroberfläche erheblich, da keine manuelle Verwaltung des App-States durch den Entwickler notwendig war. Bei den anderen beiden Frameworks musste der Zustand der Anwendung manuell überwacht und auf Änderungen reagiert werden. Bei solch einfachen Anwendungen ist dies kein Problem und mit wenig Aufwand verbunden. In größeren Anwendungen sollte ein State Management Tool verwendet werden, um den Zustand der Anwendung zu verwalten. Ein bekanntes State Management Tool ist z.B. Redux [1].

5. Use-Case: Persistenz

Die Einbindung eines Key-Value Speichers basierend auf den `SharedPreferences` unter Android und den `NSUserDefaults` unter iOS konnte mit den beiden Frameworks Ionic und Flutter problemlos umgesetzt werden. Die verwendeten Pakete sind einfach und ohne viel Programmcode einsetzbar. Für das React Native Framework gibt es kein Paket, welches einen Key-Value Speicher basierend auf den `SharedPreferences` und `NSUserDefaults` implementiert. Daher musste ein Paket verwendet werden, welches auf einem eigenen Key-Value Speicher basiert.

Bei der Implementierung einer SQLite Datenbank konnten die beiden Frameworks Flutter und React Native punkten. Die verwendeten Pakete ermöglichen eine einfache Erstellung und Verwaltung einer SQLite Datenbank. Vor allem auch die gute Dokumentation der Pakete hat bei der Implementierung geholfen. Im Gegensatz dazu war die Implementierung der SQLite Datenbank mit dem Ionic Framework komplizierter, was vor allem an der veralteten und unübersichtlichen Dokumentation der Bibliothek lag.

Außerdem waren mehr Schritte notwendig, um eine SQLite Datenbank zu erstellen, was die Entwicklungszeit dieses Use-Cases verlängerte.

Auch hier war die Verbindung zwischen der Datenschicht und der Benutzerschnittstelle am einfachsten mit dem Flutter Framework zu realisieren. Aber auch bei den anderen beiden Frameworks war der Aufwand nicht sehr groß.

6. Use-Case: Custom Design

Das Styling der UI-Elemente konnte in diesem Use-Case mit allen drei Frameworks problemlos durchgeführt werden. Die drei Frameworks verwenden unterschiedliche Ansätze für das Styling von UI-Elementen. Das Ionic Framework verwendet CSS, um seine UI-Elemente visuell zu verändern, was sehr viele Möglichkeiten bietet.

React Native ist durch die Verwendung nativer UI-Elemente in seiner Flexibilität eingeschränkt und stößt vermutlich am schnellsten an seine Grenzen.

Flutter verwendet ein eigenes Widget-System, das von der Skia Engine gerendert wird. Es ist auch sehr flexibel, was das Styling der UI-Elemente betrifft.

4.2 Vergleich der Frameworks

Ionic, React Native und Flutter sind drei der bekanntesten Cross-Plattform Frameworks für die Entwicklung mobiler Anwendungen. Jedes Framework bietet seine eigenen Vor- und Nachteile und eignet sich für unterschiedliche Arten von Projekten. In diesem Vergleich werden die drei Frameworks in Bezug auf ihre Performance, UI-Elemente, Entwickler-Tools, Dokumentation, Entwickler-Community, Plattformunterstützung und Lernkurve miteinander verglichen.

Performance

Eines der wichtigsten Kriterien bei der Auswahl eines Frameworks ist die Performance. Mobile Anwendungen müssen schnell und reaktionsfähig sein, um eine optimale Benutzererfahrung zu bieten. Hinsichtlich der Performance haben alle drei Frameworks ihre Stärken und Schwächen.

Ionic basiert auf Capacitor und verwendet eine WebView, um eine mobile Anwendung auf einer Plattform zu erstellen [7]. Das bedeutet, dass die Performance von der Performance der WebView abhängt. Da die WebView zwischen der App und dem Betriebssystem liegt, kann dies zu Performance-Einschränkungen führen.

React Native verwendet eine andere Architektur, die auf nativen Komponenten basiert. Dadurch wird eine höhere Performance erreicht, da der Code direkt auf der Plattform ausgeführt wird [24]. Diese Architektur ermöglicht es Entwicklern, native Funktionen zu nutzen und eine native Benutzeroberfläche zu erstellen. Die einzige Einschränkung

besteht darin, dass der React Native Code zur Laufzeit interpretiert wird, was zu einem Performance-Unterschied zu nativen Anwendungen führen kann.

Die Architektur von Flutter unterscheidet sich von der Architektur der beiden anderen Frameworks. Es verwendet eine eigene Rendering-Engine namens Skia, die auf den meisten Plattformen eine native Geschwindigkeit ermöglicht [17]. Das bedeutet, dass Flutter in Bezug auf die Performance in der Regel besser abschneidet als Ionic und React Native.

UI-Elemente

Die Benutzeroberfläche ist ein weiterer wichtiger Faktor bei der Entwicklung mobiler Anwendungen. Alle drei Frameworks bieten die Möglichkeit, native UI-Elemente zu integrieren, um eine nahtlose Integration mit dem Betriebssystem der Zielplattform zu gewährleisten.

Flutter, Ionic und React Native bieten jedoch auch vorgefertigte UI-Elemente an, die speziell für die Erstellung mobiler Anwendungen entwickelt wurden. Dies kann die Entwicklung beschleunigen und dem Entwickler die Arbeit erleichtern.

React Native verwendet ausschließlich native UI-Elemente zur Erstellung von Benutzeroberflächen [24] und eignet sich daher besonders für die Erstellung von Benutzeroberflächen, die wie eine native App aussehen sollen. Flutter hingegen punktet vor allem durch eine hohe Flexibilität und Konsistenz bei der Gestaltung der Benutzeroberfläche. Dies wird durch das eigene Widgetsystem zur Erstellung von benutzerdefinierten UI-Elementen ermöglicht. Auch Ionic bietet durch die Verwendung bekannter Webentwicklungstechnologien eine hohe Flexibilität bei der Erstellung von UI-Elementen.

Entwickler-Tools

Ein weiterer wichtiger Faktor bei der Auswahl eines Frameworks sind die mitgelieferte Entwicklertools, welche die Entwicklungsgeschwindigkeit erheblich beeinflussen können. Die drei Frameworks Flutter, Ionic und React Native bieten sehr ähnliche Entwickler-Tools, weshalb es kaum Unterschiede zwischen ihnen gibt.

Dokumentation

Eine gute Dokumentation ist unerlässlich, um die Entwicklung von Anwendungen zu erleichtern. Bei der Auswahl eines Frameworks ist es wichtig zu prüfen, ob eine ausführliche und leicht zugängliche Dokumentation vorhanden ist.

Sowohl Ionic als auch React Native verfügen über eine umfangreiche Dokumentation, die von der Community gepflegt wird. Die Dokumentation ist gut strukturiert und enthält viele Beispiele und Tutorials, die Entwicklern helfen, schnell mit der Entwicklung

von Anwendungen zu beginnen.

Flutter hat ebenfalls eine ausführliche Dokumentation, die von Google gepflegt wird. Die Dokumentation ist sehr gut strukturiert und enthält viele Beispiele, die Entwicklern helfen, schnell mit der Entwicklung von Anwendungen zu beginnen. Darüber hinaus bietet Flutter einen eigenen YouTube-Kanal [16], auf dem regelmäßig kurze Erklärvideos zu Widgets, Paketen und wichtigen Konzepten hochgeladen werden.

Entwickler-Community

Die Entwicklergemeinschaft spielt eine wichtige Rolle bei der Unterstützung und Weiterentwicklung von Frameworks. Eine große Community kann die Suche nach Support und Ressourcen erleichtern und die Entwicklung von Tools und Plugins von Drittanbietern fördern.

Ionic und React Native haben eine große Entwicklergemeinde, da sie auf JavaScript basieren, einer der am häufigsten verwendeten Programmiersprachen. Das bedeutet, dass es viele Ressourcen, Tools und Plugins gibt, die von der Community entwickelt wurden.

Flutter hat im Vergleich zu Ionic und React Native eine kleinere Entwickler-Community, da es auf der weniger verbreiteten Programmiersprache Dart basiert. Dennoch hat Flutter in den letzten Jahren schnell an Popularität gewonnen, insbesondere aufgrund seiner Leistungsfähigkeit und Flexibilität bei der Gestaltung von Benutzeroberflächen.

Unterstützung für Plattformen

Ein weiterer wichtiger Faktor bei der Auswahl eines Frameworks ist die Unterstützung verschiedener Plattformen. Alle drei Frameworks unterstützen die Entwicklung von Apps für iOS und Android. Obwohl sich diese wissenschaftliche Arbeit auf diese beiden Plattformen beschränkt, ist es wichtig zu beachten, dass die Frameworks teilweise auch andere Plattformen unterstützen.

Flutter unterstützt auch die Entwicklung von Anwendungen für das Web, Windows, Linux und MacOS. Allerdings kann es Einschränkungen bei der Plattformunterstützung geben [17]. Ionic unterstützt neben den Plattformen Android und iOS auch die Entwicklung von Web- und Electron-Anwendungen [7]. React Native hingegen unterstützt nur die Entwicklung von iOS- und Android-Apps [24].

Lernkurve

Die Lernkurve ist ein weiterer wichtiger Faktor bei der Auswahl eines plattformübergreifenden Frameworks. Entwickler möchten ein Framework wählen, das sie schnell erlernen können, um die Entwicklung ihrer Anwendungen zu beschleunigen.

Ionic und React Native basieren mit JavaScript, HTML und CSS auf weit verbreiteten Technologien, die vor allem im Web-Umfeld bereits von mehr Entwicklern beherrscht werden. Das bedeutet, dass Entwickler, die bereits Erfahrung in der Webentwicklung haben, mit diesen Frameworks schnell in die Entwicklung mobiler Anwendungen einsteigen können. Besonders hervorzuheben ist Ionic, da dieses Framework die Möglichkeit bietet, das JavaScript Framework zu wählen, während React Native Anwendungen mit dem React Framework entwickelt werden müssen.

Flutter hingegen kann aufgrund der Verwendung von Dart und des eigenen Widget-Systems eine steilere Lernkurve aufweisen. Entwickler, die noch keine Erfahrung mit Dart oder der Erstellung von benutzerdefinierten UI-Elementen haben, werden mehr Zeit benötigen, um sich mit dem Framework vertraut zu machen.

5 Zusammenfassung

Ziel dieses Kapitels ist die Zusammenfassung der identifizierten Vor- und Nachteile der drei Frameworks Flutter, React Native und Ionic.

5.1 Flutter

Im Folgenden werden die Vor- und Nachteile des Flutter Frameworks aufgelistet.

Vorteile

- Performance: Das eigene Render-System von Flutter ermöglicht eine sehr gute Performance.
- Flexibilität und Konsistenz: Durch das eigene Widget-System ist die Erstellung von Benutzeroberflächen sehr flexibel und konsistent auf allen Plattformen.
- Native Funktionen: Flutter bietet eine große Anzahl an Paketen und Bibliotheken, um native Funktionen verwenden zu können.

Nachteile

- Lernkurve: Der Einstieg in die Entwicklung von Flutter Applikationen kann aufwendig sein, da eine neue Programmiersprache und ein neues Entwicklungssystem erlernt werden muss.
- Entwicklercommunity: Die Verwendung der Programmiersprache Dart und das junge Alter des Frameworks erklärt eine kleinere Entwicklercommunity im Gegensatz zu den anderen beiden Frameworks.

5.2 Ionic

In diesem Kapitel werden die Vor- und Nachteile von dem Ionic Framework zusammengefasst.

Vorteile

- Lernkurve: Ionic verwendet bekannte Webtechnologien zur Entwicklung, weshalb der Einstieg deutlich schneller ist.
- Entwicklercommunity: Durch die Verwendung von JavaScript als Programmiersprache in Verbindung mit einem Frontend Framework, wie React, Angular oder vue.js ist die Entwicklercommunity sehr groß.
- UI-Elemente: Das Ionic SDK bietet eine große Anzahl vorgefertigter UI-Elemente. Außerdem wird CSS zum Stylen der UI-Elemente verwendet, was eine große Flexibilität mit sich bringt.

Nachteile

- Performance: Dadurch das Ionic ein hybrides Cross-Plattform Framework ist und deshalb in einer Webview ausgeführt wird gibt es Einschränkungen in der Performance.
- Native Funktionen: Aus dem selben Grund ist die Verwendung von nativen Funktionen auch nur eingeschränkt möglich.

5.3 React Native

Dieses Kapitel trägt die Vor- und Nachteile des React Native Frameworks zusammen.

Vorteile

- Performance: Die Verwendung von nativen UI-Elementen ermöglicht eine gute Performance.
- UI-Elemente: Ein weiterer Vorteil durch die Verwendung von nativen UI-Elementen ist, dass die Anwendungen wie native Anwendungen aussehen und sich auch so anfühlen.
- Native Funktionen: Auch das React Native Framework bietet eine große Anzahl an Bibliotheken zur Verwendung von nativen Funktionen.
- Entwicklercommunity: Durch die Verwendung von JavaScript und dem Frontend Framework React steht eine große Entwicklercommunity zu Verfügung.

Nachteile

- UI-Elemente: Die Verwendung von nativen UI-Elementen schränkt dagegen die Flexibilität bei der Erstellung von Benutzeroberflächen ein.
- SDK: Die React Native SDK bietet nur wenige UI-Elemente und native Funktionen. Diese müssen durch drittanbieter Pakete und Bibliotheken hinzugefügt werden, wodurch eine hohe Abhängigkeit von diesen herrscht.

6 Ausblick

Im folgenden Kapitel wird ein Ausblick auf mögliche weitere Untersuchungen gegeben. Durch die Implementierungen der sechs Use-Cases und die umfangreiche Recherchearbeit konnten bereits viele Vor- und Nachteile der drei Frameworks Flutter, React Native und Ionic herausgearbeitet werden. Die Anwendungsfälle mobiler Applikationen sind jedoch sehr vielfältig und können in einer wissenschaftlichen Arbeit nicht alle abgedeckt werden. Aus diesem Grund lohnt es sich, in zukünftigen Studien weitere Anwendungsfälle zu untersuchen, um ein breiteres Spektrum abzudecken und weitere Aussagen über die Frameworks treffen zu können.

Ein weiterer Bereich, der untersucht werden kann, ist die Cross-Plattform Entwicklung mit anderen Plattformen und Gerätetypen. In dieser wissenschaftlichen Arbeit wurde ausschließlich die Entwicklung mobiler Anwendungen für Smartphones und die Betriebssysteme Android und iOS untersucht. In weiteren Studien kann die plattformübergreifende Entwicklung für weitere Plattformen wie Web oder Desktop untersucht werden. Ebenso kann eine Studie durchgeführt werden, die die Cross-Plattform Entwicklung für Tablets oder andere Gerätetypen untersucht.

Die Entwicklung von Cross-Plattform Frameworks ist sehr dynamisch, was bedeutet, dass bestehende Frameworks stark weiterentwickelt werden und ständig neue Funktionen implementieren. Darüber hinaus werden regelmäßig neue Cross-Plattform Frameworks entwickelt, wie zum Beispiel Kotlin Multiplatform [21]. Daher ist ein Vergleich von neu entwickelten Cross-Plattform Frameworks lohnenswert. Außerdem können die Ergebnisse dieser Bachelorarbeit in Zukunft durch Weiterentwicklungen der Frameworks React Native, Ionic und Flutter veraltet sein. Aus diesem Grund wäre auch eine neue Studie in der Zukunft eine Möglichkeit, die Unterschiede erneut zu untersuchen und den Grad der Weiterentwicklung in den Vergleich mit einzubeziehen.

Quellenverzeichnis

- [1] Abramov, D. [n.d.], ‘Redux’, <https://redux.js.org/>. [Online; Stand 1. April 2023]. 55
- [2] Apple Inc. [n.d.a], ‘Apple Developer Documentation’, <https://developer.apple.com/documentation>. [Online; Stand 28. Oktober 2022]. 8, 27
- [3] Apple Inc. [n.d.b], ‘SwiftUI Overview’, <https://developer.apple.com/xcode/swiftui/>. [Online; Stand 28. Oktober 2022]. 8
- [4] @capacitor-community/sqlite [2023], <https://github.com/capacitor-community/sqlite>. [Online; Stand 25. März 2023]. 49
- [5] Capacitor HTTP Plugin [2023], <https://capacitorjs.com/docs/apis/http>. [Online; Stand 21. März 2023]. 45, 46
- [6] dart.dev [2023], ‘http | Dart package’, <https://pub.dev/packages/http>. [Online; Stand 21. März 2023]. 44
- [7] Drifty Co [n.d.], ‘Ionic Dokumentation’, <https://ionicframework.com/docs>. [Online; Stand 10. April 2023]. 10, 25, 26, 27, 28, 29, 30, 33, 34, 37, 41, 46, 53, 56, 58, 66
- [8] FakerJS [n.d.], <https://fakerjs.dev/>. [Online; Stand 15. März 2023]. 32
- [9] flutter.dev [2023a], ‘shared_preferences | Flutter package’, https://pub.dev/packages/shared_preferences. [Online; Stand 25. März 2023]. 49
- [10] flutter.dev [2023b], ‘sqflite | Flutter package’, <https://pub.dev/packages/sqflite>. [Online; Stand 25. März 2023]. 48
- [11] Google LLC [2022a], ‘Dart dokumentation’, <https://dart.dev/overview>. [Online; Stand 1. März 2023]. 12, 13
- [12] Google LLC [2022b], ‘Thinking in Compose’, <https://developer.android.com/jetpack/compose/mental-model>. [Online; Stand 28. Oktober 2022]. 8
- [13] Google LLC [2022c], ‘What is Angular?’, <https://angular.io/guide/what-is-angular>. [Online; Stand 8. März 2023]. 27
- [14] Google LLC [2023a], ‘Documentation | Android Developers’, <https://developer.android.com/docs>. [Online; Stand 10. März 2023]. 8, 28

- [15] Google LLC [2023b], ‘Skia’, <https://skia.org/>. [Online; Stand 6. Februar 2023]. 11
- [16] Google LLC [n.d.a], ‘Flutter - YouTube’, <https://www.youtube.com/@flutterdev>. [Online; Stand 30. März 2023]. 58
- [17] Google LLC [n.d.b], ‘Flutter dokumentation’, <https://docs.flutter.dev>. [Online; Stand 10. April 2023]. 10, 11, 12, 13, 14, 15, 16, 17, 33, 34, 37, 40, 44, 45, 48, 49, 52, 53, 57, 58, 66
- [18] Idaan Aaronsohn et al. [n.d.], ‘React Native Canvas’, <https://github.com/iddan/react-native-canvas>. [Online; Stand 18. März 2023]. 41
- [19] Ionic Long Press [2023], <https://github.com/wbhab/ionic-long-press>. [Online; Stand 25. März 2023]. 49
- [20] Ivanov, P. [n.d.], ‘React Virtuoso’, <https://virtuoso.dev/>. [Online; Stand 15. März 2023]. 33
- [21] Kotlin Foundation [2022], ‘Kotlin Multiplatform’, <https://kotlinlang.org/docs/multiplatform.html>. [Online; Stand 31. März 2023]. 62
- [22] Laricchia, F. [2022], ‘Mobile operating systems’ market share worldwide from January 2012 to August 2022’, <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>. [Online; Stand 28. Oktober 2022]. 7, 8
- [23] Lynch, M. [2022], ‘Capacitor vs Cordova: What are the Differences Between Them’, <https://ionic.io/resources/articles/capacitor-vs-cordova-modern-hybrid-app-development>. [Online; Stand 9. Februar 2023]. 25
- [24] Meta [n.d.], ‘React Native Dokumentation’, <https://reactnative.dev/docs/>. [Online; Stand 10. April 2023]. 10, 18, 19, 20, 21, 22, 23, 24, 27, 34, 42, 46, 50, 53, 56, 57, 58, 66
- [25] MohammadReza Keikavousi [2023], ‘Fake Store API’, <https://fakestoreapi.com/>. [Online; Stand 21. April 2023]. 43, 45
- [26] Preferences Capacitor Plugin [2023], <https://capacitorjs.com/docs/apis/preferences>. [Online; Stand 25. März 2023]. 49
- [27] react-native-async-storage [2023], <https://github.com/react-native-async-storage/async-storage>. [Online; Stand 25. März 2023]. 50

- [28] *React Native Elements* [n.d.], <https://reactnativeelements.com/>. [Online; Stand 15. März 2023]. 34
- [29] *react-native-sqlite-storage* [2023], <https://github.com/andpor/react-native-sqlite-storage>. [Online; Stand 25. März 2023]. 50
- [30] *Swiper* [n.d.], <https://swiperjs.com/>. [Online; Stand 24. März 2023]. 37
- [31] Xanthopoulos, S. and Xinogalos, S. [2013], A comparative analysis of cross-platform development approaches for mobile applications, in ‘Proceedings of the 6th Balkan Conference in Informatics’, pp. 213–220. 9, 10
- [32] You, E. [2023], ‘vue.js Introduction’, <https://vuejs.org/guide/introduction.html>. [Online; Stand 8. März 2023]. 27

Abbildungsverzeichnis

1	Architektur des Flutter Frameworks [17].	12
2	Flutter's Rendering Pipeline [17].	15
3	Übersicht der Architektur von React Native [24].	18
4	Rendering Prozess von React Native Applikationen [24].	21
5	Architektur von Ionic Applikationen [7]	26
6	Screenshots des ersten Use-Cases.	32
7	Screenshots des zweiten Use-Case.	36
8	Screenshots des dritten Use-Case.	39
9	Screenshots des vierten Use-Case.	44
10	Screenshots des fünften Use-Case.	48
11	Screenshots des sechsten Use-Case.	52

Tabellenverzeichnis

1	Ergebnisse aus dem Performance Test	35
---	---	----