MASTERARBEIT

Cross-Plattform-Framework Flutter

ausgeführt am

Studiengang
Informationstechnologien und Wirtschaftsinformatik

Von: Jürgen Reinisch
Pers. Kennz. 1610319039

Graz, am 9. Dezember 2020

....
Jürgen Reinisch

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

....
Jürgen Reinisch

Danksagung

Ich möchte meiner Familie, meinen Freunden und ganz speziell meiner Freundin Anna, dafür danken, dass sie mich während des gesamten Studienprogramms unterstützt haben. Ich kann immer auf sie zählen, wenn die Zeiten hart sind, und sie geben mir immer die nötige Rückendeckung, wenn ich ein wenig zurückgeworfen werde. Großer Dank gilt meinem Betreuer, Herrn Christian Schmid, der mir beratend zur Seite gestanden ist, mit seinem Engagement und seiner konstruktiven Kritik. Ein großer Dank gilt auch den BYTEPOETS. Jeder Kollege in diesem Unternehmen ist eine tägliche Inspiration für mich, und aufgrund ihrer Motivation und ihrer professionellen Einstellung versuche ich immer, in jedem Aspekt des Lebens, das Beste zu geben. Sie sind die kreative und unterstützende Umgebung, in der ein Mensch wachsen und neue Fähigkeiten entwickeln kann. Sie haben es mir auch ermöglicht, an dem Studienprogramm teilzunehmen, wofür ich immer dankbar sein werde.

Jürgen Reinisch

Graz, am 9. Dezember 2020

Kurzfassung

Moderne Frameworks für die Entwicklung von Anwendungen auf mobilen Plattformen entwickeln sich ständig weiter, um den sich ständig ändernden Anforderungen der Welt der Softwareentwicklung gerecht zu werden. Seit 2018 entwickelt Google das Flutter-Framework, das für die Entwicklung mobiler Anwendungen auf den Plattformen Android und iOS eingesetzt wird. Da es sich bei Flutter um das neueste Framework handelt, das eine native Anwendungsleistung verspricht, ist es wichtig, ein besseres Verständnis für die Performance der Technologie zu erhalten und zu erfahren, wie sie sich mit anderen verfügbaren Plattformen messen kann. Aus diesem Grund analysiert diese Arbeit Flutter und vergleicht das Framework mit nativen und anderen plattformübergreifenden Technologien. Prototypen mit den gleichen Funktionalitäten wurden mit Flutter und anderen Plattformen (z. B. Android, iOS, React Native) erstellt und dann getestet, um ihre Leistungsniveaus zu vergleichen. Die Ergebnisse zeigen, dass Flutter in einigen Aspekten (z. B. CPU-Nutzung) eine native Performance erreichen kann, in anderen Aspekten (z. B. Speichernutzung) jedoch schlechter abschneidet als andere Frameworks. Insgesamt zeigt die Forschung, dass Flutter ein sehr vielversprechendes Framework für die plattformübergreifende Entwicklung ist. Zukünftige Arbeiten können auf diese Arbeit aufbauen, um mit verschiedenen Anwendungsfällen zu experimentieren und einen tieferen Einblick in die Leistungsfähigkeit des Frameworks zu erhalten.

Abstract

Modern frameworks for the application development on mobile platforms are constantly evolving to meet the ever-changing requirements of the software development world. Since 2018 Google has been developing the Flutter framework, which is used to develop mobile applications on the Android and iOS platforms. As Flutter is the latest framework that promises native application performance, it is important to get a better understanding of the technology's performance and how it measures up to other available platforms. For this reason, this thesis analyzes Flutter and compares it with native and other cross-platform technologies. Prototypes with the same functionalities were created with Flutter and other platforms (e.g., Android, iOS, React Native) and then tested to compare their performance levels. The results show that Flutter can achieve native performance in some aspects (e.g., CPU usage) but performs worse than other frameworks in other aspects (e.g., memory usage). Overall, the research shows that Flutter is a very promising framework for cross-platform development. Future researchers can build upon this work to experiment with different use cases and get a deeper insight into the framework's performance.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Zielsetzung der Arbeit . . . 2
1.2 Vorgehen und Methodik . . . 3
1.3 Aufbau der Arbeit . . . 4

2 Native App Entwicklung 5
2.1 iOS . . . 6
2.1.1 Objective-C . . . 6
2.1.2 Swift . . . 8
2.1.3 SwiftUI . . . 9
2.1.4 Xcode . . . 10
2.2 Android . . . 12
2.2.1 Java . . . 12
2.2.2 Kotlin . . . 14
2.2.3 Jetpack Compose . . . 15
2.2.4 Android Studio . . . 17
2.3 Lebenszyklen . . . 17
2.3.1 Android Lifecycle . . . 18
2.3.1.1 onCreate . . . 19
2.3.1.2 onStart . . . 20
2.3.1.3 onResume . . . 20
2.3.1.4 onPause . . . 20
2.3.1.5 onStop . . . 20
2.3.1.6 onDestroy . . . 21
2.3.2 iOS Lifecycle . . . 21
2.3.2.1 viewDidLoad() . . . 22
2.3.2.2 viewWillAppear() . . . 22
2.3.2.3 viewDidAppear() . . . 23
2.3.2.4 viewWillDisappear() . . . 23
2.3.2.5 viewDidDisappear() . . . 23
2.4 Mobile Web-Apps . . . 24

3 Cross-Plattform-Technologien 25
3.1 React Native . . . 25
3.1.1 Architektur . . . 26
3.2 Ionic . . . 27
3.2.1 Architektur . . . 28
3.3 Xamarin . . . 29
3.3.1 Architektur . . . 29

3.4 Vergleich der Frameworks . . . 30
3.4.1 Google Search Trends . . . 30
3.4.2 Performance . . . 31
3.4.2.1 Xamarin . . . 31
3.4.2.2 React Native . . . 32
3.4.2.3 Ionic . . . 32
3.4.2.4 Resultat . . . 32
3.4.3 GitHub Stars . . . 32
3.4.3.1 Resultat . . . 33

4 Flutter **34**
4.1 Dart . . . 34
4.2 Architektur . . . 35
4.3 Widgets . . . 36
4.3.0.1 StatefulWidgets . . . 37
4.3.0.2 StatelessWidgets . . . 37
4.4 Lifecycle . . . 38
4.4.1 createState . . . 39
4.4.2 initState . . . 40
4.4.3 dirty . . . 40
4.4.4 build . . . 40
4.4.5 dispose . . . 40
4.4.6 didUpdateWidget . . . 40
4.4.7 setState . . . 41
4.5 Deklarative UI . . . 41
4.6 Vergleich mit anderen Frameworks . . . 42
4.6.1 Native Applikationen . . . 42
4.6.2 React Native . . . 43
4.6.3 Flutter . . . 44
4.6.4 Zusammenfassung . . . 45

5 Testszenario **46**
5.1 Bewertungsmatrix . . . 47
5.1.1 Kennzahlen . . . 47
5.1.1.1 Initiale Ladezeit . . . 48
5.1.1.2 Frame Rate . . . 48
5.1.1.3 CPU-Auslastung . . . 48
5.1.1.4 Memory-Auslastung . . . 48
5.2 Testsoftware . . . 48
5.2.1 Apptim . . . 49
5.2.2 Xcode Instruments . . . 49
5.2.3 Android Studio . . . 49
5.3 Testgeräte . . . 49
5.3.1 iOS . . . 50
5.3.2 Android . . . 50
5.4 Testszenario . . . 50

5.5 Testablauf 51

6 Prototyp **53**
6.1 Plattformen 53
6.2 Konzept 53
6.3 Applikationsstruktur 55
6.3.1 Player 56
6.3.2 ApiClient 56
6.3.3 PlayerRepository 57
6.3.4 PlayerDetail 57
6.3.5 PlayerList 59
6.3.6 PermissionWrapper 61
6.3.7 Main 62
6.3.8 Dependency Injection 62
6.3.9 Firebase 63

7 Ergebnisse **65**
7.1 CPU-Verbrauch 66
7.2 Heruntergeladene Daten 67
7.3 FPS 69
7.4 RAM 70
7.5 Batterie 71
7.6 Startzeit 72
7.7 Zusammenfassung 74

Abbildungsverzeichnis **77**

Tabellenverzeichnis **78**

Listings **79**

Literaturverzeichnis **80**

1 Einleitung

Die moderne Softwareentwicklung wird im Zeitalter der Digitalisierung immer unübersichtlicher. Neue Frameworks und Programmiersprachen entstehen täglich und es ist schwierig den Überblick zu behalten, ob diese für den produktiven Einsatz tauglich sind. Flutter ist ebenso ein Open-Source Development Kit, welches in den letzten drei Jahren immer mehr an Popularität gewonnen hat. Dadurch, dass Flutter von Google geschaffen wurde, um Applikationen auf mobilen Endgeräten zu entwickeln, ohne dabei nativen Source-Code schreiben zu müssen, macht es dieses Framework besonders attraktiv. Diese Arbeit soll dazu dienen, die Prozesse und Methoden mit denen Flutter arbeitet besser zu verstehen und klar die Unterschiede des Frameworks zu anderen Technologien aufzeigen.

In den meisten Fällen wollen sowohl Entwickler als auch Unternehmen mit einer Software so viele Anwender wie möglich erreichen. Um das zu gewährleisten, sollten Applikationen auf mehreren Plattformen funktionieren. Wenn sich Entwickler dazu entschließen Applikationen nativ für jede Plattform zu entwickeln, müssen sie mehrere Code-Basen warten und pflegen. Außerdem müssen Features individuell für jede Plattform und jede Codebase umgesetzt werden was zu mehr Entwicklungsaufwand führt und in höheren finanziellen Kosten resultiert.

Cross-Plattform-Frameworks reduzieren den Entwicklungsaufwand und die Kosten, indem nur eine Codebase für jede Plattform verwendet wird. Das führt dazu, dass nur diese eine Basis gewartet werden muss und Features nicht spezifisch für unterschiedliche Plattformen angefertigt werden müssen. Durch diesen attraktiven Ansatz sind über die letzten Jahre immer mehr Cross-Plattform-Frameworks entstanden. Jedes dieser Frameworks hat seine Vor- und Nachteile, aber in einem Punkt überschneiden sie sich: Sie haben weniger Wartungs- und Entwicklungsaufwand, aber mit dem Nachteil einer schlechteren Performance verglichen mit nativen Applikationen.

2018 wurde Flutter von Google veröffentlicht, um genaue diesen Performanceverlust zu minimieren. Flutter verspricht alle Vorteile von Cross-Plattform-Frameworks, mit einer Performance von nativen Applikationen. Durch dieses Versprechen wurde Flutter sehr viel Aufmerksamkeit in den letzten Jahren geschenkt und große Unternehmen, wie z. B. BMW, haben Applikationen mit dem Framework entwickelt. Die Community hinter Flutter wächst stetig und der schnellste wachsende Skill unter Softwareentwicklern auf der Karriere Plattform LinkedIn ist Flutter (Paul, 2019).

Dadurch, dass Flutter immer mehr an Popularität gewinnt, das Framework jedoch erst seit 2018 existiert, sind die Unterschiede zu anderen Frameworks noch unklar. Diese Arbeit nimmt sich dieser Problematik an und analysiert Flutter und die Komponenten des Frameworks, um diese mit bereits etablierten Frameworks und nativen Ansätzen zu vergleichen.

1.1 Zielsetzung der Arbeit

In der Softwareentwicklung gibt es einige Cross-Plattform-Frameworks, die sich über einen längeren Zeitraum bewährt haben, um mobile Applikationen zu entwickeln. Aus dem Grund, dass Flutter, verglichen mit anderen Frameworks wie z. B. React Native, welches bereits seit 2015 für die Entwicklung genutzt wird, ein neueres Konzept der mobilen App-Entwicklung ist, stellt sich die Frage, inwiefern sich Applikationen unterscheiden, die mit Flutter, React Native oder nativ umgesetzt wurden.

Ziel dieser Arbeit ist es daher folgende Punkte zu behandeln:

- Funktionsweisen nativer Applikationen auf den Plattformen iOS und Android.
- Analyse von bestehenden Cross-Plattform-Frameworks.
- Aufzeigen der Unterschiede der verschiedenen Frameworks.
- Auswahl eines Cross-Plattform-Framework für den Vergleich mit Flutter.
- Analyse von Flutter und den Komponenten des Frameworks.
- Analyse des Konzepts der deklarativen Programmierung.
- Anfertigung eines Prototyps pro Plattform (iOS, Android, Cross-Plattform und Flutter).
- Anfertigung einer Bewertungsskala für den Vergleich der Prototypen.
- Erstellung eines Testablaufs.
- Dokumentation und Interpretation der Ergebnisse.

Nach der Fertigstellung der oben genannten Punkte soll die Forschungsfrage **"Wie unterscheidet sich Flutter hinsichtlich der Performance und Entwicklung verglichen mit anderen Cross-Plattform-Frameworks?"** beantwortet werden können.

1.2 Vorgehen und Methodik

Im ersten Teil der Arbeit, werden durch eine Literaturrecherche die theoretischen Punkte der Arbeit geklärt werden. Es werden die theoretischen Inhalte der Funktionsweise von nativen iOS- und Android-Applikationen analysiert, um ein Verständnis für die Funktionsweise von Cross-Plattform-Frameworks zu schaffen, welche auf diesen Funktionalitäten aufbauen.

Danach wird die aktuelle Marktsituation von Cross-Plattform-Frameworks evaluiert und die stärksten Vertreter untereinander verglichen. Anhand von bestimmten Kriterien wird ein Cross-Plattform-Framework ausgewählt, für welches ein Prototyp angefertigt wird, um einen späteren Vergleich mit Flutter und den nativen Implementierungen zu ermöglichen.

Um genauer zu verstehen, wie der technische Aufbau von Flutter funktioniert, wird das Framework mit seinen Komponenten und Funktionsweisen analysiert. Zusätzlich wird die Architektur der Technologie behandelt, um das Zusammenspiel mit den nativen Plattformen zu erläutern. Dadurch, dass Flutter mit der Rendering Engine von Dart arbeitet, wird diese untersucht, um Verständnis zu schaffen, wie Code kompiliert wird um unabhängig von dem Betriebssystem, dieselben Funktionalitäten und dieselbe grafische Oberfläche am Endgerät anzuzeigen.

Die Komponenten des Frameworks werden genauer in ihrem Aufbau und der Funktionalität erläutert, um im späteren Teil der Arbeit einen Vergleich mit anderen Frameworks durchführen zu können. Dazu werden Best Practices mit Design-Patterns analysiert und der Aufbau eines Flutter Projekts näher erklärt. Die Datei-Struktur des Prototypen wird anhand der Flutter Applikation erklärt.

Im Zuge der Arbeit wird eine Bewertungsmatrix ausgearbeitet, um die verschiedenen Prototypen bewerten zu können. Die Kriterien werden durch eine qualitative Forschung erhoben und für den Vergleich herangezogen. Außerdem wird ein Testszenario definiert, welches für die unterschiedlichen Testversuche einheitlich ist.

Danach werden die Ergebnisse bewertet und interpretiert. Der Vorgang des wissenschaftlichen Arbeitens wird reflektiert. Dieser Abschnitt behandelt den Arbeitsprozess und zeigt einige der Stärken und Schwächen des Projekts auf. Es wird versucht die Forschungsfrage zu beantworten und einen Ausblick auf zukünftige Forschungen zu geben.

1.3 Aufbau der Arbeit

Die Arbeit ist in drei Teile aufgeteilt. Zu Beginn findet eine Literaturrecherche statt um die theoretischen Inhalte der Arbeit zu erarbeiten. Diese qualitative Forschung wird in den ersten vier Kapiteln der Arbeit betrieben und dient als Grundlage für die darauffolgenden Kapitel.

Der zweite Teil der Arbeit befasst sich mit der Implementierung der verschiedenen Prototypen, welche sich aus dem ersten Teil ergeben. Ausgewählter Code der Implementierung wird in diesem Teil der Arbeit näher betrachtet, um ein Verständnis für den Versuchsaufbau zu schaffen und eine Nachvollziehbarkeit zu erreichen. Außerdem wird das Testszenario erläutert und eine Bewertungsskala angefertigt.

Im dritten Teil der Arbeit werden die Ergebnisse aus den Versuchen mit den verschiedenen Implementierungen behandelt. Die erfassten Daten werden analysiert und interpretiert. Zusätzlich findet eine Bewertung der Ergebnisse statt und das wissenschaftliche Arbeiten wird reflektiert, mit einem Ausblick auf zukünftige Forschungen in diesem Bereich.

2 Native App Entwicklung

Eine native Applikation ist eine Anwendung für ein bestimmtes mobiles Gerät. Diese Applikationen werden direkt auf dem Gerät installiert. Benutzer erwerben Apps in der Regel über einen Online-Shop oder Marktplatz wie den App Store auf der Plattform iOS oder Google Play auf der Android Plattform (Unuth, 2019).

Der primäre technische Unterschied zwischen Betriebssystemen für mobile Endgeräte und Betriebssystemen, die auf Notebooks und Desktop-Computern verwendet werden, besteht darin, dass das mobile Betriebssystem kein echtes Multitasking unterstützt. Auf mobilen Geräten kann immer nur eine App aktiv sein (Iversen & Eierman, 2013). Wenn eine andere App gestartet oder durch eine andere App unterbrochen wird (z. B. durch einen Telefonanruf), wird die App, die lief, in den Hintergrund gestellt. Sie bleibt im Hintergrund, bis wieder spezifisch darauf zugegriffen wird. Wenn die Applikation zu lange im Hintergrund bleibt oder wenn der verfügbare Speicher zu gering wird, kann das Betriebssystem die App terminieren. Dieses Hin-und-Her zwischen verschiedenen Zuständen wird als Lebenszyklus der Anwendung bezeichnet (Iversen & Eierman, 2013).

Sowohl Android- als auch iOS-Applikationen haben einen Lebenszyklus. Der Lebenszyklus basiert auf der menschlichen Interaktion mit der Anwendung und dem Bedarf des Betriebssystems an Speicher- und Verarbeitungsressourcen. Wenn eine Interaktion mit dem Gerät stattfindet, ist es möglich, zwischen Anwendungen oder verschiedenen Ansichten innerhalb einer einzigen Anwendung zu wechseln. Wenn dies geschieht, durchläuft die Anwendung verschiedene Zustände. Auf diese Zustände muss bei der Entwicklung reagiert werden, um Datenverlust und inkonsistente Datenzustände zu vermeiden. Aus diesem Grund, ist es notwendig, den Lebenszyklus einer Anwendung zu verstehen, um den Verlust von Daten zu vermeiden (Iversen & Eierman, 2013).

Wie diese Lebenszyklen behandelt werden, variiert zwischen iOS und Android. Das Grundkonzept, das auf Interaktionen in der Applikation reagiert werden muss, ist bei beiden Systemen dasselbe. Jedoch unterscheiden sich iOS und Android hinsichtlich der Umsetzung. So hat iOS beispielsweise nicht nur einen Lebenszyklus für die Applikation, sondern auch einen zusätzlichen Lebenszyklus für die angezeigten Screens (Iversen & Eierman, 2013).

Android-Geräte und iOS-Geräte verfügen jeweils über spezifische Hardware- und Software-Fähigkeiten, die die Art und Weise, wie mit dem Gerät interagiert wird, für jedes Gerät unterschiedlich machen. Um die Fähigkeiten des Geräts vollständig zu erfassen und die Benutzererfahrung nicht zu beeinträchtigen, müssen diese einzigartigen Merkmale berücksichtigt werden. Diese Plattform spezifischen Eigenheiten werden in den folgenden Kapiteln näher erläutert.

2.1 iOS

IPhone Operating System (iOS) ist der Name des Betriebssystems, unter dem das iPhone, der iPod-Touch und das iPad laufen. Es ist die Kern-Software, die auf allen Geräten geladen wird, damit sie andere Apps ausführen und unterstützen können. Um eine iOS-Applikation auf einem Apple Gerät zu installieren, ist es notwendig, ein iOS App Store Package (IPA) auf dem Gerät auszuführen. Eine IPA-Datei ist eine Anwendungsarchivdatei, die eine iOS-Anwendung speichert. Jede IPA-Datei enthält eine Binärdatei und kann nur auf einem iOS-Gerät installiert werden (Feiler, 2014).

Um die Werkzeuge und Funktionsweisen, welche für die Entwicklung verwendet werden, zu verstehen, behandeln die nachfolgenden Kapitel diese Thematik. Es werden die Programmiersprachen, welche fähig sind iOS-Applikationen mithilfe einer IPA-Datei zu erstellen, beschrieben. Dies ist wichtig, um die stetige Entwicklung der Programmiersprachen aufzuzeigen und deren Intention, immer verständlicher für die menschliche Interaktion zu werden.

2.1.1 Objective-C

Objective-C ist eine Programmiersprache, die allgemein verwendet wird. Die Sprache ist nicht spezifisch an eine bestimmte Plattform oder an ein bestimmtes System gebunden, kann aber bei der Entwicklung von einer Vielzahl von Frameworks hilfreich sein. Grundsätzliche baut die Sprache auf der Programmiersprache C auf und erweitert diese um Messaging Funktionen (Kochan, 2011).

Eine der Hauptprogrammiersprachen, die von Apple für das iOS-System verwendet wird, ist Objective-C und kann aus diesem Grund ebenso für die Erstellung mobiler Anwendungen für die Plattform verwendet werden. Da die Programmiersprache C zugrunde liegt, bietet es die Möglichkeit, detaillierter, als mit C, zu arbeiten und Objekte und andere Sprachen besser zu unterstützen (Kochan, 2011).

Im Gegensatz zu vielen anderen Sprachen, die populär sind, bietet Objective-C keinen modularen Mechanismus zur Vermeidung von Kollisionen von Klassen- und Methodennamen. Da Objective-C auf C aufgebaut ist, fehlt die Möglichkeit Namespaces

zu definieren. Alle Klassen in einer Objective-C Anwendung sollten global eindeutig sein. Um Kollisionen zu vermeiden, gibt es also eine Konvention, die Namen der Klassen mit einem Präfix zu versehen. Aus diesem Grund wird das Präfix NS für die Klassen im Foundation Framework und das Präfix UI für die Klassen im UIKit verwendet (Hassan, 2019).

Ein Weiteres Problem besteht im Anwendungsfall, eine Nachricht über ein Null-Objekt zu senden, ohne dass es zum Absturz kommt. Dadurch dass eine strikte Typisierung fehlt, ist es schwer Fehler zu verfolgen und eine Lösung zu finden diese zu beheben. Außerdem ist die Sprache syntaktisch wortreich und komplex da sie im Jahr 1984 entwickelt wurde (Hassan, 2019).

Folgende Code-Snippets 2.1 und 2.2 zeigen beispielhaft die Anzeige eines Labels mit dem Text "Hello World"nachdem ein Button gedrückt wurde.

Die Klasse ViewController.h erbt die Klasse UIViewController, welche das grundlegende Verwaltungsmodell für die iOS-Anwendungen bereitstellt. Die Variable des Labels wird deklariert und die Methode definiert, welche bei der Interaktion mit der Schaltfläche dieser Header-Datei aufgerufen wird. Diese verhalten ist im Code 2.1 ersichtlich.

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController{
IBOutlet UILabel *label;
}
-(IBAction)showLabel;

@end
```

Listing 2.1: Beispiel für Objective-C ViewController

Die Datei ViewController.m enthält die grundlegenden Methoden der Logik. Parallel dazu werden die Instanzmethoden implementiert, die in der Datei ViewController.h zu finden sind. Die Datei ViewController.m ist in Snippet 2.2 angegeben.

```
#import "ViewController.h"

@interface ViewController ()

@end

```

```
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, ↩
       typically from a nib.
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
    }
-(IBAction)showLabel{
    label.text = @"Hello World";
}

@end
```

Listing 2.2: Beispiel für Objective-C

Danach müssen die Funktionen und Komponenten der definierten Dateien noch mit den UI-Komponenten über Xcode, der Entwicklungsumgebung für iOS, verbunden werden. Nachdem dies passiert ist, reagiert der Button auf die Interaktion und es wird die Funktion der Datei aufgerufen welche das Label verändert.

2.1.2 Swift

Swift ist die neueste Sprache (2014), die von Apple entwickelt wurde, um mit den verschiedenen Systemarchitekturen anstelle von komplizierten C oder Objective-C Code-Basen verwendet werden zu können. Die Sprache wurde mit neuen Funktionen erstellt und vollständig als Open-Source-Projekt entwickelt. Das Ziel von Swift ist es, sichere Programmiermuster aus etablierten Sprachen zu übernehmen und gleichzeitig die herkömmliche Programmierung einfacher, flexibler und insgesamt unterhaltsamer zu gestalten (Apple, 2020b).

Dadurch, dass static typing und die Verwendung von optionals und optional chaining möglich ist, bietet Swift eine sicherere Programmierumgebung als Objective-C. Optionals ermöglichen es in Swift mit Null-Objekten umzugehen, ohne dass die Applikation abstürzt. Aus dem Grund, dass die Programmiersprache im Jahr 2014 entwickelt wurde, ist sie weniger komplex zu lernen als Objective-C (Hassan, 2019).

In der 2019 durchgeführten StackOverflow Developer's Survey wurde Swift zu einer der beliebtesten Sprachen gewählt (StackOverflow, 2019). Darüber hinaus ist Swift

die grundlegende Sprache für alle Anwendungen, die von Apple entwickelt werden. (Apple, 2020b).

In dem folgenden Code-Snippet 2.3 wird das Beispiel, welches im Kapitel 2.1.1 umgesetzt wurde, für Swift dargestellt.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }

    @IBAction func showLabel(_ sender: Any) {
        label.text = "Hello World"
    }

}
```

Listing 2.3: Beispiel für Swift

Verglichen mit dem Objective-C-Code, entfällt die Header-Datei. Die Verknüpfung zwischen dem Label und der View findet bei Swift über das Storyboard in Xcode statt. Mit einem simplen Drag & Drop Verhalten wird die Variable zu einem Element des Storyboards zugewiesen.

2.1.3 SwiftUI

Auf der WWDC (Apple Worldwide Developers Conference) 2019, kündigte Apple das neue Framework für Swift das erste Mal an. SwiftUI ist ein innovatives Framework um Benutzeroberflächen auf allen Apple-Plattformen mit der Leistungsfähigkeit von Swift zu erstellen. Benutzeroberflächen für jedes Apple Gerät können mit einer zentralen Sammlung von Werkzeugen und APIs (Application Programming Interface) erstellt werden. Mit SwiftUI kann mit deklarativer Sprache definiert werden, was die Benutzeroberfläche der Anwendung tun soll, und kann somit Einsparungen auf Code-Ebene erzielen (Apple, 2020c). Auf die deklarative Programmierung wird in einem späteren Kapitel Bezug genommen.

In folgendem Code-Beispiel 2.4 wird dieselbe Funktionalität dargestellt wie in den Beispielen zuvor.

```
import SwiftUI

struct ContentView: View {
    State var labelText = ""

    var body: some View {
        VStack {
            Button(action: { self.labelText = "Hello World"
                }) {
                    Text("Show Label")
            }
            Text(labelText)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Listing 2.4: Beispiel für SwiftUI

Die Datei ContentView.swift enthält zwei Structs: ContentView und Preview. Das Struct ContentView ist für den Inhalt und das Layout des Bildschirms verantwortlich, während ContentView_Previews für die Erstellung der Vorschau zuständig ist, die auf der rechten Seite im Xcode Editor angezeigt wird.

In der Sektion body sind die Elemente definiert, die am Bildschirm angezeigt werden. Im Beispiel wird über die action definiert, dass sich der Wert der Variable labelText bei Interaktion ändert und nach dem Button-Element angezeigt wird.

2.1.4 Xcode

Xcode ist die Integrierte Entwicklungsumgebung (IDE), die für die Erstellung einer iOS-Applikation verwendet wird. Das bedeutet, dass es alle für die Erstellung einer Anwendung erforderlichen Werkzeuge (insbesondere einen Texteditor, einen Compiler und ein Build-System) in einem Softwarepaket zusammenfasst, anstatt sie als

eine Reihe von einzelnen Werkzeugen zu belassen, die durch Skripte miteinander verbunden sind. Neben der Erstellung von iOS-Anwendungen kann die IDE auch zur Erstellung von OSX-Anwendungen verwendet werden (Feiler, 2014).

Mit Xcode werden iOS-Applikationen geschrieben und kompiliert. Es ist eine Anwendung für die Erstellung von Apps und zusätzlich ein Code-Editor, der mehrere Sprachen unterstützt. Mit diesem Werkzeug können Benutzeroberflächen, Apps und Spiele entwickelt werden. Es ist das einzige offiziell unterstützte Werkzeug das von Apple für die Erstellung von Applikationen und die Veröffentlichung in Apples App-Store erstellt wurde (Feiler, 2014).

2.2 Android

Android ist ein Open-Source Betriebssystem welches auf Linux basiert und wird für mobile Geräte wie Smartphones und Tablet-Computer verwendet. Android wurde von der Open Handset Alliance unter der Führung von Google und anderen Unternehmen entwickelt (Erik, 2013).

Android-Anwendungen können in den Sprachen Kotlin, Java und C++ geschrieben werden. Die Werkzeuge aus dem Android Software Development Kit (SDK) kompilieren den Code zusammen mit beliebigen Daten- und Ressourcendateien in ein Android Application Package (APK), ein Android-Paket, das eine Archivdatei mit der Endung .apk ist. Eine APK-Datei enthält den gesamten Inhalt einer Android-Anwendung und ist die Datei, die von Geräten, die Android als Betriebssystem nutzen, für die Installation der Anwendung verwendet wird. Es ist das Gegenstück der IPA-Datei, welche für iOS-Applikationen verwendet wird (Erik, 2013).

In den nachfolgenden Kapiteln werden die Programmiersprachen und Werkzeuge erläutert, welche benötigt werden, um eine Android-Applikation zu entwickeln und eine APK-Datei zu erzeugen damit diese auf einem Android-Endgerät installiert werden kann.

2.2.1 Java

Java ist eine Programmiersprache und Computerplattform, die erstmals 1995 von Sun Microsystems veröffentlicht wurde. Obwohl die Sprache primär für internetbasierte Anwendungen verwendet wird, ist Java eine einfache, effiziente und allgemein einsetzbare Sprache. Java wurde ursprünglich für eingebettete Netzwerkanwendungen entwickelt, die auf verschiedenen Plattformen läuft. Sie ist eine portable, objektorientierte und interpretierte Sprache (Austerlitz, 2003).

Dadurch, dass das Android-Betriebssystem in Java geschrieben wurde, werden auch Applikationen für mobile Endgeräte, die Android verwenden in Java programmiert. Die wichtigsten Kernfunktionen von Java laut (Erik, 2013) sind:

- einfache Erlernbarkeit
- Plattformunabhängigkeit
- Sicherheit auf virtuellen Maschinen
- Objektorientiertheit

Android basiert stark auf diesen Java Grundlagen. Das Android SDK enthält viele Java Bibliotheken wie z. B. Datenstruktur-Bibliotheken, Mathematik-Bibliotheken, Grafik-Bibliotheken, Netzwerk-Bibliotheken, sowie spezielle Android-Bibliotheken, die bei der Entwicklung von Android-Anwendungen benötigt werden (Erik, 2013).

Ähnlich wie bei iOS, trennt sich der Code, der für einen Screen benötigt wird, in zwei grundlegende Dateien auf. Die Logik wird in der sogenannten Activity definiert und ist gleichzusetzen mit dem ViewController einer iOS-Applikation. Jeder Screen der Applikation wird mithilfe einer Activity abgebildet. Das zugehörige Layout wird aufgrund einer XML-Datei definiert (Erik, 2013). In den nachfolgenden Beispielen 2.5 und 2.6 ist eine Funktionalität abgebildet, die dieses Konzept widerspiegelt.

In der MainActivity verändert die showText() Methode die Sichtbarkeit der TextView und zeigt somit den Text "Hello Worldäm Bildschirm an.

```
public class MainActivity extends AppCompatActivity {

    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textView = findViewById(R.id.textView);
    }

    public void showText() {
        textView.setVisibility(View.VISIBLE);
    }
}
```

Listing 2.5: MainActivity der Hello World Java Applikation

Das Gegenstück der MainActivity wird in 2.6 dargestellt. In dieser Datei wird der Aufbau des Screens definiert. Das Button-Element weist ein onClick-Property auf, welches auf die Funktion showText() der MainActivity referenziert.

Initial ist die TextView unsichtbar, was durch das Property visibility in der Layout-Datei ersichtlich ist. Nachdem der Button gedrückt wurde, wird dieses Property verändert und der Text "Hello World"wird sichtbar.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:↩
   android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:onClick="showText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:visibility="invisible"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 2.6: Layout der Hello World Java Applikation

2.2.2 Kotlin

Kotlin ist eine Open-Source Programmiersprache, die ursprünglich für die Java Virtual Machine (JVM) und Android entwickelt wurde und objektorientierte und funktionale Programmierfunktionen kombiniert. Sie konzentriert sich auf Interoperabilität, Sicherheit, Klarheit und Tooling-Unterstützung (Horton, 2019).

Das Android SDK ist größtenteils in Java geschrieben. Über Android Studio ist es jedoch möglich, Kotlin-Code in eine funktionierende Java-Anwendung zu verwan-

deln. Der Code wird mit dem Java aus dem SDK in einer Zwischenform zusammengeführt, bevor er in ein Format namens Dalvik Executable (DEX) konvertiert wird, den das Android-Gerät für die Konvertierung in eine laufende Anwendung verwendet. Auf den DEX-Code wird in dieser Arbeit nicht näher eingegangen, da es sich um eine Funktionalität handelt, die für die Entwicklung nicht ausschlaggebend ist. Unabhängig davon, ob die Anwendung in Kotlin oder Java programmiert wurde, ist der resultierende DEX-Code derselbe (Horton, 2019).

Folgender Code 2.7 zeigt dieselbe Applikation des Java Kapitels, geschrieben in Kotlin. Dadurch, dass die Layout-Datei in XML geschrieben ist, verändert sich diese Datei nicht und ist somit unabhängig von der gewählten Programmiersprache.

```
class MainActivity : AppCompatActivity() {
    private val textView: TextView? = null // Null Safety

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    fun showText(view: View) {
        textView?.visibility = View.VISIBLE
    }
}
```

Listing 2.7: MainActivity der Hello World Kotlin Applikation

Die Hauptunterschiede verglichen mit dem Code 2.5 liegen in der Null-Safety, welche standardmäßig aktiviert ist in Kotlin und der Schreibweise, mit der man auf das Property der textView zugreift. Es gibt noch weitere Unterschiede zwischen Kotlin und Java, welche in dieser Arbeit aber nicht näher behandelt werden. Für spätere Vergleiche wird Kotlin verwendet.

2.2.3 Jetpack Compose

Google kündigte Jetpack Compose erstmals auf der I/O Entwicklerkonferenz 2019 an und ist ein ungebündeltes Toolkit, das die UI-Entwicklung durch die Kombination eines reaktiven Programmiermodells mit Kotlin vereinfachen soll. Es kann als Android Gegenstück mit des aus Kapitel 2.1.3 erwähnten Frameworks SwiftUI der iOS Plattform verglichen werden (Google, 2020f).

Jetpack Compose bedient sich ebenfalls dem Konzept der deklarativen UI was den Trend der Vereinfachung von Programmiersprachen widerspiegelt. Zum Zeitpunkt dieser Arbeit befindet sich die Programmiersprache noch in der Alpha Phase. Nachfolgend ist die Funktionalität der Applikation der vorhergehenden Kapitel mit Jetpack Compose im Snippet 2.8 abgebildet.

```kotlin
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeMasterTheme {
                // A surface container using the 'background' color from the theme
                Surface(color = MaterialTheme.colors.background) {
                    Greeting("Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(text = name)
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    var text = ""
    ComposeMasterTheme {
        Button(onClick = {text = "Hello World!"}){
            Text("Click me")
        }
        Greeting(text)
    }
}
```

Listing 2.8: MainActivity der Hello World Jetpack Compose Applikation

2.2.4 Android Studio

Android Studio ist die offizielle IDE für die Anwendungsentwicklung der Android Plattform. Sie basiert auf der IntelliJ IDEA, einer integrierten Java Entwicklungsumgebung für Software und umfasst deren Codebearbeitungs- und Entwicklungswerkzeuge. Um die Anwendungsentwicklung innerhalb des Betriebssystems von Android zu unterstützen, verwendet Android Studio ein Gradle basiertes Build-System, einen Emulator, Code-Vorlagen und GitHub-Integration (Google, 2020a).

2.3 Lebenszyklen

Wenn durch eine Anwendung navigiert wird, sie verlassen wird und zu ihr zurückgekehrt wird, durchlaufen die Aktivitätsinstanzen in der Anwendung verschiedene Zustände in ihrem Lebenszyklus. Es gibt vordefinierte Events, die es ermöglichen auf diese Änderungen in der Anwendung zu reagieren und zu erkennen, dass sich ein Zustand geändert hat. Diese werden Lifecycle Callbacks genannt (Li & Ouyang, 2017).

Ein Callback ist ein Stück ausführbarer Code, welcher als Argument an einen anderen Code übergeben wird, von dem erwartet wird, dass dieser das Argument zu einem geeigneten Zeitpunkt ausführt. Der Aufruf kann sofort erfolgen, wie bei einem synchronen Callback, oder er kann zu einem späteren Zeitpunkt erfolgen, wie bei einem asynchronen Callback (Terry Jones, 2014).

Die Lifecycle Callbacks bedienen sich dieses Konzepts der Callbacks und werden aufgerufen, sobald die Änderungen in den Aktivitätsinstanzen eintreten. Innerhalb dieser Funktionen kann definiert werden, was mit der Applikation passieren soll, falls sich der Status ändert. Diese Reaktion auf Änderungen ermöglicht es, die Anwendung robuster und leistungsfähiger zu machen. Beispielsweise kann eine gute Implementierung dieser Methoden dazu beitragen, dass die Applikation folgende Szenarien vermeidet (Li & Ouyang, 2017):

- Abstürze, wenn ein Telefonanruf eintrifft oder zu einer anderen App gewechselt wird, während die App benutzt wird.
- Den Verbrauch wertvoller Systemressourcen, wenn sie nicht aktiv genutzt werden.
- Verlust des Fortschritts in der Applikation, wenn diese verlassen wird.
- Absturz oder Verlust des Fortschritts, wenn der Bildschirm zwischen Hoch- und Querformat wechselt.

In den folgenden Kapiteln werden die Lifecycle Callbacks für Android und iOS näher untersucht um zu verstehen, wie Cross-Plattform-Frameworks auf diese zugreifen können.

2.3.1 Android Lifecycle

Um Übergänge zwischen den Phasen des Activity Lifecycle zu verwalten, bietet das System sechs Callbacks (Li & Ouyang, 2017):

- onCreate()
- onStart()
- onResume()
- onPause()
- onStop()
- onDestroy()

Das System ruft jeden dieser Callbacks auf, wenn eine Activity in einen neuen Zustand eintritt. In dem Fall, dass eine Activity verlassen wird, ruft das System Methoden auf, um diese abzubauen. In einigen Fällen passiert das nur teilweise. Die Activity befindet sich immer noch im Speicher, wenn zu einer anderen Anwendung gewechselt wird und kann immer noch in den Vordergrund zurückkehren. Wenn das passiert, wird die Activity an der Stelle fortgesetzt, an der die Applikation in den Hintergrund geladen wurde. Mit wenigen Ausnahmen ist es Apps nicht möglich, Activities zu starten, wenn sie im Hintergrund laufen (Li & Ouyang, 2017). Die Lifecycle Callbacks sind in Abbildung 2.1 ersichtlich.

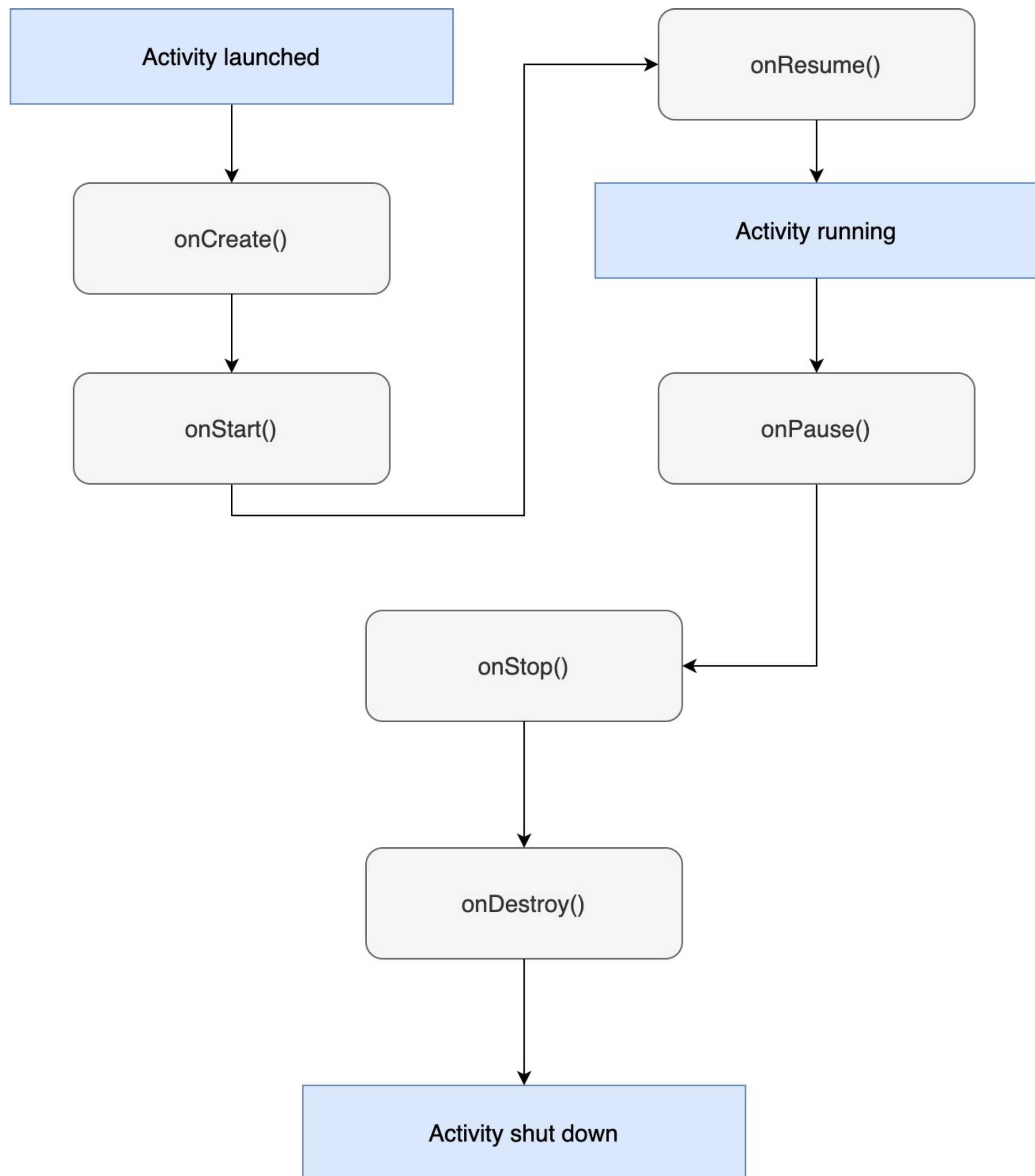

Abbildung 2.1: Android Activity Lifecycle (Quelle: vgl. Li & Ouyang, 2017)

2.3.1.1 onCreate

Dieser Callback, der ausgelöst wird, wenn das System die Activity erstellt, muss implementiert werden. Wenn die Activity erstellt wurde, geht diese in den Status Created über. In der Methode onCreate() wird die grundlegende Logik für den Start der Activity ausgeführt, die während der gesamten Lebensdauer nur einmal vorkommen sollte. Beispielsweise könnte die Implementierung von onCreate(), Daten an Listen binden, die Activity mit einem ViewModel verknüpfen und einige Klassenvariablen instanzieren. Diese Methode empfängt den Parameter savedInstanceState, bei dem es

sich um ein Objekt handelt, das den zuvor gespeicherten Zustand der Activity enthält. Wenn diese noch nie zuvor existiert hat, ist der Wert des Objekts null (Google, 2020g).

2.3.1.2 onStart

Wenn die Activity in den Zustand Started übergeht, ruft das System diesen Callback auf. Der onStart() Aufruf macht die Activity am Screen sichtbar, während sich die Anwendung darauf vorbereitet, dass diese in den Vordergrund tritt und interaktiv wird. Bei dieser Methode initialisiert die App z. B. den Code, der für die Benutzeroberfläche verantwortlich ist (Google, 2020g).

2.3.1.3 onResume

Der Resumed Status wird erreicht, wenn die Applikation in den Vordergrund gelegt wird und ruft danach den onResume() Callback auf. Dies ist der Zustand, in dem die Applikation für Interaktionen verfügbar ist. Die Anwendung bleibt in diesem Zustand, solange bis ein Ereignis eintritt, welches den Fokus verändert. Ein solches Ereignis kann z. B. ein Telefonanruf, das Navigieren zu einer anderen Activity oder das Ausschalten des Gerätebildschirms sein (Google, 2020g).

2.3.1.4 onPause

Wenn ein Unterbrechungsereignis eintritt, geht die Activity in den Zustand Paused über, und das System ruft den onPause() Callback auf. Ändert sich der Zustand von Resumed auf Paused, ruft das System erneut die Methode onResume() auf. Das System ruft den Callback als erstes Anzeichen dafür auf, dass die Activity verlassen wird und diese nicht mehr im Vordergrund steht (Google, 2020g).

2.3.1.5 onStop

Ist die Activity nicht mehr sichtbar, ist sie in den Zustand Stopped übergegangen und das System ruft den onStop() Callback auf. Dies kann z. B. dann der Fall sein, wenn eine neu gestartete Activity den gesamten Bildschirm bedeckt. Das System kann onStop() auch aufrufen, wenn die Activity abgeschlossen ist und kurz vor dem Abbruch steht (Google, 2020g).

2.3.1.6 onDestroy

onDestroy() wird aufgerufen, bevor die Activity zerstört wird. Das System ruft diesen Callback laut (Google, 2020g) unter folgenden Bedingungen auf:

- Die Activity wird beendet.
- Das System zerstört die Activity aufgrund einer Konfigurationsänderung (wie z. B. Geräterotation oder Mehrfenstermodus).

2.3.2 iOS Lifecycle

Das iOS Betriebssystem hat, ebenso wie Android, Methoden, um mit Änderungen der UI umzugehen. Der UIViewController verfügt über die Methoden, welche die View Hierarchie verwalten. Das iOS Betriebssystem ruft diese Methoden automatisch zu geeigneten Zeiten auf, wenn ein View-Controller zwischen Zuständen wechselt. Folgende Lifecycle Events sind auf der iOS Plattform laut (Apple, 2020a) verfügbar:

- viewDidLoad()
- viewWillAppear()
- viewDidAppear()
- viewWillDisappear()
- viewDidDisappear()

Diese Methoden sind in folgender Abbildung (Apple, 2020a) ersichtlich.

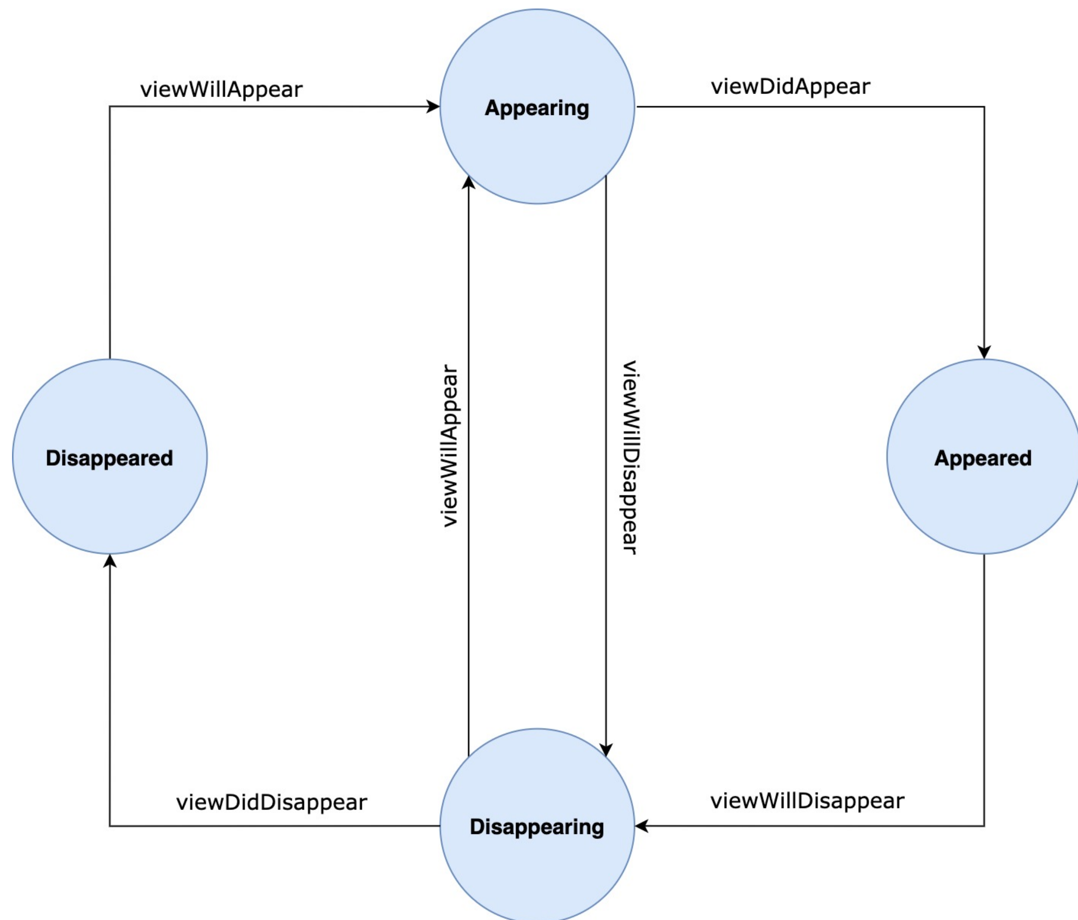

Abbildung 2.2: iOS ViewController Lifecycle (Quelle: vgl. Apple, 2020a)

2.3.2.1 viewDidLoad()

Diese Methode wird initial im Lebenszyklus des View-Controllers aufgerufen, wenn der Inhalt des View-Controllers erstellt wurde und von einem Storyboard geladen wird. Die UI-Elemente des View-Controllers haben zum Zeitpunkt des Aufrufs dieser Methode garantiert gültige Werte. Sie wird verwendet um ein zusätzliches Setup, welches ein View-Controller benötigt, auszuführen und z. B. Daten zu laden, die essenziell für die View sind (Apple, 2020a).

2.3.2.2 viewWillAppear()

Diese Methode wird kurz vor dem Hinzufügen der View des View-Controllers zur View-Hierarchie der Anwendung aufgerufen. In dieser ist die Logik implementiert, die ausgeführt werden muss, bevor die View auf dem Bildschirm präsentiert wird. Es ist jedoch keine Garantie dafür, dass die View sichtbar wird. Die Ansicht kann durch

andere Ansichten verdeckt oder ausgeblendet werden. Diese Methode zeigt lediglich an, dass die Inhaltsansicht kurz davor steht, der Hierarchie der App hinzugefügt zu werden (Apple, 2020a).

2.3.2.3 viewDidAppear()

Im Gegensatz zu der viewWillAppear() Methode, wird die viewDidAppear() Methode, kurz nachdem die View des View-Controllers zur View-Hierarchie der Anwendung hinzugefügt wurde aufgerufen. Sie wird verwendet, um alle Operationen auszulösen, die ausgeführt werden müssen, sobald die View auf dem Bildschirm präsentiert wird, wie z. B. das Abrufen von Daten oder das Starten einer Animation. Wie zuvor, garantiert der Aufruf der Methode nicht, dass die View sichtbar ist (Apple, 2020a).

2.3.2.4 viewWillDisappear()

Kurz bevor die View aus der View-Hierarchie der Anwendung entfernt wird, wird dieser Callback aufgerufen. Bereinigungsaufgaben, wie das Persistieren von Änderungen oder das Freigeben von Systemressourcen sollten in dieser Methode ausgeführt werden (Apple, 2020a).

2.3.2.5 viewDidDisappear()

Die Methode wird aufgerufen, kurz nachdem die View des View-Controllers aus der View-Hierarchie der Anwendung entfernt wurde. Sie wird verwendet, um zusätzliche Logik für die Deinitialisierung durchzuführen. Die Methode wird nur aufgerufen, wenn die View aus der Hierarchie der App entfernt wurde (Apple, 2020a).

2.4 Mobile Web-Apps

Wenn eine native Anwendung erstellt wird, müssen APIs verwendet werden, die für das Betriebssystem des Geräts spezifisch sind. Das bedeutet im Allgemeinen auch, dass mit einer plattformspezifischen Sprache und einem plattformspezifischen SDK gearbeitet werden muss. Im Fall von iOS beispielsweise wird Swift, Objective-C oder SwiftUI verwendet (Microsoft, 2012).

Eine Webanwendung wird mit den Programmiersprachen HTML, CSS und JavaScript erstellt. Die benötigten Daten werden von einem Webserver über das Internet geladen und im Webbrowser auf dem Gerät dargestellt. In den meisten Fällen ist der Browser auf dem Gerät vorinstalliert, aber viele Geräte ermöglichen es, alternative Browser zu installieren (Microsoft, 2012). Mobile Web-Apps werden in dieser Arbeit nicht näher behandelt, da diese keinen nativen Code auf dem Gerät ausführen, sondern lediglich eine Webseite in Form einer App emulieren.

3 Cross-Plattform-Technologien

Plattformübergreifende Frameworks für die Anwendungsentwicklung ermöglichen es Entwicklern, mobile Anwendungen zu erstellen, die mit mehr als einem Betriebssystem, z. B. iOS und Android, kompatibel sind. Sie bieten die Möglichkeit, den Code einmal zu schreiben und ihn danach überall, auch für andere Plattformen auszuführen, wodurch eine Software schneller, sicherer und mit besserer Qualität veröffentlicht werden kann (Scott, 2020).

Die plattformübergreifende Entwicklung bietet die Flexibilität, die Anwendung mit einer universellen Sprache wie z. B. JavaScript zu erstellen, die dann auf verschiedene Plattformen exportiert werden kann. Auf diese Weise kann eine App über mehrere Plattformen hinweg funktionieren (Scott, 2020).

Durch die Verwendung von Frameworks wie React Native, Ionic oder Xamarin kann eine Anwendung erstellt werden, die native APIs verwendet. Das ermöglicht eine Ausführung des Codes über alle Plattformen hinweg, ohne dass jede einzelne Funktion separat programmiert werden muss. Das Ergebnis ist eine Anwendung, die performant ist und eine einheitliche Codebase für alle Plattformen aufweist (Scott, 2020).

Ziel dieses Kapitels ist es, die populärsten Cross-Plattform-Technologien zu beschreiben und die speziellen Eigenschaften jedes Frameworks zu analysieren. Außerdem werden diese miteinander verglichen, um eine Technologie auszuwählen, mit der eine Applikation im Zuge dieser Arbeit umgesetzt wird. Diese Applikation dient in einem späteren Kapitel der Arbeit für den Vergleich mit nativ implementierten Applikationen.

3.1 React Native

React Native ist ein JavaScript Framework das für das Schreiben von mobilen Anwendungen für die Plattformen iOS und Android entwickelt wurde. Es basiert auf React, der JavaScript Bibliothek von Facebook zur Erstellung von Benutzeroberflächen, zielt aber nicht auf den Browser ab, sondern auf mobile Plattformen. Somit können mobile Anwendungen mithilfe von Webtechnologien geschrieben werden, die nativ aussehen und auch ein natives Verhalten aufweisen, mithilfe einer JavaScript-Bibliothek.

Da der Großteil des Codes, der geschrieben wird, zwischen den Plattformen ausgetauscht werden kann, macht es React Native möglich, gleichzeitig für Android und iOS zu entwickeln (Boduch & Derks, 2020).

Ähnlich wie bei React für die Webentwicklung, werden React Native Anwendungen mit einer Kombination aus JavaScript und XML Markup, bekannt als JSX, geschrieben. Intern ruft die React Native Bridge die nativen Funktionen der APIs in Objective-C (für iOS) oder Java (für Android) auf. Auf diese Weise wird die Anwendung mit echten mobilen UI-Komponenten und nicht mit Webviews angezeigt. React Native stellt JavaScript Schnittstellen für Plattform APIs zur Verfügung, sodass die React Native Anwendungen auf Plattformfunktionen wie der Kamera oder den Standort des Benutzers zugreifen können (Boduch & Derks, 2020).

React Native unterstützt sowohl iOS als auch Android und hat das Potenzial, auch auf zukünftige Plattformen zu expandieren. Facebook setzt diese Technologie bereits in der Produktion für benutzerorientierte Anwendungen ein (Boduch & Derks, 2020).

3.1.1 Architektur

Der größte Teil des nativen Codes ist im Falle von iOS in Objective-C oder Swift geschrieben, während dieser im Falle von Android in Java oder Kotlin geschrieben ist. Für das Schreiben einer React Native-Applikation wird nur JavaScript-Code verwendet. Die JavaScript-Virtual-Machine führt diesen JavaScript Code auf den Plattformen aus. Auf iOS oder Android-Simulatoren und Geräten verwendet React Native den JavaScriptCore, die JavaScript-Engine, die Safari betreibt. JavaScriptCore ist eine Open-Source-JavaScript-Engine, die ursprünglich für WebKit entwickelt wurde. Im Falle von iOS verwendet React Native den JavaScriptCore, welcher von der iOS-Plattform bereitgestellt wird. Es wurde erstmals in iOS 7 zusammen mit OS X Mavericks eingeführt. Im Falle von Android bündelt React Native den JavaScriptCore zusammen mit der Anwendung (Facebook, 2020).

In den meisten Fällen würde die gesamte React Native-Anwendung in JavaScript geschrieben werden. React Native bündelt den gesamten JavaScript-Code in eine einzige Datei (main.bundle.js). Wenn nun die React Native-Anwendung gestartet wird, ist das erste Element, das geladen wird, der native Einstiegspunkt. Der Native-Thread erzeugt den JavaScript-Virtual-Machine-Thread, der den gebündelten Code ausführt. Der JavaScript-Code enthält die gesamte Geschäftslogik der Anwendung (Facebook, 2020).

Der Native-Thread sendet Nachrichten über die React Native-Bridge, um die Anwendung zu starten. Die React Native-Bridge ist eine C++ bzw. Java-Brücke, die für die Kommunikation zwischen dem nativen und dem Javascript-Thread zuständig ist. Diese Bridge ist in Abbildung 3.1 ersichtlich. Danach beginnt der erzeugte

Javascript-Thread mit der Ausgabe von Anweisungen an den Native-Thread über die React Native-Bridge. Die Anweisungen beinhalten, z. B. welche Ansichten geladen oder welche Informationen von der Hardware abgerufen werden sollen. Wenn der JavaSript-Thread beispielsweise möchte, dass eine Ansicht und ein Text erstellt werden, fasst er die Anfrage in einer einzigen Nachricht zusammen und sendet sie an den Native-Thread, um sie zu rendern (Facebook, 2020).

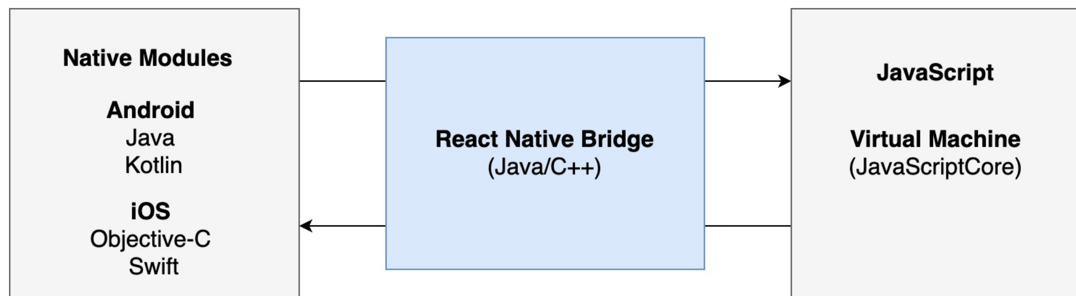

Abbildung 3.1: Architektur von React Native (Quelle: vgl. Facebook, 2020)

3.2 Ionic

Das Ionic-Framework ist ein leistungsstarkes Werkzeug zur Erstellung hybrider mobiler Anwendungen. Ionic basiert auf Webkomponenten-Standards und ist agnostisch. Agnostisch bedeutet, dass etwas verallgemeinert wird, damit es interoperabel ist. Ionic-Komponenten werden als benutzerdefinierte Elemente unter Verwendung des eigenen Open-Source-Tools Stencil erstellt. Verwendete Frameworks können frei gewählt werden, einschließlich Angular, React und Vue (Cheng, 2018).

Die Ionic-Komponenten sind so gestaltet, dass sie ein ähnliches Aussehen und Gefühl wie native Komponenten in Anwendungen haben. Mit diesen integrierten Komponenten können schnell Prototypen mit ausreichend guten Benutzeroberflächen erstellt werden (Cheng, 2018).

Das Framework nutzt Apache Cordova als Laufzeitumgebung für die Kommunikation mit nativen Plattformen. Apache Cordova ist ein Open-Source-Framework, das es ermöglicht, HTML-, CSS- und JavaScript-Inhalte zur Erstellung einer nativen Anwendung für eine Vielzahl von mobilen Plattformen zu verwenden. Cordova rendert die Webanwendung in eine WebView. Eine WebView ist eine Anwendungskomponente, die zur Anzeige von Web-Inhalten innerhalb einer nativen Anwendung verwendet wird (Cheng, 2018).

Die Webanwendung, die in diesem Container läuft, ist wie jede andere Webanwendung, die in einem mobilen Browser laufen würde und kann zusätzliche HTML-Seiten öffnen, JavaScript-Code ausführen, Mediendateien abspielen und mit Remote-

Servern kommunizieren. Diese Art von mobiler Anwendung wird oft als Hybridanwendung bezeichnet. Ionic-Anwendungen können alle Cordova-Plugins verwenden, um mit der nativen Plattform zu interagieren. Die aktuelle Release-Version, zum Zeitpunkt dieser Arbeit, des Ionic-Frameworks ist 5.0 (Cheng, 2018).

3.2.1 Architektur

Ionic-Applikationen werden mit Cordova erstellt. Cordova ist ein Werkzeug zur Konvertierung von HTML, CSS und JavaScript in Anwendungen, die auf Mobil- und Desktop-Geräten ausgeführt werden können und bietet eine Plugin-Architektur für den Zugriff auf native Funktionen, die außerhalb der Reichweite von JavaScript liegen und über einen Webbrowser ausgeführt werden (Ionic, 2020).

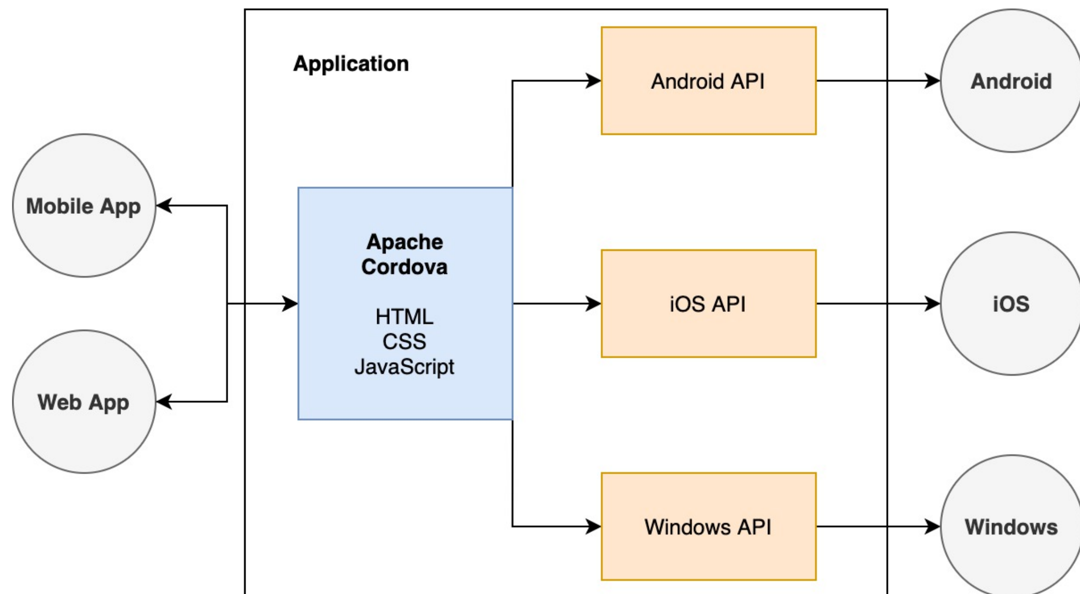

Abbildung 3.2: Architektur von Ionic (Quelle: vgl. Ionic, 2020)

Verglichen mit dem React Native-Framework, bei dem der JavaScript-Code als Bridge fungiert und der Code die nativen Oberflächenelemente steuert, laufen bei Ionic die UI-Komponenten, die von der Applikation benutzt werden, wie in Abbildung 3.2 ersichtlich, tatsächlich plattformübergreifend. In einer mobilen Anwendung werden diese Komponenten in einem Webview-Container ausgeführt. In einer Progressive Web-App werden sie im Browser ausgeführt. Und in einer nativen Anwendung für den Desktop würden diese in einem Desktop-Container wie Electron laufen (Ionic, 2020).

3.3 Xamarin

Xamarin ist eine Open-Source-Plattform zur Erstellung von Anwendungen für iOS, Android und Windows mit C#. Das Framework ist eine Abstraktionsschicht, die die Kommunikation von gemeinsam genutzten Code mit dem zugrunde liegenden Code der Plattform verwaltet. Xamarin läuft in einer verwalteten Umgebung, die Vorteile wie Speicherzuweisung und Garbage Collection bietet (Bennett, 2018).

Mit Xamarin können Anwendungen plattformübergreifend genutzt werden. Dieses Vorgehen ermöglicht es, wie bei den zuvor genannten Frameworks, die gesamte Geschäftslogik in einer einzigen Sprache zu schreiben oder den vorhandenen Anwendungscode wiederzuverwenden und die native Leistung, das Erscheinungsbild und das Gefühl auf jeder Plattform zu erreichen (Bennett, 2018).

Xamarin-Anwendungen können auf dem Windows- oder Mac-Betriebssystem geschrieben und in native Anwendungspakete kompiliert werden, wie z. B. eine .apk-Datei auf Android oder eine .ipa-Datei auf iOS (Bennett, 2018).

3.3.1 Architektur

React Native und Ionic verwenden in der Regel HTML und JavaScript. Mithilfe dieser Frameworks werden Anwendungen wie beispielsweise eine Webseite für eine mobile Anwendung unter Verwendung von JavaScript-Bibliotheken entwickelt. Danach wird die Webseite in einen Container gepackt, der die Funktionsweise einer nativen Anwendung vermittelt (Microsoft, 2020).

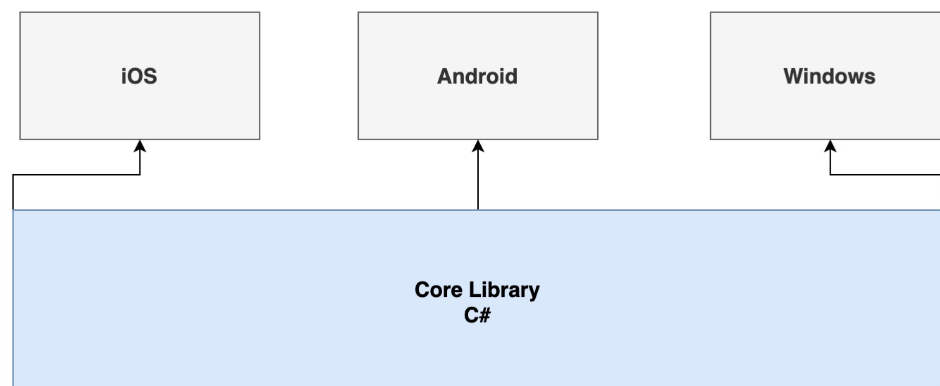

Abbildung 3.3: Architektur von Xamarin (Quelle: vgl. Microsoft, 2020)

Xamarin benutzt eine einzige Sprache C# mit einer einzigen Laufzeit, die auf drei mobilen Plattformen (Android, iOS und Windows) funktioniert. Mit der C#-Code-Basis, die Zugriff auf alle Funktionen ermöglicht, die für das native SDK verfügbar sind,

können native Benutzeroberflächen speziell für jede Plattform entworfen werden. Diese Funktionalität ist mit der Abbildung 3.3 dargestellt. Der Umfang der Code-Wiederverwendung hängt davon ab, wie viel Code im Kern verbleibt und wie viel in die Benutzerschnittstelle wandert (Microsoft, 2020).

3.4 Vergleich der Frameworks

Um ein Framework auswählen zu können, mit dem eine Applikation für den späteren Vergleich umgesetzt wird, müssen die zuvor erwähnten Technologien bewertet werden. In diesem Kapitel werden React Native, Ionic und Xamarin anhand der nachfolgenden Kriterien analysiert und verglichen.

3.4.1 Google Search Trends

Google Search Trends ist ein kostenloses Tool, das die relative Beliebtheit von Suchanfragen entsprechend der Gesamtzahl der Suchanfragen bei Google in einer bestimmten Zeit und an einem bestimmten Ort anzeigt. Es nimmt Echtzeit- und historische Datenproben von Google-Suchanfragen in einem bestimmten Zeitraum und stellt sie, in für menschlich lesbare Formate, dar (Google, 2020d).

Die Abbildung 3.4 zeigt die weltweiten Suchanfragen von React Native, Ionic und Xamarin im Zeitraum von September 2019 bis September 2020. Die Zahlen auf der Y-Achse stellen das Suchinteresse in Bezug auf den höchsten Punkt auf dem Diagramm für die gegebene Region und Zeit dar. Ein Wert von 100 ist die höchste Popularität für den Begriff. Ein Wert von 50 bedeutet, dass der Begriff halb so populär ist. Ein Wert von 0 bedeutet, dass es nicht genügend Daten für diesen Begriff gab (Google, 2020d).

Es ist zu erkennen, dass das Interesse an Xamarin und Ionic deutlich weniger ist als an React Native. Außerdem zeigt der Graph 3.4, dass die Popularität von React Native im Jahr 2019 - 2020 sich konstant im oberen Drittel bewegt. Das weltweite Interesse an React Native ist am größten von den ausgewählten Frameworks.

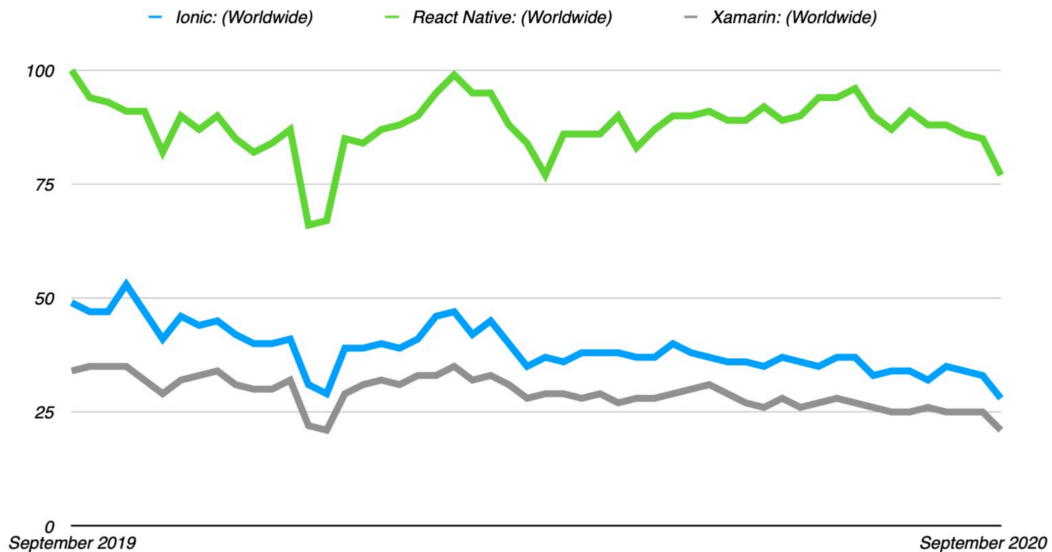

Abbildung 3.4: Vergleich von Ionic, React Native & Xamarin mit den Google Search Trends

3.4.2 Performance

Sowohl C# als auch JavaScript werden nicht in den nativen Code der Ziel-CPU kompiliert. C# wird in Bytecode kompiliert, und JavaScript wird interpretiert. Somit ergeben sich Performance Probleme (Clark, 2017).

Es gibt zwei Möglichkeiten, die Leistung solcher Sprachen zu verbessern: JIT und AOT. Just-in-Time-Kompilierung (JIT) ist der Prozess der Kompilierung eines Bytecodes oder eines Quellcodes in den nativen Code zur Laufzeit. Daher wird ein Stück Code nicht bei jeder Ausführung zur Laufzeit interpretiert, sondern wird nur einmal zur Laufzeit interpretiert, und bei jeder zusätzlichen Ausführung läuft ein schneller nativer Code. Die Ahead-of-Time-Kompilierung (AOT) ist derselbe Prozess, der vor dem Start der Anwendung zur Zeit der Kompilierung durchgeführt wird (Clark, 2017).

3.4.2.1 Xamarin

C# kann entweder JIT-kompiliert oder AOT-kompiliert sein. Da eine Kompilierung mit JIT für iOS nicht möglich ist, kompiliert Xamarin die Anwendung standardmäßig für die iOS Plattform AOT. Ein weiterer großer Vorteil von Xamarin ist, dass es native Widgets zur Darstellung der Benutzeroberfläche verwendet. Die Verwendung der nativen Methoden zum Layout der Benutzeroberfläche ist genauso schnell wie bei einer Anwendung, die nativ geschrieben wurde (Microsoft, 2017).

3.4.2.2 React Native

React Native hat keine Möglichkeit um fehlendes JIT auf iOS zu ersetzen und greift dadurch auf die Interpretation von JavaScript-Code zurück. Intern verwendet es den von iOS bereitgestellten JavaScriptCore. Obwohl JavaScriptCore auch auf Android verwendet wird, kann JIT auf dieser Plattform verwendet werden (cruxlab, 2020).

Das Framework bietet eine benutzerdefinierte XML-basierte Syntax für das Layout der Benutzeroberfläche und benutzerdefinierte Stylesheets. Beide sind in den Code eingebettet. Das benutzerdefinierte React Native-DOM ist schneller als das HTML-DOM, und Stylesheets sind standardmäßig nicht kaskadiert, was sie ebenfalls schneller macht. Es hat eine benutzerdefinierte Hierarchie von Widgets, die nicht in HTML übersetzt werden muss. Trotzdem muss React Native das DOM parsen, um native Widgets zu erstellen (cruxlab, 2020).

3.4.2.3 Ionic

Ionic verwendet eine WebView für die Darstellung und diese unterstützt JIT auf iOS. JIT auf Android wird standardmäßig unterstützt. Ionic basiert auf dem Angular-Framework und Komponenten davon müssen ebenfalls kompiliert werden. JIT- und AOT-Compiler können verwendet werden, aber das Endresultat ist kein nativer Code, sondern JavaScript-Code. Die Benutzeroberfläche wird mithilfe von HTML und CSS erstellt. Ionic bietet eine Reihe von Angular-Komponenten, die die Widgets der Plattform imitieren, um die Anwendung nativ aussehen zu lassen (cruxlab, 2020).

3.4.2.4 Resultat

Xamarin erzielt die beste native Performance. Dadurch dass auf iOS und Android AOT- und JIT-Kompilierung ohne Interpreter unterstützt wird, ergeben sich Vorteile hinsichtlich der Performance, verglichen mit React Native und Xamarin.

3.4.3 GitHub Stars

Neben einem git-basierten Versionskontrollsystem integriert GitHub mehrere soziale Funktionen. Insbesondere können GitHub-Benutzer Projekte mit Sternen bewerten. Diese Sterne sind eine Metrik für das Interesse der Community an einem Projekt (Borges & Tulio Valente, 2018).

Dadurch, dass Xamarin über kein Open-Source-Repository auf GitHub verfügt, wird stellvertretend das populärste Xamarin-Repository Xamarin.Forms für den Vergleich verwendet. Folgende Daten aus Tabelle 3.1 zeigen die GitHub-Stars der zuvor genannten Repositories und wurden im September 2020 erhoben:

Framework	GitHub Stars
React Native	90.100
Ionic	41.700
Xamarin.Forms	4.800

Tabelle 3.1: GitHub Stars der Repositories von React Native, Ionic & Xamarin.Forms

Mit 90.100 Sternen, ist React Native deutlich vor Ionic mit 41.700 Sternen und Xamarin.Forms, welches abgeschlagen mit 4.800 Sternen am wenigsten Interesse der Community aufweist. Dieses Ergebnis wird durch die Google Search Trends bestätigt.

3.4.3.1 Resultat

Laut der Recherche aus den vorhergehenden Kapiteln zeigt sich, dass React Native das populärste Framework, aus den untersuchten Technologien dieser Arbeit, mit der größten Community ist. Auch wenn Xamarin bessere Performance-Ergebnisse liefert, spiegelt React Native das weltweite Interesse an einem Vergleich mit Flutter besser wider. Aus diesem Grund wird das React Native-Framework für die Programmierung eines Prototyps herangezogen.

4 Flutter

Flutter ist ein Framework von Google zur Erstellung plattformübergreifender mobiler Anwendungen für iOS und Android. Laut der offiziellen Website (www.flutter.io), zielt Flutter darauf ab, die Entwicklung so einfach, schnell und produktiv wie möglich zu gestalten (Mainkar & Giordano, 2019).

Der Ursprung von Flutter liegt bei Google. Ursprünglich war Flutter ein Experiment, der Entwickler bei Google, die versuchten, einige Kompatibilitätsunterstützungen aus Chrome zu entfernen, um die Software flüssiger laufen zu lassen. Nachdem viele der Kompatibilitätsprobleme entfernt wurden, stellten die Entwickler fest, dass sie etwas hatten, das 20 Mal schneller renderte als Chrome, und sahen das Potenzial dieses Ergebnisses (Mainkar & Giordano, 2019).

Google hatte ein Framework geschaffen, das direkt mit der Central Processing Unit (CPU) und der Graphics Processing Unit (GPU) kommunizierte, um es den Entwicklern zu ermöglichen, die Anwendungen so weit wie möglich anzupassen. Die erste stabile Version 1.0 von Flutter wurde am 04. Dezember 2018 von Google released (Mainkar & Giordano, 2019).

Im Zeitraum von 2018 bis 2020, der Zeitpunkt, an dem diese Arbeit verfasst wurde, wurden dem Flutter Repository auf GitHub 108.000 Sterne vergeben (Flutter, 2020a). Wenn diese Zahl, der Zahl von React Native aus Kapitel 3, in dem das populärste Cross-Plattform-Framework neben Flutter analysiert wurde, gegenübergestellt wird, ergibt sich eine positive Differenz von 17.900 für das Flutter-Framework. Somit ist Flutter in zwei Jahren zum populärsten Cross-Plattform-Framework geworden.

In diesem Kapitel wird Flutter mit seinen Bestandsteilen näher analysiert, um einen Einblick in die Funktionsweise des Frameworks zu bekommen.

4.1 Dart

In Flutter werden alle Anwendungen in der Programmiersprache Dart geschrieben. Dart ist eine Programmiersprache, die von Google entwickelt wurde und wird von

Google intern häufig genutzt. Es wurde bewiesen, dass es mit dieser Sprache möglich ist, umfangreiche Webanwendungen wie Google AdWords zu entwickeln. Dart wurde ursprünglich als Ersatz und Nachfolger von JavaScript entwickelt. Daher implementiert es die meisten wichtigen Merkmale des nächsten JavaScript-Standards ES7. Um jedoch Entwickler anzuziehen, die nicht vertraut sind mit JavaScript, hat Dart eine Java-ähnliche Syntax (Google, 2020c).

4.2 Architektur

Für das zugrundeliegende Betriebssystem werden Flutter Anwendungen auf die gleiche Weise verpackt wie jede andere native Anwendung. Ein plattformspezifischer Embedder bietet einen Einstiegspunkt, koordiniert mit dem zugrunde liegenden Betriebssystem den Zugriff auf Dienste wie Rendering-Oberflächen und Eingabemasken und verwaltet die Message-Events Schleife. Der Embedder ist in einer Sprache geschrieben, die für die Plattform geeignet ist, Java und C++ für Android, C++ und Objective-C für iOS und macOS sowie C++ für Windows und Linux. Mit dem Embedder kann der Flutter Code als Modul in eine bestehende Anwendung integriert werden, oder der Code kann der gesamte Inhalt der Anwendung sein. Flutter umfasst eine Reihe von Embeddern für gängige Zielplattformen, aber es gibt auch andere Embedder (Google, 2020b).

Das Herzstück von Flutter ist die Flutter-Engine, die größtenteils in C++ geschrieben ist und die Grundlage für alle Flutter Applikationen ist. Die Engine ist für die Rasterung zusammengesetzter Szenen verantwortlich, wenn ein neues Frame gemalt werden muss. Sie stellt die Low-Level-Implementierung der Core-API von Flutter zur Verfügung, einschließlich Grafiken, Textlayout, Datei- und Netzwerk-Input/Output, Unterstützung der Barrierefreiheit, Plugin-Architektur und eine Dart-Laufzeit- und Compiler-Toolchain (Google, 2020b).

Die Engine wird dem Flutter-Framework durch dart:ui verfügbar gemacht, das den zugrundeliegenden C++-Code in Dart-Klassen zusammenfasst. Die Bibliothek stellt die Primitive der untersten Ebene zur Verfügung, wie z. B. Klassen zur Steuerung von Eingabe-, Grafik- und Textdarstellungssubsystemen (Google, 2020b).

In der Regel wird jede Interaktion mit Flutter über das Flutter-Framework, welches ein modernes, reaktives, in der Dart-Sprache geschriebenes Framework, erstellt. Es umfasst eine umfangreiche Ansammlung von Plattform-, Layout- und grundlegenden Bibliotheken, die aus einer Reihe von Schichten bestehen. Diese Schichten sind in Abbildung 4.1 ersichtlich (Google, 2020c).

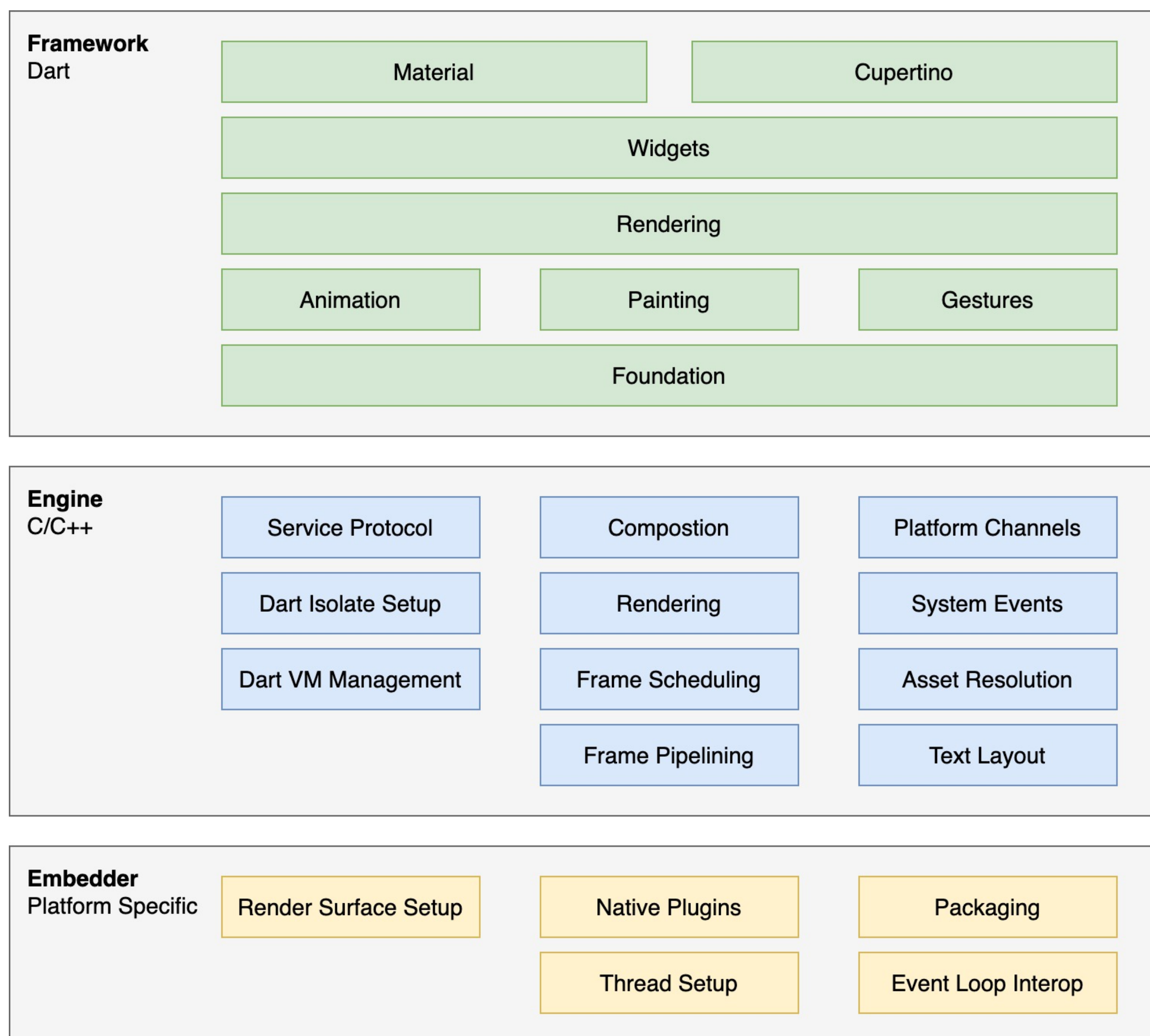

Abbildung 4.1: Architektur von Flutter (Quelle: vgl. Google, 2020b)

Das Framework interagiert mit der Flutter-Engine (in blau) über eine Abstraktionsschicht, genannt Window. Diese Abstraktionsschicht stellt eine Reihe von APIs zur Verfügung, um indirekt mit dem Gerät zu kommunizieren.

4.3 Widgets

Widgets werden unter Verwendung des Konzepts der Komponenten gebaut, welches von React inspiriert wurde. Die zentrale Idee ist, dass die UI aus Widgets gebaut wird. Widgets beschreiben, wie die UI angesichts der aktuellen Konfiguration und des aktuellen Zustands visuell aufbereitet ist. Wenn sich der Zustand eines Widgets ändert, baut das Widget seinen Zustand neu auf, welchen das Framework vom vorherigen

Zustand unterscheidet, um die minimalen Änderungen zu bestimmen, die im zugrunde liegenden Renderbaum für den Übergang von einem Zustand zum nächsten erforderlich sind (Google, 2020e).

Flutter verwendet keine nativen Systemkomponenten. Stattdessen bietet es eine Ansammlung an Widgets seines eigenen benutzerdefinierten Widget-Katalogs, der von der Engine des Frameworks gerendert und verwaltet wird.

Alles in Flutter kann mit Widgets erstellt werden. Es ist das Hauptmerkmal von Flutter, und alles, von einer einfachen Schaltfläche bis hin zu einer Animation oder Geste, wird mithilfe von Widgets erstellt. Dieser Aufbau erlaubt es, die Komposition über die Vererbung zu wählen, wodurch der Aufbau einer Applikation simpler wird. Widgets sind die Bausteine, mit der die Applikation Stück für Stück zusammengebaut wird (Mainkar & Giordano, 2019).

In nachfolgendem Snippet 4.1 wird das Container-Widget beschrieben. Es wird verwendet, um ein untergeordnetes Widget innerhalb eines übergeordneten Widgets zu enthalten (Flutter, 2020b).

```
1 Container(
2         margin: const EdgeInsets.all(10.0),
3         color: Colors.white,
4         width: 50.0,
5         height: 50.0,
6         child: Text('Hello World!')
7 );
```

Listing 4.1: Container Widget

4.3.0.1 StatefulWidgets

StatefulWidgets sind dynamische Komponenten, die einen internen Zustand zu verwalten haben. Ein StatefulWidget kann auf Zustandsänderungen reagieren und sich entsprechend ändern (Flutter, 2020b).

4.3.0.2 StatelessWidgets

StatelessWidgets bleiben gleich, auch wenn der Benutzer mit ihnen interagiert. Diese Art von Widgets haben keinen Zustand, sodass sie sich nicht entsprechend eines internen Zustands ändern können. Sie können nur auf Widget-Änderungen reagieren,

welche von einem in der Hierarchie höher stehenden Widgets, angestoßen wurden (Flutter, 2020b).

4.4 Lifecycle

Da im Flutter-Framework eine Unterscheidung zwischen Stateful und StatelessWidgets stattfindet, unterscheiden sich auch die Lebenszyklen dieser beiden Komponenten. Ein StatelessWidget kann nur einmal gezeichnet werden, wenn das Widget geladen oder gebaut wird. Ein StatelessWidget kann nicht aufgrund von Ereignissen oder Benutzeraktionen neu gezeichnet werden.

Das StatefulWidget ist veränderbar, weshalb es innerhalb seiner Lebensdauer mehrfach gezeichnet werden kann. Es ist nützlich, wenn der Bildschirm der Anwendung bei Benutzeraktionen dynamisch aktualisiert wird. Die Methode build() kann während der Lebensdauer mehrmals aufgerufen werden und jeder Aufruf kann neue oder andere Widgets basierend auf verschiedenen Parametern zurückgeben (Flutter, 2020b).

Der Flutter-Embedder ist die native Betriebssystemanwendung, die alle Inhalte von Flutter bereitstellt und als Bindeglied zwischen dem Host-Betriebssystem und Flutter fungiert. Der Embedder ist auch für den Lebenszyklus der Anwendung verantwortlich. Er greift auf die aus Kapitel 2 erwähnten nativen Callbacks zurück. Wenn das Flutter-Framework ein StatefulWidget erstellt, erzeugt es einen Objektzustand. In diesem Objekt werden alle veränderbaren Zustände für dieses Widget gehalten. Ein StatefulWidget hat laut (Napoli, 2020) den folgenden Lebenszyklus aus Abbildung 4.2:

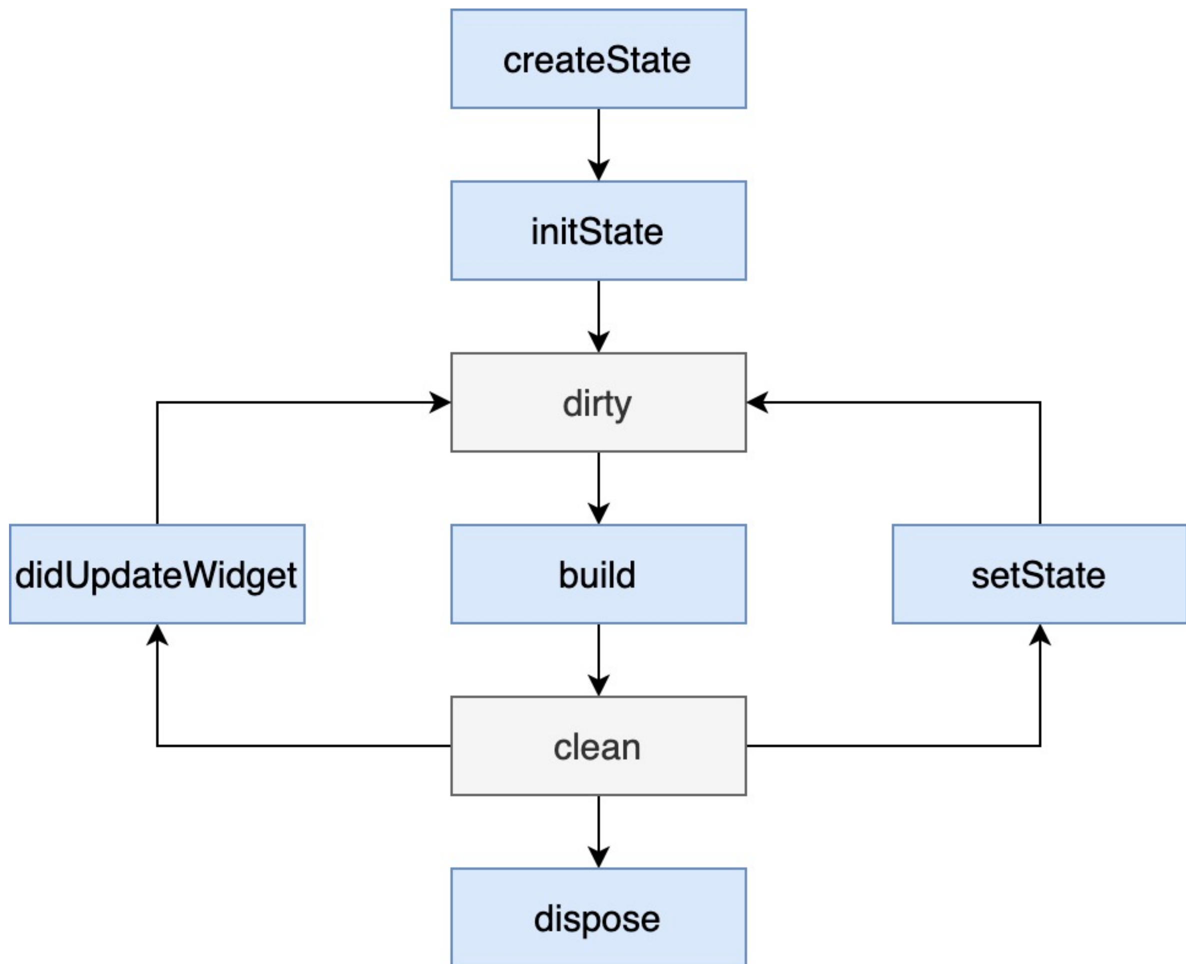

Abbildung 4.2: Lifecycle eines StatefulWidgets (Quelle: vgl. Napoli, 2020)

4.4.1 createState

Diese Funktion muss implementiert werden im Widget, da diese den State des Objekts setzt. Das Framework ruft createState immer dann auf, wenn es ein Stateful-Widget erstellt, was bedeutet, dass mehrere Stateful-Objekte mit demselben Stateful-Widget verknüpft sein können, wenn dieses Widget an mehreren Stellen in den Baum eingefügt wurde. Wenn ein StatefulWidget aus dem Baum entfernt und später wieder in den Baum eingefügt wird, ruft das Framework createState erneut auf, um ein neues State-Objekt zu erstellen, wodurch der Lebenszyklus von State-Objekten vereinfacht wird (Napoli, 2020).

4.4.2 initState

Diese Funktion wird einmal aufgerufen, wenn ein Objekt in den Widget-Baum eingefügt wird. Das Framework ruft diese Methode genau einmal für jedes State-Objekt das erstellt wird auf. In diesem Teil des Widgets sollte die gesamte Logik (wie z. B. die Initialisierung von Variablen) stattfinden, welche vor der build() Methode aufgerufen werden soll (Napoli, 2020).

4.4.3 dirty

Der State dirty zeigt an, dass das Widget in einem Zustand ist, indem es erneut gebaut werden muss. Wenn das Element in den dirty Status übergeht, wird implizit die build() Methode aufgerufen und alle Widgets, die im StatefulWidget enthalten sind, werden neu gezeichnet (Napoli, 2020).

4.4.4 build

Diese Methode wird aufgerufen, wenn das Widget in den Baum eingefügt wird oder wenn sich die Abhängigkeiten des Widgets ändern. Das Framework ersetzt den Teilbaum unterhalb dieses Widgets durch das von dieser Methode zurückgegebene Widget, entweder durch Aktualisierung des vorhandenen Teilbaums oder durch Entfernen des Teilbaums und das Erstellen eines neuen Teilbaums, je nachdem, ob das von dieser Methode zurückgegebene Widget die Wurzel des vorhandenen Teilbaums aktualisieren kann (Napoli, 2020).

4.4.5 dispose

Dieser Callback wird aufgerufen, wenn das Objekt aus dem Baum permanent entfernt wird. Nachdem dispose() aufgerufen wurde, wird das State-Objekt als entsorgt betrachtet. Diese Phase des Lebenszyklus ist endgültig und es gibt keine Möglichkeit, ein State-Objekt, welches entsorgt wurde, wieder aufzurufen (Napoli, 2020).

4.4.6 didUpdateWidget

Sobald eine Änderung einer Abhängigkeit des State-Objekts registriert wird, wird diese Methode aufgerufen. Außerdem wird diese Methode sofort nach der initState()

Methode aufgerufen. Diese Funktionalität sollte dafür verwendet werden, um Statusänderungen des Widgets zu vergleichen, bevor das Widget neu gebaut wird. Wenn das übergeordnete Widget neu aufgebaut wird und anfordert, dass diese Stelle in der Baumstruktur aktualisiert wird, aktualisiert das Framework die Widget-Eigenschaft dieses Zustandsobjekts, um auf das neue Widget zu verweisen, und ruft dann diese Methode mit dem vorherigen Widget als Argument auf (Napoli, 2020).

4.4.7 setState

Die setState() Methode wird verwendet, um Änderungen des interne States des Objekts durchzuführen. Dies führt dazu, dass das Framework plant, das UI für diesen Teilbaum neu zu zeichnen (Napoli, 2020).

4.5 Deklarative UI

Frameworks verwenden in der Regel einen imperativen Stil der UI-Programmierung. Mit diesem Stil werden manuell voll funktionsfähige UI-Entitäten (wie z. B. eine UI-View oder ein Äquivalent) konstruiert und später mithilfe von Methoden und Settern mutiert, wenn sich die UI ändert (Flutter, 2020b).

Folgendes Beispiel 4.2 zeigt den Übergang von einer View in einen neuen Status. Im imperativen Stil würde man das Objekt von ViewB abrufen und darauf Mutationen anwenden.

```
// Imperative style
b.setColor(red)
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)
```

Listing 4.2: Imperative UI

Im deklarativen Stil sind Views (wie z. B. Flutter Widgets) unveränderlich und stellen nur die Struktur dar. Um die UI zu ändern, löst ein Widget die Methode build() auf sich selbst aus und konstruiert einen neuen Widget-Teilbaum (Flutter, 2020b). Nachfolgendes Code-Snippet 4.3 zeigt eine solche deklarative UI.

```
// Declarative style
return ViewB(
  color: red,
  child: ViewC(...),
)
```

Listing 4.3: Deklarative UI

Anstatt eine alte Instanz b zu mutieren, wenn sich die Benutzeroberfläche ändert, konstruiert Flutter neue Widget-Instanzen. Das Framework verwaltet viele der Verantwortlichkeiten eines traditionellen UI-Objekts (wie z. B. die Aufrechterhaltung des Zustands des Layouts) im Hintergrund mit RenderObjects. RenderObjects bleiben zwischen den Frames bestehen, und die leichtgewichtigen Widgets von Flutter weisen das Framework an, die RenderObjects zwischen den Zuständen zu mutieren. Das Flutter-Framework erledigt den Rest (Napoli, 2020).

Die Entscheidung für deklarative statt imperativer Lösungen kann die allgemeine Code-Komplexität massiv reduzieren und viele Vorteile bieten, darunter schnellere Codierung, bessere Code-Qualität, verbesserte Lesbarkeit, weniger Tests und geringere Wartungskosten sind beispielsweise einige dieser Vorteile. Dadurch, dass Flutter den Ansatz der deklarativen UI verfolgt, bietet das Framework auch die Vorteile der deklarativen Programmierung (Napoli, 2020).

4.6 Vergleich mit anderen Frameworks

Wie der grundlegende Rendering-Prozess von den in Kapitel 3 analysierten Frameworks und Flutter funktioniert, wird in diesem Kapitel behandelt. Es deckt die Eigenheiten der verschiedenen Plattformen auf und zeigt Vor- und Nachteile der jeweiligen Technologien.

4.6.1 Native Applikationen

Native Frameworks wie Android und iOS wurden bereits im Kapitel 2 behandelt. Sie sind die stabilste Wahl für die Entwicklung mobiler Anwendungen und haben viele verfügbare Anwendungen, die gründlich getestet wurden. Das folgende Diagramm 4.3 zeigt die Funktionsweise nativer Anwendungen (Fayzullaev, 2018).

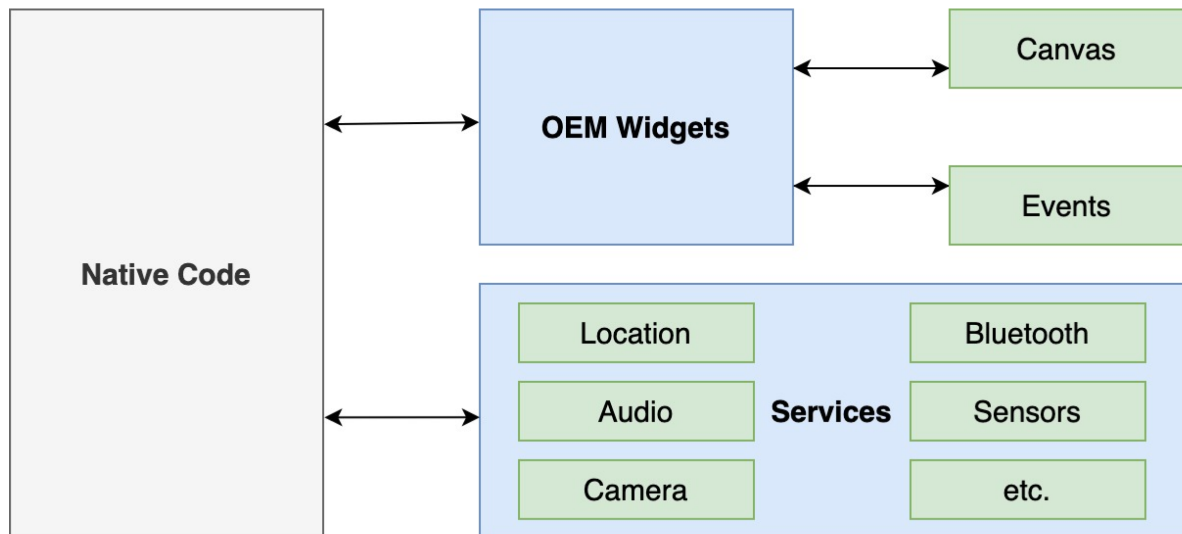

Abbildung 4.3: Prozess einer nativen Applikation (Quelle: vgl. Fayzullaev, 2018)

Wie in Abbildung 4.3 gezeigt wird, kommuniziert die App direkt mit dem System. Dies macht das native Framework zur leistungsstärksten Wahl in Bezug auf die Funktionalität. Der große Nachteil in der Entwicklung liegt hierbei in der Tatsache, dass zwei verschiedene Sprachen erlernt werden müssen. Kotlin oder Java für Android, Objective-C oder Swift für iOS. Diese Sprachen werden verwendet, um zwei verschiedene Anwendungen mit den gleichen Funktionalitäten zu schreiben. Jede Änderung muss auf beiden Plattformen dupliziert werden, und der Prozess ist möglicherweise sehr aufwendig (Fayzullaev, 2018).

4.6.2 React Native

Die Recherche aus dem Kapitel 3 ergab, dass React Native eines der populärsten Frameworks für Cross-Plattform-Entwicklung ist. React Native hängt stark von den OEM-Widgets der nativen Systeme ab. OEM-Widgets sind native UI-Komponenten des Systems. Abhängig von der Plattform, die für die Ausführung einer React Native-Anwendung verwendet wird, zeigt die App native UI-Elemente an, mit denen die Benutzer von anderen Anwendungen vertraut sind. Dieser Prozess ist in Abbildung 4.4 ersichtlich (Wu, 2018).

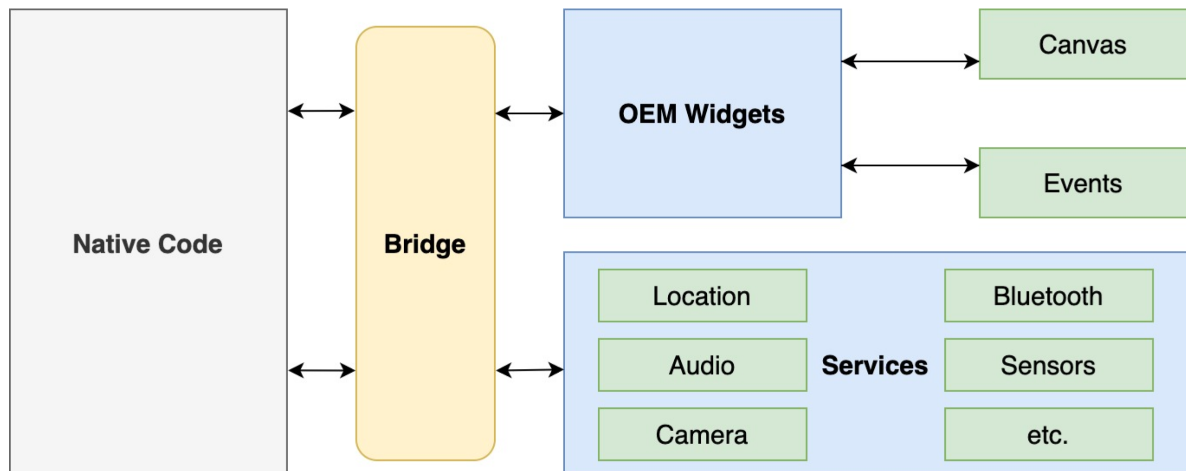

Abbildung 4.4: Prozess einer React Native-Applikation (Quelle: vgl. Wu, 2018)

React Native benutzt das Bridge-Konzept und nutzt es nicht nur für Dienste, sondern auch für die Erstellung von Widgets. Dieses Verhalten hat großen Einfluss auf die Performance. So kann beispielsweise eine Komponente während einer Animation hunderte Male gebaut werden, aber aufgrund des Bridge-Konzepts kann diese Komponente stark verlangsamt werden, weil sie nicht direkt mit der nativen Applikation kommuniziert, sondern über die Bridge. Dies könnte auch zu anderen Problemen führen, insbesondere auf Android, dem am stärksten fragmentierten Betriebssystem (Wu, 2018).

4.6.3 Flutter

Flutter kompiliert den Code gesamten Code AOT und nicht JIT, wie die JavaScript Lösungen. Außerdem entfällt das Konzept der Bridge und ist nicht auf die OEM-Plattform angewiesen. Es erlaubt es jedoch, dass benutzerdefinierte Komponenten alle Pixel auf dem Bildschirm verwenden können. Es bedeutet im Wesentlichen, dass die App auf jeder Version von Android und iOS gleich angezeigt wird. Die Funktionsweise von Flutter ist in Abbildung 4.5 beschrieben (Mainkar & Giordano, 2019).

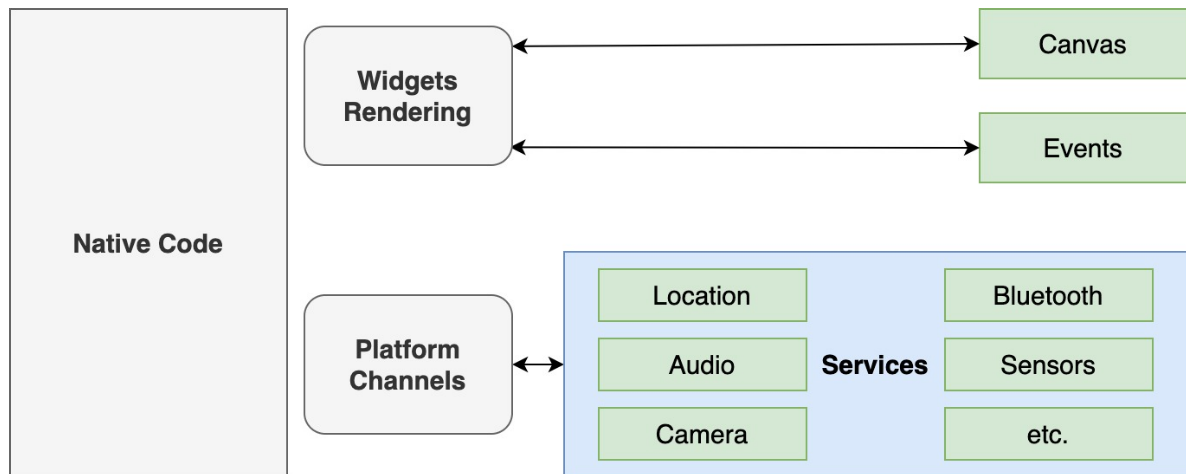

Abbildung 4.5: Prozess einer Flutter Applikation (Quelle: vgl. Mainkar & Giordano, 2019)

In der Abbildung ist der Unterschied zwischen anderen plattformübergreifenden Ansätzen und Flutter erkennbar. Wie bereits erwähnt, hat Flutter die Brücke und die OEM-Plattform eliminiert und verwendet stattdessen Widgets, um mit den Ereignissen zu arbeiten. Außerdem verwendet es Plattform-Kanäle, um die Dienste zu nutzen. Darüber hinaus ist es möglich, auf Plattform-APIs mit einem asynchronen Messaging-System zuzugreifen um eine bestimmte Android- oder iOS-Funktion zu verwenden (Mainkar & Giordano, 2019). Aus diesem Grund entfällt der Zwischenschritt der Bridge verglichen mit React Native, was wiederum zu einer besseren Performance führt.

4.6.4 Zusammenfassung

Zusammenfassend ist das Flutter-Framework ein sehr leistungsstarkes SDK, das bereits in der Popularität alle anderen Cross-Plattform-Frameworks überholt hat. Es verfügt über die Tools und Bibliotheken, die bei der Entwicklung von Mobil-, Web- oder Desktop-Anwendung helfen. Darüber hinaus arbeiten sowohl Google als auch die Community ständig an neuen Funktionen, um das SDK zu erweitern. Außerdem liefert Flutter mit dem deklarativen Programmierstil eine sehr niedrige Eintrittsbarriere in das Erlernen des Frameworks und bietet eine Lösung für den Performanceverlust des Rendering Prozesses, von JavaScript basierten Frameworks.

5 Testszenario

Das Testszenario gilt für alle Plattformen. Als erster Screen wird ein Home-Screen angezeigt, der einen Button enthält, welcher auf eine Liste navigiert. In dieser Liste sind Einträge mit Text und Bild ersichtlich. Die Anzahl der Elemente variiert zwischen den verschiedenen Versuchen, um zu zeigen, wie sich die Menge, der zu verarbeitenden Datensätze, auf die Performance der Applikation auswirkt.

Die Datensätze werden von einer Firebase-Schnittstelle über das Internet geladen. Firebase ist ein Backend as a Service (BAAS) von Google. Das bedeutet, es ist ein Cloud-Computing-Service, welcher als Zwischenschicht dient und es ermöglicht, die API mit Cloud-Diensten zu verbinden. Für den Anwendungszweck der Applikation dieser Arbeit, wird der Cloud Firestore verwendet. Über den Firestore, werden die Daten in einem Format, welches Dokumenten ähnelt, bereitgestellt. Die Architektur der Applikation ist im Diagramm 5.1 abgebildet.

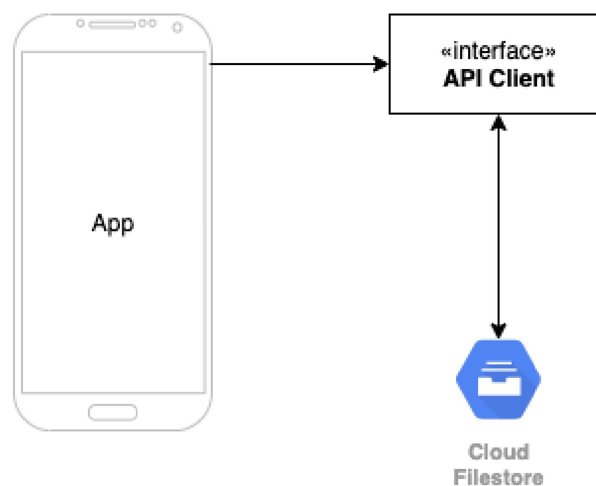

Abbildung 5.1: Architektur des Prototypen

Der Listeneintrag der Applikation navigiert auf einen Detail-Screen weiter. Auf diesem Screen wird ein Bild, mit einer höheren Auflösung und einer größeren Dateigröße angezeigt. Zusätzlich dazu, werden Text-Einträge angezeigt, die von der API geladen wurden. Um eine Vergleich über den Zugriff auf Hardware-Ressourcen der nativen und Cross-Plattform-Frameworks zu erhalten, ist eine Kamera- und Galerie-Funktion auf diesem Screen implementiert.

Das bedeutet, es wird auf die Kamera des Geräts zugegriffen, um ein Bild zu erstellen. Dieses Bild wird danach im Detail-Screen angezeigt. Dieselbe Funktionalität gilt für die Galerie-Funktion. Es wird auf die Galerie des Gerätes zugegriffen und das ausgewählte Bild wird danach im Detail-Screen angezeigt. Bei diesem Zugriff auf die Hardware ist vor allem die Auswirkung auf die Performance der Applikation interessant.

Dieser Ablauf ist in folgendem Diagramm 5.2 dargestellt.

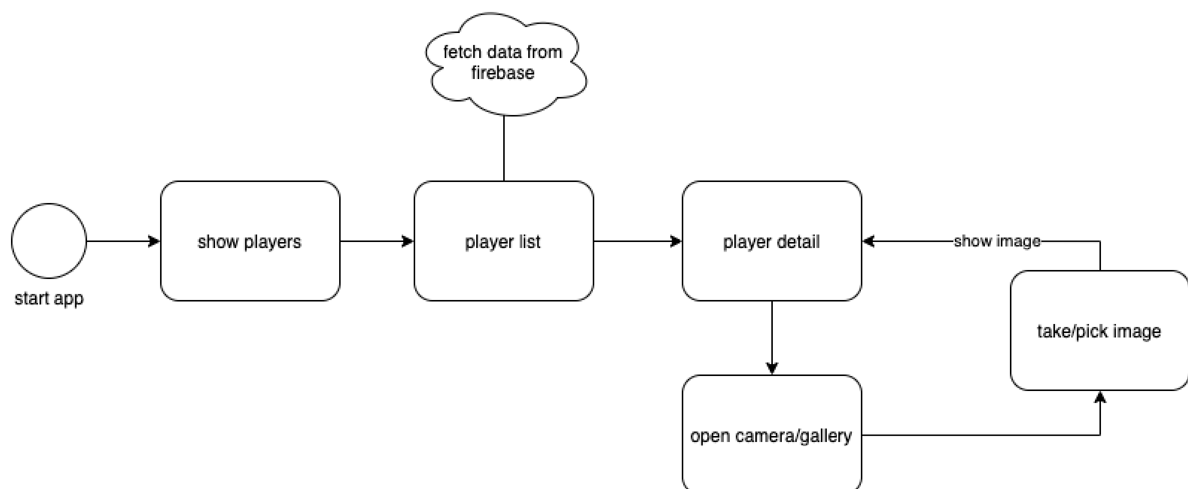

Abbildung 5.2: Ablauf des Prototypen

5.1 Bewertungsmatrix

Um einen Vergleich, zwischen den Prototypen der jeweiligen Plattform, ziehen zu können, ist es nötig eine Bewertungsmatrix anzufertigen. In diesem Kapitel wird eine solche Matrix angefertigt, die später verwendet wird, um die Resultate zu vergleichen.

5.1.1 Kennzahlen

Eine Kennzahl kann als ein Maß definiert werden, mit dem die Leistung einer Tätigkeit gemessen werden kann. In diesem Fall liegt der Fokus auf Kennzahlen, welche die technische Performance einer mobilen Applikation messen. Die folgenden Kennzahlen werden für die implementierten Applikationen erhoben um die Ergebnisse vergleichen zu können (Parmenter, 2015).

5.1.1.1 Initiale Ladezeit

Die Initiale Ladezeit definiert die Zeit, die vom Klick des App-Icons bis zur Anzeige des Home-Screens vergeht. In dieser Zeit findet die Initialisierung der Applikation statt. Es werden im Hintergrund die benötigten Ressourcen geladen, damit sie der Applikation zur Verfügung stehen, wenn sie benötigt werden.

5.1.1.2 Frame Rate

Mit der durchschnittlichen Frame Rate wird gemessen, wie stabil die Bildrate über einen bestimmten Zeitraum ist. So kann beispielsweise der Einfluss einer Animation oder von rechenintensiven Algorithmen auf die Frame-Rate gemessen werden. Die Applikationen für den Test sind für eine Bildrate von 30 FPS ausgelegt, das heißt auch, wenn die Applikation mehr FPS darstellen könnte, werden maximal 30 angezeigt, um die Batterie zu entlasten.

5.1.1.3 CPU-Auslastung

Die CPU-Auslastung ist der Prozentsatz der gesamten CPU-Kapazität des Geräts, der von der Anwendung in dem gemessenen Zeitintervall benutzt wird. Die CPU wird für alle Prozesse am Gerät verwendet.

5.1.1.4 Memory-Auslastung

Der Parameter für die Speichernutzung spiegelt die Größe des Random-Access Memory (RAM) wider, der von der Anwendung benötigt wird.

5.2 Testsoftware

Um denselben Ablauf für alle Applikationen zu garantieren, werden verschiedene Software-Tools benutzt. In diesen ist es möglich, die Werte der Kennzahlen zu bestimmen. Dieses Kapitel beschreibt die eingesetzten Werkzeuge.

5.2.1 Apptim

Apptim ist eine Software, die es ermöglicht, Anwendungen zu testen und die Leistung zu analysieren. Sie misst die Renderzeiten von Apps, den Stromverbrauch, die Ressourcennutzun und Abstürze auf Android- und iOS-Geräten. In dieser Arbeit wird Apptim benutzt, um genaue diese Werte aus den Applikationen zu bestimmen. Dazu ist es lediglich notwendig die Testgeräte mit dem Notebook zu verbinden, um das Tracking der Daten zu ermöglichen.

5.2.2 Xcode Instruments

Xcode Instruments ist ein Entwicklertool, das kostenlos mit Xcode geliefert wird. Es enthält viele nützliche Tools zur Überprüfung und Verbesserung von Anwendungen. Es ist ein leistungsstarkes und flexibles Leistungsanalyse- und Testwerkzeug. Um die Aussagekräftigkeit der Resultate, die aus der Software Apptim geliefert werden, zu überprüfen, werden die iOS-Applikationen zusätzlich mit Xcode Instruments getestet.

5.2.3 Android Studio

Android Studio bietet Profiling-Tools zur Aufzeichnung und Visualisierung der Renderingleistung, Rechenleistung, Speicherleistung und Akkuleistung von Anwendungen. Auch die Android-Applikationen werden auf dieselbe Weise getestet wie die iOS-Applikationen, welche bereits im Kapitel 5.2.2 beschrieben wurden. Lediglich mit der Änderung, dass Android Studio mit den Profiling-Tools verwendet wird für den Vergleich.

5.3 Testgeräte

Um die Applikationen auf physischen Endgeräten, anstatt einer simulierten Umgebung, zu testen, wurde für jede Plattform ein Gerät ausgewählt. Nacholgend ist die Hardware der eingesetzten Geräte beschrieben.

5.3.1 iOS

Für die iOS-Plattform wurde ein iPhone XS gewählt. Das Gerät weist die Spezifikationen aus folgender Tabelle auf 5.1:

CPU-Architektur	Apple A12 Bionic (arm64e)
iOS-Version	14.2
Screen resolution	2436x1125
Display density	458
RAM	4GB DDR4

Tabelle 5.1: iOS Testgerät

5.3.2 Android

Für die Android-Plattform wurde ein OnePlus A6003 gewählt. Das Gerät weist die Spezifikationen aus folgender Tabelle auf 5.2:

CPU-Architektur	arm64-v8a
Android-Version	9
Screen resolution	1080x2280
Display density	420dpi
RAM	8GB

Tabelle 5.2: Android Testgerät

5.4 Testszenario

Die Applikationen werden auf den beiden Geräten, dem iPhone XS und dem OnePlus A6003, vorinstalliert. Das bedeutet, es sind auf jedem Gerät drei Applikationen. Jeweils eine native Applikation, eine React Native-Applikation und eine Flutter Applikation. Mit der Testsoftware Apptim erfolgt ein Testdurchlauf für jede Applikation. Die Ergebnisse werden mit den Ergebnissen aus Xcode Instruments und Android Studio verglichen und in einem Excel-Sheet dokumentiert, welches für den späteren Vergleich herangezogen wird. Die Veränderung der Anzahl der zu verarbeitenden Elemente, findet über Firebase statt und benötigt aus diesem Grund keine Veränderung im Source-Code der Applikationen.

Um die Bilder zu generieren wird der freie Dienst Lorem Picsum verwendet. Über diesen Dienst ist es möglich, zufälliger Bilder in bestimmter Auflösung über das Internet zu laden. Um das zu erreichen, wird lediglich ein Aufruf eines Links benötigt. Hierbei wird die Funktion Seed verwendet. Diese liefert ein statisches Bild basierend auf einer Zahl zurück. Dieser Seed ist fortlaufend in der Applikation und garantiert, dass keine Bilder im Cache des Systems landen. Somit kann garantiert werden, dass jedes Bild geladen wird und jede Applikation dieselbe Funktionalität aufweist.

Nachdem alle Daten erhoben wurden und die Tests abgeschlossen sind, wird der durchschnittliche Wert aller Tests für den Vergleich herangezogen.

5.5 Testablauf

Für das Testen der Applikation wird ein manueller Test auf jeder Applikation ausgeführt. Um zu garantieren, dass auf allen Applikationen dieselbe Funktionalität getestet wird, wird ein klarer Testablauf definiert. Folgende Aktionen werden sequentiell für den Test ausgeführt:

1. Ein neuer Testfall wird mithilfe von der Software Apptim erstellt.
2. Es wird gewartet bis die Applikation startet.
3. Mit einem Klick auf den 'Playerlist' Button wird auf die Listenansicht navigiert.
4. Auf der Listenansicht wird auf das Ende der Liste navigiert.
5. Es wird gewartet bis das Bild des letzten Listeintrags geladen wurde.
6. Danach wird auf den Listeneintrag gedrückt, um in die Detailansicht zu navigieren.
7. In der Detailansicht wird gewartet bis das Bild geladen wurde.
8. Mit der Kamera-Funktion wird ein Bild erstellt und angezeigt.
9. Mit der Galerie-Funktion wird ein Bild ausgewählt und angezeigt.
10. Mit der Navigation in der Navigationsleiste wird zurücknavigiert.
11. Auf der Listenansicht wird bis zum ersten Listeneintrag navigiert.

12. Mit der Navigation in der Navigationsleiste wird zurück auf dem Home-Screen navigiert.

13. Der Test wird mithilfe der Software Apptim beendet.

Nachdem diese Schritte durchgeführt wurden, werden die Ergebnisse von der Software Apptim in ein Excel-Dokument übertragen. Danach werden dieselben Schritte mit einer höheren Anzahl an Elementen wiederholt.

6 Prototyp

Im Zuge dieser Arbeit wurden vier Prototypen angefertigt die mit demselben Design-Pattern und Umfang umgesetzt wurden. In diesem Kapitel wird die Implementierung näher beschrieben.

6.1 Plattformen

Um den Vergleich zwischen nativen und Cross-Plattform-Technologien zu ermöglichen, wurden Applikationen für jede Plattform erstellt. In diesem Fall ergeben sich sechs Prototypen:

- native iOS-Applikation
- native Android-Applikation
- React-Native iOS-Applikation
- React-Native Android-Applikation
- Flutter iOS-Applikation
- Flutter Android-Applikation

6.2 Konzept

Die Implementierungen auf den jeweiligen Plattformen differenzieren sich und unterliegen unterschiedlichen Konzepten. So wurde beispielsweise für die Implementierung der Android-Applikation Kotlin anstatt Java verwendet und für iOS Swift-UI anstatt Swift. Auf die Performance sollte das jedoch wenig Auswirkungen haben,

da der Compiler den Swift-UI-Code in das gleiche Format bringt, wie beispielsweise einen Swift-Code.

Als Design-Pattern wurde das MVVM-Pattern verwendet. Model-View-View-Modell (MVVM) ist ein Software-Architektur-Pattern, das die Trennung der Entwicklung der grafischen Benutzeroberfläche (der View) von der Entwicklung der Geschäftslogik oder der Back-End-Logik trennt, so dass die View nicht von einer bestimmten Modellplattform abhängig ist. Das View-Modell ist dafür verantwortlich, die Datenobjekte des Modells so zu konvertieren, dass die Objekte leicht verwaltet und präsentiert werden können. In dieser Hinsicht ist das View-Modell mehr Modell als View und übernimmt die meisten, wenn nicht sogar die gesamte Anzeigelogik der View (Garofalo, 2011).

Um Änderungen an das UI weiterzuleiten, wird die Bibliothek ReactiveX verwendet. Reaktiv bedeutet, auf eine Situation zu reagieren. Reaktives Programmieren ähnelt dieser Definition, d. h. das Schreiben von Code, der auf Veränderungen reagiert. Es handelt sich dabei um eine Art der Programmierung mit asynchronen Datenströmen, Anwendungen und Schnittstellen, die dynamisch auf Datenänderungen reagieren (Maglie, 2016). Das heißt, sobald Daten von der API geladen werden und sich der Datenstrom ändert, ändert sich auch das UI. Die Kombination aus ReactiveX und MVVM ist in Abbildung 6.1 ersichtlich.

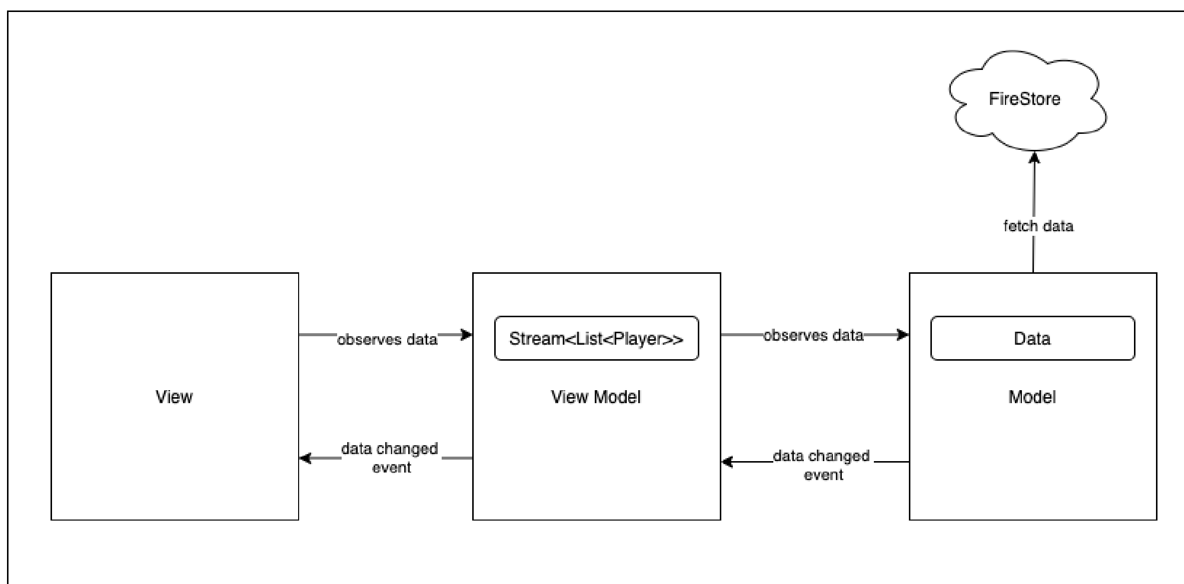

Abbildung 6.1: MVVM-Pattern mit ReactiveX

6.3 Applikationsstruktur

Die Applikationsstruktur wurde über alle Prototypen hinweg versucht auf dieselbe Art zu implementiert. Jedoch gibt es plattformspezifische Abweichungen, die eingehalten werden mussten. Um den Ablauf und den Aufbau der Prototypen genauer zu erklären, wird die Implementierung der Flutter-Applikation verwendet. Folgende Abbildung 6.2 zeigt die Struktur der Dateien:

```
lib
├─ models
│  └─ player.dart
├─ networking
│  └─ api_client.dart
├─ repositories
│  └─ player_repository.dart
├─ UI
│  ├─ player_detail
│  │  ├─ player_detail.dart
│  │  └─ player_detail_view_model.dart
│  └─ player_list
│     ├─ player_list.dart
│     └─ player_list_view_model.dart
├─ util
│  └─ permission_wrapper.dart
├─ main.dart
└─ di.dart
```

Abbildung 6.2: Flutter Dateistruktur

Die nachfolgenden Kapitel beschreiben die wichtigsten Funktionalitäten und Konzepte der Dateien im Detail. Hierbei wird nicht der ganze Code der Dateien abgebildet sondern nur die wichtigsten Funktionalitäten. Der genaue Code liegt als komprimiertes Archiv oder als Verlinkung auf ein GitHub-Repository der Arbeit bei.

6.3.1 Player

Die Player-Klasse ist das Datenmodell mit allen Attributen eines Spielers. Für die Initialisierung wird eine Factory verwendet. Dies ermöglicht die spätere Umwandlung einer JSON Response in das Player-Objekt. Der Code der Klasse ist in Listing 6.1 ersichtlich:

```
1  abstract class Player with _$Player {
2    factory Player({
3      String description,
4      String firstName,
5      String lastName,
6      String profileImg,
7      int ranking,
8      double winRatio,
9    }) = _Player;
10
11   factory Player.fromJson(Map<String, dynamic> json) =>
       _$PlayerFromJson(json);
12 }
```

Listing 6.1: player.dart

6.3.2 ApiClient

Der ApiClient lädt die Daten der Firebase-Schnittstelle. Für diese Kommunikation wird das Flutter-Package Dio verwendet. Über dieses Package ist es möglich die Response direkt in ein Objekt vom Typ Player zu transformieren. Listing 6.2 zeigt den ApiClient.

```
1  abstract class ApiClient {
2    factory ApiClient(Dio dio, {@required String baseUrl}) =
       _ApiClient;
3
4    @GET('/players.json')
5    Future<List<Player>> getAllPlayers();
6  }
```

Listing 6.2: api_client.dart

6.3.3 PlayerRepository

Das PlayerRepository ist die einzige Klasse der Applikation, die mit dem ApiClient kommuniziert und die Daten im Speicher behaltet, um diese den Rest der Applikation bereitzustellen. Die _playerList-Variable mit dem Datentyp BehaviorSubject sendet somit jede Veränderung der Liste an die Variable playerList weiter, auf welche die PlayerList und PlayerDetail Klassen zugreifen. Die ganze Applikation benutzt dadurch einen einzigen Datenstand. Dieses Verhalten ist in Listing 6.3 abgebildet.

```
class PlayerRepository {
  final _playerList = BehaviorSubject<List<Player>>();

  Stream<List<Player>> get playerList => _playerList;

  Future<List<Player>> getPlayers() async {
    final response = await apiClient.getAllPlayers();
    _playerList.add(response);
    return _playerList.value;
  }

  Player getPlayer(int id) {
    return _playerList.value[id];
  }
}
```

Listing 6.3: player_repository.dart

6.3.4 PlayerDetail

Das PlayerDetailViewModel wird mit einer PlayerId von der PlayerList initialisiert. Aufgrund dieser ID wird der entsprechende Spieler aus der Liste im PlayerRepository gesucht und der View bereitgestellt. Das Listing 6.4 zeigt das PlayerDetailViewModel.

```
class PlayerDetailViewModel {
  final playerId;

  PlayerDetailViewModel(this.playerId);

  Player get player => playerRepository.getPlayer(playerId);
```

```
7 }
```

Listing 6.4: player_detail_view_model.dart

Im PlayerDetail Screen werden die zusätzlichen Attribute des Player-Objekts angezeigt. In dieser Datei, welche in Listing 6.5 dargestellt ist, findet auch der Zugriff auf die Kamera bzw. auf die Galerie des Geräts statt. Dadurch, dass Flutter diese Funktionalitäten bereits liefert, muss nur eine ImagePicker Klasse initialisiert werden (Zeile 3) und diese kann verwendet werden, um Bilder von der Kamera aufzunehmen oder aus der Galerie auszuwählen (Zeile 22 & 28).

Bevor dies geschehen kann, müssen die Berechtigungen überprüft und falls sie nicht vorhanden sind, eingefordert werden, um die Funktionalität zu benutzen. Dies wird mithilfe der PermissionWrapper-Klasse ermöglicht (Zeile 21 & 27). In der Funktion _handleImage() wird auf das Ergebnis der Kamera bzw. der Galerie gewartet, um das Bild am Screen anzuzeigen.

```
1  class _PlayerDetailState extends State<PlayerDetail> {
2    final PlayerDetailViewModel _vm;
3    static final _imagePicker = ImagePicker();
4    String imagePath = "";
5
6    _PlayerDetailState(int playerId) : _vm =
        PlayerDetailViewModel(playerId);
7
8    ...
9
10   Row(
11     children: [
12       FlatButton(onPressed: () => _takeImageFromCamera(
            context), child: Text("Photo from Camera")),
13       FlatButton(onPressed: () => _takeImageFromGallery(
            context), child: Text("Photo from Gallery")),
14     ],
15   ),
16   Image.file(File(imagePath), fit: BoxFit.cover)
17
18   ...
19
20   void _takeImageFromCamera(BuildContext context) async {
21     if (await PermissionWrapper(context, 'permissions.
          take_image').request(PermissionWrapper.
          CAMERA_PERMISSIONS)) {
```

```
22      _handleImage(_imagePicker.getImage(source: ImageSource.
           camera));
23    }
24  }
25
26  void _takeImageFromGallery(BuildContext context) async {
27    if (await PermissionWrapper(context, 'permissions.
          take_image').request(PermissionWrapper.
          GALLERY_PERMISSIONS)) {
28      _handleImage(_imagePicker.getImage(source: ImageSource.
           gallery));
29    }
30  }
31
32  void _handleImage(Future<PickedFile> file) async {
33    final result = await file;
34    if (result != null) {
35      final sourceFile = File(result.path);
36      final directory = await
           getApplicationDocumentsDirectory();
37      final tmpFile = await sourceFile.copy(p.join(directory.
           path, p.basename(sourceFile.path)));
38      setState(() {
39        imagePath = tmpFile.path;
40      });
41    }
42  }
43 }
```

Listing 6.5: player_detail.dart

6.3.5 PlayerList

Das PlayerListViewModel initiiert den Ladeprozess der Daten über das PlayerRepository bei der Initialisierung der Klasse. Danach stellt es der PlayerList diese Daten über die Variable allPlayers bereit, wie in Listing 6.6 dargestellt ist. Dadurch, dass die Daten für den Zweck dieser Applikation nur ein einziges Mal geladen werden müssen, gibt es ansonsten keine Möglichkeit die Daten zu erneuern.

```
1 class PlayerListViewModel {
2   Stream<List<Player>> get allPlayers => playerRepository.
       playerList;
```

```
3
4   PlayerListViewModel() {
5     _init();
6   }
7
8   _init() {
9     _fetchPlayers();
10  }
11
12  _fetchPlayers() {
13    playerRepository.getPlayers();
14  }
15 }
```

Listing 6.6: player_list_view_model.dart

Die PlayerList initialisiert das PlayerListViewModel. Das StreamBuilder-Element observiert die Liste der Spieler aus dem ViewModel und zeichnet daraufhin eine Liste an Spielern. Die Anzahl der zu zeichnenden Elemente ergibt sich in diesem Fall aus der Liste des ViewModels. Dadurch, dass die Applikation mit ReactiveX arbeitet, erhält der StreamBuilder automatisch ein Update, sobald sich die Liste im ViewModel ändert. Daher ändert sich die Liste dynamisch mit dem Stream aus dem ViewModel. Jeder Listeneintrag enthält ein Klick-Event, welches auf das PlayerDetail weiterleitet. Ein Auszug aus der PlayerList-Klasse wird in Listing 6.7 gezeigt.

```
1  class _PlayerListState extends State<PlayerList> {
2    final _vm = PlayerListViewModel();
3
4    ...
5
6    Container(
7      padding: EdgeInsets.all(20),
8      child: StreamBuilder<List<Player>>(
9        stream: _vm.allPlayers,
10       initialData: [],
11       builder: (context, snapshot) {
12         return ListView.builder(
13           shrinkWrap: true,
14           itemCount: snapshot.data.length,
15           itemBuilder: (_, index) {
16             final player = snapshot.data[index];
17             return InkWell(
18               onTap: () => _pushDetail(index),
19               child: Card(
```

```
20                  child: Column(
21                    children: [
22                      Row(
23                        mainAxisAlignment: MainAxisAlignment.
                            center,
24                        children: [
25                          Image.network(player.profileImg,
26                              width: 100, height: 100),
27                        ],
28                      ),
29                      Row(
30                        children: [
31                          Text(player.firstName),
32                        ],
33                      ),
34                    ],
35                  ),
36                ),
37              );
38            },
39          );
40        },
41      ),
42    )
43
44    ...
45
46    _pushDetail(int id) {
47      Navigator.push(context, MaterialPageRoute(builder: (
          context) {
48        return PlayerDetail(playerId: id);
49      }));
50    }
51  }
```

Listing 6.7: player_list.dart

6.3.6 PermissionWrapper

Die permission_wrapper.dart Datei ist dafür verantwortlich die Berechtigungen auf den Kamera- bzw. den Galerie-Zugriff zu verwalten. In dieser Datei findet auch die Unterscheidung zwischen der Android- und iOS-Plattform statt, da Berechtigungen auf den verschiedenen Plattformen unterschiedlich behandelt werden. Die Datei wird

in dieser Arbeit nicht näher analysiert, da sie nur native Konfigurationen der Info.plist bzw. der AndroidManifest.xml Datei vornimmt.

6.3.7 Main

Diese Datei ist der Einstiegspunkt der Applikation. Die Initialisierung der Dependency Injection findet in der Zeile 2 statt. Außerdem dient die Main-Klasse als Start-Screen der Applikation und enthält einen Link auf die Liste der Spieler (Zeile 9 & 16 im Listing 6.8).

```
1  void main() {
2    init();
3    runApp(MyApp());
4  }
5
6  ...
7
8  FlatButton(
9    onPressed: _pushPlayersScreen,
10   child: Text("Players"),
11 )
12
13 ...
14
15   _pushPlayersScreen() {
16     Navigator.push(context, MaterialPageRoute(builder: (
          context) {
17       return PlayerList();
18     }));
19   }
20 }
```

Listing 6.8: main.dart

6.3.8 Dependency Injection

Für die Dependency Injection wird das Flutter Package get_it verwendet. Hierbei wird das PlayerRepository und der ApiClient für die gesamte Applikation als Singleton bereitgestellt. Dies hat den Vorteil, dass die Klassen nur ein einziges Mal initialisiert werden müssen. Der Code aus dem Listing 6.9 zeigt dies.

```
final _di = GetIt.instance;

ApiClient get apiClient => _di.get<ApiClient>();

PlayerRepository get playerRepository => _di.get<
    PlayerRepository>();

void init() {
  _di.registerSingleton<PlayerRepository>(PlayerRepository())
      ;
  _initApiClient();
}

void _initApiClient() {
  final dio = Dio(BaseOptions(connectTimeout: 60 * 1000,
      receiveTimeout: 60 * 1000));
  _di.registerSingleton<ApiClient>(ApiClient(dio, baseUrl: '
      https://master-tennis.firebaseio.com'));
}
```

Listing 6.9: di.dart

6.3.9 Firebase

Um die Anzahl der zu verarbeitenden Datensätze zu verändern, werden für die verschiedenen Tests, verschiedenen JSON-Dateien verwendet. Diese werden über Firebase importiert und ersetzen somit den Datenstand. Listing 6.10 zeigt eine solche JSON-Datei.

```
{
    "players": [
        {
            "firstName": "Hans",
            "description": "Lorem ipsum dolor sit amet,
               consectetur adipiscing elit, sed do eiusmod
               tempor incididunt ut labore et dolore magna
               aliqua.",
            "lastName": "Peter",
            "winRatio": 98,
            "ranking": 1,
```

```
9                 "profileImg": "https://picsum.photos/seed
                    /1/200/300"
10            },
11             {
12                "firstName": "Josef",
13                "description": "Lorem ipsum dolor sit amet,
                    consectetur adipiscing elit, sed do eiusmod
                    tempor incididunt ut labore et dolore magna
                    aliqua.",
14                "lastName": "Peter",
15                "winRatio": 67,
16                "ranking": 34,
17                "profileImg": "https://picsum.photos/seed
                    /2/200/300"
18            }
19        ]
20 }
```

Listing 6.10: Datensätze für den Import

7 Ergebnisse

Die Ergebnisse wurden mithilfe der, aus Kapitel 5 erwähnten Methoden und Kennzahlen, erhoben und ausgewertet. Die Zahlen wurden in ein Excel-Dokument eingetragen. Für die nachfolgenden Grafiken wurde der durchschnittliche Wert aller Testversuche berechnet und in Form eines Balkendiagramms dargestellt. Dies sind die Ergebnisse aus den Testversuchen.

7.1 CPU-Verbrauch

Abbildung 7.1 zeigt den durchschnittlichen CPU-Verbrauch der Applikationen. React Native weist, sowohl auf Android als auch auf iOS, hierbei die größte CPU-Auslastung auf mit 5,67%. Die Flutter-iOS-Applikation, mit einer durchschnittlichen Auslastung von 2,67%, schneidet nur 0,67% schlechter ab als die native SwiftUI-Applikation mit 2%.

Die Android-Flutter-Applikation weist eine durchschnittliceh Auslastung der CPU von 4,33% auf. Durchschnittlich benötigt die native Applikation 3% der CPU. Somit trägt auch auf der Android-Plattform die native Applikation am wenigsten zur CPU-Auslastung bei.

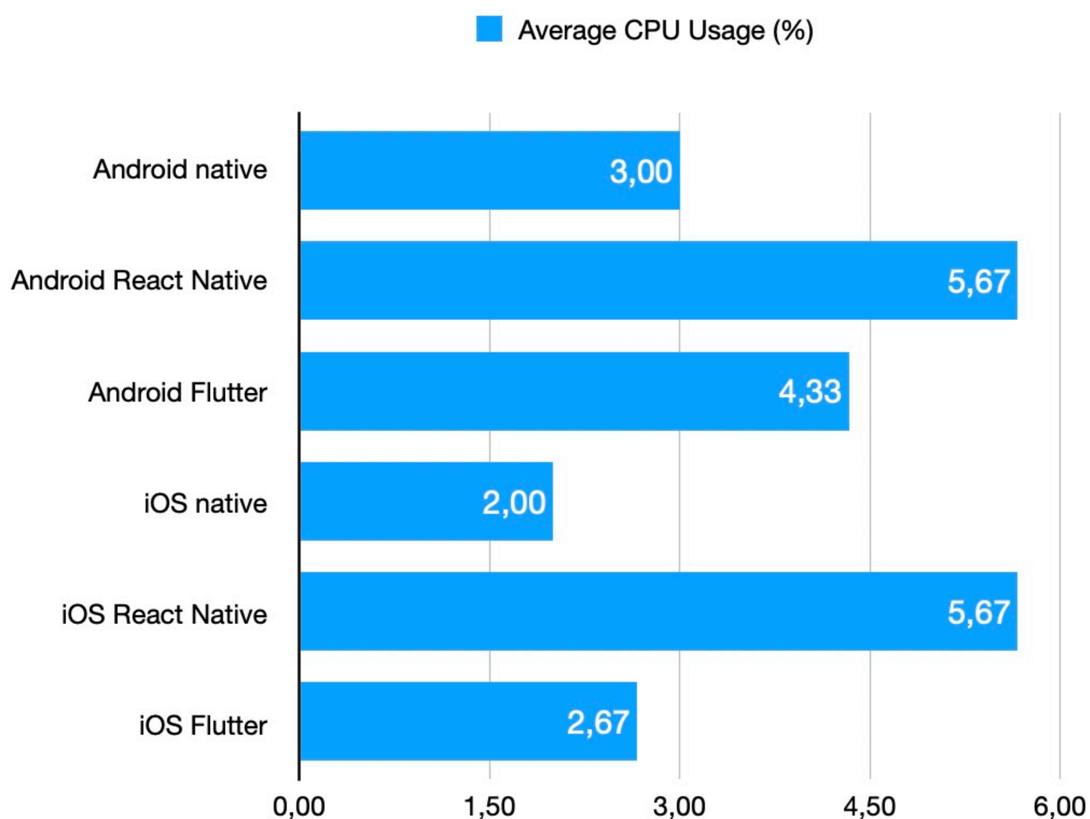

Abbildung 7.1: Durchschnittliche CPU-Auslastung

Aufgrund dieser Daten weist Flutter eine bessere Performance, in Hinsicht auf die CPU-Auslastung, als React Native auf. Die native Implementierung liefert die besten Performance-Werte. Dadurch dass Cross-Plattform-Frameworks, wie React Native,

über eine Brücke mit dem nativen Code kommunizieren, werden mehr Hardwareressourcen wie z. B. die CPU benötigt. Die bessere Flutter-Performance lässt sich durch die Eliminierung dieser Brücke erklären. Wie in Kapitel 4.6.3 beschrieben wird, arbeitet Flutter mit Widgets anstatt einer Bridge. Man kann über diese Ergebnisse nicht auf alle Cross-Plattform-Frameworks schließen, aber dadurch, dass React Native eines der populärsten Frameworks für Cross-Plattform-Entwicklung ist, geben die Ergebnisse einen guten allgemeinen Richtwert für Cross-Plattform-Frameworks.

7.2 Heruntergeladene Daten

Bei den durchschnittlich heruntergeladenen Daten, welche in Abbildung 7.2 ersichtlich sind, lädt die React Native iOS-Applikation deutliche mehr Daten aus dem Internet (3782,33 kB), als alle anderen Applikationen. Dass React Native fast zehnmal so viele Daten lädt (wie z. B. Flutter), liegt womöglich an den Debugging-Tools. Alle Applikationen wurden mit einem Profil erstellt, welches für die Entwicklung genutzt wird. Aus diesem Grund entspricht es nicht ganz einer Applikation die z. B. über den App Store zur Verfügung gestellt wird.

Da es sich bei diesem Wert sehr wahrscheinlich um eine Fehlinterpretation der Test-Software handelt und mithilfe von Xcode Instruments dieser Wert nicht verifiziert werden konnte, wird für den nachfolgenden Vergleich die React Native-Applikation der iOS-Plattform entfernt, um einen Vergleich zu ermöglichen.

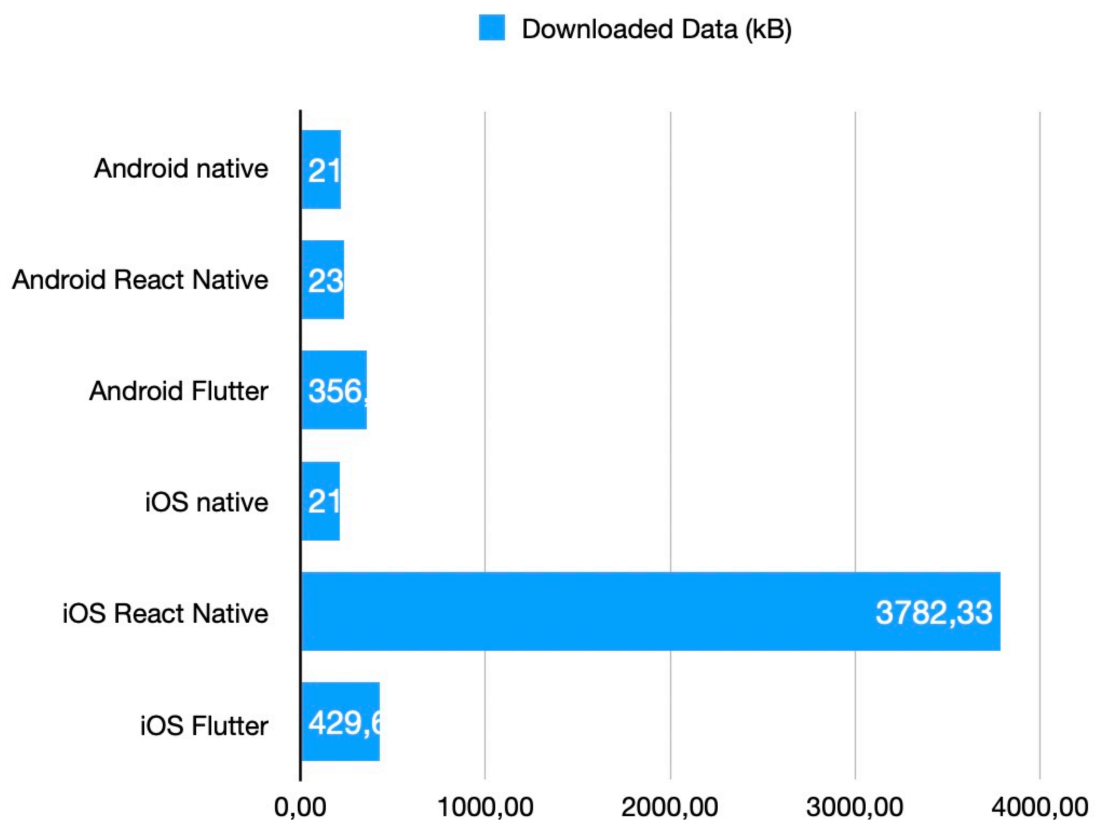

Abbildung 7.2: Durchschnittliche heruntergeladene Daten

Das Balkendiagramm, ohne die React Native iOS-Applikation, ist in Abbildung 7.3 ersichtlich. Flutter lädt auf iOS durchschnittlich 429,67 kB Daten aus dem Netzwerk und die native Applikation 212 kB.

Auf der Android-Plattform lädt die Flutter-Applikation die größte Datenmenge mit 356 kB. React Native liegt mit 235,33 kB sehr nahe an der nativen Applikation, welche 216,67 kB Daten lädt. Die Datensätze von React Native für die Android-Plattform weißen keine verfälschten Daten auf.

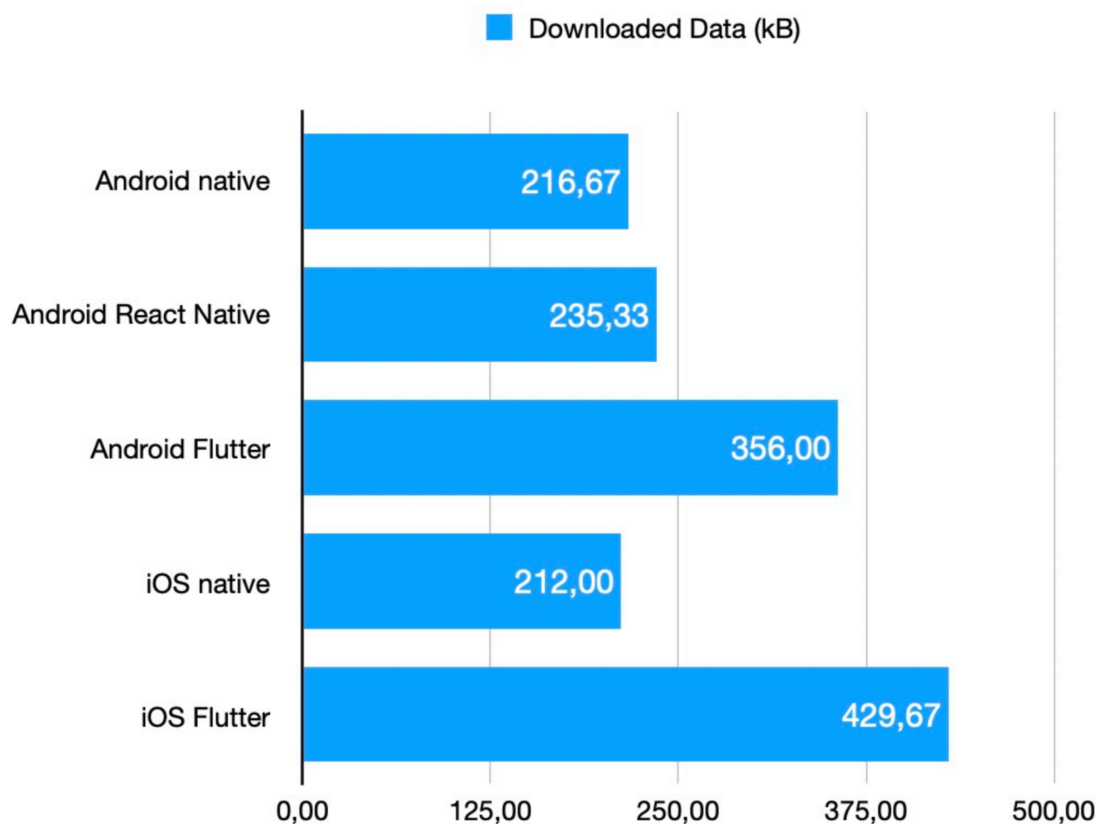

Abbildung 7.3: Durchschnittliche heruntergeladene Daten ohne React Native iOS

Die Größe der heruntergeladenen Daten hängt sehr stark mit dem asynchronen Ladeprozess der Bilder und der Geschwindigkeit mit der zum Ende der Liste navigiert wird zusammen. Die nur sehr geringe Abweichung von wenigen hunderten kB könnte sich dadurch erklären lassen.

7.3 FPS

Die durchschnittliche Bildrate der Applikationen ist in Abbildung 7.4 abgebildet. Flutter schneidet hier auf iOS am schlechtesten ab mit 27,96 FPS. React Native liegt etwas vor Flutter mit 28,07 FPS. Die native Implementierung liefert, mit 30,76 FPS, die besten Ergebnisse.

Die Datensätze für die Android-Plattform sind denen von iOS sehr ähnlich. Die native Applikation weist die besten Werte, mit 28,70 FPS, auf. Flutter schneidet mit 26,52 FPS besser ab als React Native mit 24,94 FPS.

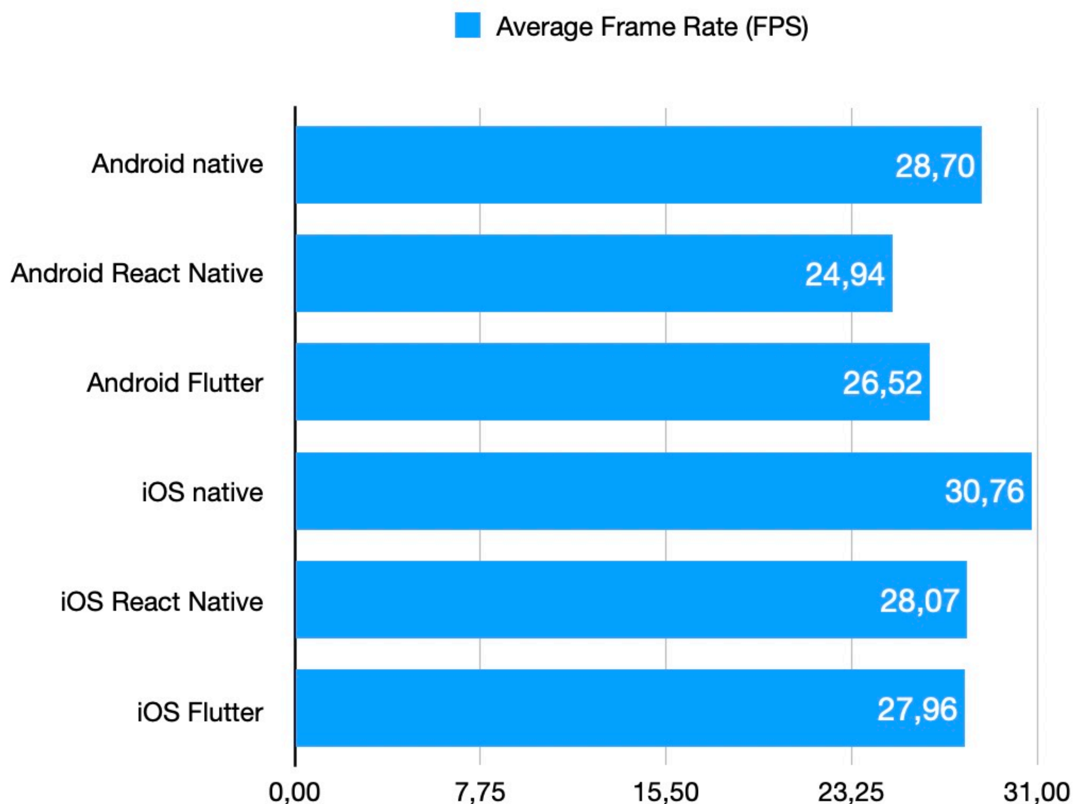

Abbildung 7.4: Durchschnittliche FPS

Der Unterschied von 0,11 FPS auf der iOS-Plattform bzw. 1,58 FPS auf der Android-Plattform, ist sehr gering zwischen React Native und Flutter und die Applikationen liegen bei diesem Vergleich deshalb gleich auf. Die bessere Performance der nativen Applikationen lässt sich durch das Fehlen der benötigten Schnittstellen zu den Cross-Plattform-Frameworks erklären.

7.4 RAM

Wie viel Memory die jeweiligen Applikationen durchschnittlich genutzt haben, ist in Abbildung 7.5 ersichtlich. Die Flutter-iOS-Applikation benötigt durchschnittlich 152,67 MB Speicher. React Native liegt an zweiter Stelle mit 73,33 MB und die native Applikation weist die beste Performance mit 37,33 MB auf.

Dasselbe Muster ist auf Android erkennbar. Flutter benötigt am meisten Speicher mit 315 MB. React Native liegt an zweiter Stelle mit 245 MB und die native Implementierung weist den besten Wert mit 126 MB auf.

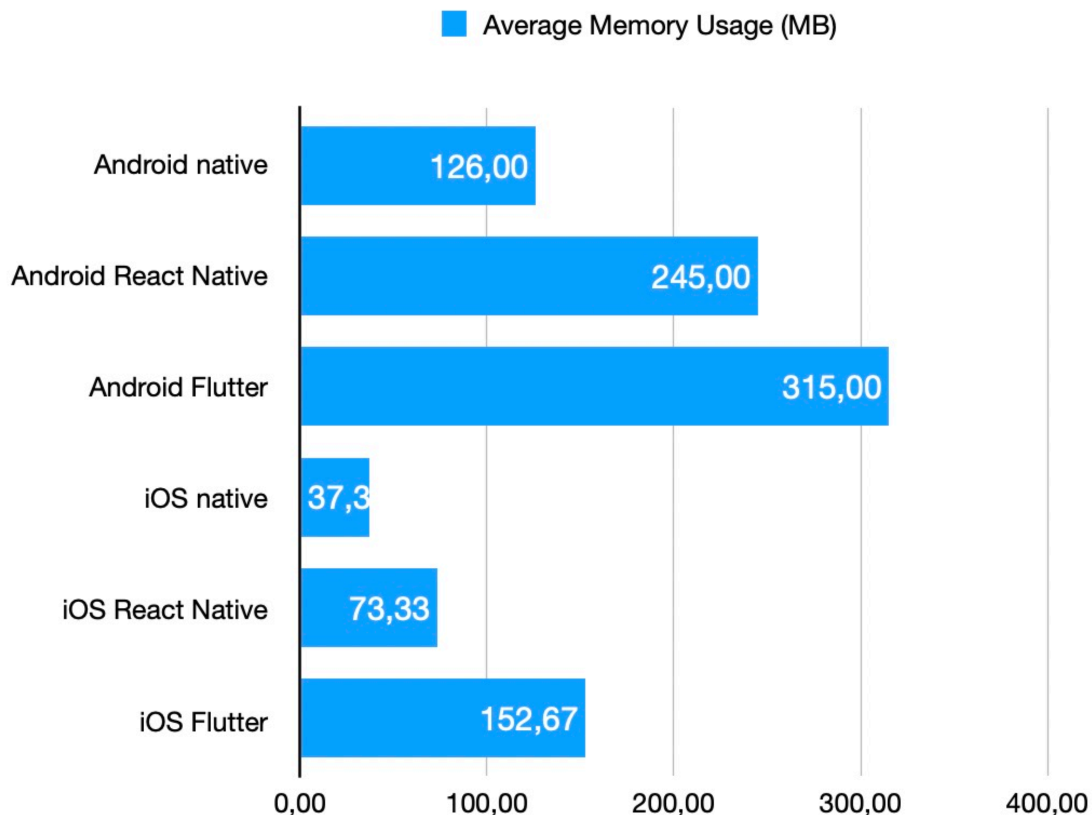

Abbildung 7.5: Durchschnittlich genutzter Speicher

Interessant in diesem Vergleich ist der Unterschied des durchschnittlich genutzten Speichers der Plattformen. Die Android-Plattform scheint grundsätzlich mehr Speicher zu benötigen als die iOS-Plattform. Bei beiden Plattformen benötigt Flutter jedoch den meisten RAM.

7.5 Batterie

Die durchschnittliche Nutzung der Batterie in % ist in Abbildung 7.6 ersichtlich. Auf der iOS- als auch auf der Android-Plattform beansprucht React Native die Batterie mit 35% am größten. Überraschender Weise benötigt Flutter mit 32%, bzw. 28,67%, weniger Batterieleistung als die native iOS- und Android-Applikation mit 33,33% und 31,33%.

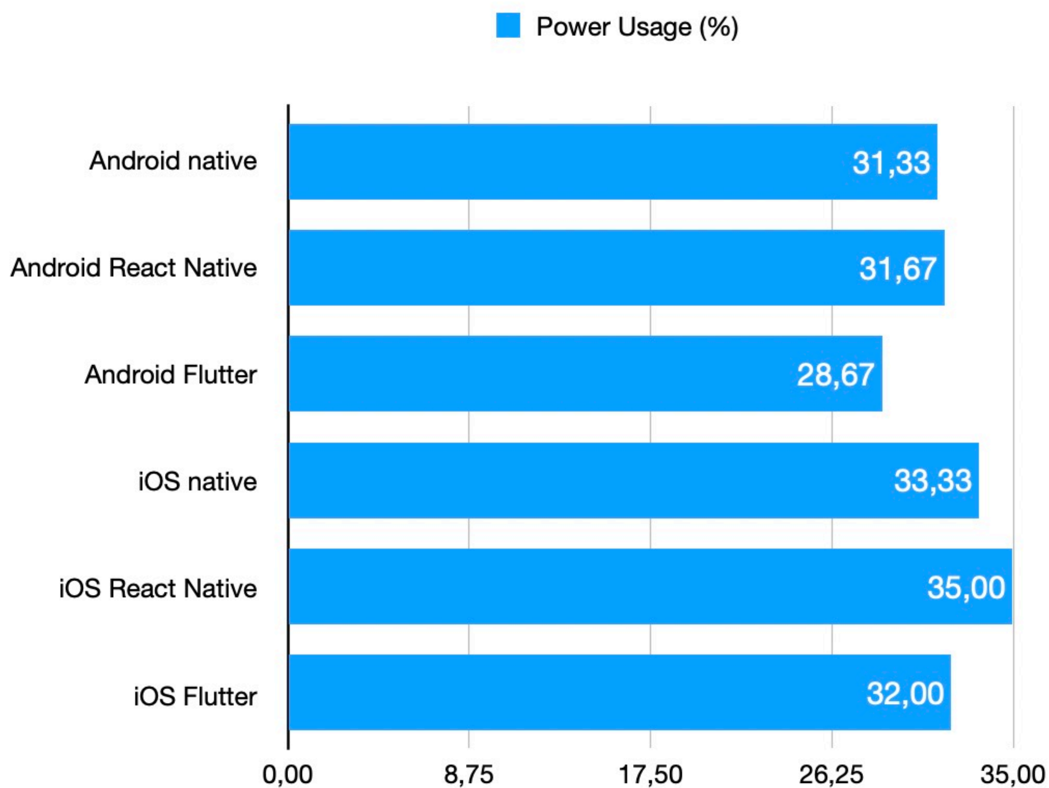

Abbildung 7.6: Durchschnittlicher Batterieverbrauch

7.6 Startzeit

Die benötigte Zeit bis der Home-Screen der Applikation angezeigt wird, ist im Diagramm 7.7 ersichtlich. Bei der Android-Plattform liegen die Werte sehr nah aneinander. Die native als auch die React Native-Applikation zeigten nach durchschnittlich 0,34 Sekunden den Home-Screen an. Die Flutter-Applikation nach durchschnittlich 0,30 Sekunden.

Auch auf der iOS-Plattform lieferte die Flutter-Applikation die besten Ergebnisse, mit einer durchschnittlichen Zeit von 0,36 Sekunden. Die native Applikation benötigte 0,41 Sekunden und die React Native-Applikation 0,52 Sekunden.

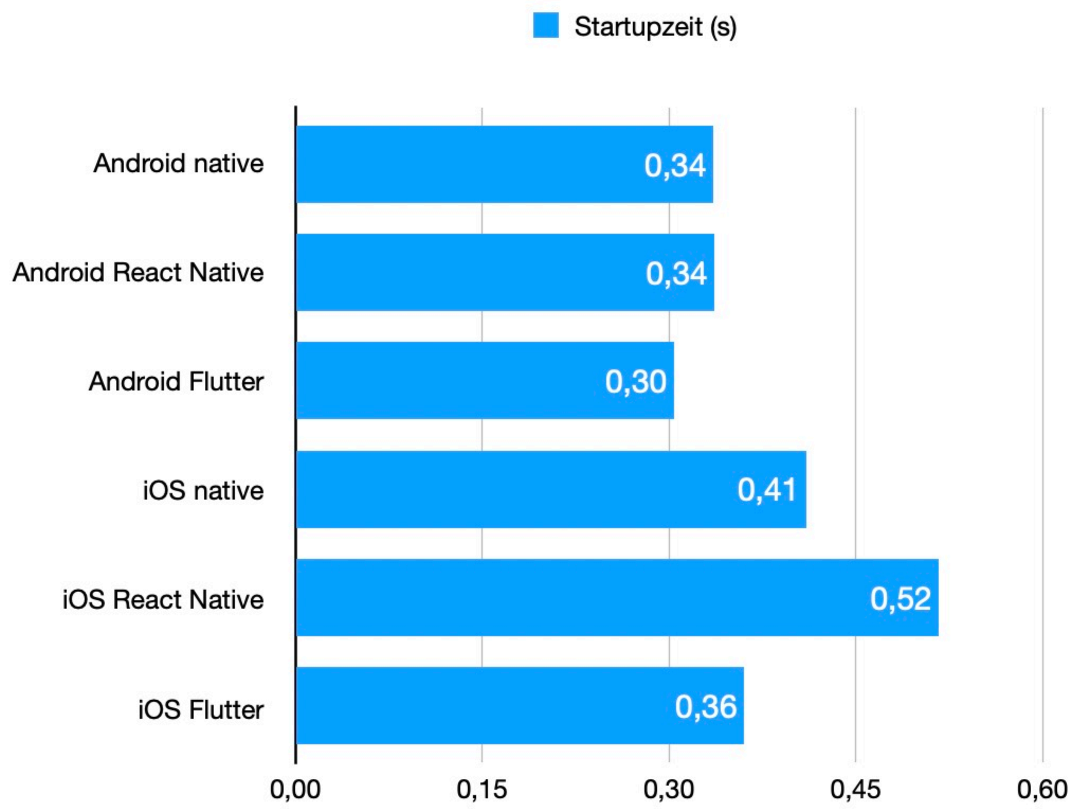

Abbildung 7.7: Durchschnittliche Startdauer

Da sich die Werte im Millisekundenbereich unterscheiden, würde ein Mensch kaum einen Unterschied bemerken. Die Startzeit kann auch von Hintergrund-Prozessen beeinflusst werden und dadurch variieren.

7.7 Zusammenfassung

Die Recherche der theoretischen Inhalte haben aufgezeigt, wie native Applikationen technisch aufgebaut sind und Cross-Plattform-Frameworks auf dieser technischen Basis aufbauen. Dadurch, dass ein Bindeglied zwischen der Kommunikation von nativen Elementen und den plattformunabhängigen Frameworks geschafft werden muss, sind native Applikationen performanter als Applikation welche mit Cross-Plattform-Technologien umgesetzt wurden.

Der große Vorteil von Cross-Plattform-Frameworks liegt in der einheitlichen Code-Basis für die unterschiedlichen Plattformen. Dies ermöglicht eine schnellere Entwicklung und vorallem einen geringeren finanziellen Aufwand für Unternehmen, da der Code nur ein einziges Mal geschrieben werden muss.

Für gewöhnliche Apps mit wenig Anspruch der Ressourcen des Geräts spielt die Technologie keine Rolle. Wenn rechenintensive Vorgänge, Inhalte der Applikation sind, besitzen native Applikationen die beste Performance. Flutter und React Native liefern eine ähnliche Performance. Während Flutter CPU intensive Prozesse performant abbilden kann, fällt bei React Native eine höhere Auslastung an durch das Konzept der Bridge.

Welches Framework gewählt wird, hängt von dem spezifischen Produkt und Geschäftsfall ab. Für den Fall, dass ein Prototyp für eine einzige Plattform entwickelt werden muss, bieten sich native Applikationen an. Cross-Plattform-Anwendungen bringen jedoch den Vorteil, dass sie für Android als auch für iOS verwendet werden können. Zusätzlich bieten Frameworks wie Flutter die Möglichkeit, Applikationen für Web und Desktop bereitzustellen. Diese Plattformen zu testen war kein Ziel und auch kein Inhalt dieser Arbeit, jedoch bietet Flutter diese Möglichkeit an. React Native hat keine Funktionalität die dieses Verhalten ermöglicht, da nur die Plattformen iOS und Android unterstützt werden.

Die Resultate zeigen, dass native Applikationen immer eine bessere Performance als React Native oder Flutter aufweisen. Der Unterschied zwischen Flutter und React Native ist jedoch zu gering um eine klare Aussage treffen zu können ob das jeweilige Framework besser oder schlechter ist. Um diese Frage zu klären, müssten weitere Versuche mit einem umfangreicheren Prototypen durchgeführt werden.

Die Forschungsfrage **"Wie unterscheidet sich Flutter hinsichtlich der Performance und Entwicklung verglichen mit anderen Cross-Plattform-Frameworks?"** kann mithilfe der Ergebnisse dieser Arbeit beantwortet werden. Die Testversuche zeigen, dass die Performance von Flutter mit anderen Frameworks (z. B. React Native) vergleichbar ist und in manchen Aspekten (z. B. der CPU-Auslastung) sogar bessere Resultate liefert.

Flutter ist, verglichen mit anderen Cross-Plattform-Technologien aus dieser Arbeit, das jüngste Konzept. Jedoch hat das Framework, in dieser kurzen Zeit, alle anderen Cross-Plattform-Entwicklungskonzepte in der Popularität überholt. Dieser Fakt und die Tatsache, dass das Unternehmen Google die Technologie stets weiterentwickelt, zeigen, dass Flutter ein sehr zukunftssicheres Framework ist.

Die Vorteile der deklarativen Programmierung, verglichen mit der imperativen Programmierung, können durch Flutter genutzt werden. Dieses Konzept erleichtert den Einstieg in die Programmierung und beschleunigt den Entwicklungsprozess. Auch die nativen Programmiersprachen SwiftUI und Jetpack Compose nutzen die deklarative Programmierung und definieren somit die Richtung in welche sich die Entwicklung von nativen Applikationen ausrichtet.

Um genauere Aussagen über die Performance von Flutter machen zu können, müssten weitere Versuche mit einem größeren Umfang durchgeführt werden. Zukünftige Arbeiten, die sich mit dieser Thematik auseinandersetzen, können auf die Prototypen und Ergebnisse dieser Arbeit aufbauen.

Abbildungsverzeichnis

2.1 Android Activity Lifecycle (Quelle: vgl. Li & Ouyang, 2017) 19
2.2 iOS ViewController Lifecycle (Quelle: vgl. Apple, 2020a) 22

3.1 Architektur von React Native (Quelle: vgl. Facebook, 2020) 27
3.2 Architektur von Ionic (Quelle: vgl. Ionic, 2020) 28
3.3 Architektur von Xamarin (Quelle: vgl. Microsoft, 2020) 29
3.4 Vergleich von Ionic, React Native & Xamarin mit den Google Search Trends 31

4.1 Architektur von Flutter (Quelle: vgl. Google, 2020b) 36
4.2 Lifecycle eines StatefulWidgets (Quelle: vgl. Napoli, 2020) 39
4.3 Prozess einer nativen Applikation (Quelle: vgl. Fayzullaev, 2018) . . . 43
4.4 Prozess einer React Native-Applikation (Quelle: vgl. Wu, 2018) 44
4.5 Prozess einer Flutter Applikation (Quelle: vgl. Mainkar & Giordano, 2019) 45

5.1 Architektur des Prototypen 46
5.2 Ablauf des Prototypen 47

6.1 MVVM-Pattern mit ReactiveX 54
6.2 Flutter Dateistruktur 55

7.1 Durchschnittliche CPU-Auslastung 66
7.2 Durchschnittliche heruntergeladene Daten 68
7.3 Durchschnittliche heruntergeladene Daten ohne React Native iOS . . . 69
7.4 Durchschnittliche FPS 70
7.5 Durchschnittlich genutzter Speicher 71
7.6 Durchschnittlicher Batterieverbrauch 72
7.7 Durchschnittliche Startdauer 73

Tabellenverzeichnis

3.1 GitHub Stars der Repositories von React Native, Ionic & Xamarin.Forms 33

5.1 iOS Testgerät 50
5.2 Android Testgerät 50

Listings

2.1 Beispiel für Objective-C ViewController 7
2.2 Beispiel für Objective-C 7
2.3 Beispiel für Swift 9
2.4 Beispiel für SwiftUI 10
2.5 MainActivity der Hello World Java Applikation 13
2.6 Layout der Hello World Java Applikation 13
2.7 MainActivity der Hello World Kotlin Applikation 15
2.8 MainActivity der Hello World Jetpack Compose Applikation 16

4.1 Container Widget 37
4.2 Imperative UI 41
4.3 Deklarative UI 41

6.1 player.dart 56
6.2 api_client.dart 56
6.3 player_repository.dart 57
6.4 player_detail_view_model.dart 57
6.5 player_detail.dart 58
6.6 player_list_view_model.dart 59
6.7 player_list.dart 60
6.8 main.dart 62
6.9 di.dart 63
6.10 Datensätze für den Import 63

Literaturverzeichnis

Apple. (2020a). *Start developing ios apps.* Zugriff am 06.09.2020 auf https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/WorkWithViewControllers.html

Apple. (2020b). *Swift.* Zugriff am 26.06.2020 auf https://developer.apple.com/swift/

Apple. (2020c). *Swiftui.* Zugriff am 26.06.2020 auf https://developer.apple.com/xcode/swiftui/

Austerlitz, H. (2003). *Data acquisition techniques using pcs*. Academic Press.

Bennett, J. (2018). *Xamarin in action: Creating native cross-platform mobile apps*. Manning Publications.

Boduch, A. & Derks, R. (2020). *React and react native: A complete hands-on guide to modern web and mobile development with react.js, 3rd edition* (3. Aufl.). Packt Publishing Ltd.

Borges, H. & Tulio Valente, M. (2018, Dec). What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, *146*, 112–129. Zugriff auf http://dx.doi.org/10.1016/j.jss.2018.09.016 doi: 10.1016/j.jss.2018.09.016

Cheng, F. (2018). *Build mobile apps with ionic 4 and firebase: Hybrid mobile app development* (2nd ed. Aufl.). Apress.

Clark, L. (2017). *A crash course in just-in-time (jit) compilers.* Zugriff am 15.09.2020 auf https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

cruxlab. (2020). *Xamarin vs ionic vs react native: differences under the hood.* Zugriff am 21.09.2020 auf https://cruxlab.com/blog/reactnative-vs-xamarin/

Erik, H. (2013). *Android programming*. West Sussex, PO19 8SQ, United Kingdom: John Wiley & Sons, Incorporated., Wiley Imprint.

Facebook. (2020). *React native internals.* Zugriff am 14.09.2020 auf https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html

Fayzullaev, J. (2018). Native-like cross-platform mobile development: Multi-os engine & kotlin native vs flutter.

Feiler, J. (2014). *ios app development for dummies*. 111 River Street, Hoboken: John Wiley & Sons, Inc.

Flutter. (2020a). *Flutter.* Zugriff am 29.11.2020 auf https://github.com/flutter/flutter

Flutter. (2020b). *Flutter.* Zugriff am 29.11.2020 auf https://api.flutter.dev/flutter/widgets/Container-class.html

Garofalo, R. (2011). *Building enterprise applications with windows presentation foundation and the model view viewmodel pattern*. Microsoft Press.

Google. (2020a). *Android studio.* Zugriff am 06.09.2020 auf https://developer.android.com/studio/intro

Google. (2020b). *Architectural overview.* Autor. Zugriff am 21.10.2020 auf https://flutter.dev/docs/resources/architectural-overview

Google. (2020c). *Effective dart.* Autor. Zugriff am 21.10.2020 auf https://dart.dev/guides/language/effective-dart

Google. (2020d). *Google search trends.* Zugriff am 15.09.2020 auf https://trends.google.com/trends/?geo=US

Google. (2020e). *Introduction to widgets.* Autor. Zugriff am 21.10.2020 auf https://flutter.dev/docs/development/ui/widgets-intro

Google. (2020f). *Jetpack compose basics.* Zugriff am 06.09.2020 auf https://codelabs.developers.google.com/codelabs/jetpack-compose-basics/#0

Google. (2020g). *Understand the activity lifecycle.* Zugriff am 06.09.2020 auf https://developer.android.com/guide/components/activities/activity-lifecycle

Hassan, M. I. (2019). *Swift vs objective-c in 2019.* Zugriff am 11.07.2020 auf https://medium.com/swiftify/swift-vs-objective-c-comparison-32aba9dad4e3

Horton, J. (2019). *Android programming with kotlin for beginners: Build android apps starting from zero programming experience with the new kotlin programming language* (1. Aufl.). Packt Publishing.

Ionic. (2020). *Comparing cross-plattform-frameworks.* Zugriff am 14.09.2020 auf https://ionicframework.com/resources/articles/ionic-vs-react-native-a-comparison-guide

Iversen, J. & Eierman, M. (2013). *Learning mobile app development*. New Jersey: Addison-Wesley.

Kochan, S. G. (2011). *Programming in objective-c*. 111 River Street, Hoboken: Addison-Wesley Professional.

Li, Y. & Ouyang, J. (2017). Data flow analysis on android platform with fragment lifecycle modeling and callbacks. *EAI Endorsed Transactions on Security and Safety*, *4*. Zugriff auf https://www.researchgate.net/publication/321675426_Data_Flow_Analysis_on_Android_Platform_with_Fragment_Lifecycle_Modeling_and_Callbacks

Maglie, A. (2016). *Reactivex and rxjava*. Apress.

Mainkar, P. & Giordano, S. (2019). *Google flutter mobile development quick start guide*. Packt Publishing.

Microsoft. (2012). *Developing modern mobile web app*. Autor.

Microsoft. (2017). *Ios-app-architektur.* Zugriff am 15.09.2020 auf https://docs

.microsoft.com/de-de/xamarin/ios/internals/architecture

Microsoft. (2020). *What is xamarin?* Zugriff am 14.09.2020 auf https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin

Napoli, M. (2020). *Beginning flutter: A hands on guide to app development.* John Wiley & Sons. Zugriff am 21.10.2020 auf https://books.google.at/books?hl=en&lr=&id=ex-tDwAAQBAJ&oi=fnd&pg=PR21&dq=flutter+lifecycle&ots=YVIua17hMe&sig=Ks24jFNYGec9QgmIIwX4xQtwr5c&redir_esc=y#v=onepage&q=flutter%20lifecycle&f=false

Parmenter, D. (2015). *Key performance indicators: developing, implementing, and using winning kpis.* John Wiley & Sons. Zugriff am 21.10.2020 auf https://books.google.at/books?hl=en&lr=&id=bKkxBwAAQBAJ&oi=fnd&pg=PA101&dq=kpis&ots=cZT2i_iYcp&sig=3iw_60EPKUp3ZLQVw4Vgo0vuptU&redir_esc=y#v=onepage&q=kpis&f=false

Paul. (2019). *The fastest growing skills among software engineers.* Zugriff am 26.06.2020 auf https://www.linkedin.com/business/learning/blog/productivity-tips/the-fastest-growing-skills-among-software-engineers-and-how-to

Scott, A. D. (2020). *Javascript everywhere*. O'Reilly Media, Inc.

StackOverflow. (2019). *Developer survey results.* Zugriff am 26.06.2020 auf https://insights.stackoverflow.com/survey/2019

Terry Jones, N. H. T. (2014). *Learning jquery deferreds: Taming callback hell with deferreds and promises*. O'Reilly Media.

Unuth, N. (2019). *What is multitasking in smartphones.* Zugriff am 26.06.2020 auf https://www.lifewire.com/what-is-multitasking-in-smartphones-3426819

Wu, W. (2018). *React native vs flutter, cross-platforms mobile application frameworks.* Metropolia University of Applied Sciences. Zugriff auf https://www.theseus.fi/bitstream/handle/10024/146232/thesis.pdf?sequence=1