THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Bachelorarbeit

Brownfield Hybridisierung von nativen iOS und Android Apps mittels React Native

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

eingereicht im Fachbereich Mathematik, Naturwissenschaften und Informatik an der Technischen Hochschule Mittelhessen

von

Florian Maximilian Dörr

Oktober 2023

Referent: Prof. Dr. Steffen Vaupel

Korreferent: Sebastian Süß, M.Sc.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Buseck, den 26. Oktober 2023 Florian Maximilian Dörr

Danksagung

Zunächst möchte ich mich besonders bei allen Personen bedanken, die mich bei der Erstellung dieser Ausarbeitung unterstützt haben. Dabei ist mein Betreuer Steffen Vaupel hervorzuheben, der mich in besonderem Maße durch konstruktive Kritik sowie bei der Beantwortung von fachlichen Fragestellungen unterstützt hat.

Ohne die Praktikumstätigkeit bei der CURSOR Software AG und die mir zur Seite gestellten Betreuenden Anne-Maria Lange sowie Manuel Krug, wäre eine intensive Einarbeitung in die React Native Bibliothek nicht möglich gewesen. Nicht zuletzt durch diese Tätigkeit kam das Thema dieser Ausarbeitung erst zustande.

Zusätzlich gilt mein besonderer Dank den Korrekturlesenden Silke und Wolfram Dörr, Sarah Matheja, Manuel Krug sowie Hendrik Wagner.

Weiterhin danke ich Nicola Corti – Entwickler von React Native – für die Aufnahme innerhalb der geschlossenen Diskussionsplattform[1] zur neuen Architektur React Natives. Die Antworten durch Nicola Corti sowie Riccardo Cipolleschi auf meine dort gestellten Fragen gaben mir einen tieferen Einblick in die weniger gut dokumentierten Aspekte React Natives, wodurch diese Ausarbeitung in hohem Maße profitieren konnte.

1 vgl. `https://github.com/reactwg/react-native-new-architecture`, abgerufen am 26. Oktober 2023.

Abstract

The two most commonly used mobile operating systems worldwide are iOS and Android. The market share of iOS is 29.58 %, while Android's is 69.74 %. [1] Thus, to cover $\sim 99\,\%$ of all mobile devices, it is necessary to develop two separate apps. This evidently results in double the development effort needed for such apps. Nowadays, there are several hybrid app frameworks that allow one to develop apps for both iOS and Android.

But what if you decided to develop two native apps in the past? Is it possible to hybridize existing apps, or is it necessary to start from scratch? At first, this paper provides a comparison of two case studies for the integration of React Native into existing apps. This is to be followed by an introduction to React as well as React Native. Next, an integration of React Native into a model iOS and Android app is performed. Lastly, three possibilities for synchronizing the data from React Native and the native apps are illustrated.

Zusammenfassung

Die beiden weltweit am häufigsten verwendeten mobilen Betriebssysteme sind iOS und Android. Der Marktanteil von iOS liegt bei 29,58 %, der von Android bei 69,74 %. [1] Um folglich $\sim 99\,\%$ aller mobilen Geräte abzudecken, ist es notwendig, zwei separate Apps zu entwickeln. Infolgedessen ist ein doppelter Entwicklungsaufwand für Apps erforderlich, welche sowohl iOS als auch Android abdecken sollen. Heutzutage existieren mehrere Hybrid-App-Frameworks, welche es ermöglichen, Apps sowohl für iOS als auch für Android zu entwickeln. Wie sollte jedoch vorgegangen werden, wenn in der Vergangenheit der Ansatz der Entwicklung zweier nativer Apps gewählt wurde? Ist es möglich, bestehende Apps zu hybridisieren oder ist es notwendig, gänzlich von vorne zu beginnen? Diese Ausarbeitung bietet zunächst einen Vergleich von zwei Fallstudien zur Integration von React Native in bestehende Apps. Darauf folgt eine Einführung in React sowie in React Native. Anschließend wird eine Integration von React Native in eine beispielhafte iOS und Android App durchgeführt. Zuletzt werden drei Möglichkeiten zur Synchronisation der Daten aus React Native und den nativen Apps illustriert.

Inhaltsverzeichnis

1 Einleitung

Die Smartphone-Landschaft ist in zwei große Betriebssystem-Lager fragmentiert: Googles Android und Apples iOS.

Mit einem Marktanteil von 69,74 % bleibt Android auch im Jahr 2023 weltweit an der Spitze der mobilen Betriebssysteme, während iOS einen Marktanteil von 29,58 % aufweist. [1] Um eine mobile Anwendung für beide Plattformen bereitzustellen, war es vor einigen Jahren notwendig, zwei separate, native Apps zu entwickeln. Da Konzeption, Implementierung sowie Tests für jede Plattform durchgeführt werden müssen, ist das Entwickeln von nativen Apps für iOS und Android evident zeitintensiv und somit teuer. Dennoch setzten einige Unternehmen auf die Entwicklung von zwei nativen Apps, da hybride Technologien, welche es erlauben, eine App für beide Plattformen zu entwickeln, zu diesem Zeitpunkt unzureichend ausgereift waren. Diesen Unternehmen bleibt nun nichts anderes übrig, als diesen Ansatz weiterzuverfolgen, obschon hybride Technologien in den vergangenen Jahren in vielen Aspekten reifer geworden sind. Diese Arbeit untersucht, inwiefern eine nachträgliche Hybridisierung von nativen iOS und Android Apps möglich ist, mit dem Ziel, den künftigen Entwicklungsaufwand zu minimieren.

1.1 Motivation

Neben der Möglichkeit zur Entwicklung von nativen Apps, ist es weiterhin möglich, eine mobile Webseite oder eine hybride App zu entwickeln. Doch wenn bereits umfangreiche native Apps entwickelt wurden, da zum Projektstart der mobilen Anwendungen hybride Ansätze weniger weitverbreitet waren, ist es notwendig, diese im vollen Funktionsumfang als neue Webseite oder als hybride App umzusetzen. Dabei birgt eine Neuentwicklung mit einem unbekannten Web-Framework oder einer hybriden Technologie unvorhersehbare Risiken. Im Verlauf der Ausarbeitung wird dargelegt, ob eine nachträgliche Hybridisierung der nativen mobilen Applikationen diese Risiken und den großen Aufwand minimieren können.

Dabei zeigen Auswertungen, dass hybride Ansätze prinzipiell zu einer Kosteneinsparung bei der App-Entwicklung führen. Weiterhin ermöglicht das Verwenden eines hybriden Ansatzes eine verkürzte Entwicklungszeit, vorausgesetzt die zu implementierende App

benötigt keine umfangreichen selbst implementierten Plugins für den Zugriff auf native Funktionalitäten. [2][1]

Das hybride App-Framework React Native bietet die Möglichkeit, in bereits bestehende iOS und Android Projekte integriert zu werden. Dieser Ansatz wird dabei gemeinhin als *Brownfield*-Entwicklung[2] bezeichnet. Dabei wird eine alte Implementierung – in diesem konkreten Fall die nativen iOS und Android Apps – weiterhin zu Teilen verwendet. Das Gegenteil der *Brownfield*-Entwicklung wird wiederum als *Greenfield*-Entwicklung[3] bezeichnet. Dabei ersetzt eine Neuimplementierung die alte Implementierung vollumfänglich zu einem gewissen Stichtag.

Zugleich ist eine *Greenfield*-Entwicklung großer mobiler Applikationen mit einer hybriden Technologie mit Risiken und einem enormen Aufwand verbunden. So muss die alte App so lange gewartet werden, bis die neue App bereit ist, die alte zu ersetzen.

Unter Berücksichtigung dieser Faktoren erscheint eine *Brownfield*-Entwicklung als eine attraktive Option. Ein solcher Ansatz ermöglicht den inkrementelle Austausch alter Komponenten oder aber die Implementierung von neuen Komponenten mit einer hybriden Technologie. Dennoch birgt auch die *Brownfield*-Entwicklung einige Schwierigkeiten, die nicht außer Acht gelassen werden dürfen.

So ist es notwendig, die Integration von React Native in offlinefähige Apps sorgfältiger zu betrachten. Befindet sich die *Brownfield*-Entwicklung in einem frühen Stadium, wird die Situation entstehen, dass die React Native App, Daten vom Server benötigt. Besteht zwischen dem Gerät und dem Server dabei eine zuverlässige Verbindung, funktioniert der Abruf der Daten einwandfrei. Befindet sich das Gerät jedoch im Offlinezustand und eine Version der anzufragenden Daten befindet sich nicht bereits im Offlinespeicher von React Native, kann es dennoch möglich sein, dass die native App diese Daten bereithält, da diese die Daten zuvor abgefragt hat, bevor das Gerät die Verbindung verloren hat. Ohne weitere Synchronisationen der Offlinezustände der nativen App und React Native, kommt es in einem solchen Szenario zu Inkonsistenzen zwischen diesen beiden Softwarekomponenten. Folglich besteht die Notwendigkeit, ein Verfahren zu verwenden, um derartige Inkonsistenzen zu verhindern. Werden etwa Daten innerhalb von React Native vom Server abgefragt und im Offlinespeicher persistiert, besteht für die native App zunächst keine Möglichkeit, auf diese Daten zuzugreifen.

1 Diese Quelle wurde zwar in einem Journal veröffentlicht, die darin wiederum verwendeten Quellen sind jedoch als wenig vertrauenswürdig einzustufen, obschon die getätigten Aussagen per se schlüssig sind.

2 Eine *Brownfield*-Entwicklung beschreibt die Integration einer neuen Technologie in ein bereits bestehendes Projekt. Die alte Implementierung kann dabei teilweise weiterverwendet werden. [3, 4]

3 Eine *Greenfield*-Entwicklung beschreibt eine komplette Neuentwicklung. Dabei wird die alte Implementierung vollumfänglich ersetzt. [3, 4]

1.2 Zielgruppe

Diese Ausarbeitung richtet sich primär an Unternehmen, genauer gesagt an deren App Entwicklerinnen und Entwickler, die bereits umfangreiche native iOS und Android Apps verfasst haben, jedoch das Ziel definiert haben, den Entwicklungsaufwand der Apps zu zukünftig reduzieren. Weiterhin ist es dienlich, wenn bereits eine *Greenfield*-Entwicklung mit einem hybriden App-Framework in Betracht gezogen wurde, diese jedoch mit einem für das Unternehmen nicht tragbaren Entwicklungsaufwand bzw. Risiko verbunden ist. Auch wenn noch keine Auswertungen über die Auswirkungen einer möglichen *Greenfield*-Entwicklung in Betracht gezogen wurden, kann diese Ausarbeitung, insbesondere ausgehend von den in Kapitel 2 ausgewerteten Fallstudien, Einblicke darüber liefern, ob eine *Brownfield*-Entwicklung oder eine *Greenfield*-Entwicklung gelegener ist. Weiterhin sind die in dieser Ausarbeitung vermittelten Einblicke für Personen interessant, welche sich bereits in der Vergangenheit mit einer Integration von React Native beschäftigt haben, da diese Ausarbeitung ebenfalls auf neuartige Aspekte React Natives eingeht.

1.3 Gliederung der Ausarbeitung

Diese Ausarbeitung ist in sieben Kapitel unterteilt. Kapitel 1 gibt eine Einleitung in die Arbeit und deren Aufbau.

Das zweite Kapitel beschäftigt sich mit zwei konkreten Fallstudien zur Integration von React Native in bestehende native iOS und Android Apps. Zunächst wird die gescheiterte *Brownfield*-Entwicklung in Verbindung mit React Native von Airbnb behandelt. Airbnb startete 2016 mit dem Versuch, React Native in die bestehenden nativen iOS und Android Apps zu integrieren. 2018 wurde diese Unternehmung jedoch aus verschiedenen Gründen abgebrochen. Ein Mitarbeiter von Airbnb verfasste daraufhin einen Artikel, in dem er die Gründe für den Abbruch des Integrationsprojektes offenlegt. Darauffolgend wird die Integration von React Native in die nativen iOS und Android Apps der CURSOR Software AG behandelt. Auch die CURSOR Software AG unternahm 2023 die Integration von React Native. Diese Integration wird dabei zum aktuellen Zeitpunkt weitergeführt und beeinflusste das Thema dieser Ausarbeitung damit maßgeblich, ausgehend der Praktikumstätigkeit des Autors in diesem Projekt. Aus diesen beiden Erfahrungsberichten werden die verschiedenen Motivationen zur Integration von React Native extrahiert. Weiterhin werden die Befunde über aufgedeckte Risiken, Hürden und Vorteile bei der *Brownfield*-Entwicklung mit React Native innerhalb der beiden Projekte gegenübergestellt. Die durch dieses Kapitel vermittelten Erfahrungen können dabei bei der Entscheidungsfindung über eine *Brownfield*-Entwicklung oder eine *Greenfield*-Entwicklung bedeutsam sein. So ist es möglich, einige der misslungenen

Aspekte bereits zu Beginn einer *Brownfield*-Entwicklung besonders zu berücksichtigen, um diese möglichst zu vermeiden.

In Kapitel 3 folgt eine Einführung in die JavaScript-Bibliothek React. Ein Grundverständnis von React ist für die Verwendung von React Native dabei zwingend notwendig, da React Native wiederum React verwendet. Sollten sich Lesende ein tieferes Verständnis zu React aneignen wollen, werden einige Literaturempfehlungen angeführt. In Abschnitt 3.1 werden zunächst zwei Möglichkeiten zur Definition von React Komponenten behandelt; funktionale Komponenten und Klassen-Komponenten. Zusätzlich werden die in React als Übergabeparamter fungierenden *Props* erläutert. Weiterhin wird die von React verwendete *JavaScript Syntax Extension* (*JSX*) als Auszeichnungssprache Reacts vorgestellt. Abschließend wird die Rolle des *Virtual Document Object Models* (*VDOMs*) erklärt und dargelegt, warum dieser für die Funktionsweise von React Native unerlässlich ist. Dieses Kapitel ist folglich besonders für Lesende relevant, die zuvor keine bis wenig Erfahrungen mit React sammeln konnten.

Darauf aufbauend wird in Kapitel 4 React Native selbst behandelt. Dabei wird zunächst die allgemeine Funktionsweise von React Native erläutert. Weiterhin werden einige Aspekte einer neuen Architektur React Natives behandelt. Anschließend wird anhand eines konkreten Beispiels die Verwendung von nativen Modulen erläutert. Native Module sind dabei besonders wichtig für die Verwendung von React Native in einer *Brownfield*-Umgebung. So erlauben diese den Zugriff auf Ressourcen der nativen Apps und schlagen somit eine Brücke zwischen React Native und den nativen iOS und Android Apps. Da bei der Verwendung von nativen Modulen stets React Native die Kommunikation initiieren muss, existieren zusätzlich sogenannte *Event Emitter*. Bei der Verwendung von *Event Emittern* muss im Gegensatz zu nativen Modulen die native App die Kommunikation einleiten. Die Verwendung von *Event Emittern* wird dabei ebenfalls anhand eines praktischen Beispiels erläutert.

Native Module verwenden dabei jedoch nicht die neu in React Native eingeführte Architektur. Dafür sind *Turbo Native Modules* vorgesehen. Um die neue Architektur von React Native mit *Turbo Native Modules* zu nutzen, wird zunächst beschrieben, wie diese für iOS und Android aktiviert werden kann. Aufgrund der signifikanten Unterschiede in der Verwendung von *Turbo Native Modules* im Vergleich zu nativen Modulen wird das in Abschnitt 4.2 dargestellte Beispiel in Abschnitt 4.4 unter Verwendung von *Turbo Native Modules* implementiert. Dieses Kapitel vermittelt dabei fortgeschrittene Aspekte von React Native, welche dennoch unerlässlich für die *Brownfield*-Entwicklung mit React Native sind.

Kapitel 5 behandelt die *Brownfield*-Entwicklung mit React Native. Dazu wurden im Vorfeld native iOS und Android Apps erstellt, welche zuvor vom Autor erstellt wurden. Anhand dieser Anwendungen wird beispielhaft die schrittweise Integration von React

Native demonstriert. Das Ziel bei der Erstellung der nativen Anwendungen bestand darin, dass diese möglichst repräsentativ gestaltet werden, um eine Integration in ähnliche Anwendungen möglichst einfach zu gestalten. Damit React Native in bestehenden iOS und Android Apps verwendet werden kann, wird zunächst beschrieben, wie React Native in einem bestehenden Projekt initiiert werden muss. Anschließend werden notwendige Schritte für iOS dargelegt. Dabei sind zunächst Anpassungen am *Build* der iOS App notwendig. Anschließend wird aufgezeigt, wie die native iOS App eine Verbindung zu React Native herstellen und letztlich eine zuvor erstellte React Native Komponente zur Anzeige bringen kann. Weiterhin werden spezifische Anpassungen vorgestellt, die für einen erfolgreichen *Build* der iOS App im Produktivmodus erforderlich sind. Daraufhin werden notwendige Anpassungen der Android App vorgeführt. Zunächst ist es dabei notwendig – wie auch unter iOS – einige Anpassungen am *Build*-Prozess der App vorzunehmen. Anschließend wird dargestellt, wie eine Verbindung zu React Native hergestellt und die gleiche beispielhafte React Native Komponente wie auch unter iOS innerhalb der Android App angezeigt werden kann. Um in einem *Brownfield*-Umfeld die neue Architektur React Natives aktivieren zu können, müssen neben den Schritten, die bereits in Kapitel 4 aufgezeigt wurden, weitere Anpassungen getätigt werden. Dieses Kapitel zeigt folglich, wie Lesende React Native praktisch in bestehende iOS und Android Apps integrieren kann. Überdies werden Aspekte aus Kapitel 4 erneut aufgegriffen, wodurch die Relevanz der zuvor vermittelten Aspekte in einem *Brownfield*-Umfeld deutlich werden.

Viele native Apps nutzen Datenbanken, um einen Offline-Betrieb der App zu ermöglichen. Dies bedeutet, dass auch React Native Komponenten in der Lage sein müssen, Offlinedaten bereitzuhalten. Eine Herausforderung dabei ist die Entwicklung einer geeigneten Synchronisationsstrategie der Offline-Datenbanken im Verbund aus Android und iOS App sowie von React Native, um konsistente Datenstände aufrechterhalten zu können.

Dabei werden in Kapitel 6 drei konkrete Synchronisationsstrategien behandelt und Implementierungsansätze empfohlen. Die diskutierten Ansätze sollen dabei sicherstellen, dass React Native möglichst unabhängig von der Datenhaltung der nativen Apps ist. Grund hierfür ist, dass auf diese Weise das Entfernen der Speicherlogiken der nativen Apps möglich ist, sobald alle Komponenten der nativen App durch React Native ersetzt wurden. Zunächst wird eine Spiegelung der React Native Datenbank (Ansatz ⓐ) behandelt. Bei diesem Ansatz werden dabei die Datenbanken der nativen Apps eliminiert und in die Datenbank React Natives migriert. Insbesondere werden die umfangreichen Anpassungen der bestehenden Apps sowie der Zugriff auf die React Native Datenbank thematisiert. Als weiterer Ansatz wird die Nachbildung von aufgerufenen API-Endpunkten (Ansatz ⓑ) behandelt. Dabei müssen die nativen Apps sowie React Native die gleichen Endpunkte bereitstellen, welche auch der angefragte Server bereitstellt. Bei

der Abfrage von Daten im Offline-Szenario werden die Anfragen an die nativen Apps oder React Native weitergeleitet, statt von dem Server bearbeitet zu werden. Als letzter Ansatz wird eine Synchronisationsstrategie mittels des *Command Patterns* sowie eines nativen Moduls (Ansatz ⓒ) behandelt. Abschließend werden die Strategien (ⓐ, ⓑ und ⓒ) gegenübergestellt und bewertet. Dieses Kapitel vermittelt folglich, ausgehend von der Problematik um inkonsistente offline Datenstände, drei mögliche Lösungsansätze, welche innerhalb einer *Brownfield*-Integration von Lesenden praktisch umgesetzt werden können, um die *Brownfield*-Entwicklung letztlich abzurunden.

Zuletzt erfolgt in Kapitel 7 eine Zusammenfassung der dargelegten Erkenntnisse sowie ein Ausblick über weitere Forschungsmöglichkeiten im Umfeld der Brownfield Hybridisierung von nativen iOS und Android Apps mittels React Native.

1.4 Abgrenzungen

Das Thema um die *Brownfield* Hybridisierung von nativen iOS und Android Apps mittels React Native ist überaus umfangreich. So ist es nicht möglich, alle Aspekte dieses Ansatzes umfassend zu betrachten. Angesichts dessen werden einige Abgrenzungen definiert, welche nicht Teil dieser Ausarbeitung sein werden.

Diese Ausarbeitung befasst sich mit den technischen Aspekten der *Brownfield*-Entwicklung innerhalb von nativen iOS und Android Apps mit der Bibliothek React Native, insbesondere mit der nahtlosen Kommunikation beider Softwareeinheiten. Der vorangestellte und fortlaufende Software-Entwicklungsprozess wird dabei nicht behandelt.

Weiterhin werden notwendige Grundlagen von React in dieser Ausarbeitung dargelegt. Diese ausgewählten Aspekte sind dabei besonders für die Verwendung von React Native relevant, jedoch keineswegs als erschöpfende Wissenssammlung der Bibliothek React zu verstehen. React bietet komplexe Aspekte wie *Higher-Order-Components* (HOCs), welche für die Entwicklung robuster und großer React-Anwendungen unerlässlich sind. Solch fortgeschrittene Aspekte Reacts finden ebenfalls – aufgrund des begrenzten Umfanges – keinen Platz in dieser Ausarbeitung.

Zusätzlich wird eine Affinität mit der Programmiersprache TypeScript vorausgesetzt, da diese in Verbindung mit React zum Einsatz kommt. Wegen des eingeschränkten Umfanges erfolgt dabei keine Einführung in die Programmiersprache TypeScript.

Überdies werden Grundkenntnisse der Programmiersprachen Swift, Obective-C sowie Java erwartet. Diese Annahme beruht auf der Tatsache, dass es insbesondere bei der Integration von React Native notwendig ist, plattformspezifischen Code innerhalb dieser Programmiersprachen zu verfassen. In der iOS App kommt dabei Swift bzw. Objective-C

zum Einsatz. In der Regel ist es dabei möglich – abhängig von der eigenen Präferenz – sowohl Swift oder Objective-C zu verwenden. Dennoch sind kleinere Abschnitte zwingend in Objective-C zu verfassen. In der Android App kommt Java zum Einsatz. Das Verwenden von Kotlin ist dabei ebenfalls möglich, jegliche Code-Beispiele innerhalb dieser Ausarbeitung wurden jedoch mit Java verfasst.

In Verbindung mit React Native kommt die Zustandsbibliothek *Redux* zum Einsatz. Während die Funktionsweise von *Redux* erwähnt wird, wird die Verwendung von *Redux* in Verbindung mit *Redux Toolkit* jedoch nicht weiter behandelt, da *Redux* an sich bereits eine signifikante Komplexität aufweist. Demzufolge empfiehlt es sich, ein Grundverständnis von *Redux* sowie *Redux Toolkit* zu besitzen.

2 Fallstudien der Brownfield Integration von React Native

Im Folgenden wird die *Brownfield*-Entwicklung mittels React Native genauer betrachtet. Zunächst werden zwei Erfahrungsberichte ausgewertet. Dabei wird zuerst die gescheiterte React Native Integration Airbnbs (vgl. Abschnitt 2.1) sowie die erfolgreiche React Native Integration der CURSOR Software AG (vgl. Abschnitt 2.2) behandelt.

2.1 Fallstudie Airbnb

Airbnb startete 2016 mit einer *Brownfield*-Integration React Natives in die bestehende iOS und Android App. Der Airbnb-Mitarbeiter Gabriel Peal verfasste 2018 einen fünfteiligen Artikel darüber, warum die Integration bei Airbnb Ende 2017 letztlich missglückte. Der Artikel liefert dabei einen Einblick über die Limitationen und Potenziale einer *Brownfield*-Integration mittels React Native innerhalb des Zeitraums von 2016 bis 2017.

Dabei ist anzumerken, dass das Integrationsprojekt bei Airbnb von 2016 bis 2017 durchgeführt und der Artikel 2018 verfasst wurde. Somit spiegelt der Artikel nicht den aktuellen technischen Stand React Natives wider, was Peal selbst ausdrücklich zu Beginn des Artikels angemerkt. Der Erfahrungsbericht der CURSOR Software AG ist dabei sehr aktuell (2023). (vgl. Abschnitt 2.2)

2.1.1 Ausgangssituation

Airbnb erkannte die wachsende Bedeutung von mobilen Apps für ihr Geschäftsmodell und investierte daher verstärkt in deren Entwicklung, um die gesetzten Ziele zu erreichen. Der große Aufwand mündete letztlich in einem Personalengpass. Durch die Integration von React Native erhoffte man sich dabei, den benötigten Aufwand zu reduzieren und folglich den Personalengpass zu mitigieren, aufgrund der Tatsache, dass React Native sowohl unter iOS als auch unter Android gleichermaßen lauffähig ist. Zu Beginn des Integrationsprojektes wurden spezifische Ziele definiert, die für eine erfolgreiche

Integration von React Native erreicht werden sollten. Dazu gehörten die Steigerung der Geschwindigkeit im gesamten Unternehmen, die Sicherstellung einer hohen Qualität der React Native Module, die Möglichkeit, Programmcode nur einmal schreiben zu müssen, und die Verbesserung der *Developer Experience* (DX)[1]. Die Qualität sollte dabei anhand der bestehenden nativen Apps gemessen und verglichen werden.

Ausgehend von einer experimentellen Integration, setzte Airbnb kurz darauf einige Komponenten mithilfe React Natives um.

Dennoch scheiterte die Integration von React Native in die Airbnb Apps, wobei die Gründe dafür sowohl technischer als auch organisatorischer Natur waren. [6]

Zunächst konzentriert sich Peal auf Aspekte der Integration, die gut, genauer gesagt nach Plan verlaufen sind. [7]

Im Folgenden werden diese Aspekte ausgehend der Erfahrungen der beiden Projekte behandelt. Einzelne Themen werden dabei nummeriert. Weisen dabei zwei Paragraphen der Auswertung der Fallstudie von Airbnb wie auch der Erfahrungen der CURSOR Software AG die gleiche Bezifferung auf, handelt es sich um den gleichen Aspekt.

2.1.2 Gelungene Aspekte der Integration

1. **Plattformspezifischer Code:** Zunächst war es möglich, den plattformspezifische Code tatsächlich drastisch zu minimieren. Lediglich 0,2 % des React Native Codes musste spezifisch an iOS oder Android angepasst werden.

2. **Design:** Die Verwendung des firmeneigenen Design-Systems Airbnbs hat sich als erfolgreich erwiesen. Dabei wurde beschlossen, die UI-Komponenten neu zu implementieren, anstatt die bereits bestehenden iOS- und Android-Komponenten durch die Verwendung von *Native Components* (vgl. Abschnitt 4.5) wiederzuverwenden. Die Definition von *Native Components* erlaubt dabei das Verwenden von nativen iOS und Android Komponenten innerhalb von React Native. Die Begründung für die Reimplementierung ist ein reduzierter Wartungsaufwand, welcher laut Peal bei Verwendung von *Native Components* aufgrund der Komplexität umfangreicher gewesen wäre. Dieser Ansatz weist jedoch den Nachteil auf, dass Design-Änderungen an den nativen Komponenten nicht automatisch innerhalb von React Native zur Verfügung stehen. Diese Änderungen müssen folglich manuell in die React Native Komponenten übertragen werden. Der Übertrag der Änderungen wurde dabei nicht immer gewissenhaft durchgeführt, was zu

1 Die *Developer Experience* (DX) beschreibt, wie glücklich bzw. zufrieden Entwickelnde mit den ihnen zur Verfügung gestellten Tools/Frameworks sind. Eine bessere DX kann dabei in einer höheren Produktivität sowie in einer besseren Mitarbeiterbindung münden. [5]

Inkonsistenzen zwischen den Komponenten der nativen Apps und der Komponenten React Natives führte.

3. ***Lifecycle*:** Ebenso wird React insbesondere für seine Komponenten-Struktur sowie einen vereinfachten Lebenszyklus[1] im Vergleich zu iOS und Android sowie seinen deklarativen Aufbau geschätzt.

4. **Hot-Reloading**: Hot-Reloading[2] wurde ebenfalls als positiv wahrgenommen. So dauerte im besten Fall das Bereitstellen der nativen Apps zum Testen 15 Sekunden, maximal aber 20 Minuten. React Natives Hot-Reloading hingegen nahm lediglich $1 - 2$ Sekunden in Anspruch. Dies führte zu einer verbesserten Iterationsgeschwindigkeit.

5. **Brückenelemente:** Zusätzlich stellte Peal fest, dass es die richtige Entscheidung war, viel Zeit in die Integrations- bzw. Brückenelemente zwischen React Native und den nativen Apps zu investieren. So seien es diese Brückenelemente gewesen, die sowohl die User-Experience (UX) als auch die DX ausschlaggebend positiv geprägt haben. Dennoch bezeichnet er diese als eine der aufwendigsten Elemente der React Native Integration. Nicht zuletzt durch die ständige Wartung dieser API bei Änderungen in den nativen Apps.

6. **Performance:** Eine größere Sorge des Entwicklerteams Airbnbs war die Performance der React Native Komponenten. Doch traten Performance-Probleme in React Native auf, wurden diese in den meisten Fällen durch ineffizienten React Code ausgelöst, welcher allerdings meist optimiert werden konnte. Überdies war die Performance React Natives im Vergleich zu Komponenten der nativen Apps identisch, so Peal. Dabei gilt anzumerken, dass zu dem Zeitpunkt der Airbnb Integration weder die neue *Fabric-Render-Engine* (vgl. Abschnitt 4.1) noch *react-native-reanimated*[3] (vgl. `https://docs.swmansion.com/react-native-reanimated/`, abgerufen am 26. Oktober 2023) zur Verfügung standen.

7. **Animationen:** Auch flüssige Animation konnten dank der Animated-API React Natives umgesetzt werden.

8. ***Redux:*** Das Team von Airbnb verwendete für die Speicherung des Zustandes der React Native Anwendung *Redux*. *Redux* verwaltet dabei den globalen Zustand einer Anwendung. Durch den Zugriff aller Komponenten auf diesen Zustand wird ein konsistenter Zustand

1 Der Lebenszyklus bzw. *Lifecycle* einer Anwendung beschriebt dessen aktuellen Zustand. Im Kontext einer App existieren mehrere mögliche Stadien. So kann eine App in diesem Moment ausgeführt werden, gerade geschlossen werden, sich gerade in der Hintergrundausführung befinden, o. ä. [8, 9]

2 Ist eine Anwendung Hot-Reloading-fähig, so können Entwickelnde die im Code gemachten Änderungen unmittelbar in einer Anwendung testen.

3 *react-native-reanimated* erlaubt das Erstellen von Animationen, welche auf dem UI-Thread des Gerätes ausgeführt werden, um möglichst flüssig dargestellt zu werden. [10]

der einzelnen Komponenten gewährleistet. Nachteilig bewertet wird jedoch die Tatsache, dass bei Redux das Verfassen von einem großen Anteil an Boilerplate Code vonnöten ist. Dies ist jedoch kein Problem mit React Native per se, sondern mit *Redux* selbst. Ebenso konnte gemeinsam genutzter Code der Web-App Airbnbs, in welcher ebenfalls *Redux* zum Einsatz kam, in React Native wiederverwendet werden, was den Initialaufwand minimierte. Hierbei ist erneut anzumerken, dass Redux seit 2016/2017 mit *Redux Toolkit* sehr viel Boilerplate Code reduzieren konnte (vgl. `https://redux-toolkit.js.org/`, abgerufen am 26. Oktober 2023).

9. **Native Module/Komponenten:** Überraschend für das zuständige Team bei Airbnb waren zusätzlich die Möglichkeiten zur Verbindung der nativen Apps und React Native. Die Integration von nativen Modulen (vgl. Abschnitt 4.2) oder nativen Komponenten (vgl. Abschnitt 4.5) in einige native Komponenten/Funktionalitäten ermöglichte deren Verwendung im Umfeld von React Native. Die Möglichkeiten, die sich durch die Verwendung von nativen Modulen und nativen Komponenten ergaben, übertrafen dabei Peals ursprüngliche Erwartungen.

10. **Web-Tools:** Airbnb konnte bereits im Web-Umfeld Erfahrungen mit React sammeln. Folglich war es möglich, die gleichen statischen Analysetools zu verwenden. Die Integration von React Native ermöglichte aufgrund ihrer Neuartigkeit das Potenzial, zuvor unbekannte Tools zu evaluieren. Diese wiesen Synergien zu Airbnbs Web-Projekten auf, wodurch diese ebenfalls durch diese Tools angereichert werden konnten.

11. **JavaScript Bibliotheken:** Die Tatsache, dass React Native eine JavaScript Engine nutzt, erlaubt meist die Verwendung von klassischen JavaScript-Bibliotheken, selbst wenn diese nicht explizit für die Verwendung mit React Native entwickelt wurden. *Redux* zählt z. B. zu diesen Bibliotheken, aber auch das Testing-Framework *Jest* (vgl. `https://jestjs.io/`, abgerufen am 26. Oktober 2023).

12. **CSS:** In React Native kommt *Yoga* (vgl. Kapitel 4) zur Interpretation von *Cascading Style Sheets* (CSS) inklusive der Flexbox API zum Einsatz. Auch wenn durch das Fehlen von einigen CSS-Features Probleme entstanden, wird *Yoga* von Peal dennoch positiv hervorgehoben. [6]

2.1.3 Misslungene Aspekte der Integration

13. **Technischer Fortschritt:** Zunächst kritisierte Peal, dass React Native zum Zeitpunkt der Integration in technischen Aspekten nicht ausreichend fortgeschritten war. Dies führte dazu, dass einige Aufgaben in React Native deutlich komplizierter zu lösen waren, als in den nativen Apps. Laut Peal waren dafür einige Fallstricke verantwortlich, welche zu Beginn des Projekts nicht ersichtlich waren. Dabei war nicht absehbar, ob die Lösung

des Problems einen Tag oder eine Woche in Anspruch nehmen würde. Mitunter durch diese Fallstricke, war es für Airbnb notwendig, einen Fork React Natives zu erstellen. Auf diesem Fork musste Airbnb einige Fehler von React Native eigenständig beheben. [7]

14. **JavaScript Engine:** Vor 2021 wurde in den iOS und Android Apps zum Ausführen des JavaScript-Codes standardmäßig die JavaScript-Engine des Apple Safari-Browsers namens *JavaScriptCore* verwendet. Dabei erfordert die Verwendung von *JavaScriptCore* unter iOS keine zusätzlichen Abhängigkeiten, da es bereits in den Safari-Browser integriert ist. Unter Android hingegen ist es notwendig, *JavaScriptCore* als Abhängigkeit einzufügen. Besonders aufgrund der Tatsache, dass die in Android integrierte Version von *JavaScriptCore* sehr alt war, kam es zu Inkonsistenzen zwischen iOS und Android. Um dieses Problem zu lösen war es notwendig, eine eigene, modernisierte Version von *JavaScriptCore* einzupflegen. Weiterhin ist es möglich, den JavaScript Code mit dem Browser *Chrome* zu debuggen. In diesem Szenario wird jedoch Chromes V8-Engine für die Ausführung des JavaScript-Codes verwendet und nicht das auf dem Gerät befindliche *JavaScriptCore*. Auch hierbei konnte es zur unterschiedlichen Ausführung des JavaScript-Codes kommen, da Chrome nicht dieselbe JavaScript-Engine verwendet. Dies führte zu Problemen, die nur bei der Verwendung der einen Engine auftraten, aber nicht bei der anderen. Dies erschwerte die Fehleranalyse erheblich. [7, 11]

15. **React Native Bibliotheken:** Ein weiteres Manko laut Peal waren die von der Community betriebenen, quelloffenen Bibliotheken für React Native. Insbesondere solche, die native Module (vgl. Abschnitt 4.2) oder native Komponenten (vgl. Abschnitt 4.5) bereitstellen. So waren häufig Inkonsistenzen zwischen iOS und Android aufgetreten. Peal geht davon aus, dass Entwickelnde dieser Bibliotheken entweder mit iOS oder Android besser vertraut waren und somit eine Implementierung der nativen Module/Komponenten nicht ausreichend berücksichtigt wurde. Weiterhin waren einige Bibliotheken so konfiguriert, dass sie den Ordner `node_modules` über einen relativen Pfad lokalisierten. Dadurch traten Fehler auf, wenn sich der Ordner `node_modules` nicht genau an dem vorgesehenen Ort befand.

16. **Native Module:** Da Airbnb bereits über eine umfangreiche Code-Basis verfügte, mussten oft native Module als Brücken zwischen React Native und den nativen Apps verwendet oder die benötigte Funktionalität in React Native neu implementiert werden. Dies verzögerte die Entwicklung der eigentlichen Funktionen signifikant.

17. **Absturzauswertung:** Die von Airbnb eingesetzte Bibliothek zur Auswertung von Abstürzen im Produktivumfeld besaß zunächst keine Möglichkeit zur Integration in React Native. Aus diesem Grund musste diese zusammen mit Entwickelnden der Bibliothek implementiert werden. Dabei scheiterte zeitweise das Hochladen der Fehlermeldungen.

Ebenso waren Fehlermeldungen besonders schwer zu analysieren, welche sich sowohl über die nativen Apps als auch über React Native erstreckten.

18. **Abstürze nativer Module:** Die Implementierung der nativen Module stellte für Entwickelnde von Airbnb eine kontraintuitive und mühsame Aufgabe dar. Kam es bei der Verwendung von nativen Modulen zu Fehlern, so stürzte die Android App ab, die iOS App hingegen gelegentlich nicht. Dies erschwerte die Fehlersuche unter iOS signifikant, da keine Fehlermeldung existierte, die Aufschluss über die mögliche Fehlerursache geben konnte. Ebenso wurden Zahlenwerte häufiger in Zeichenketten umgewandelt, obschon dies nicht gewünscht war. Ebenso wurde die Abwesenheit der Typisierung von native Modulen als Faktor identifiziert, der den Entwicklungsprozess beeinträchtigte.

19. **Startzeiten:** In Peals Bewertung war die initiale Ladezeit von React Native inakzeptabel. Dabei dauerte es mehrere Sekunden, bis React Native vollständig initialisiert war. Dies machte React Native für die Verwendung in Startbildschirmen unbrauchbar. Durch die vorzeitige Initialisierung von React Native beim Start der nativen Apps konnte dieses Problem jedoch umgangen werden.

20. **Appgröße:** React Native hatte einen nicht vernachlässigbaren Einfluss auf die Größe der Apps. Das Konglomerat an Java, JavaScript und anderen internen React Native Komponenten besaß unter Android eine Größe von ungefähr 8 MB. In einer APK[1] – die alle Plattformen bedienen kann, sogar 12 MB. Das Erstellen einer 64-Bit-Anwendungsdatei war zu diesem Zeitpunkt nicht möglich.

21. **Gestensteuerung:** Aufgrund des Fehlens einer Bibliothek zur einheitlichen Verwendung von Gesten in React Native Komponenten konnte React Native nicht eingesetzt werden, sofern eine komplexere Gestensteuerung benötigt wurde.

22. **Listen:** Der Umgang mit längeren Listen stellte sich ebenfalls als problematisch heraus. So existierte zwar eine Bibliothek für die Verwendung von virtualisierten[2] Listen, welche jedoch im Vergleich zu nativen virtualisierten Listen technisch weniger fortgeschritten waren und so nicht die gleichen Leistungsoptimierungen erzielen konnten.

23. **Updates:** Während die meisten Updates von React Native problemlos verliefen, gab es ein spezielles Update, das erhebliche Schwierigkeiten bereitete. Grund hierfür war, dass die Zielversion eine Alpha/Beta-Version von React voraussetzte, welche von einigen anderen Bibliotheken, die React als Abhängigkeit besitzen, nicht unterstützt wurde.

1 ,

2 Klassische Listen zeigen auch Listenelemente an, welche außerhalb des Bildschirms liegen. Virtualisierte Listen hingegen zeigen lediglich Listenelemente an, welche auch vom Nutzenden gesehen werden können. Dies hat meistens eine bessere Performance zur Folge.

24. **Zugänglichkeit:** Parallel zur Integration von React Native, führte Airbnb ein Programm zu Verbesserung der Zugänglichkeit ihrer Anwendungen durch. In Bezug auf die Umsetzbarkeit der Barrierefreiheit erwiesen sich die Möglichkeiten von React Native als unzureichend. So wurden einige Fehler in der Zugänglichkeits-API aufgedeckt, welche von Airbnb innerhalb des Forks behoben werden mussten.

25. **Hintergrundprozess:** Unter Android besteht die Möglichkeit, dass der Hintergrundprozess der App jederzeit beendet wird. Sollte es dazu kommen, ist es möglich, Daten in einem *Bundle*[1] zwischenzuspeichern. Die Speicherlogik muss dabei synchron abgehandelt werden. Ein Zugriff auf den *Redux*-State kann jedoch nur innerhalb des JavaScript-Prozesses erfolgen, wodurch lediglich ein asynchroner Zugriff über native Module umsetzbar ist. Ein zu großer *Redux*-State kann dazu führen, dass dieser nicht in einem *Bundle* gespeichert werden kann, was einen Absturz der nativen App zur Folge haben kann. [7]

2.2 Fallstudie CURSOR Software AG

Die CURSOR Software AG startete 2023 ebenfalls eine *Brownfield*-Integration React Natives. Zunächst war eine *Greenfield*-Entwicklung geplant, welche jedoch aufgrund der hohen Unsicherheiten und damit verbundenen Risiken wie auch dem enormen Aufwand verworfen wurde.

2.2.1 Ausgangssituation

Zunächst wurde ein Performance-Vergleich zwischen nativen Apps, dem Ionic Framework sowie React Native unternommen. [14] Aufgrund der daraus gewonnenen Erkenntnisse sowie der Affinität einiger Mitarbeitenden der CURSOR Software AG mit React ist die Wahl letztlich auf React Native gefallen. Die Projektphase des Autors dieser Arbeit umfasste dabei den Start der Integration und den Austausch einer zentralen Komponente der nativen Apps mit React Native. Dabei konnten bereits erste Eindrücke mit React Native in einer *Brownfield*-Entwicklung gesammelt werden. Während Airbnb das Projekt abgebrochen hat, befindet sich das Integrationsprojekt der CURSOR Software AG weiterhin in der Entwicklung. Folglich ist das Projekt zum aktuellen Zeitpunkt weder fehlgeschlagen noch gelungen. Dennoch konnten während der Entwicklungszeit einige positive Aspekte der *Brownfield*-Entwicklung mit React Native als auch Herausfor-

1 Ein *Bundle* definiert sich durch eine Zuordnung von einem *Key* zu einem Wert, wobei der Wert stets eine Instanz von *Parcelable* ist. [12] Ein *Parcelable* erlaubt dabei das Speichern und Wiederherstellen eines Objekts durch das Speichern von Werten in einem *Parcel*. [13]

derungen aufgedeckt werden. Im Folgenden werden diese analog zum Erfahrungsbericht Peals (vgl. Abschnitt 2.1) thematisiert.

2.2.2 Gelungene Aspekte der Integration

1. **Plattformspezifischer Code:** Während der Entwicklung von React Native Komponenten war es in keinem Fall notwendig, plattformspezifische Komponenten zu verfassen. Nur an wenigen Stellen innerhalb einzelner Komponenten war es notwendig, kleinere Unterschiede zwischen iOS und Android zu berücksichtigen, z. B. bei Animationen, Berechtigungen oder bei dem Verschieben von UI-Elementen, sobald die Tastatur geöffnet wird.

2. **Design:** Da beide bestehenden Apps auf das von Google entwickelte Material Design[1] setzen, sollte auch React Native dieses Designkonzept verwenden. Dafür wurde die Sammlung von UI-Elementen namens *React Native Paper* verwendet. (vgl. `https://reactnativepaper.com/`, abgerufen am 26. Oktober 2023)

4. **Hot-Reloading**: React Native beeindruckte mit seiner Hot-Reload Fähigkeit. Dabei ist es möglich, sowohl die iOS als auch die Android App über einen einzigen Entwicklungs-Server zu betreiben. Das Experimentieren mit dem UI wird durch die gleichzeitige Aktualisierung beider Apps bei Änderungen erheblich vereinfacht. Diese Funktionalität beeindruckte dabei besonders Entwickelnde der nativen iOS und Android Apps im Unternehmen. Aufgrund der geringen Größe des JavaScript-Bundles dauerte ein Hot-Reload dabei weniger als eine Sekunde.

6. **Performance:** Obwohl die Performance größtenteils zufriedenstellend war, besteht dennoch Verbesserungspotenzial. (vgl. Abschnitt 2.2.3)

8. ***Redux:*** Ebenso wie bei Airbnb kommt dabei für die Verwaltung eines globalen Zustandes *Redux* zum Einsatz. Überdies wird *Redux Toolkit* verwendet, was dabei hilft, die notwendige Menge an Boilerplate-Code drastisch zu reduzieren. *Redux* wird darüber hinaus mit *Redux Persist* erweitert. *Redux Persist* erlaubt das Persistieren des *Redux*-Zustandes auf dem Gerät des Nutzenden, wodurch die Umsetzung eines Offline-Betriebs sichergestellt werden kann. Dieser Ansatz eliminiert darüber hinaus die Notwendigkeit, den Zustand von React Native innerhalb eines *Bundle*s unter Android zwischenzuspeichern, da dieser stets in den lokalen Speicher der App gesichert wird.

9. **Native Module/Komponenten:** Die mit React Native verfügbare API in Form von nativen Modulen oder nativen Komponenten war ebenfalls überzeugend. So war es

1 Material Design ist eine von Google definierte Designsprache. Diese kommt dabei besonders in nativen Android Apps zum Einsatz, ist aber prinzipiell plattformunabhängig. [15]

durch diese API möglich, die Offline-Daten React Natives und der nativen App zu synchronisieren. (vgl. Abschnitt 6.3) Native Module in Verbindung mit *Event Emittern* wurden dabei zusätzlich verwendet, um Aktionen z. B. nach dem Log-in, Log-out etc. in React Native auszuführen. Dies ermöglichte z. B. das Vorladen von Offline-Daten oder das Bereinigen von Daten nach dem Log-out. Weiterhin war es mit der Verwendung von nativen Modulen möglich, neue Ebenen/Views der nativen App zu öffnen, welche bislang nicht mit React Native ausgetauscht wurden.

11. **JavaScript Bibliotheken:** Bemerkenswert war die Möglichkeit des Verwendens von klassischen JavaScript-Bibliotheken. So wurde z. B. für den Abruf von Daten einer REST-API die Bibliothek *Axios*[1] verwendet. *Axios* wird dabei ebenfalls in den Web-Projekten der CURSOR Software AG verwendet. So war es möglich, die React Native App möglichst ähnlich zu den Web-Projekten zu gestalten, um eine ähnliche DX sicherzustellen.

12. **CSS:** Besonders vorteilhaft war die Möglichkeit, größtenteils standardmäßiges CSS zu verwenden. Bedauerlicherweise war das CSS-Property *gap* zu Beginn der Integration nicht implementiert. Dieses wurde jedoch in die nächste Version der CSS-Engine *Yoga* integriert. Als einziges Defizit ist zu erwähnen, dass das CSS-Grid Layout weiterhin nicht in *Yoga* umgesetzt worden ist.

26. **TypeScript:** Die Kombination von TypeScript und React Native verlief ohne Schwierigkeiten. TypeScript ermöglicht es darüber hinaus, Interfaces für native Module zu definieren, um auch beim Zugriff auf Methoden der nativen Module typsicher zu agieren. (vgl. Abschnitt 4.2.3)

2.2.3 Misslungene Aspekte der Integration

27. **Performance:** Aus Sicht des Autors liegt das größte Verbesserungspotenzial React Natives in der Performance bzw. der Möglichkeiten der Performanceoptimierung durch Entwickelnde. So weist die Performance im Debug-Modus extreme Unterschiede zum Produktions-Modus auf. Die Dokumentation gibt an, dass dies ein unvermeidbarer Umstand ist. [16] Dabei ist die Android App im Debug-Modus teilweise sehr schwer bedienbar, während sie im Produktions-Modus performant ist. Unter iOS war der Performance-Unterschied dabei hingegen deutlich geringer als unter Android, da die Performance im Debug-Modus deutlich besser ist.

1 *Axios* ist ein HTTP client, welcher für den Abruf von REST-APIs verwendet werden kann. *Axios* baut dabei auf einen bereits in Node.JS integrierten HTTP client auf und reichert diesen durch weitere Funktionen, wie z. B. einen *Promise*-basierten Ansatz an. (vgl. `https://axios-http.com/docs/intro`, abgerufen am 26. Oktober 2023)

28. **Bild-Komponente:** Zudem wurde ein Fehler in React Native bei der Verwendung der `<Image />`-Komponente festgestellt. So verhindert ein # im Dateipfad das Laden eines lokalen Bildes in einer `<Image />`-Komponente. Für diesen Fehler wurde ein Bug-Report angelegt, welcher seit Monaten keine Aktivität aufweist. (vgl. `https://github.com/facebook/react-native/issues/37592`, abgerufen am 26. Oktober 2023) Obschon der Fehler selbst weniger kritisch ist, ist die lange Wartezeit auf eine Antwort der Entwickelnden des Open-Source-Frameworks unerfreulich.

29. **Debugging:** Die Integration des Debug-Tools *Flipper* ist leider sehr aufwendig oder gar nicht erst möglich. So existiert für iOS keine Möglichkeit *Flipper* zu verwenden, sollte im *Podfile* `use_frameworks!` zum Einsatz kommen. Das dazugehörige GitHub-Issue[1] existiert dabei seit Mitte 2021, wurde jedoch bisher nicht behoben. Per se bietet *Flipper* dennoch sehr hilfreiche Features wie die React-Debugging-Tools, aber auch die Möglichkeit zum Installieren von Plugins. Eines dieser Plugins ermöglicht z. B. das Inspizieren des *Redux*-States. Neben *Flipper* ist es darüber hinaus möglich, die React-Debugging-Tools im Browser aufzurufen. Diese Methode erwies sich jedoch als unzuverlässig.

30. **HTML-Tags:** Ein weiterer Nachteil in der Funktionsweise von React Native gegenüber React ist, dass HTML-Tags wie `<div>`, `<span>`, o. Ä. nicht verwendet werden können. In React Native werden dafür z. B. `<View>`-Komponenten bereitgestellt. Diese Umstellung erfordert zu Beginn etwas Anpassung. Die Umstellung ist jedoch schnell verinnerlicht.

2.3 Gegenüberstellung der Erfahrungsberichte

Zunächst kann festgestellt werden, dass sich beide Erfahrungsberichte unterscheiden, besonders die Motivationen der React Native Integration. So war bei Airbnb zunächst das Ziel, einzelne, neue Komponenten mit React Native zu implementieren. Bei CURSOR hingegen ist das Ziel, bereits bestehende Komponenten nach einem iterativen Vorgehensmodell auszutauschen. Der Integrationsansatz Airbnbs sieht somit nicht zwangsläufig vor, die bereits bestehende App zeitnah zu ersetzen. Dementsprechend wurden die bestehenden nativen Apps während der Integration von React Native nicht nur instand gehalten, sondern darüber hinaus kontinuierlich weiterentwickelt. Demzufolge unterscheiden sich die Integrationsarten sowie die Integrationsmotivationen signifikant.

1 vgl. `https://github.com/facebook/flipper/issues/2414`, abgerufen am 26. Oktober 2023

Weiterhin ist festzuhalten, dass an der Integration von Airbnb weitaus mehr Entwickelnde beteiligt waren als bei der CURSOR Software AG.

Zusätzlich ist der zeitliche Aspekt hervorzuheben. So wurde die Integration von React Native bei Airbnb 2016/2017 durchgeführt, während die Integration der CURSOR Software AG 2023 begann. Dabei ergaben sich einige grundlegende Änderungen innerhalb von React Native in dem Zeitraum von 2017 bis 2023. So wurde z. B. einige Tage vor der Veröffentlichung des Artikels über die misslungene React Native Integration Airbnbs ein Artikel der React Native Entwicklerin Sophie Alpert veröffentlicht, welcher die neue Architektur (vgl. Kapitel 4) React Natives ankündigt. [17]

2.3.1 Gelungene Aspekte der Integrationen

Bei beiden Integrationen war ein Verfassen von plattformspezifischem Code (vgl. Aspekt 1) entweder gar nicht oder nur selten notwendig. Weiterhin konnte in beiden Fällen eine passende Design-Bibliothek (vgl. Aspekt 2) gefunden oder selbst implementiert werden. Die Hot-Reload Möglichkeit (vgl. Aspekt 4) überzeugte sowohl Airbnb als auch CURSOR. Bei beiden Integrationen kam *Redux* (vgl. Aspekt 8) als Zustandsmanagement Bibliothek zum Einsatz. Bei CURSOR wurde allerdings zusätzlich `redux-toolkit` sowie `redux-persist` verwendet. Der Autor dieser Thesis wie auch das Entwicklerteam Airbnbs waren gleichermaßen positiv über die Möglichkeiten der nativen Module wie auch nativen Komponenten (vgl. Aspekt 9) überrascht. Sowohl Airbnb als auch CURSOR heben die Bedeutung von *Yoga* für die Interpretation von CSS (vgl. Aspekt 12) hervor.

Größtenteils ist der Erfahrungsbericht der gelungenen Aspekte ähnlich. Es gibt vereinzelt Aspekte, welche nur von Airbnb oder von CURSOR hervorgehoben wurde. So hebt CURSOR z. B. die Verwendung von TypeScript (vgl. Aspekt 26) hervor, Airbnb tat dies nicht, da eine Verwendung von TypeScript nicht stattfand. Airbnb hingegen betont z. B. die Möglichkeiten zur Verwendung von Animationen (vgl. Aspekt 7). CURSOR verwendete diese jedoch nicht, weswegen diese nicht explizit erwähnt wurden.

Eine Übersicht der gelungenen Aspekte ist in Tabelle 2.1 zusammengestellt.

2.3.2 Misslungene Aspekte der Integrationen

Die misslungenen Aspekte weisen im Kontrast zu den gelungenen deutlich größere Unterschiede auf. Dies ist teilweise auf die Tatsache zurückzuführen, dass sich im Anschluss an den Abbruch des Projekts bei Airbnb innerhalb von React Native einige Veränderungen ergeben haben. Eine bedeutende Veränderung ist die Einführung von *Hermes*

#	Aspekt	Airbnb	CURSOR
1	Kein plattformspezifischer Code	+	++
2	Design System	+	++
3	Lebenszyklus	++	∅
4	Hot-Reloading	++	++
5	Brückenelemente	+	∅
6	Performance	++	+
7	Animationen	++	?
8	Globaler Zustand	+	++
9	Kommunikation mit nativen Apps	++	++
10	Neuartige Tools	++	?
11	JavaScript Drittbibliotheken	++	++
12	CSS	+	+
26	TypeScript	?	++

++/+: (sehr) positiv
?: Nicht überprüft
∅: Nicht zutreffend

Tabelle 2.1: Übersicht der positiven Aspekte der verglichenen Fallstudien.

als Standard-JavaScript-Engine (vgl. Kapitel 4). Des Weiteren ist zu berücksichtigen, dass die Integration von CURSOR bislang nicht im Produktivbetrieb eingesetzt wird. Aus diesem Grund können noch keine erschöpfenden Aussagen über die Auswertung von Fehlermeldungen (vgl. Aspekt 17) in den produktiven Apps sowie die Auswirkung auf die App-Größe (vgl. Aspekt 20) getroffen werden.

Dennoch sind einige der negativen Aspekte der Airbnb Integration mit aktuelleren Versionen von React Native behoben. So wird seit der Version `0.64` die Verwendung der Hermes JavaScript-Engine unterstützt. (vgl. Kapitel 4) Diese ist dabei auf allen Plattformen identisch. Ebenfalls verspricht Hermes, die App-Größe (vgl. Aspekt 20) zu minimieren. [18] Dadurch wird verhindert, dass ungleiche Versionsstände der JavaScript-Engine (vgl. Aspekt 14) entstehen, wie sie bei der Verwendung von *JavaScriptCore* auftauchten. Zusätzlich kann die Ausführung des JavaScript-Codes innerhalb von Hermes debuggt werden, wodurch die Ausführung innerhalb der V8-Engine von Chrome nicht mehr notwendig ist.

Durch den direkten Zugriff auf Werte bei der Verwendung von Turbo Native Modules (vgl. Abschnitt 4.4), sind Konvertierungsfehler, die durch eine Serialisierung ausgelöst werden, ausgeschlossen. Ebenso bieten Turbo Native Modules durch die Generierung von Interfaces eine Typisierung (vgl. Aspekt 18) der definierten Methoden, wodurch Fehler in der Verwendung mitigiert werden. (vgl. Abschnitt 4.4)

Die initiale Ladezeit (vgl. Aspekt 19) von React Native soll durch *Lazy Loading* der nativen Module beschleunigt werden. (vgl. Abschnitt 4.1.3)

Für die Verwendung von etwaigen Gestensteuerungen (vgl. Aspekt 21) ist es möglich, die Drittbibliothek `react-native-gesture-handler` zu verwenden. (vgl. `https://github.com/software-mansion/react-native-gesture-handler`, abgerufen am 26. Oktober 2023) Die Bibliothek wird dabei ebenfalls im Blog-Eintrag Airbnbs erwähnt. Diese war jedoch zu dieser Zeit noch in einem sehr frühen Entwicklungsstadium.

Für die Verwendung von langen Listen (vgl. Aspekt 22) kann die Komponente `<FlatList />` verwendet werden. `<FlatList />` verwendet dabei intern `<VirtualizedList />`. Die Virtualisierung von Listen ermöglicht eine erhebliche Verbesserung der Arbeitsspeichernutzung und Performance von größeren Listen, indem nur die auf dem Bildschirm sichtbaren Listenelemente von React Native gerendert werden. Elemente, die nach entsprechenden Scrollgesten auf dem Bildschirm sichtbar werden, werden dynamisch nachgeladen. [19, 20] Dafür ist es nicht notwendig, eine externe Bibliothek zu installieren, da `<FlatList />` sowie `<VirtualizedList />` bereits in React Native enthalten sind.

Eine tabellarische Gegenüberstellung der misslungenen Aspekte ist in Tabelle 2.2 zu finden.

#	Aspekt	Airbnb	CURSOR
13	Technische Ausgereiftheit	−−	∅
14	JavaScript Engine	−−	∅
15	React Native Drittbibliotheken	−−	∅
16	Menge der Brückenelemente	−−	∅
17	Auswertung von Abstürzen	-	?
18	Implementierung von nativen Modulen	−−	∅
19	Startzeiten	−−	∅
20	Größe der Apps	−−	?
21	Gestensteuerung	−−	∅
22	Längere Listen	-	∅
23	React Native Updates	-	∅
24	Barrierefreiheit	−−	?
25	Speicherung des Zustandes	−−	∅
27	Performance	?	-
28	Image-Komponente	?	-
29	Debugging	?	-
30	HTML-Tags	?	-

−−/-: (sehr) negativ
?: Nicht überprüft
∅: Nicht zutreffend

Tabelle 2.2: Übersicht der negativen Aspekte der verglichenen Fallstudien.

3 React

React ist ein quelloffenes JavaScript-Paket für die Entwicklung von Webanwendungen. React wird dabei federführend von Meta[1] entwickelt und in Metas Webanwendungen *Facebook* sowie *Instagram* verwendet. Ursprünglich wurde React von Jordan Walke entwickelt, welcher zu dieser Zeit bei Facebook Ads angestellt war. [21, 22]

In der jährlichen Umfrage von Stack Overflow antworteten 2023 40,57 % der Befragten auf die Frage, mit welchem Web Framework oder welcher Webtechnologie sie über das Jahr 2022 umfangreich gearbeitet haben, mit React. Die Umfrage zeigt ebenfalls, dass die Beliebtheit von React ungefähr doppelt so hoch ist wie die von JQuery, Express und Angular. [23]

Da der Fokus dieser Arbeit auf der Integration von React Native in bestehende Apps liegt, werden im Folgenden – aus Sicht des Autors – nur einige wichtige Grundkonzepte von React beschrieben. Für einen tieferen Einblick in die Bibliothek React, empfiehlt sich die Fachliteratur von Nils Hartmann und Oliver Zeigermann (vgl. [21]), Elad Elrom (vgl. [22]) sowie praxisnahe Video-Serien von Vishwas Gopinath (vgl. [24, 25, 26]).

3.1 React Komponenten

Um eine optimale Wiederverwendbarkeit zu gewährleisten, sollte eine React-Komponente einen kleinen, abgeschlossenen Teil des User-Interfaces (UI) abbilden. Komponenten ist es dabei möglich, andere React Komponenten zu verwenden oder gar zu erweitern. Dieses Verschachtelungsprinzip von React-Komponenten ermöglicht es schließlich, eine umfangreiche Gesamtanwendung zu konstruieren. In der React Dokumentation wird dieses Konzept prägnant zusammengefasst:

‘Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.’[2] [27]

1 Ehemals Facebook

2 Deutsch: „Mit Komponenten können Benutzeroberflächen in unabhängige, wiederverwendbare Teile separiert werden, wobei jedes Teil isoliert betrachtet werden kann.“

React Komponenten besitzen in der Regel einen Zustand. Dieser Zustand entscheidet dabei, wie die Komponente präsentiert wird. Ändert sich etwa der Wert einer Variable einer Komponente, so *reagiert* die Darstellung auf ebendiese Änderung. Folglich wird sich am Programmierparadigma der reaktiven Programmierung bedient, wodurch sich auch der Name *React* ableiten lässt. Ein wichtiger Aspekt, den es bei der Erstellung von React-Komponenten zu beachten gilt, ist, dass ein gegebener Zustand immer dieselbe Darstellung hervorbringen muss. So bildet eine Komponente eine Schablone, in die der Zustand eingebettet wird. Ein großer Vorteil dieser Eigenschaft ist die einfache Testbarkeit: Einzelne Komponenten können mit verschiedenen Zuständen aufgerufen werden und müssen immer ein fest definierbares Ergebnis liefern. [21, 22]

Um die Darstellung einer React Komponenten festlegen zu können, ist es notwendig *JSX* zu verwenden. *JSX* wird dabei in Abschnitt 3.2 gesondert erläutert.

Weiterhin ist es React Komponenten möglich, Übergabeparameter entgegenzunehmen. Im React-Umfeld werden diese als *Props* bezeichnet. *Props* sind dabei essenzielle Aspekte Reacts, weswegen diese in Abschnitt 3.3 im Detail behandelt werden.

Bei der Definition von Komponenten in React wird zwischen funktionalen Komponenten und Klassen-Komponenten unterschieden. Wie die Namen bereits vermuten lassen, bedienen sich funktionale Komponenten dem Programmierparadigma der funktionalen Programmierung, während Klassen-Komponenten in der objektorientierten Programmierung angesiedelt sind. [21, 22] Beide Konzepte werden im Folgenden genauer betrachtet. (vgl. Abschnitt 3.4 und 3.5) Aufgrund der Empfehlung der React-Dokumentation, bevorzugt funktionale Komponenten zu verwenden [28](vgl. orangefarbener Kasten/*Callout*), wird die Darstellung von Klassen-Komponenten lediglich oberflächlich erfolgen.

3.2 JSX

Im Gegensatz zu anderen UI-Frameworks verwendet React in der Entwicklung kein HTML, sondern vorzugsweise *JSX*[1]. *JSX* ist dabei syntaktisch verwandt zu HTML sowie XML und lässt sich durch JavaScript Ausdrücke anreichern. (siehe Quellcode 3.1) [21]

JSX ist dabei nicht in der Syntax von JavaScript[2] definiert. *JSX* ist somit eine sprachliche Erweiterung von JavaScript, welche von *Meta* ursprünglich für React entwickelt wurde. Prinzipiell ist es möglich, React ohne *JSX* zu verwenden. Der Grund dafür ist, dass *JSX* in klassisches JavaScript transpiliert werden muss. Dazu wird das *JSX*

1 *JSX* steht für JavaScript Syntax Extension
2 Bzw. genauer: In der Syntax von ECMAScript

in Funktionsaufrufe umgewandelt, welche wiederum letztlich zu JavaScript-Objekten evaluieren. Folglich ist es möglich, diese Funktionsaufrufe direkt, anstelle von *JSX* zu verwenden. (siehe Quellcode 3.1) [29, 30]

```
1  const beispielName = "Max Mustermann";
2  const beispiel1 = <h1 style={{ color: "red" }}>Hallo {beispielName}!</h1>;
3  const beispiel2 = React.createElement(
4    "h1",
5    { style: { color: "red" } },
6    `Hallo ${beispielName}!`
7  );
8
9  function beispiel3(verschachtelt: boolean = true) {
10   return (
11     <div>
12       <h1>Ich</h1>
13       <h2>bin</h2>
14       <h3>{verschachtelt ? "verschachtelt" : "nicht verschachtelt"}</h3>
15     </div>
16   );
17 }
```

Quellcode 3.1: Beispiele der *JSX*-Syntax (Beispiel 1 und 3) sowie der Definition eines React Elements ohne die Verwendung von JSX (Beispiel 2). Beispiel 1 und 2 sind dabei äquivalent.

Da klassische Browser lediglich HTML, CSS und JavaScript verarbeiten können, ist es notwendig, das *JSX* zur Laufzeit in eine vom Browser verarbeitbare Syntax zu überführen. Dabei ist es möglich diese Umwandlung entweder auf der Seite des Clients – also z. B. im Webbrowser an sich – oder auf einem Server durchzuführen. In der Regel[1] erfolgt die Umwandlung im Client.

Eine Zustandsänderung einer React Komponente erfordert in der Regel[2] eine Aktualisierung des DOMs[3], um die Änderungen in der Darstellung der React Komponente innerhalb des Browsers zu reflektieren. React geht dabei jedoch hocheffizient vor und aktualisiert lediglich DOM-Elemente, welche auch notwendigerweise angepasst werden müssen. [21, 29]

3.3 Props

Die *Props* einer Komponente beschreiben Übergabeparameter, die einer React Komponente zur Verfügung gestellt werden können. *Props* ist dabei die Kurzform von *Properties*.

1 Jedenfalls so lange, bis die Anwendungen für den Webbrowser zu rechenintensiv werden.
2 Bei der Verwendung von z. B. Memorierten Komponenten oder ähnlichen Optimierungen ist dies nicht immer der Fall.
3 DOM = Document Object Model, beschreibt die Darstellung von HTML-Tags im Browser

Die Übergabe von *Props* an eine Komponente ist dabei vergleichbar mit der Festlegung von HTML-Attributen. Konkrete Werte werden jedoch in geschweifte Klammern übergeben, statt innerhalb von Anführungszeichen[1]. Somit ist es möglich – innerhalb der geschweiften Klammern –, Werte von Variablen der aufrufenden Komponente zu übergeben. Auf *Props* ist dabei stets nur lesend zuzugreifen. Ein *Prop* kann dabei verschiedenste Datentypen annehmen, darunter fallen alle primitive Datentypen sowie Objekte oder gar Funktionen. Letzteres wird verwendet, um *Callbacks*[2] an eine Komponente zu übergeben. Beispiel der Übergabe von *Props* an eine Komponente: `<Button title="Click me!" onClick={() => console.log("clicked")} style={{color: "yellow"}} />`. [3]

Zusätzlich zu den innerhalb einer Komponente frei definierbaren *Props* definiert React einige *Props* mit spezifischen, internen Aufgaben.

Einer der vorgegebenen *Props* ist der *children-Prop*. *JSX*-Komponenten, welche sich zwischen einem öffnenden und schließenden *JSX*-Tag befinden, werden implizit in den *children-Prop* übertragen. Ansonsten ist es möglich, den *children-Prop* über die Attributssyntax zu definieren: `<Text>Dies wird zum children Prop</Text>` oder `<Text children="children Text" />`. Um eine bestmögliche Übersicht innerhalb des JSXs zu gewährleisten, wird jedoch von dem expliziten Setzen des *children-Prop* abgeraten. Wird eine Komponente mit dem *children-Prop* aufgerufen, kann diese auf die darin enthaltenden React Komponenten zugreifen. [27, 31]

Ein weiterer interner *Prop* ist der *key-Prop*. Im Gegensatz zum *children-Prop* ist dieser nicht in der aufgerufenen Komponente verfügbar, sondern wird lediglich intern von React verwendet. Der *key-Prop* sollte dabei stets dann an eine Komponente übergeben werden, sobald diese innerhalb eines *map*-Streams, *filter*-Streams, einer For-Schleife oder ähnlichem initialisiert wird. Der Grund dafür ist, dass React unter Zuhilfenahme des *key-Props* einer Komponente bestimmen kann, welche der Komponenten bei Zustandsänderungen erneut gerendert werden muss. Ohne die Definition des *key-Props*, muss im Zweifel die gesamte Liste an Komponenten neu gerendert werden, was in der Regel zu einer degradierten Performance führt. Beispiel: `[1,2,3].map((value, index) => <p key={index}>Wert: {value}</p>)`. Die Wahl des richtigen *Keys* ist hierbei entscheidend. Im besten Fall sind als *Keys* unveränderbare Identifikatoren zu verwenden. Beispiele hier-

1 Eine Ausnahme sind hierbei String-Literale. Diese können mit Anführungszeichen, auch ohne geschweifte Klammern übergeben werden.

2 *Callbacks* (Deutsch: „Rückrufe“) sind Funktionen, die an eine externe Komponente übergeben werden, welche von dieser aufgerufen werden kann. Vereinfacht gesagt, vereinbart man mit der externen Komponente einen Rückruf, sobald ein bestimmtes Ereignis eintritt (z. B. wenn ein Button betätigt wird)

3 Hierbei ist wichtig zu beachten, dass bei der Übergabe eines Objektes (`style={{...}}`) zwei geschweifte Klammern verwendet werden. Die äußere geschweifte Klammer muss dabei verwendet werden, um die Zuweisung eines Wertes innerhalb dieser geschweiften Klammern anzudeuten. Die inneren geschweiften Klammern definieren dabei ein klassisches JavaScript Objekt.

für wären Datenbank-Identifikatoren, UUIDs o. Ä. Beinhalten die verwendeten Daten dabei keine dieser eindeutigen Identifikatoren, ist es möglich, lokale Identifikatoren zu generieren und zu verwenden. Der *index* einer Liste, wie im obigen Beispiel, eignet sich dabei jedoch nur in wenigen Situationen als *key*. Ausschließlich, wenn Daten lediglich am Ende der Liste eingefügt werden, ist die Verwendung des *index* als *Key* eine performante Wahl. Wird dabei inmitten der Liste ein Element hinzugefügt, hat dies zur Folge, dass der *index* der nachfolgenden Komponenten inkrementiert wird. Aufgrund der Änderung der *keys* werden somit ebendiese Elemente von React erneut gerendert, obschon sich die dahinter liegenden Daten nicht verändert haben. [32]

Weiterhin stehen allen HTML-Komponenten wie `<div>`, `<img>` ... deren HTML-Attribute als *Props* zur Verfügung. Dabei muss jedoch berücksichtigt werden, dass sich einige HTML-Attribute namentlich von korrespondierenden React *Props* unterscheiden. Z. B. ist dem `style-Prop` in React kein `String` zuzuweisen, sondern ein Objekt. CSS-Attribute innerhalb des `style-Props` sind dabei größtenteils identisch, es sei denn, sie beinhalten einen Bindestrich. Diese CSS-Attribute werden in *CamelCase* überführt[1]. Z. B. wird `text-align` zu `textAlign`: `<p style={{textAlign: "center"}}>mittig</p>` [27]

3.4 Funktionale Komponenten

Funktionale Komponenten sind als JavaScript Funktionen definiert, welche ein Objekt als Argument – das *Props*-Objekt (vgl. Abschnitt 3.3) – entgegennimmt und ein React Element zurückgeben. React Elemente sind dabei vereinfacht illustriert die Repräsentation der Komponente als DOM-Element[2]. (siehe Quellcode 3.2) Dabei können funktionale Komponenten entweder mit dem *Keyword* `function` oder unter der Verwendung anonymer *Arrow-Functions-Expressions* definiert werden. React Komponenten, die auf diese Weise definiert werden, werden von React mit den entsprechenden *Props* aufgerufen. Ein solcher Aufruf wird dabei als *Render* bzw. *Rerender*[3] bezeichnet. [33, 34, 27]

3.4.1 Hooks

Die zuvor definierte Komponente (vgl. Abschnitt 3.4 und Quellcode 3.2) besitzt zunächst keinen eigenen, internen Zustand. Der Zustand dieser Komponenten wurde lediglich durch bereitgestellte *Props* festgelegt.

1 Grund dafür ist, dass JavaScript keine Bindestriche in Variablennamen erlaubt.
2 Noch weiter vereinfacht: Das, was der Browser letztlich anzeigt.
3 Von einem *Rerender* wird gesprochen, wenn die Komponente bereits mindestens einmal *gerendert* wurde und anschließend erneut *gerendert* wird.

```tsx
import * as React from "react";

export function Greeter1({ children }: React.PropsWithChildren<{}>) {
  return <h1>Hi, {children}!</h1>;
}

export const Greeter2 = ({ children }: React.PropsWithChildren<{}>) => (
  <h1>Hi, {children}!</h1>
);

const usage1 = <Greeter1>John</Greeter1>;
const usage2 = <Greeter2>Doe</Greeter2>;
```

Quellcode 3.2: Sehr simple funktionale React Komponente. Nimmt innerhalb der *Props* ein Objekt namens *children* entgegen und zeigt dieses innerhalb eines `h1`-Tags an.

Es existieren bestimmte Anforderungen an eine Komponente, die die Verwaltung eines internen Zustands notwendig machen. Möchte man z. B. einen Button definieren, der die Zustände *on* und *off* annehmen kann, kann der aktuelle Zustand des Buttons in einem `boolean` gespeichert werden.

Ein naiver Ansatz wäre es nun, innerhalb der Methode eine Variable zu definieren, welche den Zustand des Buttons speichert. Dieser Ansatz ist in Quellcode 3.3 nach dem Beispiel des erwähnten Buttons illustriert.

```tsx
export function Switch() {
  let state = false;
  return (
    <button onClick={() => (state = !state)}>{state ? "on" : "off"}</button>
  );
}
```

Quellcode 3.3: Naiver Ansatz zur Speicherung des Zustandes.

Dieser Ansatz ist aus zweierlei Gründen unzulänglich. Zum einen sollte beim Verwenden von funktionalen Komponenten stets berücksichtigt werden, dass diese Funktionen von React jederzeit (z. B. bei Zustandsänderungen) neu aufgerufen werden können. Im naiven Beispiel hätte dies zur Folge, dass die lokale `state` Variable stets mit dem Initialwert `false` redeklariert werden würde. Somit könnte sich der Zustand durch einen erneuten Aufruf von React spontan ändern. Zum Anderen ist zu beachten, dass die Aktualisierung der Darstellung einer React Komponente lediglich dann erfolgt, wenn sie von React erneut aufgerufen wird. Obschon das Ereignis von `onClick` den lokalen Zustand verändert, wird lediglich eine lokale Variable innerhalb der Methode aktualisiert, weshalb React keine Änderung der Komponente feststellt und dementsprechend auch keinen erneuten Aufruf der Komponente durchführt.

Demzufolge ist der korrekte Ansatz zur Verwaltung eines lokalen Zustandes einer Komponente der *Hook* `useState`. *Hook*s sind dabei ebenfalls Funktionen, welche es unter anderem erlauben, sich in interne React Features „hineinzuhaken“[1]. Der *Hook* `useState` im Speziellen erlaubt dabei die Definition eines Zustandes innerhalb einer funktionalen React Komponente. In Quellcode 3.4 wurde das Beispiel aus Quellcode 3.3 angepasst, sodass statt einer lokalen Variable der *Hook* `useState` verwendet wird. Dabei wird der *Hook* `useState` mit dem Argument `false` aufgerufen. Dieses Argument bestimmt dabei den Initialwert des Zustandes. `useState` ist dabei generisch und erlaubt die Definition des Datentyps innerhalb von spitzen Klammern. Wird hingegen kein Typ angegeben, versucht der TypeScript Transpiler den Typ aus dem Initialwert zu inferieren. Im Beispiel wird der Typ dabei aus Demonstrationszwecken angegeben, obwohl dies in diesem Fall nicht zwingend notwendig wäre, da der Typ inferiert werden kann. Der *Hook* `useState` liefert dabei einen Tupel als Rückgabewert, in dem der erste Wert den gegenwärtigen Zustand widerspiegelt und der zweite Wert eine Funktion zur Zustandsänderung darstellt. Die Funktion für die Durchführung einer Zustandsänderung kann dabei direkt mit dem neuen Wert aufgerufen werden oder aber – wie im Beispiel zu sehen – mit einer Funktion, wobei der aktuelle Wert des Zustandes im ersten Parameter der Funktion bereitgestellt wird.

```tsx
export function Switch() {
  const [state, setState] = React.useState<boolean>(false);
  return (
    <button onClick={() => setState((oldState) => !oldState)}>
      {state ? "on" : "off"}
    </button>
  );
}
```

Quellcode 3.4: Korrekter Ansatz zur Speicherung des Zustandes.

Bei der Verwendung von *Hook*s ist es dabei besonders wichtig zu beachten, dass zu jedem Zeitpunkt die Anzahl der *Hook*s sowie deren Reihenfolge identisch bleiben muss. Zunächst scheint diese Anforderung trivial umsetzbar, in größeren Komponenten muss jedoch besonders darauf geachtet werden, dass z. B. kein frühzeitiges `return` vor einem *Hook* verwendet wird. Die in Quellcode 3.5 dargestellte Komponente verletzt diese Regel. Ändert sich `onText` oder `offText` von `undefined` zu einem Wert, der nicht `undefined` ist, so wird beim ersten Aufruf von `Switch` kein Hook und im zweiten Aufruf ein Hook aufgerufen. Das aufgezeigte Szenario mündet dabei in einem Laufzeitfehler der React Anwendung. Der Fehler tritt dabei jedoch in diesem konkreten Beispiel nur auf, wenn einer der `Props` `undefined` sein sollte. Somit sind diese Arten von *Hook*-Fehlern schwer auffindbar, da diese nur in bestimmten Szenarien auftreten

1 Im Englischen *to hook into*, woraus sich der Name *Hook* ableitet.

können. Folglich gilt es stets darauf zu achten, dass *Hook*s lediglich in gleicher Zahl und identischer Reihenfolge verwendet werden. [21, 35, p. 56 - 58]

```
export function Switch({
  onText,
  offText,
}: {
  onText?: string;
  offText?: string;
}) {
  if (!onText || !offText) {
    return;
  }
  const [state, setState] = React.useState<boolean>(false);
  return (
    <button onClick={() => setState((oldState) => !oldState)}>
      {state ? onText : offText}
    </button>
  );
}
```

Quellcode 3.5: Fehlerhafte Verwendung von *Hooks* durch die Verwendung eines frühzeitigen `returns`.

Ein weiterer wichtiger *Hook* ist `useEffect`. Dieser *Hook* erlaubt die Definition einer Funktion, die nur aufgerufen werden soll, wenn sich bestimmte Werte wie *Props*, aber auch Zustände einer Komponente ändern. Das erste Argument von `useEffect` ist dabei der Callback, der aufgerufen werden soll, während das zweite eine Liste an Werten ist, die dazu führen soll, dass der Callback aufgerufen wird. Diese Liste wird dabei *Dependency-Array* genannt. Wird der *Dependency-Array* leer gelassen, wird der Callback einmalig zur Initialisierung der Komponente ausgeführt. Gibt der Callback nicht `undefined`, sondern eine Funktion zurück, wird diese vor der nächsten Zustandsänderung ausgeführt oder aber, wenn die Komponente ausgehängt wird, wenn der *Dependency-Array* leer ist. Weiterhin existieren *Hooks* wie `useCallback` oder `useMemo`, die zum Memorieren einer Funktion bzw. eines Wertes verwendet werden können. Diese *Hook*s sind insbesondere für Performanceoptimierung von Nutzen. Sowohl `useCallback` als auch `useMemo` verwenden – analog zu `useEffect` – einen *Dependency-Array* als zweites Argument. Dies hat zur Folge, dass die memorierte Funktion bzw. der Wert lediglich bei Änderungen der Werte des *Dependency-Array* reevaluiert werden. Beispielhafte Verwendung der *Hooks* `useCallback` und `useMemo`: `const onSubmit = useCallback((text: string) => s⌋ ubmit(text), [submit]); const greeter = useMemo(() => \`Hallo ${name}\`, [name⌋ ]);`. [36]

Zusätzlich ist es möglich, eigene *Hook*s zu erstellen. Ein *Hook* ist als eine Funktion definiert, welche stets mit einem `use` in dessen Namen beginnen muss. So wäre z. B. die Funktion `useSwitchState` ein valider Name eines *Hook*s. Innerhalb eines *Hooks* ist es erlaubt, beliebig viele andere *Hook*s zu verwenden.

```
useEffect(() => {
    console.log("a oder b haben sich veraendert.");
}, [a, b]);

useEffect(() => {
    console.log("Die Komponente wurde aufgerufen.");
    return () => console.log("Die Komponente wurde ausgehaengt.");
}, []);
```

Quellcode 3.6: Verwendung des *Hooks* `useEffect`.

Jedoch müssen dabei ebenfalls die o. g. Regeln eingehalten werden. Ebenso können beliebige Werte innerhalb des *Hook*s zurückgegeben werden, welche beim Aufruf des erstellten *Hook*s verwendet werden können. Die Verwendung von Übergabeparametern ist dabei nicht eingeschränkt. [21, 35]

Quellcode 3.7 zeigt dabei einen *Hook* namens `useSwitchState`, welcher den *Hook* `useState` verwendet und diesen modifiziert, um lediglich zwischen den Zuständen `true` und `false` zu wechseln. Dieser *Hook* könnte z. B. in der React Komponente `Switch` Verwendung finden. (siehe Quellcode 3.4)

```
export const useSwitchState = (initialState: boolean) => {
    const [state, setState] = React.useState(initialState);
    return [state, () => setState((oldState) => !oldState)];
}
```

Quellcode 3.7: Definition des eigenen *Hooks* `useSwitchState`.

3.5 Klassen-Komponenten

Klassenkomponenten definieren sich durch die Erweiterung der Klasse `Component` von React sowie der Implementierung der Methode `render`. Die Methode `render` gibt dabei analog zu funktionalen Komponenten (vgl. Abschnitt 3.4) ein React Element zurück. (siehe Quellcode 3.8)

```
import * as React from "react";

export class Greeter extends React.Component {
    render() {
        return <h1>Hi, {this.props.children}!</h1>;
    }
}

const usage = <Greeter>John</Greeter>;
```

Quellcode 3.8: Sehr simple Klassen-Komponente. Diese nimmt innerhalb der *Prop*s ein Objekt namens *children* entgegen und zeigt dieses innerhalb eines `h1`-Tags an.

3.6 Limitierungen von Props

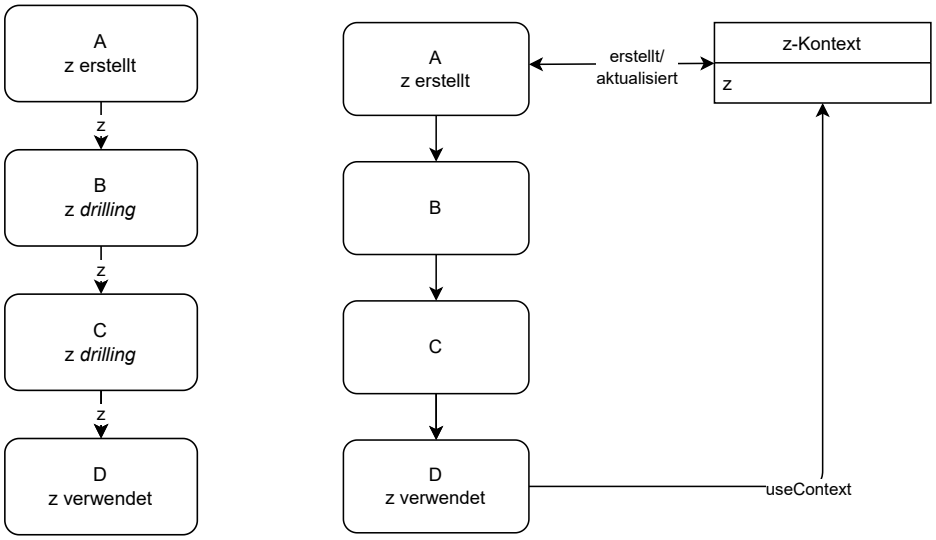
In einer wachsenden React Anwendung kommt es schnell zu einem Problem bei der Verwendung von Props. Man stelle sich eine Teil-Anwendung vor, welche aus den Komponenten A, B, C und D besteht. Dabei wird B in A, C in B und D in C verwendet. A verwaltet dabei den Zustand des Wertes z, welchen D ebenfalls benötigt. Um z mithilfe von *Props* bis zu D zu transportieren, muss es zunächst an B und anschließend an C übermittelt werden, bevor es schließlich in D verwendbar ist. (vgl. Abbildung 3.1a) Demzufolge muss in B und C ein *Prop* definiert werden, welcher von diesen Komponenten nicht genutzt wird. Dennoch ist die Definition des *Props* notwendig, da kein unmittelbarer Weg von A bis D existiert. Ein solches Szenario wird in React als *Prop-Drilling* bezeichnet. (siehe Quellcode 3.9)

```
import { useState } from "react";

type zProp = { z: string };

const A = () => {
  const [z] = useState("this is z");
  return <B z={z} />;
};

const B = (props: zProp) => <C {...props} />;

const C = (props: zProp) => <D {...props} />;

const D = ({ z }: zProp) => <p><i>z</i> is: {z}</p>;
```

Quellcode 3.9: Beispiel des *Prop-Drillings*.

Für dieses Problem existieren mehrere Lösungsansätze. Zunächst ermöglicht React die Definition eines sogenannten Kontextes. Im o. g. Beispiel würde dies bedeuten, dass A einen Kontext definiert, und innerhalb von diesem den Wert z zur Verfügung stellt.

Jede Komponente, welche im Kontext von A verwendet wird[1], kann auf den Wert z zugreifen. A wäre in diesem Fall der Ersteller (*Creator*) des Kontextes wie auch der Anbieter (*Provider*) des Wertes z. D kann nun als *Verwender* (*User*) auf z zugreifen. (vgl. Abbildung 3.1 und Quellcode 3.10) [31]



(a) Übertragen von z bis D über Props.

(b) Übertragung von z bis D mit einem Kontext.

Abbildung 3.1: Schematische Gegenüberstellung des *Prop Drillings* und der Verwendung eines Kontextes.

```
1  export const ZContext = React.createContext<string>("z");
2
3  export function A() {
4    const [z] = React.useState("z");
5    return (
6      <ZContext.Provider value={z}>
7        <B />
8      </ZContext.Provider>
9    );
10 }
11 ...
12 export function D() {
13   const z = React.useContext(ZContext);
14   return (
15     <p><i>z</i> is: {z}</p>
16   );
17 }
```

Quellcode 3.10: Beispiel für die Verwendung eines Kontextes.

Kontexte sind in diesen Anwendungsbereichen häufig ausreichend. In einigen Szenarien genügt jedoch selbst das Verwenden von Kontexten nicht mehr. Wird z. B. der in einem Kontext gespeicherte Zustand in sehr vielen oder fast allen Komponenten

1 Diese sind B, C und D, da diese *children* von A sind.

benötigt oder beschränkt sich der Zugriff auf diesen Zustand nicht mehr auf einzelne Teilbäume der Gesamtanwendung, ist der Einsatz eines globalen Zustandes sinnvoll. Die Verwaltung eines globalen Zustands bietet den Vorteil, dass alle Komponenten auf diesen Zustand zugreifen können. Ein globaler Zustand birgt dennoch Risiken. So muss stets bedacht werden, dass Zustandsänderungen im globalen Zustand eine Reevaluation aller Komponenten zur Folge haben kann, sofern diese den globalen Zustand verwenden[1]. [21]

Für die Verwaltung von globalen Zuständen ist es möglich einen zentralen Kontext zu verwenden, jedoch existieren einige Bibliotheken, die die Verwaltungskomplexität eines solchen globalen Kontextes drastisch reduzieren. Zwei solcher Bibliotheken sind *Redux*[2] (vgl. `https://redux.js.org/`) und *Zustand* (vgl. `https://github.com/pmndrs/zustand`).

3.7 DOM vs. VDOM

React orchestriert zunächst eine interne Darstellung der Komponenten, den sogenannten *virtual DOM* (VDOM), und nimmt erst anhand dieser Darstellung Veränderungen im DOM vor. Dadurch ist es der Bibliothek möglich, andere Darstellungsformen als das DOM zu verwenden, wie z. B. native Android und iOS UI-Elemente. Dies wird im folgenden anhand des React Native Frameworks genauer beschrieben. (vgl. Kapitel 4) [21, 22, 37]

Die Verwaltung eines *VDOMs* ist dabei besonders für die Performance von React relevant. So kann eine Zustandsänderung einer Komponente verschiedene Gründe haben. Dazu zählen veränderte *Props*, die Aktualisierung eines Kontextes (vgl. Abschnitt 3.6), die Aktualisierung durch *Hooks*[3] oder eine Zustandsänderung einer Elternkomponente. Letzteres kann von besonderer Bedeutung für eine allgemeine, degradierte Performance einer React-Applikation sein. Tritt eine solche Zustandsänderung auf, wird der *VDOM* in der Regel neu evaluiert[4]. Kam es zu Änderungen am *VDOM*, werden ebendiese Änderungen in den *DOM* übertragen. Sollten keine Änderungen festgestellt werden, ist eine Aktualisierung des *DOMs* folglich nicht notwendig. [38, 39, 40]

1 Dies ist insbesondere der Fall, wenn keinerlei Optimierungen beim Verwenden eines globalen Zustandes getroffen wurden (insb. Memoisation identischer Werte).

2 Bei der Verwendung von *Redux* sollte ebenfalls die Verwendung von *Redux Toolkit* in Erwägung gezogen werden. Dies reduziert den sonst notwendigen Boilerplate-Code auf ein Minimum (vgl. `https://redux-toolkit.js.org/`)

3 Z. B. `useState`

4 Etwaige Performance-Optimierungen werden bei dieser Erklärung außer Acht gelassen.

4 React Native

React Native erlaubt das Erstellen von hybriden Apps in JavaScript, Flow[1] und TypeScript. Für das Deklarieren von UI-Komponenten wird hierbei das reaktive Framework React (vgl. Kapitel 3) von Meta verwendet.

Anstatt in einer Browserumgebung interpretiert und in DOM-Nodes umgewandelt zu werden, wird der JavaScript- bzw. TypeScript-Code direkt in native Komponenten der entsprechenden Plattform transformiert. Beispielsweise wird die React Komponente `<Text />` in ein Android `TextView`, unter iOS in ein `UITextView` und im Web in ein einfaches `<p>`-Tag umgewandelt. Dies hat zur Folge, dass eine React Komponenten stets zu tatsächlichen nativen Komponenten respektive deren Plattformen korrelieren. [41]

Die Architektur unterscheidet sich dabei grundlegend von anderen bekannten hybriden App-Frameworks wie *Flutter* und *Ionic*. In *Flutter* wurden die UI-Elemente innerhalb der verwendeten Programmiersprache *Dart* reimplementiert. Dies hat zur Folge, dass innerhalb von *Flutter* keine nativen Komponenten von iOS und Android Verwendung finden. [42] *Ionic* hingegen zeigt eine mit HTML, CSS und JavaScript implementiere *Progressive Web App* (*PWA*) in einer *WebView* innerhalb einer App an. [43]

Zum Verfassungszeitpunkt dieser Ausarbeitung untergeht React Native eine bedeutende Transformation. So ist es seit der Version 0.68 möglich, eine neue Render Engine namens *Fabric* inklusive von *Turbo Modules* zu verwenden. Die Implementierung dieser beiden Aspekte stellen dabei bedeutende Meilensteine in der Einführung einer neuen Architektur dar. Die neue Architektur ist jedoch zu diesem Zeitpunkt nicht vollends ausgereift. Demzufolge unterliegt die neue Architektur stetigen Änderungen. Dennoch behandelt diese Ausarbeitung die neue Architektur, da diese voraussichtlich in Zukunft an Bedeutung gewinnen wird. Es ist jedoch zu beachten, dass aufgrund der Unausgereiftheit der neuen Architektur das Risiko besteht, dass die hier beschriebenen Aspekte und Schritte in Zukunft möglicherweise nicht mehr zutreffend sind. [17, 44]

1 Flow bietet die Möglichkeit zur statischen Typüberprüfung von JavaScript-Code. Flow wird hauptsächlich von Meta entwickelt, ist jedoch quelloffen. (vgl. `https://flow.org/`, abgerufen am 26. Oktober 2023.)

Im Folgenden werden einige allgemein wichtige Aspekte von React Native sowie einige spezifische Aspekte der neuen Architektur von React Native präsentiert. Die aufgelisteten Aspekte sind dabei jedoch keinesfalls erschöpfend. Dennoch sind die ausgewählten Themen besonders für eine Brownfield-Integration relevant. (vgl. Kapitel 5)

Über die offizielle Unterstützung von iOS und Android hinaus, bietet React Native die Möglichkeiten, weitere Plattformen zu bedienen. Dazu zählen Plattformen wie macOS, Windows, Apple TV, Android TV sowie Webseiten. Diese Plattformen werden dabei jedoch von der Community oder React Native Partnern entwickelt. [45] Diese Ausarbeitung behandelt ausschließlich die Verwendung von iOS und Android als Plattform.

Host View Tree

React Komponenten werden von React Native in dem *Host View Tree* angeordnet und beinhalten je nach Plattform native iOS oder Android UI-Komponenten. [46] Dieser *Host View Tree* wird dabei von der plattformunabhängigen Layout-Engine *Yoga* generiert. *Yoga* ermöglicht es unter anderem, CSS[1]-Flexbox-Layouts zu verwenden, welche klassischerweise von Browsern interpretiert werden. Da Yoga in *C* implementiert wurde, ist es sowohl unter Android als auch iOS lauffähig. [47]

Hermes

Seit der Version 0.70 von React Native kommt standardmäßig die JavaScript-Engine *Hermes* zum Einsatz. Zuvor kam die JavaScript Engine *JavaScriptCore* des Safari-Browsers von Apple zum Einsatz. Somit eliminiert die Verwendung von *Hermes* die inkonsistenten Versionsstände von *JavaScriptCore*, welche Airbnb in der Verwendung von React Native negativ aufgefallen sind. (vgl. Abschnitt 2.1.3) *Hermes* ist dabei speziell für den Einsatz mit React Native optimiert und verspricht gegenüber der Verwendung von *JavaScriptCore* eine verbesserte Startzeit, verringerte Arbeitsspeichernutzung sowie eine kleinere App-Größe. Bei der Verwendung von *Hermes* ist es darüber hinaus möglich, den Webbrowser Chrome zu verwenden, um die Ausführung des JavaScript-Codes innerhalb der Hermes-Engine zu debuggen. (vgl. `https://reactnative.dev/docs/hermes#debugging-js-on-hermes-using-google-chromes-devtools`, abgerufen am 26. Oktober 2023) [18, 48]

1 CSS = Cascading Style Sheets, legt z. B. Farben, Abstände, Schriftarten ... einer Webseite fest

4.1 Fabric

Mit Version 0.68 wurde eine neue Architektur in React Native implementiert. [49] Als größte Änderung setzt diese auf eine neue Render-Engine namens *Fabric*. Laut der Dokumentation bringt diese neben besserer Wartbarkeit und Codequalität auch Performanceverbesserungen mit sich. [44]

4.1.1 View Flattening

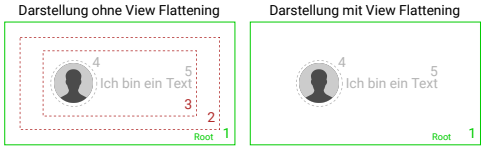
Eine dieser Verbesserungen ist dabei ein plattformübergreifendes *View Flattening*. Es ist nicht ungewöhnlich, dass bei der Entwicklung einer React-Komponente Layout-Komponenten häufig ineinander geschachtelt werden (siehe Quellcode 4.1). Solche Layout-Komponenten zeigen dabei keine konkreten Informationen an, sondern definieren lediglich das Aussehen der Komponente (siehe Abbildung 4.1). Das Resultat vieler ineinander verschachtelten Layout-Komponenten ist ein tiefer *Host View Tree* an Komponenten. Das *View Flattening* kompaktiert hierbei aufeinander folgende Layout-Komponenten, indem es das Styling der einzelnen Layout-Komponenten in eine Layout-Komponente zusammenfasst. Folglich muss der Renderer in Teilen des *Host View Tree* nicht viele Layout-Komponenten darstellen, sondern lediglich eine (siehe Abbildung 4.2). [50]

```
1     function ListViewItem() {
2       return (
3         <View id="1">
4           <View id="2" style={{ display: "flex" }}>
5             <View id="3" style={{ gap: 8 }}>
6               <Image id="4" />
7               <Text id="5">Ich bin ein Text</Text>
8             </View>
9           </View>
10        </View>
11      );
12    }
```

Quellcode 4.1: Beispiel-Komponente mit mehreren Layout-Komponenten [50]

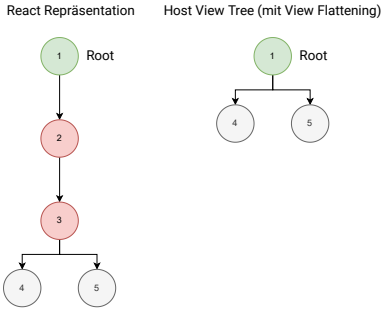
4.1.2 JavaScript Interfaces (JSI)

Besteht die Notwendigkeit, dass eine native App durch die Verwendung einer nativen Komponente oder eines nativen Moduls auf Daten React Natives zugreifen muss – z. B. für das Anzeigen eines Textes – so werden unter Verwendung der alten Architektur



Die Ziffern $1-5$ sind identisch zu den IDs aus Quellcode 4.1.
Die gestrichelte Linien stellen Layout-Komponenten dar, die für den Nutzer als solche nicht unmittelbar ersichtlich sind.

Abbildung 4.1: Schematische Darstellung des View Flattenings anhand von Layout-Komponenten. [50]



Die Ziffern $1-5$ sind identisch zu den IDs aus Quellcode 4.1.
Linker Baum: interne React Repräsentation (vgl. Abschnitt 3.7)
Rechter Baum: Host View Tree (also Darstellung auf Gerät, vgl. Kapitel 4)

Abbildung 4.2: Schematische Darstellung des View Flattenings anhand einer Baumstruktur. [50]

diese Daten zunächst in die JavaScript Object Notation (JSON) serialisiert[1]. (vgl. Abschnitt 4.2) Das serialisierte JSON-Objekt wird anschließend an die native Komponente oder das native Modul übertragen. Damit eine Komponente nun auf die Daten des JSON-Objektes zugreifen kann, muss dieses zunächst wieder deserialisiert werden. Diese beiden Schritte benötigen evident Rechenleistung.

Angesichts dessen greifen *Turbo Native Modules* (vgl. Abschnitt 4.4) und *Fabric Components* direkt auf die Werte über *JavaScript Interfaces* (JSI) zu und sparen somit die beiden Umwandlungsschritte. Dabei existiert eine bidirektionale Verknüpfung eines JavaScript-Objektes aus React Native zu einem `C++`-Objekt. Das `C++`-Objekt befindet sich dabei in der Laufzeitumgebung der nativen App. Auf diesen gespiegelten Objekten können nun beidseitig Methoden aufgerufen werden, um entweder Effekte in React Native oder aber in der nativen App auszulösen.

Dieser Ansatz erlaubt darüber hinaus eine synchrone Ausführung der Methoden, was bei der Verwendung der alten nativen Module nicht möglich war. Weiterhin besteht durch

1 Vereinfacht ausgedrückt, werden Objekte in eine Zeichenkette transformiert, die den Zustand des Objekts widerspiegelt

die Verwendung von JSIs die Möglichkeit, Aktionen auf anderen Threads zu initiieren. Zusätzlich erlaubt das Verwenden von JSIs eine Typsicherheit in sowohl `C++` als auch in TypeScript innerhalb von React Native. *Turbo Native Modules* beheben somit das Problem der fehlenden Typsicherheit, welches Airbnb bei der Integration von React Native angemerkt hatte. (vgl. Abschnitt 2.1.3) [44, 51]

4.1.3 Lazy Loading

Bei der Verwendung von Fabric werden native Module erst dann geladen, wenn sie tatsächlich benötigt werden. Bei dem Verwenden der alten Architektur werden sämtliche native Module bereits zum Start der Anwendung geladen, was potenziell zu einer Verzögerung des Starts führt. Durch das Verwenden des sogenannten *Lazy Loading*s verspricht Fabric schnellere Startzeiten. Somit wird das Problem Airbnbs der langen Startzeiten React Natives, durch *Lazy Loading* mitigiert. (vgl. Abschnitt 2.1.3) [44, 51]

4.2 Die Kommunikationsschnittstelle React Natives: Native Module

Native Module erlauben es, aus der React Native Implementierung Code innerhalb der nativen Apps auszuführen. Das Erstellen von nativen Modulen ist dabei in einem Greenfield-Umfeld weniger verbreitet, da dort selten die Notwendigkeit besteht, direkt auf Methoden der nativen Apps zuzugreifen. In einem Brownfield-Umfeld hingegen sind native Module besonders wichtig. Native Module können dabei entweder für das Erstellen von React Native Bibliotheken verwendet werden, um z. B. Brücken zu nativen Funktionalitäten der iOS oder Android App zu schlagen oder sie können in einem *Brownfield*-Umfeld verwendet werden, um auf Implementierungen oder Daten der bestehenden Apps zuzugreifen. Folglich stellen native Module eine Kommunikationsschnittstelle zwischen React Native und den nativen Apps dar. [52]

Für native Module kommt eine *Bridge* zum Einsatz. Diese *Bridge* erlaubt dabei eine bidirektionale Kommunikation der nativen App mit React Native durch die Serialisierung von JSON. [53, 1. Absatz]

Native Module verwenden dabei nicht die neue Architektur. Dafür sind Turbo Native Module vorgesehen. (vgl. Abschnitt 4.4)

Beispielhaft wird im Folgenden ein natives Modul zum Protokollieren einer Nachricht implementiert. Nach der erfolgreichen Protokollierung eines Titels und einer Beschreibung,

wird ein *Promise*[1] mit der Nachricht *success* erfüllt. Dabei ist eine synchrone Ausführung bei der Verwendung von nativen Modulen nicht möglich. Es ist somit zwingend notwendig, für etwaige Rückgabewerte *Callbacks* oder *Promises* zu verwenden. Sollte der Titel oder die Beschreibung leer sein, wird der *Promise* mit einer Fehlernachricht zurückgewiesen. Dabei dient diese Implementierung nur zur Darstellung, wie native Module funktionieren. Es sind somit deutlich komplexere native Module denkbar, die auf Ressourcen bestehender Apps zugreifen, wie z. B. lokale Datenbanken der nativen Apps.

Die ab hier verwendeten Code-Ausschnitte sind dabei stets auf die wesentlichsten Aspekte kompaktiert, da diese sonst einen Großteil der Ausarbeitung einnehmen würden. Dennoch ist der vollständige Source-Code öffentlich unter `https://github.com/flokol120/react-native-brownfield-examples` (abgerufen am 26. Oktober 2023) verfügbar. Der Source-Code ist dabei in 13 verschiedene *Branches* organisiert. Diese *Branches* wurden dabei so erstellt, dass diese thematisch aufeinander aufbauen. Die Struktur ist dabei in Abbildung A.2 und Tabelle B.1 dargestellt. Dieser Aufbau erlaubt das Vergleichen der verschiedenen *Branches*, wodurch hinzugefügte Zeilen grün und mit einem + und gelöschte Zeilen rot und mit einem - hervorgehoben werden. Somit ist deutlich, welche Änderungen zwischen den Branches getätigt wurden, um das in dieser Ausarbeitung dargestellte Ergebnis zu erhalten. Innerhalb der `README.md` des *Repositories* befindet sich ebenfalls eine Tabelle zur vereinfachten Navigation der *Branches*. Ebendiese Tabelle verweist darüber hinaus auf die in dieser Ausarbeitung relevanten Kapitel des jeweiligen *Branches*. Die vollständige Implementierung des nativen Moduls befindet sich dabei auf dem *Branch native-module-logging*. Verweisen künftige Kapitel auf andere *Branches*, wird dies zu Beginn erwähnt.

Prinzipiell ist es möglich, ein natives Modul auch als NPM-Modul bereitzustellen. Dieses kann innerhalb einer React Native App als Abhängigkeit hinzugefügt werden. Ebenso ist es möglich, native Module direkt in einer bestehenden React Native App zu definieren[2]. Letzteres Vorgehen wird dabei im erwähnten Beispiel verwendet, da somit ein potenzieller Zugriff auf Daten einer bestehenden App im Brownfield-Umfeld einfacher möglich ist. Dabei wird das native Modul jedoch zunächst in einem Greenfield-Szenario implementiert, da so der Initialaufwand zum Aufsetzen einer lauffähigen App minimal

1 Ein *Promise* beschreibt ein Versprechen, dass die aufgerufene Funktion einen Rückgabewert liefert. Die Verwendung von *Promise*s ist besonders in der asynchronen Programmierung hilfreich, da auf die Erfüllung dieses Versprechens gewartet werden kann. Promises können ausufernde Callback-Konstrukte (auch *Callback-Hell* genannt) minimieren.

2 vgl. `https://reactnative.dev/docs/native-modules-setup` (abgerufen am 26. Oktober 2023) für die Nutzung in einem NPM-Modul

ist. Das beschriebene Vorgehen ist dabei identisch zu dem Vorgehen innerhalb eines Brownfield-Umfelds[1].

Bevor ein neues React Native Projekt erstellt werden kann, muss die Entwicklungsumgebung aufgesetzt werden. Dazu kann die offizielle Dokumentation React Natives herangezogen werden: `https://reactnative.dev/docs/environment-setup?guide=native`[2] (abgerufen am 26. Oktober 2023). Dabei muss zuvor das entsprechende Entwicklungsbetriebssystem (GNU/Linux, Windows oder Mac OS) sowie das Betriebssystem des mobilen Endgerätes ausgewählt werden (iOS oder Android). Dabei gilt zu beachten, dass auf einem Gerät mit Mac OS sowohl iOS als auch Android Apps erstellt werden können, unter GNU/Linux sowie Windows ist lediglich Android möglich. Ist die Entwicklungsumgebung korrekt eingerichtet worden, kann mit `npx react-native@0.72.4 init NativeLoggingModule`[3] ein neues React Native Projekt erstellt werden. [52, 54]

4.2.1 iOS

Zunächst muss der `ios` Ordner mit Xcode[4] geöffnet werden. Unter iOS ist es möglich, das native Modul sowohl in Objective-C als auch in Swift zu verfassen. Wird ein neues React Native Projekt erstellt, wird diese im Standard mit Objective-C initialisiert, dies schließt jedoch nicht die Verwendung von Swift in nativen Module aus.

4.2.1.1 Vorgehen bei der Verwendung von Objective-C

Bei der Verwendung von Objective-C ist es zunächst notwendig, ein *Header-File* anzulegen. Diese Datei gibt dabei lediglich die Struktur der Implementierungsdatei vor. Die Datei wird dabei `RCTLoggingModule.h`[5] genannt. Nun wird das Protokoll `RCTBridgeModule` von React Native importiert. Anschließend wird ein Interface namens `RCTLoggingModule` definiert, welches `NSObject` sowie `RCTBridgeModule` erweitert. (siehe Quellcode 4.2)

1 Dazu müssen jedoch zunächst die in Kapitel 5 beschriebenen Schritte durchgeführt werden. Somit ist das Erstellen einer neuen React Native App deutlich schneller möglich

2 Da für verschiedenste Betriebssysteme unterschiedliche Schritte notwendig sind, wird das Aufsetzen der Entwicklungsumgebung nicht in dieser Ausarbeitung behandelt.

3 Der Command stellt dabei sicher, dass die Version `0.72.4` verwendet wird. Diese wurde auch für das Beispiel verwendet.

4 Prinzipiell ist es auch möglich, ohne Xcode zu arbeiten, dennoch empfiehlt sich die Verwendung, da Xcode speziell für die Entwicklung von iOS Apps vorgesehen ist. Xcode kann aus dem Apple App Store heruntergeladen werden (vgl. `https://apps.apple.com/us/app/xcode/id497799835`, abgerufen am 26. Oktober 2023) und ist folglich nur auf Geräten mit macOS verwendbar.

5 Das Präfix `RCT` steht hierbei für *React*. Dies dient lediglich zur Organisation, da Objective-C keine Möglichkeit zu Definition von Namensräumen bietet.

```
1 #import <React/RCTBridgeModule.h>
2 @interface RCTLoggingModule : NSObject <RCTBridgeModule>
3 @end
```

Quellcode 4.2: *Header-File* des Protokollierungsmoduls: `RCTLoggingModule.h`.

Daraufhin wird das zuvor erstellte *Header-File* implementiert. Dafür ist es zunächst notwendig, das *Header-File* `RCTLoggingModule.h` wie auch `RCTLog.h` von React Native in einer neuen Datei namens `RCTLoggingModule.m` zu importieren. Nun kann die Implementierung mit dem *Keyword* `@implementation` definiert werden. Anschließend ist zu beachten, das Modul React Native bekannt zu machen. Dafür wird das Makro `RCT_EXPORT_MODULE` verwendet. Wird dieses ohne Argumente aufgerufen, wird der Name des Moduls automatisch anhand des Klassennamens inferiert[1]. Anschließend können Methoden mit dem Makro `RCT_EXPORT_METHOD` deklariert werden, welche aus React Native aufrufbar sein sollen. Unter iOS wird dabei der *Promise* in einen `RCTPromiseResolveBlock` und `RCTPromiseRejectBlock` aufgeteilt. So wird die Methode `log` mit den Parametern `title` und `body` sowie dem `resolver` und `rejecter` des *Promises* deklariert. Beinhaltet `title` oder `body` dabei lediglich eine leere Zeichenkette, wird der *Promise* zurückgewiesen. Ansonsten wird `title` und `body` auf die Konsole protokolliert und der *Promise* mit der Nachricht *success* erfüllt. (siehe Quellcode 4.3) `resolve` kann dabei mit von React Native unterstützten Datentypen aufgerufen werden. Dabei beschränkt sich diese Auswahl nicht ausschließlich auf primitive Datentypen wie *String*s, *Int*s ... sondern es ist ebenso möglich, *Map*s und *Array*s zu übergeben. Alle unterstützen Datentypen, welche unter iOS zur Verfügung stehen, sind in Tabelle 4.1 aufgelistet. Die übergebenen Daten werden dabei in JSON konvertiert und sind als solches in React Native verwendbar.

Das native Modul ist nun wie in Abschnitt 4.2.3 beschrieben zu verwenden.

4.2.1.2 Vorgehen bei der Verwendung von Swift

Die Definition von nativen Modulen mit der Programmiersprache Swift ist ebenfalls möglich.

Wird das erste native Modul mit Swift erstellt, ist es zunächst notwendig, eine *Bridging Header*-Datei zu erstellen. Diese erlaubt den Zugriff auf Objective-C Dateien innerhalb von Swift Dateien bzw. den Zugriff auf Swift Dateien innerhalb von Objective-C Dateien.

1 Präfixe wie RCT werden dabei entfernt.

Tabelle 4.1: Unterstütze Datentypen der Übergabeparameter bei der Verwendung von nativen Modulen unter iOS. [55]

Objective-C	Swift	JavaScript/TypeScript
NSString	String	string, string?
BOOL	Bool	boolean
double	double	number
NSNumber	NSNumber	number?
NSArray	NSArray	Array, Array?
NSDictionary	NSDictionary	object, object?
RCTResponseSenderBlock	RCTResponseSenderBlock	Function
RCTResponseErrorBlock	RCTResponseErrorBlock	Function
RCTPromiseResolveBlock, RCTPromiseRejectBlock	RCTPromiseResolveBlock, RCTPromiseRejectBlock	Promise

```
1  #import "RCTLoggingModule.h"
2  #import <React/RCTLog.h>
3
4  @implementation RCTLoggingModule
5
6  RCT_EXPORT_MODULE();
7
8  RCT_EXPORT_METHOD(log : (NSString*)title body : (NSString*)body resolver:
   ↪  (RCTPromiseResolveBlock)resolve rejecter: (RCTPromiseRejectBlock)reject)
   ↪  {
9    if([title isEqual: @""] || [body isEqual: @""]) {
10     reject(@"log", @"no title or body specified", nil);
11     return;
12   }
13   RCTLogInfo(@"Log from React Native, title: %@, body: %@", title, body);
14   resolve(@"success");
15 }
16
17 @end
```

Quellcode 4.3: Implementierung des *Header-File*s aus Quellcode 4.2 als `RCTLogging-Module.m`. Definition der Methode, welche in React Native aufgerufen werden kann.

Dafür muss eine neue Datei in Xcode erstellt werden. Der Name kann dabei prinzipiell frei gewählt werden. In diesem Beispiel wird die Datei `LoggingModule-Bridging-Header.h` genannt. Innerhalb dieser Datei muss der Header `RCTBridgeModule.h` importiert werden. (siehe Quellcode 4.4) Anschließend ist es notwendig, den *Bridging Header* dem Swift-Compiler bekannt zu machen. Dazu muss in den *Build-Settings* des iOS Projektes innerhalb von XCode unter *Swift Compiler - General* der Name des *Bridging Header*s eingetragen werden. Existiert im iOS Projekt noch keine Swift Datei, ist das Definieren des *Bridging Headers* zu diesem Zeitpunkt nicht möglich. In diesem Fall ist dieser Schritt zunächst zu überspringen und später auszuführen. Beim Erstellen

der ersten Swift Datei innerhalb des Projektes ermöglicht Xcode die automatische Erstellung des *Bridging Headers*.[1]

```
#import <React/RCTBridgeModule.h>
```

Quellcode 4.4: Definition des *Bridging Header*s `LoggingModule-Bridging-Header.h`.

Anschließend ist es erforderlich, für jedes Modul, welches erstellt werden soll, eine Objective-C *Bridge* zu erstellen. Diese *Bridge* beinhaltet dabei lediglich die Namen und Signaturen der Methoden, welche anschließend in Swift implementiert werden. In diesem Beispiel wird diese Datei `LoggingModuleBridge.m` genannt. Wie auch bei der Verwendung von Objective-C wird das Modul mit dem Makro `RCT_EXTERN_MODULE` initialisiert und jede Methode mit `RCT_EXTERN_MODULE` deklariert. Es ist jedoch wichtig zu beachten, dass hierbei die Verwendung von `RCT_EXTERN_MODULE` keine Implementierung beinhaltet. (siehe Quellcode 4.5)

```
#import "React/RCTBridgeModule.h"

@interface RCT_EXTERN_MODULE(LoggingModule, NSObject)

RCT_EXTERN_METHOD(
                    log: (NSString *)title
                    body: (NSString *)body
                    resolver: (RCTPromiseResolveBlock)resolve
                    rejecter: (RCTPromiseRejectBlock)reject
                  )

@end
```

Quellcode 4.5: Definition des Moduls `LoggingModuleBridge.m` und der Methode `log` in Objective-C.

Die Implementierungen der Methoden erfolgt in einer Swift Datei namens `LoggingModule.swift`. Zunächst wird eine Klasse namens `LoggingModule` erstellt und mit `@objc` annotiert, damit diese mit der Objective-C Datei verknüpft wird. Anschließend ist es notwendig, die Methode `requiresMainQueueSetup` zu implementieren. Dabei muss lediglich `false` zurückgegeben werden. Die Methode steuert dabei, zu welchem Zeitpunkt das Modul initialisiert werden soll. Bei der Rückgabe von `false`, wird das Modul nicht priorisiert initialisiert. Bei der Verwendung von Objective-C wird die Methode implizit mit dem Rückgabewert `false` definiert. Anschließend wird die `log` Methode analog zur Objective-C Implementierung in Swift umgesetzt. (siehe

1 Dennoch ist der Eintrag aus Quellcode 4.4 später hinzuzufügen.

Quellcode 4.6) Die Funktionsweise von `resolve` und `reject` ist dabei identisch zu Objective-C. (vgl. Abschnitt 4.2.1.1) [55]

```swift
import Foundation

@objc(LoggingModule)
class LoggingModule: NSObject {

    @objc static func requiresMainQueueSetup() -> Bool { return false }

    @objc
    func log(_ title: String, body: String, resolver resolve:
    ↪  RCTPromiseResolveBlock, rejecter reject: RCTPromiseRejectBlock) ->
    ↪  Void {
      if(title == "" || body == "") {
        reject("log", "no title or body specified", nil)
        return
      }
      print("Log from React Native, title: \(title), body: \(body)")
      resolve("success")
    }
}
```

Quellcode 4.6: Implementierung des Moduls `LoggingModule.swift` und der Methode `log` mit Swift.

Anschließend kann das native Modul, wie in Abschnitt 4.2.3 beschrieben, verwendet werden.

4.2.2 Android

Bevor das gleiche Modul unter Android definiert werden kann, sind Anpassungen an der Android App erforderlich. Dazu empfiehlt es sich, den `android` Ordner mit Android-Studio[1] zu öffnen.

Aufgrund der Fähigkeit von Android-Studio, Java-Code in Kotlin umzuwandeln, konzentriert sich dieses Beispiel lediglich auf Java. Eine manuelle Konvertierung kann somit jederzeit vorgenommen werden, sofern Kotlin in der Android App verwendet werden soll.

Zunächst ist es dafür notwendig, eine neue Java-Datei und -Klasse anzulegen. Um eine möglichst konsistente Struktur im Vergleich zur iOS-Implementierung zu gewährleisten, sollten Dateien und Klassen die gleichen Namen tragen. Externe Methoden müssen dabei sowohl den gleichen Namen als auch eine gleiche Signatur besitzen, um

1 Prinzipiell können auch andere Editoren verwendet werden, die Kotlin/Java-Support besitzen. Android Studio ist jedoch für die Entwicklung von Android Apps vorgesehen und sollte aus diesem Grund vorgezogen werden.

in React Native verwendet werden zu können. Daher wird eine neue Datei namens `LoggingModule.java` kreiert.

Die Klasse sollte dabei `ReactContextBaseJavaModule` erweitern, mindestens jedoch `NativeModule` implementieren. Aufgrund der Empfehlung der React Native Dokumentation, `ReactContextBaseJavaModule` zu erweitern, wird diese Vorgehensweise angewendet.

Nach der Erweiterung von `ReactContextBaseJavaModule` ist es notwendig, die Methode `public String getName()` zu überschreiben. Der hier zurückgegebene Name muss dabei identisch zu dem Namen des Makro-Aufrufs aus iOS sein[1]. [56] (vgl. Abschnitt 4.2.1)

Nun können beliebig viele öffentliche Methoden mit dem Rückgabewert `void` und dem Dekorator `@ReactMethod` annotiert werden. Analog zur iOS-Implementierung, wird eine Methode `log` definiert, dessen Signatur zwei Zeichenketten beinhaltet und einen abschließenden Parameter des Typs *Promise* beinhaltet. Im Gegensatz zu iOS ist der *Promise* unter Android in einem Objekt zusammengefasst. Dieses Objekt ermöglicht sowohl die Erfüllung als auch das Zurückweisen des *Promises* mittels der Methoden `.resolve()` und `.reject()`. (siehe Quellcode 4.7)

Informationen über die Konvertierung der Datentypen unter Android sind in Tabelle 4.2 zu finden.

Nun ist es notwendig, das erstellte Modul React Native bekannt zu machen. Unter iOS wurde dies implizit durch die Verwendung des Makros gewährleistet. (vgl. Abschnitt 4.2.1.1) Unter Android muss das Modul hingegen zunächst einem *Package* zugewiesen werden. Ein React Native *Package* zeichnet sich dabei dadurch aus, dass es das Interface `ReactPackage` implementiert. Dazu ist es notwendig, die Methoden `createNativeModules` sowie `createViewManagers` zu implementieren. `create-`

Tabelle 4.2: Unterstützte Datentypen der Übergabeparameter bei der Verwendung von nativen Modulen unter Android. [56]

Java	Kotlin	JavaScript/TypeScript
String	String	string, string?
Boolean/boolean	Boolean	boolean
Double/double	double	number
ReadableArray	ReadableArray	Array, Array?
ReadableMap	ReadableMap	object, object?
Callback	Callback	Function
Promise	Promise	Promise

1 Exklusive des Präfixes RCT.

```java
public class LoggingModule extends ReactContextBaseJavaModule {
    public LoggingModule(ReactApplicationContext context) {
        super(context);
    }
    @NonNull @Override
    public String getName() {
        return "LoggingModule";
    }
    @ReactMethod
    public void log(String title, String body, Promise promise) {
        if (title.isEmpty() || body.isEmpty()) {
            promise.reject("log", "no title or body specified");
            return;
        }
        Log.i("LoggingModule",
            String.format("Log from React Native, title: %s, body: %s",
            ↪  title, body)
        );
        promise.resolve("success");
    }
}
```

Quellcode 4.7: Implementierung des `LoggingModules` unter Android.

`NativeModules` gibt dabei eine Liste an Instanzen von nativen Modulen zurück. In diesem Fall muss dafür lediglich eine Instanz des zuvor definierten `LoggingModules` erstellt werden. Dieses benötigt dafür eine Instanz des `ReactApplicationContexts`, welche an `createNativeModules` übergeben wird und folglich für die Instanziierung verwendet werden kann. Die Methode `createViewManagers` liefert in diesem Fall lediglich eine leere Liste als Rückgabewert. Grund dafür ist, dass `createViewManagers` nur für die Definition von nativen Komponenten verwendet werden muss. (vgl. Abschnitt 4.5) (siehe Quellcode 4.8)

```java
public class LoggingPackage implements ReactPackage {
    @NonNull @Override
    public List<NativeModule> createNativeModules(@NonNull
    ↪  ReactApplicationContext reactApplicationContext) {
        return List.of(new LoggingModule(reactApplicationContext));
    }

    @NonNull @Override
    public List<ViewManager> createViewManagers(@NonNull
    ↪  ReactApplicationContext reactApplicationContext) {
        return List.of();
    }
}
```

Quellcode 4.8: Implementierung des `LoggingPackages` unter Android.

Abschließend ist es notwendig, das zuvor definierte `Package` in der Klasse, die `ReactApplication` implementiert[1], in der Methode `getPackages` des `ReactNativeHosts` aufzunehmen.[2] (siehe Quellcode 4.9) [56]

```
1 ...
2 @Override
3 protected List<ReactPackage> getPackages() {
4     List<ReactPackage> packages = new PackageList(this).getPackages();
5     packages.add(new LoggingPackage());
6     return packages;
7 }
8 ...
```

Quellcode 4.9: Aufnehmen des `LoggingPackages` in der `MainApplication` aus Quellcode 4.8.

Anschließend kann das native Modul, wie in Abschnitt 4.2.3 beschrieben, verwendet werden.

4.2.3 Aufruf in React Native

Alle nativen Module werden in dem JavaScript-Objekt `NativeModules` zusammengeführt. Das Objekt wird dabei von React Native exportiert und ist somit in der eigenen React Native App importierbar. Dabei ist zu beachten, dass die Module aus `NativeModules` – selbst bei der Verwendung von TypeScript – nicht typisiert sind[3]. Deswegen bietet es sich an, das erstellte Modul zusätzlich mit einem Interface zu versehen und das Objekt so manuell zu typisieren. Dazu wird zunächst ein Interface namens `LoggingInterface` definiert, welches die Methode `log` mit den Parametern `title` sowie `body` beinhaltet. Der Rückgabewert der Methode ist dabei ein `Promise<string>`, da die nativen Implementierungen einen *Promise* mit entweder *success* oder einer Fehlermeldung aufrufen. Abschließend wird das zuvor aus `NativeModules` extrahierte Modul `LoggingModule` erneut exportiert und mit der Typisierung des Interfaces `LoggingInterface` versehen. (siehe Quellcode 4.10) Dabei ist jedoch zu beachten, dass das Interface analog zur iOS und Android Implementierung definiert werden muss. Überdies ist es wichtig zu beachten, dass das Interface bei Änderungen in der iOS- und Android-Implementierung manuell angepasst werden muss, um den Definitionen von iOS und Android zu entsprechen. [56, 55]

1 In den meisten Fällen heißt diese Klasse `MainApplication`
2 Sollte die IDE in der Klasse Fehler anzeigen, kann es sein, dass React Native bisher die notwendigen Klassen nicht generiert hat. Nach dem Ausführen von *Build → Make Project* werden die notwendigen Dateien erstellt.
3 Bzw. den Typ `any` besitzen.

Dieses Vorgehen ist dabei bei der Verwendung von Turbo Native Modules obsolet, da notwendige Interfaces für iOS und Android ausgehend eines TypeScript-Interfaces generiert werden. (vgl. Abschnitt 4.4)

Die Definition von Interfaces für native Module verbessert somit die DX, da der Compiler so in der Lage ist, Fehler in der Verwendung der nativen Module zu erkennen. Airbnb empfand die fehlende Typisierung der nativen Module als störend (vgl. Abschnitt 2.1). Dieses Vorgehen trägt dazu bei, diesen negativen Aspekt zu mildern.

```
1 import { NativeModules } from "react-native";
2
3 const { LoggingModule } = NativeModules;
4
5 interface LoggingInterface {
6     log: (title: string, body: string) => Promise<string>;
7 }
8
9 export default LoggingModule as LoggingInterface;
```

Quellcode 4.10: Reexportieren des `LoggingModules` inklusive Typisierung durch das `Interface` `LoggingInterface`

In einer React-Komponente kann das exportierte Modul anschließend verwendet werden. In einem `async`-Kontext ist es zudem möglich, auf die Erfüllung des *Promises* mittels `await` zu warten. Der Aufruf des nativen Moduls ist dabei ausgehend von dem zuvor definierten Interfaces typisiert. (siehe Quellcode 4.11)

```
import NotificationModule from './modules/NotificationModule';
...
function X() {
    ...
    return (
        ...
          <Button
            onPress={async () => {
              const response = await LoggingModule.log("Hello!", "Hello from
              ↪ React Native!");
              console.log(response); // <= success
            }}
            title="log message" />
            <Button
              onPress={async () => {
                try{
                  await LoggingModule.log("", "No title");
                }catch(e) {
                  console.warn(e); // <= always triggered
                }
              }}
              title="log erroneous message" />
        ...
    );
}
```

Quellcode 4.11: Beispielhafte Verwendung des Moduls beim Drücken eines *Button*s. Der obere Button wird stets erfüllt, während der untere immer zurückgewiesen wird, da der Titel ein leerer String ist.

4.3 Event Emitter

Native Module erlauben es, React Native Komponenten eine Kommunikation mit der nativen App zu initiieren. Muss jedoch die native App Daten React Native zur Verfügung stellen, ist dies mit nativen Modulen nicht umsetzbar, ohne dass React Native diese explizit anfragt. Ebendarum existieren *Event Emitter*. *Event Emitter* werden aus den nativen Apps aufgerufen, um Daten an React Native zu übersenden. Damit die Daten jedoch empfangen werden können, muss innerhalb von React Native auf jene definierte *Event*s gehorcht werden. Bei der Verwendung von *Event Emittern* ist es im Gegensatz zu nativen Modulen nicht möglich, *Callback*s oder *Promise*s zu verwenden. (vgl. Tabelle 4.1 und 4.2)

Dabei wird im Folgenden beispielhaft ein *Event Emitter* implementiert, welcher eine Nachricht innerhalb der React Native Laufzeitumgebung protokolliert.

Die vollständige Implementierung des *Event Emitters* ist im Git-Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) unter dem *Branch* `event-emitter` verfügbar.

4.3.1 Horchen auf Event Emitter

Zunächst ist es notwendig, dass innerhalb der React Native Anwendung mindestens eine Komponente auf den *Event Emitter* horcht. Dies bedeutet konkret, dass eine Komponente einen *Callback* registrieren muss, welcher aufgerufen wird, sobald der *Event Emitter* von der nativen App verwendet wird. Ein *Event Emitter* identifiziert sich dabei immer über einen Namen innerhalb eines nativen Moduls.[1] Dieses native Modul wird dabei, wie in Abschnitt 4.2 beschrieben, erstellt. Zusätzlich erweitert das Interface dabei, welches in TypeScript definiert wird, den `type NativeModule`, welcher von React Native exportiert wird. Dieser `type` stellt dabei sicher, dass das Modul die Methoden `addListener` sowie `removeListeners` beinhaltet.[2] Diese Methoden werden von React Native intern verwendet, um die Registrierung der Horchenden zu verwalten. Die Registrierung des *Listener*s in einer React Komponente sollte dabei innerhalb eines `useEffects` (vgl. Abschnitt 3.4.1) durchgeführt werden. Wird der *Dependency Array* dabei leer gelassen, wird der Code innerhalb der Funktion nur beim ersten *Render* der Komponente aufgerufen. Die zurückgegebene Funktion hingegen wird aufgerufen, wenn die Komponente ausgehängt wird. Daher eignet sich diese Funktion gut, um den erstellten *Listener* wieder freizugeben, sobald die Komponente nicht mehr aktiv ist. (siehe Quellcode 4.12) Alternativ ist die Registrierung außerhalb einer React Komponente möglich.[3]

```
1 React.useEffect(() => {
2   const eventEmitter = new NativeEventEmitter(EventEmitterModule);
3   const listener = [
4     eventEmitter.addListener('logToReactNative', (message: string) =>
5       console.log(message)
6     ),
7   ];
8   return () => listener.forEach((currListener) => currListener.remove());
9 }, []);
```

Quellcode 4.12: Registrierung eines *Callbacks* für einen *Event Emitter* in React Native.

Anschließend muss das im *Event Emitter* verwendete native Modul in iOS und Android erstellt werden. Dies ist dabei analog zu Abschnitt 4.2.1 und 4.2.2 durchzuführen.

1 Lediglich iOS erzwingt die Verwendung von *Event Emittern* innerhalb eines nativen Moduls. Um beide Plattformen gleichzubehandeln, wird jedoch auch unter Android der *Event Emitter* innerhalb eines nativen Moduls verwendet.
2 Vgl. `EventEmitterModule.ts` im Git-Repository.
3 Dieser Ansatz funktioniert jedoch ausschließlich, wenn keine Daten der Komponente wie z. B. *Props* oder Zustandsvariablen benötigt werden.

4.3.2 Registrierung des *Event Emitters* in iOS

Nach der Erstellung eines nativen Moduls nach Abschnitt 4.2.1, müssen Anpassungen an diesem Modul vorgenommen werden, damit *Event Emitter* innerhalb dieses Moduls verwendet werden können. Dabei wird angenommen, dass das native Modul mit Swift erstellt wurde. Zunächst ist es notwendig, zusätzlich `#import <React/RCTEventEmitter.h>` in der `Bridging-Header`-Datei aufzunehmen. Die `<ModuleName>Bridge.m`-Datei muss angepasst werden, indem das erstellte Interface nicht `NSObject` erweitert, sondern stattdessen `RCTEventEmitter`: `@interface RCT_EXTERN_MODULE(EventEmitterModule, RCTEventEmitter)`. Weiterhin muss `RCTEventEmitter` importiert werden: `#import "React/RCTEventEmitter.h"`. Anschließend ist es notwendig, die Swift Implementierung des nativen Moduls anzupassen. Dazu wird die Klasse zunächst so modifiziert, dass diese ebenfalls `RCTEventEmitter` erweitert. Weiterhin ist es notwendig, die im Konstruktor erstellte Instanz des nativen Moduls in einer statischen Variable zu speichern. Dazu wird die statische Klassenvariable `sharedInstance` erstellt und zunächst mit `nil` initialisiert. Im Konstruktor wird anschließend das erstellte Objekt `sharedInstance` zugewiesen. Zuletzt wird die statische Methode `emitLogToReactNative` implementiert. Diese prüft zunächst, ob `sharedInstance` bereits mit der erstellten Instanz initialisiert wurde und erzeugt einen Fehler, wenn dies nicht der Fall sein sollte. Sollte `sharedInstance` hingegen bereits initialisiert sein, wird über die Methode `sendEvent` von `RCTEventEmitter` das *Event* `logToReactNative` mit dem übergebenem Parameter aufgerufen. Weiterhin ist es notwendig, die Methode `supportedEvents` zu überschreiben. Diese muss dabei einen Array der aufrufbaren *Event*-Namen zurückgeben. In diesem Fall ist dies lediglich das *Event* `logToReactNative`. (siehe Quellcode 4.13)

Um den `Event Emitter` innerhalb der iOS App zu verwenden, muss die erstellte statische Methode mittels `EventEmitterModule.emitLogToReactNative("Hallo")` aufgerufen werden. Eine Voraussetzung für die Verwendung des *Event Emitter* ist die Initialisierung des `EventEmitterModules` durch React Native, da ohne diese Initialisierung die erstellte statische Instanz ebenfalls nicht korrekt initialisiert wird. Wird der *Event Emitter* vor der Initialisierung durch React Native verwendet, kommt es zum Fehler.

4.3.3 Registrierung des *Event Emitters* in Android

Auch unter Android sind Anpassungen des nativen Moduls notwendig. Zunächst wird die Instanz von `ReactApplicationContext`, welche im Konstruktor des nativen Moduls übergeben wird, einer privaten, statischen Variablen zugewiesen. Anschließend wird eine statische Methode namens `emitLogToReactNative` implementiert, welche die zu übergebene `message` als Parameter entgegennimmt. Die Methode ruft dabei die `emit` Methode des zuvor gespeicherten Kontextes mit dem gewünschten Parameter auf. Dieser

```
...
class EventEmitterModule: RCTEventEmitter {
  ...
  private static var sharedInstance: EventEmitterModule? = nil
  override init() {
    super.init()
    EventEmitterModule.sharedInstance = self
  }
  public static func emitLogToReactNative(message: String) throws {
    if(sharedInstance == nil) {
      throw EventEmitterError.uninitialize
    }
    sharedInstance?.sendEvent(withName: "logToReactNative", body:
    ↪  ["message": message])
  }
  @objc open override func supportedEvents() -> [String] {
    return ["logToReactNative"]
  }
}
enum EventEmitterError: String, Error {
  case uninitialize = "Event Emitter not initialized yet!"
}
```

Quellcode 4.13: Verwendung eines *EventEmitters* unter iOS in Swift.

Parameter wird zuvor in eine `WritableMap` übertragen. Der Aufruf der `emit` Methode spezifiziert dabei zugleich den Namen des *Listeners*; `emitLogToReactNative`. (siehe Quellcode 4.14)

Der *Event Emitter* kann nun dank der statischen Methode in der gesamten App mittels `EventEmitterModule.emitLogToReactNative("Hallo");` verwendet werden. Wie auch in Abschnitt 4.3.2 muss beachtet werden, dass der *Event Emitter* erst verwendet werden kann, sobald das native Modul von React Native initialisiert wurde.

```java
public class EventEmitterModule extends ReactContextBaseJavaModule {
    private static ReactApplicationContext context;
    public EventEmitterModule(@Nullable ReactApplicationContext
    ↪ reactContext) {
        super(reactContext);
        context = reactContext;
    }
    ...
    public static void emitLogToReactNative(String message) throws Exception
    ↪ {
        if(context == null) {
            throw new Exception("Event Emitter not initialized yet!");
        }
        WritableMap params = Arguments.createMap();
        params.putString("message", message);
        context
            .getJSModule(DeviceEventManagerModule
                .RCTDeviceEventEmitter
                .class)
            .emit("logToReactNative", params);
    }
    ...
}
```

Quellcode 4.14: Anpassungen an einem nativen Modul unter Android für die Verwendung eines *Event Emitters*.

4.4 Turbo Native Modules

Turbo Native Modules stellen die nächste Iteration von nativen Modulen dar (vgl. Abschnitt 4.2). *Turbo Native Modules* verzichten dabei gänzlich auf die Verwendung der *Bridge*. Stattdessen kommt das Konzept um *JSIs* zum Einsatz (vgl. Abschnitt 4.1.2). Bei der Verwendung von *Turbo Native Modules* wird darüber hinaus *CodeGen* verwendet.[1] *CodeGen* generiert dabei ausgehend eines TypeScript Interfaces, dazu korrelierende Interfaces für iOS in Objective-C und für Android in Java. Dadurch kann eine typsichere Kommunikation zwischen React Native und den nativen Apps sichergestellt werden. Weiterhin ist es möglich, statt nativen iOS Swift/Objective-C und Android Java/Kotlin Code zu verfassen, *Turbo Native Modules* in C++ zu implementieren. Lässt ein konkreter Anwendungsfall dies zu[2], ermöglicht dieser Ansatz, die Verwendung der Implementierung auf beiden Plattformen, ohne die Notwendigkeit, plattformspezifischen Code zu verfassen, da sowohl iOS als auch Android C++-Code interpretieren können. Die Möglichkeit, *Turbo Native Modules* mit C++ zu implementieren, wird jedoch in dieser Ausarbeitung nicht weiter vertieft. [57, 53]

1 Die Verwendung von *Turbo Native Modules* ist auch ohne *CodeGen* möglich, in diesem Fall muss jedoch sehr viel Boilerplate-Code händisch verfasst werden, weswegen davon abgeraten wird.
2 Z. B. wenn keine nativen iOS oder Android Ressourcen in Anspruch genommen werden müssen.

Da *Turbo Native Modules* neuartig sind, ist die Dokumentation zum Zeitpunkt der Verschriftlichung weniger umfangreich, als die Dokumentation zu nativen Modulen (vgl. Abschnitt 4.2). So beschreibt die offizielle Dokumentation lediglich das Verfassen von *Turbo Native Modules* innerhalb eines eigenständigen Projektes für die Verwendung innerhalb eines neuen NPM-Moduls. [53] Für eine tiefere Integration in bereits bestehende Apps, ist dieses Vorgehen jedoch hinderlich. Durch die direkte Integration von *Turbo Native Modules* in native Apps wird ein unmittelbarer Zugriff auf die nativen Ressourcen der Apps ermöglicht.

Deswegen wird im Folgenden das Vorgehen beschrieben, *Turbo Native Modules* direkt in bereits bestehende Apps zu integrieren, ohne die Notwendigkeit, ein neues Projekt für die *Turbo Native Modules* zu erstellen. (siehe Abschnitt 4.4.2, 4.4.3 und 4.4.4) Dieses Vorgehen ist dabei jedoch nicht offiziell dokumentiert. Der Entwickler Corti React Natives hat angegeben, dass das hier erarbeitete Verfahren künftig in der offiziellen Dokumentation berücksichtigt werden soll. [58]

Dabei wird beispielhaft das gleiche Protokollierungsmodul wie aus Abschnitt 4.2 unter Zuhilfenahme von *Turbo Native Modules* implementiert.

Die vollständige Implementierung des *Turbo Native Modules* ist im Git-Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) unter dem *Branch* `turbo-native-module` verfügbar.

Damit *Turbo Native Modules* verwendet werden können, ist es zunächst notwendig, die neue Architektur zu aktivieren. [53] Die erforderlichen Maßnahmen werden dabei in Abschnitt 4.4.1 dargelegt.

4.4.1 Aktivieren der neuen Architektur

Bevor die Aktivierung der neuen Architektur in Erwägung gezogen wird, sollte beachtet werden, dass verwendete Drittbibliotheken die neue Architektur unterstützen. Mit der Version `0.72` wurde ein Kompatibilitätsmodus eingeführt, der es ermöglicht, nicht unterstützte Komponenten dennoch in der neuen Architektur zu verwenden. Damit nicht unterstützte native Komponenten mit *Fabric* betrieben werden können, ist es notwendig, diese für die Nutzung des Kompatibilitätsmodus zu konfigurieren. Die durchzuführenden Schritte sind dabei in Abschnitt 4.6 aufgeführt.

Nachdem die Kompatibilität aller verwendeten Komponenten mit *Fabric* und der neuen Architektur sichergestellt wurde, ist es möglich diese zu aktivieren. Die Prozedur zur Aktivierung der neuen Architektur variiert zwischen iOS und Android, weshalb iOS und Android individuell behandelt werden. [53]

4.4.1.1 iOS

Die Aktivierung unter iOS wird über die Installation der richtigen *Pods* gesteuert. Dabei müssen alle *Pods* mit dem Kommando `USE_FABRIC=1 RCT_NEW_ARCH_ENABLED=1 pod install` neu installiert werden. Anschließend ist es notwendig, die App erneut zu bauen. Nun ist die neue Architektur aktiviert und betriebsbereit. [49]

4.4.1.2 Android

Unter Android ist es notwendig, die Datei `gradle.properties` anzupassen. Dabei wird die Konfigurationsvariable `newArchEnabled=true` gesetzt. Sollte die Variable nicht bereits in der `gradle.properties` vorhanden sein, kann diese hinzugefügt werden. Zuletzt muss die App erneut gebaut werden. Anschließend ist die neue Architektur in der Android App aktiviert. [49]

4.4.2 TypeScript Interface

Bei der Definition eines *Turbo Native Modules* in React Native, ist es zunächst notwendig, ein TypeScript Interface anzulegen. Dieses Interface wird von *CodeGen* als Grundlage verwendet, um Interfaces für iOS und Android zu generieren, welche in den nativen Apps implementiert werden müssen.

Dazu sind zunächst Anpassungen der `package.json` erforderlich, damit *CodeGen* in der Lage ist, das erstellte Interface aufzufinden. Dafür wird ein neues Feld aufgenommen, welches ein Objekt für die Konfiguration des *Turbo Native Modules* beinhaltet. Das Objekt besteht dabei aus dem Namen des *Turbo Native Modules*, der Art des Moduls, dem Pfad zu den erstellten Interfaces sowie einem Android spezifischen *Package Name*. Der Name kann dabei prinzipiell frei gewählt werden, die offizielle Dokumentation empfiehlt jedoch die Verwendung des Präfixes `RTN`, weswegen dieser ebenfalls in diesem Beispiel verwendet wird. Die Art des Moduls ist in diesem Fall `modules`. Der Pfad `jsSrcsDir` wird mit dem Ordner `turbo-files` festgelegt. Der in der Android App spezifizierte *Package Name* sollte dabei ebenfalls für `javaPackageName` verwendet werden.[1] (siehe Quellcode 4.15)

Anschließend ist es notwendig, einen Ordner namens `turbo-files` zu erstellen. Innerhalb dieses Ordners können beliebig viele Dateien abgelegt werden, welche TypeScript Interfaces definieren.

1 Der *Package Name* ist dabei in der Regel identisch mit dem `namespace`, welcher in der Datei `android/app/build.gradle` definiert ist.

```
{
  ...
  "codegenConfig": {
    "name": "RTNLoggingSpec",
    "type": "modules",
    "jsSrcsDir": "turbo-files",
    "android": {
      "javaPackageName": "com.rtnlogging"
    }
  }
}
```

Quellcode 4.15: Erweiterung der `package.json` für die Definition von Turbo Native Modules.

CodeGen macht dabei die Vorgabe, dass alle Dateien, welche *Turbo Native Modules* beinhalten, mit dem Präfix `Native<ModulName>.ts` beginnen müssen. Daher wird eine Datei namens `NativeLogging.ts` erstellt. Das in der Datei definierte Interface muss dabei zwingend `Spec` genannt werden und `TurboModule` erweitern. Innerhalb des Interfaces besteht die Möglichkeit, Methodensignaturen zu definieren, die in iOS und Android umgesetzt werden sollen. Zuletzt ist es notwendig, das von React Native initialisierte Modul zu exportieren. Dabei ist es zwingend erforderlich, `export default` zu verwenden, damit dieses von React Native korrekt initialisiert werden kann. Beim Aufruf der Methode `TurboModuleRegistry.get()` wird dabei ebenfalls der Name der von *CodeGen* generierten Dateien festgelegt, indem dieser als Argument der Funktion `TurboModuleRegistry.get()` übergeben wird. (siehe Quellcode 4.16)

```
import type { TurboModule } from
↪  "react-native/Libraries/TurboModule/RCTExport";
import { TurboModuleRegistry } from "react-native";

export interface Spec extends TurboModule {
  log(title: string, body: string): Promise<string>;
}

export default TurboModuleRegistry.get<Spec>("RTNLogging");
```

Quellcode 4.16: Definition des Turbo Native Modules `NativeLogging.ts`.

Wird statt `TurboModuleRegistry.get()` `TurboModuleRegistry.getEnforcing()` verwendet, kommt es beim Starten von React Native zu einem Fehler, wenn das Modul in der nativen App nicht implementiert wurde oder es bisher nicht von React Native initialisiert wurde. Dafür ist jedoch das verwendete Modul niemals `undefined`, was bei der Verwendung von `TurboModuleRegistry.get()` möglich ist.

Für die Verwendung des *Turbo Native Modules*, ist lediglich der Wert von *export default* ... aus `NativeLogging.ts` zu importieren. Anschließend kann das Modul typsicher verwendet werden. (siehe Quellcode 4.17) [53]

```
1  import TurboLoggingModule from './turbo-files/NativeLogging';
2  ...
3  function X() {
4      ...
5      return (
6          ...
7          <Button
8              onPress={async () => {
9              const response = await TurboLoggingModule?.log(
10                 'Hello!',
11                 'Hello from React Native using Turbo Modules!',
12             );
13             console.log(response);
14             }}
15             title="log message (turbo)"
16         />
17         <Button
18             onPress={async () => {
19             try {
20                 const response = await TurboLoggingModule?.log('', 'No
                   ↪  title');
21                 console.log(response);
22             } catch (e) {
23                 console.warn(e);
24             }
25             }}
26             title="log erroneous message (turbo)"
27         />
28         ...
29     );
30 }
```

Quellcode 4.17: Verwendung des Turbo Native Modules in einer React Komponente.

4.4.3 iOS

Damit die notwendigen Objective-C und *Header*-Dateien von *CodeGen* erstellt werden können, ist es notwendig, die *Pods* erneut zu installieren. Dazu muss `USE_FABRIC=1 RCT_NEW_ARCH_ENABLED=1 pod install` verwendet werden.[1] Alternativ ist es möglich, die *CodeGen*-Artefakte manuell erstellen zu lassen. Dies kann besonders hilfreich sein, wenn häufiger Änderungen am TypeScript Interface vorgenommen werden. Da-

1 Sollte man hier vergessen, Fabric bzw. die neue Architektur zu aktivieren, werden erneut *Pods* für die alte Architektur installiert. Deswegen sollte besonders darauf geachtet werden, das richtige Kommando auszuführen.

zu kann das Kommando `node node_modules/react-native/scripts/generate-codegen-artifacts.js --path . --outputPath ./ios` verwendet werden.

Anschließend besteht die Notwendigkeit, die generierten *CodeGen*-Artefakte XCode bekannt zu machen, damit diese verwendet werden können. Dafür ist es erforderlich, das iOS-Projekt-Fenster zu öffnen, in die *Build-Settings* zu navigieren und anschließend nach *Header Search Paths* zu filtern.[1] Über einen Doppelklick auf *Header Search Paths* ist anschließend der neue Eintrag `"$(SRCROOT)/build/generated/ios"` in der Liste hinzuzufügen. Dabei ist besonders wichtig, dass innerhalb des Dropdown-Menüs des erstellten Eintrages *recursive* ausgewählt ist. (vgl. Abbildung A.1)

Für die Implementierung des definierten TypeScript-Interfaces ist es zunächst notwendig, eine *Header*-Datei zu erstellen, welche die Struktur der Implementierung vorgibt. Diese Datei wird `RTNLogging.h` genannt. Nun wird zu Beginn die *Header*-Datei inkludiert, welche von *CodeGen* erstellt wurde. Der Name der Datei wird anhand der Konfiguration des Namens aus der `package.json` festgelegt. In diesem Falle hat *CodeGen* die generierten Artefakte folglich in dem Ordner `RTNLoggingSpec` abgelegt. Anschließend kann ein Interface definiert werden, welches *RTNLogging* genannt wird und `NativeLoggingSpec` erweitert. `NativeLoggingSpec` ist dabei ebenfalls von *CodeGen* erstellt worden. (siehe Quellcode 4.18)

```
1 #import <RTNLoggingSpec/RTNLoggingSpec.h>
2
3 NS_ASSUME_NONNULL_BEGIN
4
5 @interface RTNLogging : NSObject <NativeLoggingSpec>
6
7 @end
8
9 NS_ASSUME_NONNULL_END
```

Quellcode 4.18: Inhalt der Datei `RTNLogging.h` für die Implementierung des Interfaces `NativeLoggingSpec`.

Nun ist es vonnöten, eine neue C++-Datei zu erstellen, welche `RTNLogging` genannt wird. Da XCode lediglich das Erstellen von C++-Dateien erlaubt, jedoch nicht von Objective-C++, muss der Typ der erstellten Datei nachträglich angepasst werden. Dazu muss die Datei selektiert und im rechten Menü der *File Inspector*-Tab ausgewählt werden. Dabei ist die Endung der Datei auf *.mm* und der Typ auf *Objective-C++ Source* zu ändern.

1 Wichtig ist, dass nicht das Test *Target* ausgewählt ist und *All* sowie *Combined* ausgewählt ist. Ansonsten wird die Einstellung nicht gefunden.

Daraufhin ist es erforderlich, das Interface, welches zuvor definiert wurde, innerhalb der Objective-C++ Klasse zu implementieren. Wie auch bei der Verwendung von nativen Modulen (vgl. Abschnitt 4.2.1) ist es erforderlich, das Makro `RCT_EXPORT_MODULE` aufzurufen, um das Modul bei React Native zu registrieren. Anschließend kann die Methode `log` implementiert werden. Die Signatur der Methode wurde dabei automatisch von *CodeGen* erstellt. Zuletzt ist es notwendig, die Methode *getTurboModule* zu implementieren. Beim Rückgabewert ist dabei lediglich der Name des *JSIs* anzupassen[1]. (siehe Quellcode 4.19)

```
#import "RTNLoggingSpec.h"
#import "RTNLogging.h"
#import <React/RCTLog.h>

@implementation RTNLogging

RCT_EXPORT_MODULE()

- (void)log:(NSString *)title
       body:(NSString *)body
    resolve:(RCTPromiseResolveBlock)resolve
     reject:(RCTPromiseRejectBlock)reject {
  if([title isEqual: @""] || [body isEqual: @""]) {
    reject(@"log", @"no title or body specified", nil);
    return;
  }
  RCTLogInfo(@"Log from React Native, title: %@, body: %@", title, body);
  resolve(@"success");
}

- (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:
    (const facebook::react::ObjCTurboModule::InitParams &)params
{
    return std::make_shared<facebook::react::NativeLoggingSpecJSI>(params);
}

@end
```

Quellcode 4.19: Inhalt der Datei `RTNLogging.mm` für die Implementierung des Interfaces `NativeLoggingSpec`

Nun kann die App gebaut und das *Turbo Native Module* innerhalb von React Native verwendet werden.

4.4.4 Android

Damit unter Android die von *CodeGen* erstellten C++-Dateien korrekt in den *Build* integriert werden, sind einige manuelle Anpassungen notwendig. Zunächst ist das Er-

1 in diesem Fall `NativeLoggingSpecJSI`

stellen von zwei Dateien erforderlich. Die Datei `CMakeLists.txt` beinhaltet dabei notwendige Abhängigkeiten sowie weitere Metadaten. (siehe Quellcode 4.20) Die Datei `OnLoad.cpp` registriert dabei in der Funktion `javaModuleProvider()` das von *CodeGen* generierte Basis-Modul. Werden weitere Module verwendet, welche ebenfalls nicht automatisch registriert werden können, ist es notwendig, ein solches Modul in dieser Datei zusätzlich aufzunehmen. (siehe Quellcode 4.21) Beide Dateien werden dabei im Ordner `android/app/src/main/jni` erstellt.[1] [59]

```
# Copyright (c) Meta Platforms, Inc. and affiliates.
cmake_minimum_required(VERSION 3.13)
project(appmodules)
include(${REACT_ANDROID_DIR}/cmake-utils/ReactNative-application.cmake)
```

Quellcode 4.20: Inhalt der Datei `CMakeLists.txt`. [59, 60]

```
...
namespace facebook {
namespace react {

...

std::shared_ptr<TurboModule> javaModuleProvider(
    const std::string &name,
    const JavaTurboModule::InitParams &params) {
  auto module = RTNLoggingSpec_ModuleProvider(name, params);
  if(module != nullptr) {
    return module;
  }
  return rncli_ModuleProvider(name, params);
}

} // namespace react
} // namespace facebook

...
```

Quellcode 4.21: Inhalt der Datei `OnLoad.cpp`. [59, 61]

Die Integration der zuvor erstellten Dateien in den *Gradle-Build* setzt eine Änderung der Datei `android/app/build.gradle` voraus, um die Datei `CMakeLists.txt` in den Build zu inkludieren. (siehe Quellcode 4.22) [59]

Im nächsten Schritt wird die Klasse erstellt, die das von *CodeGen* generierte Interface implementiert. Diese Klasse wird dabei `TurboLoggingModule` genannt und erweitert `NativeLoggingSpec`. Damit die abstrakte Klasse `NativeLoggingSpec` vorhanden ist, muss diese zunächst von *CodeGen* erstellt werden.

1 Sollten die Ordner nicht existieren, sind diese anzulegen.

```
...
android {
    ...
   externalNativeBuild {
       cmake {
           path "src/main/jni/CMakeLists.txt"
       }
   }
}
...
```

Quellcode 4.22: Anpassungen an der Datei `build.gradle`

Unter Android ist es möglich, die Klasse manuell mit dem *Gradle-Task* `./gradlew generateCodegenArtifactsFromSchema` zu erstellen. Beim Bauen der App wird dieser Task jedoch automatisch ausgeführt. Die Klasse muss dabei die abstrakte Methode `log()` implementieren, welche innerhalb von Abschnitt 4.4.2 in TypeScript definiert wurde. Ebenfalls ist es notwendig, einen Konstruktor zu definieren, da `NativeLoggingSpec` abstrakt ist. Der Konstruktor ruft dabei lediglich den `super`-Konstruktor auf. (siehe Quellcode 4.23)

```
...
public class TurboLoggingModule extends NativeLoggingSpec {
    public TurboLoggingModule(ReactApplicationContext reactContext) {
        super(reactContext);
    }

    @Override
    public void log(String title, String body, Promise promise) {
        if (title.isEmpty() || body.isEmpty()) {
            promise.reject("log", "no title or body specified");
        }
        Log.i("LoggingModule", String.format("Log from React Native, title:
        ↪  %s, body: %s", title, body));
        promise.resolve("success");
    }
}
```

Quellcode 4.23: Inhalt der Datei `TurboLoggingModule.java`.

Nun ist es erforderlich – wie auch bei nativen Modulen (vgl. Abschnitt 4.2.2) – ein *ReactPackage* anzulegen. Bei der Verwendung von *Turbo Native Modules* ist es hingegen notwendig, die Klasse `TurboReactPackage` statt `ReactPackage` zu erweitern. Dabei müssen die Funktionen `getModule()` sowie `getReactModuleInfoProvider()` implementiert werden. `getModule()` instanziiert dabei anhand des Modulnamens die korrekte Klasse des jeweiligen Moduls. `getReactModuleInfoProvider()` definiert hingegen Informationen über das erstellte Modul. Durch die Implementierung von `getReactModuleInfoProvider()` werden Informationen über das Modul definiert,

wie der Name, die Möglichkeit andere Module zu überschreiben, die Initialisierung zum Start React Natives, das Vorhandensein von Konstanten, die Verwendung von reinem C++-Code sowie die Identifikation als *Turbo Native Module*. (siehe Quellcode 4.24)

Zuletzt ist es erforderlich, das erstellte *ReactPackage* zu registrieren. Diese Registrierung erfolgt analog zu nativen Modulen. Dabei ist eine Instanz des *Turbo Native Modules* zu erstellen und zu verwenden. (siehe Quellcode 4.9)

```
1  package com.nativeloggingmodule;
2
3  import android.util.Log;
4
5  ...
6  public class TurboLoggingPackage extends TurboReactPackage {
7
8      @Nullable
9      @Override
10     public NativeModule getModule(String name, ReactApplicationContext
       ↪  reactContext) {
11         if (name.equals(TurboLoggingModule.NAME)) {
12             return new TurboLoggingModule(reactContext);
13         } else {
14             return null;
15         }
16     }
17
18     @Override
19     public ReactModuleInfoProvider getReactModuleInfoProvider() {
20         return () -> {
21             final Map<String, ReactModuleInfo> moduleInfos = new
               ↪  HashMap<>();
22             moduleInfos.put(
23                     TurboLoggingModule.NAME,
24                     new ReactModuleInfo(
25                             TurboLoggingModule.NAME,
26                             TurboLoggingModule.NAME,
27                             true, // canOverrideExistingModule
28                             true, // needsEagerInit
29                             false, // hasConstants
30                             false, // isCxxModule
31                             true // isTurboModule
32                     ));
33             return moduleInfos;
34         };
35     }
36 }
```

Quellcode 4.24: Inhalt der Datei `TurboLoggingPackage.java`.

Nach Abschluss dieser Schritte kann die Android App erneut gebaut werden. Anschließend kann das erstellte *Turbo Native Module* verwendet werden.

4.5 Native Komponenten

Im Brownfield-Umfeld kann trotz der Bereitstellung von UI-Komponenten React Natives wie Textfeldern, Buttons und anderen durch React Native die Notwendigkeit bestehen, eine native Komponente innerhalb von React Native zu integrieren. Die Umwandlung von nativen iOS- und Android-UI-Elementen in eine React-Komponente wird dabei durch die Definition von nativen Komponenten ermöglicht. Die erstellte native Komponente kann im Anschluss wie eine gewöhnliche React Komponenten innerhalb von React Native verwendet werden. [62]

Beispielhaft wird dazu eine horizontale Fortschrittsanzeige[1] mit einer nativen Komponente für React Native verwendbar gemacht. Sowohl iOS als auch Android bieten dafür ein UI-Element an. Unter iOS kommt dabei `UIProgressView` und unter Android `ProgressBar` zum Einsatz. Die zu erstellende React Native Komponente `ProgressBar` soll dabei über den *Prop* `progress` steuerbar sein. `progress` nimmt dabei eine Gleitkommazahl zwischen 0,0 und 1,0 entgegen, wobei 0,0 keinem Fortschritt entspricht und 1,0 den maximalen Fortschritt abbildet. Zusätzlich soll es über das *Prop* `onPress` möglich sein, auf ein *Touch-Event* der Fortschrittsanzeige zu reagieren.

Für die Implementierung der Fortschrittsanzeige als native Komponente, wird zunächst der iOS spezifische Teil in Abschnitt 4.5.1, anschließend der Android spezifische Teil in Abschnitt 4.5.2 und zuletzt in Abschnitt 4.5.3 die Verwendung innerhalb von React Native demonstriert.

Die vollständige Implementierung der nativen Komponente ist im Git-Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) unter dem *Branch* `native-component` verfügbar.

4.5.1 iOS

Wie auch bei der Verwendung von nativen Modulen, ist es möglich native Komponenten entweder in Objective-C oder in Swift zu verfassen. (vgl. Abschnitt 4.2.1). Bei der Definition von nativen Komponenten ist die Unterscheidung zwischen Objective-C und Swift jedoch nicht ausschlaggebend. Daher wird die native Komponente zunächst in Swift verfasst. Zusätzlich werden Änderungen beschrieben, die notwendig sind, wenn die native Komponente in Objective-C verfasst werden soll.

1 React Native stellt zwar bereits eine Komponente für eine horizontale Fortschrittsanzeige unter iOS und Android bereit, diese ist jedoch als *Deprecated* markiert und kann somit zeitnah entfernt werden. (vgl. [63] und [64])

Zunächst ist es erforderlich `#import <React/RCTViewManager.h>` in den *Bridging Header* aufzunehmen, damit die Definition der nativen Komponente in Swift erfolgen kann. Wird Objective-C verwendet, ist dies nicht notwendig.

Anschließend ist es notwendig, die Klasse `UIProgressView` zu erweitern, welche später in React Native verwendet werden kann. Dazu wird eine neue Swift-Datei namens `ReactProgressBar` angelegt. Die Erweiterung der Klasse `UIProgressView` kann ebenso in Objective-C erfolgen, sofern keine Umsetzung mit Swift gewünscht ist. Die gleichnamige Klasse `ReactProgressBar` erweitert dabei die Klasse `UIProgressView`. Die Konstruktoren rufen dabei in diesem Beispiel lediglich die Konstruktoren der `UIProgressView` auf. So ist es möglich, innerhalb der `init`-Methoden die initiierte *View* anzupassen, wenn dies notwendig sein sollte. Daraufhin wird das `onPress`-*Event* definiert, welches vom Typ `RCTBubblingEventBlock` ist. Damit das `onPress`-*Event* aufgerufen werden kann, sobald eine Nutzerinteraktion mit `UIProgressView` ausgelöst wird, ist es notwendig, die Methode `touchesEnded` zu überschreiben. Innerhalb von `touchesEnded` wird zunächst überprüft, ob die zuvor definierte Variable `onPress` von React Native bereits initialisiert wurde. Sollte dies nicht der Fall sein, wird die Implementierung von `touchesEnded` des `UIProgressViews` verwendet. Andernfalls wird die Funktion `onPress` aufgerufen, wobei als Argument `nil` verwendet wird, da in diesem Fall keine Wertübergabe erforderlich ist. Dennoch wäre es möglich, die in Tabelle 4.1 aufgeführten Typen der Funktion zu übergeben, sofern dies erforderlich ist.[1]. Zuletzt wird beispielhaft demonstriert, wie der Wert des *Props* `progress` in das `UIProgressView` übertragen werden kann. Da jedoch lediglich der Wert von `self.progress` `super.progress` zugewiesen wird, ist das Überschreiben von `progress` in diesem Fall redundant. Sollte jedoch der zu überschreibende Wert nicht gleich mit dem Namen des verwendeten *Props* sein, ist es bei diesem Vorgehen möglich, den übergebenen Wert einer anderen Variablen zuzuweisen[2]. (siehe Quellcode 4.25) [62]

Anschließend ist es notwendig, eine Objective-C Datei namens `RTNProgressBarM⌋anager.m` anzulegen. Innerhalb dieser Datei wird lediglich ein Interface anhand des Makros `RCT_EXTERN_MODULE` definiert. `RCT_EXTERN_MODULE` nimmt dabei als erstes Argument den Namen der nativen Komponente entgegen[3] und als zweites Argument die zuvor importierte Klasse `RCTViewManager`. Daraufhin werden die beiden *Props* `progress` und `onPress` unter Zuhilfenahme des Makros `RCT_EXPORT_VIEW_PRO⌋PERTY` deklariert. Dabei ist es notwendig, den Typ des *Props* zu definieren.

1 Außer die `Callbacks` und `Promises`
2 Würde `progress` in `UIProgressView` z. B. `currentProgress` heißen, wäre die Zuweisung von `super.currentProgress = self.progress` notwendig.
3 Bei der Verwendung des Makros müssen keine Anführungszeichen zur Definition eines String-Literals verwendet werden.

```swift
class ReactProgressBar: UIProgressView {
  override init(frame: CGRect) {
    super.init(frame: frame)
  }

  required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
  }

  @objc var onPress: RCTBubblingEventBlock?

  override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?)
  ↪ {
      guard let onPress = self.onPress else {
        super.touchesEnded(touches, with: event)
        return
      }
      onPress(nil)
      super.touchesEnded(touches, with: event)
  }

  @objc override var progress: Float {
    didSet {
      super.progress = self.progress
    }
  }
}
```

Quellcode 4.25: Definition der Klasse `ReactProgressBar` in Swift, welche `UIProgressView` erweitert.

Bei `progress` ist dies `float`, bei `onPress` hingegen `RCTBubblingEventBlock`[1]. (siehe Quellcode 4.26) [62]

```objc
#import <React/RCTViewManager.h>
@interface RCT_EXTERN_MODULE(RTNProgressBarManager, RCTViewManager)
    RCT_EXPORT_VIEW_PROPERTY(progress, float)
    RCT_EXPORT_VIEW_PROPERTY(onPress, RCTBubblingEventBlock)
@end
```

Quellcode 4.26: Inhalt der Datei `RTNProgressBarManager.m` für die Definition des `RTNProgressBarManager`-Interfaces.

Schließlich muss bei der Verwendung von Swift eine neue Swift-Klasse erstellt werden, welche die vorgegebenen Methoden von `RCTViewManager` implementiert. `RCTViewManager` ist dabei ein Interface, welches notwendigerweise implementiert werden muss, um ein natives UI-Element für React Native zu definieren. Dazu wird eine neue Swift Datei namens `RTNProgressBarManager.swift` erstellt. Mit der Verwendung der

1 Die Typen korrelieren dabei direkt zu den Typen, welche in Quellcode 4.25 festgelegt wurden.

Annotation `@objc(RTNProgressBarManager)` wird angegeben, dass diese Swift-Klasse Teil der Definition der Datei `RTNProgressBarManager.m` ist. Nach der Definition der Klasse inklusive der Erweiterung von `RCTViewManager`, ist es erforderlich, die Methoden `requiresMainQueueSetup` sowie `view` zu implementieren.[1] Für die Methode `requiresMainQueueSetup` wird dabei lediglich `true` zurückgegeben.[2]. Die Methode `view` stellt wiederum die `UIView` bereit, welche für React Native verfügbar gemacht werden soll. Dafür wird eine neue Instanz der Klasse `ReactProgressBar` (siehe Quellcode 4.25) erstellt und zurückgegeben. Bei der Verwendung von `view` ist zu berücksichtigen, dass stets eine neue Instanz von `ReactProgressBar` erstellt werden muss. [55, 65] (siehe Quellcode 4.27)

4.5.2 Android

Für die Definition der nativen Komponente unter Android, wird zunächst eine neue Klasse namens `ReactProgressBar` angelegt, welche das Android UI-Element `ProgressBar` erweitert. Anschließend werden alle von `ProgressBar` definierten Konstruktoren überschrieben. Dies ist zwar in diesem Beispiel nicht zwingend erforderlich, da die Konstruktoren von `ProgressBar` ohne Modifikation aufgerufen werden. Dennoch ist es in diesen überschriebenen Konstruktoren möglich, Initialwerte der `ProgressBar` zu definieren, sofern dies erforderlich ist. Die Implementierung von `ProgressBar` kann dabei größtenteils wiederverwendet werden, lediglich die Methode `onTouchEvent` muss überschrieben werden. Diese löst im Falle eines *Touch Events* das React Native Event `onPress` aus. welches in Quellcode 4.29 Verwendung findet. (siehe Quellcode 4.28) [66]

Daraufhin ist es notwendig, eine weitere Klasse zu erstellen. Diese wird dabei `ReactProgressBarManager` genannt und erweitert `SimpleViewManager<>`.

```
1 @objc(RTNProgressBarManager)
2 class RTNProgressBarManager: RCTViewManager {
3   override static func requiresMainQueueSetup() -> Bool {
4     return true
5   }
6   override func view() -> UIView! {
7     return ReactProgressBar()
8   }
9 }
```

Quellcode 4.27: Inhalt der Datei `RTNProgressBarManager.swift`.

1 Soll Objective-C verwendet werden, können diese Funktionen innerhalb der Datei `RTNProgressBarManager.m` implementiert werden.

2 `requiresMainQueueSetup` ist dabei die gleiche Methode wie aus den nativen Modulen. (vgl. Abschnitt 4.2.1)

```
1  public class ReactProgressBar extends ProgressBar {
2      public ReactProgressBar(Context context) {
3          super(context);
4      }
5
6      public ReactProgressBar(Context context, AttributeSet attrs) {
7          super(context, attrs);
8      }
9
10     public ReactProgressBar(Context context, AttributeSet attrs, int
       ↪  defStyleAttr) {
11         super(context, attrs, defStyleAttr);
12     }
13
14     public ReactProgressBar(Context context, AttributeSet attrs, int
       ↪  defStyleAttr, int defStyleRes) {
15         super(context, attrs, defStyleAttr, defStyleRes);
16     }
17
18     @Override
19     public boolean onTouchEvent(MotionEvent event) {
20         ReactContext reactContext = (ReactContext) getContext();
21         reactContext
22                 .getJSModule(RCTEventEmitter.class)
23                 .receiveEvent(getId(), "onPress", null);
24         return super.onTouchEvent(event);
25     }
26 }
```

Quellcode 4.28: Inhalt der Datei `ReactProgressBar.java`.

Die generische Klasse `SimpleViewManager<>` wird dabei mit der zuvor definierten Klasse `ReactProgressBar` verwendet, da die Aufgabe der Klasse darin besteht, `ReactProgressBar` zu verwalten. Im Konstruktor wird der zur Verfügung gestellte `ReactApplicationContext` als Klassenvariable gespeichert, sollte dieser in Zukunft Verwendung finden. Der Rückgabewert der Methode `getName` definiert den Namen der nativen Komponente. Dieser Name sollte dabei identisch zu dem verwendeten Namen aus Abschnitt 4.5.1 sein. Ebenso ist es notwendig, die Methode `createViewInstance`[1] zu implementieren. Diese Methode erwartet als Rückgabewert den Typ, der als *Generic* verwendet wurde. In diesem Fall ist dies `ReactProgressBar`. Dazu wird eine neue Instanz von `ReactProgressBar` erstellt, wobei der `progress`-Wert initial auf 0 festgelegt und zuletzt die erstellte Instanz zurückgegeben wird. Anschließend wird eine Methode namens `setProgress` definiert, welche zusätzlich mit `@ReactProp` annotiert wird. Die Annotation `@ReactProp` nimmt dabei einen `name` entgegen, welcher zu einem *Prop* aus der React Native Definition korrelieren muss. Der Name des *Props* wird dabei ausgehend der Definition und der iOS-Implementierung als `progress` festgelegt.

1 `createViewInstance` ist dabei das Pendant der Methode `view` aus iOS (vgl. Abschnitt 4.5.1)

Weiterhin ist es möglich, Standardwerte für primitive Datentypen über die Attribute `defaultFloat`, `defaultInt` und `defaultBoolean` festzulegen. In diesem Fall wird der Standardwert von `progress` mit 0.0f initialisiert. Die Methode `setProgress` nimmt als Argumente zum einen die von React Native verwaltete `ReactProgressBar` und zum anderen einen `float` Wert entgegen. Der `float` spiegelt dabei den Wert des *Props* `progress` wider. Demzufolge wird der `float`-Wert auf der Instanz von `ReactProgressBar` über die Methode `setProgress` aktualisiert. Dabei gilt es, die vorige Definition zu beachten, dass `progress` einer Fließkommazahl zwischen 0,0 und 1,0 entspricht. `ProgressBar` benötigt als Wert für `progress` jedoch einen Wert zwischen 0 und 100, weswegen der Wert von `progress` mit 100 multipliziert und anschließend in eine Ganzzahl umgewandelt wird. Die Methode `getExportedCustomBubblingEventTypeConstants` sorgt dafür, dass das in Quellcode 4.28 definierte *Event* `onPress` an React Native weitergeleitet wird. (siehe Quellcode 4.29)

Zuletzt besteht die Notwendigkeit, den erstellten `ViewManager` bei React Native zu registrieren. Dazu ist es, wie auch bei nativen Modulen (vgl. Abschnitt 4.2.2), erforderlich, ein `Package` zu erstellen. Das erweiterte Interface `ReactPackage` gibt dabei, neben `createNativeModules`, ebenfalls die Methode `createViewManagers` vor. In letzterer Methode muss der erstellte `ViewManager` instanziiert und zurückgegeben werden. (siehe Quellcode 4.30) [66, 67]

4.5.3 Verwendung der nativen Komponente

Damit die erstellte native Komponente als React Komponente verwendet werden kann, muss diese zunächst von React Native importiert werden. Dazu ist es notwendig, einen neuen Ordner namens `native-components` anzulegen. Innerhalb des Ordners wird die Datei `ProgressBar.tsx` angelegt. Die Datei beinhaltet dabei einen Import der Funktion `requireNativeComponent` von React Native. `requireNativeComponent` erlaubt dabei das Selektieren einer nativen Komponente über den dazugehörigen `viewName`[1]. Da die Funktion `requireNativeComponent` generisch ist, ist es beim Aufruf der Funktion möglich, die von der nativen Komponente angebotenen *Props* festzulegen. Der Rückgabewert von `requireNativeComponent` bei der Verwendung von *RTNProgressBar* als `viewName` wird dabei der Variable `ProgressBar` zugewiesen und gleichzeitig für die Verwendung in anderen Komponenten exportiert. (siehe Quellcode 4.31) Bei komplexeren nativen Komponenten empfiehlt es sich, eine neue React Komponente zu definieren, welche wiederum `ProgressBar` verwendet.

1 Dieser wurde unter iOS durch die Verwendung des Makros und unter Android durch das Überschreiben der Methode `getName` festgelegt.

```java
public class ReactProgressBarManager extends
    SimpleViewManager<ReactProgressBar> {

    private final ReactApplicationContext mCallerContext;

    public ReactProgressBarManager(ReactApplicationContext reactContext) {
        mCallerContext = reactContext;
    }

    @NonNull
    @Override
    public String getName() {
        return "RTNProgressBar";
    }

    @NonNull
    @Override
    protected ReactProgressBar createViewInstance(@NonNull
        ThemedReactContext themedReactContext) {
        ReactProgressBar bar = new ReactProgressBar(themedReactContext,
            null, android.R.attr.progressBarStyleHorizontal);
        bar.setProgress(0);
        return bar;
    }

    @ReactProp(name = "progress", defaultFloat = 0.0f)
    public void setProgress(ReactProgressBar bar, @Nullable float progress)
        {
        bar.setProgress((int) (progress * 100));
    }

    public Map getExportedCustomBubblingEventTypeConstants() {
        return MapBuilder.builder().put(
                "onPress",
                MapBuilder.of(
                        "phasedRegistrationNames",
                        MapBuilder.of("bubbled", "onPress")
                )
        ).build();
    }
}
```

Quellcode 4.29: Inhalt der Datei `ReactProgressBarManager.java`.

Dadurch können innerhalb dieser neuen React Komponente z. B. Hooks verwendet werden, um das Verhalten der nativen Komponente den Gegebenheiten anzupassen. [62, 66]

```
... implements ReactPackage {
    @NonNull @Override
    public List<ViewManager> createViewManagers(@NonNull
    ↪  ReactApplicationContext reactApplicationContext) {
        return List.of(new
        ↪  ReactProgressBarManager(reactApplicationContext));
    }
}
```

Quellcode 4.30: Methode `createViewManagers` eines `ReactPackages`.

```
import { requireNativeComponent, StyleProp } from 'react-native';

export const ProgressBar = requireNativeComponent<{
  progress?: number;
  onPress?: () => void;
  style?: StyleProp<any>;
}>('RTNProgressBar');
...
export function App() {
    ...
    return (
        <View>
            ...
            <ProgressBar progress={0.5} style={{width: '80%'}} onPress={()
            ↪  => console.log('pressed!')} />
        </View>
    );
}
```

Quellcode 4.31: Inhalt der Datei `ProgressBar.tsx` und die Verwendung von `<ProgressBar />`.

4.6 Fabric Native Components

Fabric Native Components sind analog zu nativen Komponenten (vgl. Abschnitt 4.5), mit dem Unterschied, dass diese die Render Engine *Fabric* (vgl. Abschnitt 4.1) verwenden. Die Dokumentationsqualität und -dichte ist dabei vergleichbar gering zu der Dokumentation der *Turbo Native Modules* (vgl. Abschnitt 4.4). Ferner beinhaltet die Dokumentation keine Information darüber, wie *Fabric Native Components* innerhalb bereits bestehender React Native Projekte erstellt werden können.

Aufgrund der geringen Informationsdichte von *Fabric Native Components* sowie der Tatsache, dass diese im Verlauf dieser Ausarbeitung keine weitere Verwendung finden, wird das Verfahren zum Erstellen von *Turbo Native Components* nicht weiter besprochen. Da es zusätzlich möglich ist, native Komponenten (vgl. Abschnitt 4.5) über einen Kompatibilitätsmodus mit *Fabric* zu betreiben, können alternativ native Komponenten

verwendet werden, falls diese benötigt werden. Dazu ist es lediglich notwendig, eine Datei namens `react-native.config.js` im gleichen Verzeichnis der `package.json` zu erstellen, in welcher die Komponenten eingetragen werden, welche *Fabric* bislang nicht unterstützen und den Kompatibilitätsmodus verwenden sollen. (siehe Quellcode 4.32) Unterstützt eine Komponente die neue Architektur nicht, ist dies daran zu erkennen, dass anstelle der Komponente innerhalb der App ein rot/pinkfarbener Bereich angezeigt wird, welcher die Nachricht *'<Komponente>' is not Fabric compatible yet.* trägt. [49, 68]

```
module.exports = {
    project:{
      android: {
        unstable_reactLegacyComponentNames: [ "ComponentName" ]
      },
      ios: {
        unstable_reactLegacyComponentNames: [ "ComponentName" ]
      }
    },
  };
```

Quellcode 4.32: Beispiel von `react-native.config.js` zur Aktivierung des Kompatibilitätsmodus spezifischer Komponenten.

4.7 Autolinking

Wird eine Drittbibliothek verwendet, welche nativen Modulen und Komponenten bzw. *Turbo Native Modules* und *Fabric Native Components* beinhaltet, existieren in dieser Bibliothek Unterprojekte für iOS und Android. Innerhalb dieser Unterprojekte befinden sich dabei die Implementierungen der Module und Komponenten. Dabei besteht die Notwendigkeit, dass diese Implementierungen bei der Installation der Bibliothek in die eigenen nativen iOS und Android Apps übertragen werden, sodass diese in der eigenen Anwendung verwendet werden können.

Diese Aufgabe übernimmt dabei das sogenannte *Autolinking*. Ausgehend einer Definition innerhalb der installierten Bibliothek, werden das iOS Unterprojekt über *CocoaPods* und das Android Unterprojekt über *Gradle* in die nativen Apps integriert. Würde *Autolinking* nicht existieren, bestände die Notwendigkeit, jede React Native Bibliothek mit nativen Modulen oder nativen Komponenten manuell in den Build-Prozess der Anwendung aufzunehmen und bei Änderungen innerhalb der Bibliothek anzupassen.

Da *CocoaPods* kein fester Bestandteil des Build-Prozesses bei iOS sind[1], ist es notwendig, nach der Installation einer Bibliothek die verwendeten *CocoaPods* neu zu installieren[2], um so den *Autolinking*-Prozess zu initiieren. Da *Gradle* unter Android ein fester Bestandteil des Build-Prozesses ist, besteht keine Notwendigkeit zur manuellen Initiierung des *Autolinkings*. [69]

1 Eine iOS-Anwendung ist ohne die Verwendung von *CocoaPods* möglich.

2 Dies ist mit dem Kommando `pod install` bzw. `USE_FABRIC=1 RCT_NEW_ARCH_ENABLED=1 pod install` im `ios`-Ordner möglich.

5 Beispielhafte Integration von React Native in bestehende Apps

Auf Grundlage der Fallstudien (vgl. Kapitel 2) und der technischen Einführung in React Native (vgl. Kapitel 4), wird nun eine exemplarische Integration von React Native in bestehende Apps vorgenommen.

5.1 Beispielapps und methodisches Vorgehen

Die Integration von React Native in bereits existierende iOS und Android Apps ist dabei ausreichend dokumentiert. Hierbei gilt es jedoch zu beachten, dass sich das Vorgehen von Version zu Version React Natives unterscheiden kann. Dabei wird die zum Zeitpunkt der Verschriftlichung aktuellste React Native Version `0.72.4` verwendet. React Native stellt jedoch zu jeder Versionsänderung eine Migrationshilfe bereit. Diese ist unter `https://react-native-community.github.io/upgrade-helper/?from=0.72.4` zu erreichen.

Die vorab erstellten Apps wurden unter Verwendung von XCode bzw. Android Studio als neue Anwendungen erstellt. Anschließend wurden diese Apps durch native Bedienelemente erweitert. So verwendet die iOS App eine Bottom Navigation, während die Android App einen Navigation Drawer beinhaltet. Dies soll die nativen Apps später eindeutig von der React Native Integration unterscheidbar machen und ebenso ein realistischeres und repräsentatives Szenario abbilden.

Die iOS App wurde dabei mit Swift initialisiert. Grund hierfür ist, dass eine Auswertung aus dem Jahr 2022 offengelegt hat, dass 90 % der bekanntesten iOS Apps Swift verwenden. Folglich ist eine Swift App repräsentativer als eine Objective-C App.[1] [70] Die Android App hingegen wurde mit Java und nicht mit Kotlin initialisiert. Auch wenn laut Android selbst 95 % der Top 1000 Apps Kotlin verwenden. [71] Grund dafür ist, dass

1 Die Auswertung beinhaltet die 100 umsatzstärksten Apps des Apple App Stores, welche zugleich keine Spiele sind.

Android Studio es ermöglicht, Java-Code automatisiert in Kotlin-Code zu konvertieren.[1] Allerdings bietet die Entwicklungsumgebung keine Möglichkeit, Kotlin-Code in Java-Code umzuwandeln. Mithilfe der Konvertierungsfunktion von Android Studio kann eine App in Java beide Programmiersprachen abbilden und ist daher repräsentativ. Eine vergleichbare Konvertierungsfunktion ist innerhalb von XCode für Objective-C nicht verfügbar.

Für die Plattformen iOS und Android wird im Folgenden die Integration React Natives schrittweise durchgeführt. Nebst der textuellen Beschreibungen der durchzuführenden Schritte, stehen die verwendeten Apps als öffentliches Repository bereit. Die Anleitung orientiert sich stark an der offiziellen Dokumentation zur Integration von React Native in bestehende Apps. [72]

Für die Integration von React Native in native iOS und Android Apps werden Kenntnisse über die Entwicklung von iOS Apps mit Swift oder Objective-C sowie die Entwicklung von Android Apps mit Java vorausgesetzt. Ebenso wird die Annahme getroffen, dass sich die bestehenden iOS und Android Apps innerhalb eines *Monorepos*[2] befinden und die Implementierung der iOS App sich auf derselben Ordnerebene befindet wie die Android App. Weiterhin ist es der Übersichtlichkeit dienlich, wenn sich die iOS und Android App in einem separaten Ordner befinden, da es für die Integration notwendig ist, relativ zur iOS und Android Implementierung Dateien zu erstellen. Konkret bedeutet dies, dass die empfohlene Ordnerstruktur diesem Beispiel folgt: `app/ios` für die iOS sowie `app/android` für die Android App. Der Name des Ordners `app` kann dabei frei gewählt werden, sollte jedoch neben den zwei beschriebenen Ordnern ansonsten möglichst leer sein.

Die vollständige Implementierung der Integration ist im Git-Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) unter dem *Branch* `react-native-integration` verfügbar. Abbildung A.2 und Tabelle B.1 dienen als Navigationshilfen der verfügbaren *Branches*.

Bei den vorgestellten Integrationen ist besonders hervorzuheben, dass React Native in bereits bestehende *Views* hinzugefügt wird. So ist es z. B. möglich, zwischen der nativen Lösung und der React Native Integration im laufenden Betrieb zu wechseln. Dies könnte dabei anhand einer App-Einstellung umgesetzt werden, welche darüber entscheidet, ob die React Native oder die native Implementierung verwendet werden soll.

1 Innerhalb einer geöffneten Java-Datei ist dies über den Menüpunkt *Code → Convert Java File to Kotlin File* möglich.

2 Ein *Monorepo* beschreibt versioniertes Repository, in dem sich mehrere unterschiedliche Projekte befinden. [73, 74] In diesem konkreten Beispiel müssen sich demnach das Projekt der iOS App im selbigen Repository befinden wie die Android App.

Die Community-Bibliothek `react-native-brownfield` zielt darauf ab, die Integration von React Native in bestehende Apps zu vereinfachen. Da diese Bibliothek jedoch seit 2019 nicht mehr aktualisiert wurde und somit veraltete Versionen zum Einsatz kommen, wird lediglich die offizielle Dokumentation für die Integration verwendet. (vgl. `https://github.com/callstack/react-native-brownfield`, aufgerufen am 26. Oktober 2023)

Die Abschnitte 5.2 bis 5.6 behandeln dabei die folgenden Aspekte:

1. Abschnitt 5.2: Initialisierung von React Native in den nativen Apps.
2. Abschnitt 5.3: Definition einer Beispiel React Native Komponente, welche im weiteren Verlauf innerhalb der nativen Apps angezeigt wird.
3. Abschnitt 5.4 und 5.5: Notwendige Anpassungen an der iOS und Android App für das Anzeigen von React Native Komponenten.
4. Abschnitt 5.6: Behandlung der erforderlichen Modifikationen an den nativen iOS und Android Apps für die Verwendung der neuen Architektur bzw. der *Fabric* (vgl. Abschnitt 4.1) Render-Engine.

5.2 Initialisierung von React Native

Als erster Schritt ist es notwendig, ein React Native Projekt zu initialisieren. Dazu kommt der Package-Manager *yarn* zum Einsatz. Alternativ ist es möglich, *npm* oder andere Node-Package-Manager zu verwenden. Die folgenden *yarn* Befehle müssen in diesem Fall jedoch manuell angepasst werden.

Zunächst wird eine neues *yarn*-Projekt mit dem Kommando `yarn init` initialisiert. Die interaktiven Abfragen im Terminal können dabei entweder bereits befüllt oder aber mit *Enter* bestätigt werden. Nun ist es notwendig, React Native als Abhängigkeit mit dem Kommando `yarn add react-native@0.72.4` hinzuzufügen. Dabei ist vorgesehen, dass eine Warnung bezüglich einer fehlenden *Peer-Dependency* durch *React* angezeigt wird. In dieser Warnung ist dabei stets eine konkrete Version der Abhängigkeit React angegeben. Im Falle von React Native `0.72.4` ist dies React `18.2.0`, welches mittels `yarn add react@18.2.0` installiert werden muss.

Damit TypeScript verwendet werden kann, ist es erforderlich, diese Abhängigkeit zu installieren. Zusätzlich ist es notwendig `@react-native/metro-config` sowie `@tsconfig/react-native` hinzuzufügen. *Metro* ist dabei dafür zuständig, den React-Code in ein *Bundle* zusammenzuführen und dieses den Apps zur Verfügung zu stellen. *Watchman* ist in der Lage, Änderungen an Dateien zu erkennen. Wird

eine solche Änderung erkannt, wird das *Bundle* durch *Metro* aktualisiert. Dies ermöglicht somit das *Hot-Reload*-Feature, wodurch schnell mittels Änderungen an den Apps experimentiert werden kann. `@tsconfig/react-native` hingegen wird verwendet, um die korrekte TypeScript Konfiguration in Verbindung mit React Native zu verwenden. Da React von sich aus keine TypeScript-Unterstützung anbietet, ist es notwendig, die *Types* aus `@types/react` nachträglich zu installieren. Alles ist mit `yarn add -D typescript@5.2.2 @react-native/metro-config@0.72.11 @tsconfig/react-native @types/react@18.2.0`[1] zu installieren. Nun kann TypeScript mit dem Kommando `npx tsc --init` initialisiert werden. (siehe Abbildung 5.1) Zuletzt ist es erforderlich, eine Konfigurationsdatei für Metro anzulegen. Zunächst beinhaltet diese Datei lediglich die Standardkonfiguration *Metros*. Im Verlauf der fortschreitenden Entwicklung mit React Native kann es erforderlich sein, weitere *Plugins* in dieser Konfigurationsdatei aufzunehmen, sofern dies z. B. externe Bibliotheken vorgibt. Diese Datei trägt dabei den Namen `metro.config.js` und wird im selben Verzeichnis wie die Datei `package.json` angelegt. (siehe Quellcode 5.1)

```
> yarn init
> yarn add react-native@0.72.4
> yarn add react@18.2.0
> yarn add -D typescript@5.2.2 @react-native/metro-config@0.72.11 @tsconfig/react-native @types/react@18.2.0
> npx tsc --init
```

(Erstellt mit `https://carbon.now.sh/`, abgerufen am 26. Oktober 2023)

Abbildung 5.1: Übersicht aller auszuführenden Kommandos zur Initialisierung von React Native.

```
1  const { getDefaultConfig, mergeConfig } =
   ↪  require('@react-native/metro-config');
2
3  const config = {};
4
5  module.exports = mergeConfig(getDefaultConfig(__dirname), config);
```

(vgl. `https://raw.githubusercontent.com/react-native-community/rn-diff-purge/release/0.72.3/RnDiffApp/metro.config.js`)

Quellcode 5.1: Inhalt der Datei `metro.config.js`.

1 `-D` installiert die notwendigen Abhängigkeiten als `devDependencies`.

Damit die korrekte TypeScript-Konfiguration aus `@tsconfig/react-native` verwendet werden kann, ist es notwendig, die Datei `tsconfig.json`[1] anzupassen. Dabei ist es notwendig, den gesamten Inhalt der Datei mit Quellcode 5.2 auszutauschen.

```
1 {
2   "extends": "@tsconfig/react-native/tsconfig.json"
3 }
```

Quellcode 5.2: Inhalt der Datei `tsconfig.json`.

Aus Gründen der Übersichtlichkeit wird ein Ordner `react-native` angelegt, in welchem sich später jeglicher React Code befinden wird. Im gleichen Ordner wie die Datei `package.json` ist es darüber hinaus notwendig, eine Datei namens `index.js` anzulegen. Diese wird im Verlauf der Entwicklung alle weiteren erstellten Komponenten inkludieren. React Native verwendet diese Datei als Einstiegspunkt für die gesamte React Anwendung.

In der Datei *package.json* ist es notwendig, einige Startskripte aufzunehmen, die die Entwicklung mit React Native vereinfachen. Diese sind dabei am Ende der Datei anzufügen. (siehe Quellcode 5.3)

```
1 {
2     ...,
3     "scripts": {
4         "start": "react-native start",
5         "ios": "react-native run-ios",
6         "android": "react-native run-android",
7         "android:port": "adb reverse tcp:8081 tcp:8081"
8     }
9 }
```

Quellcode 5.3: Startskripte der `package.json`.

Nun besteht die Möglichkeit, den *Metro-Bundler* mit dem Kommando `yarn start` zu starten. In Abbildung 5.2 ist die resultierende Ordnerstruktur nach der Initialisierung von React Native abgebildet.

1 Diese Datei wurde mit dem Kommando `npx tsc -init` angelegt.

```
.
├── android
│   └── ...
├── index.js
├── ios
│   └── ...
├── metro.config.js
├── package.json
├── react-native
│   └── components
├── react-native.config.js
├── result.html
├── tsconfig.json
└── yarn.lock
```

(Erstellt mit `tree` (vgl. `https://wiki.ubuntuusers.de/tree/`), abgerufen am 26. Oktober 2023)

Abbildung 5.2: Übersicht der resultierenden Ordnerstruktur innerhalb des *Monorepos*.

5.3 Beispiel React Native Komponente

Um eine React Native Komponente in die bestehenden iOS und Android Apps zu integrieren, wird zunächst eine Beispielkomponente namens `Counter.tsx` implementiert. Diese Komponente soll dabei einen Zähler abbilden, welcher über einen Button erhöht und über einen anderen verringert werden kann. Der Initialwert des Zählers kann dabei über das *Prop* `initialCount` der Komponente übergeben werden. Wird kein `initialCount` definiert, startet der Zähler bei 0. Zunächst wird für die Verwaltung des Zählerwerts der *Hook* (vgl. Abschnitt 3.4.1) `useState` verwendet.

Die Komponente `Counter` zeigt dabei zum einen den aktuellen Zählerstand an, darüber hinaus beinhaltet die Komponente einen React Native Schriftzug, welcher die React Native Komponente deutlich von der nativen App unterscheiden soll.

In der Greenfield-Entwicklung mit React Native existiert meist nur eine React Native Komponente, welche von den nativen Apps angezeigt wird. Bei der Verwendung von React Native in einem Brownfield-Umfeld müssen React Native Komponenten explizit für die Verwendung in den nativen Apps exportiert[1] werden. Dazu wird die Methode `AppRegistry.registerComponent()` verwendet. Diese muss dabei mit den Argumenten `appKey` und einer `getComponentFunc`-Funktion aufgerufen werden. Der `appKey` ist ein eindeutiger Identifikator zur Selektion der React Native Komponente innerhalb der nativen Apps. `getComponentFunc` ist eine Funktion, die eine React-

1 Dabei reicht keine Verwendung des JavaScript Keywords `export`.

Komponente erzeugen muss. In diesem Fall ist es lediglich erforderlich, `Counter` als Rückgabewert einer Funktion zurückzugeben. (siehe Quellcode 5.4)

```
type CounterProps = {initialCount?: number};

export function Counter({initialCount}: CounterProps) {
  const [count, setCount] = useState(initialCount ?? 0);

  const increment = useCallback(
    () => setCount(oldCount => oldCount + 1),
    [setCount],
  );

  const decrement = useCallback(
    () => setCount(oldCount => oldCount - 1),
    [setCount],
  );

  return (
    <View>
      <View style={styles.container}>
        <View style={styles.reactNativeIndicator}>
          <Text style={{fontSize: 32, color: 'rgba(64, 64, 64, .4)'}}>
            React Native
          </Text>
        </View>
        <Text style={{fontSize: 32}}>The current count is: {count}</Text>
        <View style={styles.buttonView}>
          <Button title="increment (+1)" onPress={increment} />
          <Button title="decrement (-1)" onPress={decrement} />
        </View>
      </View>
    </View>
  );
}

const styles = StyleSheet.create({...});

AppRegistry.registerComponent('Counter', () => Counter);
```

Quellcode 5.4: Definition der beispielhaften React Native Komponente `Counter.tsx` für die Verwendung in den nativen iOS und Android Apps.

Da in React Native standardmäßig die Datei `index.js` als Einstiegsdatei konfiguriert ist, besteht die Notwendigkeit, diese Datei anzulegen. Innerhalb von `index.js` ist es dabei erforderlich, alle verwendeten Komponenten[1] zu importieren.

```
import "./react-native/components/Counter";
```

Quellcode 5.5: Inhalt der Einstiegsdatei `index.js`.

1 Dies betrifft alle React Komponenten, welche über die `AppRegistry` von React Native registriert werden.

Die Darstellung der *Counter*-Komponente ist in Abbildung 5.4a für iOS und in Abbildung 5.5a für Android zu sehen.

5.4 Integration von React Native in eine iOS-App

Die Integration von React Native in die iOS App erfolgt in vier Phasen. Zunächst ist es notwendig, Änderungen am Build-Prozess der iOS-App vorzunehmen. (vgl. Abschnitt 5.4.1) Anschließend kann die iOS App erweitert werden, um eine Verbindung mit dem *Metro-Bundler* aufzunehmen. Auf diese Weise ist es der iOS App möglich, aktualisierte *Bundles* aufzurufen. (vgl. Abschnitt 5.4.2) Nach weiteren Anpassungen ist es möglich, React Native Komponenten innerhalb der iOS-App darzustellen. Dabei werden zwei unterschiedliche Methoden zur Integration der React Native Komponenten behandelt. (vgl. Abschnitt 5.4.3) Abschließend wird gezeigt, welche speziellen Modifikationen notwendig sind, um die App mit der React Native Integration für den Einsatz in der Produktion zu konfigurieren. (vgl. Abschnitt 5.4.4)

5.4.1 Build-Anpassungen

Damit die Integration von React Native in eine bestehende iOS App erfolgen kann, ist es zunächst notwendig sicherzustellen, dass im Projekt *CocoaPods* initialisiert sind. Grund hierfür ist, dass React Native *Pods* für die Integration in iOS Apps verwendet. Sollte *CocoaPods* nicht bereits initialisiert sein, ist es notwendig innerhalb des `ios`-Ordners das Kommando `pod init`[1] auszuführen. Die Voraussetzung dafür ist jedoch, dass *CocoaPods* auf dem verwendeten Gerät bereits installiert[2] ist. [75]

Anschließend ist es erforderlich, einige Änderungen an der `Podfile`-Datei vorzunehmen. Die Verwendung von `use_frameworks!` innerhalb des *Podfiles* ist dabei inkompatibel mit dem React Native Debugger *Flipper*. Um dieses Problem zu umgehen, ist es entweder notwendig, die Zeile `:flipper_configuration =>flipper_config,` auszukommentieren oder die Verwendung von `use_frameworks!` zu entfernen. (siehe Quellcode 5.6) Zeile 3−7 ermöglichen dabei die Integration der Datei `react_native_pods.rb`. Diese Datei von React Native beinhaltet dabei weitere Abhängigkeiten, welche von React Native benötigt werden. [76] Zeile 9 legt die minimale iOS-Version für das Projekt fest. Der Aufruf von `prepare_react_native_project` in Zeile 10, welcher in `react_native_pods.rb` definiert wird, ist unter anderem für die Unterdrückung von

1 Sollte nach einem `pod install` kein Bauen des Projektes mehr möglich sein, ist es notwendig die Datei mit der Endung `.xcworkspace` statt `.xcodeproj` mit Xcode zu öffnen.
2 Informationen über die Installation von *CocoaPods* sind in der angegebenen Quelle zu finden.

Fehlern von duplizierten Identifikatoren notwendig. [76] Zeile 12 aktiviert die Integration des Debuggers *Flipper*, falls dies nicht über die Umgebungsvariable `NO_FLIPPER` explizit deaktiviert wurde. Sollte die Verwendung von `use_frameworks!` notwendig sein, ist es bei der Integration von React Native erforderlich, die Verknüpfung von *Pods* zu konfigurieren. Diese Konfiguration wird in den Zeilen $14 - 18$ vorgenommen. In Zeile $22 - 39$ werden weitere Konfigurationen von React Native vorgenommen. Darunter werden zunächst `flags` aus den Umgebungsvariablen ausgelesen, welche im Folgenden für den Aufruf von `use_react_native` verwendet werden. `use_react_native` legt dabei den Einstiegspfad von React Native, die Konfiguration des Debuggers *Flipper*, den Pfad der iOS App sowie die Aktivierung von *Hermes* und *Fabric* fest. Zuletzt ist es notwendig, `react_native_post_install` aufzurufen.

Nachdem die Änderungen am *Podfile* durchgeführt wurden, ist es notwendig, die aktualisierten *Pods* mit dem Kommando `pod install` zu installieren.

5.4.2 Verbindung zu React Native

Um im Debug-Modus der App eine Verbindung mit dem *Metro-Bundler* herzustellen, sind Anpassungen am Einstiegspunkt der iOS-App vorzunehmen. Diese Datei ist dabei mit einem `@main` oder `@UIApplicationMain` annotiert[1]. Dazu ist es zunächst notwendig, React zu importieren, wobei React von React Native bereitgestellt wird. Anschließend ist es erforderlich, das Protokoll[2] `RCTBridgeDelegate` in der Klasse zu implementieren. Daraufhin ist es notwendig, die Methode `sourceURL` zu definieren. Diese beinhaltet die URL zum *Metro*-Server oder den Pfad zum lokalen *Bundle* im Produktionsumfeld.[3] Daraufhin werden zwei Klassen-Variablen definiert. Ersterer beinhaltet dabei die initialisierte `RCTBridge` und stellt diese statisch bereit, letzterer beinhaltet ein `UIWindow`, welches jedoch nicht manuell initialisiert werden muss. Die Initialisierung des `UIWindows` übernimmt React Native. Zuletzt wird innerhalb der Methode `func application` die Instanz der `RCTBridge` zugleich initialisiert wie auch zugewiesen. (siehe Quellcode 5.7)

Weitere Änderungen für die Verbindung im Debug-Modus der App sind nicht notwendig. Für die korrekte Funktionsweise des Produktionsbuilds, sind jedoch weitere Modifikationen erforderlich. Diese Anpassungen sind in Abschnitt 5.4.4 beschrieben.

1 In der Regel heißt diese Klasse *AppDelegate.swift*
2 Protokolle in Swift sind mit Interfaces vergleichbar.
3 Wird ein Simulator auf demselben Mac verwendet, auf dem der *Metro*-Server gestartet wurde, ist die Verwendung von `localhost` möglich. Ansonsten ist es notwendig, `localhost` mit die IP-Adresse oder der Hostname des Rechners auszutauschen, auf dem der *Metro*-Server ausgeführt wurde.

```ruby
...
# Resolve react_native_pods.rb with node to allow for hoisting
require Pod::Executable.execute_command('node', ['-p',
  'require.resolve(
    "react-native/scripts/react_native_pods.rb",
    {paths: [process.argv[1]]},
  )', __dir__]).strip

platform :ios, min_ios_version_supported
prepare_react_native_project!

flipper_config = ENV['NO_FLIPPER'] == "1" ? FlipperConfiguration.disabled :
↪  FlipperConfiguration.enabled

linkage = ENV['USE_FRAMEWORKS']
if linkage != nil
  Pod::UI.puts "Configuring Pod with #{linkage}ally linked Frameworks".green
  use_frameworks! :linkage => linkage.to_sym
end

target 'example integration' do
  ...
  config = use_native_modules!

  # Flags change depending on the env values.
  flags = get_default_flags()

  use_react_native!(
    :path => config[:reactNativePath],
    :hermes_enabled => flags[:hermes_enabled],
    :fabric_enabled => flags[:fabric_enabled],
    :flipper_configuration => flipper_config,
    :app_path => "#{Pod::Config.instance.installation_root}/.."
  )

  post_install do |installer|react_native_post_install(
      installer,
      config[:reactNativePath],
      :mac_catalyst_enabled => false
    )
  end
end
```

(übernommen aus `https://github.com/facebook/react-native/blob/04ad34d6ad64c702d6619aa5a358ddd376f8be8e/packages/react-native/template/ios/Podfile`)

Quellcode 5.6: Inhalt der Datei `Podfile` mit den nötigen Anpassungen.

5.4.3 Aufruf einer React Native Komponente

Da nun eine Verbindung zum Dev-Server oder zum lokalen JavaScript-Bundle aufgebaut werden kann, besteht die Möglichkeit, eine exportierte React Native Komponente zur Anzeige zu bringen. Dabei werden zwei verschiedene Arten der Integration vorgestellt.

```
...
import React

@main
class AppDelegate: UIResponder, UIApplicationDelegate, RCTBridgeDelegate {

    func sourceURL(for bridge: RCTBridge!) -> URL! {
#if DEBUG
        return URL(string:
        ↪ "http://localhost:8081/index.bundle?platform=ios")
#else
        return Bundle.main.url(forResource: "main", withExtension:
        ↪ "jsbundle")
#endif
    }

    static var bridge: RCTBridge?
    var window: UIWindow?

    func application(_ application: UIApplication,
    ↪ didFinishLaunchingWithOptions launchOptions:
    ↪ [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        AppDelegate.bridge = RCTBridge(delegate: self, launchOptions:
        ↪ launchOptions)
        return true
    }
    ...
}

```

Quellcode 5.7: Inhalt der Datei `AppDelegate.swift` mit den nötigen Anpassungen.

Die erste Methode ist besonders für bildschirmfüllende React Native Komponenten geeignet. Die zweite Methode hingegen ermöglicht das Anzeigen einer Komponente neben bestehenden, nativen Eingabeelementen. Für beide vorgestellten Methoden ist es notwendig, eine Klasse zu verwenden, welche `UIViewController` erweitert.

5.4.3.1 Bildschirmfüllende Methode

Bei dieser Methode wird lediglich der `view`-Member des `UIViewControllers` ersetzt. Der `view`-Member einer `UIViewController`-Klasse ist verantwortlich für die Komponente, welche letztlich auf dem Bildschirm dargestellt wird. Eine React Native View kann dabei mittels `RCTRootView` erstellt werden. Der Konstruktor[1] dieser Klasse nimmt dabei die zuvor erstellte statische *Bridge* und den in Abschnitt 5.3 definierten Komponentennamen entgegen. Eine auf diese Weise erstellte Instanz kann so dem `view`-Member direkt zugewiesen werden. Weiterhin ist es möglich, der ausgewählten

1 Bzw. die `init`-Methode).

Komponente `initialProperties` zu übergeben, welche von der React Komponente verarbeitet werden können. Im konkreten Beispiel der *Counter*-Komponente, ist dies der *Prop* `initialCount`, welcher mit einem zufälligen Zählerwert zwischen 1 und 100 vorbelegt wird. (siehe Quellcode 5.8)

```
1  import UIKit
2  import React
3
4  class FullscreenViewController: UIViewController {
5      override func viewDidLoad() {
6          super.viewDidLoad()
7          title = "Fullscreen"
8          self.view = RCTRootView(bridge: AppDelegate.bridge!, moduleName:
           ↪  "Counter", initialProperties: ["initialCount": Int.random(in:
           ↪  1..<101)])
9      }
10 }
```

Quellcode 5.8: Inhalt der Datei `FullscreenViewController.swift`.

Bei diesem Verfahren ist zu berücksichtigen, dass bestehende Elemente des `UIViewControllers` vollständig ersetzt werden. Somit ist auch darauf zu achten, dass die React Native Komponente innerhalb einer *SafeArea* verwendet wird. Ansonsten kann dies zur Folge haben, dass Elemente der React Native Komponente, welche sich z. B. unterhalb der Notch[1] oder Titelleiste befinden, für den Nutzenden nicht sichtbar sind.

5.4.3.2 Nicht bildschirmfüllende Methode

Zusätzlich ist es möglich eine React Native Komponente auf einer View zu integrieren, die bereits native iOS-Elemente beinhaltet. Dabei ist es ebenso möglich, mittels nativer iOS-Eingabeelemente die React Native View zu steuern. So wird im folgenden Beispiel die React Native View nicht zu Beginn, sondern erst beim Betätigen eines Buttons geöffnet. Weiterhin ist es über native iOS Buttons möglich, den Zählerwert der React Native Komponente zu verändern.

Der Aufbau des `UIViewController` ist in Abbildung 5.3 zu sehen.

1 Einige iOS-Geräte besitzen eine Aussparung für die Frontkamera. In diesem Bereich sollten keine UI-Elemente angezeigt werden.

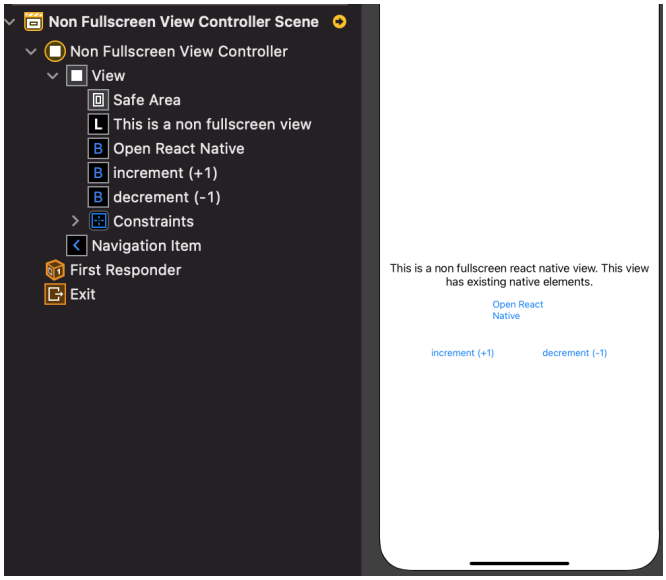


Abbildung 5.3: Aufbau der mit dem `UIViewController` verknüpften View. Zu sehen sind native iOS Komponenten, welche ebenfalls zur Steuerung des Zählers verwendet werden sollen. Dazu existieren zwei Buttons zum Inkrementieren bzw. Dekrementieren des Zählers, ein Button zum Öffnen der React Native Komponente und eine Textkomponente. Die definierten *Constraints* ermöglichen dabei, dass die Komponenten auf ähnliche Weise angeordnet bleiben, auch wenn das verfügbare Testgeräte/Simulatoren verschiedene Displaydimensionen besitzen.

Im Unterschied zur bildschirmfüllenden Methode wird bei der nicht bildschirmfüllenden Methode die `view` des `UIViewControllers` nicht durch die erstellte React Native *View* ersetzt, um zu vermeiden, dass alle zuvor definierten nativen iOS UI-Elemente überschrieben werden. Demzufolge wird die erstellte React Native View über die Methode `addSubview` hinzugefügt. Auf diese Weise hinzugefügte `views` sind dabei jedoch nicht zwangsläufig korrekt positioniert. So ist es notwendig, unter Zuhilfenahme von `NSLayoutConstraint` die notwendigen *Constraints* zu definieren, damit diese *View*, wie gewünscht positioniert wird. In diesem Fall wurden diese *Constraints* auf eine Weise gewählt, sodass die React Native Komponente unterhalb des Buttons zum Erhöhen des Zählers und oberhalb des Endes der umgebenen `view` positioniert wird. Weitere *Constraints* stellen sicher, dass die React Native Komponente die gesamte Breite der `view` einnimmt. Die Logik für die Platzierung der React Native Komponente befindet sich dabei in einer Methode namens `openReactNative`, welche beim Betätigen des entsprechenden Buttons aufgerufen wird. Die Buttons zum Erhöhen und Verringern des Zählers verändern dabei eine lokale Zählvariable in der nativen iOS App, die beim Erstellen des `UIViewControllers` zufällig gewählt wird. Zusätzlich wird `openReactNative` beim Ändern des Zählers erneut aufgerufen, um den aktualisierten Zählerwert der React Native Komponente zur Verfügung zu stellen. (siehe Quellcode 5.9)

```swift
import UIKit
import React
class NonFullscreenViewController: UIViewController {

    var count = 0

    public required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        self.count = Int.random(in: 1..<101)
    }
    override func viewDidLoad() {
        super.viewDidLoad()
        title = "Non Fullscreen"
    }
    @IBAction func openReactNative() {
        let reactNativeView = RCTRootView(bridge: AppDelegate.bridge!,
        ↪  moduleName: "Counter", initialProperties: ["initialCount":
        ↪  self.count])
        self.view.addSubview(reactNativeView)
        reactNativeView.translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate([
            reactNativeView.leadingAnchor.constraint(equalTo:
            ↪  view.leadingAnchor),
            reactNativeView.trailingAnchor.constraint(equalTo:
            ↪  view.trailingAnchor)
        ])
        let anchor = self.view.viewWithTag(1)!
        NSLayoutConstraint.activate([
            reactNativeView.topAnchor.constraint(equalTo:
            ↪  anchor.bottomAnchor),
            reactNativeView.bottomAnchor.constraint(equalTo:
            ↪  view.bottomAnchor)
        ])
    }
    @IBAction func increment() {
        self.count += 1
        self.openReactNative()
    }
    @IBAction func decrement() {
        self.count -= 1
        self.openReactNative()
    }
}
```

Quellcode 5.9: Inhalt der Datei `NonFullscreenViewController.swift`.

5.4.4 Anpassungen für einen Produktionsbuild

Für die Verwendung von React Native in einer Produktivumgebung ist es unter iOS notwendig, zusätzliche Anpassungen vorzunehmen. Im Produktivmodus bezieht React Native das aktuelle JavaScript-Bundle nicht mehr über den *Metro-Bundler*. Das Bundle muss für die App zugänglich im Dateisystem des Gerätes vorliegend sein. Daher ist es

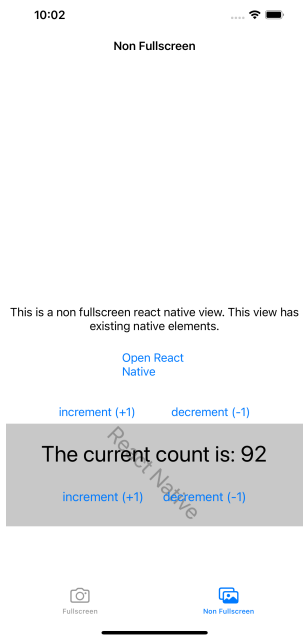
notwendig, das JavaScript-Bundle im Falle eines Produktionsbuilds zunächst zu erstellen und anschließend in das erstellte App-Bundle zu integrieren.

React Native stellt dafür ein Skript namens `react-native-xcode.sh` zur Verfügung, welches ebendiese Aufgaben übernimmt. Dafür ist es lediglich notwendig, innerhalb des iOS-Projects in Xcode in den *Build Phases* eine neue *Run Script Phase* hinzuzufügen. Der Name dieser kann dabei frei gewählt werden, in diesem Beispiel wird dafür *Bundle React Native code* verwendet. Der Inhalt des auszuführenden Skriptes ist Quellcode 5.10 zu entnehmen. [72]

```
1 cd $PROJECT_DIR/..
2 export NODE_BINARY=node
3 ./node_modules/react-native/scripts/react-native-xcode.sh
```

Quellcode 5.10: Inhalt der neuen *Run Script Phase*.

(a) Bildschirmfüllende Integration durch eigene Layout-Datei.



(b) Nicht bildschirmfüllende Integration durch Verwenden eines *FrameLayouts* in einer bestehenden Layout-Datei.

Abbildung 5.4: Bildschirmfüllende (vgl. Abbildung 5.4a) und nicht bildschirmfüllende (vgl. Abbildung 5.4b) Integration von React Native in eine bestehende Android-App.

5.5 Integration von React Native in eine Android-App

Die Integration von React Native in die Android App verläuft in drei Phasen. Dabei werden die gleichen Schritte wie bei der Integration in die iOS-App durchlaufen, allerdings sind keine speziellen Anpassungen für einen Produktionsbuild erforderlich.

5.5.1 Build-Anpassungen

Auch für die Integration von React Native in eine bestehende Android App existieren bestimmte Voraussetzungen. So ist es notwendig, dass die Android-App mindestens die Gradle-Version `8.0.1` verwendet.

React Native verwendet für die Integration in eine Android-App *Gradle*. Deswegen ist es zunächst erforderlich, das React Native Plugin für *Gradle* zu installieren. Dies ist über das Kommando `yarn add -D react-native-gradle-plugin@0.71.19` durchzuführen. Anschließend ist es möglich, in der Datei `settings.gradle` im `android`-Ordner die Zeile `includeBuild('../node_modules/@react-native/gradle-plugin')` aufzunehmen. Weiterhin ist es erforderlich, weitere Zeilen für die Nutzung von *auto-linking* aufzunehmen. Der Name der Anwendung ist dabei entsprechend anzupassen. (siehe Quellcode 5.11)

```
1 rootProject.name = "example integration"
2
3 apply from: file("../node_modules/@react-native-community/cli-platform-andr
  ↪  oid/native_modules.gradle");
4 applyNativeModulesSettingsGradle(settings);
5
6 include ':app'
7 includeBuild('../node_modules/@react-native/gradle-plugin')
```

Quellcode 5.11: Inhalt der Datei `settings.gradle` mit den nötigen Anpassungen.

In der Datei `android/build.gradle` ist es ebenfalls erforderlich, eine Zeile hinzuzufügen, um das `react-native-gradle-plugin` zu integrieren. Weiterhin ist es notwendig, die *Gradle*-Abhängigkeit anzupassen, indem die Version entfernt wird. Die *Gradle* Version wird künftig von React Native festgelegt. (siehe Quellcode 5.12)

Abschließend besteht die Notwendigkeit, die Datei `android/app/build.gradle` anzupassen. Dabei ist es erforderlich, React als Plugin sowie Hermes (vgl. Kapitel 4) als Abhängigkeiten zu integrieren. Um *auto-linking* (vgl. Abschnitt 4.7) zu konfigurieren, sind am Ende der Datei zwei weitere Zeilen einzufügen. (siehe Quellcode 5.13)

```
buildscript {
    dependencies {
        ...
        classpath("com.android.tools.build:gradle")
        classpath("com.facebook.react:react-native-gradle-plugin")
    }
}
...
```

Quellcode 5.12: Inhalt der Datei `build.gradle` im Ordner `android` mit den nötigen Anpassungen.

```
plugins {
    id 'com.android.application'
    // React Native
    id 'com.facebook.react'
}

android {
   ...
}

dependencies {
    ...
    constraints {
        implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.8.20") {
            because("kotlin-stdlib-jdk7 is now a part of kotlin-stdlib")
        }
        implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.8.20") {
            because("kotlin-stdlib-jdk8 is now a part of kotlin-stdlib")
        }
    }
    // React Native
    implementation "com.facebook.react:react-android"
    implementation "com.facebook.react:hermes-android"
    ...
}

// React Native
apply from: file("../../node_modules/@react-native-community/cli-platform-a
↪  ndroid/native_modules.gradle")
applyNativeModulesAppBuildGradle(project)
```

Quellcode 5.13: Inhalt der Datei `build.gradle` im Ordner `android/app` mit den nötigen Anpassungen.

Sollte es zu diesem Zeitpunkt in Android Studio beim Synchronisieren der Gradle-Einstellungen zu einem Fehler kommen, bei dem mehrere Java-Klassen von Kotlin dupliziert sind, ist der `constraints {...}`-Part aufzunehmen. (siehe Quellcode 5.13) Sollte es zusätzlich zum Fehler `Cause: defaultConfig contains custom BuildConfig fields, but the feature is disabled.` kommen,

muss `android.defaults.buildfeatures.buildconfig=true` in der Datei `android/gradle.properties` aufgenommen werden.

5.5.2 Verbindung zu React Native

Ist der Build funktionstüchtig, sind Modifikationen am Code der Android App notwendig, um die Verbindung zwischen der Android App und dem *Hermes Bundler* herzustellen.

Zunächst sind dabei Anpassungen an der Manifest-Datei `AndroidManifest.xml` vorzunehmen. Da diese Modifikationen nur im Debug-Modus erforderlich sind, ist es ratsam, sie speziell für ebendiesen Modus zu implementieren. Eine Notwendigkeit für die Anpassungen an der Manifest-Datei rührt vorwiegend daher, dass React Native eine unverschlüsselte Verbindung mit *Metro* herstellt. Eine solche unverschlüsselte Verbindung muss in Android explizit freigeschaltet werden, da dies ein Sicherheitsrisiko darstellt. Für das Aktivieren dieser unverschlüsselten Verbindung ist `android:usesCleartextTraffic="true"` verantwortlich. Zusätzlich wird die *activity* des Debug-Menüs React Natives hinzugefügt, welche in einer Produktivumgebung nicht notwendig ist. Zuletzt benötigt die App die Berechtigung, mit dem Internet bzw. mit dem lokalen Dev-Server *Metro* zu kommunizieren. Obschon diese Berechtigung in der Regel bereits im Produktionsmanifest aufgeführt ist, wird sie sicherheitshalber auch im Debugmanifest ergänzt. Zum Verwenden eines separaten Debugmanifests, ist es notwendig, eine neue Datei namens `AndroidManifest.xml` unter `android/app/src/debug` anzulegen. Sollte der Ordner `debug` dabei nicht bereits existieren, ist dieser zu erstellen.[1] (siehe Quellcode 5.14) Beim Starten eines Build-Vorgangs werden beide Manifest-Dateien dabei zusammengeführt.

Für die Verbindung zum *Metro Bundler* ist es notwendig, die Klasse, welche `Application` erweitert[2], anzupassen. Sollte noch keine solche Klasse existieren, ist es erforderlich, eine solche Klasse anzulegen. Zusätzlich ist diese Klasse im `<application>`-Tag der Datei `AndroidManifest.xml` zu hinterlegen: `android:name=".MainApplication"`. Die Klasse muss nun um die Implementierung des Interfaces `ReactApplication` erweitert werden. Dazu ist es notwendig, die Methode `getReactNativeHost` zu definieren, welche eine zuvor erstellte Instanz des `ReactNativeHosts` zurückgibt. Die Instanz von `ReactNativeHost` wird dabei anhand der abstrakten Klasse `DefaultReactNativeHost` implementiert. Dafür wird die abstrakte Klasse instanziiert, indem das Java-Feature der *Anonymous Inner Class* angewandt wird.[3] Dabei ist es notwendig,

1 Sollte die Klasse, welche `Application` erweitert nicht `MainApplication` heißen, ist der Name der Klasse in der `AndroidManifest.xml` anzupassen.
2 In der Regel heißt diese Datei `MainApplication.java`
3 Dabei kann eine abstrakte Klasse initiiert werden, indem die Implementierung der abstrakten Klasse direkt hinter dem Konstruktor-Aufruf definiert wird.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:name=".MainApplication"
        android:usesCleartextTraffic="true"
        tools:node="merge"
        tools:targetApi="28">

        <activity
        ↪  android:name="com.facebook.react.devsupport.DevSettingsActivity"
        ↪  />
    </application>

</manifest>
```

Quellcode 5.14: Inhalt der Datei `AndroidManifest.xml` im Ordner `android/app/src/debug`.

innerhalb der *Anonymous Inner Class* die Methoden `getUseDeveloperSupport` sowie `getPackages` zu implementieren. `getUseDeveloperSupport` erfordert dabei einen `boolean` als Rückgabewert, welcher steuert, ob der Entwicklungsmodus aktiviert ist. Diese Information kann automatisch über die von React Native generierte `BuildConfig` ausgelesen werden. `getPackages` hingegen gibt eine Liste an React Native *Packages* zurück. Dabei werden zunächst die vom *auto-linking* gefundenen *Packages* bereitgestellt. (siehe Quellcode 5.15) Bei der Definition von selbst erstellten *Packages*, ist es notwendig, diese in der Liste der *packages* manuell zu ergänzen. Das Vorgehen kann dabei aus Abschnitt 4.2 bzw. Abschnitt 4.4 zur Erstellung eines nativen Moduls entnommen werden.

Da das Android-Betriebssystem eine native Rücknavigation über einen Zurück-Button oder eine vergleichbare Geste ermöglicht, ist es erforderlich, dass React Native diese Aktion an die native Android-App weiterleitet. Nur so kann die Rücknavigation in der nativen App verarbeitet werden, falls React Native diese Aktion außer Acht lässt. Dazu muss innerhalb der Android-Aktivität[1] das Interface `DefaultHardwareBackBtnHandler` von React Native bzw. dessen Methode `invokeDefaultOnBackPressed` implementiert werden. Dazu wird lediglich innerhalb der Methode `invokeDefaultOnBackPressed` die Methode der Aktivität namens `onBackPressed` aufgerufen. (siehe Quellcode 5.16)

1 In der Regel heißt diese Klasse `MainActivity`

```
public class MainApplication extends Application implements ReactApplication {

    @Override public void onCreate() {
        super.onCreate();
        SoLoader.init(this, false);
    }

    private final ReactNativeHost mReactNativeHost = new DefaultReactNativeHost(this) {
        @Override
        public boolean getUseDeveloperSupport() {
            return BuildConfig.DEBUG;
        }

        protected List<ReactPackage> getPackages() {
            List<ReactPackage> packages = new PackageList(this).getPackages();
            return packages;
        }
    };

    @Override public ReactNativeHost getReactNativeHost() {
        return mReactNativeHost;
    }
}
```

Quellcode 5.15: Inhalt der Datei `MainApplication.java` mit den nötigen Anpassungen.

```
public class MainActivity extends AppCompatActivity implements DefaultHardwareBackBtnHandler {
    ...
    @Override
    public void invokeDefaultOnBackPressed() {
        this.onBackPressed();
    }
}
```

Quellcode 5.16: Inhalt der Datei `MainActivity.java` mit den nötigen Anpassungen.

5.5.3 Aufruf einer React Native Komponente

Da die native App nun in der Lage ist, mit dem *Metro-Bundler* zu kommunizieren, besteht nun die Möglichkeit, eine von React Native exportierte Komponente zur Anzeige zu bringen. Unter Android ergeben sich in diesem Zusammenhang unterschiedliche Ansätze. So ist es möglich, eine neue Aktivität zur Anzeige einer Komponente zu verwenden. Für eine Integration in eine bestehende Aktivität ist es hingegen möglich, *Fragments* zu verwenden. Letzteres ermöglicht dabei zusätzlich die Integration von nicht

bildschirmfüllenden React Native Komponenten in bestehende Views. *Fragments* können dabei ebenfalls für bildschirmfüllende Komponenten verwendet werden, weswegen im Folgenden beispielhaft die Verwendung einer React Native Komponente innerhalb eines *Fragments* demonstriert wird, da diese flexibler einsetzbar sind als Aktivitäten.

5.5.3.1 Verwendung eines bildschirmfüllenden *Fragments*

Zunächst ist es notwendig, eine Layout-Datei anzulegen. Diese Layout-Datei beinhaltet dabei ein Frame-Layout, in das React Native alle UI-Elemente einer bildschirmfüllenden Komponente platziert. Der Name der Datei ist dabei prinzipiell frei wählbar, für dieses Beispiel wird diese als `react_native_view.xml` unter `android/res/layout` erstellt. (siehe Quellcode 5.17)

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/react_native_fragment"
4     android:layout_width="wrap_content"
5     android:layout_height="wrap_content" />
```

Quellcode 5.17: Inhalt der Datei `react_native_view.xml`.

Im Anschluss wird der Rückgabewert der `onCreateView`-Methode eines Fragments so modifiziert, dass eine React Native Komponente dargestellt wird. Dazu ist es notwendig, die Instanz des `LayoutInflaters` mit dem zuvor erstellte Layout zu befüllen. Anschließend wird der *Builder* des `ReactFragments` verwendet, um eine neue Instanz eines `ReactFragments` zu erzeugen. Dabei ist es zwingend notwendig, einen Komponentennamen anzugeben sowie Fabric zu deaktivieren bzw. zu aktivieren. Optional besteht die Möglichkeit zur Übergabe von *Props* an die React Komponente. In diesem Fall wird zufällig eine Startzahl zwischen 1 und 100 gewählt und der React Native Komponente zur Verfügung gestellt. Über den *FragmentManager* wird das erstellte *Fragment* anschließend zur Anzeige gebracht und zuletzt die durch den `LayoutInflater` erstellte *View* zurückgegeben. (siehe Quellcode 5.18)

5.5.3.2 Verwendung eines nicht bildschirmfüllenden *Fragments*

Eine weitere Möglichkeit der Integration besteht darin, statt einer eigenen Layout-Datei, eine bereits vorhandene Layout-Datei zu verwenden. Dabei ist es notwendig, ein *FrameLayout* innerhalb der Layout-Datei hinzuzufügen und dieses entsprechend dem Anwendungsfall zu positionieren. Beispielhaft wird im Folgenden die React Native Counter Komponente unter nativen Eingabeelementen eines bestehenden Fragments innerhalb eines *ConstraintLayouts* hinzugefügt.

```java
public class FullscreenCounterFragment extends Fragment {

    private ReactFragment reactNativeFragment;

    public View onCreateView(@NonNull LayoutInflater inflater,
                             ViewGroup container, Bundle
                             ↪ savedInstanceState) {
        if(reactNativeFragment != null) {
            // needs to be destroyed if already displayed or an exception
            ↪ would be thrown
            reactNativeFragment.onDestroy();
        }
        View view = inflater.inflate(R.layout.react_native_view, container,
        ↪ false);
        Bundle options = new Bundle();
        options.putInt("initialCount", new Random().nextInt(100) + 1);
        if (reactNativeFragment == null) {
            reactNativeFragment = new ReactFragment.Builder()
                    .setComponentName("Counter")
                    .setLaunchOptions(options)
                    .setFabricEnabled(false)
                    .build();
            getChildFragmentManager().beginTransaction().add(R.id.react_nat
            ↪ ive_fragment, reactNativeFragment).commit();
        }
        return view;
    }
}
```

Quellcode 5.18: Inhalt eines Fragments `FullscreenCounterFragment`, in dem eine bildschirmfüllende React Native Komponente angezeigt werden soll.

Die React Native Komponente wird dabei indes geladen, wenn ein Button auf der nativen Maske betätigt wird. Zusätzlich existieren zwei weitere Buttons, welche den Wert der React Native Counter Komponente entweder erhöhen oder verringern.

Dazu ist es notwendig, ein neues *FrameLayout* in der Layout-Datei des Fragments hinzuzufügen.[1] Als `id` wird dabei `react_native_counter` festgelegt. (siehe Quellcode 5.19)

Daraufhin wird die entsprechende Logik in der Fragment-Klasse implementiert. Der Aufruf der React Native Komponente ist dabei prinzipiell ähnlich zum beschriebenen Vorgehen aus Abschnitt 5.5.3.1, nur wird in der Methode `onCreateView` keine neue *View* erstellt, sondern die bereits erstellte View verwendet. Zusätzlich wird im Aufruf der *FragmentManagers* die ID des zuvor definierten *FrameLayouts* (siehe Quellcode 5.19)

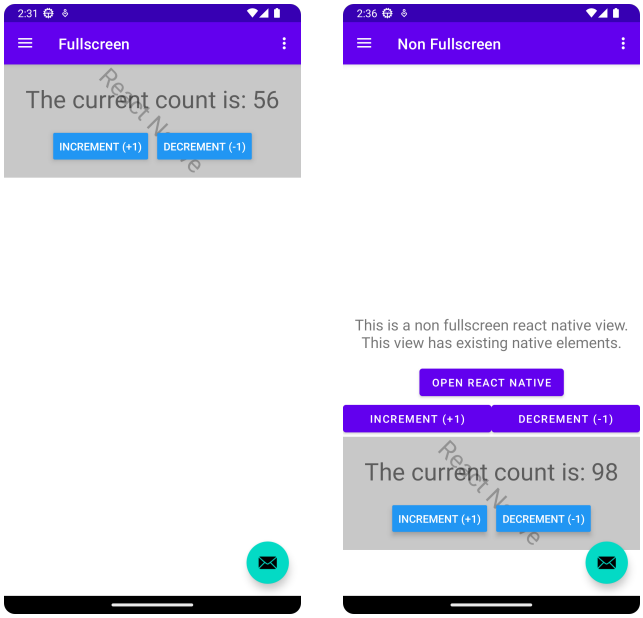
1 Die Definition von `layout_width` bzw. `layout_height` von `0dp` erscheint ohne das Wissen über das `ConstraintLayout` zunächst kontraintuitiv. Das Layout ist dabei nicht `0dp` breit und hoch, sondern passt sich aufgrund der Angabe von `0dp` automatisch den im `ConstraintLayout` definierten Einschränkungen an. [77]

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout ...>

    <FrameLayout
        android:id="@+id/react_native_counter"
        android:layout_width="0dp"
        android:layout_height="0dp"
        ... />

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

Quellcode 5.19: Beispielhaftes Frame-Layout in einer Layout-Datei eines Fragments.

der wiederverwendeten Layout-Datei verwendet. Beim Klick der nativen *increment*- und *decrement*-Buttons wird dabei ein interner Zähler inkrementiert bzw. dekrementiert. Zugleich wird bei jeder Änderung dieses internen Zählers die React Native Komponente mit dem veränderten Zähler neu aufgerufen. Dies hat zur Folge, dass das Betätigen der nativen Buttons auch eine Veränderung der Anzeige der React Native Komponente zur Folge hat. (siehe Quellcode 5.20)

Die Integration ist nun erfolgreich abgeschlossen. Das Ergebnis der beispielhaften Integration ist dabei in Abbildung 5.5 zu sehen.



(a) Bildschirmfüllenende Integration durch eigene Layout-Datei.

(b) Nicht bildschirmfüllende Integration durch Verwenden eines *FrameLayouts* in einer bestehenden Layout-Datei.

Abbildung 5.5: Bildschirmfüllende (vgl. Abbildung 5.5a) und nicht bildschirmfüllende (vgl. Abbildung 5.5b) Integration von React Native in eine bestehende Android-App.

```
public class NonFullscreenCounterFragment extends Fragment {
    ...
    private int count = new Random().nextInt(100) + 1;

    public View onCreateView(@NonNull LayoutInflater inflater,
                             ViewGroup container, Bundle savedInstanceState)
                             ↪ {
        ...

        Button button = root.findViewById(R.id.openReactNative);
        button.setOnClickListener(v -> createReactNativeView());

        root.findViewById(R.id.increment).setOnClickListener(v -> {
            count++;
            createReactNativeView();
        });
        root.findViewById(R.id.decrement).setOnClickListener(v -> {
            count--;
            createReactNativeView();
        });

        return root;
    }

    private void createReactNativeView() {
        Bundle options = new Bundle();
        options.putInt("initialCount", count);
        Fragment reactNativeFragment = new ReactFragment.Builder()
                .setComponentName("Counter")
                .setLaunchOptions(options)
                .setFabricEnabled(false)
                .build();
        getChildFragmentManager()
            .beginTransaction()
            .add(R.id.react_native_counter, reactNativeFragment)
            .commit();
    }
    ...
}
```

Quellcode 5.20: Inhalt eines Fragments `NonFullscreenCounterFragment`, in dem eine nicht bildschirmfüllende React Native Komponente angezeigt werden soll.

Durch diese Art der Integration ist es möglich, einen veränderten Zustand der nativen App durch ein erneutes Aufrufen der React Native Komponente widerzuspiegeln. Dabei tritt jedoch schnell ein Problem auf: Wird der Zustand in der nativen App verändert, wird zugleich der Zustand der React Native Komponente durch einen erneuten Aufruf aktualisiert. Somit funktioniert die Synchronisation der Daten in dieser Kommunikationsrichtung problemlos. Werden hingegen die Buttons der React Native Komponente betätigt, wird zwar der Zustand der React Native Komponente verändert, die native App erhält jedoch keine Kenntnis über diese Zustandsänderung. Somit ist es möglich, dass in diesem Szenario die Zustände der nativen App und der Zustand der React Native

Komponente inkonsistent werden. In diesem kleinen Beispiel scheint die Lösung für das Problem recht trivial zu sein, dennoch ist bei größeren Anwendungen mit globalen Zuständen – wie z. B. Datenbanken – eine Synchronisation der Zustände weniger offensichtlich. Somit empfiehlt es sich, einen Synchronisationsmechanismus zu entwickeln, um inkonsistente Zustände der nativen Apps und der React Native Komponente zu vermeiden. Drei mögliche Lösungen sowie deren Vor- und Nachteile werden in Kapitel 6 behandelt.

5.6 Aktivieren der Fabric Render Engine

Damit die *Fabric* Render Engine verwendet werden kann, ist es zunächst notwendig, die neue Architektur innerhalb der *Brownfield*-Integration zu aktivieren. Dazu sind zunächst die Schritte aus Abschnitt 4.4.1 durchzuführen. Weiterhin ist es notwendig, weitere Anpassungen an den nativen Apps vorzunehmen. Die notwendigen Schritte finden dabei jedoch kaum Erwähnung innerhalb der offiziellen Dokumentation. Angesichts dessen wurden die durchzuführenden Schritte aus dem offiziellen React Native Repository und der darin enthaltenden Beispiele inferiert. Weiterhin sollte vor dem Wechsel auf die neue Architektur geprüft werden, ob alle verwendeten React Native Community Bibliotheken ebendiese unterstützen. Alternativ ist es möglich Komponenten, welche von deren Autorinnen und Autoren bislang nicht migriert wurden, ab der Version `0.72` mit einem automatisiertem Konvertierungsmechanismus mit der neuen Architektur zu verwenden. (vgl. Abschnitt 4.6) [68]

Die vollständige Implementierung der Integration inklusive aktivierter neuen Architektur sowie Fabric ist im Git-Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) unter dem *Branch* `react-native-integration-fabric` verfügbar.

5.6.1 Anpassungen der iOS App

Zum Veröffentlichungszeitpunkt dieser Ausarbeitung besteht noch keine Möglichkeit, Fabric mit Swift zu initialisieren. Deswegen ist es notwendig, die Klasse `AppDelegate` zwingend mit Objective-C/Objective-C++ zu implementieren. [78] Sollte dies nicht der Fall sein, muss diese Datei migriert werden, wenn *Fabric* aktiviert werden soll. Dafür empfiehlt es sich, ein neues iOS Projekt anzulegen, welches mit Objective-C initialisiert wird. Alle anderen Klassen können dabei dennoch weiterhin mit Swift implementiert werden.

Wie auch in Abschnitt 4.4.1.1 beschrieben, müssen zunächst die korrekten *Pods* installiert werden. Dafür muss das Kommando `USE_FABRIC=1 RCT_NEW_ARCH_ENABLED=1 pod install` im `ios`-Ordner ausgeführt werden.

Dabei sind für die Aktivierung von *Fabric* Anpassungen an der `AppDelegate.m` notwendig. Dafür muss der Typ der Datei von *Objective-C Source* auf *Objective-C++ Source* geändert werden. Anschließend ist die passende Header-Datei `AppDelegate.h` um öffentliche *Properties* zu erweitern, welche anschließend in der `AppDelegate.m` initialisiert werden. (siehe Quellcode 5.21)

```
1 @class RCTBridge;
2
3 @interface AppDelegate : UIResponder <UIApplicationDelegate>
4
5 @property (nonatomic, strong) UIWindow *window;
6 @property (nonatomic, readonly) RCTBridge *bridge;
7
8 @end
9
```

Quellcode 5.21: Anpassungen an der Header-Datei `AppDelegate.h`.

Weiterhin sind umfangreiche Änderungen der Datei `AppDelegate.m` notwendig. Zunächst sind *Imports* zu ergänzen, welche für die Verwendung von Fabric notwendig sind. Anschließend muss die Interface-Definition von `AppDelegate` `RCTCxxBridgeDelegate` sowie `RCTTurboModuleManagerDelegate` erweitern. Ebenso wird das Interface um die Variablen `_runtimeScheduler`, `_bridgeAdapter`, `_reactNativeConfig` sowie `_contextContainer` ergänzt. Diese werden dabei in der Implementierung von `AppDelegate` initialisiert. Daraufhin ist es notwendig, die Funktion `application` der `AppDelegate` Implementierung zu erweitern. In dieser wird zunächst die *Bridge* initialisiert. Die passende URL für den Metro Bundler oder das Produktions-Bundle liefert dabei die Funktion `sourceURLForBridge`. Die Signatur der Methode wird dabei von `RCTTurboModuleManagerDelegate` vorgegeben. Nun wird, sofern Fabric aktiviert wurde, der `surfacePresenter` innerhalb der *Bridge* ausgetauscht. Dies ermöglicht das Anzeigen einer React Native Komponente mit der Render Engine *Fabric*. (siehe Quellcode 5.22 und 5.23) [79]

Weiterhin ist es erforderlich, die Klassen, welche `UIViewController` erweitern und eine React Native Komponente anzeigen, anzupassen, damit diese *Fabric* verwenden können. Dabei ist es möglich, die *UIViewController* weiterhin mit Swift zu implementieren. Statt der Klasse `RCTRootView` ist es notwendig, `RCTFabricSurfaceHoistingProxyRootView` für die Verwendung der *Fabric* Render Engine zu verwenden. Die Signatur des aufzurufenden Konstruktors bleibt dabei größtenteils identisch. Der Zu-

```
...
@interface AppDelegate () <RCTCxxBridgeDelegate,
↪  RCTTurboModuleManagerDelegate> {
  RCTSurfacePresenterBridgeAdapter *_bridgeAdapter;
  std::shared_ptr<const facebook::react::ReactNativeConfig>
  ↪  _reactNativeConfig;
  facebook::react::ContextContainer::Shared _contextContainer;
  RCTTurboModuleManager *_turboModuleManager;
}
@end
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
↪  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    _bridge = [[RCTBridge alloc] initWithDelegate:self
    ↪  launchOptions:launchOptions];
    _contextContainer = std::make_shared<facebook::react::ContextContainer
    ↪  const>();
    _reactNativeConfig =
    ↪  std::make_shared<facebook::react::EmptyReactNativeConfig const>();
    _contextContainer->insert("ReactNativeConfig", _reactNativeConfig);
    _bridgeAdapter = [[RCTSurfacePresenterBridgeAdapter alloc]
    ↪  initWithBridge:_bridge contextContainer:_contextContainer];
    _bridge.surfacePresenter = _bridgeAdapter.surfacePresenter;
    return YES;
}
...
- (NSURL *)sourceURLForBridge:(RCTBridge *)bridge
{
#if DEBUG
    return [NSURL
    ↪  URLWithString:@"http://localhost:8081/index.bundle?platform=ios"];
#else
    return [[NSBundle mainBundle] URLForResource:@"main"
    ↪  withExtension:@"jsbundle"];
#endif
}
...
```

Quellcode 5.22: Anpassungen an der Datei `AppDelegate.m` der Funktion `application` und `sourceURLForBridge`.

griff auf die *Bridge* ändert sich dennoch geringfügig, aufgrund der Verwendung von Objective-C innerhalb der `AppDelegate` Klasse. (siehe Quellcode 5.24) [79]

Damit Swift Klassen weiterhin auf die erstellte *Bridge* zugreifen können, ist es notwendig `AppDelegate.h` in den *Bridging Header* aufzunehmen. Die Aufnahme von `RCTFabricSurfaceHoistingProxyRootView` für die Verwendung in Swift Klassen im *Bridging Header* ist dabei ebenfalls erforderlich. (siehe Quellcode 5.25)

```
...
- (std::unique_ptr<facebook::react::JSExecutorFactory>)jsExecutorFactoryForBridge:(RCTBridge
  *)bridge
{
  _turboModuleManager = [[RCTTurboModuleManager alloc]
    initWithBridge:bridge delegate:self jsInvoker:bridge.jsCallInvoker];
  [bridge setRCTTurboModuleRegistry:_turboModuleManager];
#if RCT_DEV
  [_turboModuleManager moduleForName:"RCTDevMenu"];
#endif
  __weak __typeof(self) weakSelf = self;
#if RCT_USE_HERMES
  return std::make_unique<facebook::react::HermesExecutorFactory>(
      facebook::react::RCTJSIExecutorRuntimeInstaller([weakSelf,
        bridge](facebook::jsi::Runtime &runtime) {
        if (!bridge) {
          return;
        }
        __typeof(self) strongSelf = weakSelf;
        if (strongSelf) {
          facebook::react::RuntimeExecutor syncRuntimeExecutor =
              [&](std::function<void(facebook::jsi::Runtime & runtime_)>
                &&callback) { callback(runtime); };
          [strongSelf->_turboModuleManager
            installJSBindingWithRuntimeExecutor:syncRuntimeExecutor];
        }
      }));
}
...
```

Quellcode 5.23: Anpassungen an der Datei `AppDelegate.m` der Funktion `jsExecutorFactoryForBridge`.

```
class ExampleViewController: UIViewController {
    override func viewDidLoad() {
        ...
        let appDelegate = UIApplication.shared.delegate as! AppDelegate
        self.view = RCTFabricSurfaceHostingProxyRootView(bridge:
          appDelegate.bridge!, moduleName: "Counter", initialProperties:
          ["initialCount": MockDB.instance().setRandomCounter()])
    }
}
```

Quellcode 5.24: Anpassungen an einem `UIViewController`.

```
#import "AppDelegate.h"
#import <react/RCTFabricSurfaceHostingProxyRootView.h>
...
```

Quellcode 5.25: Anpassungen an der Datei `*-Bridging-Header.h`.

5.6.2 Anpassungen der Android App

Unter Android ist es notwendig, Anpassungen an der Implementierung des `DefaultReactNativeHosts` in der Klasse, welche `ReactApplication` implementiert durchzuführen.[1] Dabei ist es erforderlich, die Methode `getJSIModulePackage` zu überschreiben, da dies sonst zum Fehler bei der Aktivierung von *Fabric* führt. (siehe Quellcode 5.26) Leider bietet die Dokumentation von React Native wenig Informationen über die genaue Funktionsweise der Methode `getJSIModulePackage`. [80]

Anschließend ist es möglich, innerhalb des `ReactFragment.Builders` den Methodenaufruf `.setFabricEnabled(false)` in `.setFabricEnabled(true)` zu verändern. Auf diese Weise können einzelne Komponenten wahlweise mit der *Fabric* Render Engine betrieben werden.

1 Zur Erinnerung: Diese heißt in der Regel `MainApplication`

```
1  public class MainApplication extends Application implements
   ↪  ReactApplication {
2      ...
3      private final ReactNativeHost mReactNativeHost = new
       ↪  DefaultReactNativeHost(this) {
4          ...
5          @NonNull
6          @Override
7          protected JSIModulePackage getJSIModulePackage() {
8              return (reactApplicationContext, jsContext) -> {
9                  final List<JSIModuleSpec> specs = new ArrayList<>();
10                 specs.add(new JSIModuleSpec() {
11                     @Override
12                     public JSIModuleType getJSIModuleType() {
13                         return JSIModuleType.UIManager;
14                     }
15
16                     @Override
17                     public JSIModuleProvider<UIManager>
                       ↪  getJSIModuleProvider() {
18                         final ComponentFactory componentFactory = new
                           ↪  ComponentFactory();
19                         CoreComponentsRegistry.register(componentFactory);
20                         final ReactInstanceManager reactInstanceManager =
                           ↪  getReactInstanceManager();
21
22                         ViewManagerRegistry viewManagerRegistry =
23                                 new ViewManagerRegistry(
24                                         reactInstanceManager.getOrCreateVie
                                           ↪  wManagers(
25                                                 reactApplicationContext));
26
27                         return new FabricJSIModuleProvider(
28                                 reactApplicationContext,
29                                 componentFactory,
30                                 ReactNativeConfig.DEFAULT_CONFIG,
31                                 viewManagerRegistry);
32                     }
33                 });
34                 return specs;
35             };
36         }
37     };
38     ...
39 }
```

Quellcode 5.26: Anpassungen an der Implementierung des `DefaultReactNativeHosts`. [80]

6 Datensynchronität zwischen React Native und nativen Apps

Werden in den bestehenden iOS und Android Apps Offline-Daten benötigt, welche ebenfalls innerhalb einer React Native Komponente zum Einsatz kommen sollen, ist es notwendig, einen Datenaustausch zwischen den beiden Softwarekomponenten sicherzustellen. Erhält die native App aktualisierte Daten z. B. von einem Server oder verändert diese selbst, ist es erforderlich, diese ebenfalls in der React Native Komponente zu aktualisieren oder eine aktualisierte Referenz auf diese bereitzustellen. Werden die Daten wiederum innerhalb von React Native angepasst, müssen diese ebenfalls synchronisiert werden. Im Zähler Beispiel aus Kapitel 5 wurde lediglich eine unidirektionale Aktualisierung des Zählerwertes durchgeführt; dabei wurde die React Native Komponente mit aktualisierten Werten der nativen App neu aufgerufen. Abgesehen von der fehlenden bidirektionalen Kommunikation weist dieser Ansatz zusätzlich den Nachteil auf, dass die React Native Komponente vollständig neu erstellt werden muss, sobald sich ein Wert ändert. Dies hat zur Folge, dass Zustände, welche lokal innerhalb der React Native Komponente vorgehalten werden, beim erneuten Aufrufen der Komponente verloren gehen und Optimierungen von React für das *Rerendering* nicht zum Tragen kommen. (vgl. Abbildung 6.1)

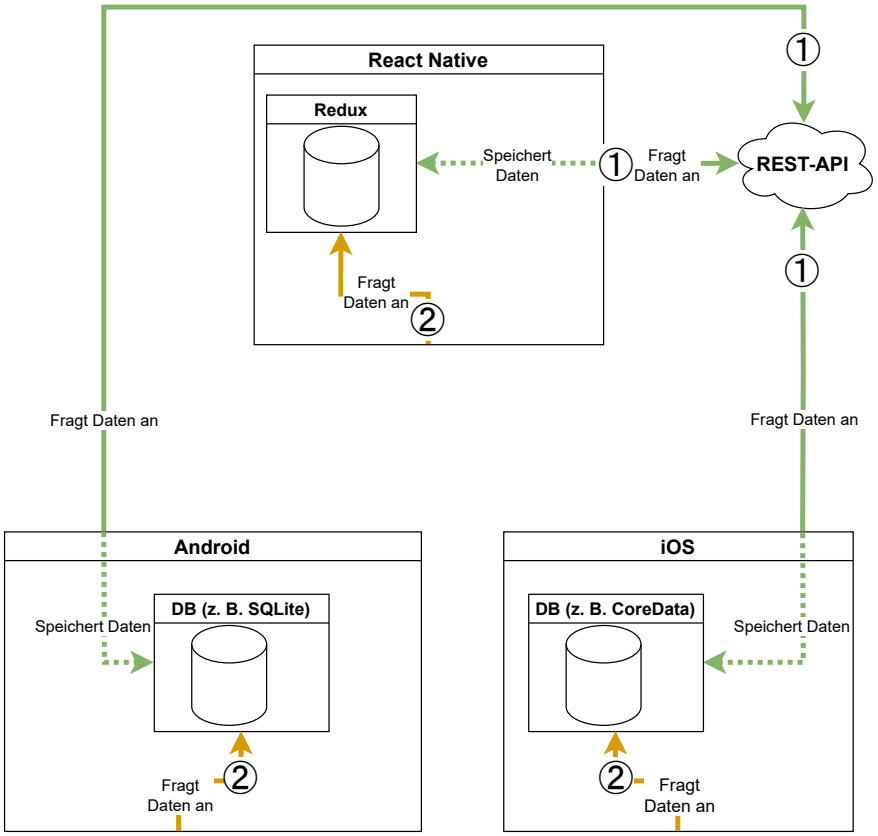


Abbildung 6.1: Schematisch vereinfachter Aufbau der nativen Apps und React Native inklusive der dazugehörigen Offline-Datenbanken ohne Synchronisation.

Grüne Pfeile: Daten werden von der REST-API abgefragt, sofern eine Verbindung zum entsprechenden Server besteht und in der lokalen Datenbank gespeichert.

Orange Pfeile: Daten werden aus *Redux* oder den nativen lokalen Datenbanken ausgelesen, sofern keine Verbindung zum Server der REST API aufgebaut werden kann.

①: Daten werden im Online-Szenario von der nativen iOS bzw. Android App oder React Native von einem Server angefragt und das Ergebnis in der jeweiligen lokalen Datenbanken gespeichert. Ohne explizite Synchronisation der Offline-Datenbanken ist dies die einzige Möglichkeit für die beiden Softwarekomponenten einen konsistenten Datenbestand aufrechtzuerhalten.

②: Daten werden im Offline-Szenario aus den Offlinedatenbanken ausgelesen. Hat z. B. React Native kurz bevor die Verbindung zum Server abgebrochen ist, die aktuellsten Daten abgerufen und gespeichert, die native App jedoch nicht, hat die native App in diesem Fall keine Möglichkeit, auf diese aktualisierten Daten zuzugreifen.

Mitunter aus diesem Grund, ist es sinnvoller, eine Kommunikationsschnittstelle zu verwenden, um lokale Werte der nativen Apps mit React Native zu synchronisieren. Insbesondere die Standardkommunikationsschnittstelle – native Module (vgl. Abschnitt 4.2) – eignen sich für diesen Anwendungsfall. Native Module erlauben dabei beim Aufruf aus React Native eine bidirektionale Kommunikation unter Zuhilfenahme von *Callback*s oder *Promise*s. Beim Aufruf aus der nativen App kann die bidirektionale Kommunikation mittels *Event Emittern* in Verbindung mit nativen Modulen umgesetzt werden.

In den iOS und Android Apps kommt eine beispielhafte Datenbank zum Einsatz, welche `MockDB` genannt wird. Diese Datenbank ist eine Klasse, die den Zählerwert des Beispiels aus Kapitel 5 speichert. Eine persistente Speicherung ist nicht umgesetzt, der Wert wird lediglich zur Laufzeit gespeichert. Die Zugriffe auf die `MockDB` sind dabei als symbolische Datenbankzugriffe zu verstehen, da so keine konkrete Datenbank vorgegeben wird. Folglich ist es in einem produktiven Umfeld notwendig, die Zugriffe auf die `MockDB` durch Zugriffe auf die bestehenden Datenbanken der nativen Apps auszutauschen. Apps, welche offlinefähig sind, verwenden in der Regel Datenbanken, um bestimmte Daten vorzuhalten. Damit React Native ebenfalls den Offlinebetrieb gewährleisten kann, wird innerhalb von React Native in diesem konkreten Beispiel *Redux* als State-Management-Bibliothek zum Speichern des Applikationszustandes verwendet. *Redux* kommt dabei häufig in Verbindung mit React zum Einsatz, wie die Fallstudie von Airbnb (vgl. Abschnitt 2.1) wie auch die Fachliteratur zeigt. [21, 22] Eine solcher Ansatz ist folglich möglichst repräsentativ.

Somit ergibt sich effektiv ein Szenario, in welchem zwei lokale Datenbanken miteinander synchronisiert werden müssen. Die Synchronisation von Datenbanken zeichnet sich dabei dadurch aus, die Konsistenz mehrerer Kopien von Datensätzen zu gewährleisten. Die Art der Synchronisation von Datenbanken auf einem Gerät unterscheidet sich dabei jedoch grundlegend von den in der Fachliteratur behandelten Datenbanksynchronisationen. So besteht bei der Synchronisation beider offline Datenbanken keine Notwendigkeit zur Beachtung eines Mehrbenutzerbetriebs, da immer nur eine Datenbankverbindung besteht.[1] Darüber hinaus ist es ebenfalls nicht notwendig, einen möglichen Verbindungsabbruch der Datenbank zu behandeln, da diese auf dem Gerät selbst verfügbar ist. Weiterhin bedarf es keiner komplexen Konfliktbehandlung von Daten. [82] Der Grund hierfür ist, dass stets der aktuellste Wert im Offline-Szenario gewählt werden muss. Dabei muss jedoch sichergestellt sein, dass der verwendete Server bereits eine Konfliktbehandlung beim Aufruf der entsprechenden Schnittstelle durchführt und diese innerhalb der nativen App bereits behandelt wird. Folglich besteht die notwendige Synchronisation lediglich aus dem Austausch der vorliegenden Daten der jeweiligen offline Datenbank.

Im Folgenden werden drei konkrete Ansätze, deren Vor- und Nachteile sowie mögliche Anwendungsszenarien herausgearbeitet.

1 Z. B. SQLite, welches unter Android häufig zum Einsatz kommt, erlaubt zwar einen Mehrbenutzerbetrieb, jedoch wird die gesamte Datenbank gesperrt, wenn ein Benutzer schreibend auf die Datenbank zugreift, um die Konsistenz der Datenbank zu gewährleisten. [81]

6.1 Spiegelung des Redux States ⓐ

Der erste Ansatz verfolgt das Ziel, lediglich eine Datenquelle für beide Applikationsteile zu verwenden. Auf diesen Ansatz wird im weiteren Verlauf als Ansatz ⓐ verwiesen. Die Verwaltung der Daten soll dabei ausschließlich in der React Native Anwendung erfolgen. Jegliche Änderungen an diesem Zustand werden dabei der nativen App als lesbare Kopie zur Verfügung gestellt. Dafür wird eine Methode innerhalb eines nativen Moduls verwendet, welche bei jeder Zustandsänderungen den neuen Zustand in die native App spiegelt. Folglich haben beide Programmteile Zugriff auf die Daten, ohne, dass die nativen Apps explizit über *Event Emitter* bestimmte Daten aus dem Zustand von React Native anfragen müssen. Da Daten innerhalb von *Redux* als `JSON` gespeichert werden, muss notwendigerweise eine Konvertierung für die nativen Apps stattfinden. Erfreulicherweise führt diese Konvertierung React Native bei der Verwendung eines nativen Moduls automatisch durch. Unter iOS sind die möglicherweise tief verschachtelte *JSON*-Strukturen so innerhalb eines `NSDictionary` zugreifbar. Unter Android hingegen werden diese innerhalb einer `ReadableMap`[1] abgebildet. Für das Aktualisieren des Zustandes aus der nativen App ist es dabei zusätzlich notwendig, ein *Event Emitter* zu definieren, welcher es erlaubt, eine *Redux Action*[2] mit einem optionalen *Payload*[3] auszuführen. Da jede Zustandsänderung zu einer Aktualisierung der Kopie der nativen App führt, ist die Kopie nach einer Zustandsänderung über einen solchen *Event Emitter* stets implizit aktuell. Dabei besteht zum einen die Möglichkeit, jede *Action* als separaten *Event Emitter* umzusetzen oder zum anderen alle *Action*s in einem *Event Emitter* zu bündeln. Ersteres ist dabei zwar aufwendiger, erlaubt jedoch das typsichere Verwenden der *Action*s und der dazugehörigen *Payload*. Letzteres hingegen ist deutlich einfacher zu implementieren, besitzt dafür jedoch keinerlei Typisierung. Folglich kommt es bei dem nicht typisierten Ansatz bei falscher Verwendung des *Event Emitter*s zur Laufzeit zum Fehler, während beim typisierten Ansatz der Fehler bereits zur Kompilierzeit auftritt. (vgl. Abbildung 6.2)

1 Diese Datenstruktur ist von React Native, ähnelt dabei jedoch einer klassischen Map, dessen *Key* ein `String` ist und dessen *Value* ein Wert aus Tabelle 4.2 (außer `Promises` und `Callbacks`) zugeordnet wird.

2 Eine solche Aktion führt eine Änderung am *Redux State* durch, indem die Aktion die Ausführung einer Folge von Instruktionen innerhalb eines *Reducers* instruiert. (vgl. `https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#actions`, aufgerufen am 26. Oktober 2023)

3 Die *Payload* einer Aktion spiegeln in der Regel die aktualisierten Daten wider, welche innerhalb des *Reducers* in den *Redux State* integriert werden.

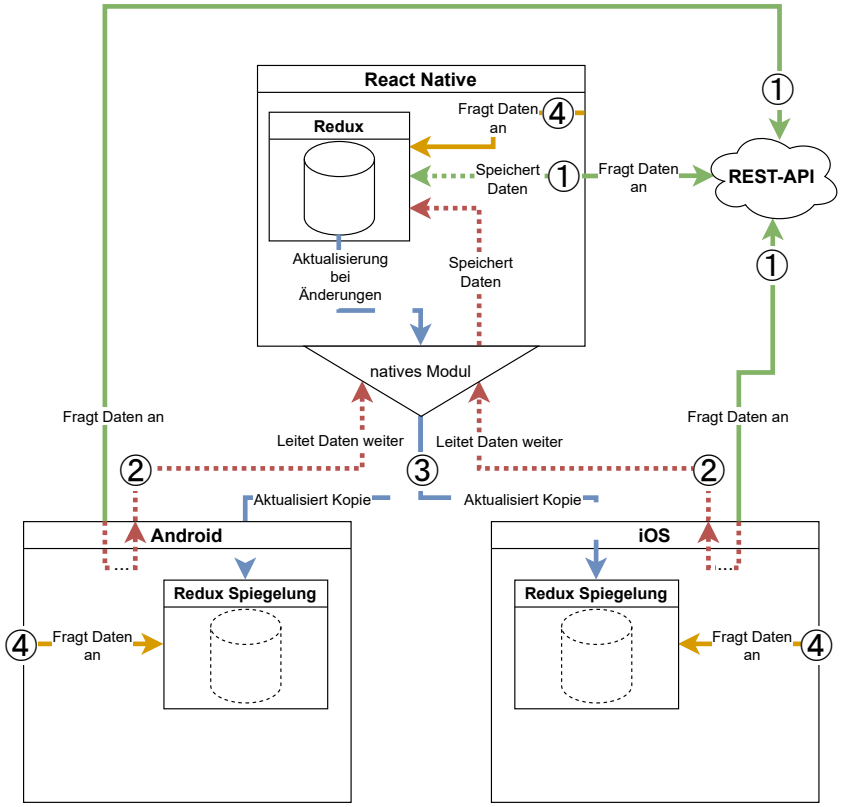


Abbildung 6.2: Schematisch vereinfachter Aufbau von Ansatz ⓐ.

Grüne Pfeile: Daten werden von der REST-API abgefragt, sofern eine Verbindung zum entsprechenden Server besteht.

Orange Pfeile: Daten werden aus *Redux* oder der Spiegelung ausgelesen, sofern keine Verbindung zum Server der REST API aufgebaut werden kann.

Rote (gepunktete) Pfeile: Abgefragte Daten der REST-API werden an ein natives Modul weitergeleitet und anschließend im *Redux* Zustand gespeichert. Durch die Zustandsänderung wird automatisch die Spiegelung der nativen App aktualisiert.

Blaue (gestrichelte) Pfeile: Aktualisierung der *Redux* Spiegelung bei Änderungen am *Redux* Zustand von React Native.

①: Daten werden im Online-Szenario von der nativen iOS bzw. Android App oder React Native von einem Server angefragt. Wurden die Daten von React Native angefragt, wird das Ergebnis direkt in dem lokalen *Redux-State* abgespeichert.

②: Wurden die Daten von einer der nativen Apps angefragt, werden die Daten über ein natives Modul (vgl. Abschnitt 4.2) an React Native weitergeleitet und im *Redux-State* abgespeichert.

③: Werden Änderungen am *Redux-State* durchgeführt, z. B. durch angefragte Daten vom Server, wird über ein natives Modul (vgl. Abschnitt 4.2) die Spiegelung des *Redux-States* aktualisiert.

④: Im Offline-Szenario erhält die native App die Daten aus der lokale *Redux* Spiegelung, während React Native die Daten direkt aus dem *Redux-State* auslesen kann.

Für eine konkrete Umsetzung bedeutet dies zunächst, dass die gesamte existierende Speicherlogik der nativen Apps migriert werden muss. Dabei gilt es zu beachten, dass die nativen Apps in der Regel relationale oder objektorientierte Organisationsmuster

verwenden, während innerhalb React Natives – nach der obigen Definition – ein dokumentenorientierter Ansatz verfolgt wird. Je nach Tiefe und Komplexität der Relationen innerhalb einer relationalen Datenbank, sind dabei ebenso komplexe Migrationsschritte notwendig, da innerhalb eines dokumentenorientierten Ansatzes keine klassischen Relationen existieren[1]. Weiterhin ist es notwendig zunächst zu prüfen, ob Datenbankzugriffe innerhalb der nativen Apps zentralisiert sind, oder ob an sehr vielen verschiedenen Stellen im Code Datenbankzugriffe orchestriert werden. Sollten die Datenbankzugriffe dabei vereinheitlicht sein, ist das Austauschen der Speicher-/Abfragelogik weniger aufwendig als wenn direkte Datenbankzugriffe über die gesamte Anwendung verteilt sind.

Innerhalb von React Native Komponenten kann bei erfolgreicher Migration nun ohne zusätzliche Anpassungen auf dem *Redux* Zustand operiert werden. Innerhalb der nativen Apps hingegen müssen lesende Zugriffe fortan auf den gespiegelten *Redux* Zustand erfolgen, der stets durch das native Modul aktualisiert wird. Ebenfalls sind alle schreibenden Zugriffe anzupassen. Dafür ist es möglich, ein *Event* namens `triggerAction` für den nicht typisierten Ansatz zu definieren. `triggerAction` nimmt den Namen einer *Redux Action* und ein optionales *Payload* entgegen, je nachdem, ob dieses von der *Redux Action* benötigt wird. Anschließend ist es notwendig, alle schreibenden Zugriffe durch den Aufruf dieses *Events* zu ersetzen. Zusätzlich ist es erforderlich, React Native parallel zum Start der nativen App zu starten, damit ein direkter Zugriff auf den Zustand der App möglich ist, selbst wenn noch keine React Native Komponente zu diesem Zeitpunkt angezeigt werden soll. Dafür eignet sich eine leere React Komponente, welche `null` zurückgibt und folglich nichts anzeigt. Die Komponente ist lediglich dafür zuständig, bei Änderungen am Zustand der Anwendung eine Aktualisierung der Spiegelung in der nativen App durchzuführen.

Aufgrund der kontinuierlichen Aktualisierung der Zustandskopie innerhalb der nativen Apps besteht Grund zur Annahme, dass dieser Ansatz einen degradierenden Einfluss auf die Performance der gesamten Anwendung ausübt. Daher wurden unter iOS und Android ein Profiling durchgeführt. Um eine möglichst realitätsnahes und repräsentatives Szenario zu gewährleisten, wurde der *Redux State* zu Beginn des Profilings mit einer umfangreichen Datenmenge befüllt. Dafür wurden beispielhaft Auswertungen der NASA über Meteoritenlandungen auf der Erde gewählt, da dieses JSON-Objekt eine hinreichend große Datenmenge von ca. 250 KB aufweist. [83] Zunächst wurde der Test mit einem JSON-Objekt ausgeführt, anschließend mit 2, 4, 8 und 16. Sollte die Spiegelung des *Redux* Zustandes somit ein lineares Wachstum aufweisen, müssten die Werte der Auswertung ebenfalls mit einem Faktor von 2 wachsen. Während der Messungen des Profilings wurde in einem Intervall von 500 ms Änderungen am *Redux State* vorgenommen. Anschließend

1 Natürlich ist es dennoch möglich, Relationen abzubilden, jedoch gehen dabei Funktionen wie z. B. das kaskadierende Löschen verloren. Solche Funktionen müssten manuell implementiert werden.

wurde die maximale CPU-Auslastung (in Prozent) zum Zeitpunkt der Modifikation sowie die aktuelle Arbeitsspeicherauslastung (in MB) abgelesen. Dieses Verfahren wurde zehnmal durchgeführt und alle Werte wurden arithmetisch gemittelt. Da unter iOS das Abrufen der Daten nicht über ein *Command-Line-Interface* oder *Script* durchgeführt werden kann, besteht das Risiko von Ungenauigkeiten durch das manuelle Ablesen der Daten.

Die Auswertung unter iOS zeigt einen moderaten Anstieg der Arbeitsspeichernutzung sowie einen rasanten Anstieg der CPU-Auslastung. Dabei ist zu beobachten, dass die Arbeitsspeicherauslastung ohne die Spiegelung des *Redux* Zustandes zunächst höher ist als bei der Verwendung eines Objekts. Dies erscheint zunächst kontraintuitiv. Eine mögliche Erklärung hierfür ist, dass bei der Verwaltung von keinem Objekt keine Aktivität in der App stattfand und somit keine *Garbage Collection* von iOS unternommen wurde. Trotz dieser Anomalie zeichnet sich ein Wachstum der Arbeitsspeichernutzung bei der Erhöhung der Objekte ab. Die CPU-Auslastung wird dabei direkt von der Spiegelung beeinflusst. Bereits ab vier Objekten lastet die Aktualisierung der Spiegelung beide CPUs vollends aus. (vgl. Tabelle 6.1)

Die Auswertung unter Android zeigt einen starken Anstieg der Arbeitsspeichernutzung und einen moderaten Anstieg der CPU-Nutzung. Unter Android kam es dabei jedoch nicht zur gleichen Anomalie der Arbeitsspeichernutzung wie unter iOS. Der Anstieg von 0 auf 32 Objekte ließ die Arbeitsspeichernutzung um einen Faktor von ca. 12,5

Tabelle 6.1: Profiling Auswertung der Redux Spiegelung unter iOS auf dem Gerät *iPhone 7 Plus*.

# Objekte	CPU	RAM
0	0 %	36 MB
1	197,5 %	21,7 MB
2	198 %	32,3 MB
4	200 %	43,8 MB
8	200 %	58,3 MB
16	200 %	62,4 MB
32	200 %	67,9 MB

Tabelle 6.2: Profiling Auswertung der Redux Spiegelung unter Android auf dem Gerät *OnePlus 8T*.

# Objekte	CPU	RAM
0	1 %	115 MB
1	4,9 %	139 MB
2	6,5 %	166,1 MB
4	10,8 %	214,9 MB
8	15,4 %	349,4 MB
16	20,7 %	537,7 MB
32	23,3 %	1438 MB

ansteigen. Die Android App verbrauchte bei der Verwaltung von 32 Objekten annähernd 1,5 GB RAM. Der Anstieg der CPU-Auslastung ist im Vergleich zu iOS zwar weniger signifikant, dennoch deutlich messbar. (vgl. Tabelle 6.2)

Bei der Betrachtung der Testergebnisse muss beachtet werden, dass die Aktualisierung der Spiegelung stets dann ausgeführt wird, wenn Änderungen am *Redux* Zustand vorgenommen werden. So ist es möglich, dass die Spiegelung aktualisiert wird, die native App jedoch die Daten nicht benötigt. Dies hat im Zweifel zur Folge, dass die aufgebrachte Rechenleistung nutzlos war. Ein Ansatz, um dieses Problem zu lösen wäre, die Spiegelung lediglich zu aktualisieren, wenn die native App auf diese zugreifen will. Das Problem hierbei ist jedoch die asynchrone Natur der nativen Module. So müsste eine Aktualisierung der Spiegelung zunächst mittels *Event Emittern* angefragt werden, welche anschließend durch native Module durchgeführt wird. Die Daten werden dabei jedoch synchron angefragt, wodurch es nötig wäre, auf diese zu warten.

Eine beispielhafte Implementierung der Spiegelung des *Redux States* ausgehend von der hybridisierten App aus Kapitel 5 ist im Git Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) auf dem *Branch* `data-sync-redux-mirror` bereitgestellt.

6.1.1 Vorteile

Dieser Ansatz bietet eine hervorragende DX, da die Datenstruktur in den nativen Apps und React Native identisch ist. Folglich verhält sich ein Zugriff auf Daten in React Native, iOS sowie Android fast identisch. Überdies ist hervorzuheben, dass keine explizite Synchronisation notwendig ist, da die Daten bei einer Aktualisierung automatisch an die nativen Apps übertragen werden und folglich stets aktuell sind. Weiterhin wird lediglich ein Speicherort verwendet, wodurch Inkonsistenzen zwischen zwei Datenhaltungsorten unmöglich sind. Zusätzlich besteht die Möglichkeit, Daten auszutauschen, welche von der App selbst erstellt wurden und nicht von einem Server stammen.

6.1.2 Nachteile

Durch die Elimination der nativen Datenbanken ist es notwendig umfangreiche Anpassungen für die Speicherung und den Abruf von Daten durchzuführen. So müssen lesende Zugriffe in Zukunft auf die Spiegelung durchgeführt werden, während schreibende Zugriffe durch Ausführen der *Redux* Aktionen realisiert werden müssen. Dabei ist es notwendig, **sämtliche** Datenstrukturen zu Beginn zu migrieren. Der Initialaufwand ist folglich groß, insbesondere, wenn Datenbankzugriffe nicht zentralisiert wurden und somit viele Codestellen betroffen sind. Die Performanceeinbußen dieses Ansatzes sind

aufgrund der Häufigkeit von Änderungen am *Redux* Zustand nicht vernachlässigbar. Dieser Nachteil wird jedoch hinfällig, sobald die native App hinreichend durch React Native Komponenten ersetzt wurde, sodass ein Zugriff auf Offlinedaten innerhalb der nativen Apps nicht mehr notwendig ist.

6.2 Nachbilden der API-Endpoints ⓑ

Der Ansatz zum Nachbilden der API-Endpoints setzt voraus, dass Daten stets über vom Server definierte REST-Endpoints aktualisiert werden. Ebenso ist es notwendig, dass die App diese Daten aktiv anfragt. Folglich kann dieser Ansatz nicht verwendet werden, wenn innerhalb der Apps *Websockets*[1] oder ähnliche *Event-* bzw. *Subscription*-basierte Ansätze zum Einsatz kommen.

Die grundlegende Idee dieses Ansatzes ist die vollumfängliche Nachbildung der API-Endpoints des Servers innerhalb der nativen Apps sowie innerhalb von React Native. Im Falle einer Datenanforderung durch eine React Native Komponente über einen Endpoint, wird diese an den Server weitergeleitet, sofern das Gerät in der Lage ist, eine Verbindung zu diesem aufzubauen. Wenn die App jedoch offline ist, wird anstelle des Servers die native App angefragt, welche den gleichen Endpoint bereitstellt. Hält die native App dabei aktuelle Daten bereit, sind diese von der React Native Komponente abrufbar. Anschließend können diese Daten im Zustand der React Native Komponente gespeichert werden. Sollte die native App keine Daten vorhalten, resultiert dies in einem Fehler. Auf diesen Ansatz wird mit ⓑ referenziert.

Da bei diesem Vorgehen beide Datenbanken isoliert zu betrachten sind, ist es zugleich notwendig, dass React Native ebenfalls die Endpoints nachstellt, sollte die native Anwendung im offline-Szenario Daten benötigen. (vgl. Abbildung 6.3)

1 *Websockets* erlauben eine bidirektionale Kommunikation zwischen Client und Server. [84]

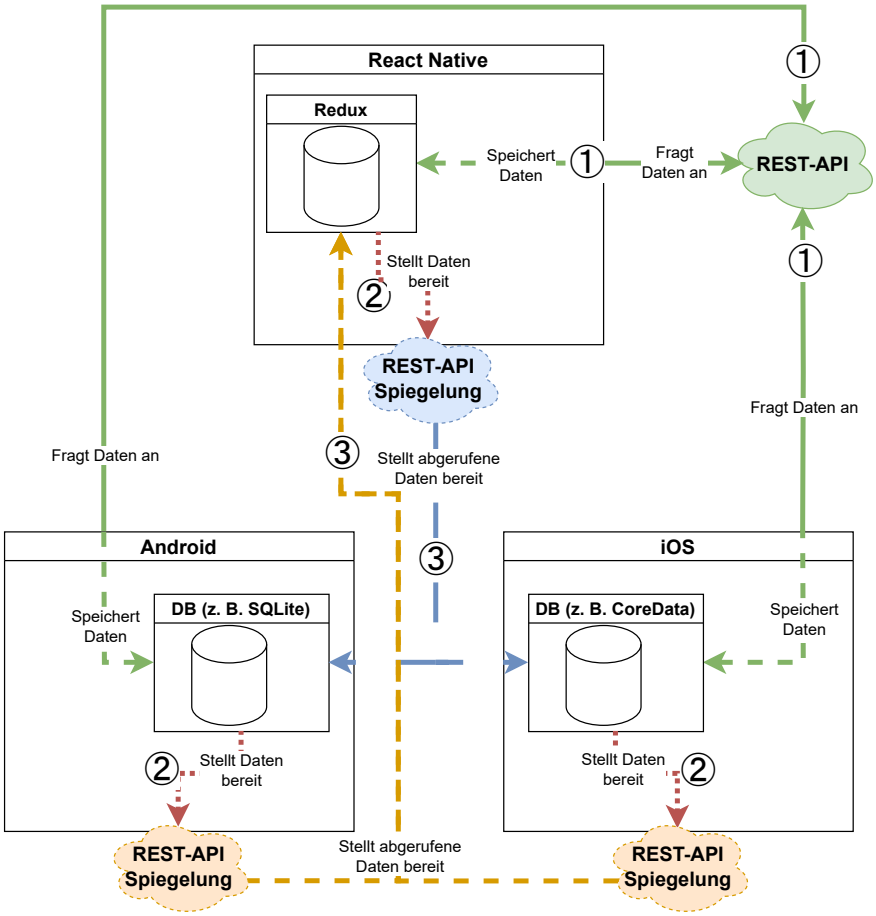


Abbildung 6.3: Schematisch vereinfachter Aufbau von ⓑ.

Grüne Pfeile: Daten werden von der REST-API abgefragt und in der lokalen Datenbank gespeichert, sofern eine Verbindung zum entsprechenden Server besteht.

Orange und blaue (gestrichelte) Pfeile: Daten werden über die lokale REST-API abgerufen, sofern keine Verbindung zum Server besteht.

Rote (gepunktete) Pfeile: Die Daten für die lokale REST-Schnittstelle werden von der lokalen Datenbank abgerufen.

①: Daten werden im Online-Szenario von der nativen iOS bzw. Android App oder React Native von einem Server angefragt. Anschließend werden die Daten in der jeweiligen lokalen Datenbank persistiert.

②: Die von den nativen Apps und React Native bereitgestellte REST-API bezieht die Daten aus der jeweiligen Offline-Datenbank.

③: Im Offline-Szenario werden die Daten aus der lokalen REST-API abgerufen, sofern diese dort bereitgestellt werden können.

Dabei ist evident, dass dieser Ansatz mit einem enormen Entwicklungsaufwand verbunden ist. So ist es notwendig, alle von der App angefragten API-Endpoints innerhalb der iOS App, der Android App sowie React Native zu implementieren. Weiterhin ist es

erforderlich, alle drei Implementierungen anzupassen, sofern ein Endpoint des Servers geändert wird.

Um diesen Ansatz innerhalb der nativen Apps sowie React Native umzusetzen, empfiehlt es sich Bibliotheken zu verwenden, welche das Erstellen von REST-APIs innerhalb der betreffenden Plattform ermöglichen. Unter iOS kann dafür *GCDWebServer* (vgl. `https://github.com/swisspol/GCDWebServer`, abgerufen am 26. Oktober 2023) zum Einsatz kommen. Unter Android ist es möglich, *restserver* (vgl. `https://github.com/skornei/restserver`, abgerufen am 26. Oktober 2023) zu verwenden. Innerhalb von React Native ist es dabei nicht möglich Bibliotheken zu verwenden, welche auf das von Node implementierte Modul *http* zurückgreifen.[1] Grund dafür ist, dass dieses in React Native nicht implementiert ist. Deswegen ist es notwendig auf z. B. *react-native-http-bridge-refurbished* (vgl. `https://github.com/Alwinator/react-native-http-bridge-refurbished` abgerufen am 26. Oktober 2023) zurückzugreifen. *react-native-http-bridge-refurbished* stellt dabei native Module für iOS und Android zur Verfügung, welche innerhalb der nativen Apps einen Webserver bereitstellen. Dies hat zur Folge, dass innerhalb der nativen Apps effektiv zwei Webserver bereitgestellt werden. Somit ist es notwendig, diesen beiden Servern bei der Initialisierung verschiedene Ports zuzuweisen.

Bei der Nutzung dieses Ansatzes in einer Produktivumgebung sollte sorgfältig darauf geachtet werden, Anfragen an den Webserver lediglich innerhalb der App zu beantworten, um zu verhindern, dass die bereitgestellte API als möglicher Angriffsvektor dient.

Eine beispielhafte Implementierung der Nachbildung der API-Endpoints, ausgehend von der hybridisierten App aus Kapitel 5, ist im Git-Repository (vgl. `https://github.com/flokol120/react-native-brownfield-examples`, abgerufen am 26. Oktober 2023) unter dem *Branch* `data-sync-rest-api` verfügbar.

6.2.1 Vorteile

Dieser Ansatz erlaubt eine starke Isolation beider Softwareteile. React Native und die nativen Apps sind dabei größtenteils eigenständig betreibbar. So ist es lediglich erforderlich, dass sowohl React Native als auch die nativen Apps ihre eigenen Datenbanken verwalten, ohne die Notwendigkeit einer expliziten Synchronisation der Datenbanken oder einer Anpassung der Datenbankzugriffe. Überdies bleibt der Aufruf der Endpoints bis auf den angefragten Hostname[2] stets identisch.

1 Z. B. *express*, *fastify*, o. ä.
2 Der Hostname ist dabei entweder der Server, die native App oder React Native

6.2.2 Nachteile

Bei der Replikation der API-Endpoints entsteht ein beträchtlicher Entwicklungs- und Wartungsaufwand, da sämtliche API-Endpoints dreimal umgesetzt werden müssen. Da die API-Endpoints für einen Aufruf aus den nativen Apps stets zur Verfügung stehen müssen, ist es notwendig, React Native bereits zum Start der App zu initialisieren. Bei der Umsetzung dieses Ansatzes, ist es unmöglich native Module bzw. *Turbo Native Modules* zu verwenden, obschon dies die bevorzugte Kommunikationsschnittstelle ist, da es erforderlich ist einen, HTTP-Server bereitzustellen. Bei der Verwendung dieses Ansatzes ist es lediglich möglich, Daten zu synchronisieren, welche über REST-APIs abgerufen werden. Daten, welche z. B. innerhalb der nativen App transformiert oder gar selbst erstellt werden, können in diesem Fall nur dann synchronisiert werden, wenn dafür künstliche API-Endpoints erstellt werden, welche vom Server nicht angeboten werden. Weiterhin sind die zur Verfügung stehenden Bibliotheken zum Bereitstellen einer REST API stark veraltet. *GCDWebServer* wird nicht mehr gewartet, die letzte Änderung fand vor ca. drei Jahren statt und an *restserver* wurden zuletzt vor ca. sechs Jahren Anpassungen durchgeführt. Lediglich `react-native-http-bridge-refurbished` wird aktiv entwickelt.[1]

6.3 Synchronisation nativer Datenbanken und Redux ©

Die Synchronisation der nativen Datenbanken mit dem dokumentenorientierten Speicher aus React native ist der letzte in dieser Ausarbeitung besprochener Ansatz. Die Synchronisation zweier Datenbanken kann dabei prinzipiell auf verschiedenste Arten sichergestellt werden. Im Folgenden wird dabei ein konkreter Ansatz verfolgt, welcher in Abbildung 6.4 bzw. Abbildung 6.5 schematisch dargestellt wird. © ist als Referenzsymbol für diesen Ansatz vorgesehen.

1 Bevor diese Bibliothek für diese Ausarbeitung verwendet werden konnte, waren manuelle Anpassungen des Autors an dieser notwendig, um einen Fehler zu beheben. Dieser Fehler wurde nach der Einreichung eines *Merge Requests* des Autors auch im offiziellen Repository behoben.

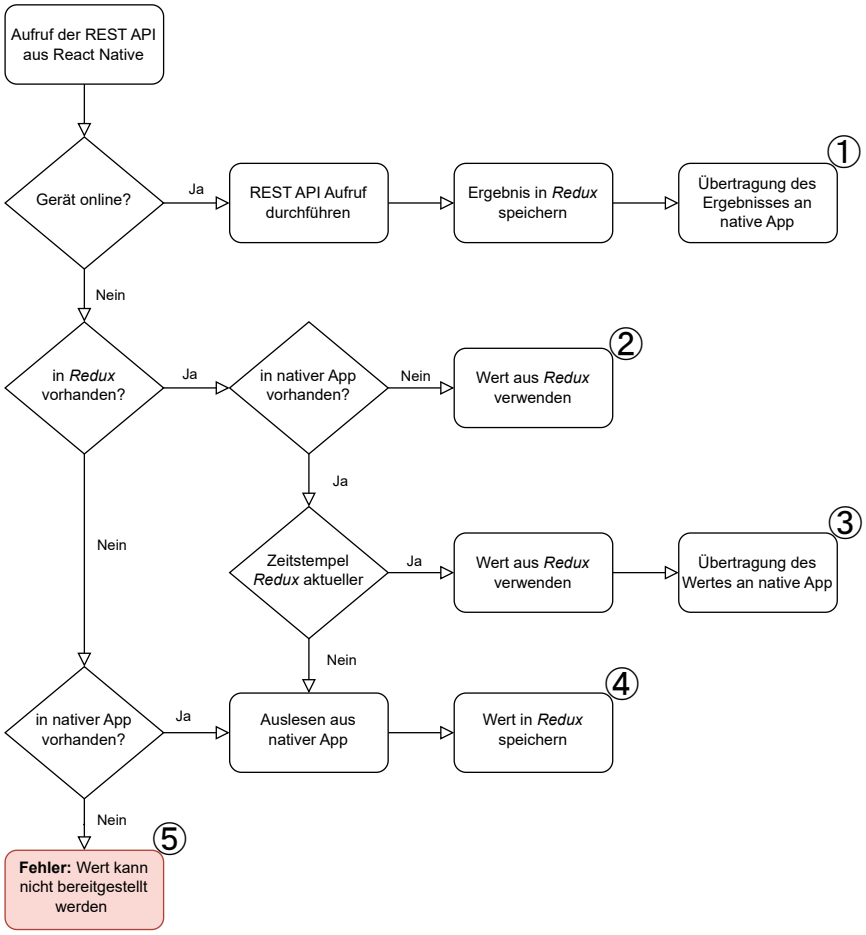


Abbildung 6.4: Schematisch vereinfachter Ablauf eines REST-Aufrufs.

Zunächst wird geprüft, ob das Gerät eine Verbindung zum Server aufbauen kann. Ist eine Verbindung erfolgreich zustande gekommen, werden die benötigten Daten abgefragt, in den *Redux Store* React Natives gespeichert und zuletzt über eine Methode eines nativen Moduls auch in der nativen App persistiert. (1)

Konnte keine Serververbindung aufgebaut werden, wird der Wert aus dem lokalen *Redux Store* geladen, vorausgesetzt der Wert ist in *Redux* vorhanden, in der nativen App jedoch nicht. (2)

Ist der Wert sowohl innerhalb von *Redux* als auch in der nativen App vorhanden, ist es notwendig den aktuelleren Wert zu identifizieren. Dazu kann z. B. ein Zeitstempel verwendet werden, welcher jedoch beim Abruf der Daten zusätzlich persistiert werden muss. Ist der Wert aus *Redux* aktueller bzw. identisch mit dem der nativen App, wird der Wert aus *Redux* verwendet. Der ausgelesene Wert wird in die native App übertragen. (Eine Übertragung an die native App kann in diesem Fall übersprungen werden, sofern die Zeitstempel identisch sind) (3)

Ist innerhalb von *Redux* kein Wert gespeichert, wird über eine Methode eines Native Modules, der Wert stattdessen aus der lokalen Datenbank der nativen App abgerufen und anschließend im Redux Store React Natives persistiert. Dieses Vorgehen wird ebenfalls angewandt, sofern die Daten in *Redux* vorhanden sind, die Werte der nativen App jedoch aktueller sind.(4)

Sind die Daten nicht in der Datenbank der nativen App zu finden, können die Daten im Offlineszenario nicht bereitgestellt werden und es kommt zu einem Fehler. (5)

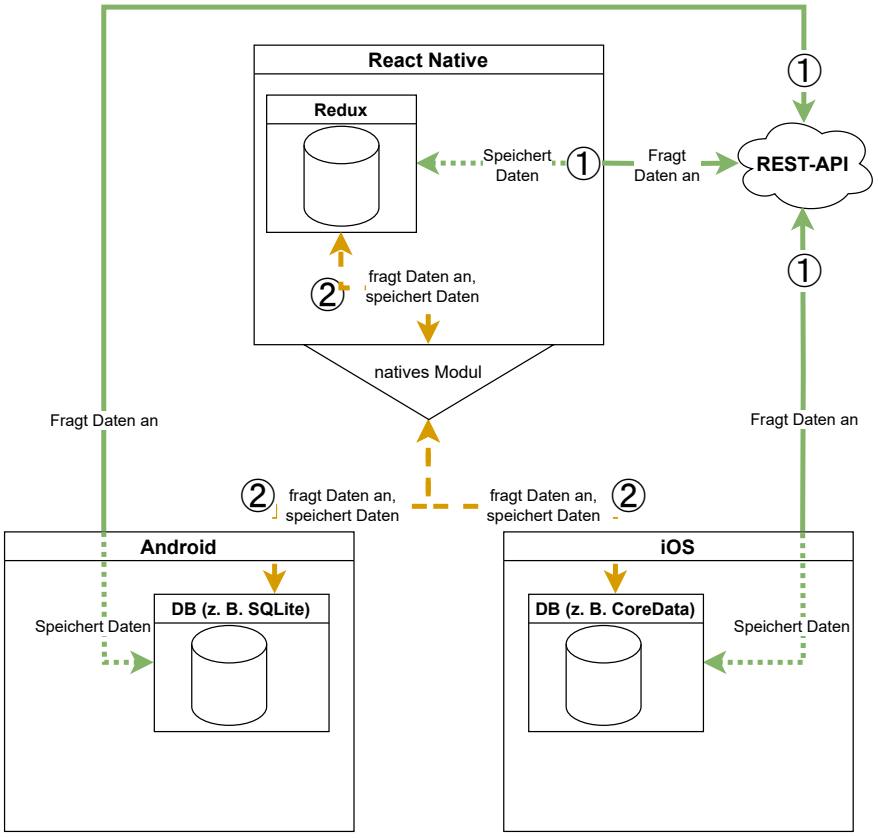


Abbildung 6.5: Schematisch vereinfachter Aufbau von Ansatz ©.

Grüne Pfeile: Daten werden von der REST-API abgefragt, sofern eine Verbindung zum entsprechenden Server besteht.

Orange (gestrichelte) Pfeile: Daten werden unter Zuhilfenahme eines nativen Moduls – falls möglich – aus den nativen Datenbanken oder Redux ausgelesen, sofern keine Verbindung zum Server der REST API aufgebaut werden kann.

①: Daten werden im Online-Szenario von der nativen iOS bzw. Android App oder React Native von einem Server angefragt. Anschließend werden die Daten in der jeweiligen lokalen Datenbank persistiert.

②: Daten werden über ein natives Modul (vgl. Abschnitt 4.2) aus der nativen App ausgelesen bzw. an diese übertragen.

Für die Implementierung des Ansatzes ist es dabei möglich, sowohl native Module (vgl. Abschnitt 4.2) als auch *Turbo Native Modules* (vgl. Abschnitt 4.4) zu verwenden. Aufgrund der besseren Performance, die *Turbo Native Modules* versprechen, insbesondere durch die Verwendung von *JSI* (vgl. Abschnitt 4.1.2), wurde auf dem *Branch* `data-sync-command-syncing` beispielhaft ein *Turbo Native Module* implementiert, um die Synchronisation abzubilden. Dieser *Branch* ist unter `https://github.com/flokol120/react-native-brownfield-examples` (abgerufen am 26. Oktober 2023) verfügbar.

Für die Umsetzung innerhalb von React Native eignet sich unter anderem das *Command Pattern*. Beispielhaft werden die API-Zugriffe aus diesem Grund mit dem *Command Pattern* umgesetzt.

Einschub: Command Pattern

Commands übernehmen immer eine bestimmte Aufgabe, z. B. das Abrufen einer REST-API oder das Einfügen von Daten in die lokale Datenbank. *Commands* sind somit auf den ersten Blick ähnlich zu klassischen Funktionen, da sie jedoch Klassen bzw. Objekte sein können, besitzen *Commands* zusätzlich einen Zustand und können andere Klassen/*Commands* erweitern, um Funktionalitäten von anderen Klassen zu übernehmen. Der Zustand des *Commands* kann dabei z. B. Informationen wie den *Redux State* der Anwendung, den Netzwerkstatus des Gerätes o. Ä. beinhalten. Jeder *Command* stellt dabei eine `execute`-Methode bereit, welche den Effekt des *Commands* letztlich ausführt. [85, 86]

Ein *Command Controller* reiht dabei hinzugefügte *Commands* in eine Warteschlange ein und arbeitet diese sequenziell durch den Aufruf der `execute`-Methode des *Commands* ab. In Quellcode C.1 ist eine beispielhafte Implementierung eines abstrakten *Commands* zu sehen.

Insbesondere für den Abruf von Daten des Servers (REST-API) eignet sich das *Command Pattern* gut, da die Commands meist asynchron ablaufen können und somit innerhalb eigener Threads ausführbar sind. Dadurch blockieren sie den Haupt-Thread der Apps nicht[1].

In den Abbildungen 6.4 und 6.5 ist ausschließlich die Synchronisation nach einem erfolgreichen Abruf neuer Daten über die eine REST API abgebildet. Wird ein Wert jedoch von React Native selbst erstellt oder aktualisiert, ist es erforderlich, dass dieser Wert ebenfalls in die native App übertragen wird. Dennoch eignet sich das *Command Pattern* ebenfalls für die Übertragung der Daten an die native App. Werden die Daten wiederum in der nativen App erstellt oder modifiziert, ist eine Übertragung nicht notwendig, sofern der Zeitstempel der Daten ebenfalls angepasst wird.

1 Dies trifft dabei nur auf die nativen Apps zu. React Native besitzt zum Verfassungszeitpunkt keine Möglichkeit zum Verwenden mehrerer Threads. Mit der Weiterentwicklung der neuen Architektur soll dies jedoch in Zukunft möglich sein.

6.3.1 Vorteile

Dieser Ansatz bietet einen erheblichen Mehrwert, wenn das Ziel darin besteht, eine inkrementelle Hybridisierung der bestehenden Apps durchzuführen.[1] Dies liegt daran, dass die hybridisierte App nicht mehr von der nativen Speicherlogik abhängig ist. Somit kann die alte Speicherlogik restlos entfernt werden, sobald alle nativen Komponenten mittels React Native hybridisiert wurden. Ansonsten bietet dieser Ansatz den Vorteil, dass die Synchronisation der Daten ebenfalls inkrementell erfolgen kann. Es ist möglich, die Synchronisation der Daten erst dann durchzuführen, wenn diese innerhalb von React Native benötigt werden. Zusätzlich bietet dieser Ansatz einen einheitlichen Ansatz zur Synchronisation durch die Verwendung des *Command Patterns*. Weiterhin ist es möglich, für die Kommunikation zwischen React Native und den nativen Apps *Turbo Native Modules* zu verwenden. Da Daten stets aktiv von React Native beim Abruf synchronisiert werden, ist es nicht notwendig, React Native bereits zum Start der nativen App zu initialisieren. Da Daten, die über die REST-API abgerufen werden, in der Regel im JSON-Format vorliegen, bestand bereits zuvor die Notwendigkeit, die Daten für die Offlinedatenbanken in den nativen Apps zu transformieren. Somit ist es möglich, die Konvertierungsmechanismen bei der Übertragung der Daten von React Native in die nativen Apps wiederzuverwenden, vorausgesetzt, das Format der Daten bleibt identisch. Zusätzlich ist es möglich, Daten auszutauschen, welche von der App selbst erstellt bzw. transformiert wurden und nicht vom Server stammen.

6.3.2 Nachteile

Der Entwicklungsaufwand dieses Ansatzes ist hoch, da es notwendig ist, abgerufene Daten innerhalb eines nativen Moduls in die Datenbanken der nativen iOS und Android Apps zu speichern. Folglich ist es notwendig, für jede Datensynchronisation plattformspezifischen Code in Objective-C/Swift und Java/Kotlin zu verfassen. Weiterhin besteht die Möglichkeit, dass es zu Inkonsistenzen kommt, sofern die Synchronisation in die nativen Apps ungewissenhaft bzw. fehlerhaft implementiert wird.

6.4 Gegenüberstellung der Synchronisationsstrategien

Die Ermittlung der optimalen Methode aus den drei dargelegten Synchronisationsstrategien ist ohne Kenntnis über eine konkrete zu hybridisierenden App nicht möglich.

1 Das Ziel ist es hierbei, die nativen Apps Inkrement für Inkrement durch eine hybride React Native Lösung auszutauschen. Folglich ist es dienlich, wenn perspektivisch alle relevanten Offline-Daten innerhalb der React Native Implementierung zur Verfügung stehen.

So existieren Szenarien, in welchen Ansatz ⓒ geeigneter ist, als die Verwendung von Ansatz ⓐ. Sind z. B. innerhalb der nativen Apps Datenbankzugriffe nicht in einer allgemeinen Funktion gebündelt, ist es möglich, dass der Aufwand der Einführung der *Redux* Spiegelung (ⓐ) dem Aufwand zur Synchronisation der nativen Datenbanken und *Redux* (ⓒ) überwiegt. Folglich sind konkrete Anwendungsfälle stets individuell zu betrachten, um den bestmöglichen Synchronisationsansatz für die entsprechende App zu wählen. Dabei ist es notwendig, die dargelegten Vor- und Nachteile individuell gegeneinander abzuwägen, um den bestmöglichen Ansatz zu wählen.

Lediglich das Vorgehen nach ⓑ ist als weniger geeignet als die Ansätze ⓐ und ⓒ zu bewerten. In Ansatz ⓑ ist der Entwicklungsaufwand für das Einrichten von drei separaten API-Endpoints signifikant höher als bei den beiden alternativen Vorgehensweisen. Besonders mangelhaft an diesem Vorgehen ist, dass lediglich eine direkte Synchronisation der Daten möglich ist, welche auch vom Server angeboten werden. Es ist nicht ungewöhnlich, dass Apps Daten vorhalten, die sie selbst erzeugt haben. Die Synchronisation dieser Daten ist ebenso unerlässlich.

In Tabelle 6.3 sind einige Indikationen sowie Kontraindikationen zusammengefasst, welche dazu dienen, die Auswahl eines Ansatzes zu vereinfachen.

Tabelle 6.3: Zusammenfassung von Indikationen und Kontraindikationen als Entscheidungshilfe zur Auswahl des passenden Ansatzes.

Ansatz	Indikationen	Kontraindikationen
ⓐ *Redux* Spiegelung	• Zentralisierte Datenbankzugriffe gegeben. • Anzahl der Offline-Daten gering. • React Native soll die nativen Apps in Zukunft vollumfänglich ersetzen.	• Datenbankzugriffe sind in gesamter App verteilt und nicht zentralisiert. • Niedrige Arbeitsspeicher- und CPU-Auslastung erforderlich. • Häufige Aktualisierung der Offline-Daten.
ⓑ REST-API	• Offline-Daten werden lediglich von einem Server abgerufen und unbearbeitet persistiert.	• Unzureichendes Zeitkontingent für Umsetzung und Wartung. • Regelmäßige Anpassungen an der Server-Schnittstelle. • Daten vom Server werden modifiziert oder lokale Daten werden erstellt und persistiert.
ⓒ Synchronisation	• Datenbankzugriffe sind in gesamter App verteilt und nicht zentralisiert. • Teile der nativen Apps sollen in Zukunft neben der React Native Implementierung weiterhin betrieben werden. • Zu Beginn der Integration ist es nicht notwendig, dass alle Offlinedaten der nativen Apps innerhalb von React Native zu verarbeiten sind.[1]	• React Native soll die nativen Apps sehr zeitnah[2] vollumfänglich ersetzen.

1 Dies ist für diesen Ansatz dienlich, da die Daten so erst dann übertragen und synchronisiert werden müssen, wenn diese in React Native benötigt werden. Der Entwicklungsaufwand lässt sich so besser verteilen.

2 Soll React Native die native App recht schnell ersetzen, wird unnötigerweise viel Zeit in die notwendigen Brückenelemente investiert.

7 Fazit

Die Ergebnisse dieser Arbeit zeigen, dass eine Brownfield Hybridisierung von nativen iOS und Android Apps mittels React Native umsetzbar ist. Kapitel 2 bietet dazu einen Einstieg in die Brownfield Hybridisierung, wodurch mögliche Risiken, aber auch Potenziale dieses Ansatzes aus konkreten Fallstudien extrahiert und gegenübergestellt werden. Die in Kapitel 3 - 4 vermittelten technischen Aspekte, schaffen dabei die Grundlage für die Durchführung einer Brownfield Hybridisierung mit React Native. Kapitel 5 stellt die notwendigen Schritte zur Integration von React Native in bestehende native iOS und Android Apps detailliert dar. Dies instruiert Lesende, wie die Integration an anderen Apps zu vollführen ist. Ermöglicht wird dies durch die repräsentativen Beispielapps, wodurch die Reproduzierbarkeit in ähnlichen Apps geschaffen wird. Dabei ist zu beachten, dass die dargestellte Integration ausschließlich in die zuvor erstellten iOS und Android Apps durchgeführt wurde. Folglich ist die dargestellte Integration trotz der Verwendung von repräsentativen iOS und Android Apps nicht quantitativ innerhalb dieser Ausarbeitung verifiziert. In Verbindung mit den in Kapitel 6 dargestellten Synchronisationsmechanismen wird eine möglichst nahtlose Integration bei gleichzeitiger Wahrung der Konsistenz von Daten der mobilen Applikationen für den Offlinebetrieb sichergestellt.

7.1 Zusammenfassung

Technische Aspekte

Zunächst erfolgt eine technische Einführung in die Bibliothek React von Meta. Dabei konzentriert sich dieses Kapitel besonders auf das Erstellen von React Komponenten, (vgl. Abschnitt 3.1) da dies einen essenziellen Bestandteil bei der Verwendung von React darstellt. React Komponenten werden dabei mittels *JSX* (vgl. Abschnitt 3.2) definiert, weswegen die Syntax von *JSX* zunächst genauer betrachtet wird. Daraufhin wird das Konzept um *Props* (vgl. Abschnitt 3.3) erläutert, welche für den Informationsfluss innerhalb von React Native von besonderer Bedeutung sind. Weiterhin werden funktionale Komponenten (vgl. Abschnitt 3.4) sowie *Hooks* (vgl. Abschnitt 3.4.1) eingeführt. Da die React Dokumentation das Verwenden von funktionalen Komponenten vorzieht, werden diese detaillierter als Klassen-Komponenten behandelt. (vgl. Abschnitt 3.5) Ausgehend

von der Limitierung von *Props* wird der Nutzen von Kontexten sowie eines globalen Zustandes aufgezeigt. (vgl. Abschnitt 3.6) Als letzter Aspekt innerhalb Reacts wird der *VDOM* erläutert, da dieser essenziell für die Existenz von React Native ist. (vgl. Abschnitt 3.7)

Ausgehend von dieser Erläuterung werden Grundkonzepte React Natives von Meta behandelt. Zunächst wird das Konzept um den *Host View Tree* aufgegriffen, um ein Grundverständnis der Architektur React Natives zu vermitteln. Daraufhin wird *Hermes* als JavaScript-Engine eingeführt. (vgl. Kapitel 4) Anschließend werden die wichtigsten Aspekte der neuen Architektur – insbesondere der neuen Render Engine *Fabric* – wie das *View Flattening*, (vgl. Abschnitt 4.1.1) *JSI*s (vgl. Abschnitt 4.1.2) sowie das *Lazy Loading* (vgl. Abschnitt 4.1.3) aufgezeigt. Diese Aspekte sollen dabei dem Lesenden eine Entscheidungsgrundlage vermitteln, ob bei der Integration von React Native die alte oder die neue Architektur verwendet werden soll. Im Anschluss wird die Kommunikationsschnittstelle React Natives anhand eines konkreten Beispiels im Detail betrachtet. (vgl. Abschnitt 4.2) Diese ist dabei besonders wichtig bei der Verwendung von React Native in einer *Brownfield*-Entwicklung, da auf diese Weise eine Kommunikation mit den nativen Apps möglich ist. Dazu wird ein einfaches natives Modul zur Protokollierung einer Nachricht entwickelt, wobei spezifische Schritte für iOS, Android und React Native gesondert beschrieben werden. Native Module bieten dabei eine bidirektionale Kommunikationsmöglichkeit, sofern die Kommunikation von React Native initiiert wurde. Ist es erforderlich, Daten der nativen Apps React Native zu übermitteln, können *Event Emitter* verwendet werden. Um dies zu demonstrieren, wird exemplarisch ein *Event Emitter* in Verbindung mit einem nativen Modul entwickelt. (vgl. Abschnitt 4.3) Da bei der Verwendung von nativen Modulen nicht die neue Architektur zum Tragen kommt, wird zusätzlich die Definition von *Turbo Native Modules* behandelt. *Turbo Native Modules* funktionieren dabei in Verbindung mit der neuen Architektur. Der größte Unterschied in der Definition von *Turbo Native Modules* im Vergleich zu nativen Modules ist, dass ausgehend von einem TypeScript Interface unter Zuhilfenahme des Tools *Codegen* bereits vordefinierte Interfaces für iOS und Android generiert werden. So ermöglichen *Turbo Native Modules* eine typsichere Kommunikation, was bei der Verwendung von nativen Modulen nur bedingt möglich ist. Die Voraussetzung für die Verwendung von *Turbo Native Modules* ist jedoch, dass die neue Architektur in der iOS und Android App aktiviert ist. Das notwendige Vorgehen dafür wird dabei für beide Plattformen beschrieben. Anschließend wird ein *Turbo Native Module* implementiert, welches – wie auch das zuvor erstellte native Modul – eine Nachricht in die nativen Apps überträgt. (vgl. Abschnitt 4.4) Daraufhin wird die Möglichkeit zur Definition von neuen React Native Komponenten erläutert. Dazu wird eine native Komponente implementiert, welches die nativen horizontalen Fortschrittsbalken unter iOS und Android abbildet. (vgl. Abschnitt 4.5) Zuletzt wird die Existenz von *Turbo Native Components* erwähnt. *Turbo Native Components* sind dabei analog zu *Turbo Native Modules* für die

Verwendung mit der neuen Architektur bzw. mit der *Fabric* Render Engine vorgesehen. Da es jedoch im Gegensatz zu nativen Modulen möglich ist, native Komponenten über einen Kompatibilitätsmodus mit der neuen Architektur sowie *Fabric* zu betreiben und die Dokumentationsdichte zum Erstellen von *Turbo Native Components* sehr gering ist, wird das Erstellen von *Turbo Native Components* nicht weiter betrachtet. Lediglich die Verwendung des Kompatibilitätsmodus ausgewählter Komponenten wird dargelegt. (vgl. Abschnitt 4.6)

Überdies wird eine beispielhafte Integration von React Native in bestehende iOS und Android Apps vorgenommen. (vgl. Kapitel 5) Dazu werden zunächst die erstellten iOS und Android Apps beschrieben. Anschließend wird React Native innerhalb des bestehenden Projekts initialisiert. Daraufhin wird eine beispielhafte React Komponente implementiert, welche im Folgenden innerhalb der nativen iOS und Android Apps angezeigt werden soll. Diese Komponente ist dabei in der Lage, einen Zählerwert anzuzeigen. Der Zählerwert ist dabei über zwei Buttons inkrementierbar und dekrementierbar. (vgl. Abschnitt 5.3) Im Anschluss werden die notwendigen Schritte zur Integration von React Native in eine native iOS Apps beschrieben. Dazu sind zunächst Anpassungen am Buildprozess der Anwendung notwendig, anschließend wird eine Verbindung zu React Native aufgebaut, um daraufhin die zuvor erstellte React Komponente zur Anzeige zu bringen. Zuletzt sind unter iOS spezifische Anpassungen notwendig, um React Native in einem produktiven Umfeld betreiben zu können. (vgl. Abschnitt 5.4) Anschließend werden die erforderlichen Schritte zur Integration von React Native in die native Android App beschrieben. Dazu sind auch unter Android zunächst Anpassungen am Build-Prozess notwendig, anschließend muss eine Verbindung zu React Native sichergestellt werden und zuletzt wird die erstellte React Komponente zur Anzeige gebracht. (vgl. Abschnitt 5.5) Sowohl unter iOS als auch unter Android werden zwischen zwei Integrationsmöglichkeiten unterschieden. So ist es möglich, die React Native Komponenten den gesamten zur Verfügung stehenden Platz zu ersetzen oder aber die React Native Komponente in eine bereits bestehende Ansicht neben nativen UI-Elementen einzubetten. Die Verwendung dieser Ansätze wird dabei für beide Plattformen anhand der Beispiel-Komponente demonstriert. Im vorherigen Kapitel wurde bereits beschrieben, welche Schritte notwendig sind, um die neue Architektur innerhalb von React Native zu aktivieren. Bei der *Brownfield*-Entwicklung sind jedoch weitere manuelle Eingriffe notwendig, um die neue Architektur zu aktiveren. Diese Schritte werden abschließend dargelegt. (vgl. Abschnitt 5.6)

Bei der Integration der zuvor erstellten React Komponente stellte sich heraus, dass es bei der Verwendung der nicht bildschirmfüllenden Integration zu Inkonsistenzen der Zählerwerte kommen kann. Ebendarum beschäftigt sich das letzte Kapitel mit der Möglichkeit zur Synchronisation der Zustände, um Inkonsistenzen zu vermeiden. (vgl. Kapitel 6) Zunächst wird die Spiegelung des Zustandes innerhalb von React Native besprochen. Dazu entfallen die Zustände der nativen Apps gänzlich, lediglich der Zustand

der React Native Anwendung wird verwendet. Folglich ist es erforderlich eine Möglichkeit zu schaffen, bestehende lesende wie auch schreibende Zugriffe der nativen App auf den Zustand von React Native zu migrieren. Weiterhin wird die Leistungsfähigkeit dieses Ansatzes genauer betrachtet. (vgl. Abschnitt 6.1) Ein weiterer Ansatz zur Wahrung der Konsistenz der Zustände beider Software-Komponenten wird durch die Nachbildung der API-Endpoints dargelegt. Dazu ist es notwendig in React Native, iOS sowie Android alle angefragten API-Endpoints zu implementieren. Für ein offline Szenario ist es somit lediglich notwendig, den Hostnamen der Anfrage anzupassen. Dennoch birgt dieser Ansatz einige Nachteile. So entsteht neben dem enormen Initialaufwand zur Umsetzung von drei APIs zusätzlich ein großer Wartungsaufwand. Weiterhin ist es notwendig, die React Native bereits zu Beginn zu initialisieren, damit ein Zugriff auf die API-Schnittstelle erfolgen kann. Zusätzlich macht dieser Ansatz keinen Gebrauch von nativen Modulen. (vgl. Abschnitt 6.2) Zuletzt wird ein Ansatz zur Synchronisation der nativen Datenbank sowie dem React Native Zustand dargelegt. Dabei kommt das *Command Pattern* zum Einsatz, um die Synchronisation architekturell zu gestalten. Als Vorteile werden dabei der vereinheitliche Ansatz der Synchronisation durch die Verwendung des *Command Patterns*, die Möglichkeit zur Nutzung von *Turbo Native Modules* sowie das Potenzial zur restlosen Entfernung der nativen Datenbank nach Abschluss der Hybridisierung hervorgehoben. (vgl. Abschnitt 6.3)

Software-Engineering Aspekte

Bevor die Integration von React Native in native iOS und Android Apps behandelt wird, werden zunächst zwei Fallstudien zur Brownfield Hybridisierung der Softwareunternehmen Airbnb (vgl. Abschnitt 2.1) sowie der CURSOR Software AG (vgl. Abschnitt 2.2) zusammengetragen. Ausgehend von diesen Erfahrungen werden dabei sowohl gelungene als auch misslungene Aspekte der Integration gegenübergestellt. (vgl. Abschnitt 2.3) Die Auswertung zeigt insbesondere, dass sich React Native nach der abgebrochenen Integration in die bestehenden nativen iOS und Android Apps von Airbnb positiv weiterentwickelt hat. Die Auswertung der Fallstudie der CURSOR Software AG hat dabei hervorgebracht, dass einige Aspekte der Fallstudie von Airbnb aufgrund von Änderungen von React Native nicht mehr bestätigt werden konnten und folglich obsolet sind. Dieses Kapitel zielt hauptsächlich darauf ab, dem Lesenden mögliche Risiken, aber auch Potenziale der Integration von React Native aufzuzeigen. (vgl. Kapitel 2)

Im letzten Kapitel werden drei konkrete Synchronisationsmechanismen zur Wahrung der Konsistenz der Offlinedaten präsentiert. Dafür werden drei konkrete Möglichkeiten dargelegt. (vgl. Kapitel 6) Zuletzt werden die drei vorgestellten Synchronisationsmöglichkeiten gegenübergestellt und Empfehlungen zur Auswahl eines angemessenen Mechanismus für konkrete Anwendungsfälle ausgesprochen. (vgl. Abschnitt 6.4)

7.2 Ausblick

Die Brownfield Hybridisierung von nativen iOS und Android Apps mittels React Native bietet viele zusätzliche Forschungsmöglichkeiten. Grund dafür ist, dass zu dieser Thematik bisher überaus wenige wissenschaftliche Ausarbeitungen existieren.

Zunächst ist es möglich, sich mit den Software-Engineering-Aspekten einer *Brownfield*-Entwicklung zu beschäftigen. Dabei können zunächst Themen wie mögliche Harmonisierungen der nativen iOS und Android Apps untersucht werden. Bei der Untersuchung einer solchen Harmonisierung wäre es interessant zu beobachten, ob diese eine daran anschließende *Brownfield*-Entwicklung erleichtern könnte. Weiterhin könnte ein Priorisierungsverfahren entwickelt werden, um Bereiche der nativen Apps anhand einer so erstellten Prioritätenliste schrittweise zu hybridisieren.

Überdies wäre es möglich, das in dieser Ausarbeitung dargelegte technische Verfahren zur *Brownfield*-Entwicklung mit React Native an einigen tatsächlich existierenden nativen iOS und Android Apps durchzuführen. Dabei könnte ebenfalls untersucht werden, inwiefern Teile der Integration automatisiert werden können, um diese Integrationen zu beschleunigen. Dadurch wäre es möglich quantitativ zu überprüfen, ob die Annahmen der Repräsentativität der in Abschnitt 5.1 beispielhaft erstellten iOS und Android App zutreffend sind. Dazu wäre es jedoch notwendig, Open-Source-Projekte zu finden, welche native iOS und Android Apps beinhalten. Eine weitere Herausforderung wäre es dabei, die Integration von React Native in Projekten durchzuführen, in denen der Code vollends unbekannt ist. Dieser Forschungsansatz würde folglich viel Zeit in Anspruch nehmen.

Weiterhin ist es möglich, die Entwicklung von *Turbo Native Modules* mit C++ zu ergründen. Dieser Ansatz ist jedoch nur dann im Kontext einer Brownfield Hybridisierung von nativen iOS und Android Apps sinnvoll, wenn bereits C++ Code in beiden nativen Apps vorliegt, welcher auf diese Art zentralisiert werden kann. Da wenige Apps auf C++-Implementierungen zurückgreifen, könnte dieser Ansatz mit der Harmonisierung der nativen iOS und Android Apps verzahnt werden. So wäre es möglich, geeignete Funktionalität der nativen iOS und Android Apps in C++ umzusetzen, um diese Implementierung später mit *Turbo Native Modules* zu verbinden.

Zusätzlich wäre es sinnvoll, weitere Synchronisationsansätze zu betrachten. So könnte untersucht werden, inwiefern der Zugriff auf die nativen Datenbanken aus React Native umsetzbar sind. Dieser Ansatz würde die Notwendigkeit eines globalen Zustandes innerhalb von React Native eliminieren. Zugleich würde dadurch jedoch die reaktiven Eigenschaften des *Redux* Zustandes verloren gehen. Weiterhin wäre es möglich zu untersuchen, ob bei der Verwendung der *Redux* Spiegelung (vgl. Abschnitt 6.1) dahin gehend optimiert werden kann, indem die Kopie nicht bei jeder Zustandsänderung

aktualisiert wird, sondern lediglich bei einem lesenden Zugriff der nativen App. Mögliche Hindernisse sind hierbei jedoch die Asynchronität der Methoden von nativen Modulen.

Literaturverzeichnis

[1] *Mobile Operating System Market Share Worldwide*, Statcounter, Okt. 2023. [Online]. Verfügbar unter: https://gs.statcounter.com/os-market-share/mobile/worldwide/

[2] A. Khandeparkar, R. Gupta, und B. Sindhya, "An Introduction to Hybrid Platform Mobile Application Development," *International Journal of Computer Applications*, Ausgabe 118, Nr. 15, ff. 31–33, Mai 2015, Herausgeber: Foundation of Computer Science (FCS). [Online]. Verfügbar unter: https://www.ijcaonline.org/archives/volume118/number15/20824-3463

[3] S. Laningham, G. Booch, L. Nackman, und W. Royce, "developerWorks Interviews: Booch, Nackman, and Royce on IBM Rational ...," Feb. 2008. [Online]. Verfügbar unter: https://archive.ph/OO8al

[4] *Brownfield vs. Greenfield Development: What's the Difference in Software?*, Synoptek. [Online]. Verfügbar unter: https://synoptek.com/insights/it-blogs/greenfield-vs-brownfield-software-development/

[5] M. Greiler, M.-A. Storey, und A. Noda, "An Actionable Framework for Understanding and Improving Developer Experience," *IEEE Transactions on Software Engineering*, Ausgabe 49, Nr. 4, ff. 1411–1425, Apr. 2023, name der Konferenz: IEEE Transactions on Software Engineering.

[6] G. Peal, "React Native at Airbnb," Jul. 2023. [Online]. Verfügbar unter: https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c

[7] ——, "React Native at Airbnb: The Technology," Jul. 2023. [Online]. Verfügbar unter: https://medium.com/airbnb-engineering/react-native-at-airbnb-the-technology-dafd0b43838

[8] *Managing your app's life cycle*, Apple, Sep. 2023. [Online]. Verfügbar unter: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle

[9] *The activity lifecycle*, Android, Apr. 2023. [Online]. Verfügbar unter: https://developer.android.com/guide/components/activities/activity-lifecycle

[10] *Getting started*, React Native Reanimated, Sep. 2023. [Online]. Verfügbar unter: https://docs.swmansion.com/react-native-reanimated/docs/fundamentals/getting-started/

[11] *JavaScript Environment*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/javascript-environment.html

[12] *Bundle | Android Developers*, Android, Jun. 2023. [Online]. Verfügbar unter: https://developer.android.com/reference/android/os/Bundle

[13] *Parcelable | Android Developers*, Android, Sep. 2023. [Online]. Verfügbar unter: https://developer.android.com/reference/android/os/Parcelable

[14] F. M. Dörr, "Vergleich zwischen React Native und Progressive Web Apps für die Verwendung auf mobilen Endgeräten," 2023. [Online]. Verfügbar unter: https://git.thm.de/fmdr54/hauptseminar/-/raw/main/hauptseminar.pdf

[15] *Material Design for Android*, Android, Jun. 2023. [Online]. Verfügbar unter: https://developer.android.com/develop/ui/views/theming/look-and-feel

[16] *Performance Overview*, React Native, Nov. 2022. [Online]. Verfügbar unter: https://reactnative.dev/docs/performance#running-in-development-mode-devtrue

[17] S. Alpert, *State of React Native 2018*, React Native, Jun. 2018. [Online]. Verfügbar unter: https://reactnative.dev/blog/2018/06/14/state-of-react-native-2018

[18] *Using List Views*, React Native, Jan. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/using-a-listview

[19] *VirtualizedList*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/virtualizedlist

[20] *FlatList*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/flatlist

[21] Nils Hartmann und Oliver Zeigermann, *React : Grundlagen, fortgeschrittene Techniken und Praxistipps – mit TypeScript und Redux*. Heidelberg, Baden-Württemberg, Deutschland: dpunkt.verlag, 2019, Ausgabe 2. [Online]. Verfügbar unter: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2325266&site=ehost-live

[22] E. Elrom, *React and Libraries : Your Complete Guide to the React Ecosystem*. Berkeley, CA, USA: Apress L. P., 2021. [Online]. Verfügbar unter: http://ebookcentral.proquest.com/lib/thm/detail.action?docID=6511510

[23] *Stack Overflow Developer Survey 2023*, Stack Overflow. [Online]. Verfügbar unter: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023

[24] V. Gopinath, "ReactJS Tutorial - 1 - Introduction," Okt. 2018. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=QFaFIcGhPoM&list=PLC3y8-rFHvwgg3vaYJgHGnModB54rxOk3

[25] ——, "React TypeScript Tutorial - 1 - Introduction," Sep. 2021. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=TiSGujM22OI&list=PLC3y8-rFHvwi1AXijGTKM0BKtHzVC-LSK

[26] ——, "React Hooks Tutorial - 1 - Introduction," Mai 2019. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=cF2lQ_gZeA8&list=PLC3y8-rFHvwisvxhZ135pogtX7_Oe3Q3A

[27] *Components and Props*, React, Jul. 2023. [Online]. Verfügbar unter: https://legacy.reactjs.org/docs/components-and-props.html

[28] *Component*, React, Aug. 2023. [Online]. Verfügbar unter: https://react.dev/reference/react/Component

[29] *Introducing JSX*, React, Dez. 2022. [Online]. Verfügbar unter: https://reactjs.org/docs/introducing-jsx.html

[30] *JSX*, Meta. [Online]. Verfügbar unter: https://facebook.github.io/jsx/

[31] *Passing Props to a Component*, React, Jul. 2023. [Online]. Verfügbar unter: https://react.dev/learn/passing-props-to-a-component

[32] *Rendering Lists*, React, Aug. 2023. [Online]. Verfügbar unter: https://react.dev/learn/rendering-lists

[33] *Arrow function expressions - JavaScript | MDN*, Mozilla, Aug. 2023. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[34] *Your First Component*, React, Aug. 2023. [Online]. Verfügbar unter: https://react.dev/learn/your-first-component

[35] *Reusing Logic with Custom Hooks*, React, Jul. 2023. [Online]. Verfügbar unter: https://react.dev/learn/reusing-logic-with-custom-hooks

[36] *Built-in React Hooks*, React, Jul. 2023. [Online]. Verfügbar unter: https://react.dev/reference/react

[37] *Virtual DOM and Internals*, React, Dez. 2022. [Online]. Verfügbar unter: https://reactjs.org/docs/faq-internals.html

[38] *Rendering Elements*, React, Okt. 2023. [Online]. Verfügbar unter: https://legacy.reactjs.org/docs/rendering-elements.html

[39] *Render and Commit*, React, Okt. 2023. [Online]. Verfügbar unter: https://react.dev/learn/render-and-commit

[40] *Reconciliation*, React, Okt. 2023. [Online]. Verfügbar unter: https://legacy.reactjs.org/docs/reconciliation.html

[41] *Core Components and Native Components*, React Native, Dez. 2022. [Online]. Verfügbar unter: https://reactnative.dev/docs/intro-react-native-components

[42] *Flutter architectural overview*, Flutter, Aug. 2023. [Online]. Verfügbar unter: https://docs.flutter.dev/resources/architectural-overview

[43] *Core Concepts | Ionic Documentation*, Ionic, Aug. 2023. [Online]. Verfügbar unter: https://ionicframework.com/docs/core-concepts/fundamentals

[44] *Fabric*, React Native, Okt. 2022. [Online]. Verfügbar unter: https://reactnative.dev/architecture/fabric-renderer

[45] *Out-of-Tree Platforms*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/out-of-tree-platforms

[46] *Glossary · Host View Tree (and Host View)*, React Native, Dez. 2022. [Online]. Verfügbar unter: https://reactnative.dev/architecture/glossary

[47] E. Sjölander, "Yoga: A cross-platform layout engine," Dez. 2016. [Online]. Verfügbar unter: https://engineering.fb.com/2016/12/07/android/yoga-a-cross-platform-layout-engine/

[48] R. Nabors, *Meet Hermes, a new JavaScript Engine optimized for React Native*, React Native, Jul. 2019. [Online]. Verfügbar unter: https://reactnative.dev/blog/2019/07/17/hermes

[49] *Prerequisites for Applications*, React Native, Dez. 2022. [Online]. Verfügbar unter: https://reactnative.dev/docs/new-architecture-app-intro

[50] *View*, React Native. [Online]. Verfügbar unter: https://reactnative.dev/docs/view

[51] R. Pancholi, "Fabric Architecture-React Native," Mär. 2022. [Online]. Verfügbar unter: https://medium.com/mindful-engineering/fabric-architecture-react-native-a4f5fd96b6d2

[52] *Native Modules Intro*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/native-modules-intro

[53] *Turbo Native Modules*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/the-new-architecture/pillars-turbomodules

[54] *Setting up the development environment*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/environment-setup

[55] *iOS Native Modules*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/native-modules-ios

[56] *Android Native Modules*, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/native-modules-android

[57] *Why a New Architecture*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/the-new-architecture/why

[58] N. Corti und F. M. Dörr, "Usage of Turbo Native Modules within existing iOS and Android apps · Discussion #142," Jul. 2023. [Online]. Verfügbar unter: https://github.com/reactwg/react-native-new-architecture/discussions/142

[59] *C++ Turbo Native Modules*, React Native, Apr. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/next/the-new-architecture/cxx-cxxturbomodules

[60] *CMakeLists.txt facebook/react-native*, React Native. [Online]. Verfügbar unter: https://github.com/facebook/react-native/blob/cc059bf6aae06c9d14331c392cccfd7ff9c05670/packages/react-native/ReactAndroid/cmake-utils/default-app-setup/CMakeLists.txt

[61] *OnLoad.cpp facebook/react-native*, React Native, Sep. 2023. [Online]. Verfügbar unter: https://github.com/facebook/react-native/blob/cc059bf6aae06c9d14331c392cccfd7ff9c05670/packages/react-native/ReactAndroid/cmake-utils/default-app-setup/OnLoad.cpp

[62] *iOS Native UI Components*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/native-components-ios

[63] *ProgressViewIOS*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/progressviewios

[64] *ProgressBarAndroid*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/progressbarandroid

[65] *RCTViewManager.h · facebook/react-native*, React Native, Feb. 2015. [Online]. Verfügbar unter: https://github.com/facebook/react-native/blob/714b502b0c7a5f897432dbad388c02d3b75b4689/packages/react-native/React/Views/RCTViewManager.h

[66] *Android Native UI Components*, React Native, Jun. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/native-components-android

[67] *ViewManager.java · facebook/react-native*, React Native, Sep. 2015. [Online]. Verfügbar unter: https://github.com/facebook/react-native/blob/585057d7468b5ae8844fa8210df7ad1f8e0ae1e8/packages/react-native/ReactAndroid/src/main/java/com/facebook/react/uimanager/ViewManager.java

[68] R. Cipolleschi, "New Renderer Interop Layer · Discussion #135." [Online]. Verfügbar unter: https://github.com/reactwg/react-native-new-architecture/discussions/135

[69] *Autolinking*, React Native, Jan. 2023. [Online]. Verfügbar unter: https://github.com/react-native-community/cli/blob/main/docs/autolinking.md

[70] Swiftify und A. Madsen, "How many apps use Swift in 2022?" Sep. 2022. [Online]. Verfügbar unter: https://blog.swiftify.com/how-many-apps-use-swift-in-2022-cd325935f0d3

[71] *Build Better Apps with Kotlin*, Android, Aug. 2023. [Online]. Verfügbar unter: https://developer.android.com/kotlin/build-better-apps

[72] *Integration with Existing Apps*, React Native, Mär. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/integration-with-existing-apps

[73] K. Morris, *Infrastructure as Code*, 2zweite ed. Sebastopol, CA, USA: O'Reilly, Dez. 2020. [Online]. Verfügbar unter: https://www.thoughtworks.com/insights/books/infrastructure-as-code-2nd-edition

[74] R. Potvin und J. Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM*, Ausgabe 59, ff. 78–87, Jul. 2016. [Online]. Verfügbar unter: https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

[75] *CocoaPods Guides*, CocoaPods. [Online]. Verfügbar unter: https://guides.cocoapods.org

[76] *react_native_pods.rb · facebook/react-native*, React Native, Okt. 2023. [Online]. Verfügbar unter: https://github.com/facebook/react-native/blob/main/packages/react-native/scripts/react_native_pods.rb

[77] *Build a responsive UI with ConstraintLayout*, Android, Aug. 2023. [Online]. Verfügbar unter: https://developer.android.com/develop/ui/views/layout/constraint-layout

[78] R. Cipolleschi und F. M. Dörr, "Brownfield usage of Fabric with Swift · Discussion #143." [Online]. Verfügbar unter: https://github.com/reactwg/react-native-new-architecture/discussions/143

[79] *Enabling Fabric on iOS*, React Native, Mär. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/0.70/new-architecture-app-renderer-ios

[80] B. Kaszubowski, *Enabling Fabric on Android*, React Native, Mär. 2023. [Online]. Verfügbar unter: https://reactnative.dev/docs/0.70/new-architecture-app-renderer-android

[81] *Appropriate Uses For SQLite*, SQLite, Sep. 2023. [Online]. Verfügbar unter: https://www.sqlite.org/whentouse.html

[82] M. Faiz und U. Shanker, "Data synchronization in distributed client-server applications," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, Mär. 2016, ff. 611–616.

[83] *Earth Meteorite Landings JSON*, NASA, Sep. 2023. [Online]. Verfügbar unter: https://data.nasa.gov/resource/y77d-th95.json

[84] *The WebSocket API (WebSockets) - Web APIs | MDN*, Mozilla, Sep. 2023. [Online]. Verfügbar unter: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[85] E. Gamma, R. Helm, R. Johnson, und J. Vlissides, *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995, ab p. 233. [Online]. Verfügbar unter: http://archive.org/details/designpatternsel00gamm

[86] V. Sarcar, "Command Patterns," in *Java Design Patterns: A tour of 23 gang of four design patterns in Java*, V. Sarcar, Ed. Berkeley, CA: Apress, 2016, ff. 53–57. [Online]. Verfügbar unter: https://doi.org/10.1007/978-1-4842-1802-0_9

[87] R. Lämmel und S. Peyton Jones, "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming," in *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, Ausgabe 38, Mär. 2003, ff. 26–37.

Glossar

Boilerplate

Code, der häufig wiederholt werden muss, in großen Teilen aber identisch mit bereits geschriebenem Code ist. Generatoren für Boilerplate-Code erleichtern die Arbeit dabei enorm. [87]

Brownfield

Eine *Brownfield*-Entwicklung beschreibt die Integration einer neuen Technologie in ein bereits bestehendes Projekt. Die alte Implementierung kann dabei teilweise weiterverwendet werden. [3, 4]

Fork

Ein Fork im Umfeld von Git ist eine Abgabelung vom Haupt-Repository. Ein Fork ist somit eine Kopie eines Repositories zu einem bestimmten Zeitpunkt (Commit). Auf diesem Fork können Änderungen am Repository vorgenommen werden, ohne Schreibrechte auf dem Haupt-Repository zu haben. Forks können zu einem späteren Zeitpunkt wieder in das Haupt-Repository überführt werden, wenn der Autor des Haupt-Repositories zustimmt (Merge/Pull-Request).

Greenfield

Eine *Greenfield*-Entwicklung beschreibt eine komplette Neuentwicklung. Dabei wird die alte Implementierung vollumfänglich ersetzt. [3, 4]

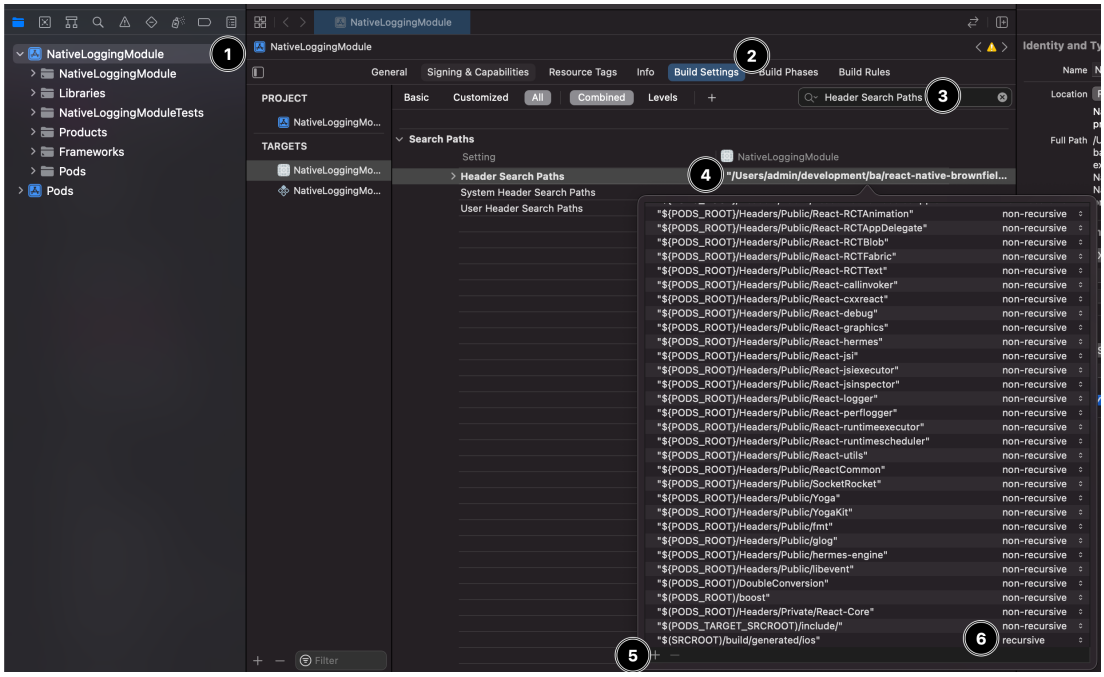
Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

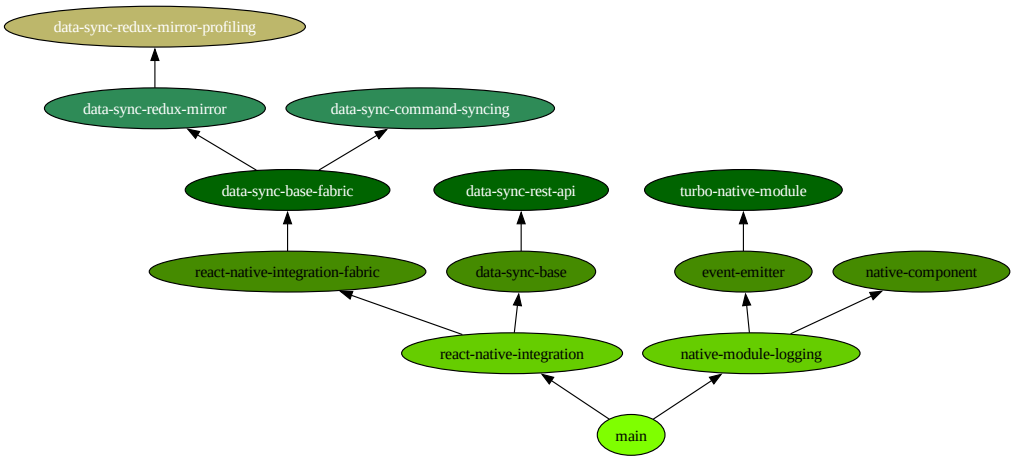
A Bilder

Abbildung A.1: Vorgehen zum Hinzufügen eines *Header Search Paths*



Die Ziffern zeigen die Reihenfolge der Navigation, um einen neuen *Header Search Path* hinzuzufügen.

Abbildung A.2: Grafische Darstellung der verfügbaren *Branches*.



Baumstruktur der *Branches* des Repositories `https://github.com/flokol120/react-native-brownfield-examples` (abgerufen am 26. Oktober 2023).

B Tabellen

Tabelle B.1: Übersicht der *Branches* von `https://github.com/flokol120/react-native-brownfield-examples` (abgerufen am 26. Oktober 2023).

Branch	Kapitel	Änderungen im Vergleich zum vorherigen Branch
main	-	-
native-module-logging	4.2	main..native-module-logging
event-emitter	4.3	native-module-logging..event-emitter
turbo-native-module	4.4	event-emitter..turbo-native-module
native-component	4.5	native-module-logging..native-component
react-native-integration	5.1 - 5.5	main..react-native-integration
react-native-integration-fabric	5.6	react-native-integration..react-native-integration-fabric
data-sync-base	-	react-native-integration..data-sync-base
data-sync-base-fabric	-	data-sync-base..data-sync-base-fabricreact-native-integration..data-sync-base-fabric
data-sync-redux-mirror	6.1	data-sync-base-fabric..data-sync-redux-mirror
data-sync-redux-mirror-profiling	6.1	data-sync-redux-mirror..data-sync-redux-mirror-profiling
data-sync-rest-api	6.2	data-sync-base..data-sync-rest-api
data-sync-command-syncing	6.3	data-sync-base-fabric..data-sync-command-syncing

C Quellcode

```
export abstract class Command {
    ...

    abstract doExecute(proceed: (result?: any) => void, cancel: (reason?:
    ↪  any) => void): void;

    protected executeCommandPromise() {
        const promise = () => new Promise<any>((resolve, reject) =>
        ↪  this.doExecute(resolve, reject));
        switch (this.executionMode) {
            case "ANIMATION_FRAME":
                return requestAnimationFramePromise(promise);
            case "AFTER_INTERACTIONS":
                return runAfterInteractions(promise);
            default:
                return promise();
        }
    }

    public async execute(): Promise<any> {
        this._executionState = CommandState.PROCESSING;
        console.debug(`${this.name} >> ${this._executionState}`);
        try {
            const result = await this.executeCommandPromise();
            this._executionState = CommandState.DONE;
            console.debug(`${this.name} >> DONE`);
            return result;
        } catch (e) {
            console.debug(`${this.name} --> cancelled with exception:
            ↪  ${e}`);
            this._executionState = CommandState.CANCELLED;
            // re-throw to reject Promise
            throw e;
        }
    }
    ...
}
```

(vgl. https://github.com/flokol120/react-native-brownfield-examples/blob/data-sync-command-syncing/example-integration/react-native/command/common/CommandController.ts, abgerufen am 26. Oktober 2023)

Quellcode C.1: Beispielhafte Umsetzung des *Command Patterns* in TypeScript.