

BACHELORTHESIS
Florian Matzeit

Hybride Appentwicklung

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Florian Matzeit

Hybride Appentwicklung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Michael Schäfers

Eingereicht am: 07. April 2020

Florian Matzeit

Thema der Arbeit

Hybride Appentwicklung

Stichworte

React Native, hybrid, Expo, JavaScript, Framework, Design, Backendkommunikation, MVC

Kurzzusammenfassung

Portierung nativer Android- und iOS-Applikationen zu einer hybriden React Native-Applikation anhand eines praxisnahen Beispiels

Florian Matzeit

Title of Thesis

Hybrid app development

Keywords

React Native, hybrid, Expo, JavaScript, framework, design, backendcommunication, MVC

Abstract

Porting native Android- and iOS applications to a hybrid React Native application by using a practical example

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Tabellenverzeichnis	ix
Abkürzungen	x
1 Einleitung	1
1.1 Mobile Software und Plattformen	1
1.2 Motivation	3
1.3 Mein o2-Applikation Software (zu deutsch: Anwendungssoftware) (App)	4
1.4 Ziel dieser Arbeit	5
1.4.1 Aktuelle Projektsituation	5
1.4.2 Ziel	5
2 Grundlagen	6
2.1 Native Apps	7
2.2 Hybride Appentwicklung	8
2.2.1 Ionic	8
2.2.2 React Native	9
2.2.3 Xamarin	12
2.3 Warum React Native?	13
2.4 JavaScript	14
2.4.1 Interpreter	16
2.5 Fetch-API	17
2.5.1 Promise-Objekt	18
2.5.2 Callback-Funktionen	18
2.6 Test Driven Development	19
2.6.1 Modultests	19
2.6.2 Integrationstests	21

3	Anforderungsanalyse	23
3.1	Ist-Situation	24
3.2	Funktionale und nichtfunktionale Anforderungen	25
3.2.1	Nichtfunktionale Anforderungen	25
3.2.2	Funktionale Anforderungen	26
3.3	Transfersituation	27
4	Methodik	28
4.1	Architektur	29
4.1.1	Model-View-Controller	29
4.1.2	Aufbau des Frontends	30
4.1.3	Backend	31
4.2	Softwarestack	32
4.2.1	Entwicklungsumgebung	32
4.2.2	Expo-Client	33
4.2.3	JEST	36
5	Realisierung	37
5.1	Testgetriebene Entwicklung	37
5.2	Backendkommunikation	39
5.2.1	Senden der Login Anfrage mittels Fetch-API	40
5.2.2	Kommunikation mit dem zentralen Backend mittels Fetch-API	42
5.3	Aufbau des User Interfaces	43
5.3.1	Design-Komponenten	43
5.3.2	Screendesign-Vergleich zwischen beiden Plattformen	46
5.4	Probleme im Laufe des Entwicklungsprozesses	47
5.4.1	Styling-Eigenschaften	47
5.5	Aufbau nach Unified Modeling Language (UML)	49
5.6	Deployment	50
5.6.1	Vorraussetzungen	50
5.6.2	Generierung nativen Codes	50
5.6.3	Bauen nativer Apps	51
6	Systemtests	52
6.1	Anwendungsszenario	52
6.1.1	Use Case Sequenzdiagramm	52
6.1.2	App-Durchlauf	53

6.2	Fehlerszenarien	58
6.2.1	Fehlende Internetverbindung	58
6.2.2	Nicht unterstützter Nutzer	60
6.2.3	Fehler im Backendsystem	61
6.2.4	Falscheingaben des Anwenders	63
7	Fazit	64
7.1	Zusammenfassung	64
7.1.1	Vorteile einer hybriden Lösung mit React Native	65
7.1.2	Einschränkungen	66
7.1.3	Aussicht	66
	Literaturverzeichnis	68
	A Anhang	71
	Glossar	72
	Selbstständigkeitserklärung	74

Abbildungsverzeichnis

1.1	Betriebssystem als Schnittstelle zwischen Soft- und Hardware (Quelle: [27])	2
1.2	Verteilung der Marktanteile mobiler Betriebssysteme in Deutschland (Stand: Mai 2019 Quelle: [4])	3
2.1	Die <i>Bridge</i> fungiert als Übersetzungseinheit zwischen Quellcode und nativer Umgebung (vgl. [11])	9
2.2	Klassische Vererbung im Vergleich gegen prototypische Vererbung (vgl. [25])	15
2.3	Unterschiede Übersetzungsabfolge zwischen Interpreter und Compiler (vgl. [22])	16
2.4	Unterschiedliche Zustände eines Promise-Objektes (Quelle: [21])	19
3.1	Entwickeln wir das Richtige? (Quelle: [1])	23
4.1	Architekturentscheidungen treffen als Teil des Softwareentwicklungszyklus	28
4.2	Model View Controller (MVC) als <i>Divide-and-Conquer</i> Paradigma	29
4.3	Beschreibt die Architektur als Komponentendiagramm nach MVC Ansatz	30
4.4	Beschreibt den Backendaufbau als Komponentendiagramm	31
4.5	Änderungen am Code werden vom React Native Packager direkt an die Expo-App übertragen (vgl.[12])	33
4.6	Expo gibt bei Kompilierungs- sowie Laufzeitfehlern den Stack trace an . . .	35
5.1	Ausgabe des Ergebnisses aller fünf Testmuster der getesteten Funktion <i>funcGetUsageDivisorForMobileData</i>	39
5.2	Die Komponente <i>Toggle</i> wird plattformtypisch dargestellt. Sie bildet ein natives Modul und wird aus den jeweiligen Bibliotheken übersetzt	46
5.3	Verwendung eines festen Wertes für das Attribut <i>fontWeight</i> . Die Beschriftung der angezeigten Elemente wird lediglich innerhalb der Android Umgebung fett markiert	47

5.4	Beim Absturz des Expo-Clients wird diese Meldung auf der Konsole angezeigt. Ein Neustart des Clients ist dann erforderlich	48
5.5	short	49
6.1	Beschreibt den positiv verlaufenden Anwendungsfall. Es treten keine Fehler auf	52
6.2	Der Kunde gibt seine Logindaten direkt in der App ein (hier Android) . .	53
6.3	Die App signalisiert dem Verbraucher den laufenden Prozess	54
6.4	Die Positionierung des Seitentitels unterscheidet sich ebenfalls plattformcharakteristisch	55
6.5	Über die Wischgeste <i>Scrollen</i> gelangt man zu den Informationen über den aktuellen Tarif	56
6.6	Durch Betätigen der Schaltfläche <i>Daten außerhalb EU</i> erhält man Informationen über den Auslandsverbrauch	57
6.7	Eine nicht bestehende Internetverbindung wird dem Anwender direkt mitgeteilt	58
6.8	Die App prüft regelmäßig auf eine bestehende Internetverbindung	59
6.9	Bricht die Internetverbindung des Anwenders nach dem Login ab, wird ihm ebenfalls die Information angezeigt.	59
6.10	Ist der Nutzer kein Postpaidkunde wird sein Vertragsmodell von der App noch nicht unterstützt	60
6.11	Die angezeigte Nachricht ist nativ	60
6.12	Das zentrale Backend agiert fehlerhaft	61
6.13	In diesem Fall kann das Backend aufgrund einer fehlerhaften Url nicht erreicht werden. Daher wird ein 404 Fehlercode angezeigt	61
6.14	Der Fehler wird durch die verschiedenen Systeme kommuniziert	62
6.15	Information über die fehlgeschlagene Anfrage der Nutzungsdaten	62
6.16	Bei fehlgeschlagenem Login Versuch werden die Informationen aus dem Login-Backend an den Anwender weitergeleitet	63
6.17	Der Anwender erhält Informationen über seinen fehlgeschlagenen Login Versuch	63

Tabellenverzeichnis

2.1	Unterscheidung nativer Entwicklung zw. Android und iOS (vgl. [13]) . . .	7
2.2	Gegenüberstellung der Attribute Xamarin vs. React Native (vgl. [5]) . . .	13

Abkürzungen

.apk Android Package.

.ipa Iphone Application.

AOT Ahead-Of-Time.

API Application Programming Interface.

App Applikation Software (zu deutsch: Anwendungssoftware).

JIT Just-In-Time.

JSON JavaScript Object Notation.

MVC Model View Controller.

UI User Interface.

UML Unified Modeling Language.

UX User Experience (zu deutsch: Benutzererfahrung).

WHATWG Web Hypertext Application Technology Working Group.

XML Extensible Markup Language.

1 Einleitung

Diese Arbeit umfasst die Thematik der Entwicklung portabler Anwendungssoftware. Der Fokus liegt hierbei auf dem dafür vorgesehenen Framework **React Native**. Es wird anhand eines praxisnahen Beispiels erleuchtet, welche Vor- und Nachteile sich in der Umsetzung ergeben, sowie auf mögliche Einschränkungen und Schwierigkeiten in Hinblick auf die Realisierung des Gesamtprojekts eingegangen.

1.1 Mobile Software und Plattformen

Mobile Applikationen nutzen wir täglich und überall. Sie sind ein nützlicher Begleiter und helfen uns miteinander zu kommunizieren, bestimmte Teile unseres Lebens zu organisieren, komplexe Aufgaben zu lösen, oder uns schlicht zu unterhalten.

Ganz egal, auf welchen Endgerätetypen sie ausgeführt werden, ob Smartphone, Tablet, Laptop oder Notebook, die Portabilität steht hier im entscheidenden Fokus. Doch wird in der Thematik der Entwicklung von Anwendungssoftware nicht immer zwangsläufig an Portabilität gedacht.

Zunächst unterscheidet man zwischen Anwendungssoftware also Applikationen, die wir täglich gebrauchen, um bestimmte Aufgaben mit ihrer Hilfe zu erfüllen und Betriebssystemsoftware, auch bezeichnet als Plattform, die als Basis für die Umgebung der Anwendungssoftware agiert.

Betriebssystem

Wie in **Abb. 1.1** gezeigt, arbeitet ein Betriebssystem vage gesprochen als Schnittstelle zwischen Soft- und Hardware. Es verwaltet darüber hinaus die Systemressourcen eines Endgeräts wie Arbeitsspeicher, Festplatten, sowie Ein- und Ausgabegeräte und stellt diese den Anwendungsprogrammen zur Verfügung. Jede Anwendungssoftware braucht eine Plattform, also ein Betriebssystem, auf dem diese ausgeführt werden kann. Möchte man also eine Applikation entwickeln, so sollte man sich zunächst Gedanken machen, auf welcher /n Plattform /en diese lauffähig sein soll. Man spricht von der Kompatibilität mit der (Plattform-) Umgebung.

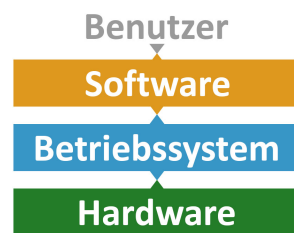


Abbildung 1.1: Betriebssystem als Schnittstelle zwischen Soft- und Hardware (Quelle: [27])

1.2 Motivation

Jedoch ist es möglich, auch Anwendungssoftware mit portablen Eigenschaften zu entwickeln.

Das bedeutet, sie besitzt die Fähigkeit auf unterschiedlichen Plattformen ohne weiteren Aufwand ausführbar sein, welches Entwicklungsprozesse vereinfachen und zu geringerem Projektaufwand führen kann.

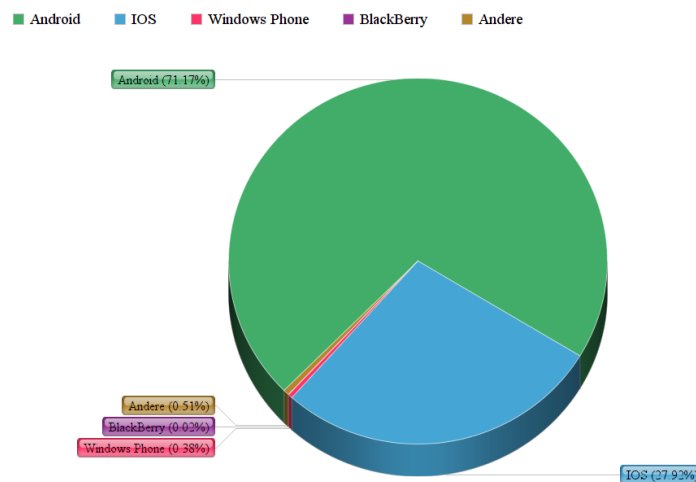


Abbildung 1.2: Verteilung der Marktanteile mobiler Betriebssysteme in Deutschland (Stand: Mai 2019 Quelle: [4])

Abbildung **1.2** zeigt deutlich, welche Plattformen zur Zeit den Markt dominieren. Um eine möglichst hohe Verteilung seines Produktes erzielen zu können, macht es Sinn, die Marktverteilung im Auge zu behalten. In dieser Arbeit wird die partielle Entwicklung eines Prototypen durchgeführt, der auf den beiden populärsten Plattformen lauffähig ist.

Damit nicht an mehreren Projekten parallel gearbeitet werden muss, um beide Systeme zu erreichen, gibt es Frameworks für die Implementierung hybrider Applikationen. Diese werden, nicht zuletzt aufgrund ihrer in den letzten Jahren gemachten Fortschritte, immer interessanter.

Idealerweise merkt der Anwender gar nicht, dass die App, die er gerne und regelmäßig nutzt, hybrid geschrieben wurde, da sie über dieselben Merkmale einer nativ (siehe Abschnitt **2.1**) geschriebenen Applikation verfügt.

1.3 Mein o2-App

Die mobile Mein o2-App dient Kunden deutschlandweit zur Vertragsverwaltung. Mit ihrer Hilfe ist es dem Endverbraucher möglich, seine aktuellen Tarifinformationen und Datenverbrauch im Auge zu behalten.

Die Abfrage und Buchung optionaler Tarifierweiterungen ist direkt über die App möglich.

Des Weiteren können mit Hilfe der App die Accountdaten des Kunden von diesem abgefragt und gegebenenfalls bearbeitet werden. Dies beschreibt die Kernkompetenz der Anwendung.

Verschiedene Vertragstypen, die über die App verwaltet werden können, sind folgende:

Postpaid

Der Kunde stimmt einem Vertrag mit fester Laufzeit zu. Am Ende eines Abrechnungszeitraums werden die Kosten seines Verbrauchs (mobile Daten, Telefonminuten, SMS etc.) zusammengeführt und er bekommt über eine Rechnung die Aufforderung diese nach [*post*] Nutzung zu begleichen.

Prepaid

Bei diesem Modell wird es dem Kunden ermöglicht, seine SIM-Karte vorab in einer bestimmten Höhe mit Guthaben aufzuladen, welches dann der freien Nutzung zur Verfügung steht. Somit wird hier das Geld vor [*pre*] der Nutzung bereits gezahlt.

My Handy

Für Kunden interessant, die lediglich Interesse an einem Endgerät besitzen, dessen Preis über dieses Vertragsmodell nach einer Anzahlung in monatlichen Raten abzuleisten ist. Auch bezeichnet als *Hardware Only*.

Um die oben aufgeführten Anwendungsszenarien durchführen zu können, muss ein neuer Kunde zunächst eine Registrierung durchführen. Anschließend folgt die Aufforderung für den Login, mit den bei der Registrierung angelegten Daten.

In dieser Arbeit betrachten wir den Anwendungsfall eines bereits registrierten Postpaid-Kunden, der seine Verbrauchsdaten abfragen möchte, welche ihm als Information direkt auf dem Startbildschirm angezeigt werden, zu dem nach erfolgreichem Login navigiert wird (siehe Abschnitt **6.1.2**).

1.4 Ziel dieser Arbeit

1.4.1 Aktuelle Projektsituation

Nach aktuellem Stand wird der Entwicklungsprozess der Mein o2-App parallel von zwei getrennten Entwicklerteams durchgeführt. Jedes Entwicklerteam ist für die lauffähige Version, sowie Updates, Einbau neuer Funktionen und Fehlerbehebungen der Applikation auf einer bestimmten Plattform zuständig.

Beide Teams sind darüber hinaus verantwortlich, am Ende eines Releases eine Version ihrer plattformabhängigen Anwendung heraus zu bringen die, zueinander im Vergleich, ein einheitliches User Interface aufweisen und zugleich dem Anwender ein, für seine Plattform, vertrautes Design bieten.

1.4.2 Ziel

Das Ziel dieser Arbeit ist es, anhand einer partiellen Entwicklung der im vorigen Unterkapitel erläuterten Applikation (siehe Abschnitt 1.3) die Möglichkeit zu zeigen, ein Produkt zu entwickeln, welches parallel auf den Plattformen Android und iOS lauffähig ist.

Ferner soll durch diese Arbeit eine erste Abschätzung über die Realisierung des Gesamtprojektes, hinsichtlich des Aufwands und der Vor- und Nachteile, sowie den damit verbundenen langfristigen Zielen gesetzt werden.

Erstrebt wird der dauerhafte Zusammenschluss beider Entwicklerteams, die gemeinsam an einer Codebasis arbeiten können.

2 Grundlagen

In diesem Kapitel geht es um die Grundlagen der hybriden App Entwicklung mit Hilfe des React Native Frameworks. Ferner werden in Kurzform mögliche Alternativen vorgestellt und darauf eingegangen, warum gerade in diesem Projektkontext React Native verwendet wurde.

Darüber hinaus werden die Funktionen verschiedener Werkzeuge vorgestellt, die für die Entwicklung hybrider Applikationen mit React Native erforderlich sind.

Für den Einstieg in die Thematik wurde unter anderem auf folgendes Buch zurück gegriffen:

React Native. Native Apps parallel für Android und iOS entwickeln [13]

2.1 Native Apps

Eine Möglichkeit, mobile Applikationen zu entwickeln, ist diese nativ und an eine bestimmte Plattform gebunden zu implementieren. Dafür stellen Google und Apple ihre eigenen Frameworks und Entwicklungssprachen zur Verfügung.

Entwickelt man eine App nativ, so kann sie, je nach verwendeter Entwicklungsumgebung, entweder auf einem Android- oder auf einem iOS-Gerät ausgeführt werden, jedoch nicht auf beiden plattformübergreifend. Die Art und Weise der Frontenddesigngestaltung unterscheidet sich maßgeblich.

Wird im nativen Android Kontext unter anderem noch deklarativ mit dem Extensible Markup Language (XML)-Layout gearbeitet, stellt iOS dem Entwickler eigene grafische Tools in Form eines Interface Builders zur Verfügung.

Native Applikationen besitzen die Eigenschaft, aufgrund der zielgerichteten Entwicklung auf einer Plattform, äußerst performant zu sein. Darüber hinaus weisen sie ein für diese Plattform charakteristisches Design auf.

	Android	iOS
Programmiersprache	Java oder Kotlin	Objective-C oder Swift
Entwicklungsumgebung	Android Studio	Xcode
Unterstützte Betriebssysteme	Linux, macOS, Windows	macOS
User Interface (UI) Erstellung	deklarativ mit XML-Layout oder programmatisch mit Java, sowie grafische Tools in Android Studio	grafische Tools in Xcode (Interface Builder)

Tabelle 2.1: Unterscheidung nativer Entwicklung zw. Android und iOS (vgl. [13])

2.2 Hybride Appentwicklung

Anders als bei nativen Applikationen ist es ihren hybriden Gegenübern möglich, anhand einer Codebasis auf mehreren unterschiedlichen Plattformen mit äquivalentem Funktionsumfang ausführbar zu sein.

Der Vorteil gegenüber der nativen Entwicklung liegt maßgeblich in der Simplizität, mit mehr als bloß einer Plattform kompatibel zu sein. War es vorher erforderlich gleich mehrere Entwicklungsumgebungen, sowie Sprachkerne in eigenständigen Teams beherrschen zu müssen, ist es mit dem hybriden Ansatz möglich, das erforderliche Know-How eines plattformübergreifenden Projektes auf einen Sprachkern und einen Softwarestack zu beschränken.

Im Folgenden werden drei bekannte Technologien für die Entwicklung hybrider Anwendungssoftware vorgestellt, die alle den zuletzt erwähnten Vorteil mit sich bringen. Es gibt jedoch noch Weitere, die im Zuge des Projektkontextes mehr Sinn ergeben können.

2.2.1 Ionic

Ionic (vgl. [10]) ist ein Framework, welches auf die Nutzung von bekannten Web Technologie Standards, wie HTML, CSS und JavaScript für die Gestaltung von Frontendlösungen zurückgreift.

Neben dem gemeinsamen Ziel verfolgt Ionic darüber hinaus die Absicht, den Einstieg für Anwender mit Webentwicklungskennntnissen so einfach wie möglich zu machen und ein eigenes Ökosystem für ständige Updates, Fehlerbehebungen und Erweiterungen zu bieten. Jedoch liegt der Fokus von Ionic nicht darin, eine nativ wirkende Oberfläche designen zu können.

Im Grunde ist eine Applikation, die mit Ionic geschrieben wurde, eine reine Webapplikation, die in einem geschlossenen Container ausgeführt wird und dem unerfahrenen Benutzer nicht direkt als solche erscheinen soll.

2.2.2 React Native

Von Facebook entwickelt und im März 2015 veröffentlicht.

Seit Release **0.59** (März 2019) bringt React Native auch für Android Unterstützung für die 64-Bit Architektur heraus (vgl. [7]).

Im Gegensatz zu Ionic ermöglicht React Native (vgl. [13]) es dem Entwickler, native Apps plattformübergreifend zu entwickeln. Das bedeutet es handelt sich hierbei nicht um eine Webapplikation, die sich dem Anwender anhand ihrer Designkomponenten auch als solche erkennen lässt, vielmehr ist es mit diesem Framework möglich reale native Komponenten für das User Interface zu verwenden.

Wer also Wert darauf legt, dass sein Produkt die plattformspezifischen Charakteristika behält, für den ist React Native deutlich interessanter als das zuvor beschriebene Framework Ionic (siehe Abschnitt 2.2.1).

Doch wie funktioniert das genau? Die folgende Abbildung beschreibt, wie React Native dies möglich macht.

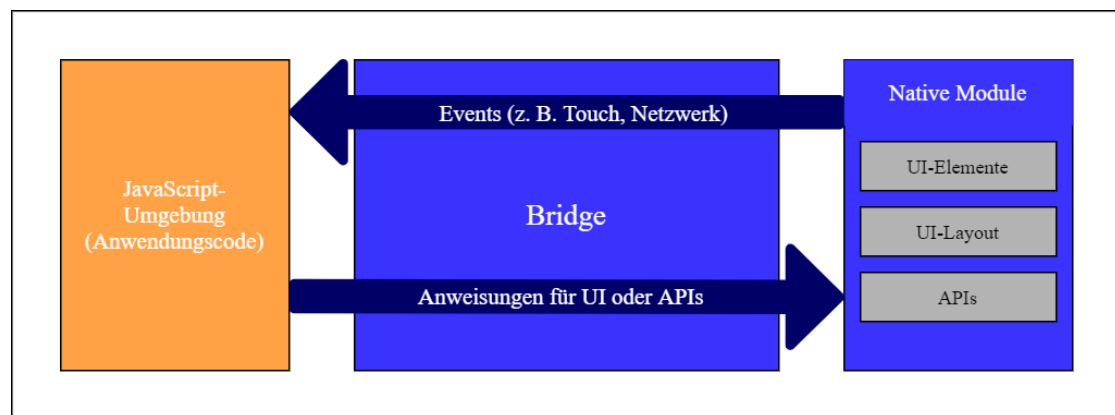


Abbildung 2.1: Die *Bridge* fungiert als Übersetzungseinheit zwischen Quellcode und nativer Umgebung (vgl. [11])

Wie in **Abb. 2.1** gezeigt, werden beim Erstellen von Designkomponenten im JavaScript-Quellcode native Module aus den jeweiligen Bibliotheken durch die Bridge „übersetzt“.

Dabei erfolgt die Kommunikation über die Bridge in Form von Nachrichten, die in der bekannten JavaScript Object Notation (JSON)-Struktur gehalten sind, welches ein Textformat für den Datenaustausch beschreibt und für den Menschen verständlicher zu lesen ist.

Damit sich ausführende Threads nicht gegenseitig blockieren, werden diese asynchron übertragen. Für den Entwickler entscheidend ist, dass dieser stetige Prozess im Hintergrund ausgeführt wird und von ihm somit nicht weiter beachtet werden muss, sondern lediglich von den Vorteilen profitieren braucht:

Neben dem großen Vorteil eine **native** App entwickeln zu können die, an einer Codebasis geschrieben, auf mehreren Plattformen lauffähig ist, bietet dieses Verfahren darüber hinaus die Möglichkeit vorgenommene Änderungen im Quellcode, ohne diesen kompilieren zu müssen, sofort einsehen zu können (siehe Abschnitt **4.2.2**).

Dafür wird der geänderte Programmabschnitt, oder das gesamte Programm erneut an die App übertragen. Dies spart viel Zeit in der Entwicklungsphase, was sich, je nach wachsender Projektgröße, immer deutlicher abzeichnet.

Einen weiteren, wichtigen Vorteil weist React Native in Sachen Performanz auf (vgl. [5]). Durch das Anwenden von *Just-In-Time (JIT)*-Kompilierung werden zur Laufzeit insbesondere dynamische Inhalte effizient übersetzt. So können zum Beispiel optimierte Versionen eines Codes für den verwendeten Datentypen erstellt werden.

```
1  function arraySum(arr) {
2    var sum = 0;
3    for (var i = 0; i < arr.length; i++) {
4      sum += arr[i];
5    }
6  }
```

Listing 2.1: Beispielcode einer Iteration in der numerische Werte summiert werden

Der Codeausschnitt **2.1** dient als Beispiel für die Steigerung der Effizienz durch Bestimmung des in der Iterationsschleife verwendeten Datentyps.

Durch den Fokus dieser Arbeit auf die Realisierung hybrider nativer Apps mit Hilfe von React Native, sind weitere Informationen bezüglich Funktionsweise im Abschnitt **4.2** aufgeführt.

2.2.3 Xamarin

Xamarin ist, wie React Native, für die hybride Entwicklung nativer Apps gedacht. Sie unterscheiden sich im Wesentlichen durch ihre Sprachumgebung (vgl. [2] [5]).

Im Gegensatz zu React Native, welches im iOS Umfeld auf Apples JavaScriptCore Framework (unter der Nutzung des JavaScript Interpreters) zurückgreifen muss, unterstützt Xamarin die *Ahead-Of-Time (AOT)*-Kompilierung, welche ermöglicht, Programmcode bereits vor der Laufzeit zu übersetzen.

Dies ist relevant, da Apples Betriebssystem iOS keine JIT-Kompilierung unterstützt.

Xamarin bringt das *Fast Deployment*-Feature für Android mit, durch welches Bauelemente der App direkt auf dem Gerätespeicher abgelegt werden, anstatt sie bei der Kompilierung jedes Mal direkt in die Android Package (.apk)-Datei einzubetten.

Dies reduziert die benötigte Zeit für die Verteilung der Applikation auf die Verbraucher nach Einbau neuer Funktionen.

2.3 Warum React Native?

Wie in Abschnitt 1.4 bereits erwähnt, ist es für dieses Projekt maßgebend, nicht auf native Designelemente verzichten zu müssen. Aus diesem Grund scheidet Ionic direkt aus.

Die Entscheidung liegt also zwischen React Native und Xamarin. Betrachten wir noch einmal die wesentlichen, für dieses Projekt interessanten Attribute dieser beiden Frameworks und stellen sie gegenüber.

		Xamarin	React Native
Programmiersprache		C#	JavaScript
Kompilierung	Android	JIT/AOT	JIT
	iOS	AOT	Interpreter
64-bit Support	Android	Ja	(seit März 2019)
	iOS	Ja	Ja
Programmiersprache	Android & iOS	Native Komponenten	Native Komponenten
Hot Swapping	Android	Nein	Hot Reloading
	iOS	Nein	Hot Reloading
Cold Swapping	Android	Fast Deployment	Live Reloading
	iOS	Nein	Live Reloading

Tabelle 2.2: Gegenüberstellung der Attribute Xamarin vs. React Native (vgl. [5])

Fazit

Da beide Frameworks die für das Projekt wesentlichen Attribute bieten, wird die Entscheidung letztendlich auf Basis der Höhe der Nachfrage des Kunden entschieden, die zu Gunsten von React Native ausfällt.

Das mag mit hoher Wahrscheinlichkeit auch daran liegen, dass der Einstieg in React Native gerade für Webentwickler sehr leicht ist.

2.4 JavaScript

JavaScript (vgl. [20]) bildet den Sprachkern, der von React Native verwendet wird.

Es ist eine Skriptsprache, was bedeutet, dass Programme, die mit ihr geschrieben sind durch einen Interpreter (siehe Abschnitt 2.4.1) ausgeführt werden.

Der Unterschied einer Skriptsprache zu einer reichen, beispielsweise objektorientierten Programmiersprache, wie Java liegt unter anderem in dem Verzicht auf umfangreiche Sprachelemente, deren Nutzen erst bei komplexeren Aufgaben zum Tragen kommt.

Der Vorteil, dass ein in JavaScript geschriebenes Programm auf jeder Umgebung ausgeführt werden kann, die über einen JavaScript Interpreter verfügt, macht diese Sprache besonders geeignet für Cross-Plattform basierende Anwendungen und Frameworks.

Sie ist multiparadigmatisch, besitzt auch Sprachelemente aus der objektorientierten Umgebung, wodurch sich bekannte Architekturmuster, wie zum Beispiel das MVC-Pattern (siehe Abschnitt 4.1.1), relativ gut umsetzen lassen.

Der objektorientierte Ansatz basiert in JavaScript nicht auf Klassen, jedoch wird dieser ab ECMAScript 6 mit einer Syntax ermöglicht, die auch bei klassenbasierten Sprachen üblich ist.

Die Skriptsprache bietet hierfür Objekt-Prototypen und -Templates, die über objektprototypische Vererbungsstrategien verfügen.

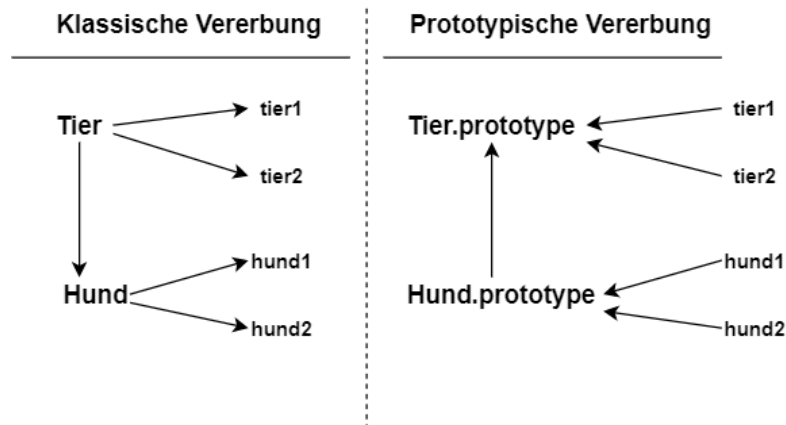


Abbildung 2.2: Klassische Vererbung im Vergleich gegen prototypische Vererbung (vgl. [25])

Abbildung 2.2 beschreibt das Delegationsmuster in JavaScript. Im Gegensatz zu dem klassischen Ansatz bilden die Instanziierungen aus JavaScript keine gesonderten Kopien, sondern jeweils eine Verlinkung auf den Prototypen, um dessen Verhalten anzunehmen. Dies wird im Ansatz rechts mit der entgegengesetzten Pfeilrichtung dargestellt.

Daher spricht man vom *Behavior Delegation Pattern*, welches weitestgehend als prototypische Vererbung in JavaScript bekannt ist.

Auch gibt es Unterschiede in der Semantik der Referenzierung auf ein Objekt, befindet man sich beispielsweise innerhalb eines Funktionsrumpfes:

Betrachten wir die Iterationsschleife aus Codeabschnitt 2.1, so würde man annehmen, dass unter der Verwendung von `this` innerhalb der Funktion das Objekt der darum liegenden Klasse referenziert wird.

Tatsächlich besitzt jede Funktion in JavaScript jedoch ihren eigenen Pointer, `this` referenziert somit auf die Funktion selber und würde `undefined` zurück geben, da dieser Wert zur Laufzeit nicht zugewiesen werden kann (vgl.[14]).

2.4.1 Interpreter

Ein Interpreter ist ein Programm, welches Anweisungen in einem vorgegebenen Format direkt ausführt. Er übersetzt jeweils eine Anweisung zur selben Zeit.

Im Vergleich zu einem Compiler verläuft die Analyse des Quellcodes schneller, jedoch geht die vollständige Übersetzung langsamer vonstatten.

Ein Interpreter arbeitet Speichereffizient, da der Präprozessor kein Objektcode mit zusätzlichen Referenzen erzeugt, welches keinen weiteren Speicherplatz bedarf.

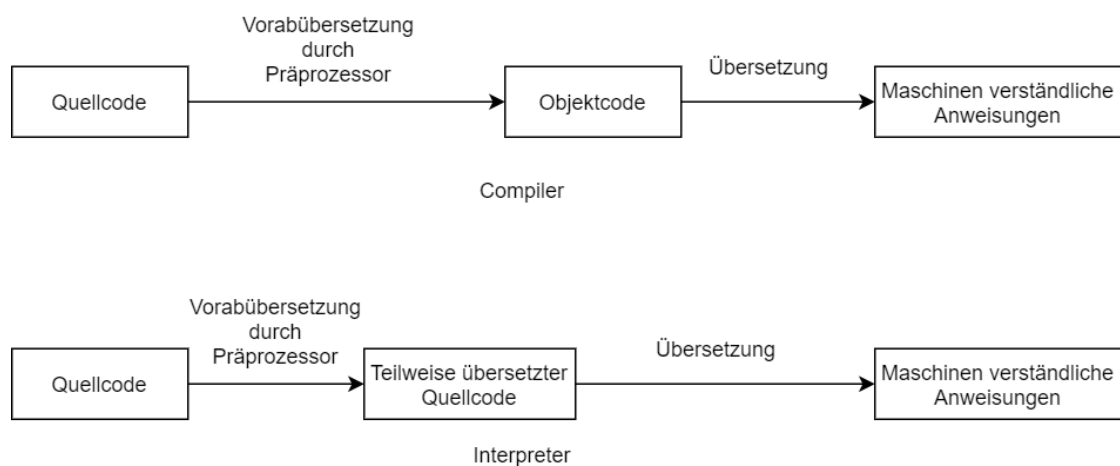


Abbildung 2.3: Unterschiede Übersetzungsabfolge zwischen Interpreter und Compiler (vgl. [22])

2.5 Fetch-API

Von der *Web Hypertext Application Technology Working Group (WHATWG)* entwickelt, bietet das Fetch Standard Application Programming Interface (API) (vgl. [15]), unter anderem auch in JavaScript, schon seit einigen Jahren eine erleichterte Alternative zum *XMLHttpRequest*, welcher in den meisten Browsern für das Abfragen von Daten zur Verfügung steht.

Das Hauptaugenmerk der Fetch Standard API ist darauf gerichtet, die Komplexität der Kommunikation mit einer entfernten Instanz für den Austausch von Daten möglichst gering zu halten. Durch seine Funktion `fetch` stellt JavaScript dem Entwickler diese API zur Verfügung.

Für das Format der transferierten Daten wird JSON als, im Vergleich zu XML, neuerer Standard verwendet.

Da diese Funktion in JavaScripts globalem Namensraum existiert, ist es nicht erforderlich, ein besonderes Modul zu importieren.

Es gilt zu beachten, dass `fetch` eine asynchrone Funktion ist, die ein Promise-Objekt (siehe Abschnitt **2.5.1**) zurückliefert, da es sich um einen Transfer von Daten über das Netzwerk handelt, bei dem weder deren Vollständigkeit, noch der genaue Zeitpunkt des Erhalts garantiert werden kann.

Möchte man also davon ausgehen, dass in dem darunter liegenden Programmabschnitt auf die angefragten Daten zugegriffen werden kann, so muss dies über sogenannte Callback-Funktionen (siehe Abschnitt **2.5.2**) geschehen.

2.5.1 Promise-Objekt

Ein Promise-Objekt (vgl. [21]) wird für asynchrone Berechnungen, oder den Austausch von Daten verwendet, wo nicht zu einem festen, vorhersehbarem Zeitpunkt das Ergebnis erwartet werden kann.

Folgende drei Zustände, in dem sich ein Promise-Objekt befinden kann, sind möglich:

Pending (schwebend)

Initialer Zustand

Fulfilled (erfüllt)

Operation erfolgreich

Rejected (zurück gewiesen)

Operation gescheitert

Es repräsentiert also einen Wert, der bei der Erstellung des Promise-Objektes nicht zwingend bekannt ist.

2.5.2 Callback-Funktionen

werden am Ende einer asynchronen Funktion über den Funktionsnamen **then** aufgerufen und liefern die angefragten Daten im Falle eines Erfolges im JSON-Format zurück. Schlägt die Operation fehl, kann der Fehler über das **Promise.prototype.catch()** abgefangen werden. Eine Verkettung von **then** und **catch** wird Komposition genannt.

Ändert sich der Zustand eines Promise-Objektes von **pending** zu **fulfilled**, oder zu **rejected**, so wird die Callback-Funktion aufgerufen, innerhalb der sich dann die Daten verarbeiten, oder nach außen über ein **return** weiter geben lassen.

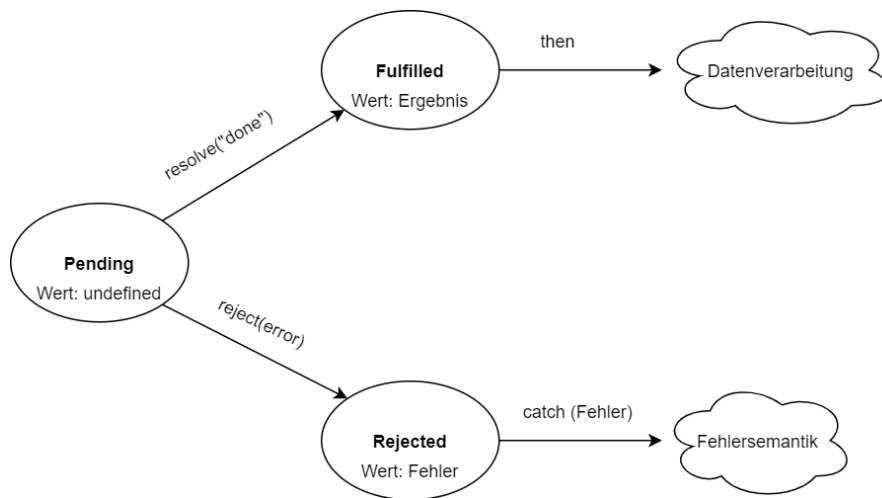


Abbildung 2.4: Unterschiedliche Zustände eines Promise-Objektes (Quelle: [21])

2.6 Test Driven Development

Zu Deutsch "Testgetriebene Entwicklung" (vgl. [26]) beschreibt sowohl eine agile Anforderung, als auch eine agile Designtechnik, mit dem Ziel, qualitativ hochwertigen Code zu schreiben.

2.6.1 Modultests

Das Vorgehen auf Modulebene deskribiert das Schreiben funktionaler Tests von atomaren Operationen, bevor diese überhaupt implementiert sind.

Die Testfälle werden dafür aus Benutzersicht definiert und müssen so elementar und einfach gehalten werden, dass sie bis zum Ende des Entwicklungsprozesses angewendet werden können.

Dies bedeutet, dass sich ihre semantische Funktion, auch nach weiterem Ausbau der Software, nicht ändert.

Der Vorteil liegt hierbei in der nicht vorhandenen Notwendigkeit, lange und komplexe Test-Szenarien im Falle neuer Anforderungen umschreiben zu müssen, sondern gezielt höhere Geschwindigkeit durch die Flexibilität essenzieller Tests zu erreichen.

Durch das Vordefinieren solcher Tests kann der Entwickler eindeutig bestimmen, welche Aufgabe einzelne Module zu erfüllen haben, ob und mit welchem Input sie "gefüttert" werden und welches Ergebnis in bestimmten Eingabefällen zu erwarten ist.

Für die richtige Durchführung dieses Testmusters ist es wichtig, die Module so kohärent wie möglich zu trennen, sowie eine atomare Funktionsbeschreibung bei zu behalten.

Beispiel

Folgendes Beispiel zeigt einen solch beschriebenen Modultest, sowie die später implementierte Funktion dazu. Das hier verwendete JavaScript-Testframework ist JEST (siehe Abschnitt 4.2.3)

```
1
2 test('Returns sum of given params x,y. Params: 3, 1', () => {
3   expect(funcSum(3, 1)).toBe(4);
4 })
```

Listing 2.2: Beispielcode eines Modultests geschrieben mit JEST

Mit dem Test sind Name, Input, Funktion und Rückgabewert der unten abgebildeten Operation `funcSum(x, y)` definiert. Er beschreibt somit ihre Signatur, als auch den erwarteten Wert für den gegebenen Input. Die Eingabereihenfolge der Parameter ist beliebig, da eine Addition kommutativ ist.

```
1
2 function funcSum(x, y) {
3   return x + y;
4 }
```

Listing 2.3: Quellcode der zu testenden Operation für die Summenbildung aus x und y

Modultests, wie oben gezeigt, können in Folge und in Form eines Skriptes, auch automatisiert, ausgeführt werden. Um wirklich auf Modulebene zu bleiben, sollten diese keine Abhängigkeiten zueinander aufweisen.

2.6.2 Integrationstests

Integrationstests (vgl. [17]) werden durchgeführt, um die zusammenhängende Funktion einzelner Operationen, unter anderem in einer bestimmten Abfolge, im Laufe des weiteren Entwicklungsprozesses prüfen zu können.

Die Voraussetzung eines Integrationstests ist es, vorher die einzelnen betreffenden Module auf ihre korrekte Funktionsweise hin getestet zu haben.

In größeren Projekten sieht ein Entwickler stets nur einen kleinen, für seine aktuelle Aufgabe relevanten Teil dessen, wodurch die Gefahr besteht, dass gekoppelte Teilfunktionen der Software auch an ganz anderer Stelle unerwartetes Verhalten zeigen.

Die Umsetzung von Integrationstests kann unter anderem über folgende zwei unterschiedliche Strategien erfolgen:

Testzielorientiert

Nur die für das Testszenario relevanten Teile des Systems werden im Zusammenspiel getestet

Vorgehensorientiert

Testszenario ist abhängig von der Integrationsreihenfolge aus der Systemarchitektur

Inkrementelle und nicht-inkrementelle Integrationstests

Darüber hinaus unterscheidet man zwischen inkrementellen und nicht-inkrementellen Integrationstests.

Inkrementelle Integrationstests

Definition Die Systemkomponenten werden in kleinen Gruppen getestet, wobei nicht verfügbare Komponenten virtualisiert, oder durch Platzhalter ersetzt werden.

Vorteil Leicht zu konstruierende Testfälle und eine hohe Testabdeckung sind möglich.

Nachteil Gegebenenfalls hoher Virtualisierungsaufwand von noch nicht bereits implementierten Komponenten.

Nicht-inkrementelle Integrationstests

Definition Es werden sehr viele, oder sogar alle Systemkomponenten, eventuell innerhalb eines Teilbereiches, getestet.

Vorteil Keine Virtualisierung notwendig.

Nachteil Aufgrund der Testkomplexität sind Fehler schwer zu lokalisieren. Um diesen Ansatz durchführen zu können, müssen alle Komponenten bereits implementiert sein.

3 Anforderungsanalyse

Bevor über Architektur und Design, oder gar über die Implementierung einer Software gesprochen wird, sollten zunächst die Anforderungen, die an sie gestellt sind, geklärt werden.

Dafür muss klar definiert sein, was die Software können und wie sie sich in bestimmten (Fehler-) Situationen verhalten soll. Hierfür unterscheidet man zwischen funktionalen und nichtfunktionalen Anforderungen (vgl. [19]).

From startup idea to software

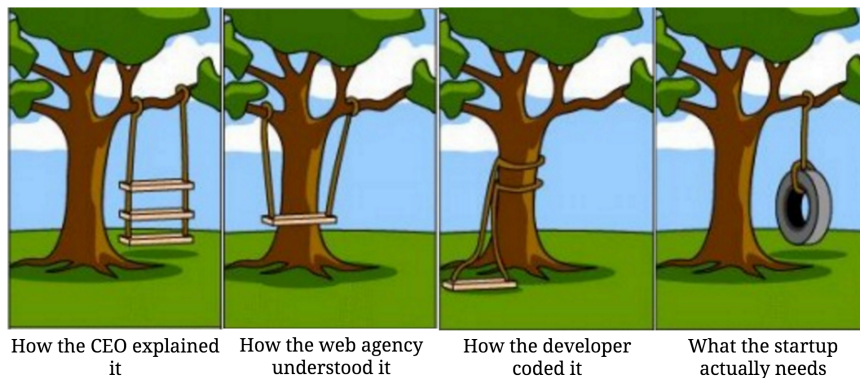


Abbildung 3.1: Entwickeln wir das Richtige? (Quelle: [1])

3.1 Ist-Situation

Wie in Abschnitt 1.4.1 beschrieben, wird die mobile Applikation *Mein o2* (siehe Abschnitt 1.3) aktuell über zwei organisatorisch getrennte Entwicklerteams entwickelt.

Jedes Entwicklerteam muss das erforderliche Know-How für die Implementierung und den dafür zu verwendenden Softwarestack mitbringen, die sich maßgeblich unterscheiden (siehe Tabelle 2.1).

Dies hat einen höheren Koordinationsaufwand zur Folge, da ein neues Feature auf den beiden nativen Applikationen in seiner Form und Funktion, absolut äquivalent sein muss.

Erreicht wird diese vom Kunden gestellte Anforderung nach der Implementierungsphase, über den sogenannten *Screenabgleich*, bei dem sich die für das Feature verantwortlichen Entwickler beider Teams zusammensetzen und die neue Funktion bis in das kleinste Detail abgleichen. Zeigt die neue Funktion in beiden nativen Anwendungen unterschiedliches Verhalten? Weisen sie erhebliche Designunterschiede auf, wie zum Beispiel Abweichungen des angezeigten Textes, oder nicht gewollte Unterschiede in der Farbgebung einzelner Designkomponenten?

Ist auch nur eine dieser Fragen mit *ja* zu beantworten, so muss der dafür verantwortliche Entwickler, auch wenn die Deadline so gut wie erreicht ist, Nacharbeitung leisten. Im Worst Case ist es sogar möglich, dass aus beiden Teams eine Nachbearbeitung, zum Beispiel an unterschiedlichen Teilaufgaben, notwendig ist.

Ein weiterer Nachteil der aktuellen Organisation ist die damit geschaffene Inflexibilität, auf eine hohe Zahl ausgefallener Ressourcen in einem Team reagieren zu können.

Unter Umständen sind Mitarbeiter aus dem anderen Team nicht ohne Einarbeitung in den für sie nicht ausreichend bekannten Softwarestack, mit dem das andere Team agiert, in der Lage unverzüglich die ausgefallenen Ressourcen zu ersetzen.

Die Gefahr eines zeitlichen Engpasses in einem der Teams ist somit gegeben, der auch dazu führen kann, dass nicht erlaubte Differenzen zuletzt implementierter Software entstehen können, die bei dem *Screenabgleich* nicht erkannt werden.

3.2 Funktionale und nichtfunktionale Anforderungen

Im Folgenden werden die Anforderungen an das Produkt dieses Projektes erläutert, um unter anderem den oben genannten Nachteilen entgegen zu wirken, sowie die wesentlichen Eigenschaften, die es erfüllen muss.

3.2.1 Nichtfunktionale Anforderungen

Um einen möglichst unproblematischen Ausbau der hybriden Anwendung, sowie eine leichtere Einarbeitung in das bestehende Projekt zu ermöglichen, ist es notwendig einen hohen Wert auf die Wartbarkeit der Software zu legen.

Das kann mit Architekturentscheidungen erreicht werden, bei denen einzelne Teilbereiche der Software durch ihre klar definierte Aufgabe voneinander getrennt entwickelt werden können, ohne die Stabilität und Funktion des Gesamtsystems zu gefährden.

Wird zum Beispiel an dem Design einer bestimmten Seite der Anwendung gearbeitet, so dürfen hier vorgenommene Änderungen nicht automatisch auf andere Seiten durchschlagen, oder gar nicht kalkulierte funktionale Änderungen zur Folge haben.

Um den organisatorischen Aufwand zu verringern ist es erforderlich, dass die Applikation lediglich an einer Codebasis unter der Verwendung eines Softwarestacks geschrieben werden kann, jedoch auf mehr als nur einer Plattform lauffähig ist. Sie hat also die Anforderung, **portabel** zu sein.

Dabei darf sie allerdings ihre charakteristischen Designmerkmale innerhalb der jeweiligen Umgebungsplattform nicht verlieren. Ein wichtiges Kriterium ist also die Transparenz hinsichtlich ihrer hybriden Eigenschaft.

Entscheidend ist auch die Transparenz, was die Performanz der Applikation betrifft. So soll dem Anwender nicht negativ auffallen, zum Beispiel in Form von Eingabeverzögerungen, dass es sich nicht um eine explizit, für seine genutzte Plattform, nativ geschriebene Anwendung handelt.

3.2.2 Funktionale Anforderungen

Ein Produkt darf aufgrund seiner Portabilität nicht an Funktionsumfang verlieren. Es muss kompatibel mit dem Rest des Systems sein und alle Aufgaben bewältigen können, die an dieses Produkt gestellt werden.

Die hybride Applikation muss mindestens genau so zuverlässig agieren, wie das jeweilige nativ geschriebene Original und somit eine mindestens genauso geringe Ausfallrate aufweisen können.

Dafür entscheidend ist, auf Fehlertoleranzen zu achten, wie Falscheingaben des Anwenders, oder etwa Umgebungsfehler, zum Beispiel der Ausfall der benötigten Internetverbindung.

Falls notwendig, muss auf Fehler angemessen reagiert werden, um Abstürze oder unerwartetes Verhalten der Anwendung zu vermeiden.

Ferner ist es sinnvoll, den Anwender auf Fehler, wie unter anderem Falscheingaben, oder eine fehlende Verbindung mit dem Internet hinzuweisen.

Die hybride Software darf zusammengefasst also nicht an Funktionalität, Zuverlässigkeit oder Benutzbarkeit verlieren.

Als präsentabler Funktionsumfang soll mit der App der reale Login eines Postpaid-Kunden möglich sein.

Die nach erfolgreichem Anmeldevorgang aufgerufene Seite zeigt dem Anwender seinen Inlands- sowie Auslandsverbrauch an.

Diese Informationen müssen über die Frontend-Applikation direkt aus dem bereits bestehenden Backend (siehe Abschnitt 4.1.3) bezogen werden.

3.3 Transfersituation

Die Transfersituation beschreibt eine portable, mobile Frontend-Anwendung, für die es lediglich ein Entwicklerteam benötigt. Dadurch wird der notwendige *Screenabgleich* auf ein Minimum reduziert, welches weniger Angriffsfläche für vermeidbare Fehler in der Qualitätssicherung des Produktes bietet.

Das zusammengeschweißte Team kann sich durch die Arbeit an derselben Codebasis untereinander besser organisieren und flexibler auf Ausfälle an zeitkritischen Teilaufgaben reagieren.

Beispielsweise kommt somit ein Mitarbeiter für den temporären Ersatz eines Anderen in Frage, der sich nach aktueller Situation in einem anderen Entwicklerteam befindet und sich erst in den unvertrauten Technologiestack einarbeiten müsste.

Somit entsteht ein reales Team für die Frontendlösung.

4 Methodik

In diesem Kapitel wird unter anderem die Architektur der hybriden Frontendlösung erläutert und es werden Werkzeuge, die für die Realisierung verwendet wurden, vorgestellt.

Ferner wird der komponentenbasierte Aufbau des Backend-Systems geschildert, mit dem die Frontend-Applikation kommuniziert, um die Daten zu erhalten, die von ihr präsentiert werden sollen.

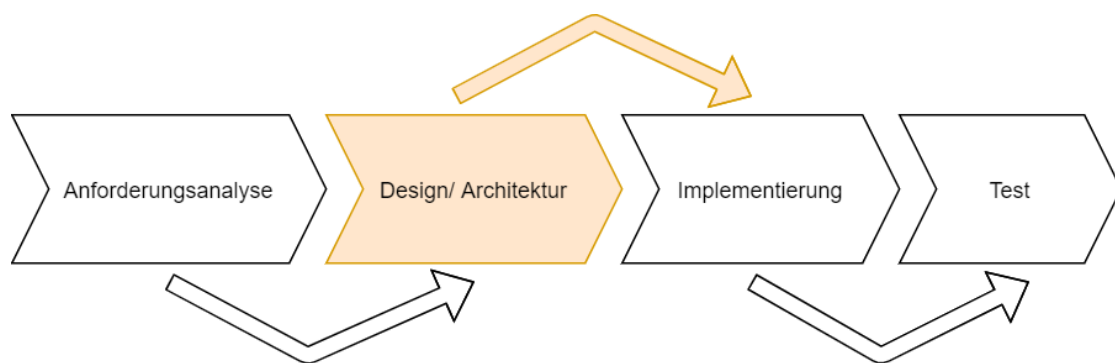


Abbildung 4.1: Architekturentscheidungen treffen als Teil des Softwareentwicklungszyklus

4.1 Architektur

4.1.1 Model-View-Controller

Abgekürzt MVC, beschreibt das *Model-View-Controller-Pattern* den objektorientierten Ansatz der strikten Trennung von Präsentation (View), Datenmodell (Model) und Steuerung (Controller), um den Aufbau der Software flexibel zu halten, sodass Änderungen oder Ergänzungen erleichtert werden können.

Dementsprechend fiel die Entscheidung aufgrund der Anforderung an die Wartbarkeit und Stabilität der Software, wie in Abschnitt **3.2.2** beschrieben, auf das Model-View-Controller-Pattern (vgl. [23]).

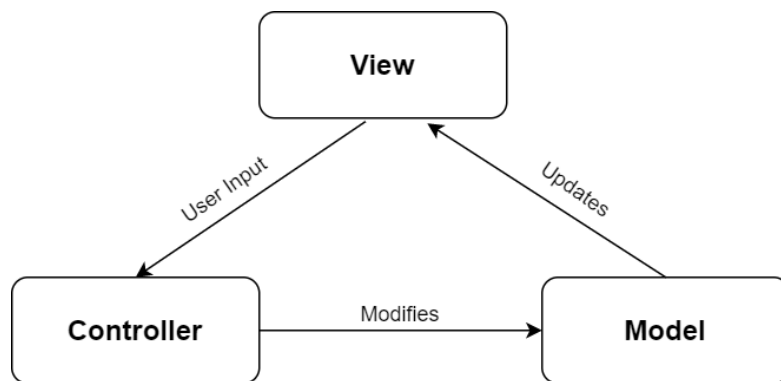


Abbildung 4.2: MVC als *Divide-and-Conquer* Paradigma

Der Anwender interagiert mit der Software über ihre Benutzerschnittstelle (Präsentation) und löst dabei Events aus, zum Beispiel die Autorisierung in der App oder das Aktualisieren von Daten, die ihm angezeigt werden sollen.

Diese Events werden von dem Controller, der Steuerungseinheit, verarbeitet und das Ergebnis wird in dafür definierten Datenmodellen abgelegt.

Damit die neuen oder veränderten Daten dem Anwender auch angezeigt werden können, muss die Präsentation aktualisiert werden.

Dies kann entweder aktiv durch das Datenmodell geschehen, oder durch die View selber, die mittels eigener Rendermethoden, die auf die Veränderung der Daten sensitiv sind, ihr User Interface neu aufbaut.

4.1.2 Aufbau des Frontends

Das folgende Komponenten-Diagramm beschreibt den architektonischen Aufbau der wesentlichen Komponenten der Frontend-Software.

Die Relationen zwischen den Komponenten sind wie in Abschnitt 4.1.1 nach dem MVC-Pattern gehalten mit dem Detail, dass die Komponenten in der View durch ihre jeweils eigenen Rendermethoden aktualisiert werden sollen.

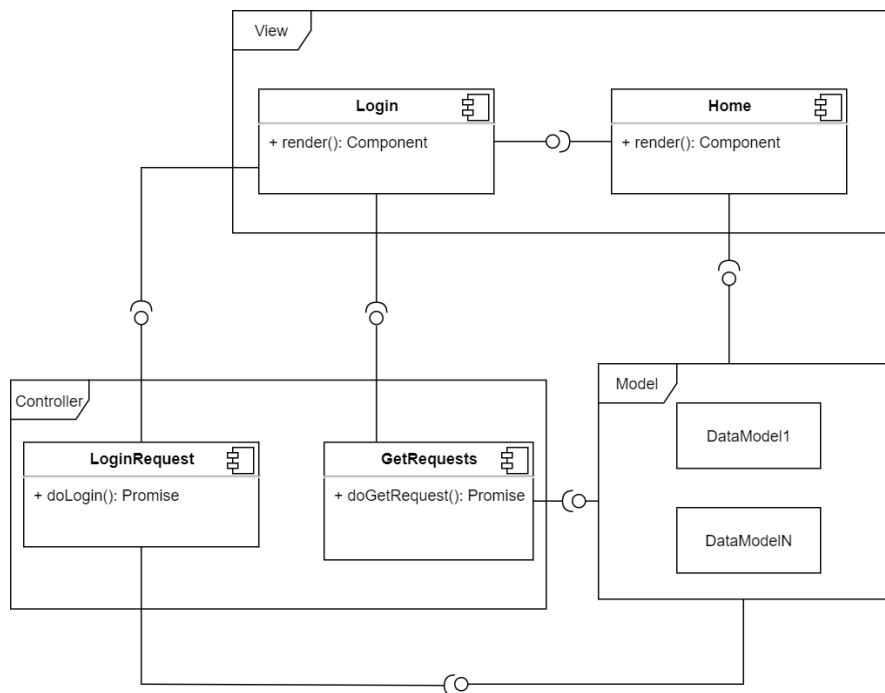


Abbildung 4.3: Beschreibt die Architektur als Komponentendiagramm nach MVC Ansatz

4.1.3 Backend

Eine mobile Applikation bildet oftmals lediglich das Frontend. Sie bietet eine grafische Oberfläche, über die ein Anwender mit dem Rest des Systems interagieren kann.

Die Aufgabe der Frontendapplikation ist es, über seine Benutzereingaben mit dem Backend zu kommunizieren, sprich, die relevanten Daten abzufragen und sie für die Präsentation aufzubereiten. Ein Backend kann aus lediglich einer oder, wie es oft der Fall ist, mehreren aktiven und passiven Komponenten bestehen. Eine passive Komponente wäre zum Beispiel eine Datenbank, aus der lediglich Datensätze, wie statische Texte, abgefragt werden.

Auch in diesem Projekt fungiert im Hintergrund, für den Verbraucher nicht sichtbar, ein Backendsystem, bestehend aus mehreren Komponenten, welches für die Datenhaltung, sowie -bearbeitung verantwortlich ist.

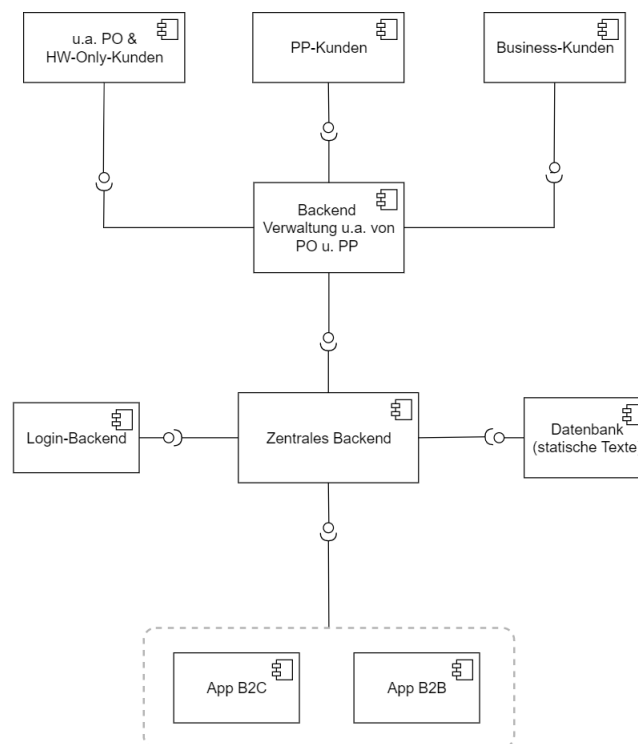


Abbildung 4.4: Beschreibt den Backendaufbau als Komponentendiagramm

4.2 Softwarestack

In diesem Abschnitt wird auf die Werkzeuge eingegangen, die verwendet wurden.

Es werden Im Besonderen die Aspekte hervorgehoben, die für einen Entwickler relevant sind, der bisher lediglich native Applikationen implementiert, oder sich noch gar nicht mit der Thematik befasst hat.

4.2.1 Entwicklungsumgebung

Um eine hybride Applikation mit React Native schreiben zu können, erfordert es lediglich einen einfachen Editor, wie beispielsweise Visual Studio Code, den man kostenlos verwenden kann.

Es eignen sich aber auch Umgebungen, wie *Atom IDE*, *WebStorm*, *Brackets* und viele mehr. Die hybride Applikation, die in diesem Projekt vorgestellt wird, wurde jedoch mit *Visual Studio Code* (vgl. [9]) geschrieben.

Mit der Unterstützung für über 40 Programmiersprachen und seinen vielfältigen optionalen Plugins, eignet sich VSC als Cross-Plattform-IDE im Allgemeinen sehr gut für die Entwicklung hybrider Frontendlösungen.

Für die Versionsverwaltung sind bereits in die IDE integrierte Git-Kommandos vorhanden, falls eine grafische Oberfläche anstelle einer Bash präferiert wird.

Auch verfügt VSC über eine Debug Funktion, sowie die Verwaltung von Geräte-Emulatoren. Diese müssen für Android jedoch zuvor über die IDE (Android Studio) installiert und konfiguriert werden, bevor sie mit VSC verknüpft werden können.

Für iOS wurde kein Emulator verwendet, in dem Fall wurde ausschließlich auf einem realen Gerät getestet.

Alternativ kann auch über den im angehängten Handbuch aufgeführten Link, direkt im eigenen Browser, mit der Entwicklung einer React Native Applikation gestartet werden.

Da das Hochladen des Projektes aktuell erforderlich für den möglichen Export auf den lokalen Datenträger ist, wird im Fall sensibler Daten davon abgeraten.

Dennoch bietet diese Möglichkeit einen guten Einstieg in die Thematik, da das Webtool bereits über Emulatoren für beide Plattformen verfügt.

4.2.2 Expo-Client

Der Expo-Client wird auf einem stationären PC, oder Laptop und falls nicht mit einem Emulator gearbeitet wird, darüber hinaus auf dem mobilen Testgerät installiert.

Er ist kompatibel mit einer aktuellen Windows-, macOS- sowie Linux Distribution.

Wird der Expo-Client ausgeführt, startet dieser einen Applikationsserver über eine TCP-
Socket-Verbindung auf dem Rechner.

Über diese Verbindung werden alle benötigten Projektdateien, wie Module und der Quellcode, auf die Expo-App des mobilen Testgeräts gestreamt.

Dies übernimmt der React Native Packager, der auf Node.js basiert.

Voraussetzung dafür ist, dass der Applikationsserver und das Testgerät mit dem gleichen Netzwerk verbunden sind.

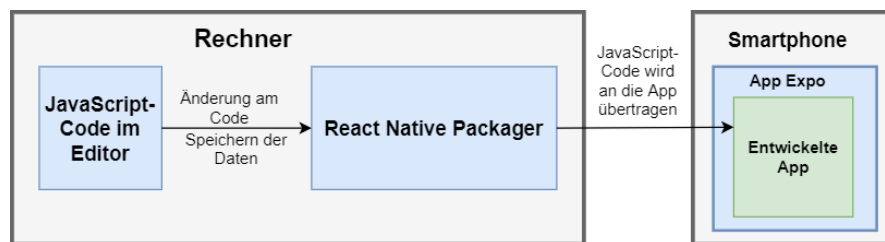


Abbildung 4.5: Änderungen am Code werden vom React Native Packager direkt an die Expo-App übertragen (vgl.[12])

Ein Feature, welches die Entwicklungsphase sehr erleichtert, ist in der Abbildung 4.5 dargestellte Aktualisieren der App.

Dadurch wird viel Zeit durch nicht notwendige Kompilierung des Quellcodes gewonnen, um gemachte Änderungen sehen zu können.

Man unterscheidet zwischen *Live Reload* und *Hot Reloading* (vgl. [16]).

Live Reload

Der *React Native Packager* überträgt den vollständigen Quellcode der App auf das Testgerät, sobald Änderungen an diesem vorgenommen wurden.

Dieser Vorgang erfolgt nach aktivem oder inaktivem Speichern vollkommen automatisch.

Wird der Expo-Client das erste Mal gestartet, ist diese Funktion standardmäßig aktiviert.

Hot Reloading

Im Gegensatz zu *Live Reload* werden unter dieser Option nur die geänderten Fragmente des Quellcodes erneut übertragen.

Damit können Änderungen noch schneller angezeigt und getestet werden, da nicht stets die komplette App neu geladen wird.

Ein weiterer Vorteil ist, dass ihr Zustand erhalten bleibt. So behalten zum Beispiel Textfelder ihren Inhalt, während lediglich die Farbe des Seitenhintergrunds angepasst wurde.

Debugging

Für eine effiziente Fehleranalyse hat der Expo-Client einen eingebauten Debugger, der dem Entwickler Informationen über den Stack trace liefert.

Über die Speicherzurückverfolgung lassen sich Fehlerquellen schneller identifizieren. Es wird genau auf die Quellcodezeilen innerhalb der jeweiligen Datei verweisen, in denen der Programmabsturz dokumentiert wurde.

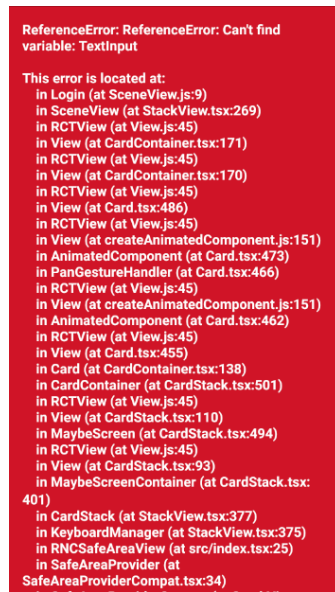


Abbildung 4.6: Expo gibt bei Kompilierungs- sowie Laufzeitfehlern den Stack trace an

Die Abbildung 4.6 zeigt den Output von Expo im Fehlerfall an. Wie sich schnell erkennen lässt fehlt hier der Import der React Native eigenen Komponente *TextInput*.

Nach Beheben des Fehlers, in diesem Fall das Hinzufügen des fehlenden Imports von *TextInput*, wird die App neu geladen und wie erwartet ausgeführt.

4.2.3 JEST

JEST (vgl. [18]) ist ein von Facebook entwickeltes Testframework, mit Hilfe dessen sich, unter anderem, Modultests (siehe Abschnitt **2.6.1**) in JavaScript realisieren lassen.

Durch den globalen Zugriff auf die einzelnen Tests können diese parallel ausgeführt werden. Zuvor fehlgeschlagene Tests werden dabei höher priorisiert.

Für eine umfangreiche Objektsimulation unterstützt *JEST* die *Mock Functions API*.

Damit können zum Beispiel relevante, jedoch noch nicht implementierte Funktionen eines Backends simuliert werden, um ein damit zusammenhängendes Ablaufszenario durchspielen zu können.

Im Laufe der Entwicklung kann es oft von Relevanz sein, mit der Implementierung neuer Features beginnen zu können, ohne bereits auf eine reale Schnittstelle zugreifen zu können.

Dies kann der Fall sein, wenn an einem neuen Feature im Frontend, sowie im Backend parallel gearbeitet wird.

5 Realisierung

Das Ziel in diesem Kapitel ist es, die in dem Projekt vorgenommene Herangehensweise an die Umsetzung zu erläutern.

Es wird beschrieben, wie das App-Design und die Kommunikation mit dem Backend realisiert wurde.

Ferner wird auf Schwierigkeiten und Tücken im Laufe des Entwicklungsprozesses eingegangen.

5.1 Testgetriebene Entwicklung

Die entwickelte Applikation greift unter anderem auf eigens definierte Hilfsfunktionen zu, zum Beispiel für das Festlegen des Divisors einer bestimmten Maßeinheit.

Diese Hilfsfunktionen wurden bereits vor ihrer Implementierung in kohärenter Form, mittels Unterbringung in separaten Dateien, ihrer Semantik betreffend definiert.

Jede Datei beschreibt die verschiedenen Anwendungsfälle, betreffend der Eingabeparameter und das jeweils erwartete Ergebnis.

Betrachten wir im Folgenden die testgetriebene Entwicklung der Funktion *funcGetUsageDivisorForMobileData* als Beispiel.

Jede Hilfsfunktion wurde auf diese Weise implementiert.

```
1
2 import { funcGetUsageDivisorForMobileData } from './RNWorkspace/myhybridapp/utills/HomeUtils';
3 test('get divisor based on unit [KB]', () => {
4   expect(funcGetUsageDivisorForMobileData('KB')).toBe(1000);
5 });
6 test('get divisor based on unit [MB]', () => {
7   expect(funcGetUsageDivisorForMobileData('MB')).toBe(1);
8 });
9 test('get divisor based on unit [GB]', () => {
10  expect(funcGetUsageDivisorForMobileData('GB')).toBe(0.001);
11 });
12 test('get divisor based on unit [TB]', () => {
13  expect(funcGetUsageDivisorForMobileData('TB')).toBe(0.000001);
14 });
15 test('get divisor based on unit default', () => {
16  expect(funcGetUsageDivisorForMobileData('anythingElse')).toBe(1);
17 });
```

Listing 5.1: Die Tests definieren die zu implementierende Funktion. Sie werden auf Modulebene sequentiell ausgeführt

Der Quellcode **5.1** zeigt den Aufbau einer solcher Testdatei, die mit dem Framework JEST (siehe Abschnitt **4.2.3**) geschrieben wurde.

Es folgt der Codeausschnitt **5.2** der Funktion, die auf Basis dieses Tests implementiert wurde.

```
1
2 export function funcGetUsageDivisorForMobileData(unit){
3   let divisor
4   switch(unit){
5     case 'KB':
6       divisor = 1000;
7     break;
8     case 'MB':
9       divisor = 1;
10    break;
11    case 'GB':
12      divisor = 0.001;
13    break;
14    case 'TB':
15      divisor = 0.000001;
16    break;
17    default:
18      divisor = 1;
19    break;
20  }
21  return divisor;
22 }
```

Listing 5.2: Die Funktion liefert den passenden Divisor für die Ausgangs-Maßeinheit *MB* zurück

Anhand des im Vorfeld geschriebenen Tests kann die korrekte Verhaltensweise der Funktion umgehend mit unterschiedlichen Eingabemustern geprüft werden.

Wird *JEST* über den *Node Packager* mit dem Befehl **npm** und dem Skriptnamen **test** als Parameter nun ausgeführt, liefert das Framework folgende Ausgabe:

```
> jest "FuncGetUsageDivisorForMobileData.test.js"
PASS test/HomeUtilsTest/FuncGetUsageDivisorForMobileData.test.js
  ✓ get divisor based on unit [KB] (3ms)
  ✓ get divisor based on unit [MB]
  ✓ get divisor based on unit [GB]
  ✓ get divisor based on unit [TB]
  ✓ get divisor based on unit default (1ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:  0 total
Time:        1.65s, estimated 2s
Ran all test suites matching /FuncGetUsageDivisorForMobileData.test.js/i.
```

Abbildung 5.1: Ausgabe des Ergebnisses aller fünf Testmuster der getesteten Funktion *funcGetUsageDivisorForMobileData*

Damit ist das korrekte Verhalten dieser Funktion für alle getesteten Eingaben sichergestellt und belegbar.

5.2 Backendkommunikation

Damit die Frontendapplikation die benötigten Daten aus dem zentralen Backend (siehe Abbildung 4.4) abfragen kann, muss zunächst über einen validen Postpaid-Account eine Authentifizierung erfolgen.

Dies geschieht auf dem Login Screen über die Eingabe der MSISDN-Nummer im oberen Input-Textfeld und der Eingabe eines gültigen Passwortes, des im sich darunter befindlichen Eingabefeldes (siehe Abbildung 6.2).

5.2.1 Senden der Login Anfrage mittels Fetch-API

Um eine valide Anfrage an das Login-Backend zu senden, muss der Header des Requests alle dafür benötigten Informationen enthalten, wie Passwort, Benutzername und die Adress-Information des Hosts.

```
1
2 static async doLogin(username, password){
3 let postData = EMPTY;
4 try{
5 const resp = await fetch (LOGIN_URL, {
6 method: 'POST',
7 headers: {
8 'X-OpenAM-Username': username,
9 'X-OpenAM-Password': password,
10 'Host': 'LOGIN_HOST_ADDRESS',
11 'Cache-Control': 'no-cache',
12 'Accept': '*/*',
13 'Accept-Encoding': 'gzip, deflate',
14 'Accept-API-Version': 'resource=2.0, protocol=1.0',
15 },
16 body: postData,
17 credentials: 'include',
18 });
19 console.log(resp);
20 return resp;
21 } catch (err){
22 console.log(err)
23 }
24 }
```

Listing 5.3: Beschreibt den Aufbau der asynchronen Funktion *doLogin*

Der Aufruf der **await fetch**-Funktion muss in einem asynchronen Kontext durch die Deklaration **async** erfolgen, da ein Promise-Objekt (siehe Abschnitt 2.5.1) zurückgegeben wird.

Der Benutzername und das Passwort werden über die Eingabeparameter der Funktion an den Header übergeben. Für das Handling dieser Informationen wird die Open Source Zugriffsverwaltung *OpenAM* genutzt.

Da kein Payload für die Anfrage erforderlich ist, wird ein leerer Body übergeben. Durch die Zuordnung in Zeile 17 im Ausschnitt 5.3 wird der plattformübergreifende Austausch von Cookies ermöglicht.

Die Login-Daten werden als **POST**-Request (vgl. [24]) zur Validierung direkt über die Frontend-App an das Login-Backend gesendet, welches im Falle eines erfolgreichen Logins ein Promise-Objekt mit http-Status **200** für **Ok** zurücksendet.

Das Anzeigen der Informationen des zurückgegebenen Promise-Objektes erfolgt auf der Konsole, **nachdem** die Daten vorliegen. Zwischenzeitlich wird Programmcode außerhalb der asynchronen Funktion ausgeführt. Die Informationen eines Promise-Objektes werden über Callback-Funktionen verarbeitet (siehe Abschnitt **2.5.2**).

Da vorher nicht garantiert werden kann, ein positives Ergebnis zurück zu bekommen, geschieht der asynchrone Aufruf innerhalb eines **Try-Catch**-Blockes.

Im Header der Antwort ist unter anderem das Cookie enthalten, welches für die Authentifizierung der Datenanfragen an das zentrale Backend benötigt wird. Des Weiteren erhalten wir das Ergebnis unseres Login Versuchs in Plain-Textform, sowie Informationen des Tariftypens.

Im Payload der Antwort ist ein Token enthalten, welches für die Autorisierung bestimmter Backend-Endpunkte erforderlich ist, die in diesem Projekt jedoch nicht angefragt wurden.

Damit sind alle Informationen vorhanden, die für die Kommunikation mit dem zentralen Backend benötigt werden.

5.2.2 Kommunikation mit dem zentralen Backend mittels Fetch-API

Die unten in dem Codeabschnitt 5.4 abgebildete Funktion beschreibt das Template einer Datenanfrage an das zentrale Backend, für die kein Token benötigt wird.

```
1
2 static async doNormalLoginGetRequest(url, cookie){
3 let postData = EMPTY;
4 try{
5 const resp = await fetch (url, {
6 method: 'GET',
7 cache: 'no-cache',
8 credentials: 'include',
9 headers: {
10 'cookie': O2_COOKIE_NAME + '=' + cookie,
11 },
12 body: postData
13 });
14 console.log(resp);
15 return resp;
16 } catch (err){
17 console.log(err)
18 }
19 }
```

Listing 5.4: Template für **GET**-Requests an das zentrale Backend

Für eine simple Datenanfrage wird die Request-Methode **GET** benutzt.

Um die Berechtigung nachzuweisen, wird das dafür benötigte Cookie als Parameter, zusammen mit der Url des Backend-Endpunktes an die Funktion übergeben.

Der spezifische Aufbau einer solchen Anfrage unter Verwendung des Funktions-Templates wird im folgenden Codeabschnitt 5.5 gezeigt.

```
1
2 static async getCustomerData(cookie){
3 let responseStatus;
4 await this.doNormalLoginGetRequest(CUSTOMERS_CALL, cookie)
5 .then(data => {
6 responseStatus = data.status;
7 return data.json()
8 }).then(dataJson => {
9 if (responseStatus === 200){
10 this.fillCustomerBaseInfos(dataJson);
11 this.fillAccountInfos(dataJson);
12 }
13 })
14 return responseStatus;
15 }
```

Listing 5.5: Nach Erhalt der Kundendaten werden diese innerhalb der Callbackfunktion in den Datenmodellen abgelegt (Zeilen 11-12)

War der Login Versuch erfolgreich, erfolgt die oben gezeigte Abfrage automatisch und direkt im Anschluss. Im Fall eines erfolgreichen Ablaufs können die Daten aus den Modellen nun auf der Oberfläche der App angezeigt werden.

5.3 Aufbau des User Interfaces

Wie in Abschnitt 4.1.2 bereits erläutert, verfügen die einzelnen Seiten der App über eigene Rendermethoden, die immer dann aufgerufen werden, sobald der Zustand eines Attributs der Seite sich verändert.

Das User Interface wird dann erneut aufgebaut und zeigt den aktuellen Zustand der Applikation an.

5.3.1 Design-Komponenten

Für den Aufbau der Benutzerschnittstelle bedarf es Komponenten, wie beispielsweise einer Schaltfläche zur Ausführung einer Aktion durch den Anwender, einem Textfeld für Eingaben, oder Andere.

Risiko durch Komponenten aus Quellen Dritter

Bevor man jedoch eine Komponente, seiner View hinzufügen möchte, muss man zwischen React Native eigenen Komponenten und denen aus Quellen Dritter unterscheiden.

Mögen Letztere im ersten Moment attraktiver erscheinen, so muss man sich darüber im Klaren sein, dass diese keine langfristige Garantie bezogen auf ihre Wartung oder Updates bieten.

Unter deren Verwendung riskiert man somit später auftretende Instabilität der Software, sowie damit verbundenen hohen Refactoring-Aufwand.

Da Framework eigene Komponenten diesbezüglich eine deutlich höhere Gewährleistung bieten, wurden vordefinierte Komponenten ausschließlich aus Modulen von React Native verwendet.

Deklaration und Initialisierung einer Designkomponente

Jede sich dem Verbraucher präsentierende Seite besitzt eine Rendermethode, die den Aufbau des User Interfaces ausführt, sobald sich ihr Zustand verändert.

Damit die Komponenten wie erwartet auf dem Screen erscheinen, müssen sie über diese Methode deklariert werden.

```

1
2 import { TextInput, View, StyleSheet } from 'react-native';
3 (... )
4 <View style={styles.container}>
5   <TextInput
6     style={styles.inputTxt}
7     onChangeText={({txtName}) => {
8       this.setState({txtName});
9       this.setState({btnDisabled: funcTxtIsEmpty(this.state.txtName,
10        this.state.txtPass)});
11     value={this.state.txtName}
12   />
13 (... )
14 </View>
15 (... )

```

Listing 5.6: Eine Komponente muss immer von einer übergeordneten View-Komponente umgeben sein

Abbildung 5.6 zeigt die Implementierung der Komponente *TextInput*. Die Festlegung des Designs dieser Komponente erfolgt der Wartbarkeit dienend in dafür entsprechend definierten Objekten.

Anhand ihrer Objektattribute können Form, Farbe und Größe, sowie Positionierung innerhalb der übergeordneten Komponente bestimmt werden.

Eine Designkomponente kann auf ein vom Anwender ausgelöstes Event reagieren. In Zeile 9 wird über die Funktion `onChangeText` durch jede getätigte Eingabe der Zustand des Seitenattributs *txtName* aktualisiert.

Damit seine Eingabe dem Anwender angezeigt werden kann, wird dem Wert der Komponente (siehe Zeile 14) der Zustand von *txtName* zugewiesen.

Durch jede Änderung von *txtName* wird nun die Rendermethode der Seite aufgerufen, wodurch das Eingabefeld dem Anwender mit seinem neuen Wert präsentiert wird.

Styling von Komponenten

Wie im letzten Abschnitt **5.3.1** beschrieben, wurde das Erscheinungsbild der verwendeten Komponenten durch Objekte definiert.

```
1 (...)  
2 const styles = StyleSheet.create({  
3   container: {  
4     flex: 1,  
5     alignItems: 'center',  
6     paddingTop: 70,  
7   },  
8   (...)  
9 })
```

Listing 5.7: Das Objekt *container* besitzt Eigenschaften die das Design und die Positionierung einer Komponente beschreiben

Objekte weisen den Vorteil der Wiederverwendbarkeit auf.

Dies dient der Wartbarkeit und ist ein effektives Mittel gegen redundanten Code.

Die Attribuierung geschieht bei nativen Modulen für beide Plattform gleichermaßen.

Es kann jedoch dennoch zu visuellen Unterschieden kommen, wie in Abschnitt **5.4.1** erläutert.

5.3.2 Screendesign-Vergleich zwischen beiden Plattformen

Gemäß der erläuterten Funktion der React Native Bridge in Abbildung 2.1, werden die angelegten Komponenten direkt aus den Bibliotheken der Plattformen Android und iOS übersetzt.

Das Resultat bilden Designelemente, die plattformspezifische Charakteristika aufweisen.

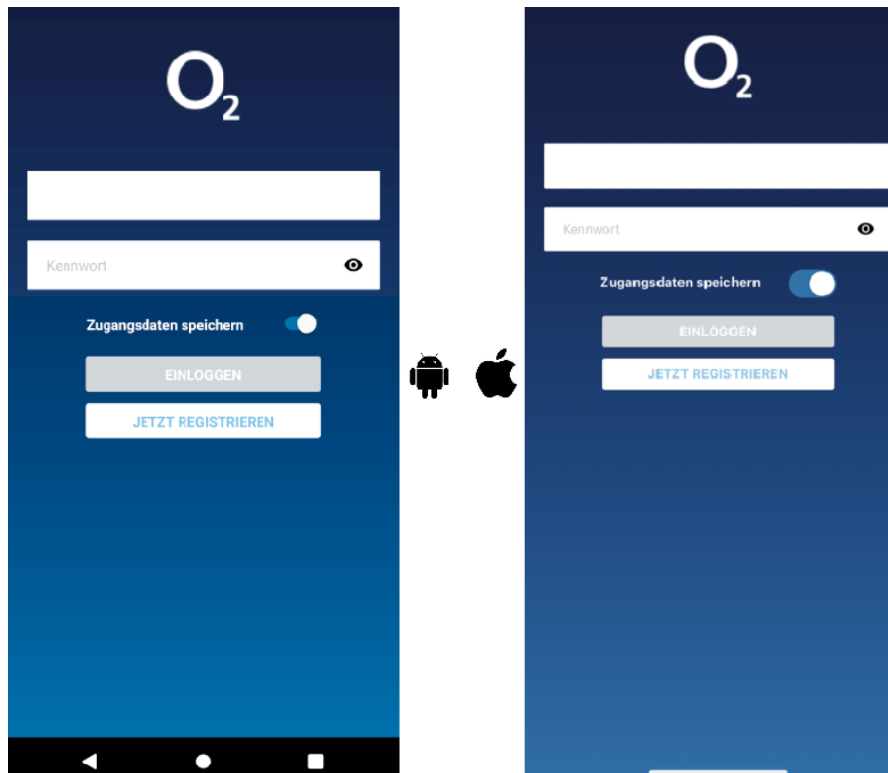


Abbildung 5.2: Die Komponente *Toggle* wird plattformtypisch dargestellt. Sie bildet ein natives Modul und wird aus den jeweiligen Bibliotheken übersetzt

5.4 Probleme im Laufe des Entwicklungsprozesses

5.4.1 Styling-Eigenschaften

Bei dem Festlegen des Designs einer Komponente kann es zu unterschiedlichen Darstellungen dieser gemäß der jeweiligen Plattformumgebung kommen.

Bestimmte Eigenschaften sind nicht in allen Bibliotheken wiederzufinden, oder werden anders interpretiert, was einen stetigen Abgleich notwendig macht.

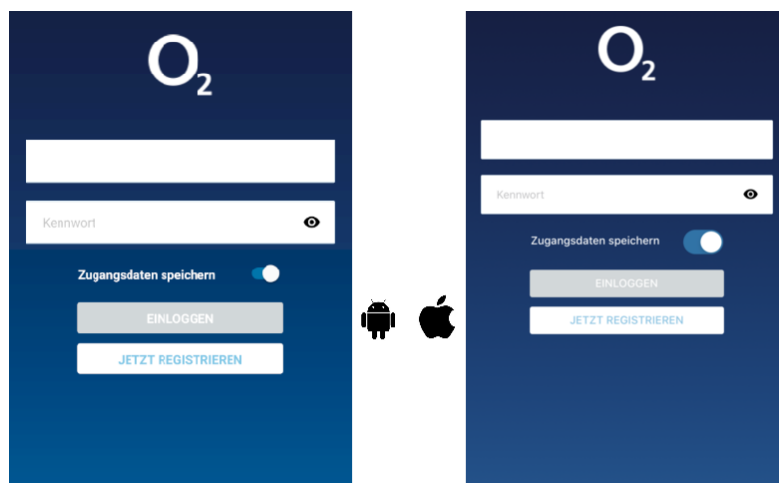


Abbildung 5.3: Verwendung eines festen Wertes für das Attribut *fontweight*. Die Beschriftung der angezeigten Elemente wird lediglich innerhalb der Android Umgebung fett markiert

Wie in Abbildung 5.3 zu sehen, fällt die fehlende Umsetzung insbesondere bei der Beschriftung des Toggles *Zugangsdaten speichern* auf.

Die Ursache des Problems ist eine unterschiedliche Interpretation des Wertes für das Gewicht der Schrift.

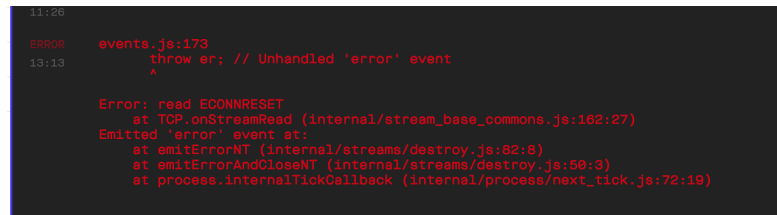
Auf einem iOS-Gerät sind die erwarteten Auswirkungen erst ab einem deutlich höheren Wert sichtbar.

Eine simple Lösung bietet die Verwendung der Konstante *bold*, welche auf beiden Systemen ein äquivalentes Ergebnis liefert (siehe Abbildung 5.2).

Aufgrund verschiedener Bildschirmabmessungen kann es dennoch, je nach Vergleichsgerät, zu subjektiv wahrgenommenen Unterschieden im Direktvergleich kommen.

Abstürze

Abstürze der Applikation, zum Beispiel durch Laufzeitfehler, können einen erforderlichen Neustart des Expo-Clients provozieren, da die Socket Verbindung in diesem Fall unterbrochen wird.



```
11:26
ERROR events.js:173
13:13      throw er; // Unhandled 'error' event
      ^

Error: read ECONNRESET
    at TCP.onStreamRead (internal/stream_base_commons.js:162:27)
Emitted 'error' event at:
    at emitErrorNT (internal/streams/destroy.js:82:8)
    at emitErrorAndCloseNT (internal/streams/destroy.js:58:3)
    at process.internalTickCallback (internal/process/next_tick.js:72:19)
```

Abbildung 5.4: Beim Absturz des Expo-Clients wird diese Meldung auf der Konsole angezeigt. Ein Neustart des Clients ist dann erforderlich

Dies kann in Summe einiges an Zeit kosten, da dadurch die App jedes Mal wieder in ihren initialen Zustand zurückgesetzt wird.

Framework spezifische Fehler

Kein Framework ist fehlerfrei, auch React Native nicht.

Ein bekanntes Problem ist die unvollständige Anzeige von erhaltenen Cookies in einem Promise-Objekt im Android Kontext.

Wohingegen auf einem iOS-Gerät alle empfangenen Cookies im Log angezeigt werden können, sind diese Informationen unter der Verwendung eines Android-Gerätes unvollständig.

Es wird lediglich ein beliebiges Cookie sichtbar.

Dadurch geht man zunächst davon aus, die nicht angezeigten Informationen würden gänzlich fehlen.

In der Tat jedoch lässt sich auch auf nicht angezeigte Cookies zugreifen.

Durch eine Irreführung dieser Art kann ebenfalls viel Zeit verloren gehen.

5.5 Aufbau nach UML

Hilfsfunktionen, die von den View-Komponenten verwendet werden, sind in separate Dateien untergebracht. Dies führt zu mehr Übersicht, insbesondere in Hinblick auf die mit JEST zu testenden Module (siehe Abschnitt 5.1).

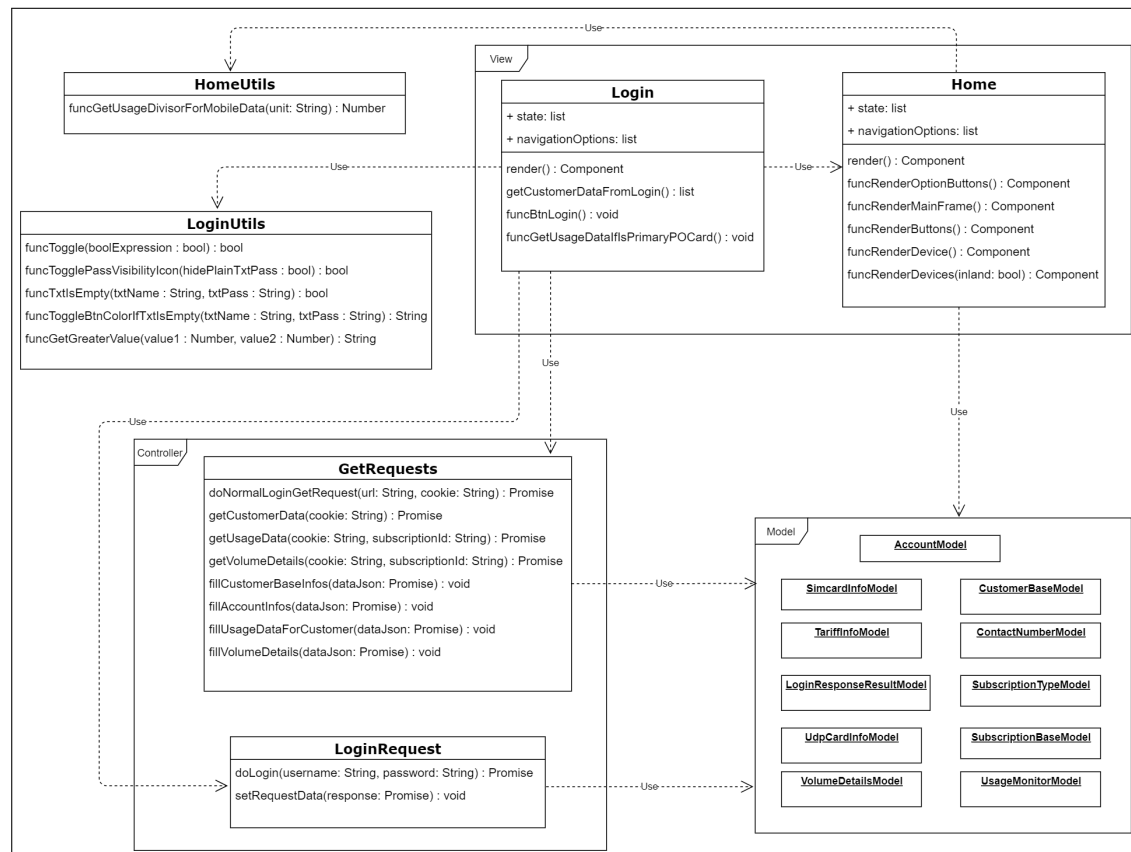


Abbildung 5.5: Aufbau der Architektur des Frontends nach MVC-Ansatz

Die Controller-Instanzen werden durch Benutzerevents, wie einem Login, aufgerufen und speichern die angefragten Informationen in den Datenmodellen ab.

Ihr Inhalt wird von den View-Komponenten für die Präsentation verwendet.

Der Aufbau der UI geschieht mittels eigener Rendermethoden der View.

5.6 Deployment

Aufgrund unterschiedlicher Richtlinien betreffend Googles Play Store und dem App Store von Apple, kann ein Deployment nicht automatisiert über React Native geschehen. Hat man die Entwicklungsphase abgeschlossen und möchte man seine React Native-Applikation veröffentlichen, so müssen hierfür bestimmte Voraussetzungen erfüllt sein.

5.6.1 Voraussetzungen

Bezüglich der Präsentation der App gelten die allgemeinen Vorschriften der jeweiligen App Stores, betreffend Icon-Größe, Kategorie, Schlüsselwörter etc. Es muss sichergestellt werden, dass alle relevanten Informationen in der **app.json** enthalten sind, wie das nachfolgende Beispiel zeigt.

```
1  {
2      "expo": {
3          "name": "App Name",
4          "icon": "./path/to/your/app-icon.png",
5          "version": "1.0.0",
6          "slug": "app-slug",
7          "sdkVersion": "XX.0.0",
8          "ios": {
9              "bundleIdentifier": "com.companyname.appname"
10         },
11         "android": {
12             "package": "com.companyname.appname"
13         }
14     }
15 }
```

Listing 5.8: Die Informationen in der Datei **app.json** müssen ein vollständiges Mindestmaß erfüllen (vgl. [8])

5.6.2 Generierung nativen Codes

Nativer Code lässt sich durch einen Expo-Befehlssatz erzeugen.

Durch die Ausführung des Kommandozeilenbefehls **expo publish** in einer Shell oder ähnlichem, wird ein Link geteilt, unter dem die zwei nativen Codeversionen generiert werden. Eine erneute Ausführung des Prozesses ist nach jedem Update der App erforderlich.

5.6.3 Bauen nativer Apps

Für den Bau der Appversionen wird ein Expo-Account benötigt.

Des Weiteren ist für einen iOS-Build der Besitz eines Entwickler-Accounts Voraussetzung. Mit dem Befehl **expo build:[*android/ ios*]** wird der Vorgang über einen öffentlichen Expo Build-Server eingeleitet.

Unter Verwendung des von Expo in seiner Konsole geteilten Links erhält man Einsicht über den Log des Builds. Da das Warten in der Queue viel Zeit in Anspruch nehmen kann, lohnt es sich das Feature für *Priority Builds* kostenpflichtig zu erwerben.

Nach erfolgreichem Durchlauf erhält man von Expo einen Link, über den man die .apk oder Iphone Application (.ipa)-Datei herunterladen kann. Diese kann dann im jeweiligen Store als herunterladbares Installationsmedium veröffentlicht werden.

Möchte man nicht auf einen öffentlichen Build-Server von Expo zurückgreifen, so besteht die Möglichkeit das von ihm genutzte Open Source Build-Tool *turtle-cli* zu verwenden (vgl. [6]).

Somit wird das Warten in einer Queue umgangen und der Build-Prozess der App geschieht privat.

6 Systemtests

Die Systemtests wurden in Form eines Anwendungsszenarios und den unterschiedlichen Fehlerszenarien mit Hilfe des *Hot Reloading*-Features (siehe Abschnitt 4.2.2) des Expo-Clients durchgeführt.

6.1 Anwendungsszenario

6.1.1 Use Case Sequenzdiagramm

Um die in Abschnitt 3.2.2 beschriebene funktionale Anforderung der Applikation zu beschreiben, wurde das nachfolgende Use Case-Szenario als Ablaufdiagramm definiert.

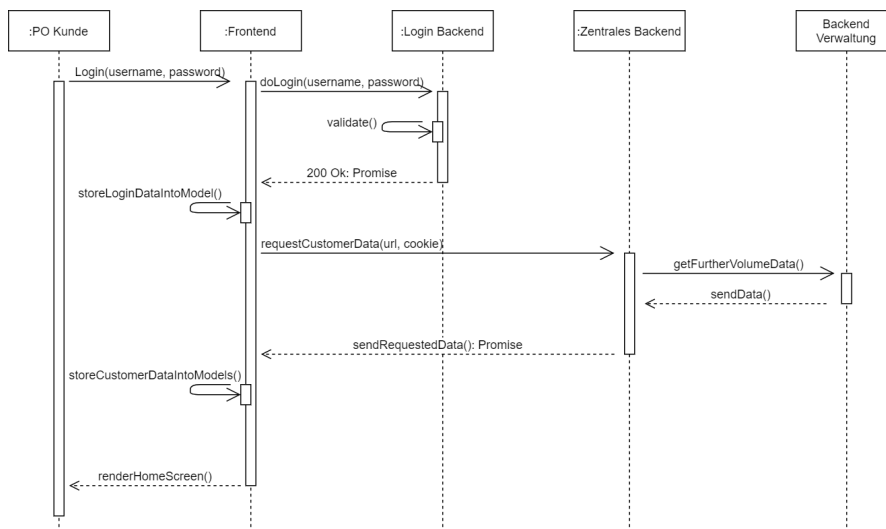


Abbildung 6.1: Beschreibt den positiv verlaufenden Anwendungsfall. Es treten keine Fehler auf

6.1.2 App-Durchlauf

Es wird der Ablauf des vom Postpaidkunden erfolgreich durchgeführten Logins und die im Anschluss folgende, automatische Navigation zur Verbraucherseite beschrieben.

Wie in Abbildung 6.1 gezeigt, treten während des gesamten Prozesses in diesem Szenario keine Fehler auf.

Die Applikation verhält sich dem Verbraucher gegenüber wie von ihm erwartet.

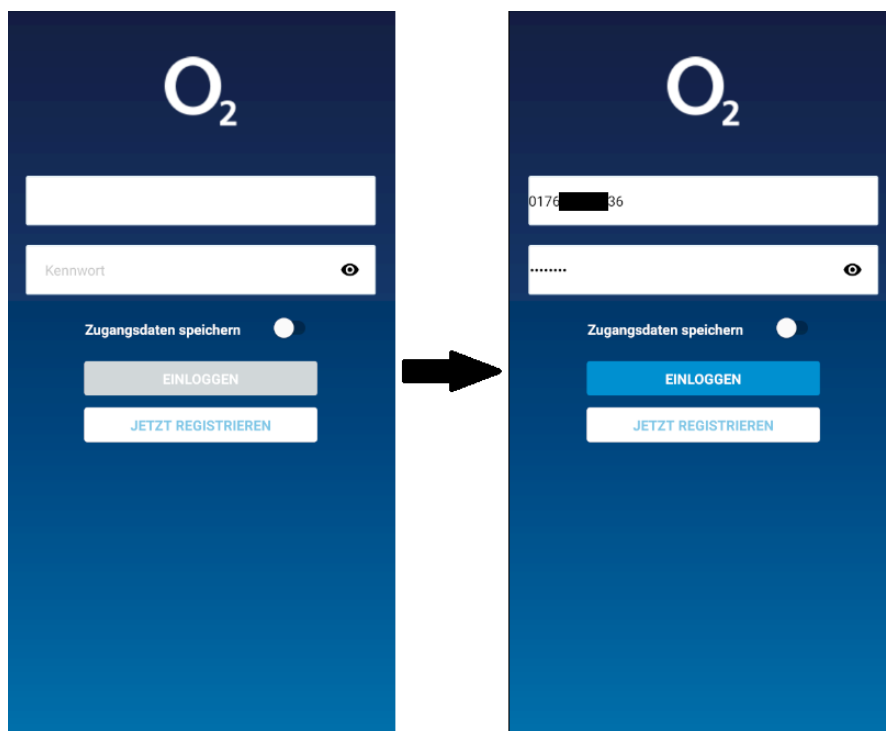


Abbildung 6.2: Der Kunde gibt seine Logindaten direkt in der App ein (hier Android)

Sind beide Eingabefelder ohne Inhalt, ist die Schaltfläche *EINLOGGEN* deaktiviert, da eine Login Anfrage ohne Eingabe von Benutzername und Passwort nicht zielführend ist. Ihr Hintergrund ist zur Erkennung grau, für deaktiviert.

Dies beschreibt den initialen Zustand der Applikation.

Füllen sich beide Textfelder mit Inhalt, kann der Verbraucher über die Schaltfläche *EINLOGGEN* den Login Vorgang starten.

Die Schaltfläche wird aktiviert und ihr Hintergrund wechselt zu einem markentypischen Blauton.

Nativer Aktivitätsindikator

In der nachfolgenden Abbildung **6.3** ist das plattformcharakteristische Design des Aktivitätsindikators zu sehen.

Um diesen besser erkennen zu können, ist sein Hintergrund weiß gehalten. Während der Prozess der Applikation läuft, wird der Login Screen verdunkelt.

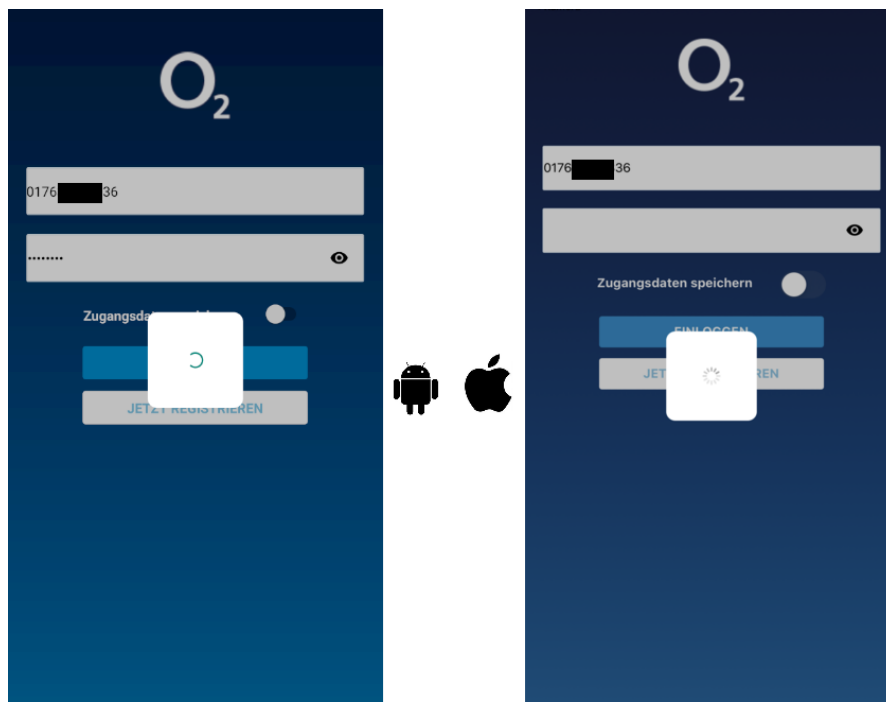


Abbildung 6.3: Die App signalisiert dem Verbraucher den laufenden Prozess

Einen weiteren Unterschied bildet das Zurücksetzen des unteren Eingabefeldes für das Passwort auf dem iOS-Gerät, nach Betätigen der Schaltfläche für den Login.

Verbraucherseite Inland

War der Login Versuch erfolgreich und konnten alle erforderlichen Daten aus dem Backend angefragt werden, navigiert die App zu ihrer Startseite.

Den Initialzustand bildet die Anzeigeeoption *Inland*.

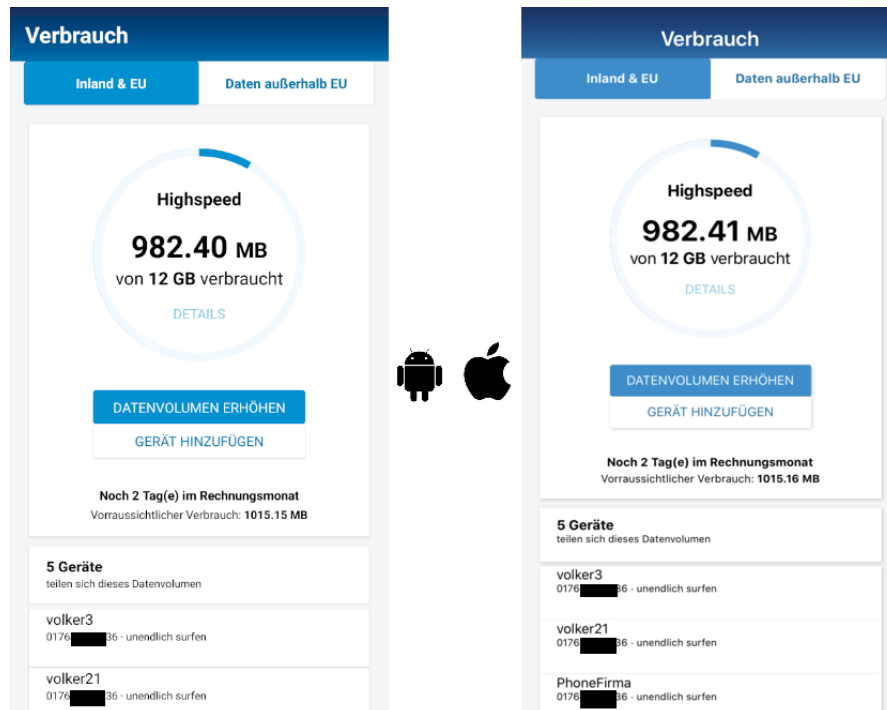


Abbildung 6.4: Die Positionierung des Seitentitels unterscheidet sich ebenfalls plattformcharakteristisch

Auf dem Homescreen in Abbildung 6.4 gezeigt, erhält der Verbraucher Informationen über seinen aktuellen Inlands-, sowie Auslandsverbrauch.

Es wird angezeigt, wie viele Tage bis zum neuen Rechnungsmonat noch vorliegen und wie viel Datenvolumen, gemessen am aktuellen Nutzungsverhalten, bis dahin voraussichtlich verbraucht wird.

Die in Blau hervorgehobene Schaltfläche gibt an, welche Option aktuell gewählt ist.

Des Weiteren wird ein dynamischer Inhalt erzeugt, der den Verbraucher darüber informiert, wie viele seiner unter dieser Kundennummer verwalteten Geräte sich dieses Datenvolumen teilen.

Zuletzt wird in dieser Applikation der Name des Tarifs und die primäre Geräte-MSISDN dazu eingeblendet.

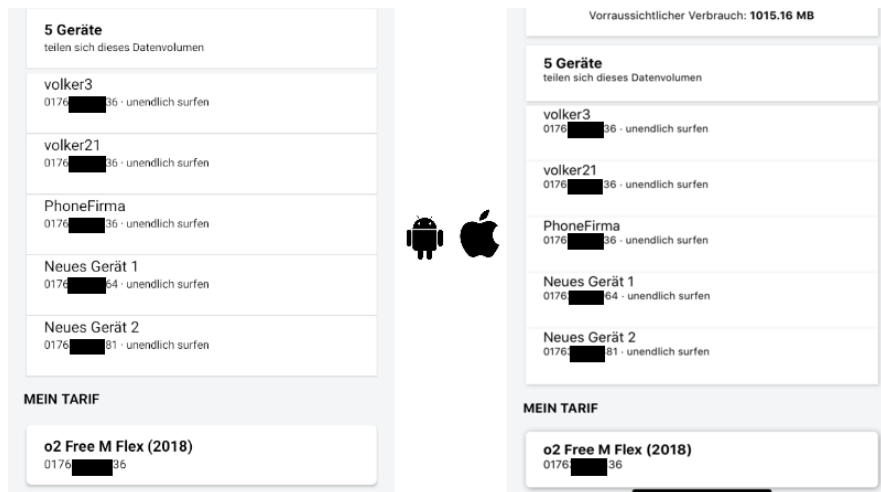


Abbildung 6.5: Über die Wischgeste *Scrollen* gelangt man zu den Informationen über den aktuellen Tarif

Verbraucherseite Ausland

Durch die obere, rechte Schaltfläche *Daten außerhalb EU* wird das User Interface der Seite mit den Informationen über den Auslandsverbrauch neu aufgebaut.

Alle weiteren Schaltflächen sind ohne Aktion. Sie dienen aktuell der Designvervollständigung.

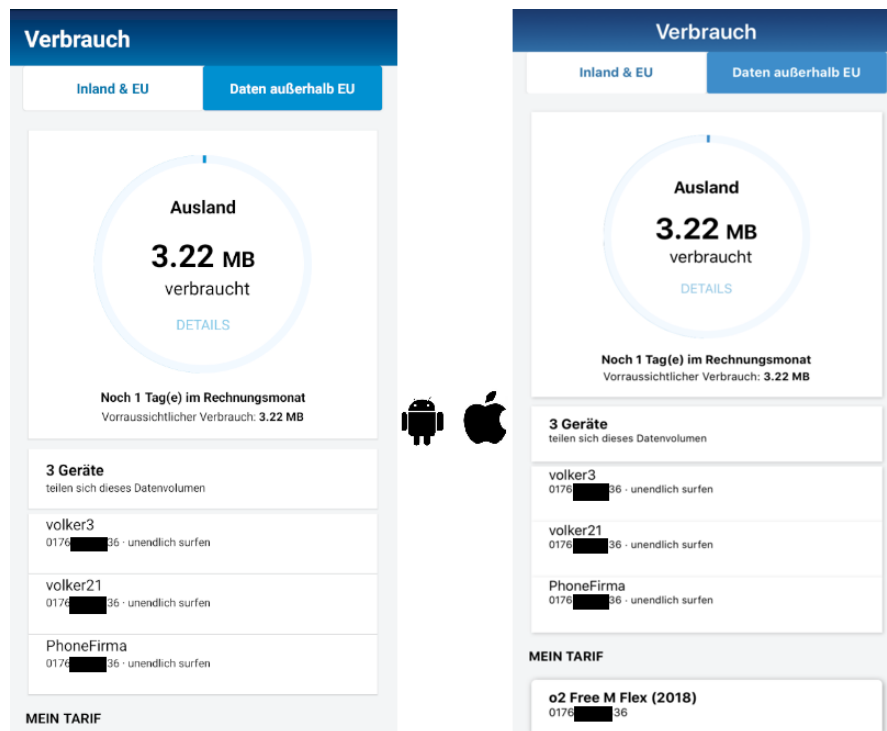


Abbildung 6.6: Durch Betätigen der Schaltfläche *Daten außerhalb EU* erhält man Informationen über den Auslandsverbrauch

Die Schaltflächen *DATENVOLUMEN ERHÖHEN* und *GERÄT HINZUFÜGEN* sind für diese Ansicht nicht vorgesehen.

Aus diesem Grund werden sie ausgeblendet.

Ebenfalls aktualisiert sich die Liste der Geräte, die zusammen die Höhe des angezeigten Auslandsverbrauchs erzeugen.

6.2 Fehlerszenarien

Entscheidend ist neben dem positiv ablaufenden Anwendung Szenario auch die Antwort auf die Frage, wie die Applikation auf Umgebungsfehler, oder fehlerhafte Eingaben des Anwenders reagiert.

Wie in Abschnitt **3.2.2** beschrieben, soll es zu keinem Absturz oder unerwartetem Verhalten der Software führen.

Darüber hinaus soll dem Anwender mitgeteilt werden, was schiefgelaufen ist.

6.2.1 Fehlende Internetverbindung

Ohne bestehende Internetverbindung hat die App keine Möglichkeit, mit dem Backend zu kommunizieren. Demzufolge ist auch kein Login möglich.

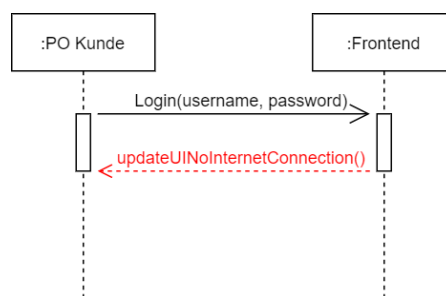


Abbildung 6.7: Eine nicht bestehende Internetverbindung wird dem Anwender direkt mitgeteilt

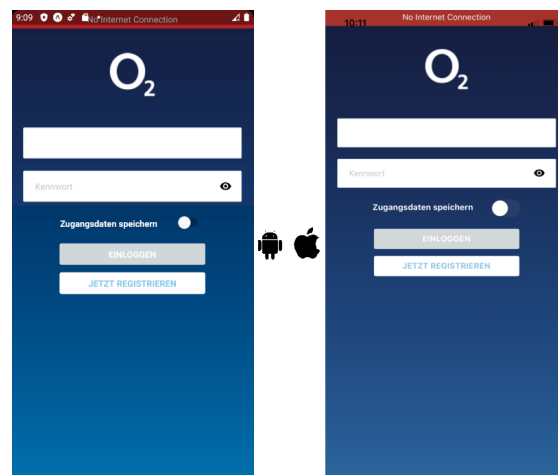


Abbildung 6.8: Die App prüft regelmäßig auf eine bestehende Internetverbindung

Der eingeblendete Informationstext erscheint jeweils oben in einem auffälligen Rotton, der dem Benutzer einen Umgebungsfehler signalisieren soll.

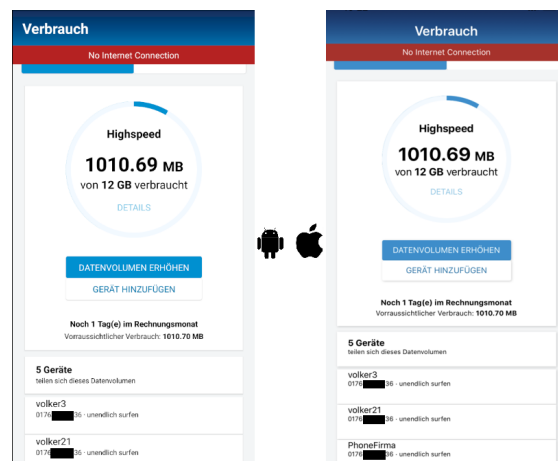


Abbildung 6.9: Bricht die Internetverbindung des Anwenders nach dem Login ab, wird ihm ebenfalls die Information angezeigt.

Sobald die Verbindung mit dem Internet wiederhergestellt ist, wird der Hinweis nicht weiter angezeigt und der Anwender kann die App wie gewohnt nutzen.

6.2.2 Nicht unterstützter Nutzer

Da die hybride Applikation aktuell nur für Postpaidkunden implementiert wurde, andere Vertragskunden sich jedoch über das Login-Backend anmelden können, werden die Logindaten in der App nach Erhalt zunächst auf die Vertragsart geprüft.

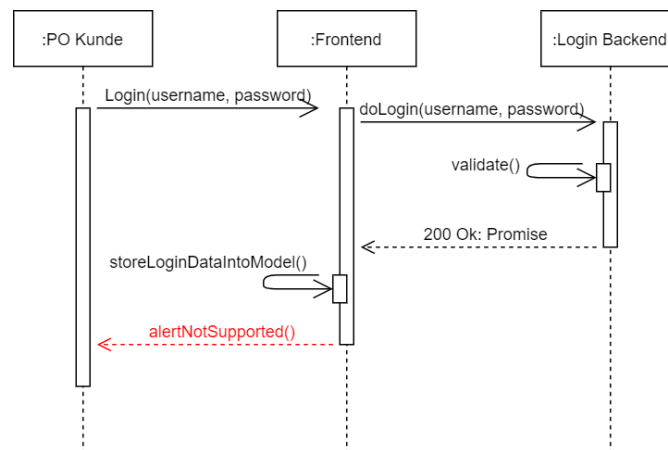


Abbildung 6.10: Ist der Nutzer kein Postpaidkunde wird sein Vertragsmodell von der App noch nicht unterstützt

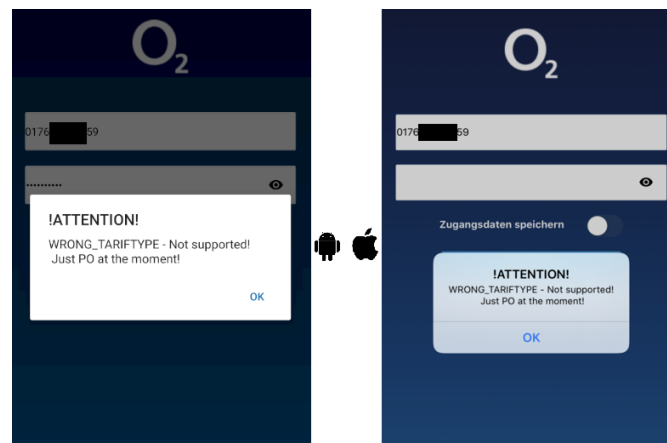


Abbildung 6.11: Die angezeigte Nachricht ist nativ

In der App wird die Information über die native Komponente *Alert* geliefert.

Nach Bestätigung hat der Kunde die Möglichkeit, sich mit einem von der App unterstützten Account anzumelden.

6.2.3 Fehler im Backendsystem

Das zentrale Backend, sowie das Verwaltungsbackend können temporär fehlerhaft, oder nicht erreichbar sein. Es ist wichtig zu wissen, welches Backend den Fehler verursacht hat.

Fehler zentrales Backend

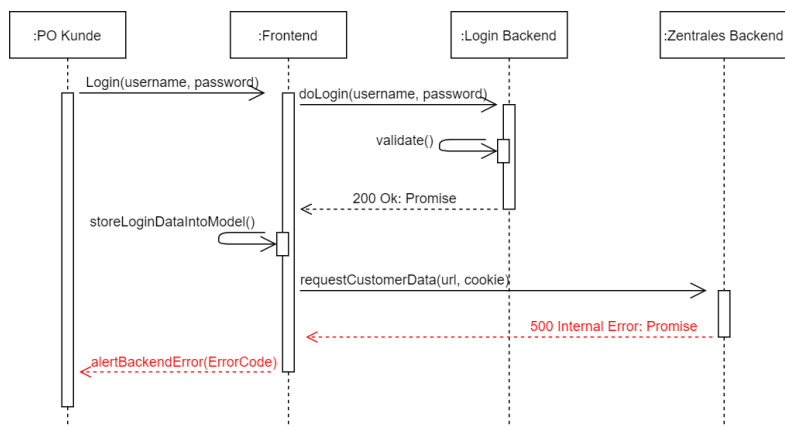


Abbildung 6.12: Das zentrale Backend agiert fehlerhaft

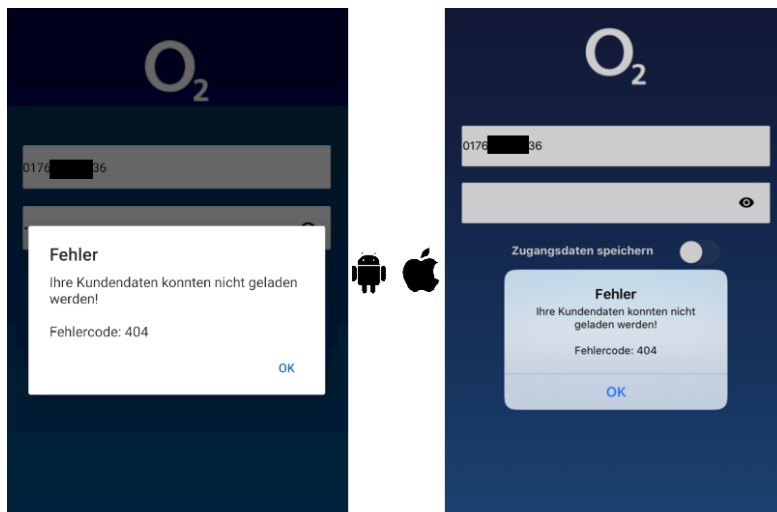


Abbildung 6.13: In diesem Fall kann das Backend aufgrund einer fehlerhaften Url nicht erreicht werden. Daher wird ein 404 Fehlercode angezeigt

Fehler Verwaltungs Backend

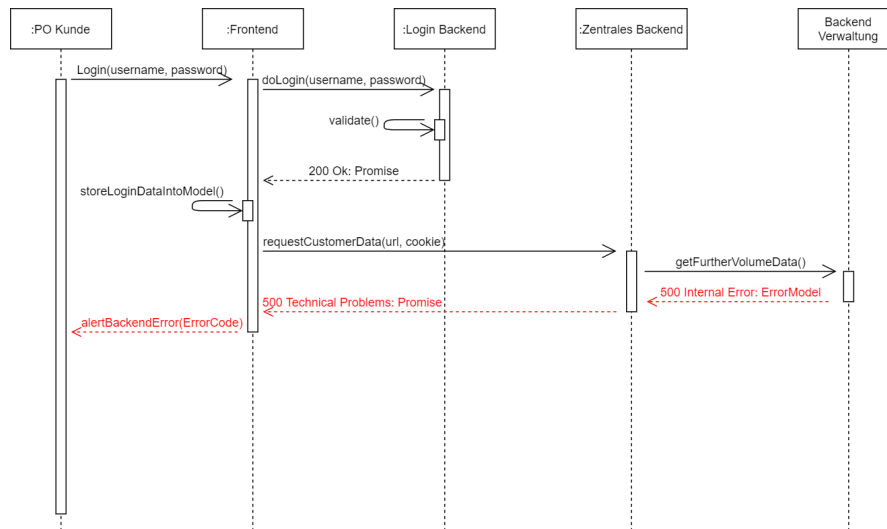


Abbildung 6.14: Der Fehler wird durch die verschiedenen Systeme kommuniziert

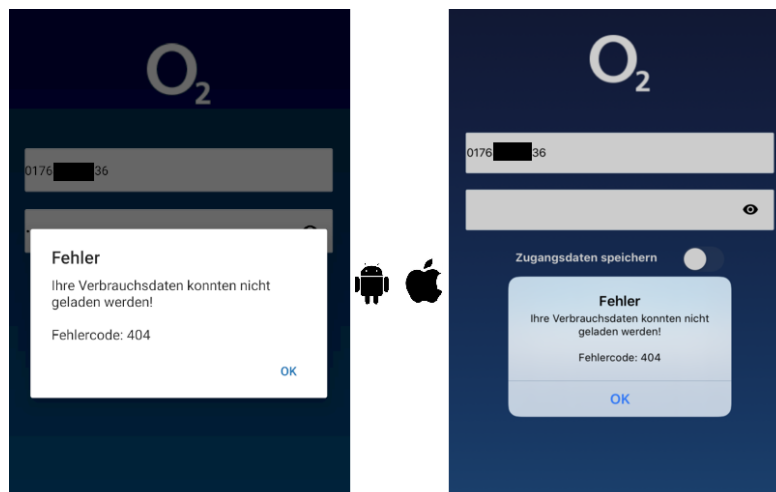


Abbildung 6.15: Information über die fehlgeschlagene Anfrage der Nutzungsdaten

Damit keine unvollständigen oder fehlerhaften Verbrauchsdaten angezeigt werden, wird der Anwender über den aufgetretenen Fehler informiert.

Als Konsequenz wird der Homescreen solange nicht aufgerufen, bis die Daten vollständig angefragt werden können.

6.2.4 Falscheingaben des Anwenders

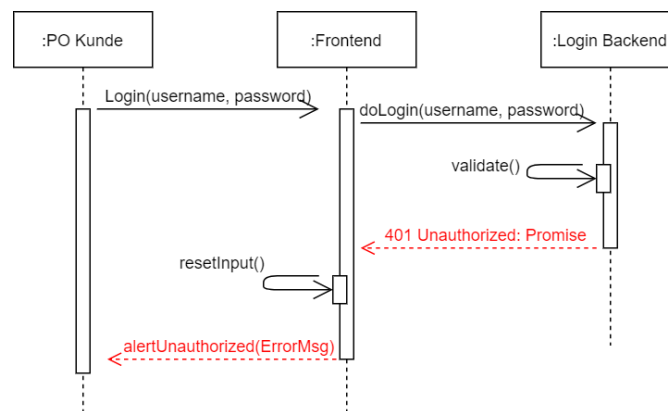


Abbildung 6.16: Bei fehlgeschlagenem Login Versuch werden die Informationen aus dem Login-Backend an den Anwender weitergeleitet

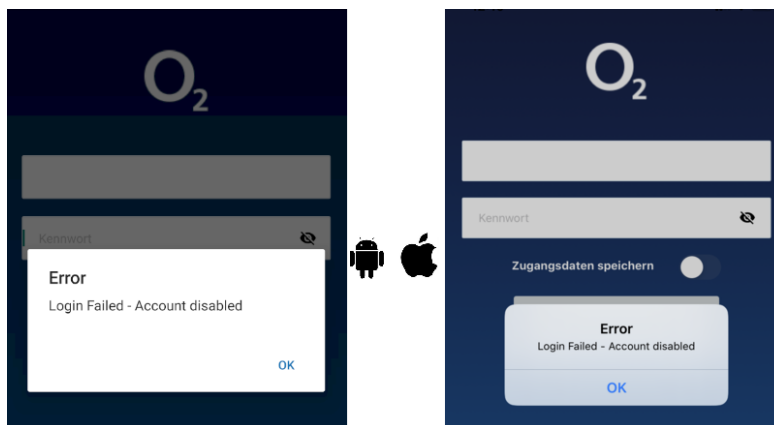


Abbildung 6.17: Der Anwender erhält Informationen über seinen fehlgeschlagenen Login Versuch

Wie in Abbildung 6.17 zu sehen werden direkt nach einem fehlgeschlagenen Anmeldeversuch beide Eingabefelder zurück gesetzt und der Anwender kann einen nächsten Versuch vornehmen.

Damit sind die in den Anforderungen beschriebenen Fehlerszenarien abgedeckt.

7 Fazit

7.1 Zusammenfassung

Mobile Applikationen sind heutzutage nicht mehr wegzudenken.

Hat man das Ziel, möglichst viele Endverbraucher zu erreichen, reicht die Betreuung auf einer Plattform nicht aus.

So muss als native Lösung eine Applikation für jede Plattform, die unterstützt werden soll, geschrieben werden.

Der Mehraufwand und die damit verbundenen Kosten steigen somit proportional für jede weitere unterstützte Plattform an.

Es müssen spezielle Ressourcen für jede Umgebung gefunden werden, da verwendete Technologien der einzelnen nativen Lösungen sich stark voneinander unterscheiden.

Darüber hinaus muss ein erhöhter Aufwand erbracht werden, um ungewollte Unterschiede in der Darstellung, sowie im Funktionsumfang zu vermeiden.

Die Verwendung eines hybriden Frameworks, wie React Native, ermöglicht es anhand einer Codebasis eine Applikation zu entwickeln, die auf mehr als nur einer Plattform ausgeführt werden kann.

In Addition nutzt dieses Framework native Komponenten für den Aufbau der UI, welches dem Anwender bei der Nutzung nicht auffallen lässt, eine hybride Applikation zu verwenden.

Am Ende dieses Projekts entstand eine hybride Applikation, die Kompatibilität mit Android, sowie iOS als Eigenschaft besitzt. Sie enthält native Module aus beiden Plattformen und ist in der Lage, mit dem bestehenden Backend zu kommunizieren.

Die Fehlersemantik dient dazu, Abstürze oder unerwartetes Verhalten zu vermeiden und sie mit dem Endverbraucher zu kommunizieren.

Das UI reagiert vertraut schnell und zu keinem Zeitpunkt verzögert.

7.1.1 Vorteile einer hybriden Lösung mit React Native

Der entscheidende Vorteil der Verwendung von React Native ist die Realisierung des Ziels, anhand einer Codebasis mehr als eine Plattform erreichen zu können.

React Native ist ein Open Source-Framework, was den Einstieg in die Thematik mit lizenzfreier Software ermöglicht.

Darüber hinaus kann man eigene Bibliotheken entwickeln und sie der Community für die Verwendung oder Mitarbeit bereitstellen, oder selbst auf geteilte Fremdprojekte zurückgreifen.

Der Entwicklungsprozess vereinfacht sich durch das Minimieren von zeit lastigen und fehleranfälligen Arbeitsschritten, in denen native Lösungen aus unterschiedlichen Codebasen gegen geprüft werden müssen.

Der gesamte Aspekt der Qualitätssicherung beschränkt sich somit auf ein Produkt.

Jeder an diesem Projekt arbeitende Entwickler kennt sich mit den verwendeten Technologien aus, welches eine flexiblere Arbeitsweise, gerade Ausfälle betreffend, im Gesamtteam möglich macht.

Mit Hilfe nützlicher Features, wie dem Hot Loading des Expo-Clients, ist es nicht mehr erforderlich, die gesamte Software neu zu kompilieren, um gemachte Änderungen sehen zu können.

Dies beschleunigt Entwicklungsphasen erheblich.

7.1.2 Einschränkungen

Von Facebook entwickelt, war React Native im mobilen Kontext zunächst ausschließlich für Android und iOS gedacht.

Eine React Native-Applikation ist also nicht auf einem mobilen Gerät mit beispielsweise *Microsoft Windows Phone*, oder seinem Vorgänger *Windows Mobile* lauffähig.

Auch wäre React Native zum jetzigen Zeitpunkt mit keiner bisher noch nicht erschienenen Plattform kompatibel. Die Motivation hierfür lässt sich noch nicht erkennen.

Die Möglichkeit, auf von der Community geschaffene Lösungen zurückgreifen zu können, besteht. Man sollte jedoch beachten, dass Bibliotheken von Dritten keine Garantie auf Wartung, Qualitätssicherung oder Sicherheit im Allgemeinen bieten.

Hybride Frameworks, wie React Native sind hoch performant, jedoch nicht in gleichem Maße wie eine native Lösung.

Dieser Aspekt sollte bei Projekten über harte Echtzeitsysteme berücksichtigt werden. In diesem konkreten Fall ist es sinnvoll, hardwarenahe Programmiersprachen in Betracht zu ziehen, anstelle von Sprachen, die einen Interpreter verwenden, oder die auf einer virtuellen Maschine ausgeführt werden.

Eine weitere Einschränkung findet sich im verwendeten Softwarestack. Kann, oder möchte man es nicht vermeiden native Frameworks zu verwenden, werden dementsprechend die dafür erforderlichen Werkzeuge benötigt.

Der in dieser Arbeit vorgestellte Technologie Stack reicht dann alleine nicht mehr aus.

7.1.3 Aussicht

Durch diese Arbeit kann eine erste Prognose hinsichtlich der Sinnhaftigkeit der Umsetzung des Gesamtprojektes gestellt werden.

Der hybride Prototyp beinhaltet den Kern der wesentlichen Funktionalitäten, wie die Kommunikation mit den Backendsystemen, sowie die Präsentation der Daten unter Verwendung nativer Komponenten.

Dadurch, dass React Native ein Open Source-Framework ist, können spezielle Designelemente, die nicht in offiziellen Bibliotheken enthalten sind, nachgebaut werden.

Dies war jedoch nicht Teil dieser Arbeit.

Im Laufe der durchgeführten Systemtests der App ließen sich zu keinem Zeitpunkt merkliche Nachteile in der Performanz feststellen, die User Experience (zu deutsch: Benutzererfahrung) (UX) hat sich demnach nicht verschlechtert.

Somit entsteht der Eindruck, dass der Umstieg der nativen Versionen auf die hybride Lösung langfristig gesehen, gewichtigere Vorteile mit sich bringt.

Da bei einer rein hybriden Lösung ein ganz anderer Sprachkern verwendet wird, nimmt der Umstieg jedoch Zeit und Ressourcen in Anspruch.

Demzufolge muss dieser Vorgang parallel zur Betreuung der nativen Systeme stattfinden, was einen hohen Aufwandsanstieg in den ersten Phasen bedeutet.

Aufgrund der Komplexität des bereits bestehenden Produktes, muss die hierfür eingeplante Zeit hoch geschätzt werden.

Ein weiterer wichtiger Punkt ist herauszufinden, ob und in wie weit bereits verwendete APIs mit React Native genutzt werden können, ohne dass ein partieller nativer Umstieg erfolgen muss.

Eventuell kann zunächst eine *Light*-Version der nativen Applikationen in Betracht gezogen werden, die ohne der Vielzahl an APIs auskommt und die Kernkompetenz erfüllt, die einem Teil der Endverbraucher schon ausreicht.

Literaturverzeichnis

- [1] : *From startup idea to software.* – URL <https://i.pinimg.com/originals/c4/21/f8/c421f8b6fe66f3f741f06f0d23f9a705.png>
- [2] AUGSTEN, Stephan: *Was ist Xamarin?* Vogel Communications Group GmbH u. Co. KG, 2019. – URL <https://www.dev-insider.de/was-ist-xamarin-a-872281>
- [3] BENJAMIN: *ECMAScript 6/ ECMAScript 2015.* 2016. – URL <https://www.netzbewegung.com/de/lab/ecmascript-6-ecmascript-2015>
- [4] CONGSTAR: *Handy-Betriebssysteme im Vergleich.* – URL <https://www.congstar.de/handys/im-vergleich/betriebssysteme-vergleich>
- [5] CRUXLAB, Inc.: *Xamarin vs Ionic vs React Native.* Cruxlab, Inc., 2017. – URL <https://cruxlab.com/blog/reactnative-vs-xamarin>
- [6] CUSSOL, Robin: *Build Standalone Expo .apk and .ipa with Turtle CLI.* 2020. – URL <https://www.robincussol.com/build-standalone-expo-apk-ipa-with-turtle-cli>
- [7] DROZDOV, Andrii: *ReactNative and Android: 64 bit.* – URL <https://medium.com/@andriidrozdov/reactnative-and-android-64-bit-new-google-play-market-rules-what-to-do-584b067d6f1a>
- [8] ELHADY, Hady: *How to deploy a React Native App for iOS and Android.* 2019. – URL <https://instabug.com/blog/react-native-app-ios-android>
- [9] ELIZABETH, Jane: *Top 5 JavaScript IDEs.* URL <https://jaxenter.com/top-5-javascript-ide-146609.html>, 2018
- [10] ELY LUCAS, Cam W.: *What is Ionic Framework?.* – URL <https://ionicframework.com/docs/intro>

- [11] ERIK, Behrends: *React Native/Native Apps parallel für Android und iOS entwickeln*. O' Reilly Media Inc., 2018. – 5 – 6 S. – URL <https://books.google.de/books?id=o6d3DwAAQBAJ&lpg=PP1&hl=de&pg=PP1#v=onepage&q&f=false>. – ISBN 9783960090663
- [12] ERIK, Behrends: *React Native/Native Apps parallel für Android und iOS entwickeln*. O' Reilly Media Inc., 2018. – 11 S. – URL <https://books.google.de/books?id=o6d3DwAAQBAJ&lpg=PP1&hl=de&pg=PP1#v=onepage&q&f=false>. – ISBN 9783960090663
- [13] ERIK, Behrends: *React Native/Native Apps parallel für Android und iOS entwickeln*. O' Reilly Media Inc., 2018. – URL <https://books.google.de/books?id=o6d3DwAAQBAJ&lpg=PP1&hl=de&pg=PP1#v=onepage&q&f=false>. – ISBN 9783960090663
- [14] ERIK, Behrends: *React Native/Native Apps parallel für Android und iOS entwickeln*. O' Reilly Media Inc., 2018. – 38 – 40 S. – URL <https://books.google.de/books?id=o6d3DwAAQBAJ&lpg=PP1&hl=de&pg=PP1#v=onepage&q&f=false>. – ISBN 9783960090663
- [15] ERIK, Behrends: *React Native/Native Apps parallel für Android und iOS entwickeln*. O' Reilly Media Inc., 2018. – 40 S. – URL <https://books.google.de/books?id=o6d3DwAAQBAJ&lpg=PP1&hl=de&pg=PP1#v=onepage&q&f=false>. – ISBN 9783960090663
- [16] ERIK, Behrends: *React Native/Native Apps parallel für Android und iOS entwickeln*. O' Reilly Media Inc., 2018. – 23 – 25 S. – URL <https://books.google.de/books?id=o6d3DwAAQBAJ&lpg=PP1&hl=de&pg=PP1#v=onepage&q&f=false>. – ISBN 9783960090663
- [17] GRÄBE, Prof. Dr. Hans-Gert: *Software-Qualitätsmanagement*. Universität Leipzig. – 4 – 10 S. – URL http://bis.informatik.uni-leipzig.de/de/Lehre/0405/SS/SQM/files?get=2005s_sqm_v_10.pdf
- [18] INC., 2020 F.: *JEST*. – URL <https://jestjs.io>
- [19] JOHNER, Prof. Dr. C.: *Funktionale Anforderungen versus nicht-funktionale Anforderungen*. Johner Institut. – URL <https://www.johner-institut.de/blog/iec-62304-medizinische-software/funktionale-und-nicht-funktionale-anforderungen>

- [20] KANTOR, Ilya: *An Introduction to JavaScript*. – URL <https://javascript.info>
- [21] KANTOR, Ilya: *Promise*. – URL <https://javascript.info>
- [22] LABS, Parewa: *Interpreter Vs Compiler: Difference between Interpreter and Compiler*. – URL <https://www.programiz.com/article/difference-compiler-interpreter>
- [23] LAHRES, Bernhard: *Praxisbuch Objektorientierung*. Rheinwerk Verlag GmbH. – URL http://openbook.rheinwerk-verlag.de/oo/oo_06_moduleundarchitektur_001.htm
- [24] MOZILLA ; CONTRIBUTORS individual: *HTTP request methods*. 2019. – URL <https://developer.mozilla.org/de/docs/Web/HTTP/Methods>
- [25] N., Patro: *Classical Inheritance vs Prototypal Inheritance*. URL <https://codeburst.io/javascript-inheritance-25fe61ab9f85>, 2018
- [26] SASCHA, Rezagholinia: *Hohe Qualität durch testgetriebene Entwicklung*. SQ Magazin, 2017. – 34 – 37 S. – URL https://www.sogeti.de/globalassets/germany/download/veroeffentlichungen-von-fachbeitragen/2017_tdd_rezagholinia_final.pdf
- [27] SCHANZE, Robert: *Was ist ein Betriebssystem?*. – URL <https://www.giga.de/extra/betriebssystem/tipps/was-ist-ein-betriebssystem-erklaerung-fuer-laien-profis>

A Anhang

Glossar

Backend Ein System welches für den Anwender transparent ist und für die Datensynchronisation und -verarbeitung, sowie die Benutzerauthentifizierung verantwortlich ist.

Codebasis Beschreibt alle Quelltextdateien in einem Softwareprojekt.

Compiler Auf deutsch *Kompilierer* dient der Übersetzung geschriebenen Quellcodes in für Computer verständliche Maschinensprache.

Cross-Plattform Plattformübergreifend.

ECMAScript Offizielle Spezifikation der umgangssprachlich genannten Programmiersprache JavaScript (vgl. [3]).

Feature Englisches Wort für Merkmal/ Funktion hier im Kontext eines Softwaresystems.

Framework Ein Rahmenwerk welches dem Entwickler ein Programmiergerüst zur Verfügung stellt, um bestimmte Probleme zu lösen.

Frontend Ein verwendeter Begriff für die grafische Benutzeroberfläche über die der Anwender mit dem System kommunizieren kann.

Header In diesem Kontext: Teil einer Nachricht mit Informationen über Absender und Empfänger.

Interface Builder Eine Software mit Hilfe derer grafische Benutzeroberflächen für Programme erstellt werden.

Know-How Das Wissen über prozedurale Vorgänge.

Log Ausgabe von Informationen laufender Prozesse eines Programms.

Open Source Bedeutet kostenlos. Bietet die Möglichkeit der Erweiterung des Umfangs durch Dritte.

Payload In diesem Kontext: Nutzdaten einer Nachricht.

Queue Englisch für Warteschlange.

Request Englisch für Anfrage.

Token Englisch für Wertmarke und wird als Begriff für ein Authentifizierungsmittel besonderer Anfragen verwendet.

Tool Englisches Wort für Werkzeug/ Hilfsmittel.

User Interface Grafische Schnittstelle zwischen Anwender und (Software-) System.

View Damit ist die Darstellungs- beziehungsweise Ansichtskomponente gemeint.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Hybride Appentwicklung

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original