# Konzeption und Entwicklung eines Backend-Driven UI Frameworks zur Umsetzung mobiler Cross-Plattform Anwendungen

Masterarbeit

ausgearbeitet von

Dominik Deimel

zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt an der

Technischen Hochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang
Medieninformatik

| | |
|---|---|
| Erster Prüfer/in: | Prof. Dr. Christian Kohls<br>Technische Hochschule Köln |
| Zweiter Prüfer/in: | Alexander Dobrynin, M.Sc.<br>Technische Hochschule Köln |

Gummersbach, im März 2023

**Adressen:** Dominik Deimel
ORCID:
https://orcid.org/0000-0002-9909-7097
dominik.deimel@th-koeln.de

Prof. Dr. Christian Kohls
Technische Hochschule Köln
Cologne Institute for Digital Ecosystems
Steinmüllerallee 1
51643 Gummersbach
christian.kohls@th-koeln.de

Alexander Dobrynin, M.Sc.
Technische Hochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
alexander.dobrynin@th-koeln.de

# Inhaltsverzeichnis

# Gender-Hinweis

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

# Kurzfassung

Smartphones und die Nutzung von mobilen Anwendungen gewinnen aufgrund der stetig voranschreitenden Digitalisierung weiter an Relevanz. Im April 2022 lag die Zahl der genutzten mobilen Endgeräte bei knapp 6 Milliarden, wobei die Hersteller Apple und Google einen Marktanteil von knapp 99 % einnehmen. Um als Entwickler eine möglichst große Zielgruppe anzusprechen, ist es daher wichtig, die eigene Anwendung auf möglichst vielen Plattformen zur Verfügung stellen zu können. Die native Entwicklung ist einer der am meist verbreitetste Entwicklungsansatz für mobile Anwendungen. Dort werden plattformspezifische Tools sowie native Programmiersprachen verwendet. Hierbei ergeben sich allerdings diverse Herausforderungen für Entwickler entlang des gesamten Entwicklungsprozesses. Neben der nativen Entwicklung existiert der Ansatz von Cross-Plattform. Hierbei handelt es sich um einen Sammelbegriff für verschiedene Ansätze, welche das Ziel verfolgen eine Anwendung durch die Nutzung einer einheitlichen Codebasis auf mehreren Plattformen zur Verfügung zu stellen. Einer dieser Unteransätze ist Backend-Driven UI, worin die gesamte Anwendung innerhalb eines strukturierten Datenformats beschrieben und serverseitig verwaltet wird. Der Client kann diese Daten dann dynamisch anfragen und zur Laufzeit in native UI überführen. Das Konzept von Backend-Driven UI wird bereits durch Unternehmen wie Airbnb oder SiriusXM eingesetzt, ist hier allerdings stark auf anwendungsbezogene Prozesse zugeschnitten. Es bedarf daher eines Backend-Driven UI Frameworks, welches es Entwicklern ermöglichen soll, mobile Anwendungen, ohne anwendungsspezifische Einschränkungen plattformübergreifend zu entwickeln. Im Rahmen dieser Arbeit wurde daher zunächst eine plattformunabhängige und erweiterbare Architektur eines solchen Frameworks entwickelt und in Form eines ersten Prototyps auf ihre Umsetzbarkeit geprüft. Insgesamt stellt der Prototyp eine mögliche Umsetzung von Backend-Driven UI dar und ist in der Lage erste Anwendungsszenarien abzudecken. Es konnten dennoch Bereiche ermittelt werden, welche im Zuge zukünftiger Entwicklung ausgebaut werden können. Dazu zählt unter anderem das Umsetzen von serverseitig bestimmter Anwendungslogik sowie der Umgang mit Unterschieden im plattformspezifischen Funktionsumfang.

# Abstract

Smartphones and the use of mobile applications continue to gain relevance due to the steady advance of digitalization. In April 2022, the number of mobile devices in use was just under 6 billion, with the manufacturers Apple and Google holding a market share of just under 99 %. In order to address the largest possible target group as a developer, it is therefore important to be able to make your own application available on as many platforms as possible. One of the most common development approach for mobile applications is native development, in which platform-specific tools and programming languages are used. However, this presents various challenges for developers along the entire development process. In addition to native development, there is also the cross-platform approach. This is a collective term for various approaches that pursue the goal of making an application available on multiple platforms by using a uniform code base. One of these sub-approaches is Backend-Driven UI, whereby the entire application is described within a structured data format and managed on the server side. The client can then dynamically request this data and transform it into native UI at runtime. The concept of Backend-Driven UI is already used by companies such as Airbnb or SiriusXM, but here it is strongly tailored to application-related processes. Therefore, a Backend-Driven UI framework is needed, which should enable developers to develop mobile applications, without application-specific restrictions, across platforms. In the context of this work, a platform-independent and extensible architecture of such a framework was developed and tested for its feasibility in the form of a first prototype. Overall, the prototype represents a possible implementation of backend-driven UI and is capable of covering initial application scenarios. Nevertheless, it was possible to identify areas that can be expanded in the course of future development. This includes, among other things, the implementation of server-side application logic and dealing with differences in platform-specific functionality

# 1. Einführung

Zu Beginn der Arbeit soll zuerst eine Einführung erfolgen. Dafür wird zunächst die Problemstellung sowie die Motivation erläutert. Anschließend erfolgt die Zielsetzung und die Vorstellung der Forschungsfragen. Abschließend wird dann ein Überblick über das weitere Vorgehen sowie den inhaltlichen Aufbau der Arbeit gegeben.

## 1.1. Problemstellung

Durch die stetig voranschreitende Digitalisierung spielen Smartphones eine immer größere Rolle und sind mittlerweile ein essenzieller Bestandteil im Leben vieler Menschen Li u. a. (2020). Im April 2022 liegt die Zahl der weltweit genutzten Smartphones bei knapp 6 Milliarden Geräten (Rogers u. Gratch, 2022). Durchgesetzt haben sich primär die Hersteller Apple und Google, respektive die Betriebssysteme iOS und Android, welche einen gemeinsamen weltweiten Marktanteil von knapp 99 % einnehmen. Die Popularität von Smartphones begründet sich zudem durch die Verwendung von mobilen Anwendungen. Hierbei handelt es sich um Anwendungen, welche primär auf mobile Betriebssysteme zugeschnitten sind und über Online-Stores direkt auf das Endgerät heruntergeladen werden können (Fayzullaev, 2018). Im Jahr 2021 konnte der Apple AppStore ein Portfolio von ca. 2,2 Millionen Anwendungen vorweisen, der Google PlayStore, als direkter Konkurrent, sogar ca. 3 Millionen Anwendungen. Der gemeinsame Umsatz der beiden Online-Stores lag bei knapp 135 Milliarden Dollar (Chan, 2021). Für Entwickler ist es wichtig, die eigene Anwendung einer möglichst großen Zielgruppe zur Verfügung zu stellen, weshalb entsprechend beide Plattformen für die Bereitstellung relevant sind. Hierbei ergeben sich für Entwickler diverse Herausforderungen, welche während des kompletten Entwicklungszyklus auftreten (Rucker, 2021).

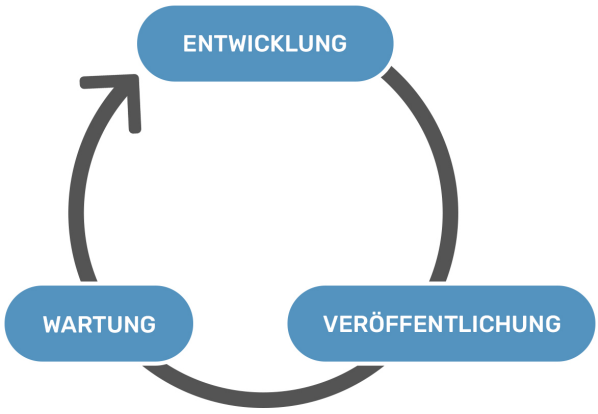



Abbildung 1.1.: Entwicklungszyklus einer Anwendung nach Rucker (2021)

Der Entwicklungszyklus einer Anwendung, dargestellt in Abbildung 1.1, ist ein iterativer Prozess, bestehend aus drei Teilbereichen. Im Teilbereich der Entwicklung geht es zunächst um die grundlegende Konzeption und Entwicklung der Anwendung. Im Anschluss an den Entwicklungsprozess folgt die Veröffentlichung und anschließend der Bereich der Wartung. Das Beheben von Fehlern oder das Implementieren von zusätzlichen Features kann dann wieder einen neuen Entwicklungsprozess starten und so die nächste Iteration des Entwicklungszyklus beginnen. Der Entwicklungszyklus ist bei allen Arten von Anwendungen vertreten, jedoch heben sich mobile Anwendungen besonders im Rahmen ihrer Veröffentlichung ab (Rucker, 2021). Während Desktop-Anwendung meist auf proprietären Webseiten zum Kauf oder in Form eines Abo-Modells zur Verfügung stehen, werden mobile Anwendungen über die nativen Online-Stores der Plattformbetreiber vertrieben (Lamhaddab u. a., 2019). Doch auch in anderen Entwicklungsphasen können sich Hürden und Herausforderungen bei der mobilen Entwicklung ergeben (Rucker, 2021).

Zu Beginn des Entwicklungszyklus folgt der Entwicklungsprozess. Zur Umsetzung von iOS-Apps wird die Programmiersprache Swift verwendet, welche die vorherige Sprache Objective-C nahezu vollständig abgelöst hat. Als Entwicklungsumgebung wird seitens Apple die IDE Xcode vorgegeben, welche ausschließlich auf macOS Geräten zur Verfügung steht. Für die Entwicklung von Android-Apps wird seit 2017 offiziell die Sprache Kotlin verwendet, welche im Android-Kontext ebenfalls auf der Java-Virtuell-Machine aufbaut und so eine Interoperabilität zum Vorgänger Java bietet (Fayzullaev, 2018). Als IDE kommt das von JetBrains entwickelte Android Studio zum Einsatz (Góis Mateus u. Martinez, 2019). Als Resultat aus den plattformspezifischen Vorgaben ergeben sich wesentlich höhere Anforderungen an die Entwickler, da nun ein entsprechendes Fachwissen für beide Programmiersprachen sowie Kenntnisse über die jeweiligen IDEs benötigt wird (Biørn-Hansen u. a., 2018). Zudem wird für die Verwendung von Xcode spezielle Peripherie, in Form eines macOS Geräts, benötigt. Dies, in Kombination mit dem zusätzlichen Entwicklungsaufwand für zwei Plattformen, resultiert in einem höheren benötigten Budget (Biørn-Hansen u. a., 2019).

Im Anschluss an die Entwicklung folgt die Veröffentlichung in den jeweiligen App-Stores. Bevor die Anwendung den Nutzern allerdings zur Verfügung steht, erfolgt zunächst ein Review-Prozess vonseiten des jeweiligen Plattformbetreibers. Ausgehend von einer Veröffentlichung auf beiden Plattformen, kann dieser Prozess, abhängig vom Umfang der Anwendung, einige Zeit in Anspruch nehmen. Als Resultat ergibt sich eine Verlangsamung des weiteren Entwicklungsprozesses, da so nicht nur die generelle Iteration des Entwicklungszyklus, sondern auch die Feedbackschleife zwischen Entwickler und Nutzer verzögert wird (Rucker, 2021).

Auch nach der Veröffentlichung einer Anwendung muss diese weiterhin gewartet und aktuell gehalten werden, um sich auf dem stark umkämpften mobilen Markt durchzusetzen. Zur regelmäßigen Wartung zählt zunächst das Beheben von Bugs sowie das Schließen von Sicherheitslücken (Hassan u. a., 2020). Ein weiterer wichtiger Aspekt ist allerdings das Erweitern der Anwendung mit neuen Funktionalitäten. Dies ist besonders wichtig, um bestehende Nutzer weiter an die Anwendung zu binden, neue Nutzer zu erreichen und die allgemeine Kundenzufriedenheit beizubehalten (Li u. a., 2020). In der Anwendung angezeigte Daten, wie Namen, Zahlen oder Inhalte von Tabellen, können bereits zur Laufzeit von einem Server angefragt und so ohne weiteren

Aufwand dynamisch angepasst werden (Rucker, 2021). Für größere Anpassungen des *User Interfaces* (UI) sowie Änderungen der Funktionalität ist allerdings weiterhin ein vollständiges Update der App und somit eine weitere Iteration des Entwicklungszyklus notwendig (Biørn-Hansen u. a., 2019).

Doch selbst nach der Autorisierung und Veröffentlichung des Updates können Entwickler nicht von einer sofortigen Adaption ausgehen, da Updates häufig unregelmäßig, stark verzögert oder gar nicht von Nutzern durchgeführt werden (Li u. a., 2020). Durch die unterschiedlichen Softwarestände kann es zu einer Fragmentierung der App-Version unter den Nutzern kommen, da diese nun verschiedene Softwarestände nutzen. Diese unterschiedlichen Softwarestände müssen vonseiten der Entwickler unterstützt werden und resultieren ebenfalls in einem Mehraufwand. Letztlich steht die Anwendung oder ein entsprechendes Update nach der Veröffentlichung allen Nutzern der jeweiligen Plattform zur Verfügung. Die Bereitstellung einzelner Funktionalitäten für eine spezifische Zielgruppe ist somit nicht möglich und würde einen starken Overhead innerhalb der Anwendung erfordern. Dies gilt beispielsweise für ein spezifisches Testing von Zielgruppen. (Maximo, 2020).

Zusammengefasst ergeben sich also einige Hürden und Herausforderungen, welche nicht zuletzt in steigenden technischen Anforderungen an die Entwickler, den Bedarf an Peripherie sowie Budget und dadurch einem erhöhten Entwicklungs- und Kostenaufwand resultieren. Dennoch ergeben sich auch Vorteile in der Unterstützung aller genannten Plattformen, insbesondere ihr großer Marktanteil und das damit verbundene Einkommenspotential, weshalb diese Hürden meist in Kauf genommen werden müssen.

## 1.2. Motivation

Die in Kapitel 1.1 beschriebenen Herausforderungen basieren auf der Nutzung des nativen Entwicklungsansatzes. Der Begriff „Nativ“ im Kontext der mobilen Anwendungsentwicklung beschreibt ein Vorgehen, bei welchem eine Anwendung vollständig mit der jeweils plattformspezifischen Programmiersprache und Entwicklungsumgebung umgesetzt wird (Biørn-Hansen u. a., 2019). Es existieren aber auch alternative Entwicklungsansätze wie die Cross-Plattform-Entwicklung, welche den genannten Schwächen entgegenwirken sollen. Der Begriff „Cross-Plattform“ beschreibt allerdings keinen konkreten Entwicklungsansatz, sondern stellt vielmehr ein Konzept dar, welches die Umsetzung einer Anwendung für mehrere Plattformen, auf Basis einer einzigen, einheitlichen Code-Basis unterstützt. Dazu handelt es sich bei Cross-Plattform um einen Sammelbegriff für diverse Unteransätze, wie dem hybriden oder interpretierten Entwicklungsansatz (Hassan u. a., 2020).

Ein weiterer Unteransatz der Cross-Plattform-Entwicklung ist Backend-Driven UI. Konkret beschreibt dieser die Umsetzung von mobilen Anwendungen, indem die Darstellung des User-Interfaces sowie die ausgeführte Anwendungslogik auf der Antwort eines Servers basiert (Rucker, 2021). Genauer wird hier der gesamte Aufbau der UI, und nicht nur einzelne darzustellende Daten, serverseitig aufbereitet und in Form von einer einheitlichen Datenstruktur, beispielsweise in Form einer Markup-Sprache, an den Cli-

ent gesendet. Der Client selbst, in diesem Fall die mobile Anwendung, umfasst nur noch Anwendungslogik zum Verarbeiten dieser Struktur und muss diese dann in eine native UI überführen. Die Verlagerung der Gestaltung des User-Interfaces und der Anwendungslogik vonseiten des Clients auf einen Server erlaubt es dynamisch Änderungen vorzunehmen, ohne dass dediziertes Update der Anwendung benötigt wird. (Birch, 2020). Zudem ist es möglich, spontan neue Funktionalität in die Anwendung einzubringen (Rucker, 2021).

Viele Herausforderungen, welche sich bei der nativen Entwicklung von mobilen Anwendungen ergeben, können durch den Einsatz von Backend-Driven UI gemindert oder vollständig umgangen werden. Zunächst reduziert sich der allgemeine Entwicklungsaufwand und das dafür benötigte Budget, da die gesamte UI nun einmalig serverseitig bestimmt wird und anschließend plattformunabhängig an Clients verteilt werden kann (Maximo, 2020). Dazu können Anpassungen an der eigentlichen Anwendung dynamisch serverseitig vorgenommen werden(Birch, 2020). Der Rollout eines App-Updates ist somit nicht mehr notwendig. Eine Ausnahme stellt hier die Anpassung der clientseitigen Parser-Pipeline dar. Vonseiten der Entwickler wird allerdings weiterhin ein entsprechendes Vorwissen für die Entwicklung des Parsers sowie die serverseitige Generierung eines User-Interfaces vorausgesetzt (Rucker, 2021).

Das Konzept von Backend-Driven UI wird bereits in der Praxis von Unternehmen wie Airbnb mit ihrem System „Lona“ oder SiriusXM eingesetzt (Maximo, 2020). Doch obwohl beispielsweise der Quellcode des Lona-Systems in Form eines öffentlichen Repositorys zur Vorschau einsehbar ist, stellen diese Systeme keine frei verwendbaren Frameworks dar und sind nicht explizit für die Nutzung von externen Entwicklern vorgesehen (AirBnB, 2017). Grund dafür ist die stark domänenspezifische Auslegung auf unternehmens- oder anwendungsspezifische Prozesse und Aufgaben (Maximo, 2020).

Dennoch zeigt die Entwicklung solcher Systeme, besonders durch in der Industrie etablierte Unternehmen, dass ein generelles Interesse an dem Konzept von Backend-Driven UI besteht. Gerade bei der Umsetzung von Cross-Plattform-Anwendungen kann der Einsatz von Backend-Driven UI den Entwicklungsaufwand reduzieren sowie die anschließende Wartung optimieren. Es bedarf allerdings eines Frameworks, welches im Vergleich zu Systemen wie Lona, domänenunabhängig eingesetzt werden kann. Zudem sollte besonders die serverseitige Gestaltung eines User-Interfaces durch ein solches Framework abstrahiert und so die Wissensanforderungen an Entwickler reduziert werden.

## 1.3. Zielsetzung

Im Rahmen der folgenden Arbeit soll ein Framework für die Umsetzung einer mobilen Cross-Plattform-Anwendung mit Backend-Driven UI konzipiert und anschließend prototypisch umgesetzt werden. Das Framework soll es ermöglichen, domänenunabhängig mobile Anwendungen mit Backend-Driven UI umzusetzen. Konzeptionell soll eine Architektur entworfen werden, welche alle benötigten Komponenten umfasst und dazu plattformunabhängig umgesetzt werden kann, um so die iOS- und Android-Plattform, mit ihren zugehörigen nativen Toolsets, in gleichem Maße abzudecken. Dazu soll die

Konzeption sowohl die Bereiche des Clients und des Servers, als auch der strukturierten Darstellung einer Anwendung betrachten.

Im Anschluss an die Konzeption soll eine prototypische Umsetzung des Frameworks für eine der aufgeführten Plattformen erfolgen. Die Umsetzung soll sich auf die Erprobung und Umsetzbarkeit des Konzepts fokussieren, weshalb hier kein vollständiger Funktionsumfang angestrebt wird.

Folglich sollen im Rahmen der Arbeit die folgenden Forschungsfragen beantwortet werden:

- Wie kann ein Framework zur Umsetzung von mobilen Anwendungen mit Backend-Driven UI aussehen?

Diese Frage dient als Leitmotiv der Arbeit und soll am Ende durch die erfolgte Konzeption und prototypische Umsetzung eines Backend-Driven UI Frameworks beantwortet werden.

- Wie kann eine plattformunabhängige Architektur aussehen, um sowohl die iOS- als auch Android-Plattform in gleichem Funktionsumfang, in Bezug auf die Darstellung von UI und Anwendungslogik, zu unterstützen?

Das Ziel soll es sein Backend-Driven UI im Rahmen der Cross-Plattform-Entwicklung einzusetzen. Daher ist es wichtig, dass ein entsprechendes Framework beide genannten Plattformen im gleichen Funktionsumfang unterstützt. Dies bezieht sich sowohl auf die Darstellung von UI-Komponenten als auch die Umsetzung von serverseitig bestimmter Anwendungslogik.

- Wie kann eine entsprechende Architektur so modular aufgebaut sein, dass Entwickler neben grundlegenden Funktionen auch dynamisch Erweiterungen vornehmen können?

Das Framework soll zunächst einen grundlegenden Funktionsumfang bereitstellen. Um Einschränkungen bei der Entwicklung allerdings zu vermeiden, muss eine modulare Architektur gewählt werden, welche bei Bedarf dynamisch erweitert werden kann.

- Wie kann eine prototypische Umsetzung des Frameworks technologisch aussehen?

Durch eine prototypische Umsetzung soll im Anschluss an die Konzeption zunächst die Umsetzbarkeit des Konzepts geprüft werden. Dazu soll eine Bestimmung geeigneter technischer Rahmenbedingungen erfolgen.

## 1.4. Vorgehen

Nach Erläuterung der Ausgangssituation sowie Beschreibung der Zielsetzung soll im Folgenden das weitere Vorgehen der Arbeit beschrieben und ein kurzer Überblick über die folgenden Kapitel gegeben werden. Zur besseren Veranschaulichung ist eine Übersicht zum Vorgehen in Abbildung 1.2 dargestellt.

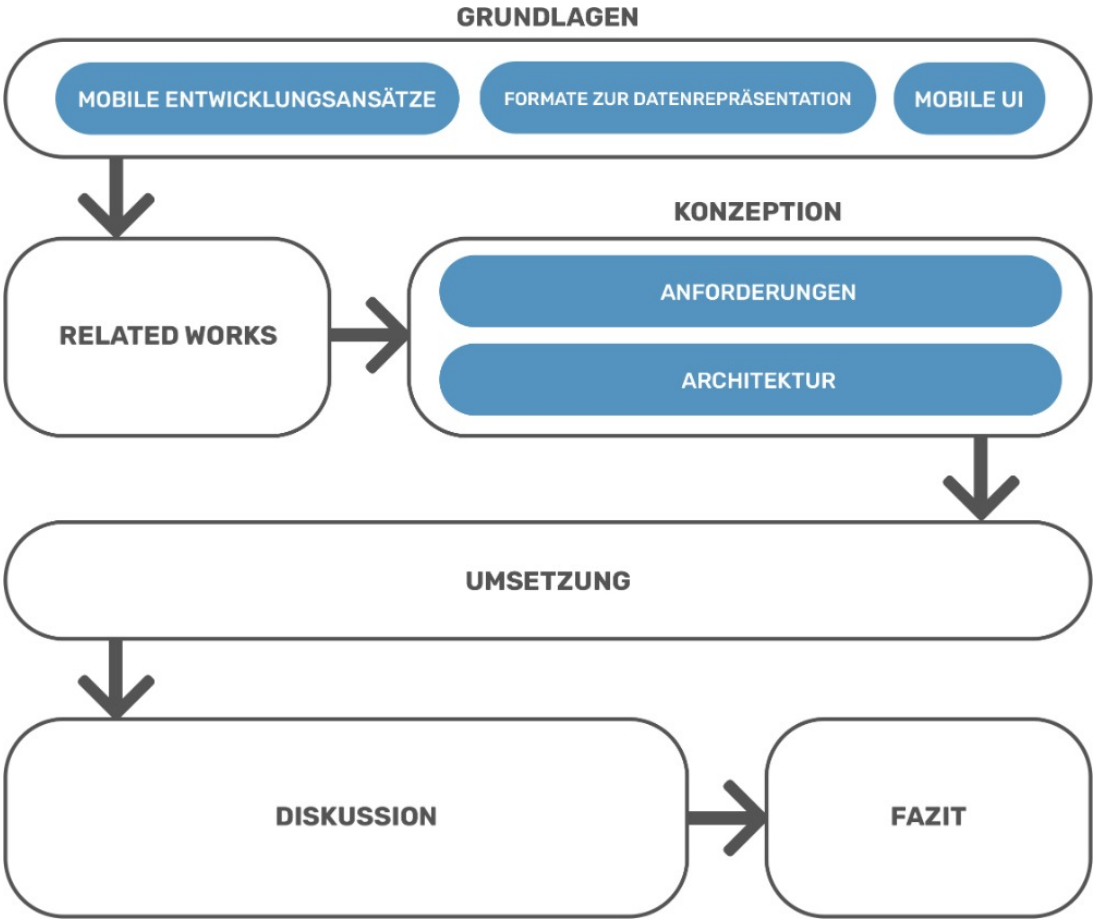



Abbildung 1.2.: Vorgehen der Arbeit

Zunächst muss eine gemeinsame Wissensbasis geschaffen werden. Hierfür soll im Rahmen der Grundlagen eine Vorstellung bestehender mobiler Entwicklungsansätze erfolgen. Aufbauend auf dem Ansatz von Backend-Driven UI werden zudem verschiedene Formate zur Datenrepräsentation und der Bereich von mobiler UI betrachtet. Um eine Ausgangslage für die weitere Entwicklung zu schaffen, erfolgt zusätzlich eine Vorstellung verwandter Arbeiten. Im Anschluss folgt nun die Konzeption des Frameworks. Hierfür werden zunächst Anforderungen auf Basis der vorgestellten Grundlagen und verwandten Arbeiten ermittelt und basierend darauf eine Architektur abgeleitet. Diese sollen dann im Rahmen der Entwicklung technisch umgesetzt werden. Der Entwicklungsprozess wird hierfür entlang der drei Hauptkomponenten des Backend-Driven UI Ansatzes betrachtet. Abschließend sollen dann die Ergebnisse der Konzeption und Entwicklung diskutiert und die Arbeit in einem Fazit zusammengefasst werden.

# 2. Grundlagen

Zur Schaffung einer gemeinsamen Wissensbasis für die anschließende Konzeption und prototypische Umsetzung eines Frameworks, sollen als Nächstes die Grundlagen vorgestellt werden. Zunächst werden dafür bestehende mobile Entwicklungsansätze betrachtet. Dazu soll dann aufbauend für die spätere Umsetzung auf verschiedene Formate zur Datenrepräsentation und den Bereich von mobiler UI eingegangen werden.

## 2.1. Mobile Entwicklungsansätze

Für die Entwicklung von mobilen Anwendungen stehen Entwicklern verschiedene Entwicklungsansätze zur Verfügung. Neben der nativen Entwicklung hat sich primär der Ansatz der Cross-Plattform-Entwicklung durchgesetzt. Im Folgenden soll nun eine Einführung in die bestehenden Ansätze gegeben werden, indem zunächst die native Entwicklung und anschließend der Cross-Plattform-Ansatz betrachtet wird. Abschließend soll das Konzept von Backend-Driven UI als mobiler Entwicklungsansatz genauer vorgestellt und in die bestehenden Entwicklungsansätze eingeordnet werden.

### 2.1.1. Nativ

Der native Entwicklungsansatz beschreibt das Nutzen von spezifischen Tools für die Umsetzung einer mobilen Anwendung anhand der gewählten Zielplattform. Spezifische Tools können hierbei konkrete Vorgaben bei der Verwendung von Programmiersprachen, aber auch vorausgesetzte Entwicklungsumgebungen sein. Die native Entwicklung stellt die grundlegendste Form und damit den ältesten Entwicklungsansatz dar. Trotz des Aufkommens von moderneren Verfahren, wie der Cross-Plattform-Entwicklung, wurde der native Ansatz im Jahr 2018 weiterhin für knapp 5,6 Millionen Apps verwendet (Biørn-Hansen u. a., 2019). Das folgende Kapitel soll die native Entwicklung genauer vorstellen und so eine Grundlage für die weiteren mobilen Entwicklungsansätze schaffen. Dafür erfolgt zunächst eine Betrachtung der plattformspezifischen Tools und Vorgaben. Anschließend sollen Vorteile und besonders entstehende Herausforderungen bei der Verwendung der nativen Entwicklung herausgearbeitet werden.

Die Nutzung des nativen Entwicklungsansatzes ist mit diversen Vorgaben verbunden, welche sich anhand der gewählten Zielplattform ergeben. Zunächst einmal ist die zu verwendende Programmiersprache plattformabhängig (Biørn-Hansen u. a., 2019). Bei iOS-Anwendungen wurde ursprünglich die Programmiersprache Objective-C vorausgesetzt, welche seit 2014 sukzessive durch die Apple eigene Sprache Swift abgelöst wurde. Bei Swift handelt es sich um eine multiparadigmatische Sprache, welche neben iOS-Anwendungen auch für iPadOS, macOS und weitere Apple-spezifische Betriebssysteme eingesetzt wird García u. a. (2015). Android-Anwendungen bauten ur-

sprünglich auf der Programmiersprache Java auf, welche besonders im Bereich von Desktop-Anwendungen immer noch weit verbreitet ist. Seit 2017 kann neben Java auch die von JetBrains und Google in Kooperation entwickelte Sprache Kotlin verwendet werden. Hierbei handelt es sich um eine plattformübergreifende, statisch typisierte Programmiersprache, welche nicht nur für die Android, sondern auch für Web- und Backend-Entwicklung genutzt werden kann. Dazu baut Kotlin, im Kontext von Android-Anwendungen, auf der *Java-Virtuel-Machine*(JVM) auf und kann daher interoperabel mit der Sprache Java in einem gemeinsamen Projekt verwendet werden. 2019 wurde Kotlin durch Google zur offiziellen Android-Sprache ernannt und hat damit Java im Android-Kontext abgelöst (Hunt, 2021).

Zusätzlich zu nativen Programmiersprachen existieren plattformspezifische Entwicklungsumgebungen. Für die iOS-Plattform und die Verwendung der Programmiersprache Swift wird die Apple eigene IDE *Xcode* vorausgesetzt (García u. a., 2015). Für die Entwicklung von Android-Anwendungen wird vonseiten Googles die von JetBrains entwickelte IDE *Android Studio* empfohlen. Beide Entwicklungsumgebung bieten Entwicklern diverse Funktionen an (Hunt, 2021). Dazu zählen beispielsweise Debugging-Tools, virtuelle Emulatoren für verschiedene Hardwarespezifikationen, ein Filesystem-Inspektor für emulierte Geräte und ein Navigation-Graph (Biørn-Hansen u. a., 2019).

Die Verwendung des nativen Entwicklungsansatzes ist mit diversen Vorteilen verbunden. Nutzer der Anwendungen profitieren zunächst von einer intuitiven User--Experience (UX) (El-Kassas u. a., 2017). Die Darstellung und Bedienung der Anwendung orientiert sich hierfür an den jeweiligen Design-Richtlinien, welche durch die gewählte Plattform vorgegeben sind. iOS-Anwendung folgen den Richtlinien der Apple Human-Interface-Guidelines. Die Android-Plattform orientiert sich an den Material Design-Guidelines (Ryan, 2021). Neben den Nutzern können auch Entwickler von dem nativen Entwicklungsansatz profitieren. Native Anwendungen können zunächst eine höchstmögliche Performance vorweisen. Möglich wird dies durch die Verwendung der bereits vorgestellten nativen Programmiersprachen, welche ein optimales Ansprechen der plattformspezifischen Architektur ermöglicht (Biørn-Hansen u. a., 2018). Dazu können Geräteschnittstellen genauer angesprochen werden, zu welchen beispielsweise die Kamera, GPS-Informationen, der Zugriff auf das Dateisystem sowie weitere Sensor-Daten zählen (El-Kassas u. a., 2017).

Es ergeben sich allerdings auch diverse Limitierungen und Nachteile bei der nativen Entwicklung. Native Anwendungen sind nicht interoperabel zwischen unterschiedlichen Plattformen. Es ist daher nicht möglich, eine Android-Anwendung auf einem iOS-Gerät zu installieren und umgekehrt. Grund hierfür sind zunächst Unterschiede in der Entwicklung. Die Verwendung unterschiedlicher Programmiersprachen, wie Kotlin und Swift, resultiert in unterschiedlichen Anforderungen an den Kompilierprozess. Dazu unterscheiden sich die Plattformen anhand ihrer Architekturen. Aufgrund dieser Unterschiede wird auch ein entsprechendes Vorwissen zu beiden Plattformen benötigt. Neben den Programmiersprachen müssen Entwickler auch mit plattformspezifischen Entwicklungsumgebungen umgehen können (El-Kassas u. a., 2017). Die IDE Xcode ist dazu noch alternativlos und wird von Apple vorgegeben (García u. a., 2015). Resultierend aus den erhöhten Anforderungen an die Entwickler, ergeben sich auch steigende Entwicklungskosten (El-Kassas u. a., 2017). Dazu zählt auch der Bedarf an benötigter Hardware, da die für die Entwicklung von iOS-Anwendung benötigten Tools, nur auf

macOS Geräten zur Verfügung stehen (García u. a., 2015). Zudem ist aufgrund der erhöhten technischen Anforderungen von einer Vergrößerung des Entwicklungsteams auszugehen. Aber auch in der Entwicklung selbst kann es zu Einschränkungen durch die native Entwicklung kommen. So erhöht sich das allgemeine Fehlerpotential innerhalb der Anwendung aufgrund der steigenden Menge an Programmcode, was zusätzlich in einem höheren Wartungsaufwand resultiert. Dazu kann Code syntaktisch, aufgrund der unterschiedlichen Programmiersprachen, aber auch semantisch nicht von einer auf die andere Plattform übernommen werden (El-Kassas u. a., 2017).

Zusammenfassend stellt der native Entwicklungsansatz einen Kompromiss aus Nutzen und Aufwand dar. Während native Anwendungen besonders Nutzern eine gute Performance und eine native UI bieten, stehen Entwickler vor diversen Herausforderungen. Hier ergeben sich besonders hohe Anforderungen aufgrund technischer und architektonischer Unterschiede zwischen den Plattformen. Dazu kommen plattformspezifische Vorgaben und Empfehlungen in Bezug auf Entwicklungstools, welche schlussendlich in einem erhöhten Entwicklungsaufwand resultieren. Der Einsatz des nativen Entwicklungsansatzes sollte daher stets kontextabhängig erfolgen. Besonders bei großen Anwendungen kann es schnell zu Dopplungen und semantischen Unterschieden im Code kommen. Dennoch ergeben sich besonders bei dem Ansprechen von nativen Geräteschnittstellen erhebliche Vorteile.

### 2.1.2. Cross-Plattform

Zur Überwindung der Herausforderungen, welche im Rahmen der nativen Entwicklung entstehen, ergeben sich stetig neue mobile Entwicklungsansätze, welche die native Entwicklung entweder erweitern oder ersetzen sollen Biørn-Hansen u. a. (2019). Eine Gruppe dieser alternativen Ansätze lässt sich dem Sammelbegriff des Cross-Plattform-Ansatzes oder der Cross-Plattform-Entwicklung zuordnen, welcher nach Biørn-Hansen u. a. (2018) wie folgt zusammengefasst werden kann:

> "The ultimate goal of cross-platform mobile app development is to achieve Native app performance and run on as many platforms as possible." (Biørn-Hansen u. a., 2018)

Das Ziel des Cross-Plattform-Ansatzes ist es, auf Grundlage einer einheitlichen Codebasis, eine Anwendung auf verschiedenen Plattformen zur Verfügung zu stellen. Dies soll die primären Nachteile der nativen Entwicklung ausgleichen und eine möglichst native Performance ermöglichen. Zur Umsetzung werden zusätzlich zur einheitlichen Codebasis als Ausgangspunkt verschiedene Kompilier-, Interpretations- oder andere Middlewareprozesse genutzt, um unterschiedliche Architekturen und Plattformen anzusprechen. Die Umsetzung von Cross-Plattform kann durch die Nutzung verschiedener Unteransätze erfolgen. Diese einzelnen Ansätze basieren auf unterschiedlichen Technologien sowie Paradigmen und müssen deshalb stets kontextabhängig eingesetzt werden (El-Kassas u. a., 2017). Aufgrund dieser technischen Unterschiede erfüllen sie das Ziel von Cross-Plattform auch zu unterschiedlichen Graden und bieten individuelle Vor- und Nachteile (Biørn-Hansen u. a., 2019). Im Folgenden werden nun einige der verbreitetsten Cross-Plattform-Ansätze vorgestellt, die dazugehörigen Anwendungskontexte

und verwendeten Technologien beschrieben sowie Vor- und Nachteile betrachtet. Die Auswahl der vorzustellenden Ansätze erfolgt auf Basis ihrer Präsenz in der Fachliteratur. Grundlage waren hier primär die Meta-Studie Biørn-Hansen u. a. (2018) sowie El-Kassas u. a. (2017).

### Hybrider Ansatz

Eine Ausprägung der Cross-Plattform-Entwicklung ist der hybride Entwicklungsansatz, welcher eine Mischung aus der nativen Entwicklung und den Prinzipien von Cross-Plattform darstellt, indem native Anwendungen mit Webkomponenten kombiniert werden. Konkret handelt es sich bei einer hybriden Anwendung zunächst um eine native Anwendung, welche anhand des nativen Entwicklungsansatzes, für Plattformen wie iOS und Android umgesetzt wird. Innerhalb der Anwendung wird dann eine Web-View implementiert. Hierbei handelt es sich um einen eingebetteten Browser, welcher es ermöglicht, Webanwendungen innerhalb einer nativen Anwendung auszuführen. Die eigentliche Logik und UI der Anwendung wird hier dann mit verschiedenen Web-Technologien wie HTML, CSS und JavaScript umgesetzt und mithilfe der Web-View dargestellt. Die native mobile Anwendung stellt bei dem hybriden Ansatz also lediglich einen Wrapper für die eigentliche Anwendung dar, welche als Webanwendung implementiert wird (Biørn-Hansen u. a., 2018).

Der wichtigste Bestandteil von hybriden Anwendungen ist die Kommunikation zwischen der nativen Anwendung und der Web-View. Ein Nachteil der Nutzung von Webanwendungen auf mobilen Endgeräten ist es, dass diese nur schwer auf Geräte APIs, wie beispielsweise das Dateisystem, zugreifen können. Die native Anwendung muss in diesem Fall als Bindeglied fungieren. Die Verbindung zwischen nativem Code und Web Code wird *Bridging* genannt und ist dafür verantwortlich, Nutzern ein möglichst natives App Erlebnis zu liefern. Zur Realisierung von Bridging wird ein *Foreign-Function Interface* (FFI) eingesetzt. Hierbei handelt es sich um eine Middleware, welche als Mediator zwischen den unterschiedlichen Codebasen fungiert und Kommunikation zwischen diesen ermöglicht (Biørn-Hansen u. a., 2018).

Die Verwendung des hybriden Entwicklungsansatzes eignet sich zunächst für konventionelle, servergestützte Anwendungen, also Webanwendungen, die regelmäßig mit einem serverseitigen Backend kommunizieren und so beispielsweise Ressourcen wie Bilder oder Texte anfragen. Dazu können hybride Anwendungen aber auch auf in sich geschlossene Webanwendungen zurückgreifen, welche vollständig lokal und ohne Kommunikation mit einem externen Backend auf dem Client ausgeführt werden El-Kassas u. a. (2017). Die native Anwendung, welche in beiden Fällen benötigt wird, bleibt immer sehr schlank, da sie ausschließlich als Wrapper für die Web-View fungiert. Lediglich bei der Nutzung einer lokalen Webanwendung, kann zusätzlicher Speicherplatz für benötigte Ressourcen anfallen. Das Verwenden einer servergestützten Webanwendung erlaubt es zusätzlich dynamisch Änderungen an der eigentlichen Anwendung vorzunehmen, ohne die native Anwendung aktualisieren zu müssen. Durch den Einsatz von Bridging und einem FFI kann eine hybride Anwendung zudem auch auf native Geräte-APIs zugreifen und so beispielsweise Sensordaten wie GPS Informationen an die Webanwendung weitergeben (Biørn-Hansen u. a., 2018).

Als Nachteil ergibt sich bei hybriden Anwendungen besonders die schlechtere Performance. Grund hierfür ist die zusätzliche Interpretierung von HTML und JavaScript Code innerhalb der Web-View sowie weitere benötigte Middleware. Die eigentliche Anwendung wird bei hybriden Anwendungen als Webanwendung, durch die Nutzung von Webtechnologien, implementiert. Die Umsetzung einer möglichst nativ aussehenden UI ist daher mit erhöhten Aufwand verbunden und würde ein tiefergreifendes Styling voraussetzen. Ein weiterer Nachteil ergibt sich aus der Verwendung einer servergestützten Webanwendung innerhalb der hybriden Anwendung, da diese eine aktive Internetverbindung benötigt. Native Anwendungen werden vor der Verwendung vollständig auf das Endgerät heruntergeladen und installiert und beziehen nur zusätzliche Ressourcen von einem Backend. Bei hybriden Anwendungen umfasst die native Anwendung allerdings nur einen Wrapper für die Web-View sowie zusätzliche Bridging-Komponenten. Die UI und Anwendungslogik der Webanwendung wird serverseitig verwaltet und muss bei der Nutzung vom Server angefragt werden. Alternativ muss hier ein geeignetes Caching Verfahren eingesetzt werden, welches ebenfalls einen erhöhten Aufwand darstellt (El-Kassas u. a., 2017). Zudem stellt die Verwendung einer Web-View innerhalb einer mobilen Anwendung ein erhöhtes Sicherheitsrisiko dar. Native Anwendungen laufen abgekapselt auf dem Endgerät und besitzen einen dedizierten Speicherbereich, auf welchem sie operieren dürfen. Eine Kommunikation mit anderen Apps oder dem Betriebssystem geschieht nur über die dafür vorgesehene Schnittstellen. Das Verwenden einer Web-View bietet allerdings ein volles Angriffspotential über das Web. So sind besonders die frühe Web-View-Implementierung des Android-Betriebssystems anfällig für *Cross-Site-Scripting* (XSS) Attacken (Biørn-Hansen u. a., 2018). Unter dem Begriff XSS versteht man das Injizieren von manipulierten Programmcode in eine Webanwendung. Hierbei wird eine legitime Anfrage von einem Client an einen Server abgefangen und statt der gewünschten Server Ressource eine manipulierte Ressource als Antwort zurückgegeben. Der Client betrachtet diese manipulierte Ressource weiterhin als legitime Serverantwort und führt nun den schadhaften Code aus (Sarmah u. a., 2018). Angreifern ist so nicht nur möglich, schadhafte Skripte innerhalb der Web-View auszuführen, sondern durch das Bridging auch native Geräteschnittstellen, anzusprechen. Dies erlaubt es Angreifern beispielsweise zusätzlich auf Kontakte, Fotos oder andere Geräteinformationen zuzugreifen. Zuletzt können weitere Sicherheitsrisiken bei der Nutzung von Drittanbieter Bibliotheken entstehen, da diese in vielen Fällen nur unzureichend oder gar nicht geschützt sind (Biørn-Hansen u. a., 2018).

Insgesamt stellt eine hybride Anwendung eine Alternative zu nativen Anwendungen dar. Vorteile ergeben sich hier besonders aus der einfachen Verbreitung für weitere Plattformen und der Möglichkeit, im Falle einer serverbasierten Webanwendung, dynamisch Anpassungen vorzunehmen. Da die eigentliche Anwendungslogik und UI aber weiterhin als Webanwendung bereitgestellt wird, gelten auch die hier vorliegenden Einschränkungen. So erfordert die Nutzung serverseitiger Anwendungen eine stabile Internetverbindung und kann selbst dann nicht die Performance einer nativen Anwendung erreichen. Dazu ist die Gestaltung einer nativen UI, aufgrund der Nutzung von Webkomponenten wie HTML, ebenfalls mit einem erhöhten Aufwand verbunden. Zusätzlich bietet die Nutzung einer Web-View ein Angriffsrisiko für XSS-Angriffe, durch welche sensible Nutzerinformationen abgefangen werden können.

### Interpretierter Ansatz

Der interpretierte Ansatz setzt, ähnlich wie der hybride Ansatz, auf die Verwendung von JavaScript zur Abbildung der Anwendungslogik. Im Gegensatz zu hybriden Anwendungen wird allerdings keine eingebettete Web-View verwendet, um eine Webanwendung innerhalb einer nativen Anwendung darzustellen, sondern JavaScript direkt innerhalb der nativen Anwendung interpretiert und ausgeführt (Biørn-Hansen u. a., 2018).

Die Grundlage einer interpretierten Anwendung ist zunächst ebenfalls eine native Anwendung, in welche der eigentliche Programmcode, in Form von Script-Sprachen, wie beispielsweise JavaScript, während des Entwicklungsprozesses eingefügt wird. Zur Laufzeit wird dieser dann direkt interpretiert und in native UI überführt. Für den Interpretationsprozess werden On-Device-Interpreter, also spezielle JavaScript Interpreter, welche bereits Teil des jeweiligen Betriebssystems sind, verwendet. Auf der iOS-Plattform wird hierfür *JavaScriptCore*[1] genutzt, während das Android-Betriebssystem auf *V8*[2] zur Interpretation zurückgreift. Ähnlich zur Verwendung einer Web-View, wie beim hybriden Entwicklungsansatz, ist eine direkte Kommunikation zwischen der JavaScript-Ebene und nativen Schnittstellen nicht möglich. Interpretierte Anwendungen benötigen daher ebenfalls eine Implementation von Bridging als Abstraktionsschicht. Dies erfolgt wie bei dem hybriden Ansatz über den Einsatz eines FFIs, welches als Middleware zwischen JavaScript und nativem Code fungiert (Biørn-Hansen u. a., 2018).

Besonders im Vergleich zum hybriden Entwicklungsansatz ergeben sich beim interpretierten Ansatz diverse Vorteile. So sind interpretierte Anwendungen zunächst nicht mehr abhängig von einer stabilen Internetverbindung, da der benötigte JavaScript-Code nun während der Entwicklung in die native Anwendung eingefügt wird. Durch die Verwendung von lokal ausgeführtem JavaScript stehen Entwicklern zudem eine Vielzahl von verfügbaren JavaScript-Bibliotheken zur Verfügung. Dazu kann der JavaScript-Code ohne Weiteres auf anderen Plattformen verwendet werden. Hierfür wird allerdings eine Middleware zur Interpretation und Verarbeitung benötigt. Die Überführung von JavaScript in native UI-Elemente erlaubt zudem eine native UX (El-Kassas u. a., 2017). Besonders im Vergleich zu nativen Anwendungen ist dennoch weiterhin eine im Verhältnis niedrigere Performance zu erwarten, da der benötigte Interpretationsprozess zur Laufzeit stattfindet (Biørn-Hansen u. a., 2018). Des Weiteren wird für Änderungen an der Anwendung ein Update benötigt, da alle benötigten Ressourcen in der Anwendung gebündelt sind und so keine dynamische Anpassung erlauben (El-Kassas u. a., 2017).

Der interpretierte Ansatz stellt einen Kompromiss aus der nativen und der hybriden Entwicklung dar. Zunächst werden, wie bei der nativen Entwicklung, alle benötigten Ressourcen direkt innerhalb der Anwendung mitgeliefert. Umgesetzt wird dies allerdings nicht mit einer nativen Sprache, sondern ebenfalls in JavaScript. Dazu erlaubt es die Verwendung von JavaScript Entwicklern auf eine Vielzahl von Bibliotheken

[1] *JavaScriptCore*: https://developer.apple.com/documentation/javascriptcore
[2] *V8*: https://android.googlesource.com/platform/external/v8/

zurückzugreifen, sowie Code auf verschiedenen Plattformen wiederzuverwenden. Für die Nutzung von JavaScript wird allerdings ein zusätzlicher Interpretationsprozess zur Laufzeit benötigt, welcher sich negativ auf die Performance der Anwendung auswirken kann. Zudem wird weiterhin das Bridging benötigt, um eine Kommunikation zwischen JavaScript und nativem Code zu ermöglichen.

### Kompilierter Ansatz

Ein weiterer mobiler Entwicklungsansatz ist der Compiled-Ansatz. Im Vergleich zu den vorherigen Ansätzen wird Programmcode hier nicht interpretiert, sondern mithilfe verschiedener Compiler in andere Sprachen kompiliert (Biørn-Hansen u. a., 2018). Für den Compiled-Ansatz existieren zwei Ausprägungen, welche im Folgenden als Unteransätze betrachtet werden (El-Kassas u. a., 2017).

Eine Ausprägung ist der Cross-Compiled-Ansatz. *Cross-Compiling* bezeichnet einen Prozess, bei welchem eine High-Level-Programmiersprache in nativen Byte Code oder Assembler übersetzt wird. Bei der Umsetzung einer mobilen Cross-Compiled-Anwendung wird die eigentliche Anwendungslogik daher zunächst in einer allgemeinen Programmiersprache, wie beispielsweise *C#*, implementiert und anschließend dann in nativen Byte Code, entsprechend der Zielplattform, kompiliert. Als Ergebnis entsteht eine native Anwendung, welche direkt auf den Endgeräten ausgeführt werden kann (El-Kassas u. a., 2017). Ein zusätzliches Bridging oder eine weitere Middleware wird hier nicht benötigt. Für die Umsetzung von Cross-Compiled-Anwendungen stehen Entwicklern verschiedene Frameworks wie beispielsweise *Xamarin*[3] zur Verfügung (Biørn-Hansen u. a., 2018). Diese erlauben es auch spezifische Geräte-APIs oder native Funktionen direkt anzusprechen. Das Framework übernimmt hierbei das Mapping von einer allgemeinen Sprache in Bytecode (Fayzullaev, 2018).

Die Vorteile des Cross-Compilers liegen besonders in der Wiederverwendbarkeit des Sourcecodes. Diese kann in Kombinationen mit anderen Cross-Compilern als Ausgangssituation genutzt werden, um weitere Plattformen anzusprechen. Da das Ergebnis von Cross-Compilation zudem immer eine native Anwendung ist, profitieren Cross-Compiled-Anwendungen von weiteren nativen Vorteilen wie einer bestmöglichen Performance oder einer nativen UI. Nachteilig wiederum ist die starke Abhängigkeit von den zur Verfügung stehenden Frameworks, da diese für beispielsweise das Mapping der Schnittstellen zuständig sind. Dieser Mapping-Prozess ist komplex, weshalb diverse Frameworks nur grundlegende Funktionen unterstützen. Der Funktionsumfang der finalen Anwendung kann so eingeschränkt sein, da er an den Funktionsumfang des Frameworks geknüpft ist. Zudem erfordert jede weitere zu unterstützende Plattform einen separaten Cross-Compiler (El-Kassas u. a., 2017).

Die zweite Ausprägung der Cross-Compilation ist der Trans-Compiler. Auch hier wird Anwendungslogik zunächst in einer allgemeinen High-Level-Sprache geschrieben, allerdings im Anschluss nicht in Byte Code oder Assembler, sondern in eine andere High-Level-Sprache kompiliert. So kann beispielsweise eine Android-Anwendung, welche mit der Programmiersprache Java umgesetzt ist, in moderneren Kotlin Code

[3] *Xamarin*: https://dotnet.microsoft.com/en-us/apps/xamarin

überführt werden. Dieser kann dann als Grundlage für eine neue native Anwendung dienen. So ist es aber beispielsweise auch möglich, nativen iOS-Code aus der Sprache Swift in die Sprache Kotlin der Android-Plattform zu überführen (El-Kassas u. a., 2017).

Ähnlich wie der Cross-Compiler profitiert der Trans-Compiler von der Möglichkeit, Sourcecode für weitere Kompiliervorgänge und so für andere Plattformen wiederzuverwenden. Das Ergebnis bleibt weiterhin eine native Anwendung und ermöglicht so eine bestmögliche Performance sowie eine native UI. Wie beim Cross-Compiler bilden Trans-Compiler aber meist nur einen schmaleren Funktionsumfang ab und unterstützen grundlegende Schnittstellen der Ausgangs- und Zielplattform. Dazu müssen Trans-Compiler regelmäßig angepasst werden, um entsprechende Änderungen in Programmiersprachen abdecken zu können (El-Kassas u. a., 2017).

Insgesamt stellt der Compiled-Ansatz eine gute Option dar, welche sich besonders für schlankere Anwendungen, welche nicht auf die Nutzung diverser APIs ausgelegt sind, anbietet. Beide Unteransätze erlauben es Sourcecode in eine andere High-Level-Sprache oder direkt in nativen Byte Code zu überführen und so native Anwendungen zu generieren, welche Vorteile mit sich bringen, wie eine bessere Performance und native UI. Dennoch decken diese Compiler meist nur einen grundlegenden Funktionsumfang ab und müssen regelmäßig an Änderungen in den Programmiersprachen angepasst werden.

#### Model-Driven Ansatz

Ein weiterer Entwicklungsansatz ist der Model-Driven-Ansatz, welcher auf dem Entwicklungsparadigma des Model-Driven-Developments basiert. Der Begriff *Model-Driven-Development* beschreibt eine Sammlung von Technologien, welche es ermöglicht, aus formalen Modellen, lauffähige Software abzuleiten. Konkret soll ein plattformunabhängiges Modell in plattformspezifischen Quellcode überführt werden (Biørn-Hansen u. a., 2018).

Das Vorgehen des Model-Driven Ansatzes besteht darin, UI und Anwendungslogik zunächst mithilfe von textuellen Modellen zu bestimmen (Biørn-Hansen u. a., 2018). Konkret werden hierfür drei Modelle genutzt. Zunächst ein plattformunabhängiges Modell zur Beschreibung der Anwendung. Dazu existiert ein plattformspezifisches Modell, welches zusätzlich beschreibt, wie ein System plattformspezifische Typen und Schnittstellen ansprechen kann. Zuletzt folgt das Transformationsmodell, welches für die Überführung von einer Plattform unabhängigen in ein plattformspezifisches Model genutzt wird (El-Kassas u. a., 2017). Zur Beschreibung dieser Modelle werden Modellierungssprachen wie *UML* oder eine Domain-Specific-Language[4] verwendet Biørn-Hansen u. a. (2018).

Diese Modelle werden anschließend von einem Code-Generator in den gewünschten Quellcode umgewandelt. Die Ausgangsmodelle können dabei entweder direkt in Quellcode für alle Zielplattformen überführt oder, ähnlich dem Cross-Compiled-Ansatz, von

[4] Eine anwendungsbezogene Sprache mit einem hohen Grad an Problemspezifität (DSL) (Góis Mateus u. Martinez, 2019)

einer Programmiersprache in eine andere kompiliert werden. Abschließend kann dann aus dem generierten Quellcode mithilfe der nativen Entwicklungsumgebungen (Android Studio oder Xcode) eine lauffähige Anwendung generiert werden. In diesem Schritt ist es für Entwickler zudem noch möglich, zusätzliche Anpassungen direkt im Quellcode vorzunehmen (El-Kassas u. a., 2017).

Die Verwendung des Model-Driven Ansatzes erlaubt es Entwicklern, sich überwiegend konzeptionell mit der Funktionalität und dem Inhalt der Anwendung auseinanderzusetzen. Dies ermöglicht es, besonders Entwicklern ohne technische Kenntnisse, eigene Anwendungen zu realisieren, da lediglich Vorwissen über die Modellierung und nicht zu den nativen Sprachen benötigt wird (Biørn-Hansen u. a., 2018). Insgesamt kann der Entwicklungsprozess durch die Trennung der formalen und technischen Ebene beschleunigt werden. Da das Endprodukt aber weithin eine native Anwendung ist, welche auf Basis der Modelle generiert wird, können Nutzer weiterhin von nativen Vorteilen profitieren (El-Kassas u. a., 2017).

Der besonders starke Fokus auf die Modellierung kann aber auch zu diversen Nachteilen führen. So muss die theoretische Modellierung der Anwendung ausreichend spezifisch formuliert sein, um sowohl alle Anforderungen zu erfüllen, als auch dynamische Aspekte der gewählten Domäne abdecken zu können. Der Fokus bei der Verwendung von Modellen besteht darin, reale Entitäten abzubilden. Die größtmögliche Abdeckung kontextspezifischer Aspekte erfordert allerdings ein hohes Maß an Abstraktion, was bei der Entwicklung zu Einschränkungen führen kann. Daher kann ein großer oder komplexere Projektumfang zu Problemen bei der Modellierung führen. Dazu wird für die Überführung in eine native Anwendung pro unterstützter Plattform ein entsprechender Code-Generator benötigt (El-Kassas u. a., 2017). Letztlich kann es noch zu benötigten Änderungen an dem generierten Quellcode kommen, um beispielsweise Funktionen, welche im Rahmen der Modellierung nicht abgebildet werden können, nachträglich zu implementieren. Dies setzt allerdings wieder ein grundlegendes Fachwissen über die nativen Programmiersprachen sowie gewählte Plattform voraus. Dazu müssen komplexe Fehler, also jene, welche nicht auf Probleme bei der Spezifikation der Modelle zurückzuführen sind, während der Laufzeit untersucht werden (Biørn-Hansen u. a., 2018).

Zusammenfassend ist festzuhalten, dass der Model-Driven Ansatz besonders für nicht technische Entwickler eine Möglichkeit bietet, eigene Anwendungen zu entwickeln, da diese vorrangig Kenntnisse über die Modellierung und nicht die technische Ebene benötigen. Generell kann der Ansatz zu einer Beschleunigung des Entwicklungsprozesses beitragen. Das Endprodukt ist auch hier weiterhin eine native Anwendung, weshalb die bereits genannten Vorteile ebenfalls zutreffen. Der Fokus auf die Modellierung kann aber auch zu Herausforderungen führen, besonders wenn eine komplexe Domäne abgebildet werden soll. Dazu kann weiterhin konkretes Fachwissen benötigt werden, sofern nachträgliche Änderungen am generierten Quellcode notwendig sind.

### Progressive Web App

Als letzter Entwicklungsansatz soll die Progressive Web App (PWA) vorgestellt werden. Der Begriff *Progressive Web App* wurde erstmals 2015 von Frances Berriman und

Alex Russel verwendet und beschreibt eine Webanwendung, welche aufgrund von modernen Browserfunktionen diverse Merkmal einer nativen Anwendung besitzt (Tandel u. Jamadar, 2018).

Damit eine Webanwendung als PWA lokal installiert werden kann, sind zwei Komponenten notwendig. Zunächst wird ein Manifest benötigt. Innerhalb eines Manifests werden, von dem mobilen Endgerät benötigte Informationen festgelegt. Dazu zählen beispielsweise der Name der Anwendung oder das Logo, welches auf dem Homescreen des Endgeräts angezeigt werden soll (Tandel u. Jamadar, 2018). Wird eine installierte PWA nun vom Homescreen aus gestartet, erfolgt die Darstellung der Anwendung innerhalb eines artefaktlosen Webbrowsers. Hierbei handelt es sich um einen Browser ohne Bedienungsmöglichkeiten, wie beispielsweise einer Suchleiste. Da alle benötigten Ressourcen bereits bei der Installation auf das Endgerät heruntergeladen wurden, kann die Anwendung nun regulär, wie auch im Webbrowser verwendet werden. Durch das Starten der Anwendung vom Homescreen und der artefaktlosen Darstellung entsteht für den Nutzer so der Eindruck einer nativen Anwendung Biørn-Hansen u. a. (2018). Bei der Nutzung einer Webanwendung kann nun allerdings der Fall eintreten, dass zusätzliche Daten über einen HTTP Request angefragt werden müssen. Hierbei kommt der Service Worker, die zweite wichtige Komponenten einer PWA zum Einsatz. Implementiert wird dieser in Form einer dedizierten JavaScript Datei, welche in die Anwendung integriert wird. Die Aufgabe des Service Workers liegt zunächst darin, den internen Anwendungslifecycle zu kontrollieren. Dazu ist er für die Umsetzung von Push-Benachrichtigungen und Datensynchronisation verantwortlich (Biørn-Hansen u. a., 2018). Die Datensynchronisation umfasst zunächst das Caching von ausgehenden Anfragen. Wird also im Rahmen der Webanwendung eine Anfrage an einen Server gestellt, um beispielsweise zusätzliche Ressourcen wie Bilder zu laden, wird die Antwort des Servers zunächst durch den Service Worker abgefangen und eine Kopie der angeforderten Ressourcen in den Cache gelegt. Sollte die Anwendung dann ohne eine aktive Internetverbindung genutzt werden, können ausgehende Anfragen durch den Service Worker auf den Cache umgeleitet werden. Die mit der Verwaltung des Caches verbundenen Aufgaben, wie das Aktualisieren von gecachten Ressourcen, wird automatisiert durchgeführt (Tandel u. Jamadar, 2018).

Für die Implementierung einer PWA stehen Entwicklern verschiedenen Optionen zur Verfügung. Zunächst können bekannte Web-Frameworks wie *Angular*[5], *Vue.js*[6] oder *React*[7] verwendet werden. Zusätzlich kann die Implementierung einer PWA nach dem Frameworkless-Ansatz erfolgen. Hier wird auf die Verwendung eines vollständigen Frameworks verzichtet und stattdessen nur einzelne dedizierte JavaScript Bibliotheken wie *StencilJS*[8] oder *Svelte*[9] verwendet. Dies erlaubt es den Overhead, welcher durch die oben genannten Frameworks entsteht, zu reduzieren (Huber u. a., 2021).

Zusätzlich ergeben sich weitere Vorteile bei der Nutzung einer PWA. Zunächst kann eine PWA einer größeren Zielgruppe zur Verfügung gestellt werden, da diese nicht nur

[5] *Angular*: https://angular.io/
[6] *Vue.js*: https://vuejs.org/
[7] *React*: https://reactjs.org/
[8] *StencilJS*: https://stenciljs.com/
[9] *Svelte*: https://github.com/sveltejs/svelte

Browser, sondern auch Geräte unabhängig, ohne die Nutzung eines Appstores, installiert und genutzt werden kann (Huber u. a., 2021). Technisch ermöglicht vorrangig die Nutzung eines Service Workers eine native Nutzung, da dieser automatisiert das Cachen und Aktualisieren von gespeicherten Ressourcen übernimmt. Da es sich bei einer PWA im Kern dennoch um eine reguläre Webanwendung handelt, welche über einen Webserver bereitgestellt wird, kann durch den Einsatz von *Search Engine Optimization* (SEO) eine erhöhte Reichweite erreicht werden. Dazu können auch bestehende JavaScript Bibliotheken eingesetzt werden. Optional kann das Design der Anwendung mit Webtechnologien wie HTML und CSS noch an die Zielplattform angepasst werden. Zusätzlich sind PWAs ausschließlich über HTTPS nutzbar und daher via SSH Verschlüsselung vor Manipulation gesichert (Tandel u. Jamadar, 2018).

Trotz diverser Vorteile ergeben sich aber auch Nachteile bei der Verwendung von PWAs. Zunächst besitzen PWAs nur eine geringe Sichtbarkeit, da sie im Falle der iOS-Plattform ausschließlich über einen Webbrowser gefunden und installiert werden können. Sie grenzen sich dadurch von nativen Anwendungen ab, welche direkt über einen AppStore bezogen werden können (Huber u. a., 2021). Dazu erfordert die Implementierung einen zusätzlichen Overhead im Quellcode der Anwendung. Dieser Overhead besteht aus dem Web-App-Manifest und dem Service Worker, welche beide zwangsweise benötigt werden. Auch wenn eine PWA einige Funktionen einer nativen Anwendung vorweisen kann, wird die Performance einer nativen Anwendung nicht erreicht. Dazu ist es, ähnlich zum hybriden Ansatz, nicht direkt möglich native Geräteschnittstellen, wie das File System oder GPS-Informationen, direkt anzusprechen. Zusätzlich kommt es bei der Verwendung von PWAs auf der iOS-Plattform zu zusätzlichen Einschränkungen, da es hier nicht möglich ist, den Zustand der Anwendung zwischen Sessions zu persistieren. Dazu wird im App Switcher, einer Übersicht über alle geöffneten Anwendungen inklusive Live-Vorschau, nur ein weißer Platzhalter angezeigt. Es besteht daher noch ein Unterschied in den verfügbaren Funktionen zwischen der iOS- und Android-Plattform (Biørn-Hansen u. a., 2018). Als weiteres allgemeines Problem ergibt sich der im Vergleich zur nativen Anwendung erhöhte Stromverbrauch. Dieser resultiert aus der internen Verwendung eines Browsers und der damit verbundenen Interpretation von JavaScript im Vergleich zu nativ ausgeführtem Bytecode. Abschließend erfordert ein vollständig, natives Styling einen erhöhten Aufwand (Huber u. a., 2021).

Eine PWA stellt eine Weiterentwicklung einer konventionellen Webanwendung dar, welche durch die Nutzung moderner Browserfunktionen, wie beispielsweise einem Service Worker, diverse Funktionen von nativen Anwendungen vorweisen kann. Nutzer profitieren von der Möglichkeit, Anwendungen auf den Homescreen zu installieren, wodurch diese zur erneuten Nutzung wie eine gewöhnliche Anwendung gestartet werden kann. Zusätzlich kann die Verwendung, von eigentlich servergestützen Webanwendungen nun ohne eine aktive Internetverbindung erfolgen. Im direkten Vergleich zu nativen Anwendungen weißt eine PWA aber weiterhin eine schlechtere Performance auf, was an zusätzlichem Overhead wie beispielsweise der Interpretation von Code und der Nutzung von Service Workern liegt. Zudem bestehen noch Unterschiede zwischen der iOS- und Android-Plattform, da noch nicht alle Funktionalitäten im gleichen Ausmaß von beiden Plattformen unterstützt werden.

#### Vor- und Nachteile der Cross-Plattform-Entwicklung

Auch wenn sich der Ansatz der Cross-Plattform-Entwicklung in diverse Unteransätze aufteilt, ergeben sich doch allgemeine Vor- und Nachteile des Ansatzes im Vergleich zur nativen Entwicklung, welche im Folgenden betrachtet werden sollen.

Im Wesentlichen kann der Cross-Plattform-Ansatz dazu beitragen, den allgemeinen Entwicklungsaufwand zu reduzieren. Ziel ist es, möglichst viele Plattformen auf Grundlage einer gemeinsamen Codebasis zu unterstützen. Dies resultiert in der Reduzierung der allgemein benötigten Entwicklungsressourcen und beschleunigt den Entwicklungsprozess. Dazu sinken auch die Anforderungen an Entwickler, da aufgrund einer einheitlichen Codebasis weniger Vorwissen benötigt wird (El-Kassas u. a., 2017).

Die Cross-Plattform-Entwicklung weist aber auch einige Nachteile auf. So erfordern viele der Cross-Plattform-Ansätze verschiedene Arten von Middleware. Sei es eine Interpretationsschicht für den Einsatz von Webtechnologien wie JavaScript und HTML oder zusätzliche Compiler zur Umsetzung des Cross-Compiled-Ansatzes. Die Implementierung solcher Middleware erfordert wiederum zusätzliches Fachwissen (Biørn-Hansen u. a., 2018). Dazu können Cross-Plattform-Anwendungen nicht die Performance einer nativen Anwendung erreichen. Im Allgemeinen stellen sie immer einen Kompromiss zwischen Effizienz in der Entwicklung und Funktionsumfang dar, weshalb es nicht immer möglich ist, alle komplexen Inhalte einer möglichen Domäne in einer Anwendung abzudecken oder die Anwendung dynamisch zu skalieren (El-Kassas u. a., 2017).

### 2.1.3. Backend-Driven UI

Nachdem sowohl der Native als auch der Cross-Plattform-Ansatz vorgestellt wurde, soll nun das Konzept von Backend-Driven UI betrachtet werden. Da es sich bei Backend-Driven UI um keinen etablierten Entwicklungsansatz handelt, sondern vielmehr um ein Konzept, welches in der Praxis unterschiedlich eingesetzt wird, muss zunächst eine allgemeine Begriffserklärung erfolgen (Elye, 2021). Anschließend folgt eine technische Betrachtung des Ansatzes sowie eine Auflistung aller benötigten Komponenten. Dazu werden Vor- und Nachteile betrachtet sowie verschiedene Grade zur Umsetzung vorgestellt. Abschließend soll Backend-Driven UI in die bestehenden Entwicklungsansätze eingeordnet werden.

Backend-Driven UI (auch Server-Driven UI genannt) beschreibt ein weiteres Konzept zur Entwicklung von mobilen Anwendungen (Maximo, 2020). Wie bereits aus dem Begriff abgeleitet werden kann, wird die UI einer Anwendung hier serverseitig gesteuert. Konkret orientiert sich der Ansatz an dem Konzept eines regulären Webbrowsers, welcher eine HTML-Datei von einem Server anfragt, diese dann lokal auf dem Client interpretiert und rendert. Backend-Driven UI beschreibt dieses Konzept nun für die Umsetzung von mobilen Anwendungen. Der Inhalt sowie Anwendungslogik der eigentlichen Anwendung werden ähnlich zum in Kapitel 2.1.2 vorgestellten hybriden Ansatz serverseitig festgelegt und dann durch den Client angefragt. Der Client ist eine native Anwendung, welche nun aber keine Web-View zum Rendern einer Website nutzt,

sondern die angefragte Ressource selbst interpretiert und diese dann in native UI mit zugehöriger Anwendungslogik überführt. Im Vergleich zum interpretierten Ansatz handelt es sich bei der angefragten Ressource nicht um JavaScript Code, sondern um eine deskriptive Beschreibung der Anwendung, welche beispielsweise durch Datenformate wie JSON oder XML realisiert wird. Im Vergleich zu einer nativen Anwendung, wie in Kapitel 2.1.1 beschrieben, sind Logik und UI-Beschreibung also nicht Teil der Anwendung, sondern werden, ähnlich zu konventionellen Webanwendungen, bei Bedarf von einem Server angefragt und dann zur Laufzeit interpretiert und dargestellt (Rucker, 2021). Der Server übernimmt hier die Verwaltung der Anwendung und bestimmt auch die Navigation innerhalb dieser. Die eigentliche native Anwendung ist lediglich ein Grundgerüst, welches Logik zum Verarbeiten der Serverantworten enthält (Maximo, 2020). Auch wenn das Endresultat einer Backend-Driven UI Anwendung weiterhin eine mobile Anwendung ist, welche auf einem mobilen Endgerät genutzt wird, ist die Umsetzung des Ansatzes von dem Zusammenspiel dreier Komponenten abhängig. Diese Komponente sind zunächst in Abbildung 2.1 abgebildet und werden im Folgenden genauer erläutert (Birch, 2020).

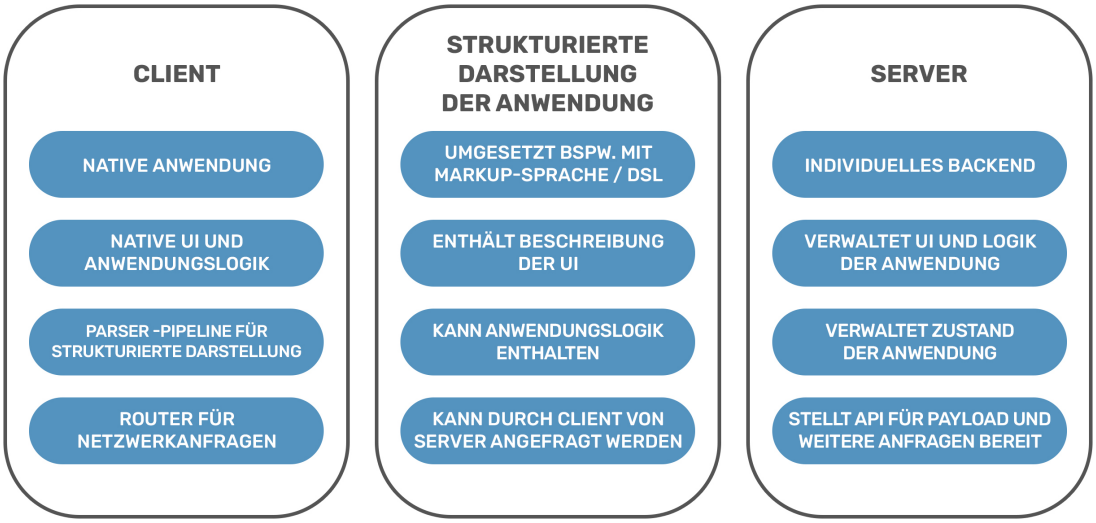



Abbildung 2.1.: Hauptkomponenten einer Backend-Driven UI Anwendung nach Birch (2020)

### Server

Die erste Komponente stellt den Server dar. Hierbei handelt es sich um ein Backend, welches dem Client in Form einer API, Endpunkte zum Anfragen von Ressourcen bereitstellt. Im Kontext von regulären nativen Anwendungen werden durch einen Webserver bereits dynamische Daten wie Bilder oder Texte bereitgestellt. Innerhalb einer Backend-Driven UI Anwendung stellt der Server hier aber zusätzlich die gesamte UI-Repräsentation sowie Anwendungslogik für die eigentliche Anwendung bereit (Maximo, 2020). Die Bereitstellung erfolgt über die strukturierte Darstellung, welche beispielsweise durch eine Markup-Sprache oder ein anderes Datenformat realisiert und über einen speziellen API-Endpunkt von dem Client angefragt werden kann (Rucker, 2021).

Zusätzlich kann der Server weitere Endpunkte bereitstellen, beispielsweise zur Realisierung eines Logins. Neben der UI wird noch der aktuelle Zustand der Anwendung serverseitig verwaltet. Zustand beschreibt hier den aktuellen Fortschritt im Anwendungsablauf sowie die damit verbundene Navigation innerhalb dieser und keine Datenhaltung, wie beispielsweise Nutzereingaben (Maximo, 2020). Die Implementierung eines Backend-Driven UI Servers ist nicht an vorgeschriebene Technologien gebunden, sodass Entwickler frei zwischen existierenden Implementierungsmöglichkeiten wählen können. Es wird aber ein Mindestmaß an Funktionalität vorausgesetzt, um beispielsweise API Endpunkte für den Client bereitstellen zu können (Asri, 2020).

#### Strukturierte Darstellung der Anwendung

Bei der zweiten Komponente handelt es sich um die strukturierte Darstellung der Anwendung, welche vonseiten des Servers an den Client gesendet wird. Die strukturierte Darstellung enthält zunächst eine Beschreibung der UI der eigentlichen Anwendung. Im Falle von nativen Anwendungen erfolgt die UI-Beschreibung direkt innerhalb des Programmcodes. Hier wird bestimmt, welche UI Elemente gerendert werden und wie diese aussehen sollen. Im Rahmen von Backend-Driven UI findet diese Beschreibung jetzt nicht mehr innerhalb der nativen Anwendung statt, sondern innerhalb des Servers. Die Darstellung kann dabei beispielsweise durch Datenformate wie JSON und XML oder auch mit einer individuellen DSL erfolgen und muss alle zum Rendern benötigten UI-Informationen beinhalten. Dazu zählen unter anderem das Layout, Farben, Fonts und die Anordnung der Elemente. Die UI muss allerdings nicht dynamisch durch den Server generiert werden, sondern kann vorab durch das Entwicklungsteam angelegt worden sein. Es ist dazu möglich serverseitig Automatisierungen oder dynamische Anpassungen an der UI-Darstellung vorzunehmen (Rucker, 2021). Neben der UI muss die strukturierte Darstellung der Anwendung allerdings auch Anwendungslogik enthalten, insofern dies durch die eigentliche Anwendung vorgegeben wird. Während Anwendungslogik bei nativen Anwendungen getrennt von der UI implementiert und dann mit dieser verbunden wird, muss diese hier nun zusammen mit UI innerhalb einer strukturierten Darstellung umgesetzt werden. Der Client kann dann einzelne Teile der Anwendung, in Form der strukturierten Darstellung, vom Server anfragen (Asri, 2020).

#### Client

Die dritte Komponente bei der Umsetzung einer Backend-Driven UI Anwendung stellt der Client dar. Dieser wird ähnlich wie bei diversen in Kapitel 2.1.2 beschriebenen Ansätzen in Form einer nativen Anwendung realisiert. Wie bereits im Rahmen der strukturierten Darstellung beschrieben, findet die Deklaration der UI sowie das Definieren von Anwendungslogik nun nicht mehr auf dem Client selbst statt, sondern die benötigten Ressourcen werden serverseitig angefragt. Hierfür benötigt der Client dennoch grundlegende Logik, um Serveranfragen umsetzen zu können (Rucker, 2021). Der wichtigste Bestandteil ist allerdings die Parser-Pipeline, welcher die angefragte strukturierte Darstellung interpretiert, in eine native UI und Anwendungslogik überführt. Die Implementation einer solchen Pipeline ist immer abhängig von der jeweiligen Platt-

form, da hier konkret auf die plattformspezifischen UI-Elemente eingegangen werden muss (Asri, 2020).

### Vorteile

Ähnlich zu den bereits in Kapitel 2.1.2 vorgestellten Ansätzen bietet der Einsatz von Backend-Driven UI, besonders im Vergleich zu nativen Anwendungen diverse Vorteile. Zunächst wird der Umfang der nativen Anwendung stark reduziert. Die Anwendung enthält nun nicht mehr die vollständigen Ressourcen wie UI-Informationen oder Anwendungslogik, sondern fragt diese dynamisch von einem Server an. Innerhalb der nativen Anwendung wird lediglich noch Logik zum Parsen und Interpretieren der Serverantwort benötigt. Dadurch lassen sich auch diverse Code Dopplungen zwischen der iOS- und Android-Version vermeiden Rucker (2021). Die meisten Vorteile ergeben sich allerdings erst, sobald es zu Änderungen in der Anwendung kommt. Diese können nun dynamisch, wie beispielsweise bei einer Webanwendung vorgenommen werden und benötigen kein komplettes Update der nativen Anwendung. Somit entfällt der Review-Prozess, welcher im Rahmen des Entwicklungszyklus in Kapitel 1.1 beschrieben wurde und je nach Umfang der Anwendung mehrere Tage bis Wochen in Anspruch nehmen kann. Neue Funktionalitäten können daher ohne externe Abhängigkeiten direkt an den Client ausgerollt werden. Die Betriebsfähigkeit der Anwendung muss so nicht mehr unterbrochen werden. Als einzige Ausnahme ergeben sich hier Änderungen an der Parser-Pipeline, welche weiterhin ein Update der Anwendung erfordern (Maximo, 2020). Im Allgemeinen können Entwickler, zunächst aufgrund einer geringen nativen Codebasis, von einer schnelleren Wartung profitieren. Dazu handelt es sich bei Backend-Driven UI um einen *Unified-Client Ansatz*. Ähnlich zu einer Webanwendung folgen alle Clients demselben Verhalten und reagieren daher sofort auf Veränderungen aus dem Backend. Das Backend dient hier als *Single Source of Truth*. Da UI und Anwendungslogik nun serverseitig bestimmt werden, ist es zudem möglich Änderungen oder spezifische Funktionen nur einer gezielten Nutzergruppe zur Verfügung zu stellen (Rucker, 2021). Dies ermöglicht es Entwicklern besonders genau auf Anforderungen seitens der Nutzer einzugehen, aber auch bereits während der Entwicklung, mithilfe von A/B-Testing, neue Funktionen zielgruppengenau zu testen. Zuletzt profitieren aber auch die Nutzer von dem Einsatz von Backend-Driven UI. Für die Nutzer selbst ist der Unterschied zu einer nativen Anwendung zunächst gar nicht ersichtlich, da die serverseitig generierte UI auf dem Client interpretiert und in Form von nativen UI-Elementen dargestellt wird. Die Anwendung nutzt daher eine vollständig native UI sowie native Navigations- und Interaktionsmöglichkeiten (Rucker, 2021). Dazu profitieren Nutzer von dem automatischen Ausrollen individueller Funktionen (Asri, 2020).

### Nachteile

Trotz diverser Vorteile bringt der Ansatz von Backend-Driven UI auch Nachteile mit sich. Zunächst ist der Entwicklungsaufwand einer mobilen Anwendung mit Backend-Driven UI deutlich erhöht. Auf der technischen Seite wird zunächst eine Serverinfrastruktur benötigt, welche als Endpunkt für den Client dient und diesen auch bestmöglich abwärtskompatibel unterstützen muss. Neben dem Server wird zudem die native An-

wendung inklusive des Parsers benötigt. Dazu muss für die Darstellung der Anwendung eine geeignete Struktur gefunden werden, welche mindestens die UI-Darstellung abdeckt und je nach Anwendung sämtliche benötigte Anwendungslogik darstellen kann. Bei besonders komplexen Anwendungen kann es zudem sein, dass diese durch bestehende Datenformate wie JSON und XML nicht mehr abgedeckt werden können und die Entwicklung einer individuellen DSL notwendig wird (Maximo, 2020). Für die eigentliche Gestaltung der Anwendung, spezieller die Form der strukturierten Darstellung, existieren keine robusten Tools, wodurch es schwer ist, Mockups zu generieren oder Änderungen an der UI zu visualisieren. Im Allgemeinen stellt der Designprozess eine Herausforderung dar, insofern keine dedizierte grafische Oberfläche existiert und stattdessen die Generierung der strukturierten Darstellung händisch durchgeführt werden muss (Asri, 2020). Da die serverseitig bestimmte UI auf dem Client in native UI-Elemente interpretiert wird, gelten zudem die Limitierungen einer nativen klassischen UI. Dazu zählen zunächst Einschränkungen bei der Nutzung von individuellen UI-Komponenten oder Animationen, wodurch es zu Kompromissen bei dem Endergebnis kommen kann (Maximo, 2020). Dazu erfordert das Wiederverwenden von bestehenden Backend-Driven UI Elementen eine gute Dokumentation (Kelly u. Silverman, 2019). Die beschleunigte Veröffentlichung von neuen Funktionen, ohne externen Review-Prozess, kann zudem zu einer Instabilität der Anwendung führen. Ein weiterer Nachteil bei der Verwendung der Anwendung ist die Netzwerkabhängigkeit, welche eine Voraussetzung für die Kommunikation zwischen Client und Server und damit für die Nutzung der Anwendung darstellt. Anfragen bei besonders komplexen Anwendungen können sehr umfangreich werden, was besonders bei der mobilen Datennutzung zu einem hohen Datenverbrauch aufseiten der Nutzer führen kann. Sollten Nutzer sich dazu an einem Ort mit schlechter oder sogar nicht vorhandener Netzabdeckung befinden, kann dies die Nutzung der Anwendung ebenfalls negativ beeinflussen, indem beispielsweise die Reaktion auf Nutzereingaben stark verzögert wird. Im schlimmsten Fall kann die Anwendung auch unbrauchbar werden(Rucker, 2021). Dem kann zwar durch den Einsatz von Caching entgegengewirkt werden, allerdings erfordert dies zusätzlichen Entwicklungsaufwand und kann, beispielsweise durch Cache-Validierung, zu zusätzlicher Latenz innerhalb der Anwendung führen. Ebenfalls sind Backend-Driven UI Anwendungen aufgrund ihrer starken Netzwerkabhängigkeit anfällig, für beispielsweise *Man in the Middle*-Attacken, durch welche schadhafte Code eingeschleust werden kann (Maximo, 2020).

### Grade von Backend-Driven UI

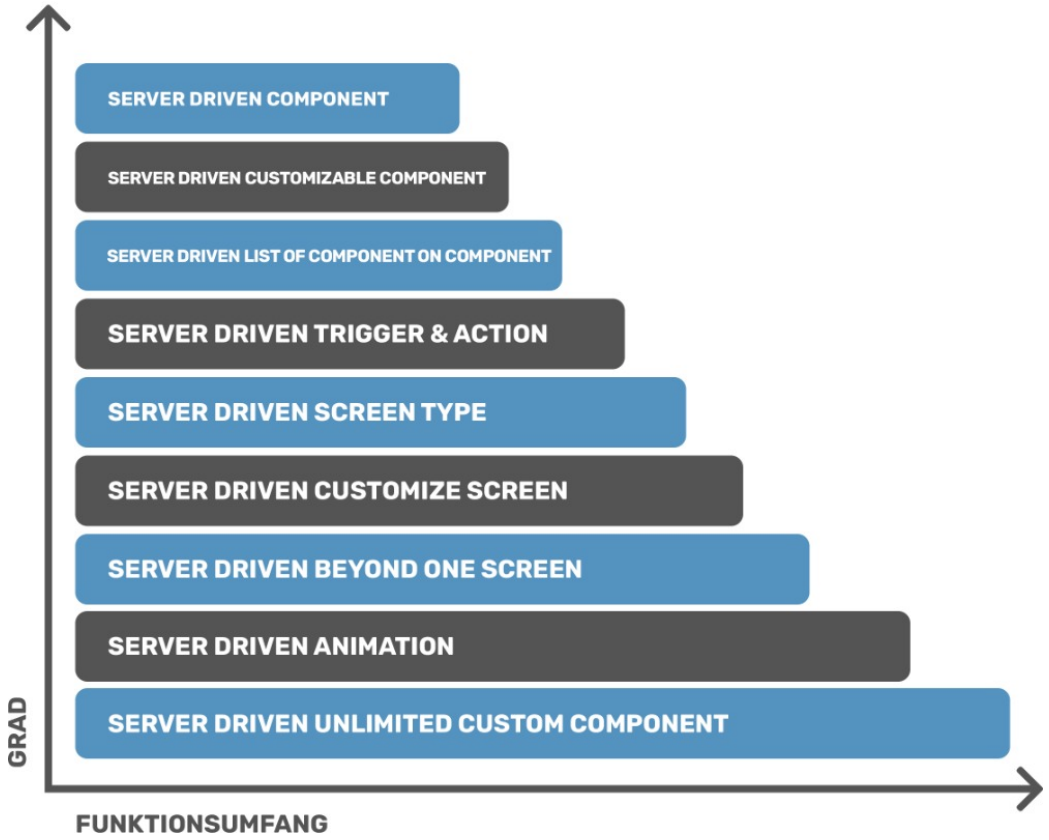



Abbildung 2.2.: Grade von Backend-Driven UI nach Elye (2021)

Nachdem nun das grundlegende Konzept, benötigte Komponenten sowie Vor- und Nachteile von Backend-Driven UI vorgestellt wurden, soll nun im Folgenden auf die Skalierbarkeit des Ansatzes eingegangen werden. Die Implementierung von Backend-Driven UI kann nach Elye (2021) zu verschiedenen Graden erfolgen, welche in Abbildung 2.2 dargestellt sind. Dies bedeutet, dass Anwendungen sowohl native UI und Backend-Driven UI parallel nutzen kann. Anhand dieses Verhältnisses lässt sich der Einsatz von Backend-Driven UI in verschiedene Grade unterteilen (Elye, 2021). Diese sollen nun im Folgenden vorgestellt werden.

### Server Driven Component

Server Driven Component beschreibt zunächst die Verwendung von Backend-Driven UI für einen einzelnen Screen innerhalb einer Anwendung. Ein Screen beschreibt hier ein konkret dargestelltes Anwendungsszenario. Die native Anwendung bietet verschiedenen UI Komponenten an, beispielsweise eine Liste oder einen Button. Der Server kann nun lediglich bestimmen, welche Komponenten angezeigt werden sollen. Für das Layout mehrerer Komponenten wird entweder vonseiten der App entweder ein horizontales oder vertikales Layout gewählt (Elye, 2021).

#### Server Driven Customizable Component

Server Driven Customizable Component baut auf der Server Driven Component auf, allerdings kann hier der Server noch zusätzlich das Zeichnen einer Komponente anpassen. Zunächst kann ein festgelegtes Set an Subkomponenten verwendet werden. Diese spiegeln die gewünschten Komponenten dar, allerdings in verschiedenen visuellen Formen. Der Server würde in diesem Fall keine konkreten Anpassungen an einer Komponente vornehmen, sondern lediglich zwischen verschiedenen Subkomponenten unterscheiden. Als nächste Stufe kann der Server selbst die Anordnung von Elementen festlegen und grundlegende Attribute wie Font, Farbe oder Abstände bestimmen. Das Festlegen der Anordnung ist hier noch beschränkt auf die Wahl einer vertikalen oder horizontalen Anordnung. Bei dem höchsten Grad der Server Driven Customizable Component kann der Server vollständig über die Anordnung von Elementen verfügen (Elye, 2021).

#### Server Driven List of Component on Component

Aufbauend auf den bisherigen Stufen ist es für den Server nun möglich Komponenten ineinander zu verschachteln. Dies erlaubt es beispielsweise Bilder innerhalb von Listenaufzählungen darzustellen (Elye, 2021).

#### Server Driven Trigger & Action

Bei den bisherigen Graden konnte der Server lediglich statische Elemente festlegen. Ein wichtiger Bestandteil einer mobilen Anwendung ist aber die Interaktion und das Reagieren auf Nutzereingaben. Bei Server Driven Trigger & Action kann der Server auch auf Nutzereingaben wie ein Klick oder Halten reagieren. Wichtig ist hierbei, dass diese Aktionen vorher definiert sein müssen (Elye, 2021).

#### Server Driven Screen Type

Bei Server Driven Screen Type setzt der Server nun nicht mehr einzelne Komponenten zu einem Screen zusammen, sondern kann auf ein Set von bereits vordefinierten Screen-Typen zurückgreifen (Elye, 2021).

#### Server Driven Customize Screen

Im Vergleich zu den vorherigen Stufen kann der Server nun bestehende Screen-Typen nutzen, diese allerdings während der Nutzung noch weiter anpassen. Dies erlaubt es, Komponenten zu bestehenden Screen-Typen hinzuzufügen (Elye, 2021).

#### Server Driven Beyond One Screen

Die bisher beschriebenen Grade für den Einsatz von Backend-Driven UI gehen ausschließlich von der Nutzung eines einzelnen Screens innerhalb der Anwendung aus. Mit Server Driven Beyond One Screen können serverseitig nun auch Multi-Screen-Anwendungen realisiert werden. Dies beinhaltet auch die Navigation innerhalb der Anwendung (Elye, 2021).

### Server Driven Animation

Server Driven Animation erlaubt es dem Server nun auch Animation zu verwenden, um beispielsweise animierte Übergänge bei der Navigation darzustellen (Elye, 2021).

### Server Driven Unlimited Custom Component

Server Driven Unlimited Custom Component beschreibt die Möglichkeit, serverseitig komplett individuelle Komponente mit eigener Logik zu realisieren. Ein mögliches Beispiel stellt eine Komponente dar, welche es ermöglicht einen Graphen zu zeichnen. Bei diesem Grad handelt es sich allerdings nur um ein theoretisches Konzept, da dies durch den Einsatz von Backend-Driven UI bisher nicht möglich ist. Entwickler müssen hierfür auf bestehende Ansätze wie den in Kapitel 2.1.2 beschriebenen hybriden Ansatz und die Verwendung einer Web-View zurückgreifen (Elye, 2021).

Insgesamt lässt sich der Ansatz von Backend-Driven UI zu den Ansätzen der Cross-Plattform-Entwicklung einordnen. Der Grundgedanke der Cross-Plattform-Entwicklung, mehrere Plattformen auf Basis einer einheitlichen Codebasis zu unterstützen, wird hier durch das Verwenden einer strukturierten serverseitigen Darstellung der Anwendung, welche für beliebige Plattformen wiederverwendet werden kann, erfüllt. Im Allgemeinen orientiert sich der Ansatz an dem in Kapitel 2.1.2 beschriebenen hybriden Ansatz, da auch hier plattformspezifische native Anwendungen zum Einsatz kommen, welche allerdings nicht die eigentliche Anwendungslogik, sondern nur eine Interpretationsschicht beinhalten. Im Vergleich zum hybriden Ansatz wird hier nicht eine Webanwendung in Form einer Web-View gerendert, sondern eine strukturierte Darstellung, welche Informationen zur UI und Anwendungslogik in einem einheitlichen Format enthält, interpretiert und in native Komponenten überführt.

### Zusammenfassung

Abschließend ist festzuhalten, dass der Ansatz von Backend-Driven UI diverse Vorteile von bestehenden Cross-Plattform-Ansätzen mit denen einer nativen Anwendung kombiniert. Entwickler profitieren nach einem initialen Setup besonders von einer einfachen Wartbarkeit und der Möglichkeit, neue Funktionalitäten dynamisch, ohne einen externen Review-Prozess, an Nutzer ausrollen zu können. Für die Nutzer der Anwendung ist kein Unterschied zu einer nativen Anwendung feststellbar, da aufseiten des Clients native UI-Elemente verwendet werden. Negativ ist dennoch die starke Netzwerkabhängigkeit, welche sich aus der Verwendung eines Servers als zentrale Komponente des Ansatzes ergibt. Dazu verlangsamt die zusätzliche Interpretation der strukturierten Darstellung den Anwendungsprozess. Besonders bei komplexeren Anwendungen kann es zu einem erhöhten Datenverbrauch für Nutzer kommen, daher sollte der Einsatz von Backend-Driven UI immer kontextabhängig bestimmt und die Komplexität der Anwendung vorher abgewägt werden.

## 2.2. Formate zur Datenrepräsentation

Im Rahmen moderner Anwendungen ist die stetige Kommunikation mit einem externen Server zum Standard geworden. Dies erlaubt es nicht nur dynamisch zusätzliche Daten anzufordern, sondern auch Teile der Anwendungslogik auszulagern, beispielsweise das Authentifizieren eines Benutzers (Rogers u. Gratch, 2022). Besonders bei der Umsetzung von Backend-Driven UI ist der Datenaustausch zwischen Client und Server, ein essenzieller Bestandteil der Anwendung und für die Übermittlung der strukturierten Darstellung wichtig (Rucker, 2021). Um einen solchen Austausch von Daten zu gewährleisten, müssen die auszutauschenden Informationen in einem strukturierten und einheitlichen Format vorliegen. Zusätzlich muss jede Seite individuell in der Lage sein, die strukturierten Daten mithilfe von Parsern zu verarbeiten (Asri, 2020).

Zur strukturierten Beschreibung von Daten haben sich die Formate XML und *JSON* in der Webkommunikation als Standard etabliert. Diese ermöglichen es, Eigenschaften, Zugehörigkeiten, Abhängigkeiten und Darstellungsformen abzubilden. Dazu erlauben sie so einen plattform- und sprachunabhängigen Austausch von Daten (Friesen, 2016).

Die Nutzung eines einheitlichen Datenformats bietet sich zur Abbildung der strukturierten Darstellung einer Backend-Driven UI Anwendung an (Rucker, 2021). In Bezug auf die folgende Konzeption eines Frameworks soll daher im Folgenden eine Betrachtung der vorgestellten Formate erfolgen. Dafür soll zunächst der strukturelle Aufbau, Vor- und Nachteile sowie bestehende Parser-Optionen für die Plattformen iOS und Android vorgestellt werden. Zusätzlich soll im Hinblick auf den Datenverbrauch innerhalb einer Backend-Driven UI Anwendung der Speicherbedarf der zwei Formate verglichen werden.

### XML

Bei XML (Extensible Markup Language) handelt es sich um eine offene und lizenzfreie Auszeichnungssprache zur strukturierten Darstellung von Daten in Textform (W3C, 2015). Diese Daten können plattformübergreifend, beispielsweise über das Web, zwischen Systemen ausgetauscht und weiterverarbeitet werden. Veröffentlicht wurde die erste Version von XML im Jahr 1998. Seit 2008 befindet sich XML in der aktuellsten Version 5 (Friesen, 2016). Im Folgenden soll nun genauer auf den syntaktischen Aufbau einer XML-Datei eingegangen werden. Hierfür wird die in Listing 1 gezeigte Darstellung einer XML-Struktur verwendet.

Eine XML-Datei besteht zunächst aus einer XML-Deklaration (siehe Listing 1, Zeile 1). Diese enthält die innerhalb der Datei genutzte XML Version und gibt an, welcher Encoding-Standard verwendet wird. Der meistgenutzte Standard ist hier UTF-8. Die XML-Deklaration sowie die enthaltenen Informationen sind zwingend notwendig, damit Systeme die Inhalte als XML-Format erkennen können. Im Anschluss an die Deklaration folgt der eigentliche Inhalt. Die Repräsentation von Daten wird durch die Verwendung von Tags erreicht und kann auf zwei unterschiedliche Schreibweisen erfolgen. Zunächst in Form eines öffnenden und schließendes Tags, wie beispielsweise in Zeile 4 - 7 dargestellt. Dazu können auch sogenannte Leertags, also Tags ohne die Angabe eines Wertes, verwendet werden, was in Zeile 40 dargestellt ist. Die Benennung

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <address>
        <city>Beispielstadt</city>
        <country>Deutschland</country>
        <postcode>12345</postcode>
        <street>Beispielstrasse 1a</street>
    </address>
    <name>Mustermann Immobilien Agentur</name>
    <portfolio>
        <realEstate>
            <referenceNumber>9da234sda02fdgd99</referenceNumber>
            <address>
                <city>Beispielstadt 2</city>
                <country>Deutschland</country>
                <postcode>54321</postcode>
                <street>Beispielstrasse 5a</street>
            </address>
            <furnishing>
                <element>Bett</element>
                <element>Kleiderschrank</element>
            </furnishing>
            <isBarrierFree>true</isBarrierFree>
            <potentialCustomer>
                <name>Familie Schmidt</name>
                <phone>01232-6318263</phone>
            </potentialCustomer>
        </realEstate>
        <realEstate
            referenceNumber="0023jskjdf203jse"
            isBarrierFree="false">
            <address
                street= "Beispielsstrasse 10b"
                city="Beispielstadt 3"
                country="Deutschland"
                postcode="15243"/>
            <furnishing>
                <element>Kueche</element>
            </furnishing>
            <potentialCustomer/>
        </realEstate>
    </portfolio>
</root>
```

Listing 1: Beispiel XML Datei

der Elemente kann frei gewählt werden und sollte die darzustellenden Daten widerspiegeln. Besitzt ein Element beispielsweise zusätzliche Eigenschaften, können diese entweder wie in den Zeilen 13 - 18 als verschachtelte Elemente oder wie in Zeile 32 - 36 als direkte Attribute eines Elements dargestellt werden. Die Verwendung von Attributen reduziert zwar den benötigten Code, führt allerdings auch zu Einschränkungen, da Attribute nicht weiter verschachtelt werden können und nur einen Wert pro Attribut besitzen. Werte können hier in Form von Zeichenketten, Ganzen- und Fließkommazahlen sowie eines Booleans dargestellt werden. Zusätzlich kann die Validierung der genutzten Datentypen durch die Verwendung eines XML-Schemas erfolgen. Hierbei handelt es sich um eine strukturierte Definition von erlaubten Datentypen für eine spezifische XML-Struktur. Ein zugehöriges XML-Schema zu der in Listing 1 dargestellten XML-Datei ist in Anhang 18 zu finden. Die Darstellung erfolgt anhand einer hierarchischen Baumstruktur. Den Ausgangspunkt einer XML-Datei stellt das Element *root* dar, welches dann weitere Elemente verschachtelt beinhaltet. Die Verschachtelung von Elementen kann zusätzlich auch über eine Listenstruktur abgebildet werden. Dies ist beispielsweise bei dem Element *portfolio* in Zeile 10 zu sehen, welches eine Liste von *realEstate* Objekten beinhalten kann (Friesen, 2016).

Zusammenfassend handelt es sich bei XML um einen Standard zur strukturellen Darstellung von Daten, welcher einen plattformübergreifenden Datenaustausch ermöglicht. Trotz einiger Einschränkungen bei der Nutzung von Attributen, erlaubt es XML komplexe Datenstrukturen abzubilden und ist daher auch für die Abbildung größerer Domänen geeignet. Dazu ist XML nicht nur maschinenlesbar, sondern kann aufgrund der hierarchisch aufgebauten Tag-Struktur gut von Menschen bearbeitet und gelesen werden. Zusätzlich können XML-Dateien mithilfe eines Schemas validiert werden. Als Nachteil ergibt sich allerdings die Redundanz bei der Nutzung eines jeweils öffnenden und schließenden Tags für ein einzelnes Element. Dazu handelt es sich bei XML um eine textuelle Darstellung, was bei der weiteren Verarbeitung eine zusätzliche Überführung voraussetzt. Hierbei kann zudem die Verwendung von Sonderzeichen zu Problemen führen.

### JSON

Bei JSON (JavaScript Object Notion) handelt es sich um ein 1997 veröffentlichtes Format zur textuellen Darstellung von Daten, welches ebenfalls einen plattformunabhängigen Datenaustausch zwischen Anwendungen ermöglicht (json.org). Im Folgenden soll wie bereits in Kapitel 2.2 zunächst auf die Struktur und im Anschluss auf die Vor- und Nachteile einer JSON Datei eingegangen werden. Zur Erläuterung wird hier die in dem folgenden Listing 2 dargestellte Struktur verwendet. Inhaltlich werden dieselben Daten wie in Listing 1 dargestellt.

Im Vergleich zu XML benötigt JSON keine Deklaration oder zusätzliche Annotationen zu Beginn einer Datei. Da der Aufbau auch hier anhand einer Baum-Struktur erfolgt, wird zunächst in der ersten Zeile das Stammelement eingeleitet. Wie auch XML besitzt jede JSON Datei ein einziges Stammelement. Elemente allgemein werden hier in Form von JavaScript Objekten dargestellt, welche beliebig ineinander verschachtelt

werden können. Einzelnen Elemente können beliebig viele Eigenschaften enthalten, welche selbst auch wieder neue JavaScript Objekte darstellen können. Bereits in Zeile 2 - 8 ist zu erkennen, dass das Stammelement zum einen eine Eigenschaft *name* vom Typ String, aber auch eine Eigenschaft *address* besitzt, welche selbst wieder ein weiteres Objekt ist. Wie bei XML stehen als Datentypen für die Wertzuweisung standardmäßig Zeichenketten, Ganz- und Fließkommazahlen sowie Booleans zur Verfügung. Zusätzlich kann aber auch *null* als leerer Wert verwendet werden. JSON erlaubt es ebenfalls Listen-Strukturen, in Form eines Arrays zu nutzen. Diese können entweder atomare Werte, siehe Zeile 18 - 21, oder sogar ganze JavaScript Objekte, siehe ab Zeile 9, beinhalten (json.org). Ähnlich zu XML kann auch hier ein JSON-Schema verwendet werden, um eine JSON Datei zu validieren. Ein zu Listing 2 zugehöriges JSON-Schema ist in Anhang 19 dargestellt. Zur zusätzlichen Optimierung ist es möglich, JSON in nativen Binärcode zu übersetzen und so komprimiert zwischen Geräten auszutauschen. Hierfür kann der Standard *BSON* genutzt werden (Friesen, 2016).

Insgesamt stellt JSON ein komprimiertes Datenformat zur strukturellen Darstellung von Daten dar und erlaubt es ähnlich wie XML diese Daten plattformübergreifend auszutauschen. JSON überzeugt besonders durch seine übersichtliche Struktur und ist durch die Verwendung von Eigenschaftsnamen sehr leserlich. Da Elemente bereits als JavaScript Objekte dargestellt werden, ist die Überführung von JSON zu nativem JavaScript sehr effizient.

```json
{
  "name": "Mustermann Immobilien Agentur",
  "address": {
    "street": "Beispielstrasse 1a",
    "city": "Beispielstadt",
    "postcode": "12345",
    "country": "Deutschland"
  },
  "portfolio": [
    {
      "referenceNumber": "9da234sda02fdgd99",
      "address": {
        "street": "Beispielstrasse 5a",
        "city": "Beispielstadt 2",
        "postcode": "54321",
        "country": "Deutschland"
      },
      "furnishing": [
        "Bett",
        "Kleiderschrank"
      ],
      "isBarrierFree": true,
      "potentialCustomer": {
        "name": "Familie Schmidt",
        "phone": "01232-6318263"
      }
    },
    {
      "referenceNumber": "0023jskjdf203jse",
      "address": {
        "street": "Beispielstrasse 10b",
        "city": "Beispielstadt 3",
        "postcode": "15243",
        "country": "Deutschland"
      },
      "furnishing": [
        "Kueche"
      ],
      "isBarrierFree": false,
      "potentialCustomer": null
    }
  ]
}
```

Listing 2: Beispiel JSON Datei

### Vergleich

Nachdem die beiden Formate nun grundlegend betrachtet wurden, soll im Folgenden verglichen werden, wie strukturierte Daten auf den Plattformen iOS und Android verarbeitet werden können. Hierfür sollen zum einen bereits existierende native Parser-Optionen und zum anderen bestehenden Drittanbieter-Bibliotheken verglichen werden. Eine Übersicht über die betrachteten Technologien zur Verarbeitung von JSON ist in Tabelle 2.1 und für XML in Tabelle 2.2 dargestellt.

| | **Native** | **Externe Bibliothek** |
|---|---|---|
| **Android** | JSONObject<br>JSONReader<br>GSON | Moshi<br>Jackson |
| **iOS** | JSONDecoder | SwiftyJSON<br>HandyJSON<br>JASON |

Tabelle 2.1.: Übersicht über Parser-Optionen für JSON

| | **Native** | **Externe Bibliothek** |
|---|---|---|
| **Android** | XmlPullParser<br>ExpatPullParser | Jackson-dataformat-xml<br>SimpleXML |
| **iOS** | XMLParser | SwiftyXMLParser<br>XMLCoder |

Tabelle 2.2.: Übersicht über Parser-Optionen für XML

Die iOS- und Android-Plattform bieten zunächst beide native Parser für JSON und XML an. Für JSON existiert unter Android zunächst der Datentyp *JSONObject*[10], eine native Repräsentation eines JSON Objektes beziehungsweise einer eingelesenen JSON Datei. Zum Generieren eines JSON-Objekts kann die interne Klasse *JSONReader*[11] genutzt werden, welche verschiedene Methoden zum Verarbeiten von JSON Daten anbietet. Weiterhin existiert die Google eigene Bibliothek GSON[12]. Vonseiten der Drittanbieter-Bibliotheken stehen Entwicklern die Bibliotheken *Moshi*[13] und *Jackson*[14] zur Verfügung. Unter iOS kann die native Funktion *JSONDecoder*[15] verwendet werden. Dazu existieren die externen Bibliotheken *SwiftyJSON*[16], *HandyJ-*

[10] *JSONObject*: https://developer.android.com/reference/org/json/JSONObject (Letzter Zugriff: 26.02.2023)
[11] *JSONReader*: https://developer.android.com/reference/android/util/JsonReader (Letzter Zugriff: 26.02.2023)
[12] *GSON*: https://github.com/google/gson (Letzter Zugriff: 26.02.2023)
[13] *Moshi*: https://github.com/square/moshi (Letzter Zugriff: 26.02.2023)
[14] *Jackson*: https://github.com/FasterXML/jackson (Letzter Zugriff: 26.02.2023)
[15] *JSONDecoder*: https://developer.apple.com/documentation/foundation/jsondecoder (Letzter Zugriff: 26.02.2023)
[16] *SwiftyJSON*: https://github.com/SwiftyJSON/SwiftyJSON (Letzter Zugriff: 26.02.2023)

*SON*[17] und *JASON*[18]. Für das Verarbeiten von XML stehen Entwicklern unter Android nativ die Klassen *XmlPullParser*[19] und *ExpatPullParser*[20] zur Verfügung. Als Drittanbieter-Bibliotheken existieren *Jackson-dataformat-xml*[21], welche eine Erweiterung des JSON-Parsers Jackson ist, und *SimpleXML*[22]. Zu SimpleXML ist anzumerken, dass es sich hierbei ursprünglich um eine Java-Bibliothek handelt, welche allerdings aufgrund der Interoperabilität mit der Programmiersprache Kotlin auch für aktuelle Android-Projekte verwendet werden kann. Die Dokumentation von SimpleXML ist allerdings ausschließlich für Java ausgerichtet. Unter iOS steht Entwicklern nativ der *XMLParser*[23] zur Verfügung. Zu den verfügbaren Drittanbieter-Bibliotheken zählt *SwiftyXMLParser*[24] und *XMLCoder*[25].

Abschließend ist festzuhalten, dass es sich bei XML und JSON um Formate zur strukturellen Beschreibung von Daten handelt. Beide unterstützen das Abbilden von einzelnen sowie verschachtelten Elementen, erlauben das Nutzen von Listen-Strukturen und bieten gängige Datentypen wie Zeichenketten, Ganz- und Fließkommazahlen und Booleans an. JSON bietet zusätzlich die Option, leere Werte durch *null* darzustellen. Strukturell unterscheiden sich die beiden Formate dennoch in ihrer Syntax. XML nutzt eine Tag-Schreibweise, während JSON eine JavaScript-Objekt-Schreibweise zur Darstellung von Elementen nutzt. Beide Formate sind vollständig maschinen- und menschenlesbar, auch wenn JSON durch seine JavaScript-Syntax weniger Zeichen benötigt.

## 2.3. Mobile UI

Die UI ist eine der wichtigsten Komponenten einer Anwendung. Sie bestimmt nicht nur, welche Informationen dargestellt werden, sondern auch, welche Interaktionsmöglichkeiten dem Nutzer bei der Verwendung einer Anwendung zur Verfügung stehen (Steinberger, 2021). Dies gilt ebenfalls für mobile Anwendungen und auch für den Einsatz von Backend-Driven UI (Rucker, 2021). Zum Abschluss der Grundlagen soll daher betrachtet werden, wie die Umsetzung von UI im Rahmen von mobilen Anwendungen erfolgen kann. Hierfür erfolgt zunächst ein Einblick in den Paradigmenwechsel von imperativer zu deklarativer UI. Im Anschluss werden die aktuellen deklarativen UI-Frameworks *SwiftUI* und *Jetpack Compose* vorgestellt und auf die UI-Guidelines der iOS- und Android-Plattform eingegangen werden.

[17] *HandyJSON*: https://github.com/alibaba/HandyJSON (Letzter Zugriff: 26.02.2023)
[18] *JASON*: https://github.com/delba/JASON (Letzter Zugriff: 26.02.2023)
[19] *XmlPullParser*: https://developer.android.com/reference/org/xmlpull/v1/XmlPullParser (Letzter Zugriff: 26.02.2023)
[20] *ExpatPullParser*: https://developer.android.com/reference/android/util/Xml#newPullParser() (Letzter Zugriff: 26.02.2023)
[21] *Jackson-dataformat-xml*: https://github.com/FasterXML/jackson-dataformat-xml (Letzter Zugriff: 26.02.2023)
[22] *SimpleXML*: https://github.com/ngallagher/simplexml (Letzter Zugriff: 26.02.2023)
[23] *XMLParser*: https://developer.apple.com/documentation/foundation/xmlparser (Letzter Zugriff: 26.02.2023)
[24] *SwiftyXMLParser*: https://github.com/yahoojapan/SwiftyXMLParser (Letzter Zugriff: 26.02.2023)
[25] *XMLCoder*: https://github.com/CoreOffice/XMLCoder (Letzter Zugriff: 26.02.2023)


Abbildung 2.3.: Storyboard innerhalb der Xcode IDE

Bei der imperativen UI Entwicklung handelt es sich um eins der verbreitetsten UI-Paradigmen im Bereich der mobilen Entwicklung. Die Basis bildet das imperative Programmierparadigma, welches besagt, dass Programme nach einer festen Abfolge von Anweisungen ablaufen. Überträgt man dieses Prinzip auf die Entwicklung von UI, liegt der Fokus bei der Entwicklung einer Anwendung auf dem Aussehen der UI und nicht auf den darzustellenden Datenstrukturen (Varma u. Varma, 2019). Im Rahmen der Android-Entwicklung bedeutet dies, dass die Auswahl und Gestaltung von UI-Komponenten zunächst in Form von XML-Markup erfolgt, während die eigentliche Zuweisung von Daten in der Anwendungslogik stattfindet (Marchenko, 2023). Dies gilt ebenfalls für den Einsatz imperativer UI auf der iOS-Plattform. Die Gestaltung der UI wird hier via Storyboards durchgeführt, dargestellt in Abbildung 2.3, während die eigentliche Datenzuweisung innerhalb des Programmcodes stattfindet (Lewis u. a., 2012). Hauptmerkmal der imperativen UI im mobilen Bereich ist das objektorientierte Vorgehen. UI-Komponenten werden bei ihrer Verwendung im Programmcode wie Objekte behandelt, welche zugewiesene Eigenschaften und Methoden besitzen können. Zusätzlich wird das Konzept von Vererbung angewendet. Wird beispielsweise ein Ob-

jekt einer UI-Komponente initialisiert, so erbt dieses von einer generischen Oberklasse, welche für jede UI-Komponente existiert (Marchenko, 2023).

Die Verwendung von imperativer UI bietet diverse Vor- und Nachteile. Da es sich hier um ein bereits bestehendes und durch Best Practices etabliertes Konzept handelt, können Entwickler zunächst von einer sehr ausgiebigen Dokumentation profitieren. Dies gilt besonders für die herstellerspezifischen Dokumentationen zur iOS- und Android-Plattform. Dazu existiert ein großes Angebot an Drittanbieter Bibliotheken, welche beispielsweise das Verwenden von externen UI-Komponenten ermöglichen. Dazu wird die Verwendung von imperativer UI durch die Plattformen iOS und Android von einer Vielzahl an Geräten und Betriebssystemen unterstützt (Muema, 2021). Nachteilig ist allerdings, dass vonseiten der Entwickler ein höheres Vorwissen gefordert wird. So werden beispielsweise bei der Android-Entwicklung nicht nur Kenntnisse zu den Programmiersprachen Java und Kotlin benötigt, sondern auch in dem Umgang mit XML Marchenko (2023). Selbiges gilt für die iOS-Plattform und den Umgang mit Storyboards. Dazu erfordert die allgemeine Verwaltung von UI hier zusätzlichen Code. Die Umsetzung von imperativer UI setzt ebenfalls zusätzliche manuelle Schritte, wie beispielsweise grundlegendes Layouting oder die Verarbeitung von Nutzereingaben (Muema, 2021). Imperative UI ist zudem anfällig für Code Redundanzen. Beispielsweise bei der Android-Entwicklung kann Layouting nicht nur über XML vorgenommen werden, sondern auch innerhalb der Anwendungslogik gesetzt werden. Die Anwendungslogik und XML sind hierbei nicht synchronisiert, wodurch es zu Inkonsistenzen und Fehlern kommen kann (Marchenko, 2023). Selbiges gilt für die Verwendung von Storyboards unter iOS. Dazu müssen Änderungen an der UI immer anhand des UI-Zyklus erfolgen, da es sonst zu Seiteneffekten kommen kann (Muema, 2021). Die folgende Abbildung 2.4 zeigt eine beispielhafte Umsetzung von UI in Android. Hier wird zunächst das grundlegende Layout sowie die Auswahl an UI-Komponenten innerhalb der XML-Struktur vorgenommen. Zusätzliche wird dieses Layout dann innerhalb der Anwendungslogik zusätzlich verändert.

```xml
<TextView
    android:id="@+id/t1"
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:text="Lorem Ipsum!"/>
```

(a) Layout in von XML

```
setContentView(R.layout.activity_main)

val t:TextView = findViewById(R.id.t1)
t.height = 200
```

(b) Zusätzliches Layouting in Anwendungslogik

Abbildung 2.4.: UI in Android mit XML

Im Jahr 2013 führte dann die Veröffentlichung des von Facebook entwickelten Frameworks *React Native* zu einem Wandel von imperativer hin zu deklarativer UI. Ziel war es, die Vorteile aus der UI-Gestaltung im Web, welche bereits deklarativ erfolgte, auf die mobilen Plattformen zu bringen. Neben Vorteilen wie einer kürzeren Entwicklungszeit und weniger Code, sollten Entwickler von Funktionen wie *Hot-Reload* profitieren. 2017 veröffentlichte dann Google mit *Flutter* ihr erstes deklaratives Framework. Trotz diverser Vorteile blieb die Adaption von deklarativer UI allerdings begrenzt. Diese änderte sich erst im Jahr 2019, als Apple und Google auf ihren jährlichen Entwicklerkonfe-

renzen, der WWDC und Google I/O, ihre eigenen plattformspezifischen deklarativen Frameworks, *SwiftUI* und *Jetpack Compose* vorstellten. Als Anmerkung ist hier zu nennen, dass es sich bei Flutter zwar um ein Google eigenes Projekt handelt, dies aber nicht Teil des Android-Entwicklungsprojekts ist (Steinberger, 2021).

Deklarative UI löst sich nun im Vergleich zur imperativen UI von der Beschreibung, wie etwas aussehen soll. Viel mehr geht es nun darum, zu beschreiben, was gezeigt werden soll (Varma u. Varma, 2019). Konkret liegt der Fokus in der Entwicklung nun auf der Datenbasis, welche durch die UI dargestellt wird. Die UI einer Anwendung muss hier nicht mehr wie ein vordefiniertes Raster manuell mit Informationen befüllt werden. Stattdessen stellt die UI lediglich eine aktuelle Repräsentation der Datenbasis dar und kann daher auch selbstständig auf Veränderungen reagieren. Dies wird durch immer modernere und schnellere CPUs ermöglicht (Steinberger, 2021). Im Vergleich wird UI jetzt auch nicht mehr objektorientiert, sondern funktional behandelt. Bei der deklarativen UI werden Komponenten wie eine zustandshaltende Funktion behandelt. Dieser Zustand ist dynamisch und kann sich beispielsweise aufgrund von Nutzereingaben oder Anwendungslogik ändern. Dies resultiert dann in einer Aktualisierung der UI, welche durch das jeweilige deklarative UI-Framework automatisch umgesetzt wird. Sollen nun bestehende UI Elemente zusätzlich modifiziert werden, um beispielsweise die Hintergrundfarbe oder die Schriftgröße zu ändern, können Modifikatoren eingesetzt werden. Insgesamt erlaubt die Nutzung von Funktionen eine schnelle Wiederverwendbarkeit von Komponenten und ermöglicht es zusätzlich noch mehrere Elemente zu einem neuen Element zu gruppieren (Marchenko, 2023). Die Implementierung von UI ist nun auch zentralisiert und findet innerhalb der Anwendungslogik statt (Muema, 2021). Das folgende Listing 3 zeigt eine beispielhafte Umsetzung von deklarativer UI, anhand einer Text-Komponente, durch die Verwendung von Jetpack Compose im Android-Kontext.

```
1 @Composable
2 fun ExampleView() {
3     Text(text = "Lorem Ipsum")
4 }
```

Listing 3: UI in Android mit Jetpack Compose

Bei der Verwendung von deklarativer UI bieten sich diverse Vor- und Nachteile. Zunächst einmal wird die Menge an benötigtem Code stark, aufgrund des Wegfalls von Boilerplate, reduziert. Hierdurch sinkt die generelle Entwicklungszeit, das Fehlerpotential und der allgemeine Wartungsaufwand. Dazu benötigen Entwickler nun nur noch eine Sprache zur Umsetzung einer Anwendung, da nun UI und Anwendungslogik auf die plattformspezifische Sprache zurückgreift. Während bei der imperativen Entwicklung die UI noch an mehreren Stellen manipuliert wurde, bei Android beispielsweise im Rahmen Anwendungslogik und in gesonderten XML Dateien, existiert bei der deklarativen UI nun eine Single Source of Truth (Marchenko, 2023). Dazu übernehmen deklarative Frameworks wie SwiftUI und Jetpack Compose automatisch die Synchronisation und das Aktualisieren von UI-Komponenten. Sollte sich die Datenbasis durch Nutzereingaben oder Anwendungslogik ändern, wird die UI automatisch aktualisiert,

ohne dass diese Änderungen durch zusätzliche Anwendungslogik durchgeführt werden müssen. Ein weiterer Vorteil ergibt sich aus der Betrachtung von UI-Komponenten in Form von Funktionen. Durch die Möglichkeit UI-Funktionen ineinander zu verschachteln, können neue UI-Komponenten gebaut werden, welche dynamisch wiederverwendet werden können. Dazu kann deklarative UI innerhalb eines Projektes interoperabel mit imperativen UI-Strukturen verwendet werden. Zusätzlich profitieren Entwickler bei der Entwicklung von optionalen Tools wie einer automatisierten Aktualisierung der Testumgebung oder einer Live-Vorschau der aktuell implementierten UI (Soininen, 2021). Dennoch ergeben sich auch einige Nachteile bei der Verwendung von deklarativer UI und den zugehörigen Frameworks. Im Vergleich zu imperativer UI und imperativen Frameworks handelt es sich hier noch um eine neue Technologie im Kontext mobiler Anwendungsentwicklung. Entsprechend ist bei der Entwicklung von einer schlechteren Dokumentation sowie einem geringeren Angebot von Drittanbieter-Bibliotheken auszugehen. Der allgemeine Funktionsumfang ist zudem geringer, sodass Entwicklern nicht alle Funktionalitäten und UI-Komponenten der imperativen UI zur Verfügung stehen (Muema, 2021). Dazu setzt die Verwendung von deklarativer UI aufseiten der Nutzer ein aktuelles Betriebssystem voraus. Das Framework SwiftUI benötigt mindestens Version iOS 14.0, während die Nutzung von Jetpack Compose mindestens ein API-Level von 21 erfordert. Zum Zeitpunkt dieser Arbeit befindet sich iOS in der Version 16.2 und Android auf API-Level 33 (Steinberger, 2021).

#### SwiftUI

Bei SwiftUI handelt es sich um ein deklaratives GUI-Framework, welches von Apple erstmalig auf der WWDC 2019 vorgestellt wurde. SwiftUI löst damit das imperative UI-Framework namens *UIKit* ab und führt als erstes Apple eigenes Framework deklarative UI sowie die Nutzung der Model-View-Viewmodel Architektur (MVVM) ein. Die Funktionsweise von SwiftUI basiert auf den folgenden vier Prinzipien (Varma u. Varma, 2019).

- **Deklarativ**: SwiftUI folgt dem deklarativen UI-Paradigma. Dahingehend liegt bei Fokus bei der UI-Umsetzung auf den Daten, welche repräsentiert werden sollen. Dies erlaubt es UI-Elemente automatisch zu aktualisieren, sobald sich die zugrunde liegende Datenbasis ändert (Varma u. Varma, 2019).
- **Automatisch**: SwiftUI bietet Entwicklern eine Vielzahl von automatisierten Funktionalitäten an. Dazu zählen die Automatisierung des UI-Stacks sowie weitere Funktionen, wie automatischer Wechsel einem Dark- und Lightmode. Dazu erlaubt SwiftUI beispielsweise eine automatische Lokalisierung der Anwendung (Varma u. Varma, 2019).
- **Komposition**: Komposition in SwiftUI bezeichnet das dynamische Verschachteln von UI-Komponenten zu neuen Komponenten. Einzelne UI-Elemente werden in SwiftUI als Funktionen realisiert und können daher, dem Paradigma der funktionalen Programmierung folgend, zusammengesetzt werden (Varma u. Varma, 2019).

- **Konsistenz**: Ein Problem der imperativen UI-Implementierung besteht darin, dass Änderungen an mehreren Stellen vorgenommen werden können, ohne dass diese synchronisiert werden. Diese Asynchronität kann zu Verzögerungen oder Fehlern führen. In SwiftUI besitzen UI-Komponenten keinen direkten Zustand mehr und stellen lediglich eine Repräsentation der Datenbasis dar. Dies erlaubt das automatische Aktualisieren der UI im Falle von Änderungen an der Datenbasis. Optional existieren dennoch Annotationen, mit denen UI-Komponenten temporär mit Zustand versehen werden können (Varma u. Varma, 2019).

### Jetpack Compose

Bei Jetpack Compose handelt es sich um ein von Google entwickeltes, modernes UI-Toolkit zum Erstellen von nativer Android UI, welches erstmalig im Jahr 2019 auf der Google I/O vorgestellt wurde (Steinberger, 2021) und ebenfalls auf der MVVM-Architektur basiert. Jetpack Compose ist das erste offizielle deklarative UI-Framework zur Android-Entwicklung, mit dem Ziel, die UI-Entwicklung zu vereinfachen und den Entwicklungsprozess so zu optimieren. Wie auch bei SwiftUI, kann die Funktionsweise von Jetpack Compose anhand der folgenden Prinzipien zusammengefasst werden (Asefa, 2022).

- **Komposition**: Wie bereits im Vorherigen beschrieben, führt die Einführung von Jetpack Compose zu einem Paradigmenwechsel innerhalb der Android-Entwicklung. Statt wie bei der imperativen UI-Entwicklung, Komponenten wie Objekte zu behandeln, welche von übergeordneten Objekten erben, nutzt Jetpack Compose *Composable*-Funktionen zur programmtechnischen Darstellung von UI-Komponenten auf Grundlage von Datenabhängigkeiten. Hierbei handelt es sich zunächst um reguläre Funktionen, welcher allerdings nur innerhalb eines UI-Kontextes verwendet werden können. Zur Darstellung eines Screens mit mehreren Elementen können diese Funktionen, gemäß der funktionalen Programmierung, ineinander verschachtelt werden. Durch die Verschachtelung von Funktionen lassen sich zudem neue UI-Komponenten abbilden, welche beispielsweise auch an anderer Stelle wiederverwendbar sind (Asefa, 2022).

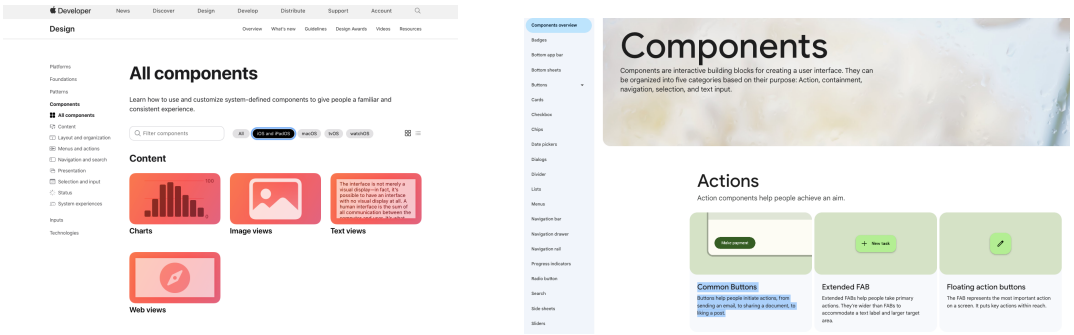
- **Kapselung**: Bei der ursprünglichen imperativen Android-Methodik wurden UI-Komponenten mit Zustand versehen. Sobald sich dieser verändert, musste dies mithilfe von *Callbacks* an die Anwendungslogik zurückgegeben werden. Deklarative UI unter Jetpack Compose wird unabhängig von der Anwendungslogik betrachtet. UI stellt hier eine direkte Repräsentation der Datenbasis dar und verwaltet daher keinen eigenen Zustand. Vielmehr ist die UI eine direkte Repräsentation des aktuellen Zustands der Anwendung. Dazu wird nun ein unidirektionaler Datenfluss angewendet, wodurch Veränderungen der Daten ausschließlich von der Anwendungslogik an die UI weitergegeben werden (Asefa, 2022).

- **Rekomposition**: Jetpack Compose verwendet Rekomposition, um als Reaktion auf Nutzereingaben oder Anwendungslogik, einzelne Elemente oder sogar die gesamte UI neu zu zeichnen. Dies ist ohne Rücksicht auf den UI-Zyklus möglich und wird automatisch durch das Framework durchgeführt (Asefa, 2022).

### Design Guidelines

Nachdem nun erläutert wurde, welche technischen Möglichkeiten es zu der Umsetzung einer UI gibt, muss die Strukturierung der eigentlichen Inhalte betrachtet werden. Die vorgestellten Frameworks bieten bereits eine Vielzahl von nativen UI-Elementen an, welche zur Darstellung der anwendungsbezogenen Daten genutzt werden können. Damit sich Anwendungen allerdings trotz kreativer Freiheiten an spezifische plattformabhängige Konventionen halten, existieren vonseiten der Plattformbetreiber definierte Design-Guidelines zur Umsetzung von mobilen Anwendungen. Diese sollen Entwickler bei der Umsetzung von UI unterstützen und dafür sorgen, dass eine Anwendung sich möglichst nativ zur gewählten Plattform nutzen lässt (Ryan, 2021).

Die Guidelines umfassen jeweils ein plattformspezifisches Set an verfügbaren UI-Ressourcen und bieten eine Sammlung an Dokumentation zum richtigen Umgang mit Plattform Konventionen an. Beide beziehen sich dabei auch nicht ausschließlich auf mobile Endgeräte, sondern decken beispielsweise auch die Bereiche Smartwatches und Desktop-Geräte ab. Insgesamt nutzen beide Plattformen eine unterschiedliche Design-Sprache, setzten dabei aber auf allgemeingültige Konventionen, wie beispielsweise die Größe von bestimmten UI-Elementen (Ryan, 2021). Eine Übersicht über die Human-Interface-Guidelines und die Material-Design-Guidelines wird in der folgenden Abbildung 2.5 dargestellt.




(a) Human-Interface-Guidelines (b) Material-Design-Guidelines

Abbildung 2.5.: Übersicht über Design Guidelines [26]

### Fazit

Eine nutzerzentrierte UI ist mittlerweile ein wichtiges Kriterium für den Erfolg einer Anwendung. Daher gab es besonders im Bereich der UI neue Entwicklungen und Technologien, die sich in den vergangenen Jahren ergeben haben. Der größte Durchbruch war der Umschwung von imperativer zu deklarativer UI-Entwicklung. In Kombination mit den vorgestellten Frameworks SwiftUI und Jetpack Compose, sowie der stärkeren Verbreitung der MVVM-Architektur, ist es nun möglich, UI und Datenbasis strikt zu

[26] Human-Interface-Guidelines: `https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/`
Material-Design-Guidelines: `https://m3.material.io/components`

trennen. Während bei der imperativen UI Komponenten meist selbst noch eigenen Zustand verwaltet haben, was zu mehreren Überschneidungen mit der eigentlichen Anwendungslogik führte, ist es nun möglich UI als reine Repräsentation der Datenbasis umzusetzen. Dies führt nicht nur zur Vereinfachung und Reduzierung von Code, sondern ermöglicht es auch UI automatisiert zu aktualisieren, wenn sich beispielsweise die Datenbasis ändert. Dennoch handelt es sich hierbei immer noch um eine sehr aktuelle und junge Technologie. Dies ist besonders im Bereich der Dokumentation, des Toolings und der Unterstützung von Drittanbieter Bibliotheken merkbar. Zusätzlich haben diese Technologien erhöhte Systemanforderungen.

# 3. Verwandte Arbeiten

Im Anschluss and die Grundlagen sollen nun verwandte Arbeiten vorgestellt werden. Im Rahmen der durchgeführten Recherche stellte sich heraus, dass das Konzept von Backend-Driven UI in der Fachliteratur nicht verbreitet ist. Individuelle Erfahrungswerte oder theoretische Überlegungen wurde hier überwiegend über Blogs und Artikel publiziert. Diese Einschätzung konnte zudem durch Biørn-Hansen u. a. (2018) bestätigt werden. Demnach findet die Veröffentlichung von Erkenntnissen im Bereich der Cross-Plattform-Entwicklung nahezu ausschließlich über Artikel, Blogs oder Vorträge statt. Grund hierfür ist die Aktualität und die damit verbundene Schnelllebigkeit des Themas, sodass Erkenntnisse schnell obsolet sein können und eine klassische Publikation zum Zeitpunkt der Veröffentlichung bereits veraltet sein kann (Biørn-Hansen u. a., 2018). Besonders bei dem Bereich von Backend-Driven UI handelt es sich um einen neuen Entwicklungsansatz, weshalb Erfahrungsberichte hier nahezu ausschließlich aus der Wirtschaft kommen (Maximo, 2020). Da besonders diese Erfahrungen im Umgang mit Backend-Driven UI bei der Konzeption eines Frameworks relevant sein können, soll im Folgenden ein Einblick in existierende Umsetzungen gegeben werden. Betrachtet werden hierfür das Framework *Lona*[1] von Airbnb sowie die Backend-Driven UI Umsetzung von SiriusXM[2].

## 3.1. Lona

Bei Lona handelt es sich um ein Server-Driven UI System von Airbnb. Airbnb ist ein US-amerikanisches Unternehmen und betreibt eine gleichnamige Website zur Buchung und Vermietung von Unterkünften (Kelly u. Silverman, 2019). Als Informationsquelle zu Lona wurde zunächst das öffentlich einsehbare Github Repository[3] herangezogen. Dieses ist explizit als Entwicklervorschau deklariert und erhebt keinen Anspruch auf eine vollständige Nutzbarkeit. Inhaltlich bietet das Repository eine kurze Beschreibung des Lona Projekts sowie Informationen zu den drei Hauptkomponenten *Lona Component*, *Lona Compiler* und *Lona Studio*. Der letzte Commit innerhalb des Repositorys wurde 01.10.2020 getätigt. Neben dem Repository wurde das Projekt Lona und die bei der Implementierung aufgetretenen Herausforderungen in einem Vortrag von Laura Kelly und Nathanael Silverman auf der KotlinConf2019 vorgestellt. Im Folgenden sollen nun die Entstehung, die Architektur sowie Vor- und Nachteile von Lona betrachtet werden (Kelly u. Silverman, 2019).

Airbnb unterstützt durch die Verwendung von React insgesamt drei Plattformen Android, iOS und Desktop. Zur Darstellung der UI nutzen alle Plattformen das Lona

[1] *Lona*: https://github.com/Lona/Lona
[2] *SiriusXM*: https://www.siriusxm.com/
[3] Lona (Developer Preview): https://github.com/Lona/Lona

System, welches auf einer Airbnb eigenen, komponentenbasierten Design Bibliothek aufbaut. Im Vergleich zu einem konventionellen deklarativen UI-Framework wie Jetpack Compose oder Swift, welche einzelne UI-Komponenten für sich betrachten, werden Elemente hier in festen Reihen zusammen gefasst. Dies kann beispielsweise eine *TitleRow* sein, welche sich dann aus mehreren Text-Komponenten zusammensetzt. Reihen werden daher als vorgefertigte Komposition von einzelnen UI-Komponenten betrachtet, welche je nach Anwendungsszenario von Entwicklern genutzt und dynamisch zusammengesetzt werden können. Eine Einteilung eines Screens in unterschiedliche Reihen ist Abbildung 3.1 dargestellt. Für die Kommunikation zwischen Client und Server wird das JSON Format verwendet. Gemäß dem Ansatz von Backend-Driven UI werden innerhalb der JSON Struktur alle für die UI benötigten Daten zusammengefasst. Im Falle von Lona sind dies zunächst alle UI-Komponenten eines Screens, inklusive der benötigten Eigenschaften wie Name, Typ oder ID (Kelly u. Silverman, 2019).

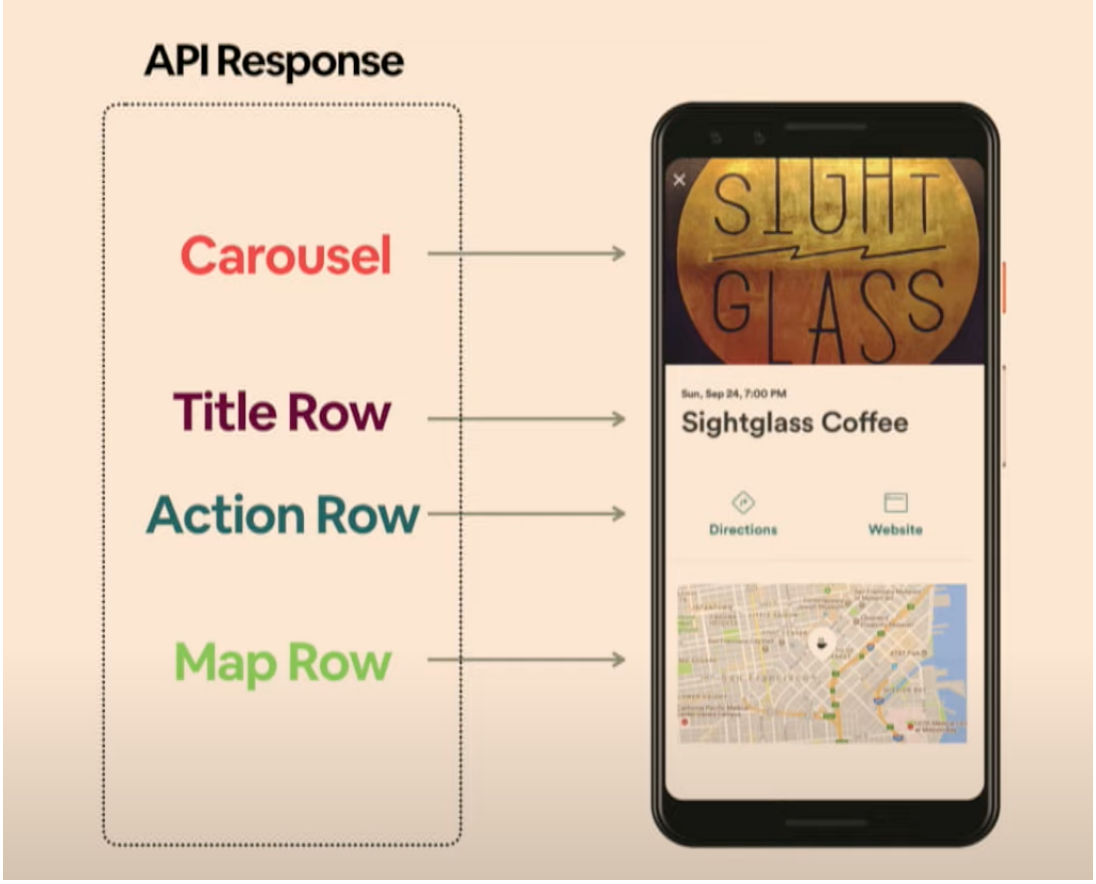



Abbildung 3.1.: Einteilung eines Screens in Reihen innerhalb des Lona-Systems Kelly u. Silverman (2019)

Nach Kelly und Nathanael sind bei der Verwendung einige Probleme aufgetreten. Zunächst einmal hat es sich als schwierig erwiesen, verschiedene Versionen von Clients zu unterstützen. Dies kann besonders kritisch sein, wenn der Server eine Komponente nutzen möchte, welche auf dem Client nicht zur Verfügung steht, weil dieser eine

ältere Version hat. Dies kann dazu führen, dass Elemente nicht angezeigt werden oder im schlimmsten Fall die Anwendung abstürzt. Neben der Versionierung ergibt sich die Synchronisation von Clients als Problem. Gemeint ist hiermit die Bereitstellung gleicher Inhalte für unterschiedliche Plattformen, insbesondere Unterschiede zwischen mobilen Anwendungen und dem Desktop Client. Da Desktops über größere Bildschirme verfügen, muss dieser Platz auch optimal genutzt werden, ohne dabei Unterschiede in den Funktionalitäten zu verursachen. Weitere Probleme ergaben sich bei der Skalierung der Anwendung. Da viele einzelne Entwicklerteams an der Anwendung arbeiten und dementsprechend individuelle Komponenten benötigen, erreichte das System laut Kelly und Nathanael einen unwartbaren Punkt, wodurch sich ebenfalls Probleme bei der Dokumentation ergaben. Eine händische Dokumentation ist mit viel Aufwand verbunden, da kontinuierlich Komponenten entfernt, neu hinzugefügt oder angepasst werden (Kelly u. Silverman, 2019) können.

Zur Lösung dieser Probleme hat Airbnb zusätzlich eine dedizierte Lona Spezifikation eingeführt. Hierbei handelt es sich um eine zusätzliche Abstraktionsebene, bestehend aus einer selbst definierten Kotlin DSL, welche als Abstraktionsschicht über der JSON Struktur des Design-Systems dient. Entwickler nutzen also zunächst die Lona Spezifikation zur Festlegung der UI. Zur Übermittlung an die Clients wird diese Spezifikation dann automatisiert in JSON überführt. Die Verwendung einer Kotlin DSL bietet laut Kelly und Nathanael diverse Vorteile. Zunächst lassen sich innerhalb der DSL verschiedene Versionsstände einer UI abbilden. Bei Anfrage eines Clients, kann die Lona Spezifikation mit der gewünschten Version dann in eine gezielte JSON Version umgewandelt und an den Client geschickt werden. Es werden also keine JSON Strukturen für alle existierenden Versionen benötigt, da diese nun automatisiert aus der Lona Spezifikation abgeleitet werden können. Da es sich bei Kotlin um eine streng typisierte Sprache handelt, ist es zudem möglich anhand der Kotlin DSL automatisch ein JSON Schema abzuleiten, welches es erlaubt, JSON Strukturen zu validieren und zu testen. Dazu kann auch eine entsprechende Dokumentation der DSL automatisch generiert werden. Zudem erlaubt die Lona Spezifikation das Verpacken von Komponenten in gesonderte Module, welche dynamisch an den Client gesendet werden können und so die Skalierbarkeit der Anwendung erhöhen (Kelly u. Silverman, 2019).

Interessierten Entwicklern ist es möglich, sich einen ersten Eindruck von Lona zu verschaffen. Hierfür steht, wie bereits angesprochen, ein spezielles Github Repository[4] zur Verfügung, welches explizit als Entwicklervorschau deklariert ist. Das Repository umfasst neben mehreren Readme-Dateien mit Erklärungen und Installationsanleitungen die drei testbaren Hauptkomponenten des Lona Systems. Auf diese Komponenten soll im Folgenden kurz eingegangen werden.

### Lona Studio

Bei Lona Studio handelt es sich um eine grafische Oberfläche zur Generierung von UI in Form von *.component*-Dateien. Entwicklern ist es möglich, Mockups schnell auf Basis bestehender Komponenten zu entwickeln, bestehende APIs zur Nutzung von Beispieldaten zu nutzen oder verschiedene Bildschirmgrößen zu simulieren. Dazu bietet Lona

[4] https://github.com/Lona/Lona

Studio die Möglichkeit, verschiedene Aufgaben, wie beispielsweise die Lokalisierung zu automatisieren. Die folgende Abbildung 3.2 zeigt die Oberfläche von Lona Studio, primär die Layer-Hierarchie sowie die Live-Vorschau von aktuell zusammengesetzten UI-Komponenten (AirBnB, 2017).

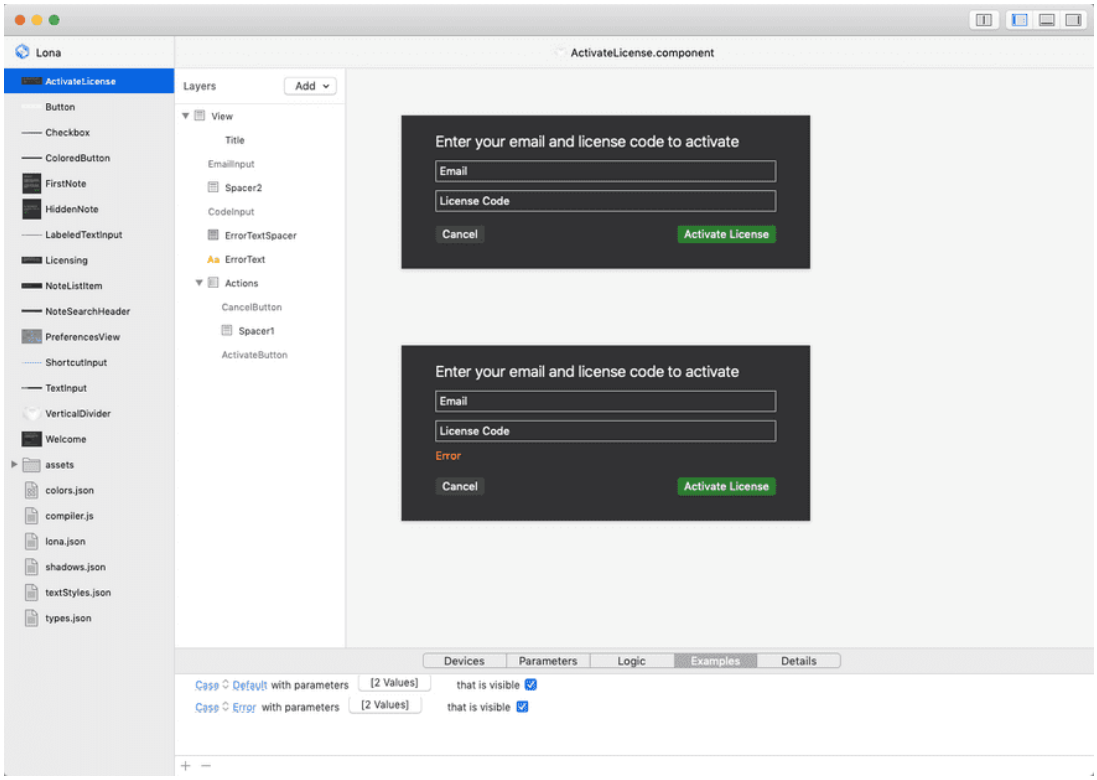



Abbildung 3.2.: Lona Studio (AirBnB, 2017)

Im Rahme der Entwicklervorschau wird bereits eine erste Installationsanleitung gegeben. Zusätzlich wird vonseiten der Entwickler angemerkt, dass das Tool zum aktuellen Zeitpunkt noch instabil ist und abstürzen kann. Angesichts dessen wird Lona Studio noch nicht offiziell bereitgestellt. Bei Interesse kann eine erste Version dennoch installiert werden (AirBnB, 2017).

### Lona Components

Lona Components umfasst das Design System des Lona Frameworks. Hierbei handelt es sich hier um eine Datenstruktur, welche zur Umsetzung der UI genutzt und anschließend durch den Lona Compiler in nativen UI-Code überführt wird. Innerhalb der Datenstruktur werden zunächst konkrete UI-Komponenten, aber auch visuelle Anpassungen durch beispielsweise Farben und Datentypen festgelegt. Die Umsetzung erfolgt durch das JSON-Format, welches laut den Entwicklern allerdings einige Schwierigkeiten aufzeigt. Genannt wird, dass JSON ineffizient zusammengeführt und durch Menschen bearbeitet werden kann. Die bereits vorgestellte Lona-Spezifikation, welche als

zusätzliche Abstraktionsschicht zur JSON-Struktur eingeführt wurde, ist nicht in der Entwicklervorschau enthalten (AirBnB, 2017).

Es werden dennoch Informationen zur genaueren Beschreibung von Komponenten innerhalb des JSON-Formats gegeben. Komponenten stellen hier konkrete Teile der UI dar und werden in Form von *.component*-Dateien definiert. Eine solche Datei umfasst zunächst Metadaten, wie beispielsweise eine Beschreibung oder zugehörige Tags. Zusätzlich werden Informationen zum Umgang mit verschiedenen Endgeräten festgelegt. Dazugehören unter anderem Größenangaben sowie Informationen, welche Endgeräte unterstützt werden. Eine *.component*-Datei enthält zusätzlich noch Test-Implementierungen für die jeweilige Komponente sowie Parameter, welche zur Datenzuweisung innerhalb der UI genutzt werden. Zuletzt besitzt jede Komponente eine *Root-Layer*. Ein *Layer* im Kontext von Lona beschreibt eine konkrete UI-Komponente, welche ebenfalls ineinander verschachtelt werden können und so eine vollständige UI ergeben. Entwickler können hier auf bereits vordefinierte Layer zurückgreifen oder individuelle Konfigurationen anlegen. Die nachfolgende Tabelle 3.1 soll den Aufbau eines Layers darstellen. Zuletzt umfasst eine *.component*-Datei Informationen zur Anwendungslogik. Dies ist allerdings noch nicht in der Entwicklervorschau abgebildet (AirBnB, 2017).

| **Eigenschaft** | **Typ** | **Vorausgesetzt** | **Beschreibung** |
|---|---|---|---|
| *id* | String | Ja | Einzigartige ID, zur Verwendung innerhalb der Anwendungslogik |
| *name* | String | Ja | Name des Layers |
| *type* | String | Ja | Typangabe für die weitere Verarbeitung |
| *params* | JSON | Ja | Benötigte Eingabeparameter zur Umsetzung der UI-Komponente |
| *children* | [Component] | Nein | Weitere verschachtelte Layer |

Tabelle 3.1.: Aufbau eines Layers innerhalb des Lona-Frameworks nach AirBnB (2017)

### Lona Compiler

Bei dem Lona Compiler handelt es sich um die letzte Hauptkomponente des Frameworks. Der Compiler wird genutzt um *.component*-Dateien in nativen UI-Code zu überführen. Aktuell werden die Plattformen iOS und macOS auf Basis der Programmiersprache Swift sowie React DOM, React Native und React SketchApp auf Basis von JavaScript unterstützt. In Zukunft soll zudem noch die Android-Plattform mit der Nutzung von Kotlin folgen. Anzumerken ist hier, dass diese Informationen der Entwicklervorschau entnommen wurden, welche zuletzt im Oktober 2020 aktualisiert wurde. Ob die Android-Plattform seid der Einführung der Lona Spezifikation unterstützt wird und falls ja in welchem Ausmaß, kann hier nicht beantwortet werden AirBnB (2017).

Insgesamt stellt Lona eine Umsetzung eines Backend-Driven UI Systems dar. Deutlich wird jedoch, dass das System stark auf die internen Abläufe und Softwareanforderungen des eigenen Unternehmens angepasst ist. Im Rahmen der Airbnb Anwendung ist eine Einteilung der UI in Reihen eine valide Umsetzung, allerdings ist die Anwendung auch dahingehend ausgelegt. Für die Nutzung externer Anwendungen ist

Lona daher nur eingeschränkt zu empfehlen, da keine freie UI-Gestaltung möglich ist. Dennoch stellt die Umsetzung von Lona einen berechtigten Ansatz zur Nutzung von Backend-Driven UI dar, welcher als Vorlage für die weitere Konzeption und Implementierung genutzt werden kann. Hierzu ist besonders die Entwicklervorschau zu nennen, welche nicht nur einen Einblick in die Codebasis bietet, sondern auch noch zusätzliche Erklärungen beinhaltet.

## 3.2. SiriusXM

Bei SiriusXM handelt es sich um ein Audio Entertainment Unternehmen aus Nordamerika, mit über 150 Millionen Nutzern. Die Marke SiriusXM umfasst dabei drei Untermarken. Zunächst die gleichnamige Marke *SiriusXM*, mit starkem Fokus auf musikalische Liveübertragungen, *Stitcher Radio*, eine Plattform zum Hören und Bereitstellen von Podcasts und *Pando*, eine Plattform zur Generierung individueller Musik-Empfehlungen. Alle drei Marken besitzen individuelle und umfangreiche Technologie-Stacks. Da dies einen starken Wartungsaufwand erfordert, liegt das Ziel von SiriusXM darin, diese Stacks zu vereinheitlichen. Auch wenn bereits einzelne Teile von SiriusXM Backend-Driven UI nutzen, gilt dies noch nicht für alle Plattformen. Dementsprechend soll ein einheitliches und systemübergreifendes Backend-Driven UI System implementiert werden. Im Folgenden soll dieser Optimierungsprozess genauer betrachtet werden. Als Ausgangslage wird der Vortrag von Thomas Wold und Stuart Horner im Rahmen der droidCon am 29.09.2022 genutzt, welcher sich mit dem Entwicklungsprozess und den damit verbundenen Entscheidungen eines einheitlichen Backend-Driven UI Systems befasst (Wolf u. Horner, 2022).

Die Einführung eines einheitlichen Backend-Driven UI Systems soll zunächst zur Vereinheitlichung der bestehenden Tech-Stacks dienen, um so Redundanzen zu verhindern und Legacy Systeme zu modernisieren. Dafür soll ein stark typisiertes Design-System implementiert werden, dessen Architektur auch die einzelnen Marken von SiriusXM unterstützt. Kunden sollen von stark dynamischen und typisierten Nutzererfahrungen und einer schnellen Marktreife profitieren. Dazu sollen durch die Vermeidung von Redundanzen zusätzliche Entwicklerressourcen geschaffen werden (Wolf u. Horner, 2022).

Im ersten Schritt der Umsetzung wurde zunächst eine individuelle Markup Sprache entwickelt. Diese war zwar sehr flexibel in ihrer Verwendung, allerdings auch schwer zu testen und zu warten. Da es sich hier um eine intern entwickelte Markupsprache handelte und daher auf keine bestehende Dokumentation zurückgegriffen werden konnte, führte die Einführung zu längeren Entwicklungsprozessen. Dazu wurde die Markupsprache ausschließlich für die Android-Plattform entwickelt und nur dort verwendet, wodurch ein architektonischer Unterschied zur iOS-Version entstand (Wolf u. Horner, 2022).

Aufgrund dieser Probleme wurde die Markup-Sprache verworfen und ein komplett dynamisches Backend-Driven UI Framework entwickelt. Als Datenformat wird JSON, mit einfachen Key-Value-Zuordnungen verwendet. Für die UI werden vordefinierte Komponenten verwendet, welche bereits auf dem Client hinterlegt sind. Innerhalb des JSON Formats können dann Hierarchien aus Containern und einzelnen Items,

wie einem Button oder Textfeld, implementiert werden. Jedes dieser Bestandteile ist jeweils wieder eine eigene Komponente und kann daher auch verschachtelt werden. Das Framework nutzt dazu eine Model-View-ViewModel Architektur und ist modular aufgebaut, was ein einfaches Austauschen von Komponenten erlaubt. Das Framework setzt sich aus mehreren Elementen zusammen. Zunächst existiert ein *AppCore* mit den Grundfunktionen. Dazu existieren optionale Bibliotheken für beispielsweise Netzwerkzugriffe, Navigation, besondere Styling Optionen und Werbung (Wolf u. Horner, 2022).

Insgesamt hat die Einführung des Frameworks dazu beigetragen, den Entwicklungsaufwand bei SiriusXM zu reduzieren und die genutzten Systeme zu vereinheitlichen. Dennoch sind laut Wolf und Stuart einige Hürden aufgetreten. Dazu zählen zunächst große visuelle Änderungen, welche beispielsweise ein komplett neues Set an UI-Komponenten benötigen. Des Weiteren können innerhalb der UI nur bereits vordefinierte Funktionalitäten integriert werden.

# 4. Konzeption

Nachdem im Rahmen der Grundlagen das Konzept von Backend-Driven UI sowie die dazugehörigen Komponenten vorgestellt wurden, soll im Folgenden die Konzeption eines Backend-Driven UI Frameworks erfolgen. Hierfür erfolgt zunächst eine Anforderungsermittlung. Anschließend soll auf dieser Grundlage eine Architektur bestimmt und benötigte Komponenten definiert werden. Diese Architektur soll dann als Ausgangspunkt für die weitere Umsetzung dienen.

## 4.1. Anforderungsermittlung

Bevor die eigentliche Konzeption des Frameworks beginnen kann, müssen Anforderungen definiert werden. Diese sollen dann im weiteren Entwicklungsprozess als Leitfaden dienen. Für die Beschreibung der Anforderungen werden die Schablonen für detaillierte funktionale und qualitative Anforderungen nach Rupp u. a. (2009) verwendet. Im Allgemeinen erfolgt zunächst die Betrachtung allgemeingültiger funktionaler Anforderungen. Im Anschluss erfolgt eine Aufteilung in die in Kapitel 2.1.3 beschriebenen drei Hauptkomponenten. Abschließend erfolgt die Aufstellung allgemeingültiger qualitativer Anforderungen.

Als Grundlage für die Anforderungsermittlung dient zunächst der allgemeine Ansatz von Backend-Driven UI. Dazu lassen sich aus den beschriebenen Vorteilen des Ansatzes erste Anforderungen ermitteln. Ergänzend sollen die verwandten Arbeiten mit Fokus auf das Lona-Framework betrachtet werden.

### 4.1.1. Allgemeine Anforderungen

Zu Beginn sollen generelle Anforderungen abgeleitet werden, welche sich auf das gesamte Framework beziehen. Hierfür wurden zunächst die in Kapitel 1.3 aufgestellten Forschungsfragen betrachtet und in übergeordnete Leitanforderungen überführt. Zusätzlich wurde die Aufteilung in die drei Hauptkomponenten *Client*, *Strukturierte Darstellung* und *Server*, anhand des in Kapitel 2.1.3 vorgestellten Backend-Driven UI Ansatzes, übernommen. Auf Grundlage des Lona-Frameworks (Kapitel 3.1) wurde zudem noch eine optionale grafische Oberfläche für die effizientere Erstellung von UI mit aufgenommen. Ebenfalls liegt der Fokus im Allgemeinen auf der plattformübergreifenden Nutzung, wodurch auch der variierende Funktionsumfang von verschiedenen Plattformen bedacht werden muss. Demnach ergebeben sich die folgenden allgemeinen Anforderungen:

- Das Framework muss Entwicklern die Möglichkeit bieten, eine Anwendung mit Backend-Driven UI umzusetzen.

- Das Framework muss fähig sein, plattformübergreifend genutzt zu werden.
- Das Framework muss fähig sein, unterschiedliche Funktionsumfänge von Plattformen auszugleichen.
- Das Framework soll fähig sein, Alternativen für Funktionen mit geringerem Funktionsumfang anzubieten.
- Das Framework muss erweiterbar sein.
- Das Framework muss fähig sein, die Hauptkomponenten *Client*, *Strukturierte Darstellung* und *Server* abzudecken.
- Das Framework soll fähig sein, eine grafische Oberfläche für die effizientere Erstellung von UI bereitzustellen.

### 4.1.2. Anforderungen an die strukturierte Darstellung

Im Anschluss an die allgemeinen Anforderungen soll konkret der Bereich der strukturierten Darstellung betrachtet werden. Dieser stellt den Kern des Backend-Driven UI Ansatzes dar und hat sowohl client- als auch serverseitige Auswirkungen. Als Grundlage dienen hier zunächst die in Kapitel 2.3 vorgestellten deklarativen UI-Frameworks in Bezug auf deren Funktionsumfang zur Gestaltung von UI. Da Backend-Driven UI eine Verbesserung der nativen Entwicklung darstellen soll, müssen grundlegende UI-Funktionen, wie Styling und Layouting ebenfalls abgebildet werden. Zusätzlich wurde die *.component*-Datenstruktur des Lona-Frameworks betrachtet, welche bereits einen breiten Funktionsumfang abbildet. Dazu wurden die Aspekte der Lona-Spezifikation mit aufgefasst. Der Bereich der Anwendungslogik wurde hier nur allgemeingültig abgedeckt, da keine spezifische Grundlage vorhanden ist. Der generelle Ansatz von Backend-Driven UI liefert hierzu keine expliziten Vorgaben. Dies gilt ebenfalls für die verwandten Arbeiten. Die Entwicklervorschau des Lona-Frameworks deckt diesen Bereich zum Zeitpunkt der Arbeit nicht ab.

Die nachfolgende Auflistung an Anforderungen wird zur genaueren Erläuterung nochmals unterteilt. Zunächst werden allgemeine Anforderungen an die strukturierte Darstellung betrachtet. Im Anschluss erfolgt eine Spezialisierung auf den Bereich der UI-Darstellung und der Anwendungslogik.

#### Allgemein

- Die strukturierte Darstellung der Anwendung muss fähig sein, die UI und Anwendungslogik einer Anwendung in einem einheitlichen Format darzustellen.
- Die strukturierte Darstellung muss fähig sein, dynamisch um weitere Informationen erweitert werden zu können.
- Die strukturierte Darstellung muss fähig sein, plattformübergreifend genutzt werden zu können.
- Die strukturierte Darstellung muss fähig sein, lokal persistiert werden zu können.

- Die strukturierte Darstellung muss fähig sein, automatisiert serialisiert und deserialisiert werden zu können.
- Die strukturierte Darstellung muss fähig sein, über Netzwerkanfragen verschickt werden zu können.
- Die strukturierte Darstellung muss fähig sein, validiert werden zu können.

#### Beschreibung von UI

- Die strukturierte Darstellung muss fähig sein, UI-Komponenten plattformübergreifend zu abstrahieren.
- Die strukturierte Darstellung muss fähig sein, UI-Komponenten in einer strukturierten Form zu beschreiben.
- Die strukturierte Darstellung muss fähig sein, UI-Komponenten in einer hierarchischen Struktur darzustellen.
- Die strukturierte Darstellung muss fähig sein, vollständige Anwendungsszenarien abzubilden.
- Die strukturierte Darstellung muss fähig sein, UI-Komponenten innerhalb eines Layouts zu positionieren.
- Die strukturierte Darstellung muss fähig sein, alle für die Initialisierung einer UI-Komponente benötigten Parameter abzudecken.
- Die strukturierte Darstellung muss fähig sein, visuelle Anpassungen von UI-Komponenten zu unterstützen.
- Die strukturierte Darstellung muss fähig sein, visuelle Anpassungen plattformübergreifend zu abstrahieren.

#### Beschreibung von Anwendungslogik

- Die strukturierte Darstellung muss fähig sein, Anwendungslogik plattformübergreifend zu abstrahieren.
- Die strukturierte Darstellung muss fähig sein, Anwendungslogik in einer strukturierten Form zu beschreiben.
- Die strukturierte Darstellung der Anwendung muss fähig sein, eine Navigation innerhalb der Anwendung zu ermöglichen.
- Die strukturierte Darstellung der Anwendung muss fähig sein, UI-Komponenten innerhalb der strukturierten Darstellungen referenzieren zu können.
- Die strukturierte Darstellung der Anwendung soll fähig sein, Algorithmik abbilden zu können.

### 4.1.3. Serverseitige Anforderungen

Nach Betrachtung der strukturierten Darstellung der Anwendung soll nun die Betrachtung der Server-Komponente erfolgen. Nach dem in Kapitel 2.1.3 beschriebenen Ansatz liegen die Aufgaben des Servers darin, UI und Anwendungslogik in Form der strukturierten Darstellung zu verwalten. Um weitere komplexe Anwendungslogik zudem nicht über die strukturierte Darstellung abbilden zu müssen, sondern diese stattdessen auf den Server auslagern zu können, sollten weitere optionale Endpunkte angeboten werden. Dieser grundlegende Funktionsumfang ist bereits aus dem Ansatz von Backend-Driven UI ableitbar. Bei der Betrachtung des Lona-Frameworks wurde zusätzlich noch die Verwendung einer DSL in Betracht gezogen, welche in Form einer zusätzlichen Abstraktionsschicht zur strukturierten Darstellung genutzt werden kann.

- Der Server muss fähig sein, die strukturierte Darstellung, in persistenter Form, verwalten zu können.
- Der Server muss fähig sein, die strukturierte Darstellung, in persistenter Form, einlesen zu können.
- Der Server soll fähig sein, eine DSL als zusätzliche Abstraktionsschicht zur strukturierten Darstellung nutzen zu können.
- Der Server muss fähig sein, auf Anfragen des Clients reagieren zu können.
- Der Server muss fähig sein, die strukturierte Darstellung, über eine Schnittstelle, dem Client zur Verfügung zu stellen.
- Der Server muss fähig sein, unterschiedliche Versionsstände von Clients zu unterstützen.
- Der Server soll fähig sein, dynamisch Änderungen für unterschiedliche Endgeräte vornehmen zu können.
- Der Server muss fähig sein, weitere Schnittstellen mit zusätzlicher Anwendungslogik bereitstellen zu können.

### 4.1.4. Clientseitige Anforderungen

Als letzte der drei Hauptkomponenten soll der Client betrachtet werden. Hierbei handelt es sich um die Anwendung, welche durch den Nutzer verwendet wird. Wie bereits in Kapitel 2.1.3 beschrieben besteht der Client hauptsächlich aus einem Parser für die strukturierte Darstellung, welche dann in native UI und Anwendungslogik überführt wird. Der Client muss daher nicht nur in der Lage sein diese zu verarbeiten, sondern diese auch aktiv durch den Server zu beziehen. Zusätzlich wurde bereits bei den Nachteilen des Backend-Driven UI Ansatzes die mögliche Implementierung eines Caches betrachtet. Wie bereits bei den Anforderungen an die strukturierte Darstellung existiert hier keine Wissensgrundlage zur Umsetzung von externer Anwendungslogik auf dem Client. Da komplexe Anwendungslogik bereits durch den Server abgedeckt werden kann, wird clientseitig ein geringerer Funktionsumfang vorausgesetzt. Der Fokus liegt

hier besonders auf der Interaktion mit UI-Komponenten sowie reaktiver Anpassungen durch Nutzereingaben.

- Der Client muss in Form einer nativen Anwendung umgesetzt werden.
- Der Client muss Nutzern die Möglichkeit bieten, mit einer funktionsfähigen Anwendung zu interagieren.
- Der Client muss fähig sein, Daten von einem Server anzufragen.
- Der Client muss fähig sein, die strukturierte Darstellung zu verarbeiten und in native UI zu überführen.
- Der Client muss fähig sein, grundlegende, plattformspezifische UI-Komponenten darzustellen.
- Der Client muss fähig sein, UI-Komponenten visuell anzupassen.
- Der Client muss fähig sein, die strukturierte Darstellung der Anwendung in native Anwendungslogik zu überführen.
- Der Client muss fähig sein, Nutzereingaben zu verarbeiten.
- Der Client muss fähig sein, auf den aktuellen Zustand von UI-Komponenten zugreifen zu können.
- Der Client muss fähig sein, Informationen zur aktuellen UI an den Server weitergeben zu können.
- Der Client soll fähig sein, den Zustand der Anwendung über eine Session hinaus zu persistieren.
- Der Client soll fähig sein, dem Server allgemeine Geräteinformationen bereitstellen zu können.
- Der Client soll fähig sein, ein latenzarmes Caching bereitstellen zu können.

### 4.1.5. Qualitative Anforderungen

Abschließend folgt die Betrachtung von qualitativen Anforderungen. Diese sind allgemeingültig für das gesamte Framework und somit für alle bereits betrachteten Komponenten relevant. Als Grundlage für die qualitativen Anforderungen, gelten besonders die in Kapitel 2.1.3 beschriebenen Nachteile des Backend-Driven UI Ansatzes, wie beispielsweise der erhöhte Datenverbrauch oder die Antwortzeit der Anwendung.

- Das Framework muss fähig sein, schnell auf Nutzereingaben zu reagieren.
- Das Framework muss fähig sein, den Datenaustausch zwischen Client und Server zu minimieren.

## 4.2. Architektur

Im Anschluss an die Anforderungsermittlung und in Bezug auf die in Kapitel 1.3 aufgestellten Forschungsfragen soll nun im Rahmen der Konzeption eine Architektur aufgestellt werden. Als Grundlage werden hier die in Kapitel 4.1 aufgestellten Anforderungen verwendet, woraus die drei Hauptkomponenten *Client*, *Strukturierte Darstellung* und *Server* abgeleitet wurden. Durch eine erste Architektur sollen zunächst Zuständigkeiten sowie Beziehungen unter den Komponenten bestimmt werden. Dazu soll eine Unterteilung in weitere Unterkomponenten erfolgen. Die in Abbildung 4.1 dargestellte Architektur soll dann als Grundlage für eine erste prototypische Umsetzung genutzt werden.

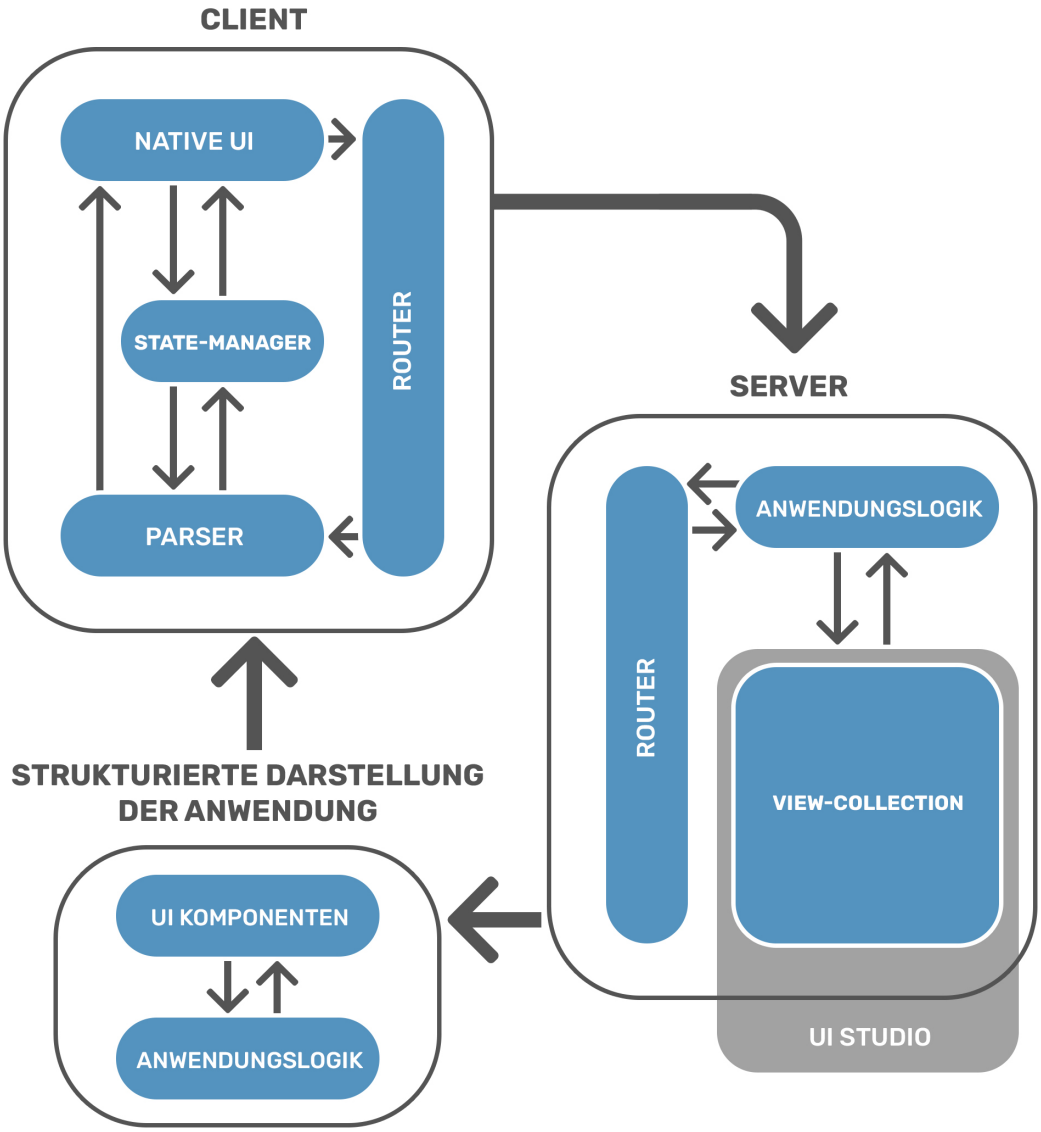



Abbildung 4.1.: Initiale Architektur

Die Ausgangssituation bildet zunächst der Client. Hierbei handelt es sich um eine native Anwendung, welche durch den Nutzer verwendet werden soll. Bei einer Backend-Driven UI Anwendung besitzt der Client keine konkreten Informationen zur UI oder Anwendungslogik. Diese muss zunächst serverseitig angefragt werden. Bei der Nutzung der Anwendung wird daher zunächst eine Anfrage an den Server gestellt. Für das Verarbeiten von Anfragen innerhalb des Servers ist der Router verantwortlich. Dieser stellt eine API bereit und leitet Anfragen dann an weitere interne Anwendungslogik weiter. Die Anwendungslogik muss hier zweierlei Aufgabenbereiche abdecken. Zunächst müssen die für den Client relevante anwendungsbezogene Informationen bereitgestellt werden. Die gesamte Anwendung wird dabei in einzelne Views unterteilt, welche zentral in Form einer View-Collection gesammelt sind. Zur besseren Erläuterung ist der Aufbau einer einzelnen View in der folgenden Abbildung 4.2 dargestellt.

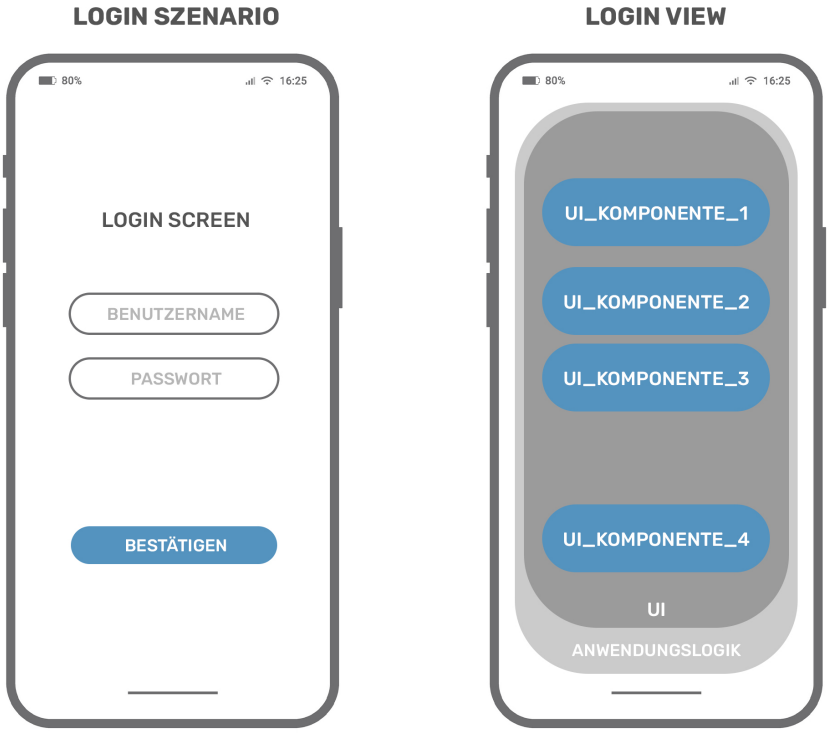



Abbildung 4.2.: Gegenüberstellung eines Szenarios und einer View am Beispiel eines Login-Screens

Eine View umfasst ein konkretes Anwendungsszenario, wie beispielsweise einen in Abbildung 4.2 dargestellten Login-Screen. Der Login-Screen besteht zunächst aus einem Titel am oberen Bildschirmrand, zwei Eingabefeldern zur Eingabe eines Benutzernamens sowie Passworts und einem Button zur Bestätigung der Eingabe. Für das dargestellte Szenario muss eine entsprechende View zunächst alle benötigten UI-Komponenten beinhalten, welche in Abbildung 4.2 als generische Komponenten zusammengefasst werden. Zudem muss eine View die zugehörige Anwendungslogik für das jeweilige Szenario bereitstellen.

Bei der in Abbildung 4.1 dargestellten View-Collection, innerhalb des Servers, handelt es sich dann um eine persistierte Sammlung der einzelnen Views in Form der strukturierten Darstellung. Die initiale Architektur sieht hierbei vor, dass diese Sammlung innerhalb des Servers verwaltet wird. Es ist dennoch auch möglich, die View-Collection in einen externen Dienst auszulagern oder in einer Datenbank abzulegen. Zusätzlich sieht die Architektur noch eine optionale UI-Studio-Komponente vor, welche eine mögliche grafische Oberfläche zur Generierung von Views darstellt. Sobald nun eine konkrete View durch den Client angefragt wird, kann diese der View-Collection entnommen und über den Router an den Client gesendet werden.

Wie bereits dargestellt, erfolgt die Persistierung einzelner Views anhand der strukturierten Darstellung. Um negative Auswirkungen auf die Performance der Anwendung zu verhindern, sollen Views nun möglichst ohne zusätzliche Konvertierungen an den Client gesendet werden können. Die strukturierte Darstellung muss daher ein gültiges Format repräsentieren, welches sowohl eine Persistierung als auch einen Versand über eine Netzwerkanfrage ohne Zwischenschritte ermöglicht. Zudem muss diese alle für das Szenario darzustellende UI-Komponenten umfassen. Aus den in Kapitel 4.1.2 aufgestellten Anforderungen muss es zudem möglich sein, UI-Komponenten hierarchisch anzuordnen und visuelle Änderungen vorzunehmen. Bei der zweiten Unterkomponente handelt es sich um die Anwendungslogik, welche für das jeweilige Szenario und die damit verbundenen UI-Komponenten benötigt wird. Um die Anzahl notwendiger Anfragen zwischen Client und Server zu reduzieren, sollte diese möglichst auch in die strukturierte Darstellung integriert sein.

Zuletzt empfängt der Client dann die angeforderte View. Diese wird zunächst innerhalb des Routers aus der Serverantwort extrahiert und an den Parser weitergegeben. Der Parser wertet die strukturierte Darstellung daraufhin aus und überführt diese in native UI-Komponenten. Dazu wird die übermittelte Anwendungslogik umgesetzt. Die daraus resultierende native UI ist ein grundlegender Bestandteil des Clients und stellt die Interaktionsebene zwischen dem System und dem Nutzer dar. Dazu ist sie für das visuelle Darstellen von Informationen sowie die Verarbeitung von Nutzereingaben zuständig. Zusätzlich können Nutzerangaben zusätzliche Netzwerkanfragen an den Server auslösen, was dann an den Router weitergegeben wird. Innerhalb des Routers ist zusätzlich die Umsetzung von Caching zum lokalen Persistieren von Netzwerkanfragen möglich.

Aus den in Kapitel 4.1.4 aufgestellten Anforderungen geht zudem hervor, dass ein Zugriff auf den aktuellen Zustand der UI-Komponenten gewährleistet sein muss, da sonst keine nachträgliche Änderung der UI oder das Weiterleiten von Zustandsinformationen an den Server möglich ist. Um dies zu unterstützen, wird ein zusätzlicher State-Manager benötigt, welcher für die Verwaltung des Zustands der aktuellen UI zuständig ist. Der initiale Zustand der UI wird nach der Generierung durch den Parser an den State-Manager übermittelt. Sollte es zu nachträglichen Zustandsänderungen, beispielsweise durch ausgeführte Anwendungslogik, kommen, wird diese von der UI direkt an den State-Manager weitergegeben. Um zusätzlich die in Kapitel 2.3 genannten Vorteile von deklarativer UI zu nutzen, dient der State-Manager für einzelne UI-Komponenten zudem als einheitliche Datenbasis. Die UI kann so beispielsweise dynamisch aktualisiert werden, sobald es eine Zustandsänderung in der Datenbasis geben sollte.

# 5. Umsetzung

Die in der Konzeption erarbeitete Architektur soll nun im Rahmen einer technischen Umsetzung implementiert werden. Der Fokus eines ersten Prototyps liegt hier auf der iOS-Plattform. Im Anschluss soll eine Erläuterung der aufgetretenen Probleme und Herausforderungen folgen. Abschließend soll dann betrachtet werden, inwieweit die Erkenntnisse aus der prototypischen Umsetzung auf die Android-Plattform anwendbar sind.

## 5.1. Entwicklung

Zunächst sollen die technische Entwicklung der in Kapitel 4.2 aufgestellten Architektur erfolgen. Diese stellt zunächst die drei Hauptkomponenten *Client*, *strukturierte Daten* und *Server* sowie den allgemeinen Datenfluss zwischen diesen dar. Dazu wurde anhand der Anforderungen aus Kapitel 4.1 eine erste Aufteilung in Unterkomponenten vorgenommen. Dies soll nun im Rahmen einer ersten prototypischen Umsetzung weiter spezifiziert werden. Mit der Bezeichnung „prototypisch“ wird sich hier auf den allgemeinen Funktionsumfang bezogen. So gilt es weiterhin, die durch die Anforderungen spezifizierten Funktionalitäten bestmöglich abzudecken, allerdings wird nur eine Teilmenge an verfügbaren UI-Komponenten betrachtet. Für den Prototyp wurde sich zudem auf die iOS-Plattform festgelegt. Grundlage hierfür ist das in Kapitel 2.3 vorgestellte und für iOS-Plattform verfügbare deklarative UI-Framework *SwiftUI*. Bereits bei der Betrachtung von mobiler UI wurden die Vorteile von deklarativer UI beschrieben. In Kapitel 4.2 wurde zudem beschrieben, wie diese Vorteile auch im Rahmen von Backend-Driven UI genutzt werden können. Auch wenn die Android-Plattform mit *Jetpack Compose* ein vergleichbares Framework anbietet, stellt SwiftUI aufgrund des längeren Bestehens die etabliertere Technologie dar (Steinberger, 2021). Des Weiteren handelt es sich bei Android zwar um das meist verbreitetste Betriebssystem, insgesamt stellt die iOS-Plattform allerdings die profitablere Plattform dar. So konnte im Jahr 2021 ein Umsatz von 85,1 Milliarden Dollar durch den digitalen Vertrieb von Anwendungen generiert werden, während der Umsatz der Android-Plattform bei 47,9 Milliarden Dollar liegt (Chan, 2021)

Im Folgenden soll daher nun die Ergebnisse des prototypischen Entwicklungsprozesses aufgezeigt werden. Zunächst folgt eine Bestimmung des umzusetzenden Funktionsumfangs. Anschließend soll die technische Umsetzung der Kapitel 4.2 beschriebenen Unterkomponenten, betrachtet werden. Der gesamte Prototyp steht zusätzlich in Form eines Github Repositorys[1] zur Verfügung.

[1] github.com/Dominikdeimel/Cross-Plattform-BackendDrivenUI

### 5.1.1. Unterstütze UI-Komponenten

Der Fokus von Backend-Driven UI liegt auf der serverseitigen Bestimmung von UI und Anwendungslogik. Damit der Client in der Lage ist, die durch den Server bereitgestellte Datenstruktur in native UI zu überführen, muss im Vorhinein bestimmt werden, welche UI-Komponenten im Rahmen des Frameworks unterstützt werden sollen. Im Folgenden soll daher als Grundlage für die strukturierte Darstellung zunächst der Umfang an zu unterstützenden UI-Komponenten bestimmt werden.

Für die Auswahl der UI-Komponenten werden die in Kapitel 2.3 beschriebenen Design Guidelines hinzugezogen. Obwohl der Fokus des Prototyps auf der Umsetzung für die iOS-Plattform liegt, soll trotzdem eine möglichst allgemeingültige Datenstruktur geschaffen werden. Die Auswahl der UI-Komponenten erfolgt daher anhand der Apple Human-Interface-Guidelines und den Material-Design-Guidelines als entsprechende Rahmenform der Android-Plattform.

Zur Bestimmung der für die prototypische Umsetzung relevanten UI-Komponenten wurde zunächst eine Übersicht aller verfügbaren Komponenten auf Basis der Design-Guidelines geschaffen. Im Anschluss wurden Gemeinsamkeiten identifiziert und eine einheitliche Sammlung an UI-Komponenten abgeleitet. Diese ist in Tabelle 5.1 dargestellt. Zur Beschreibung der Komponenten wurde zunächst eine allgemeingültige Abstraktion gewählt. Diese dient als einheitliche Bezeichnung und soll daher als Typenbezeichnung innerhalb der strukturierten Darstellung gewählt werden. Dazu enthält die Übersicht eine kurze textuelle Beschreibung der jeweiligen Komponente. In Vorbereitung auf die folgende technische Umsetzung wurden zudem noch die konkreten iOS- und Android-Komponenten aufgeführt, welche im Rahmen der in Kapitel 2.3 vorgestellten deklarativen UI-Frameworks verwendet werden können. Eine ausführliche Auflistung aller plattformspezifischen Komponenten auf Basis der Design-Guidelines ist im Anhang A.1 und A.2 zu finden.

Nach Bestimmung der UI-Komponenten muss eine Auswahl von zu unterstützenden Modifikatoren getroffen werden. Bei Modifikatoren handelt es sich um optionale Anpassungen, welche auf UI-Komponenten angewendet werden können, um beispielsweise Abstände, Farben oder Formen anzupassen (García u. a., 2015). Hierfür wurden zunächst die in Tabelle 5.1 abgebildeten Komponenten als Grundlage gewählt und jeweils verfügbare plattformspezifische Modifikatoren bestimmt. Dies ist in Tabelle 5.2 dargestellt. Anzumerken ist hierbei, dass allgemeingültige Anpassungen, welche auf alle Komponenten anwendbar sind, unter „Allgemeine Modifikatoren“ zusammengefasst sind und lediglich Komponenten mit spezifischen Modifikatoren gesondert aufgelistet sind. Anhand dieser Übersicht fand dann zunächst eine Priorisierung der Modifikatoren aufgrund ihrer plattformübergreifenden Verwendbarkeit statt. Daraufhin wurde eine finale Sammlung an zu unterstützenden UI-Komponenten abgeleitet, welche in Tabelle 5.3 abgebildet ist und ebenfalls allgemeingültige Abstraktion sowie eine kurze Beschreibung jedes Modifikators enthält.

Die so bestimme Sammlung an UI-Komponenten und zugehörigen Modifikatoren dient als Grundlage für die prototypische Umsetzung. Hierbei handelt es sich allerdings nur um einen Ausschnitt an möglicher UI, welche als Leitfaden für die Erstellung einer

geeigneten Datenstruktur genutzt werden soll. Ausblickend kann die Erweiterung um zusätzliche UI-Komponenten und Modifikatoren in Betracht gezogen werden.

| **Abstraktion** | **Beschreibung** | **iOS-Komponente** | **Android-Komponente** |
|---|---|---|---|
| Image | Darstellung von Bildern | Image | Image |
| Text | Darstellung von Text | Text | Textfield |
| TextInput | Textfeld zur Eingabe von Text durch den Nutzer | TextField | Textfield |
| List | Repräsentation von mehreren Daten in einer vertikalen Anordnung | List | Column / LazyList |
| Button | Steuerelement zur Auslösung einer Funktion | Button | Button |
| Alert | Overlay mit wichtigen Informationen, welches den Flow der Anwendung stoppt | Alert | Dialog |
| Slider | Element zur Auswahl eines Wertes innerhalb einer vorgegebenen Spanne | Slider | Slider |
| TabView | Steuerelement zur Navigation zwischen unabhängigen Bereichen | Tab bars | Navigation Bar / Navigation Tabs |
| Switch | Steuerelement zur Auswahl zwischen zwei Zuständen | Toogle | Switch |
| Card | Visuelle Gruppierung von zusammenhängenden Elementen | GroupBoxes | Card |
| Spacer | Abgrenzung zwischen Elementen, welche entweder sichtbar oder unsichtbar sein kann | Spacer | Divider |
| Modal | Overlay über Großteil der Anwendung | Sheets | bottomSheetScaffold |
| Row, Column, Box | Unsichtbare Container-Elemente zum Strukturieren von Elementen | VStack, Hstack, ZStack | Column, Row, Box |

Tabelle 5.1.: Übersicht über für die prototypische Umsetzung relevanten Komponenten

### 5.1.2. Strukturierte Darstellung der Anwendung

Die strukturierte Darstellung stellt eine der Hauptkomponenten des Frameworks. Sie ist nicht nur ausschlaggebend für den Funktionsumfang, sondern stellt auch das Bindeglied zwischen Client und Server dar. Im nachfolgenden Kapitel soll daher eine geeignete Datenstruktur für die Umsetzung der strukturierten Darstellung bestimmt werden. Dafür soll zunächst ein geeignetes Format identifiziert werden und im Anschluss die Abbildung der in Kapitel 5.1.1 bestimmten UI-Komponenten und Modifikatoren erfolgen.

#### Format

Bereits in den Grundlagen (Kapitel 2.1.3) wurde herausgearbeitet, dass sich zur Umsetzung einer strukturierten Datenstruktur der Einsatz von verschiedenen Datenformaten oder einer DSL anbietet. Die Nutzung einer DSL bietet Entwicklern eine starke Freiheit bei der Umsetzung einer Datenstruktur und ist so auch dynamisch anpassbar, insofern beispielsweise Erweiterungen eingepflegt werden sollen. Dennoch erhöht sich der Aufwand bei der Client-Server Kommunikation, da eine Überführung in ein einheitlich austauschbares Format wie JSON stattfinden muss. Es wird daher eine zusätzliche In-

| Komponente | Modifikatoren iOS | Modifikatoren Android |
|---|---|---|
| Allgemeine Modifikatoren | foreGroundColor,<br>background,<br>font,<br>fontWeight,<br>padding,<br>frame,<br>shadow,<br>disabled | background,<br>color,<br>border,<br>padding,<br>offset,<br>scale,<br>rotate,<br>shadow |
| Image | resizeable,<br>aspectRatio,<br>shape | contentScale,<br>clip |
| Text View | italic,<br>bold,<br>underline,<br>fontWidth | fontSize,<br>fontStyle,<br>fontWeight |
| Text Input | hoverEffect,<br>italic,<br>bold | fontSize,<br>fontStyle,<br>fontWeight |

Tabelle 5.2.: Übersicht über plattformspezifische Modifikatoren

| Abstraktion | Beschreibung |
|---|---|
| ForegroundColor | Anpassung der Vordergrundfarbe für bspw. Textfarbe |
| BackgroundColor | Anpassung der Hintergrundfarbe |
| FontSize | Anpassung der Schriftgröße |
| FontStyle | Anpassung des Textstyle für bspw. fett gedruckt oder kursiv |
| Border | Darstellung eines visuellen Rahmens um eine UI-Komponente |
| Shadow | Darstellung eines Schattens zu einer UI-Komponente |
| Shape | Anpassung der Form einer UI-Komponente |
| Size | Anpassung der Größe einer Komponente |
| Padding | Anpassung des Abstands einer Komponente zu ihrer Umgebung |
| isActive | Anpassung zum (De-) Aktivieren einer Komponente |

Tabelle 5.3.: Übersicht der für die prototypische Umsetzung relevanten Modifikatoren

terpretationsebene benötigt, was in einem höheren Entwicklungsaufwand resultiert. Im Rahmen des Lona Frameworks wird die DSL zur serverseitigen Bestimmung von Views genutzt. Wird eine solche View dann durch den Client angefragt, erfolgt zunächst eine Umwandlung in das JSON Format. Vorteilig ist dennoch die automatische Ableitung einer Dokumentation der Datenstruktur (Kelly u. Silverman, 2019). Um allerdings den zusätzlichen Overhead durch eine weitere Parser-Ebene zu vermeiden, soll im Rahmen des Frameworks zunächst keine DSL zum Einsatz kommen. Dies orientiert sich zudem an den in Kapitel 4.1.5 aufgestellten qualitativen Anforderungen.

Im Rahmen der betrachteten Datenformate wurde sich dann für *JSON* entschieden. Dies orientiert sich an der Umsetzung von SiriusXM (Kapitel 3.2). Zur Auswahl standen die Datenformate JSON und XML, welche in Kapitel 2.2 vorgestellt wurden. Im Vergleich wurde zunächst der Funktionsumfang betrachtet. Hierbei bieten sich grundsätzlich beide Formate zur Umsetzung der Anforderung an, da sowohl beide in der Lage sind, Entitäten mit Eigenschaften in einer hierarchischen Struktur und in Listen-Form abzubilden. Zudem existieren auf den Plattformen iOS und Android bereits native Darstellungsformen eines JSON-Objekts. Ebenfalls handelt es sich um eine im Web-Bereich etablierte Technologie, wodurch die Verwendung innerhalb eines Server-Kontextes erleichtert wird. Ausschlaggebend für Wahl von JSON ist allerdings der Unterschied im Speicherbrauch. Zum Vergleich wurden der Speicherbedarf der in Kapitel 2.2 dargestellten und inhaltlich gleichen XML- und JSON-Struktur in Tabelle 5.4 verglichen.

| | **Speicherbedarf in Byte** | **Speicherbedarf in Prozent** |
|---|---|---|
| **XML** | 1.360 Byte | 100 % |
| **JSON** | 969 Byte | 71,25 % |

Tabelle 5.4.: Vergleich des Speicherbedarfs von XML und JSON

Die XML-Datei benötigt 1.360 Byte, während die JSON-Datei lediglich 969 Byte an Speicher belegt. Eine vergleichbare JSON Datei benötigt daher ca. 28,75 % weniger Speicherplatz. Auch wenn dieses Delta aufgrund der dargestellten Daten variieren kann, ist aufgrund der effizienteren Syntax von einem kontinuierlichen Speichervorteil auszugehen. Dieser Unterschied ist vorwiegend auf die unterschiedliche Deklaration von Elementen zurückzuführen. Das XML Format nutzt hierfür Tags und benötigt für Elemente mit Werten jeweils ein öffnendes und schließendes Tag. Beide Tags enthalten die Benennung des jeweiligen Elementes, was zu einer Redundanz führt. Des Weiteren benötigt eine JSON-Datei keine zusätzliche Annotation zu Beginn einer Datei. Auch wenn der Speicherverbrauch nur ein mögliches Vergleichskriterium darstellt, eignet sich dieser für eine erste Gegenüberstellung. Weitere Kriterien, wie beispielsweise der Verarbeitungsaufwand, sind zusätzlich von der Wahl des genutzten Parsers abhängig und erfordern daher einen tiefgreifenderen Vergleich.

#### Darstellung von UI-Komponenten

Bei der eigentlichen strukturierten Datenstruktur soll es sich um ein möglichst offenes Format handeln. Dadurch soll es im Anschluss an die prototypische Umsetzung möglich sein, zusätzliche UI-Komponenten und Modifikatoren umsetzen zu können und so den Funktionsumfang des Frameworks dynamisch zu erweitern. Die Darstellung einer einzelnen View, erfolgt in Form einer dedizierten JSON-Datei. Eine oder mehrere Views stellen, wie bereits in Kapitel 4.2 beschrieben, die Repräsentation eines konkreten Anwendungsszenarios dar und setzen sich aus einer oder mehreren UI-Komponenten zusammen. Innerhalb einer View existiert, ähnlich zu dem Ansatz von HTML, immer eine Root-Komponente, welche dann weitere verschachtelte Komponenten enthalten kann.

Die Repräsentation einer UI-Komponente geschieht in Form eines einzelnen JSON-Objekts, welches mit zusätzlichen Eigenschaften versehen wird. Für jede Komponente gibt es fest vorausgesetzte Eigenschaften. Diese sind zum einen *id*, zur eindeutigen Identifizierung und zum anderen *type* zur clientseitigen Bestimmung des Typs der jeweiligen Komponente. Dazu gibt es optionale Eigenschaften, welche clientseitig zur Initialisierung nativer UI benötigt werden und von der jeweiligen Komponente abhängig sind. Die geplante Struktur soll im Folgenden anhand verschiedener Beispiele weiter verdeutlicht werden.

```
{
    "id": "t1",
    "type": "TEXT",
    "text": "Lorem Ipsum"
}
```

Listing 4: Strukturierte Darstellung einer generischen Text-Komponente in JSON

Zur genaueren Erläuterung zeigt Listing 4 eine strukturierte Darstellung einer generischen Text-Komponente. Zunächst folgen die vorausgesetzten Eigenschaften. Für *id* muss ein einzigartiger Wert angegeben werden. Dieser dient im weiteren Verlauf zur eindeutigen Identifizierung der Komponente. Bei der zweiten vorausgesetzten Eigenschaft handelt es sich um *type*, welche in einem einheitlichen Format den Typ der Komponente bestimmt. Dazu benötigt die Text-Komponente noch die Eigenschaft *text*, welche die auf dem Client darzustellende Zeichenkette repräsentiert.

```
{
    "id": "c1",
    "type": "Column",
     "children": [
        {
            "id": "t1",
            "type": "TEXT",
            "text": "Lorem Ipsum"
        },
        {
            "id": "t2",
            "type": "TEXT",
            "text": "dolor sit amet"
        },
    ]
}
```

Listing 5: Strukturierte Darstellung einer generischen Text-Komponente in JSON

Eine besondere Untergruppe von UI-Komponenten sind die Container-Komponenten *Row*, *Column*, und *Box*, welche zur Verschachtelung von Komponenten dienen. Bei einer *Row* kann die Verschachtelung entlang der x-Achse, bei einer *Column* entlang der y-Achse und bei einer *Box* entlang der z-Achse erfolgen. Diese Verschachtelung geht mit einem automatischen Layout einher. Alle beschriebenen Container-Komponenten benötigen hierfür die zusätzliche Eigenschaft *children*, welche als Array abgebildet wird.

In dieses Arrays können beliebig UI-Komponenten eingefügt oder durch die Nutzung von weiteren Containern verschachtelt werden. Eine exemplarische Darstellung einer *Column* ist in Listing 5 dargestellt. Hierbei werden die zwei Text-Komponenten *t1* und *t2* innerhalb einer *Column* horizontal angeordnet.

Wie anhand der vorangegangenen Beispiele zu erkennen ist, benötigten verschiedene Komponenten unterschiedliche Eigenschaften. Neben einer ID und einem Typen werden, abhängig von der jeweiligen Komponente, weitere Eigenschaften für die Nutzung benötigt. Dies gilt beispielsweise bei der Eigenschaft *children* für die aufgeführten Container-Komponenten. Diese zusätzlichen Eigenschaften stellen einen essenziellen Teil bei der clientseitigen Überführung in native UI dar und wurden daher anhand der für die native Initialisierung von nativen UI-Elementen benötigten Parametern bestimmt. Dazu existieren weitere, vollständig optionale Eigenschaften, welche ausschließlich bei Bedarf gesetzt werden können. Die gilt beispielsweise bei der Umsetzung von Buttons. Diese können zur genaueren Darstellung noch mit einem zusätzlichen Text versehen werden und besitzen daher eine optionale Eigenschaft *text*.

Um einen allgemeinen Überblick über die benötigten Eigenschaften zu geben, soll die folgende Tabelle 5.5 als Übersicht über alle in Kapitel 5.1.1 bestimmten UI-Komponenten dienen. Anzumerken ist hierbei, dass auf die allgemein vorausgesetzten Eigenschaften *id* und *type* aus Gründen der Redundanz verzichtet wurde. Zusätzlich werden die Datentypen *TabViewElement*, *Modifier*, *Validator* und *Action* erwähnt, welche im Folgenden genauer vorgestellt werden sollen.

| **Komponente** | **Benötigte Eigenschaften** | **Optionale Eigenschaften** |
|---|---|---|
| Image | *imagePath*: String | *modifier*: [Modifier] |
| Text | *text*: String | *modifier*: [Modifier] |
| TextInput | | *text*: String<br>*validator*: Validator<br>*modifier*: [Modifier] |
| List | *children*: [UIComponent] | *modifier*: [Modifier] |
| Button | *action*: Action | *text*: String<br>*isEnabled*: Boolean<br>*modifier*: [Modifier] |
| Alert | *text*: String | *message*: String<br>*modifier*: [Modifier] |
| Slider | *rangeStart*: Int<br>*rangeEnd*: Int | *modifier*: [Modifier] |
| TabView | *tabViews*: [TabViewElement] | *modifier*: [Modifier] |
| Toggle | *text*: String | *modifier*: [Modifier] |
| Card | *children*: [UIComponent]<br>*text*: String<br>*icon*: String | *modifier*: [Modifier] |
| Spacer | | *modifier*: [Modifier] |
| Modal | *children*: [UIComponent] | *modifier*: [Modifier] |
| Row, Column, Box | *children*: [UIComponent] | *modifier*: [Modifier] |

Tabelle 5.5.: Übersicht über UI-Komponenten mit zugehörigen Eigenschaften

Zunächst soll auf den Typ *TabViewElement* eingegangen werden. Bei einer TabView-Komponente handelt es sich um ein Steuerelement, welches die Navigation zwischen verschiedenen Tabs ermöglicht. Eine TabView-Komponente umfasst dabei mindestens zwei TabView-Elemente, wobei jedes Element eine eigenständige View darstellen kann. Eine exemplarische Darstellung einer TabView mit zwei TabView-Elementen ist in Abbildung 5.1 dargestellt. Durch die dargestellten Buttons, welche automatisch erzeugt werden, können Nutzer dann zwischen den zugehörigen Views wechseln. Um zwischen verschiedenen Tabs unterscheiden zu können, werden diese jeweils mit einem Icon sowie einem Namen versehen. Insgesamt besteht ein TabView-Element daher aus einem Namen, einem Icon so wie einer View, welche in Form von verschachtelten UI-Komponenten dargestellt wird.


(a) Tab 1 (b) Tab 2

Abbildung 5.1.: TabView mit zwei Tabs

### Darstellung von Modifikatoren

Ein weiterer zusammengesetzter Datentyp ist *Modifier*, welcher zur Abbildung von Modifikatoren genutzt wird. Bei einem Modifikator handelt es sich um eine optionale, visuelle Anpassung einer UI-Komponente. Da Modifikatoren auf alle UI-Komponenten angewendet werden können, verfügt jede in Tabelle 5.5 aufgeführte Komponente über ein Array vom Typ *Modifier*. Für den Fall, dass eine Komponente nicht modifiziert werden soll, handelt es sich bei *modifier* um eine optionale Eigenschaft. Ein Modifikator wird ebenfalls in Form eines JSON-Objekts abgebildet. Ähnlich zu den UI-Komponenten besitzen Modifikatoren fest vorausgesetzte, individuelle und optionale Eigenschaften, welche für die weitere Verarbeitung aufseiten des Clients relevant sind.

Listing 6 zeigt exemplarisch die Umsetzung einer Text-Komponente, auf welche ein Farb- und ein Textgrößen-Modifikator angewendet wird. Beide Modifikatoren werden in Form eines JSON-Objekts dargestellt und innerhalb der Eigenschaft *modifier* in einem Array abgelegt. Die Reihenfolge kann hierbei relevant sein, da das Array während

des clientseitigen Render-Prozesses sequenziell abgearbeitet wird. Jeder Modifikator muss die Eigenschaft *type* besitzen. Wie bereits bei den UI-Komponenten gibt diese Eigenschaft Auskunft über den jeweiligen Typ des Modifikators, was bei der clientseitigen Verarbeitung relevant wird. Bei den in Listing 6 dargestellten Modifikatoren stellen die Eigenschaften *color* und *fontSize* eine für den jeweiligen Modifikator relevante Eigenschaft dar. Die nachfolgende Tabelle 5.6 soll einen gesammelten Überblick über die Eigenschaften der Modifikatoren geben.

```
{
    "id": "t1",
    "type": "TEXT",
    "text": "Lorem Ipsum"
    "modifier": [
        {
            "type": "FOREGROUND_COLOR",
            "color": "BLUE"
        },
        {
            "type": "FONTSIZE",
            "fontSize": 30
        }
    ]
}
```

Listing 6: Strukturierte Darstellung einer generischen Text-Komponente mit zwei Modifikatoren

| **Modifier** | **Eigenschaften** |
|---|---|
| ForegroundColor | *color*: String |
| BackgroundColor | *color*: String |
| FontSize | *fontSize*: Int |
| FontStyle | *fontStyle*: String |
| Border | *color*: String = ”BLACK”<br>*width*: Int = 1 |
| Shadow | *color*: String = ”BLACK”<br>*radius*: Int = 10<br>*x*: Int = 5<br>*y*: Int = 5 |
| Shape | *shape*: String<br>*color*: String = ”BLACK”<br>*radius*: Int = 25<br>*stroke*: Int = 0 |
| Size | *width*: Int<br>*height*: Int = width |
| Padding | - |
| IsActive | *isActive*: Boolean |

Tabelle 5.6.: Übersicht über Modifikatoren mit zugehörigen Eigenschaften

Auch bei der Umsetzung von Modifikatoren müssen zur Initialisierung nicht alle Eigenschaften gesetzt werden. Es wird hier allerdings nicht zwischen vorausgesetzten und optionalen Werten unterschieden, sondern mit Standardwerten gearbeitet. Diese sind in der gezeigten Tabelle durch einen expliziten Wert hinter der Variablenbenennung gekennzeichnet. Der Modifikator *Border* setzt beispielsweise die Eigenschaften *color* und *width* voraus. Sollten diese allerdings innerhalb der strukturierten Darstellung nicht gesetzt werden, wird bei der clientseitigen Überführung auf die Standardwerte „*BLACK*“ und *1* zurückgegriffen. Für die genutzten Standardwerte wurde sich an den jeweiligen nativen Implementierungen der iOS- und Android-Plattform orientiert. Anzumerken sind zudem noch die Eigenschaften *fontStyle* und *shape*, da es sich um eine Enumeration handelt und ausschließlich vordefinierte Werte erwartet werden. Diese sind im Anhang A.3 genauer dargestellt. Letztlich stellt der Modifikator *Padding* eine Besonderheit dar, da dieser keine Eigenschaften besitzt, zur Vollständigkeit aber mit in Tabelle 5.6 aufgenommen wurde.

### Darstellung von Anwendungslogik

Bei dem letzten zusammengesetzten Datentypen aus Tabelle 5.5 handelt es sich um *Action*. Eine Action beschreibt eine Aktion aufgrund einer konkreten Nutzereingabe und dient im Rahmen der prototypischen Umsetzung für die Darstellung von Anwendungslogik. Zum Auslösen von Aktionen wird innerhalb des Prototyps zunächst nur die Button-Komponente in Kombination mit einer On-Click Aktion genutzt. Eine Implementierung ist aber aufgrund der offenen Datenstruktur auch nachträglich für weitere Komponenten möglich. Der Datentyp *Action* beinhaltet alle für die Ausführung von Anwendungslogik notwendigen Informationen und kann serverseitig bestimmt werden. Zur Umsetzung einer Aktion stehen bereits im Rahmen des Prototyps verschiedenen Optionen zur Verfügung, welche in der folgenden Tabelle 5.7 dargestellt sind.

| **Option** | **Beschreibung** |
|---|---|
| REQUEST_WITH_SCREEN_CHANGE | Sendet eine Anfrage an den Server und erwartet eine neue View zum Darstellen |
| REQUEST_WITH_PAYLOAD _AND_SCREEN_CHANGE | Sendet eine Anfrage mit vorab bestimmten Daten, an den Server und erwartet eine neue View zum Darstellen |
| REQUEST_WITH_PAYLOAD _AND_UI_CHANGES | Sendet eine Anfrage mit vorab bestimmten Daten, an den Server und erwartet Informationen zum Ändern der aktuell angezeigten UI |
| CHECK_WITH_UI_CHANGES | Überprüft den Zustand einer aktiven UI-Komponente auf einen bestimmten Wert und führt daraufhin Änderungen an der aktuellen UI aus |
| UI_CHANGES | Führt Änderungen an der aktuellen UI durch |
| TRIGGER_MODAL | Öffnet ein Modal, welches bereits Teil der aktiven UI ist |
| TRIGGER_ALERT | Öffnet einen Alert, welcher bereits der Teil der aktiven UI ist |

Tabelle 5.7.: Übersicht verfügbare Typen von Aktionen

Zunächst stehen verschiedene Aktionen in Kombination mit Serveranfragen zur Verfügung. Hierbei wird zunächst unterschieden, ob als Antwort eine neue View und eine damit verbundene Navigation innerhalb der Anwendung, oder Änderungen an der aktuell angezeigten UI erwartet werden. Letzteres beinhaltet beispielsweise das Ändern eines angezeigten Texts oder Bilds. Dazu besteht die Option, Informationen der aktuell dargestellten UI-Komponenten, als Teil einer Anfrage, an den Server zu übermitteln. Dies kann beispielsweise die Eingabe aus einem TextInput oder die aktuelle Einstellung eines Toggle sein. Bei den weiteren Aktionstypen handelt es sich um ausschließlich clientseitige Aktionen, wie das Überprüfen von aktuellen Komponenten, beispielsweise, ob ein Textfeld eine spezifische Eingabe enthält oder das nachträgliche Anpassen von UI-Komponenten. Schlussendlich werden Aktionen zum Öffnen von Modalen und Alerts angeboten. Anzumerken ist hierbei, dass diese wie gewöhnliche UI-Komponenten mit der Komponenten-Hierarchie abgebildet, allerdings erst durch das Ausführen einer Aktion geöffnet werden.

| **Eigenschaft** | **Beschreibung** | **Typ** |
|---|---|---|
| *type* | Beschreibt den Typ der Aktion | String |
| *destination* | Gibt die Route für eine Serveranfrage an | String |
| *payloadRequirements* | Ein Array von Komponenten, deren aktueller Zustand an den Server übermittelt werden soll | [PayloadRequirement] |
| *checkedFields* | Ein Array, welches Informationen von zu überprüfenden Komponenten beinhaltet | [FieldValue] |
| *fieldChanges* | Ein Array, welches Informationen über zu ändernde Felder beinhaltet | [FieldValue] |

Tabelle 5.8.: Eigenschaften des Typs *Action*

Zur Umsetzung der verschiedenen Aktionstypen werden durch den Client unterschiedliche Informationen vorausgesetzt. Tabelle 5.8 gibt zunächst einen Überblick über die Datenstruktur des Typs *Action*. Eine ausführliche Übersicht zu den Untertypen *PayloadRequirement* und *FieldValue* ist in Anhang A.4 und A.5 zu finden. Ausschließlich bei der Eigenschaft *type* des Typs *Action* handelt es sich um eine Pflichteigenschaft. Alle anderen Eigenschaften sind von dem jeweiligen Typ der Aktion abhängig und müssen bei Bedarf gesetzt werden.

Zum Bereich der Anwendungslogik zählt zudem der letzte zusammengesetzte Datentyp *Validator* aus Tabelle 5.5. Die dazugehörige Eigenschaft dient der Validierung einer UI-Komponente in einen booleschen Zustand. Im Vergleich zur Aktion *CHECK_WITH_UI_CHANGES*, wird die Validierung hier anhand einer REGEX-Zeichenkette oder via einer Netzwerkanfrage durchgeführt. Dies kann zum Beispiel bei der Überprüfung von Nutzereingaben relevant sein. Im Rahmen der prototypischen Umsetzung wurde dies allerdings zunächst nur für die TextInput-Komponente implementiert. Ein Validator besteht immer aus den Eigenschaften *type*, zur Angabe des Typs und der Eigenschaft *value*, welche als Referenz zur Validierung genutzt wird. Im Falle einer REGEX-Validierung enthält die Eigenschaft *value* den zugehörigen REGEX-String und im Falle einer Netzwerkanfrage die zugehörige URL. Anzumerken ist hierbei, dass die Validierung durch eine Netzwerkanfrage nicht in der prototypischen Umsetzung enthalten ist.

Um abschließend die vorgestellten Bereiche der strukturierten Darstellung zusammenzuführen, soll im Folgenden die Umsetzung einer vollständigen View gezeigt werden. Eine exemplarische Darstellung ist hierfür in Abbildung 5.2 dargestellt. Als Grundlage dient das in Abbildung 4.2 dargestellte Login-Szenario. Zusätzlich sollen nun durch die Betätigung des *Check Input*-Buttons die Eingabefelder validiert werden. Für das Feld des Benutzernamens muss mindestens ein Zeichen eingegeben werden. Das eingegebene Passwort muss mindestens acht Zeichen lang sein und mindestens eine Zahl enthalten. Die Validierung soll anhand einer REGEX-Zeichenkette erfolgen. Nur wenn diese Voraussetzungen gegeben sind, soll im Anschluss der Login-Button aktiviert werden. Zur visuellen Darstellung soll der Text am oberen Bildschirmrand fett gedruckt und die Beschriftung des *Check Input*-Buttons blau eingefärbt werden. Die zugehörige View im JSON-Format ist im Anhang 20 zu finden.

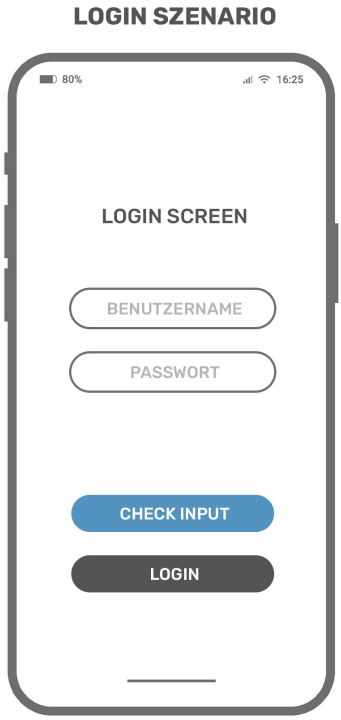



Abbildung 5.2.: Exemplarische Darstellung eines erweiterten Login-Szenarios

Aufbauend auf der beschriebenen strukturierten Darstellung soll nun im weiteren Verlauf die technische Umsetzung des Servers und des Clients betrachtet werden. Die Wahl des JSON-Formats stellt hierbei einen Kompromiss dar und kann sowohl server- als auch clientseitig effizient verarbeitet werden. Des Weiteren erlaubt die leserliche Darstellung von JSON einen guten Arbeitsfluss bei der Generierung von Views. Die Darstellung von UI-Komponenten ist so gewählt, dass lediglich ein Mindestmaß an Eigenschaften gesetzt werden muss, weitere Anpassungen können aber bei Bedarf vorgenommen werden. Dies gilt ebenfalls für die Umsetzung der Modifikatoren.

### 5.1.3. Server

Nachdem ein Format für die strukturierte Darstellung gewählt wurde, soll nun eine mögliche technische Umsetzung des Servers betrachtet werden. Im Rahmen eines Backend-Driven UI-Frameworks stellt der Server eine Hauptkomponente dar, welche den Ablauf der eigentlichen Anwendung steuert. Der Server dient als Endpunkt für den Client und stellt diesem die zur Anwendung gehörenden Views in einem einheitlichen, strukturierten Format bereit. Dazu stellt er Ressourcen wie Bilddateien oder zusätzliche Endpunkte für spezifische Anfragen zur Verfügung. Im Folgenden soll, aufbauend auf der in Abbildung 4.1 dargestellten Architektur, genauer auf die Unterkomponenten des Servers eingegangen werden. Dazu sollen zunächst die genutzten Technologien erläutert werden.

Im Rahmen der technischen Umsetzung des Servers wurde sich für *Node.js*[2] als JavaScript Umgebung und *Express.js*[3] als Web-Framework entschiedenen. Beide Technologien erlauben ein schnelles Setup eines Webservers und ermöglichen das Einbinden von externen Bibliotheken über den Paketmanager *NPM*[4]. Dazu handelt es sich um JavaScript-basierte Webtechnologien, was die Nutzung von JSON vereinfacht. Besonders für eine erste prototypische Umsetzung bietet sich der Einsatz der genannten Technologien an, da es hier zunächst um die Erprobung der in Kapitel 4.2 aufgestellten Architektur geht. Des Weiteren stellen die in Kapitel 4.1.3 beschriebenen serverseitigen Anforderungen keine speziellen Technologieanforderungen dar und können so mit einer beliebigen Backend-Umgebung umgesetzt werden.

Nach Bestimmung der zu nutzenden Technologien soll im Folgenden eine technische Betrachtung der Server-Komponente stattfinden. Die folgende Abbildung 5.3 gibt hierfür zunächst einen genaueren Überblick über die serverseitige Architektur sowie zugehörige Unterkomponenten.

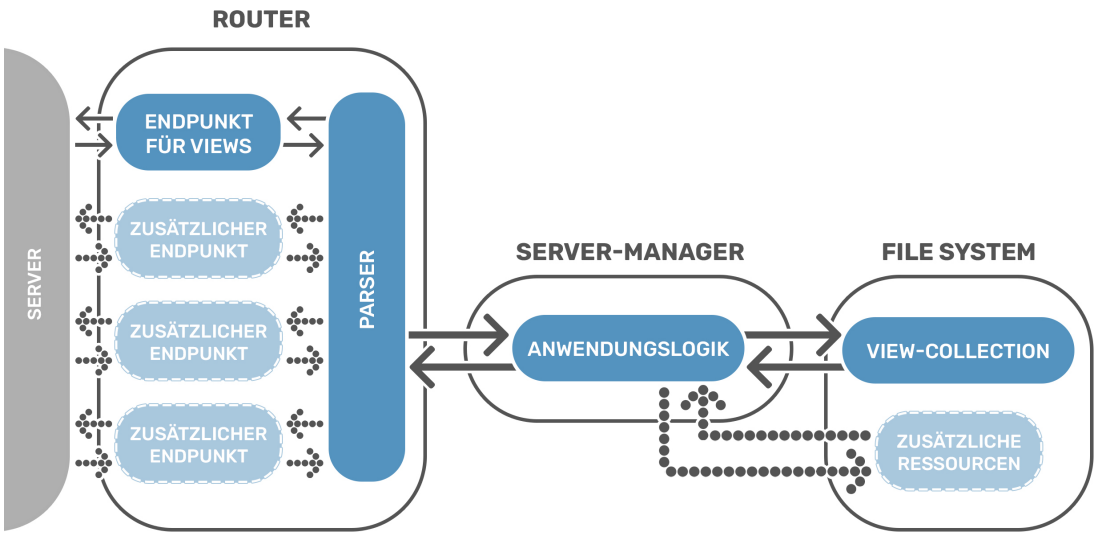



Abbildung 5.3.: Architektur des Servers mit Unterkomponenten

[2] *Node.js: https://nodejs.org/*
[3] *Express.js: https://expressjs.com/*
[4] *NPM*: https://www.npmjs.com/

Den Ausgangspunkt des Servers stellt der Router dar. Dieser bietet verschiedene Endpunkte für clientseitige Anfragen an. Die Anzahl an API-Endpunkten ist dabei variabel und abhängig von der jeweiligen Anwendung. Im Kontext von Backend-Driven UI wird aber mindestens ein Endpunkt zur Anfrage von Views benötigt. Eingehende Anfragen werden an einen Parser weitergeleitet. Dieser muss mögliche Daten aus der Anfrage, beispielsweise Nutzereingaben, extrahieren und für die weitere Nutzung umwandeln. Selbiges gilt für serverseitig aufbereitete Daten, welche an den Client versendet werden sollen. Die eigentliche Anfrage wird durch den Parser an den Server-Manager weitergeleitet. Hierbei handelt es sich nicht um eine konkrete Komponente, sondern vielmehr um eine Sammlung von serverseitiger Anwendungslogik. Diese kann lediglich aus Logik zum Bereitstellen von Views bestehen oder auch komplexere Prozesse wie einen Login abdecken. Im Rahmen der Anwendungslogik kann dann auch auf die View-Collection zugegriffen werden, welche eine Sammlung der einzelnen zur Anwendung gehörenden Views darstellt. Im Rahmen der prototypischen Umsetzung wurden einzelne Views in Form von JSON-Dateien im Dateisystem des Servers abgelegt. Bei Anfrage werden diese dann mithilfe der Bibliothek *fs*[5] eingelesen, in ein natives JavaScript-Objekt überführt und an den Client versendet. Bei einer komplexeren Anwendung mit einer erhöhten Zahl an Views kann es allerdings ratsam sein, die View-Collection in einen externen Service oder Datenbank auszulagern.

```
1  app.get('/view', async (req, res) => {
2      let viewName = req.query.viewName
3      try {
4          let rawData = await fs.promises.readFile(viewName + '.json')
5          let parsedJson = JSON.parse(rawData.toString())
6
7          res.send(parsedJson)
8      } catch (e) {
9          res.status(500)
10         res.send(e)
11     }
12 })
```

Listing 7: API-Endpunkt zur Anfrage einer View mit zugehörigem JSON-Parsing

Listing 7 zeigt eine beispielhafte Implementierung eines API-Endpunktes mit *Express.js* zur Anfrage einer View. Zur Umsetzung wird in Zeile 2 zunächst der Name der angefragten View aus den Query-Parametern extrahiert. Im Anschluss wird dann durch Nutzung der Bibliothek *fs* versucht, die angefragte View, in Form einer JSON Datei, aus dem Dateisystem einzulesen und anschließend in ein JSON-Objekt zu überführen. Im Erfolgsfall wird dies dann als Antwort an den Client gesendet. Sollte es hierbei zu einem Fehlerfall kommen, beispielsweise wenn die gesuchte View nicht im Dateisystem existiert oder die Überführung zu JSON aufgrund eines Fehlers abbricht, wird ein entsprechender Fehlerstatus zurückgeschickt.

Im Rahmen eines Backend-Driven UI Systems stellt der Server einen essenziellen Bestandteil dar. Dieser verwaltet nicht nur die eigentliche Anwendung, sondern muss diese Informationen auch einem Client bereitstellen können. Für die prototypischen

[5] *fs*: https://nodejs.org/api/fs.html

Umsetzung wurde sich für eine schlanke Implementierung der Serverstruktur entschieden, welche durch die gewählten Technologien schnell umsetzbar war. Für einen detaillierten Überblick über die Server-Implementierung wurde der in Listing 7 gezeigte API-Endpunkt mit zugehörigem *Express.js* Setup in Anhang 21 dargestellt.

### 5.1.4. Client

Bei der zuletzt umgesetzten Komponente handelt es sich um den Client, in Form einer nativen Anwendung. Im Rahmen von Backend-Driven UI stellt dieser zunächst die Interaktionsebene zwischen System sowie Nutzer dar und ist für die Darstellung der serverseitig bestimmten UI sowie Anwendungslogik zuständig. Wie bereits erwähnt, wurde sich im Rahmen der prototypischen Umsetzung für die iOS-Plattform entschieden. Zur Umsetzung wurden daher die native Programmiersprache *Swift*, das deklarative UI-Framework *SwiftUI* und die Entwicklungsumgebung *Xcode* verwendet. Im folgenden Kapitel soll nun genauer auf die technische Umsetzung des Clients eingegangen werden. Die folgende Abbildung 5.4 zeigt einen Überblick über die Architektur des Clients sowie zugehöriger Unterkomponenten.

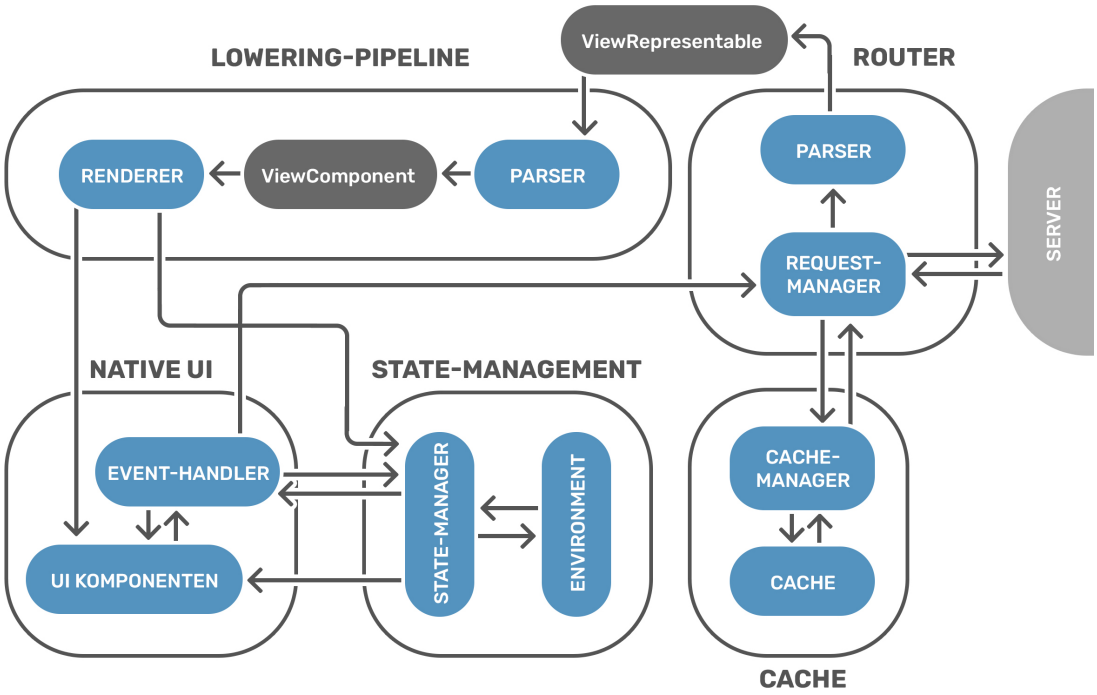



Abbildung 5.4.: Architektur des Clients mit Unterkomponenten

#### Router

Der Client verfügt selbst über eine sehr geringe oder keine native UI. Diese wird erst zur Laufzeit anhand einer vom Server bereitgestellten Datenstruktur generiert und für den Nutzer dargestellt. Ausschlaggebend für den Ablauf der Anwendung ist daher zunächst die Kommunikation mit dem Server. Diese geschieht mithilfe einer Router-Komponente, deren Aufgabe das Senden von Anfragen und das Verarbeiten von Ser-

verantworten ist. Dies ist beispielsweise bei der Anfrage einer View von dem Server notwendig. Das Stellen der Anfrage wird hier durch einen Request-Manager durchgeführt, welcher die durch den Server bereitgestellten Daten empfängt. Zur Umsetzung wurde hier die native Swift Klasse *URLSession* verwendet, welche eine Auswahl an Netzwerkfunktionalitäten bereitstellt. Die durch den Request-Manager empfangenen Daten liegen zunächst innerhalb des JSON-Formats vor und müssen für die weitere Nutzung umgewandelt werden.

```
struct ViewRepresentable: Codable {
    let id: String
    let type: String
    var children: [ViewRepresentable]? = nil
    var modifier: [ModifierRepresentable]? = nil
    var text: String? = nil
    var message: String? = nil
    var imagePath: String? = nil
    var icon: String? = nil
    var rangeStart: Int? = nil
    var rangeEnd: Int? = nil
    var tabViews: [ServerTabView]? = nil
    var action: Action? = nil
    var validator: Validator? = nil
    var isEnabled: Bool? = nil
}
```

Listing 8: Aufbau der Klasse *ViewRepresentable*

Zur Überführung einer JSON-Struktur in ein natives Objekt, welches innerhalb des Programmcodes weiterverwendet werden kann, findet bereits innerhalb des Routers ein Parsing-Prozess statt. Hierfür stehen auf der iOS-Plattform, wie in Kapitel 2.2 dargestellt, verschiedene Technologien zur Verfügung. Im Rahmen der prototypischen Umsetzung wurde sich für die Nutzung von *JSONDecoder* entschieden. Grund hierfür ist die einfache Überführung einer JSON-Struktur in eine native Klassenstruktur, durch die Nutzung der *Codable*[6] Schnittstelle. Innerhalb der Programmiersprache Swift existieren verschiedene Konstrukte zum Abbilden einer Klassenstruktur. Im Rahmen der prototypischen Umsetzung werden die Konstrukte *struct* und *class* verwendet. Zur allgemeineren Beschreibung wird allerdings zur weiteren Erläuterung des Entwicklungsprozesses der deckungsgleiche Begriff *Klasse* verwendet. Bei der Schnittstelle Codable handelt es sich um ein Protokoll, welches es Klassen ermöglicht, sich selbst aus einer externen Repräsentation zu dekodieren. Ein Protokoll stellt in Swift, abgebildet durch das Schlüsselwort *protocol*, einen deckungsgleichen Funktionsumfang zu einem Interface dar. Zur besseren Verallgemeinerung wird daher im Folgenden der Begriff *Interface* verwendet. Konkret bedeutet dies, dass eine JSON-Struktur mithilfe des *JSONDecoder* automatisiert in ein Objekt einer gleich aufgebauten Klasse überführt werden kann. Die gewünschte Klasse muss hierfür das Interface *Codable* implementieren und sämtliche Felder der JSON-Struktur namensgleich durch Eigenschaften abdecken können. Bei den serverbezogenen Daten handelt es sich um die Repräsentation einer View. Zur

[6] *Codable*: https://developer.apple.com/documentation/swift/codable

Überführung ist daher zunächst eine Klasse zum Abdecken dieser Repräsentation notwendig. Der Aufbau einer View besteht, wie in Kapitel 5.1.2 beschrieben, aus einer einzigen UI-Komponente, welche weitere UI-Komponenten in sich verschachtelt. Zur Abdeckung dieser Struktur wurde die im folgenden Listing 8 dargestellte Klasse *ViewRepresentable* verwendet.

Da innerhalb einer View nicht festgelegt ist, welche UI-Komponenten ineinander verschachtelt werden, muss die clientseitige Repräsentation zunächst möglich generisch sein, um alle unterstützten Komponenten abdecken zu können. Die Nutzung des Interfaces *Codable* erlaubt es, optionale Eigenschaften mit *nil* zu initialisieren. Daher werden zum Initialisieren eines Objekts der Klasse lediglich die Eigenschaften *id* und *type* vorausgesetzt. Dies orientiert sich an der Umsetzung der in Kapitel 5.1.2 beschriebenen Datenstruktur. Die Eigenschaft *children* stellt zur Umsetzung der Verschachtelung ein Array des Typs *ViewRepresentable* dar. Für die Umsetzung der Modifikatoren wird eine generische Klasse *ModifierRepresentable* verwendet, welche ebenfalls durch die Nutzung von *nil* alle optionalen Eigenschaften abdeckt. Eine Darstellung der zugehörigen Klassenstruktur ist in Anhang 22 zu finden. Die Klassenstruktur in Listing 8 kann nun im Rahmen des Routers zur automatischen Überführung der JSON-Daten verwendet werden.

Listing 9 und 10 sollen noch einmal eine Gegenüberstellung einer bereits in ein *ViewRepresentable* überführten View und der zugehörigen JSON-Struktur zeigen. So ist es auch in einer überführten Version möglich, verschiedene UI-Komponenten miteinander zu verschachteln. Zusätzlich zeigt Listing 9 in Zeile 5, wie die Verwendung der generischen Klasse *ModifierRepresentable* zur Überführung von Modifikatoren genutzt werden kann.

Auf Grundlage einer in eine *ViewRepresentable* überführte View wäre es zu diesem Zeitpunkt schon möglich native UI zu generieren. Durch die Anwendung von Pattern-Matching kann bereits hier anhand der Eigenschaft *type* zu einer *ViewRepresentable* eine zugehörige SwiftUI-Komponente initialisiert werden. Diese Umsetzung wäre allerdings nur schwer wartbar und erweiterbar. Dazu wären im Rahmen der UI-Initialisierung, aufgrund der Anzahl an optionalen Eigenschaften in *ViewRepresentable*, eine Vielzahl von *Nullability*-Checks notwendig. Diese Zahl würde sich zudem weiter erhöhen, insofern weitere UI-Komponenten im Rahmen des Frameworks unterstützt werden sollten, da die Klasse *ViewRepresentable* in diesem Fall um weitere optionale Eigenschaft ergänzt werden muss.

```
1  ViewRepresentable(id: "c1", type: "COLUMN", children: [
2          ViewRepresentable(id: "t1", type: "TEXT", text: "Lorem Ipsum"),
3          ViewRepresentable(id: "r1", type: "ROW", children: [
4              ViewRepresentable(id: "t2", type: "TEXT", modifier: [
5                  ModifierRepresentable(type: "FONTSTYLE", fontStyle: "BOLD")
6              ], text: "dolor sit amet"),
7              ViewRepresentable(id: "im1", type: "IMAGE", imagePath: "src/data/image1")
8          ])
9      ])
10 }
```

Listing 9: In *ViewRepresentable* überführte View

```json
{
    "id": "c1",
    "type": "Column",
    "children": [
        {
            "id": "t1",
            "type": "TEXT",
            "text": "Lorem Ipsum"
        },
        {
            "id": "r1",
            "type": "ROW",
            "children": [
                {
                    "id": "t2",
                    "type": "TEXT",
                    "text": "dolor sit amet",
                    "modifier": [
                        {
                            "type": "FONTSTYLE",
                            "fontStyle": "BOLD"
                        }
                    ]
                },
                {
                    "id": "im1",
                    "type": "IMAGE",
                    "imagePath": "src/data/image1"
                }
            ]
        }
    ]
}
```

Listing 10: View in JSON-Format

### Lowering-Pipeline

Um die Wart- und Erweiterbarkeit des Frameworks zu steigern, soll eine Lowering-Pipeline eingesetzt werden. *Lowering* bezeichnet einen Prozess, bei dem ein generisches Objekt in ein spezifischeres Objekt überführt wird. Ziel ist es daher, ein Objekt der generischen Klasse *ViewRepresentable*, welches aktuell diverse UI-Komponenten beschreiben kann, in eine konkretere UI-Komponente zu überführen. Dafür soll für jede unterstützte UI-Komponente eine separate Klassen-Struktur angelegt werden.

Um dennoch Gemeinsamkeiten einer UI-Komponente für alle konkreten Unter-Komponenten bereitzustellen, wird das Strategie-Muster eingesetzt. Hierbei handelt es sich um ein Softwareentwurfsmuster, welches die Definition einer austauschbaren Familie an Algorithmen beschreibt. Die konkrete Umsetzung sieht zunächst die Implementierung einer abstrakten Strategie vor. Diese gibt ein Verhalten in Form von Eigenschaften und Funktionen vor, welche dann von Klassen, auch konkrete Strategien genannt, übernommen werden kann. Zur Verdeutlichung ist der Aufbau des Strategie-Musters in der nachfolgenden Abbildung 5.5 anhand eines UML-Diagramms veranschaulicht.

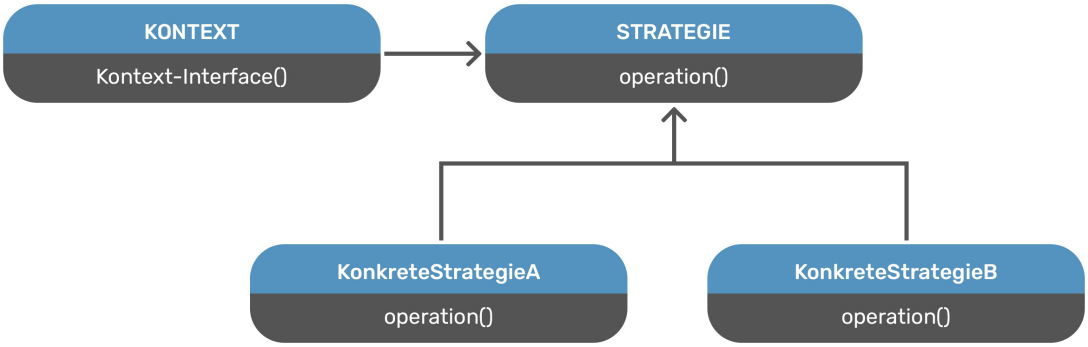



Abbildung 5.5.: UML-Diagramm des Strategie-Entwurfsmusters nach IONOS[7]

Die Implementierung der abstrakten Strategie erfolgt in Form eines Interfaces. Im Rahmen der prototypischen Umsetzung soll diese zunächst die benötigten Gemeinsamkeiten aller UI-Komponenten abbilden. Dafür wurde das in Listing 11 dargestellte Interface implementiert.

```
1 protocol ViewComponent {
2     var id: String { get }
3     var permittedModifier: [String] { get }
4     var modifier: [ModifierComponent] { get }
5 
6     func generateBaseView() -> AnyView
7 }
```

Listing 11: Aufbau des Interfaces *ViewComponent*

Das Interface *ViewComponent* gibt zunächst die Eigenschaften *id* und *modifier* vor. Wie bereits im Rahmen der strukturierten Darstellung wird ein eindeutiger Identifier für das nachträgliche Zugreifen und Referenzieren von UI-Komponenten benötigt. Die Eigenschaft *modifier* stellt hier nun ein Array vom Typ *ModifierComponent* dar. Hierbei handelt es sich ebenfalls um die Anwendung des Strategie-Musters. Es werden zunächst Gemeinsamkeiten von Modifikatoren innerhalb einer abstrakten Strategie, in Form eines Interfaces zusammengefasst. Konkrete Implementierungen des Interfaces decken dann verfügbare Modifikatoren ab. Der Aufbau des Interfaces *ModifierComponent* ist in Anhang 23 abgebildet. Durch das Konzept der Polymorphie ist es zudem möglich, beliebige Implementierungen des Interfaces *ModifierComponent* in der Eigenschaft *modifier* zusammenzufassen. Das dargestellte Interface gibt zudem noch die Eigenschaft *permittedModifier* vor. Hierbei handelt es sich um ein Array vom Typ *String*, welches eine Sammlung aller erlaubten Modifikatoren für die jeweilige UI-Komponente enthält. Auf Grundlage dieser können dann während des Render-Prozesses alle unerlaubten Modifikatoren herausgefiltert werden. Dies stellt einen zusätzlichen Sicherheitsmechanismus für den Fall einer falschen serverseitigen Zuweisung dar. Zuletzt gibt das Interface *ViewComponent* die Funktion *generateBaseView* vor. Jede konkrete Stra-

[7] https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-das-strategy-pattern/

tegie ist dadurch selbst verantwortlich, ihre individuelle SwiftUI-Komponente für den Render-Prozess bereitzustellen.

Die in Listing 12 gezeigte Klasse stellt eine konkrete Implementierung der Strategie *ViewComponent* dar und bildet eine Text-Komponente ab. Wichtig ist hier zunächst die Bereitstellung einer SwiftUI-Komponente für den weiteren Render-Prozess. Hierfür wird die interne Klasse *_TextComponent* verwendet, welche innerhalb der Eigenschaft *body* die zu rendernde SwiftUI-Komponente darstellt und durch die Funktion *generateBaseView()* zurückgeben werden kann. Im Rahmen der in Kapitel 4.1.4 aufgestellten Anforderungen, wurde das Verwalten des Zustands einer UI-Komponente vorausgesetzt. Um dies umzusetzen, besitzt jede interne Klasse eine Referenz auf ihre Oberklasse, hier in Zeile 19 dargestellt. Innerhalb der Oberklasse können dann, durch das Schlüsselwort *@Published* gekennzeichnet, persistierte Eigenschaften initialisiert werden. Die eigentliche SwiftUI-Komponente, hier in Form von *Text()* in Zeile 22 dargestellt, kann diese dann als Datenbasis referenzieren. Diese in sich geschlossene Umsetzung der MVVM-Architektur führt zu einer losen Kopplung zwischen Daten und UI-Komponenten. Dies erlaubt es zum einen, die Datenbasis nachträglich, über das Referenzieren der Klasse *TextComponent* anzupassen und zum anderen die UI automatisch zu aktualisieren, sofern sich die Datenbasis ändert. Die hier dargestellte Umsetzung der Text-Komponente kann so für alle unterstützten UI-Komponenten und Modifikatoren übernommen werden.

```
1  class TextComponent: ViewComponent, ObservableObject {
2      var id: String
3      var permittedModifier: [String] = ["FOREGROUND_COLOR", "BACKGROUND_COLOR",
4          "FONTSIZE", "FONTSTYLE", "PADDING", "BORDER", "SHADOW"]
5      var modifier: [ModifierComponent] = []
6      @Published var text = ""
7
8      init(id: String, modifier: [ModifierComponent], text: String) {
9          self.id = id
10         self.modifier = modifier
11         self.text = text
12     }
13
14     func generateBaseView() -> AnyView {
15         return AnyView(_TextComponent(textComponent: self))
16     }
17
18     struct _TextComponent: View {
19         @ObservedObject var textComponent: TextComponent
20
21         var body: some View {
22             Text(textComponent.text)
23         }
24     }
25 }
```

Listing 12: Aufbau der Klasse *TextComponent*

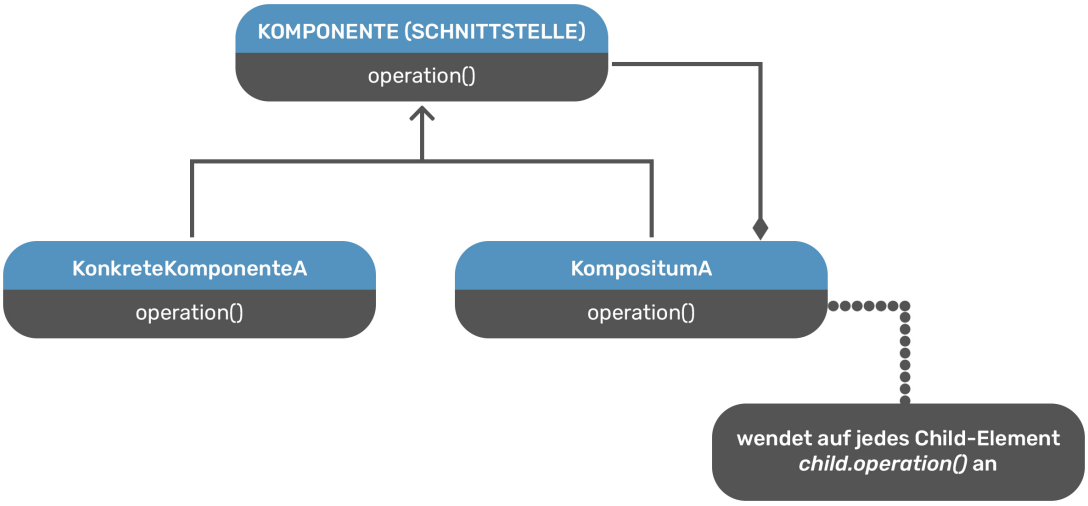



Abbildung 5.6.: UML-Diagramm des Kompositum-Entwurfsmusters nach IONOS[8]

Eine weitere Besonderheit stellt die Umsetzung der Container-Komponenten *Row*, *Column*, *Box* und *List* dar. Zur Verschachtelung von Komponenten ineinander wurde das Kompositum-Muster angewendet, welches in Abbildung 5.6 dargestellt ist. Dieses beschreibt die Implementierung einer konkreten Strategie, welche zusätzlich eine Liste von Strategien beinhaltet und diese mit zusätzlicher Anwendungslogik anreichert. Die umgesetzten Container-Komponenten nutzen hierfür die Eigenschaft *children*, welche ein Array des Typs *ViewComponent* darstellt. Exemplarisch ist der Aufbau der Klasse *RowComponent* in Anhang 24 dargestellt. Während des Render-Prozesses werden die verschachtelten Komponenten dann anhand des durch den Container vorgegeben Layouts gerendet. Da die Container-Komponenten selbst auch wieder Implementierung der Strategie *ViewComponent* sind, können diese ebenfalls beliebig in andere Komponenten verschachtelt werden.

```
1  struct ForegroundColorModifier: ModifierComponent {
2      var id: String = UUID().uuidString
3      var type: String = "FOREGROUND_COLOR"
4      let color: String
5
6      func render(view: AnyView) -> AnyView {
7          return AnyView(view.foregroundColor(getColorFromString(color: color)))
8      }
9  }
```

Listing 13: Aufbau der Klasse *ForegroundColorModifier*

Für die Umsetzung der verschiedenen Modifikatoren, werden ebenfalls konkrete Implementierungen der Strategie *ModiferComponent* verwendet. Listing 13 zeigt exemplarisch die Umsetzung der Klasse *ForegroundColorModifier*. Hierbei wird allerdings auf die Nutzung einer internen Klasse verzichtet. Vielmehr werden Modifikatoren nach

[8] https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-das-composite-pattern/

dem Dekorierer-Muster auf eine UI-Komponente angewendet. Das Dekorierer-Muster beschreibt eine Alternative zum Erweitern einer Klasse ohne die Verwendung von Unterklassen. Dabei wird durch eine Klasse, welche den Dekorierer darstellt zunächst das, im Rahmen des Strategie-Musters definierte Interface implementiert. Zusätzlich nimmt der Dekorierer, eine konkrete Implementierung der Strategie als Parameter entgegen. Die so übergebene Strategie stellt hier die zu erweiternde Klasse dar. Der Dekorierer selbst bleibt allerdings auch eine Implementierung einer Strategie und kann weiterhin polymorph eingesetzt werden. Die folgende Abbildung 5.7 zeigt den UML-Aufbau des Dekorierer-Musters.

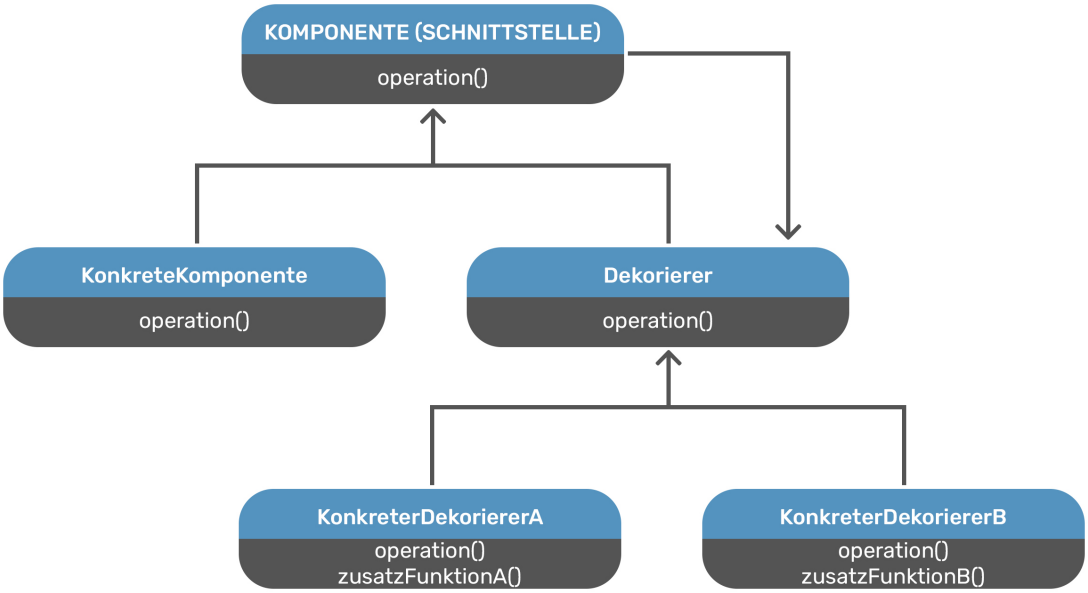



Abbildung 5.7.: UML-Diagramm des Dekorierer-Entwurfsmusters nach IONOS[9]

Die eigentliche Anwendung eines Modifikators auf eine UI-Komponente geschieht durch die Funktion *render(view: AnyView)*, welche in Zeile 6 in Listing 13 dargestellt ist. Daher wird das Dekorierer-Muster nicht auf Klassen-, sondern auf Funktionsebene angewendet. Die Anwendung des Modifikators wird während des Render-Prozesses durchgeführt. Die Funktion *render(view: AnyView)* nimmt als Parameter zunächst die jeweilige SwiftUI-Komponente entgegen, welche durch die Funktion *generateBaseView()* bereitgestellt wird. Da die grundlegende Komponente zunächst irrelevant für das Anwenden des Modifikators ist, wurde für den Parameter *view* der generische Obertyp *AnyView* verwendet. Dies gilt ebenfalls für den Rückgabetyp der Funktion, da der genaue Typ der Komponente ab diesem Zeitpunkt für den weiteren Render-Prozess irrelevant ist. Zusammenfassend besteht der Render-Prozess einer UI-Komponente daher zunächst aus dem Generieren einer SwiftUI-Komponente. Auf diese werden dann die vorgesehenen Modifikatoren angewendet. Anschließend kann die angepasste Komponente als Teil der nativen UI dargestellt werden. Der gesamte Render-Prozess ist zudem in Abbildung 5.8 dargestellt.

[9]https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-das-decorator-pattern/

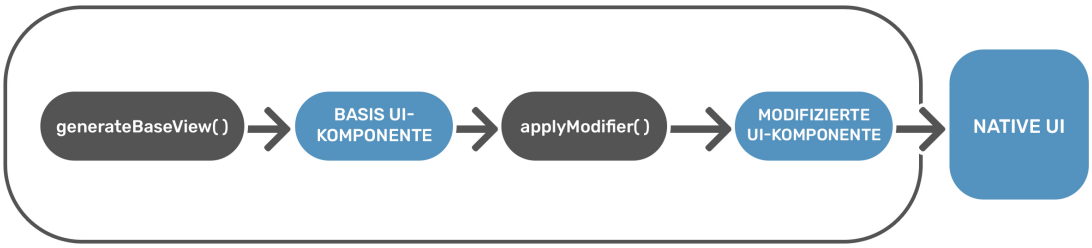



Abbildung 5.8.: Überblick über Render-Prozess einer UI-Komponente

Zuletzt ist die Überführung der generischen UI-Komponenten und Modifikatoren in konkrete Komponenten ebenfalls Teil der Lowering-Pipeline. Ausschlaggebend hierfür ist die Eigenschaft *type* der Klasse *ViewRepresentable*, anhand derer dann die gewünschte Komponente über ein Pattern-Matching initialisiert wird. Das folgende Listing 14 zeigt hierfür einen Ausschnitt der implementierten Funktion *convertViewRepresentable()*.

```
1  func convertViewRepresentable(viewRepresentable: ViewRepresentable) -> ViewComponent {
2      switch viewRepresentable.type {
3          case "TEXT" where viewRepresentable.text != nil:
4              return TextComponent(
5                  id: viewRepresentable.id,
6                  modifier: convertModifierList(viewRepresentable.modifier),
7                  text: viewRepresentable.text!
8              )
9          case "ROW" where viewRepresentable.children != nil: return RowComponent(
10             id: viewRepresentable.id,
11             modifier: convertModifierList(viewRepresentable.modifier),
12             children: viewRepresentable.children!.map(convertViewRepresentable)
13         )
14         default: return EmptyComponent(id: UUID().uuidString, modifier: [])
15     }
16 }
```

Listing 14: Pattern-Matching zur Initialisierung von konkreten UI-Komponenten

Die Funktion *convertViewRepresentable()* nimmt zunächst ein Objekt des Typs *ViewRepresentable* entgegen. Anhand des Typs wird dann über die Implementierung eines Switch-Cases ein Pattern-Matching durchgeführt. Insofern es sich um eine valide Typangabe handelt, kann eine konkrete Komponente initialisiert werden. Da die Klasse *ViewRepresentable* zunächst generisch zum Abbilden aller unterstützten Komponenten genutzt wird, werden die Eigenschaften hier durch *null*-fähige Typen abgedeckt. Daher muss zunächst überprüft werden, ob die benötigten Werte erfolgreich gesetzt wurden. Dies ist in exemplarisch in Listing 14 in Zeile 3 für die Eigenschaft *text* dargestellt. Insofern die benötigten Werte vorhanden sind, kann die gewünschte Komponente initialisiert werden. Sollte es während des Pattern-Matching zu einem Fehler kommen, weil beispielsweise eine Typangabe fehlerhaft oder benötigte Werte nicht gesetzt wurden, wird, wie in Zeile 9 dargestellt, auf eine *EmptyComponent* zurückgegriffen. Hierbei handelt es sich um eine leere Komponente, welche dann während des späteren Render-Prozesses übersprungen wird. Für die Überführung von Container-Komponenten wird

die Funktion *convertViewRepresentable()* zusätzliche rekursiv für alle Unterkomponenten aufgerufen (siehe Zeile 12). Die Überführung von Modifikatoren wird ebenfalls nach demselben Konzept durchgeführt.

### Native UI und State-Management

Wie auch in Abbildung 5.4 dargestellt, folgt im Anschluss an die Lowering-Pipeline der Bereich der nativen UI. Diese bildet die Interaktionsebene zwischen dem Nutzer und dem System und stellt die nativen UI-Komponenten dar. Zusätzlich werden auch zugehörige Event-Handler für beispielsweise Button-Komponenten initialisiert. Innerhalb des Frameworks wird das Konzept von deklarativer UI angewendet. Wie bereits in Kapitel 2.3 beschrieben, erfordert dies eine einheitliche Datenbasis, welche durch die UI dargestellt wird und als Single-Source-of-Truth gilt (Marchenko, 2023).

Zur Umsetzung dieser Datenbasis und zur Verwaltung des aktuellen UI-Zustands wird eine zusätzliche State-Management-Komponente eingesetzt. Im Rahmen von mobilen Anwendungen ist der Zustand einer UI-Komponente relevant für den Ablauf der Anwendung. Dazu kann der Zustand durch Anwendungslogik oder auch Nutzereingaben während der Laufzeit verändert werden. Innerhalb des Frameworks ist jede Komponente, wie bereits in Listing 12 dargestellt, für die Verwaltung des eigenen Zustands verantwortlich. Da UI-Komponenten zur Laufzeit allerdings als Teil einer Verschachtelung dargestellt werden, ist ein nachträglicher Zugriff auf diesen Zustand nicht möglich. Das State-Management soll diese Hürde überwinden, indem es eine Sammlung von Referenzen auf die aktuell dargestellten UI-Komponenten enthält. Während des Render-Prozesses wird daher die Referenz der aktuell zu rendernden Komponente an den State-Manager weitergeleitet. Dieser legt sie dann innerhalb des *Environments* ab. Sollten nun Änderungen an der UI durchgeführt werden müssen, beispielsweise als Reaktion auf eine Nutzereingabe oder eine Serveranfrage, werden diese durch den State-Manager vorgenommen und automatisch an die aktiven UI-Komponenten weitergeleitet, welche sich daraufhin dynamisch aktualisieren können.

```
1  class Environment {
2      private var env = [ViewComponent]()
3      private init() {}
4
5      static let instance = Environment()
6
7      func add(_ vc: ViewComponent) {
8          if !env.contains(where: { $0.id == vc.id }) {
9              self.env.append(vc)
10         }
11     }
12
13     func clear() {
14         self.env.removeAll()
15     }
16 }
```

Listing 15: Aufbau der Klasse *Environment*

Listing 15 zeigt die prototypische Umsetzung des State-Managements in Form der Klasse *Environment*. Zur Verwaltung der Referenzen wird ein Array vom Typ *ViewComponent* eingesetzt. Wie bereits im Rahmen der Modifikatoren und der Eigenschaft *children* können hier aufgrund von Polymorphie sämtliche konkreten Implementierungen der Strategie *ViewComponent* eingefügt werden. Für die einheitliche Datenhaltung und den nachträglichen Zugriff wird die statische Eigenschaft *env* eingesetzt, welche eine Initialisierung der Klasse *Environment* enthält und eine Nutzung als Singleton ermöglicht.

### Cache

Bei der letzten in Abbildung 5.4 dargestellten Komponente handelt es sich um den Cache. Diese stellt keine konkrete Umsetzung einer funktionalen Anforderung dar, sondern soll viel mehr dazu beitragen, die in Kapitel 4.1.5 aufgestellten qualitativen Anforderungen zu erfüllen. Ziel ist es hierbei, die Zahl an Serveranfragen zu minimieren, indem eingegangene Serverantworten gespeichert und bei Bedarf direkt aus dem Cache geladen werden können. Dies erlaubt ebenfalls die Nutzung der Anwendung, insofern keine stabile Netzwerkverbindung gewährleistet werden kann und diese bereits einmal erfolgreich angefordert wurden. Im Rahmen der prototypischen Umsetzung wurde sich für eine rudimentäre Implementierung des Caches entschieden, wobei die Validität lokal gespeicherter Views anhand ihres Alters bestimmt wird. Eine weitere und robustere Lösung würde die Validierung anhand einer Versionsnummer vorsehen. Um zu überprüfen, ob eine aktuellere Version einer View serverseitig verfügbar ist, müsste daher zunächst eine Anfrage zur Überprüfung der Version an den Server gestellt werden. Sofern keine neuere Version aufseiten des Servers zur Verfügung steht, kann der Client auf die gespeicherte Version zurückgreifen. Im Rahmen der prototypischen Umsetzung wurde sich allerdings gegen eine umfangreichere Lösung entschieden. Dazu würde dies eine zusätzliche Anfrage an den Server voraussetzen. Der Ablauf des implementierten Caching-Prozesses ist in der folgenden Abbildung 5.9 dargestellt.

Der Caching-Prozess wird durch die Anfrage einer View durch den Client gestartet. Zunächst wird geprüft, ob eine Version der angefragten View innerhalb des Caches vorhanden ist. Sofern dies nicht zutrifft, soll im Falle einer bestehenden Internetverbindung eine aktuelle Version der View angefragt werden. Ist auch dies nicht möglich, kann keine Darstellung erfolgen. Sofern allerdings eine View innerhalb des Cache verfügbar ist, wird zunächst das Alter der View überprüft. Alter bezeichnet hier die vergangene Zeit seit dem Speichern. Dieses Alter wird mit einer clientseitig bestimmten *Time-To-Life* (TTL) abgeglichen. Sofern diese überschritten wurde, wird im Falle einer aktiven Internetverbindung eine aktuelle Version der View angefragt. Sofern dies nicht möglich ist, wird die Version innerhalb des Caches verwendet. Dies gilt ebenfalls, sobald das Alter der gecachten Version noch gültig ist. Anzumerken ist hierbei, dass Views bereits vor dem Cachen in eine *ViewRepresentable* überführt und so bei Bedarf direkt an die Lowering-Pipeline weitergegeben werden können. Die Persistierung von gecachten Views wird über ein Array abgebildet, welches innerhalb der *UserDefaults*[10] abgelegt wird. Hierbei handelt es sich um eine Datenbank, zum Speichern von Key-Value Paaren, welche durch die iOS-Plattform zur Verfügung gestellt wird.

[10] *UserDefaults*: https://developer.apple.com/documentation/foundation/userdefaults

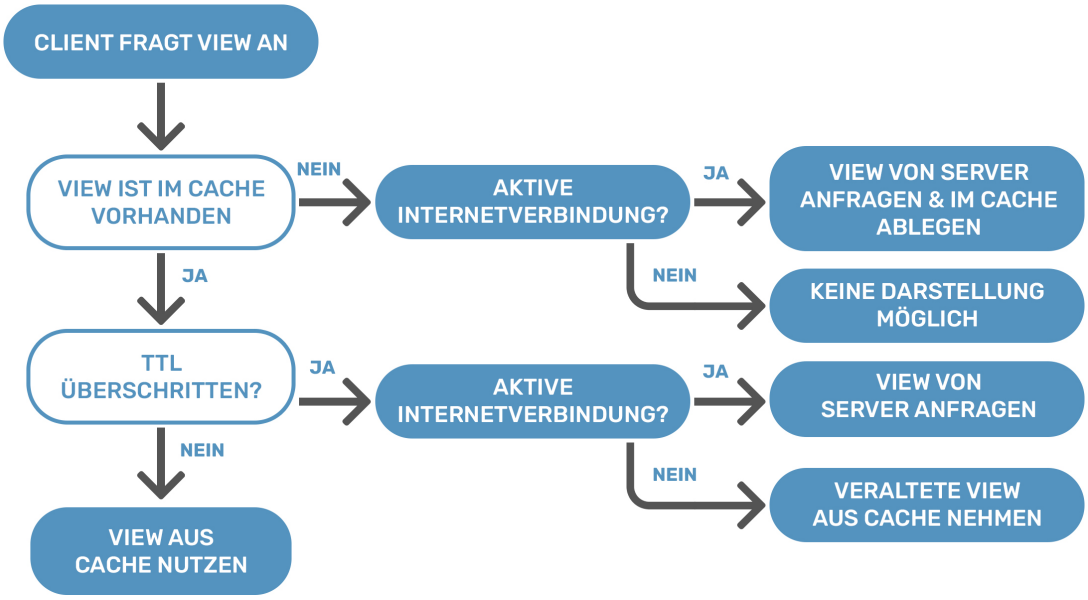



Abbildung 5.9.: Überblick über Caching-Prozes

## 5.2. Herausforderungen

Bei der in Kapitel 5.1 vorgestellten Umsetzung wurde ein erster Prototyp eines Backend-Driven UI Frameworks zur Umsetzung von Cross-Plattform-Anwendungen entwickelt. Während des Entwicklungsprozesses traten dennoch diverse Herausforderungen auf, welche im Folgenden erläutert werden sollen.

Zunächst folgte die Definition einer strukturierten Darstellung der Anwendung. Bereits durch die Anforderungen an das Framework ist klar, dass eine Vielzahl an unterschiedlichen Komponenten abgebildet werden muss. Jede dieser Komponenten besitzt individuelle Anforderungen und setzt unterschiedliche Informationen voraus, um clientseitig in eine native UI-Komponente überführt werden zu können. So benötigt beispielsweise eine Image-Komponente eine Angabe der zu nutzenden Bild-Ressource, während eine Text-Komponente eine darzustellende Zeichenkette benötigt. Um die Abbildung von Views so zugänglich wie möglich zu halten, wurde sich für eine offene JSON-Struktur entschieden. Resultierend daraus ist allerdings aufseiten des Clients eine erhöhte Zahl an Überprüfungen und Validierungen nötig, um die im JSON-Format vorliegende View in native UI zu überführen. Diese Abwägung stellt so eine Herausforderung während des Entwicklungsprozesses dar, da entschieden werden muss, inwiefern die Komplexität zur Abbildung von Komponenten, zwischen Client und der strukturierten Darstellung verteilt wird. Besonders um die Zugänglichkeit der strukturierten Darstellung zu wahren und Entwicklern eine schnelle Umsetzung von Views zu ermöglichen, wurde die Komplexität hier stark auf den Client verlagert.

Eine weitere Herausforderung, welche sowohl die strukturierte Darstellung als auch den Client betrifft, war die Abbildung von serverseitiger Anwendungslogik. Da im Rahmen der Grundlagen und der verwandten Arbeiten keine Wissensbasis geschaffen wer-

den konnte, musste abgewägt werden, zu welchem Grad die Umsetzung erfolgen soll. Fokus der prototypischen Umsetzung sollte zunächst die Abbildung einer möglichst reichhaltigen UI sein. Daher wird die Darstellung von Anwendungslogik zunächst nur in einem beschränktem Rahmen unterstützt. Dazu wurden nur grundlegende Anwendungsszenarien, wie beispielsweise das nachträgliche Anpassen von UI-Komponenten oder Senden von Anfragen abgedeckt. Bereits ersteres erlaubt es allerdings, grundlegende Logik umzusetzen. So ist, wie im Rahmen der Login-View in Abbildung 5.2 dargestellt, möglich einzelne Komponenten nachträglich dynamisch anzupassen. Grundlage hierfür sind allerdings zunächst nur konditionale Aussagen, indem der Zustand einer UI-Komponente mit einem vorgegebenen Wert verglichen wird.

## 5.3. Betrachtung der Android-Plattform

Nachdem nun bei der prototypischen Umsetzung aufgetretene Herausforderungen erläutert wurden, soll zum Abschluss die Android-Plattform betrachtet werden. Ziel ist es zu überprüfen, ob technische Entscheidungen, welche im Rahmen des Clients getroffen und in Kapitel 5.1.4 vorgestellt wurden, innerhalb der Android-Plattform umgesetzt werden können. Der Fokus liegt auf der Abbildung von UI-Komponenten und umfasst sowohl die Überführung einer in JSON-Form abgebildeten View in eine generische Komponente als auch die anschließende Überführung in konkrete UI-Komponenten, welche in Form von nativer UI dargestellt werden können. Andere Funktionen, wie das Anfragen eines Servers, die Implementierung eines State-Managements oder die Nutzung eines Caches sind nicht Teil der Betrachtung. Eine theoretische Umsetzbarkeit ist allerdings auf Grundlage des Funktionsumfangs der Android-Plattform gegeben. Für die Entwicklung eines Android-Clients wurden die in Kapitel 2.1.1 vorgestellten plattformspezifischen Tools verwendet. Die Umsetzung erfolgte mit der IDE *Android Studio* und der Programmiersprache *Kotlin*, in Kombination mit dem deklarativen UI-Framework *Jetpack Compose*.

Im Rahmen der Implementierung wurde zunächst eine generische Klasse zur Überführung der JSON-Struktur angelegt. Diese orientiert sich an der Implementierung der innerhalb des iOS-Clients verwendeten Klasse *ViewRepresentable*. Zum Verarbeiten von JSON wurde die offizielle Bibliothek *GSON* verwendet, welche es ähnlich zu dem Codable-Interface erlaubt, eine JSON-Struktur in ein natives Objekt zu überführen. Das nachfolgende Listing 16 zeigt zunächst die Klassenstruktur der Klasse *ViewRepresentable* sowie die Überführung eines JSON-Strings in ein natives Objekt der Klasse.

Ähnlich zum iOS-Client wird die Klasse *ViewRepresentable* mit diversen optionalen Eigenschaften versehen, um alle möglichen UI-Komponenten abdecken zu können. Lediglich *id* und *type* werden vorausgesetzt. Optionale Eigenschaften werden bei der Initialisierung automatisch mit dem Wert *null* initialisiert. Anhand der Eigenschaft *children* ist ebenfalls erkennbar, dass auch die Verschachtelung von Komponenten umsetzbar ist. Die Nutzung von *GSON* in Zeile 12 ähnelt ebenfalls der des *JSONDecoders* unter Swift. Beide Technologien erwarten jeweils eine Eingabe in Form eines JSON-Strings sowie eine gewünschte Typ-Angabe für die Überführung.

```
class ViewRepresentable(
    val id: String,
    val type: String,
    val text: String?,
    val imagePath: String?,
    val children: Array<ViewRepresentable>?
)

val jsonString: String = "Some JSON...."

val gson = GsonBuilder().create()
val parsedView: ViewRepresentable =
    gson.fromJson(jsonString, ViewRepresentable::class.java)
```

Listing 16: Umsetzung der generischen Klasse *ViewRepresentable* und Nutzung der *GSON* Bibliothek

Zusätzlich zur generischen Klasse *ViewRepresentable* wurden konkrete Klassen zur Umsetzung einer Text-, Image- und Column-Komponente, durch Anwendung des Strategie-Musters umgesetzt. Das folgende Listing 17 zeigt hier das Interface *ViewComponent* sowie exemplarisch die Implementierung der Klasse *TextComponent*. Eine Auflistung der weiteren im Rahmen des Android-Clients implementierten Komponenten ist im Anhang 25 zu finden.

```
interface ViewComponent {
    val id: String

    @Composable
    fun Render()
}

class TextComponent(override val id: String, val text: String): ViewComponent {
    @Composable
    override fun Render() {
       Text(text = text)
    }
}
```

Listing 17: Aufbau des Interface *ViewComponent* und der Klasse *TextComponent* im Rahmen des Android-Clients

Auch hier orientiert sich der Aufbau an dem iOS-Client. Allerdings wurde im Rahmen des Android-Clients kein State-Management oder die Anpassung durch Modifikatoren implementiert. Unterschiedlich ist zudem das Darstellen von nativer UI. Während SwiftUI-Objekte des Typs *View* verwendet, nutzt Jetpack Compose spezielle Composable-Funktionen, welche durch die Annotation *@Composable* gekennzeichnet sind (Marchenko, 2023). Innerhalb dieser Funktion können Compose-Komponenten wie beispielsweise *Text()* in Zeile 11 verwendet werden. Zum Darstellen der Komponente muss dann lediglich die Funktion *Render()* aufgerufen werde, welche durch die Strategie *ViewComponent* vorgegeben wird. Anzumerken ist hier, dass die Funktion

*Render()* großgeschrieben wird, da sie ein konkretes UI-Objekt darstellt, welche nach der Konvention in Jetpack Compose großgeschrieben werden.

Es ist daher festzuhalten, dass die Überführung einer View in JSON-Form über eine generische Komponente hin zu einer konkreten UI-Komponente auch unter Android, durch Einsatz der nativen Tools, möglich ist. Die prototypische Umsetzung deckt dabei allerdings nicht alle Bereiche des iOS-Clients ab. Anhand bestehender Dokumentation der Android-Plattform[11] und der Programmiersprache Kotlin[12] ist allerdings anzunehmen, dass die technischen Rahmenbedingungen für weitere Funktionalitäten, welche im Rahmen des iOS-Clients umgesetzt wurden, gegeben sind. Für eine mögliche Umsetzung des Caches bietet die Android-Plattform die *SharedPreferences*-API[13] an, welche funktionsgleich zu den *UserDefaults* der iOS-Plattform genutzt werden kann.

[11] Dokumentation Android: https://developer.android.com/docs
[12] Dokumentation Kotlin: https://kotlinlang.org/docs/home.html
[13] *SharedPreferences*: https://developer.android.com/reference/android/content/SharedPreferences

# 6. Diskussion

Nach Abschluss der Konzeption und der technischen Umsetzung sollen die Ergebnisse nun rückblickend diskutiert werden. Hierfür erfolgt zunächst die Betrachtung der Anforderungen und der Architektur in Bezug auf die im Rahmen der Grundlagen vermittelte Wissensbasis. Im Anschluss soll geschaut werden, inwiefern diese durch den technischen Prototypen umgesetzt werden konnten. Dazu soll eine Betrachtung möglicher Optimierungen erfolgen. Abschließend soll das im Rahmen dieser Arbeit entwickelte Framework in die Grade von Backend-Driven UI eingeordnet werden, welche in Kapitel 2.1.3 vorgestellt wurden.

### Anforderungen

Das Ziel der Arbeit liegt darin, ein Backend-Driven UI Framework zur Umsetzung einer Cross-Plattform-Anwendung zu entwickeln. Anhand der vorgestellten Grundlagen erfolgte hierfür zunächst das Ableiten von Anforderungen. Bereits durch den Ansatz von Backend-Driven UI konnten die drei Hauptkomponenten *Strukturierte Darstellung*, *Client* und *Server* bestimmt werden, wodurch eine Zuweisung der Anforderungen zu den zugehörigen Komponenten erfolgen konnte. Besonders wichtig sind allerdings die in Kapitel 4.1.5 aufgestellten qualitativen Anforderungen, welche auf Grundlage bekannter Nachteile des Ansatzes aufgestellt wurden. Insgesamt decken die aufgestellten Anforderungen einen breiten Funktionsumfang ab, bieten aber dennoch Möglichkeit zur Erweiterung, um so auch weitere mögliche Erfordernisse erfüllen zu können.

### Architektur

Anhand der aufgestellten Anforderungen wurde dann in Kapitel 4.2 eine Architektur abgeleitet, welche die drei Hauptkomponenten und ihre Interaktionen widerspiegelt. Zusätzlich konnten bereits hier weitere Unterkomponenten zur Abdeckung des Funktionsumfangs bestimmt werden. Es handelt sich bei der Architektur jedoch zunächst um ein Konzept, welches auf Basis der Anforderungen möglichst Technologie-agnostisch aufgestellt wurde. Hier ist allerdings eine Abweichung zu den beschriebenen Komponenten aus Kapitel 2.1.3 feststellbar. So findet im Rahmen des Prototyps die Verwaltung der Anwendung nicht mehr innerhalb des Servers statt, wie ursprünglich in Abbildung 2.1 dargestellt, sondern wird durch die Views selbst und damit durch die strukturierte Darstellung abgebildet. Die Verwaltung bezieht sich auch hier auf den Ablauf der Anwendung und die damit verbundene Navigation innerhalb dieser. Jede View besitzt die Informationen, welche beispielsweise für die weitere Navigation benötigt werden und kann so innerhalb des Clients spezifische Aktionen auslösen.

### Funktionsumfang

Zur Umsetzung des Prototyps wurde zunächst der Funktionsumfang anhand der zu unterstützenden UI-Komponenten skaliert. Im Allgemeinen war es das Ziel, eine individuelle Nutzung von Komponenten zu ermöglichen. Das Framework *Lona*, welches im Rahmen der verwandten Arbeiten in Kapitel 3.1 vorgestellt wurde, erlaubt lediglich die Nutzung von bereits definierten Reihen, welche dann mehrere UI-Komponenten enthalten. Anzumerken ist hier jedoch, dass Lona für einen speziellen Anwendungskontext entwickelt wurde. Das in dieser Arbeit entwickelte Framework soll dennoch eine freie Nutzung der zu unterstützenden Komponenten ohne Abhängigkeiten voneinander ermöglichen.

Für die Bestimmung der UI-Komponenten wurde sich an den plattformspezifischen Design-Guidelines orientiert, wodurch bereits Gemeinsamkeiten und Unterschiede innerhalb der angebotenen Komponenten identifiziert werden konnten. Zudem ließen sich allgemeine Abstraktionen zu Komponenten und Modifikatoren ableiten. Der gewählte Funktionsumfang ist dazu bereits ausreichend, um grundlegende Anwendungsszenarien, wie das in Abbildung 5.2 dargestellte Login-Szenario umzusetzen. Dazu können bereits visuelle Anpassung durch das Konzept von Modifikatoren auf UI-Komponenten angewendet werden. Dennoch erhebt der aktuelle Funktionsumfang keinen Anspruch auf Vollständigkeit und kann bei Bedarf erweitert werden. Einschränkungen hierbei ergeben sich lediglich aus dem bereitgestellten Funktionsumfang der deklarativen UI-Frameworks innerhalb des Clients. Dieser bestimmt ebenfalls die maximale Anpassbarkeit einer UI-Komponente.

Im Rahmen der Anforderungen wurde festgelegt, dass das Framework in der Lage sein muss, unterschiedliche Funktionsumfänge der Plattformen auszugleichen. Dies wurde im Rahmen des Prototyps noch nicht relevant, da die ausgewählten Komponenten erfolgreich über beide Plattformen abstrahiert werden konnten. Insofern der Funktionsumfang des Frameworks allerdings erweitert werden soll, sind Funktionsunterschiede nicht mehr auszuschließen. Hier muss dann abgewägt werden, inwiefern diese Unterschiede ausgeglichen werden sollen. Zunächst besteht die Option, die gesamte Komponente nicht zu unterstützten. Dies würde allerdings der Bereitstellung der einer möglichst nativen Anwendung widersprechen, da so der native Funktionsumfang eingeschränkt wird. Eine weitere Option stellt die einseitige Unterstützung der gewünschten Komponenten auf einer Zielplattform dar. Hierbei würden dann zunächst Anpassungen an der strukturierten Darstellung notwendig sein, da nun keine einheitliche Struktur für alle Zielplattformen verwendet werden kann. Als letzte Möglichkeit kann durch das Framework eine nicht unterstützte Komponente, auf Basis existierender Komponenten, nachgebaut und so zur Nutzung angeboten werden. Dies erfordert allerdings eine individuelle Prüfung der Umsetzbarkeit für jede Komponente.

### Strukturierte Darstellung

Hauptbestandteil der Umsetzung stellt die strukturierte Darstellung der Anwendung dar. Diese dient als einheitliche Sprache zur Darstellung von Views und als Format für den Austausch von Client und Server. Im Rahmen des Prototyps ist es bereits

möglich einen ersten Umfang an UI-Komponenten, Modifikatoren und grundlegender Anwendungslogik abzudecken.

Bei der Bestimmung der Datenstruktur ist es wichtig ein plattformübergreifendes Format zu wählen, weshalb sich für das JSON-Format entschieden wurde. JSON erlaubt den einfachen Austausch zwischen Client und Server und wird auf einer Vielzahl von Plattformen unterstützt. Zudem erlaubt die gewählte Darstellung von Views eine hohe Erweiterbarkeit, da bestehende JSON-Strukturen dynamisch um weitere Eigenschaften erweitert werden können. Die offene Struktur resultiert allerdings auch in einer hohen Fehleranfälligkeit, weshalb bereits serverseitig eine erste Validierung der strukturierten Darstellung erfolgen sollte. Der bisherige Prototyp deckt dies nicht ab und validiert ausschließlich innerhalb des Clients. Sollten hier Inkonsistenzen auftreten, können diese hier lediglich begrenzt, beispielsweise in Form von leeren Komponenten, ausgeglichen werden.

Um den Erstellungsprozess von Views sicherer und einfacherer für Entwickler zu gestalten, sollte eine zusätzliche grafische Oberfläche in Betracht gezogen werden. Dieser Ansatz orientiert sich an dem in Kapitel 3.1 vorgestellten Tool *Lona-Studio*, welches es ermöglicht, UI nach einem Baukasten Prinzip zu gestalten und anschließend in die gewünschte strukturierte Darstellung zu überführen. Die Verwendung einer solchen Oberfläche wurde bereits im Rahmen der Anforderungen aufgeführt, wurde aber innerhalb des Prototyps nicht abgebildet werden.

Zusätzlich kam es bei der Anwendungslogik innerhalb der strukturierten Darstellung zu Einschränkungen. Das JSON-Format unterstützt nur eine grundlegende Zahl an Datentypen. Die bisherige Umsetzung von Anwendungslogik innerhalb einer View beschränkt sich daher auf das nachträgliche Ändern von Komponenten auf Basis von simpler konditionaler Logik. Dabei wird innerhalb der JSON-Struktur bestimmt, welches Feld, anhand eines Referenzwertes, überprüft werden soll und welche Aktion daraufhin ausgeführt werden soll. Tiefergreifende Anwendungslogik, wie beispielsweise das Berechnen von Werten oder Ausführen komplexerer Algorithmen, kann im Rahmen des Prototyps nicht abgebildet werden. Alternativ könnte die Implementierung einer individuellen Syntax oder DSL erfolgen, welche, ähnlich zu beispielsweise einer SQL-Zeichenkette, komplexere Anwendungslogik darstellen kann. Die Verwendung einer eigenen Sprache würde allerdings eine weitere Parser-Ebene auf dem Client voraussetzen, wodurch die Performance der Anwendung weiter gemindert wird. Innerhalb der strukturierten Darstellung müsste dann zusätzlich ein geeignetes Format, beispielsweise in Form einer Zeichenkette, gewählt oder die generelle Nutzung einer DSL in Betracht gezogen werden. Eine weitere Alternative stellt die Nutzung von *Remote-Procedure-Calls* dar, welche das Ausführen einer Prozedur auf einem anderen System ermöglichen. Hierdurch könnten durch den Server konkrete Funktionen auf dem Client angesprochen werden. Allerdings müssen diese dafür bereits auf dem Client verfügbar sein und daher während des Entwicklungsprozesses manuell eingefügt werden. Änderungen würden zudem in einem Update der nativen Anwendung resultieren (Euler, 2005).

### Server

Im Anschluss folgte die Implementierung der Server-Komponente. Auch wenn diese im Kontext von Backend-Driven UI eine tragende Rolle hat, stellt sie im Rahmen der

prototypischen Umsetzung den geringsten Umfang dar. Die Aufgaben des Servers liegen darin, Views in Form der strukturierten Darstellung für den Client und zusätzliche Schnittstellen für weitere Anfragen bereitzustellen. Im Rahmen des Prototyps erfolgte die Umsetzung des Servers mithilfe der Web-Frameworks *Node.js* und *Express.js*. Beide Technologien erlauben es, mit einem geringen Code-Aufwand einen Web-Server zu initialisieren und ein erstes Routing bereitzustellen. Die Umsetzung der in Kapitel 4.1.3 aufgestellten Anforderungen ist allerdings nicht an die Nutzung spezieller Technologien gebunden und kann daher durch die Verwendung beliebiger Frameworks realisiert werden. Die bisherige Implementierung erlaubt es so dem Client dynamisch Informationen bereitzustellen. Dazu können diese Informationen serverseitig angepasst und so Änderungen in der Anwendung direkt an den Client ausgerollt werden. Nicht umgesetzt wurde allerdings die Anforderung in Bezug auf die serverseitige Unterstützung von verschiedenen Client-Versionen. Dies begründet sich durch den prototypischen Rahmen der Umsetzung, sollte aber besonders bei der Verwendung der Anwendung durch echte Nutzer durchgeführt werden.

### Client

Bei der zuletzt umgesetzten Komponente handelt es sich um den Client. Dieser stellt eine native Anwendung dar, dessen Aufgabe daraus bestehen, die durch den Server bereitgestellten Views in strukturierter Form in native UI zu überführen, als Interaktionsebene zwischen System und Nutzer zu dienen und Nutzereingaben zu verarbeiten.

Wie bereits im Rahmen der Architektur festgestellt, handelt es sich bei dem Parser um die Hauptkomponente des Clients, da dieser für die Überführung einer View verantwortlich ist. In einer ersten Iteration wurde die JSON-Struktur zunächst in die generische Klasse *ViewRepresentable* überführt. Eine Klasse, welche alle möglichen Eigenschaften aller unterstützten UI-Komponenten abdeckt. Bereits auf Basis dieser Klasse konnte das Konzept von Backend-Driven UI umgesetzt werden. Um allerdings die Erweiterbarkeit zu steigern und den internen Datenfluss zu verringern, wurde ein zusätzliches Lowering durch die Anwendung des Strategie-Musters umgesetzt. UI-Komponenten werden nun durch konkrete Implementierungen der Strategie *ViewComponent* dargestellt und sind so selbstständig für ihre eigene Datenhaltung sowie den Render-Prozess verantwortlich und haben keine externen Abhängigkeiten. Dies resultiert in einer besonders hohen Erweiterbarkeit, da neue UI-Komponenten nun als selbstständige Klassen implementiert werden können. Es ist allerdings weiterhin notwendig, die Klasse *ViewRepresentable* um weitere Eigenschaften anzupassen, insofern die bereitgestellten Eigenschaften nicht zur Abdeckung der neuen UI-Komponente ausreichen. Grund hierfür ist, dass die Klasse *ViewRepresentable* zu Beginn des Parsing-Prozesses für die automatische Überführung der JSON-Datei genutzt wird.

Ebenfalls musste das im Rahmen der Architektur beschriebene State-Management implementiert werden, um einen nachträglichen Zugriff auf den Zustand einzelner Komponenten zu ermöglichen. Im Rahmen der prototypischen Umsetzung ist dies durch die Klasse *Environment*, in Form eines Singleton-Objekts umgesetzt worden, welches Referenzen zu UI-Komponenten in einem Array speichert. Dies deckt zwar die grundlegenden Anforderungen ab, sollte allerdings im weiteren Verlauf optimiert werden. Hierbei sollte besonders in Betracht gezogen werden, Referenzen auf nicht mehr darge-

stellte UI-Komponenten aus dem Environment zu entfernen, um so die Zugriffszeiten zu optimieren. Zusätzlich sollte das State-Management so erweitert werden, dass eine Persistierung des Zustands über mehrere Nutzer-Sessions möglich ist.

Ergänzend zu der in Kapitel 4.2 aufgestellten Architektur wurde abschließend eine Cache-Komponente zum Persistieren von Server-Anfragen implementiert. Wie bereits in Kapitel 2.1.3 herausgestellt, ist besonders der hohe Datenverbrauch aufgrund umfangreicher Anfragen an den Server, ein Nachteil des Backend-Driven UI Ansatzes. Dies spiegelt sich ebenfalls in den aufgestellten qualitativen Anforderungen wider. Um dies bestmöglich zu minimieren, sollen angefragte Views zusätzlich in einem Cache abgelegt werden. Sobald der Client dann eine neue View anfragt, kann zunächst überprüft werden, ob bereits eine valide lokale Version vorliegt und dann auf diese zurückgegriffen werden. Dies reduziert nicht nur die Zahl an Anfragen, sondern kann sogar die Nutzung ohne aktive Internetverbindung ermöglichen. Auch hier erfolgte die Implementierung zunächst auf prototypischer Ebene, sodass die Validierung einer lokal gespeicherten View anhand ihres Erstellungsdatums auf dem Client durchgeführt wird. Besonders hier sollte im weiteren Verlauf eine Optimierung stattfinden, indem beispielsweise Views mit einer Versionsnummer versehen werden. Liegt dann eine lokale Kopie vor, kann der Client die Versionsnummer mit dem Server abgleichen und so bestimmen, ob eine aktuelle Version angefragt werden muss. Alternativ könnte auch eine Validierung anhand einer Checksumme erfolgen. Diese Optionen stellen zwar eine zusätzliche Netzwerkanfrage dar, können aber zu Aktualität der Anwendung beitragen.

### Betrachtung der Android-Plattform

Da das umgesetzte Framework zur Cross-Plattform-Entwicklung genutzt werden soll, wurde abschließend zur Umsetzung eine Betrachtung der Android-Plattform durchgeführt. Der Fokus lag hier darauf zu überprüfen, ob und inwieweit technische Entscheidungen, welche im Rahmen des iOS-Prototyps getroffen wurden, auf die Android-Plattform übertragbar sind. Die größte Hürde stellt hierbei die Überführung der JSON-Struktur und das angeschlossene Lowering dar. Im Rahmen eines *Proof of Concepts* (PoC) ließ sich dieses Konzept allerdings, wie in Kapitel 5.3 beschrieben, erfolgreich umsetzen. Auch wenn weitere Aspekte wie die Umsetzung von Modifikatoren oder Anwendungslogik nicht betrachtet wurden, ist von einer vollständigen Umsetzbarkeit des Frameworks auf der Android-Plattform auszugehen. Dies gilt ebenfalls für den gesamten Funktionsumfang aus UI-Komponenten und Modifikatoren, welche bereits in Kapitel 5.1.1 für die iOS- und Android-Plattform betrachtet wurden.

### Zusammenfassung

Zusammenfassend wurden die Ziele, welche in Kapitel 1.3 aufgestellt wurden, durch die beschriebe Konzeption und Umsetzung erfüllt. So stellt die entwickelte Architektur ein plattformunabhängiges und erweiterbares Framework dar, dessen Umsetzbarkeit in Form eines iOS-Prototyps und eines PoCs für die Android-Plattform bewiesen wurde. Dennoch erhebt das Framework keinen Anspruch auf Vollständigkeit. Besonders der gewählte Funktionsumfang wurde zunächst nur prototypisch gewählt. Insofern weitere Funktionalitäten hinzugefügt werden, sollte stets eine Abwägung aus funktionalem

Mehrwert und zusätzlich anfallendem Overhead stattfinden. Beispielsweise kann die Erweiterung des Caches den Prozess der Persistierung verbessern, im Umkehrschluss aber auch zu zusätzlichen Serveranfragen und damit zu einem erhöhten Datenverbrauch führen.

Abschließend soll das umgesetzte Framework anhand der in Kapitel 2.1.3 vorgestellten Grade von Backend-Driven UI eingeordnet werden. Das dargestellte Framework ist hier dem Grad *Server Driven Beyond One Screen* zuzuordnen. Dieser sieht zunächst anhand der vorherigen Grade vor, dass serverseitig Komponenten zur clientseitigen Darstellung bestimmt werden können. Dazu ist der Server fähig, Komponenten visuell anzupassen und diese anhand eines Layouts zu positionieren. Aufbauend darauf können vollständige Views durch das Verschachteln einzelner Komponenten dargestellt und sogar konkrete Aktionen als Reaktion auf Nutzereingaben ausgeführt werden. Dies ermöglicht es auch bereits dargestellte UI-Komponenten nachträglich visuell anzupassen. Abschließend ist es dazu möglich Anwendungen, welche aus mehreren Views bestehen, darzustellen und eine Navigation innerhalb dieser zu ermöglichen. Zur Erreichung des nächst höheren Grads *Server Driven Animation* müsste es im Weiteren möglich sein, innerhalb einer View Animationen und damit beispielsweise animierte Übergänge bei der Navigation umzusetzen.

# 7. Abschluss

Zum Abschluss der Arbeit soll nun ein Fazit gezogen werden. Hierfür erfolgt zunächst eine Zusammenfassung der Arbeit. Dazu sollen die Ergebnisse in Relation mit den ausgangs aufgestellten Forschungsfragen gesetzt werden. Abschließend soll ein Ausblick auf eine potenzielle Verwertung der Arbeit gegeben werden.

## 7.1. Fazit

Backend-Driven UI stellt einen neuen mobilen Entwicklungsansatz dar, welcher bestehende Nachteile der nativen Entwicklung minimieren soll. Der Ansatz lässt sich dem Bereich der Cross-Plattform-Entwicklung zuordnen, welche das Unterstützen mehrerer Plattformen auf Basis einer einheitlichen Code-Basis beschreibt. Im Vergleich zu einer nativen Anwendung, bei der sämtliche Logik sowie UI bereits Teil der Anwendung sind, stellt der Client hier nur eine Interpretationsschicht, ähnlich einem Webbrowser dar. Die eigentlichen Inhalte der Anwendung werden serverseitig in einer strukturierten Darstellung verwaltet und bei Bedarf dem Client bereitgestellt. Dieser muss die entsprechenden Informationen dann verarbeiten und in native UI-Komponenten überführen. Da es sich hierbei um einen aktuellen Entwicklungsansatz handelt, ist dieser innerhalb der Fachliteratur nur schwach vertreten. Besonders durch die Schnelllebigkeit im Rahmen technischer Entwicklungen finden Veröffentlichungen neuer Erkenntnisse daher überwiegend im Rahmen von Vorträgen oder Blog-Einträgen statt.

Das Ziel dieser Arbeit war es daher, ein Backend-Driven UI Framework zur Umsetzung einer Cross-Plattform-Anwendung zu konzipieren und umzusetzen. Hierfür wurde zunächst eine Wissensgrundlage geschaffen, indem bestehende Entwicklungsansätze wie die native Entwicklung oder bestehende Cross-Plattform-Ansätze vorgestellt wurden. Daraufhin wurde das Konzept von Backend-Driven UI genauer betrachtet und in die bestehenden Ansätze eingeordnet. Da die Nutzung eines einheitlichen Datenformats sowie die Überführung in native UI einen essenziellen Bestandteil des Ansatzes darstellen, wurden zudem noch Formate zur Datenrepräsentation und der Bereiche von mobiler UI im Allgemeinen betrachtet. Um die Relevanz des Ansatzes von Backend-Driven UI weiter zu unterstreichen und eine Ausgangslage für die weitere Arbeit zu schaffen, wurden bestehende Backend-Driven UI Systeme in Form des Lona- und SiriusXM-Frameworks vorgestellt. Zur Umsetzung eines Backend-Driven UI Frameworks wurden dann zunächst Anforderungen aus dem Ansatz selbst und den verwandten Arbeiten abgeleitet. Auf Grundlage der aufgestellten Anforderungen wurde dann eine plattformübergreifende und frei erweiterbare Architektur abgeleitet, welche als Grundlage für die prototypische Entwicklung diente. Die Umsetzung erfolgte dann im Rahmen eines Prototyps für die iOS-Plattform. Hier wurde zunächst ein initialer Funktionsumfang in Bezug auf unterstützte UI-Komponenten und visuelle Modifikatoren bestimmt. Im weiteren Verlauf wurde dann die technische Umsetzung der drei

Hauptkomponenten sowie zugehöriger Unterkomponenten beschrieben. Dazu wurden Herausforderungen vorgestellt und ein Android-Prototyp des Frameworks in Form eines minimalen PoCs umgesetzt. Abschließend wurden die Ergebnisse der Umsetzung dann im Rahmen der Diskussion analysiert.

Zusammenfassend stellt Backend-Driven UI eine Alternative zur nativen Entwicklung dar und reiht sich damit in die Gruppe der Cross-Plattform-Ansätze ein. So kann der Ansatz nicht nur dazu beitragen, den Release Cycle einer Anwendung zu optimieren, sondern gleichzeitig auch Nutzern die Vorteile einer nativen Anwendung bieten. Wie aber auch bei allen bestehenden Cross-Plattform-Ansätzen, bietet Backend-Driven UI individuelle Vor- und Nachteile, welche bei der Auswahl bedacht werden müssen. Daher sollte auch die Verwendung der Backend-Driven UI Ansatzes stets kontextabhängig erfolgen.

Abschließend sollen die eingangs aufgestellten Forschungsfragen betrachtet und auf Basis der vorgestellten Ergebnisse beantwortet werden.

- Wie kann eine plattformunabhängige Architektur aussehen, um sowohl die iOS- als auch Android-Plattform in gleichem Funktionsumfang, in Bezug auf die Darstellung von UI und Anwendungslogik, zu unterstützen?

Bei der entwickelten Architektur handelt es sich zunächst um ein möglichst technologiefreies Konzept, dessen Umsetzbarkeit für die iOS-Plattform durch einen ersten Prototyp belegt werden konnte. Um auch eine plattformübergreifende Nutzbarkeit zu gewährleisten, wurden bereits bei der Bestimmung des Funktionsumfanges beide Zielplattformen betrachtet. Dazu konnte die technische Umsetzbarkeit der Architektur durch ein erstes PoC, ebenfalls für Android-Plattform sichergestellt werden. Dazu werden innerhalb der strukturierten Darstellung Abstraktionen für UI-Komponenten verwendet, um so eine plattformunabhängig Beschreibung von UI zu ermöglichen. Ebenfalls erlaubt die Nutzung von JSON als Datenformat eine plattformübergreifende Nutzung. Abschließend werden zur Umsetzung des Servers nur grundlegende Rahmenbedingungen aufgestellt, wodurch Entwickler frei in der Technologiewahl sind.

- Wie kann eine entsprechende Architektur so modular aufgebaut sein, dass Entwickler neben grundlegenden Funktionen auch dynamisch Erweiterungen vornehmen können?

Zunächst wurde mit JSON ein leicht erweiterbares Format, zur Umsetzung der strukturierten Darstellung, gewählt. So ist hier möglich, dynamisch neue Komponenten innerhalb der Datenstruktur abzubilden. Neue Komponenten müssen dazu aber auch clientseitig angelegt werden. Dieser Prozess wurde durch die Verwendung des Strategie-Musters erleichtert, da die abstrakte Strategie *ViewComponent* alle zu implementierenden Eigenschaften und Funktionen vorgibt. Zusätzlich ist der Server durch das Hinzufügen weiterer Endpunkte und Anwendungslogik an spezifische Anwendungskontexte anpassbar.

- Wie kann eine prototypische Umsetzung des Frameworks technologisch aussehen?

Eine mögliche Umsetzung der entwickelten Architektur wurde durch den Prototyp dargestellt. Dafür wurde das JSON-Format zur Abbildung der strukturierten Darstellung gewählt und der Server mit *Node.js* und *Express.js* umgesetzt.

- Wie kann ein Framework zur Umsetzung von mobilen Anwendungen mit Backend-Driven UI aussehen?

Die eingangs aufgestellte Leitfrage lässt sich hier auf Basis der bereits betrachteten Forschungsfragen beantworten. Das Ergebnis dieser Arbeit ist ein Backend-Driven UI Framework, welches auf Basis einer plattformübergreifenden und erweiterbaren Architektur, prototypisch umgesetzt wurde. Dazu können bereits durch den Prototypen erste Anwendungsszenarien in Form einer Backend-Driven UI Anwendung abgebildet werden.

## 7.2. Ausblick

Nachdem im Rahmen der Arbeit ein erster Prototyp eines Backend-Driven UI Frameworks entwickelt wurde, soll nun im Folgenden ein Ausblick für ein mögliches weiteres Vorgehen erfolgen. Hierzu soll dargestellt werden, in welchem Maße das bestehende Framework erweitert werden kann.

Das umgesetzte Framework deckt zunächst nur einen selektiven Funktionsumfang ab. Dieser beschränkt sich besonders auf die Umsetzung grundlegender UI-Komponenten und erlaubt die visuelle Anpassung durch Modifikatoren. Bereits hier kann das Hinzufügen weiterer Komponenten eine mögliche Erweiterung darstellen. Dazu muss abgewägt werden, wie mit plattformspezifischen Komponenten und Modifikatoren umgegangen werden soll, welche nicht plattformübergreifend umgesetzt werden können.

Dazu bietet besonders der Bereich der Anwendungslogik Raum zur Optimierung. Durch die bisherige Darstellung einfacher konditionaler Logik lassen sich zwar bereits simple Anwendungsszenarien realisieren, allerdings sollte die Abdeckung von Anwendungslogik erweitert. In diesem Zuge wäre es zudem möglich, die Verwendung einer DSL zu betrachten. Eine weitere Alternative stellt die bereits erwähnte Verwendung von Remote-Procedure-Calls dar, bei welchen konkrete clientseitige Prozeduren und Funktionen durch den Server ausgelöst werden können.

Des Weiteren sollte mindestens eine Umsetzung des aktuellen Prototyps für die Android-Plattform erfolgen. Das bisherige PoC zeigt zwar die generelle Umsetzbarkeit, deckt allerdings auch nur einen geringen Funktionsumfang ab. Aus einer vollständigen Umsetzung könnten sich dahingehend weitere Erkenntnisse ergeben, die beispielsweise ein nachträgliches Anpassen oder Erweitern der Architektur erfordern würden.

Ebenfalls kann die Server-Komponenten erweitert werden. Denkbar ist hier beispielsweise die Nutzung von User Analytics. Da der Server für die Bereitstellung von Views verantwortlich ist, kann dieser Nutzerprofile aufgrund der Client-Anfragen zu erstellen. Basierend darauf könnten einzelne Views nutzerspezifisch angepasst und optimiert werden. Zusätzlich könnten neue Funktionalitäten durch ein zielgruppenspezifisches A/B-Testing evaluiert werden. Grundsätzlich ist auch die dynamische Generierung von Views durch den Server möglich, ohne dass diese manuell angelegt werden müssen.

Sollte eine händische Erstellung erforderlich sein, kann der Design-Prozess durch eine grafische Oberfläche vereinfacht werden.

Zuletzt wäre die Umsetzung der nativen Anwendungen durch die Verwendung bestehender Cross-Plattform-Ansätze denkbar. Da der Client keine eigene UI benötigt, ist beispielsweise eine Umsetzung mit dem Framework Kotlin-Multiplattform denkbar, welches die Umsetzung geteilter Anwendungslogik ermöglicht. So könnte nicht nur eine gemeinsame UI durch die strukturierte Darstellung, sondern auch der gesamte clientseitige Parsing-Prozess durch eine einheitliche Codebasis abgedeckt werden.

# A. Anhang

## A.1. Tabellen

| Kategorie | Komponente |
|---|---|
| Content | Charts |
| | ImageViews |
| | TextViews |
| | WebViews |
| Layout and organization | Boxes Collections |
| | Disclosure controls |
| | Labels |
| | Lists and Tables |
| | Split Views |
| Menus and actions | Activity Views |
| | Buttons |
| | Context Menus |
| | Edit Menus |
| | Menus |
| | Pop-up Buttons |
| | Pull-down Buttons |
| | Toolbars |
| Navigation and Research | Navigation bars |
| | Search Fields |
| | Sidebars |
| | Tab bars |
| Presentation | Action sheets |
| | Alerts |
| | Page controls |
| | Popovers |
| | Scoll views |
| | Sheets |
| Selection and inputs | Color wells |
| | Onscreen keyboards |
| | Pickers |
| | Segmented controlls |
| | Sliders |
| | Steppers |
| | Text fields |

| | |
|---|---|
| | Toogles |
| Status | Activity rings |
| | Gauges |
| | Progress indicators |
| System experiences | Home Screen quick actions |
| | Live Activities |
| | Notifications |
| | Status Bars |
| | Widgets |

Tabelle A.1.: Übersicht über Komponenten für die iOS Plattform anhand der Human Interface Guidelines

| **Kategorie** | **Komponente** |
|---|---|
| Actions | Common Buttons |
| | Extended FAB |
| | Floating Action Buttons |
| | Icon Buttons |
| | Segmented Button |
| Communication | Badges |
| | Progress indicators |
| | Snackbar |
| Containment | Bottom Sheets |
| | Cards |
| | Dialogs |
| | Divider |
| | Lists |
| Navigation | Bottom app bar |
| | Navigation Bar |
| | Navigation Drawer |
| | Navigation Tabs |
| | Top app bar |
| Selection | Chips |
| | Date Pickers |
| | Menus |
| | Radio Button |
| | Sliders |
| | Switch |
| | Time Pickers |
| Text Inputs | Textfields |

Tabelle A.2.: Übersicht über Komponenten für die Android-Plattform anhand der Material Design Guidelines

| Modifikator | Eigenschaft | Mögliche Werte |
|---|---|---|
| FontStyle | *fontStyle*: String | „BOLD“,<br>„ITALIC“,<br>„UNDERLINE“ |
| Shape | *shape*: String | „CIRCLE“,<br>„RECTANGLE“,<br>„ROUNDED_RECTANGLE“,<br>„CAPSULE“,<br>„ELLIPSE“ |

Tabelle A.3.: Übersicht über mögliche Werte bei FontStyle- und Shape-Modifikator

| Eigenschaft | Beschreibung | Typ |
|---|---|---|
| *id* | ID der Komponente, deren Zustand an den Server übermittelt werden soll | String |
| *type* | Typ der Komponente, deren Zustand an den Server übermittelt werden soll | String |

Tabelle A.4.: Eigenschaften des Typs *PayloadRequirement*

| Eigenschaft | Beschreibung | Typ |
|---|---|---|
| *id* | ID der Komponente zu der das Feld gehört | String |
| *type* | Typ der Komponente zu der das Feld gehört | String |
| *fieldName* | Name des Feldes innerhalb der Komponente | String |
| *value* | Wert, welcher im Falle einer Überprüfung erwartet wird oder neu gesetzt werden soll | String |

Tabelle A.5.: Eigenschaften des Typs *FieldValue*

## A.2. Code

```
<
xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:element name="city" type="xs:string"/>
  <xs:element name="country" type="xs:string"/>
  <xs:element name="postcode" type="xs:int"/>
  <xs:element name="street" type="xs:string"/>
  <xs:element name="element" type="xs:string"/>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="phone" type="xs:string"/>
  <xs:element name="referenceNumber" type="xs:string"/>
```

```
14    <xs:element name="address">
15      <xs:complexType mixed="true">
16        <xs:sequence>
17          <xs:element ref="city" minOccurs="0"/>
18          <xs:element ref="country" minOccurs="0"/>
19          <xs:element ref="postcode" minOccurs="0"/>
20          <xs:element ref="street" minOccurs="0"/>
21        </xs:sequence>
22        <xs:attribute type="xs:string" name="street" use="optional"/>
23        <xs:attribute type="xs:string" name="city" use="optional"/>
24        <xs:attribute type="xs:string" name="country" use="optional"/>
25        <xs:attribute type="xs:short" name="postcode" use="optional"/>
26      </xs:complexType>
27    </xs:element>
28    <xs:element name="furnishing">
29      <xs:complexType>
30        <xs:sequence>
31          <xs:element ref="element" maxOccurs="unbounded" minOccurs="0"/>
32        </xs:sequence>
33      </xs:complexType>
34    </xs:element>
35    <xs:element name="isBarrierFree" type="xs:string"/>
36    <xs:element name="potentialCustomer">
37      <xs:complexType mixed="true">
38        <xs:sequence>
39          <xs:element ref="name" minOccurs="0"/>
40          <xs:element ref="phone" minOccurs="0"/>
41        </xs:sequence>
42      </xs:complexType>
43    </xs:element>
44    <xs:element name="realEstate">
45      <xs:complexType>
46        <xs:sequence>
47          <xs:element ref="referenceNumber" minOccurs="0"/>
48          <xs:element ref="address"/>
49          <xs:element ref="furnishing"/>
50          <xs:element ref="isBarrierFree" minOccurs="0"/>
51          <xs:element ref="potentialCustomer"/>
52        </xs:sequence>
53        <xs:attribute type="xs:string" name="referenceNumber" use="optional"/>
54        <xs:attribute type="xs:string" name="isBarrierFree" use="optional"/>
55      </xs:complexType>
56    </xs:element>
57    <xs:element name="portfolio">
58      <xs:complexType>
59        <xs:sequence>
```

```
60          <xs:element ref="realEstate" maxOccurs="unbounded" minOccurs="0"/>
61        </xs:sequence>
62      </xs:complexType>
63    </xs:element>
64    <xs:element name="root">
65      <xs:complexType>
66        <xs:sequence>
67          <xs:element ref="address"/>
68          <xs:element ref="name"/>
69          <xs:element ref="portfolio"/>
70        </xs:sequence>
71      </xs:complexType>
72    </xs:element>
73  </xs:schema>
```

Listing 18: XML-Schema auf Grundlage der in Listing 1 gezeigten XML-Datei

```
1  {
2    "$schema": "http://json-schema.org/draft-04/schema#",
3    "type": "object",
4    "properties": {
5      "name": {
6        "type": "string"
7      },
8      "address": {
9        "type": "object",
10       "properties": {
11         "street": {
12           "type": "string"
13         },
14         "city": {
15           "type": "string"
16         },
17         "postcode": {
18           "type": "string"
19         },
20         "country": {
21           "type": "string"
22         }
23       },
24       "required": [
25         "street",
26         "city",
27         "postcode",
28         "country"
29       ]
```

```
30      },
31      "portfolio": {
32        "type": "array",
33        "items": [
34          {
35            "type": "object",
36            "properties": {
37              "referenceNumber": {
38                "type": "string"
39              },
40              "address": {
41                "type": "object",
42                "properties": {
43                  "street": {
44                    "type": "string"
45                  },
46                  "city": {
47                    "type": "string"
48                  },
49                  "postcode": {
50                    "type": "string"
51                  },
52                  "country": {
53                    "type": "string"
54                  }
55                },
56                "required": [
57                  "street",
58                  "city",
59                  "postcode",
60                  "country"
61                ]
62              },
63              "furnishing": {
64                "type": "array",
65                "items": [
66                  {
67                    "type": "string"
68                  },
69                  {
70                    "type": "string"
71                  }
72                ]
73              },
74              "isBarrierFree": {
75                "type": "boolean"
```

```
            },
            "potentialCustomer": {
              "type": "object",
              "properties": {
                "name": {
                  "type": "string"
                },
                "phone": {
                  "type": "string"
                }
              },
              "required": [
                "name",
                "phone"
              ]
            }
          },
          "required": [
            "referenceNumber",
            "address",
            "furnishing",
            "isBarrierFree",
            "potentialCustomer"
          ]
        },
        {
          "type": "object",
          "properties": {
            "referenceNumber": {
              "type": "string"
            },
            "address": {
              "type": "object",
              "properties": {
                "street": {
                  "type": "string"
                },
                "city": {
                  "type": "string"
                },
                "postcode": {
                  "type": "string"
                },
                "country": {
                  "type": "string"
                }
```

```
122                 },
123                 "required": [
124                   "street",
125                   "city",
126                   "postcode",
127                   "country"
128                 ]
129               },
130               "furnishing": {
131                 "type": "array",
132                 "items": [
133                   {
134                     "type": "string"
135                   }
136                 ]
137               },
138               "isBarrierFree": {
139                 "type": "boolean"
140               },
141               "potentialCustomer": {
142                 "type": "null"
143               }
144             },
145             "required": [
146               "referenceNumber",
147               "address",
148               "furnishing",
149               "isBarrierFree",
150               "potentialCustomer"
151             ]
152           }
153         ]
154       }
155     },
156     "required": [
157       "name",
158       "address",
159       "portfolio"
160     ]
161   }
```

Listing 19: JSON-Schema auf Grundlage der in Listing 2 gezeigten JSON-Datei

```
{
  "id": "c1",
  "type": "COLUMN",
  "children": [
    {
      "id": "t1",
      "type": "TEXT",
      "text": "Login Screen",
      "modifier":[
        {
          "type": "FONTSTYLE",
          "fontStyle": "BOLD"
        }
      ]
    },
    {
      "id": "ti1",
      "type": "TEXT_INPUT",
      "text": "Username eingeben",
      "modifier": [
        {
          "type": "BORDER"
        },
        {
          "type": "PADDING"
        }
      ],
      "validator": {
        "type": "REGEX",
        "value": "^[a-zA-Z]+$"
      }
    },
    {
      "id": "ti2",
      "type": "TEXT_INPUT",
      "text": "Password eingeben",
      "modifier": [
        {
          "type": "BORDER"
        }
      ],
      "validator": {
        "type": "REGEX",
        "value": "^(?=.*[A-Za-z])(?=.*\\d)[A-Za-z\\d]{8,}$"
```

```
45          }
46        },{
47          "id": "b1",
48          "type": "BUTTON",
49          "text": "Check",
50          "action": {
51            "type": "CHECK_WITH_UI_CHANGES",
52            "checkedFields": [
53              {
54                "id": "ti1",
55                "type": "TEXT_INPUT",
56                "fieldName": "isValid",
57                "value": "true"
58              },
59              {
60                "id": "ti2",
61                "type": "TEXT_INPUT",
62                "fieldName": "isValid",
63                "value": "true"
64              }
65            ],
66            "fieldChanges": [
67              {
68                "id": "b2",
69                "type": "BUTTON",
70                "fieldName": "isActive",
71                "value": "true"
72              }
73            ]
74          }
75        },
76        {
77          "id": "b2",
78          "type": "BUTTON",
79          "text": "Login",
80          "isEnabled": false,
81          "action": {
82            "type": "REQUEST_WITH_PAYLOAD_AND_SCREEN_CHANGE",
83            "destination": "/login",
84            "payloadRequirements": [
85              {
86                "id": "ti1",
87                "type": "TEXT_INPUT"
88              },
89              {
90                "id": "ti2",
```

```
            "type": "TEXT_INPUT"
          }
        ]
      },
      "modifier": [
        {
          "type": "FOREGROUND_COLOR",
          "color": "BLUE"
        }
      ],
    }
  ]
}
```

Listing 20: Darstellung des erweiterten Login-Szenarios als View im JSON-Format

```
const express = require('express')
const fs = require('fs')
const app = express()
const port = 3000

app.get('/view', async (req, res) => {
    let viewName = req.query.viewName
    try {
        let rawData = await fs.promises.readFile(viewName + '.json')
        let parsedJson = JSON.parse(rawData.toString())

        res.send(parsedJson)
    } catch (e) {
        res.status(500)
        res.send(e)
    }
})

app.listen(port, () => {
    console.log(`Example app listening on port ${port}`)
})
```

Listing 21: API-Endpunkt zur Anfrage einer View mit zugehörigem *Express.js* Setup

```
struct ModifierRepresentable: Codable {
    let type: String
    var color: String? = nil
    var fontSize: Int? = nil
    var fontStyle: String? = nil
    var borderWidth: Int? = nil
    var shape: String? = nil
    var elevation: String? = nil
    var radius: Int? = nil
    var x: Int? = nil
    var y: Int? = nil
    var stroke: Int? = nil
    var width: Int? = nil
    var height: Int? = nil
    var action: ClickAction? = nil
}
```

Listing 22: Aufbau der Klasse *ModifierRepresentable*

```
protocol ModifierComponent {
    var id: String { get }
    var type: String { get }
    func render(view: AnyView) -> AnyView
}
```

Listing 23: Aufbau des Interfaces *ModifierComponent*

```
class RowComponent: ViewComponent, ObservableObject {
    var id: String
    var permittedModifier: [String] = ["FOREGROUND_COLOR", "BACKGROUND_COLOR",
        "PADDING", "BORDER"]
    var modifier: [ModifierComponent] = []
    let children: [ViewComponent]

    init(id: String, modifier: [ModifierComponent], children: [ViewComponent]){
        self.id = id
        self.modifier = modifier
        self.children = children
    }

    func generateBaseView() -> AnyView {
        return AnyView(_RowComponent(rowComponent: self))
    }

```

```
18      struct _RowComponent: View {
19          @ObservedObject var rowComponent: RowComponent
20
21          var body: some View {
22              HStack {
23                  ForEach(rowComponent.children, id: \.id){ child in
24                      child.render()
25                  }
26              }
27          }
28      }
29  }
```

Listing 24: Aufbau der Klasse *RowComponent*

```
1   class ImageComponent(
2       override val id: String,
3       val imagePath: String
4   ): ViewComponent {
5
6       @Composable
7       override fun Render() {
8           val image: Painter = painterResource(id = R.drawable.test_image)
9           Image(painter = image, contentDescription = imagePath)
10      }
11  }
12
13  class ColumnComponent(
14      override val id: String,
15      val children: List<ViewComponent>
16  ): ViewComponent {
17
18      @Composable
19      override fun Render() {
20          Column {
21              children.forEach {
22                  it.Render()
23              }
24          }
25      }
26  }
```

Listing 25: Weitere Implementierte UI-Komponenten im Rahmen des Android-Clients

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[AirBnB 2017] AirBnB: *Lona Github Repository.* https://github.com/Lona/Lona. Version: 2017. – Letzter Zugriff: 26.02.2023

[Asefa 2022] Asefa, Beselam G.: Building Android Component Library Using Jetpack Compose. (2022)

[Asri 2020] Asri, Vipul: *Dynamic screens using server-driven UI in Android.* https://proandroiddev.com/dynamic-screens-using-server-driven-ui-in-android-262f1e7875c1. Version: 2020. – Letzter Zugriff: 26.02.2023

[Biørn-Hansen u. a. 2018] Biørn-Hansen, Andreas ; Grønli, Tor-Morten ; Ghinea, Gheorghita: A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. In: *ACM Comput. Surv.* 51 (2018), nov, Nr. 5. http://dx.doi.org/10.1145/3241739. – DOI 10.1145/3241739. – ISSN 0360–0300

[Biørn-Hansen u. a. 2019] Biørn-Hansen, Andreas ; Grønli, Tor-Morten ; Ghinea, Gheorghita ; Alouneh, Sahel: An empirical study of cross-platform mobile development in industry. In: *Wireless Communications and Mobile Computing* 2019 (2019)

[Birch 2020] Birch, Joe: *Server Driven UI, Part 1: The Concept.* https://joebirch.co/android/server-driven-ui-part-1-the-concept/. Version: 2020. – Letzter Zugriff: 26.02.2023

[Chan 2021] Chan, Stephanie: *Schätzung der Bruttoumsätze mit Apps nach App Stores weltweit in den Jahren 2016 bis 2021.* https://de.statista.com/statistik/daten/studie/802760/umfrage/schaetzung-des-umsatzes-mit-apps-nach-app-store-weltweit/. Version: 2021. – Letzter Zugriff: 26.02.2023

[El-Kassas u. a. 2017] El-Kassas, Wafaa S. ; Abdullah, Bassem A. ; Yousef, Ahmed H. ; Wahba, Ayman M.: Taxonomy of cross-platform mobile applications development approaches. In: *Ain Shams Engineering Journal* 8 (2017), Nr. 2, S. 163–190

[Elye 2021] Elye: *The Possible Type of Server Driven UI for Mobile.* https://medium.com/mobile-app-development-publication/the-possible-types-of-server-driven-ui-for-mobile-128e8c4433bb. Version: 2021. – Letzter Zugriff: 26.02.2023

[Euler 2005] Euler, Prof. Dr. S.: *Netzwerk-Know-how (tecCHANNEL COMPACT) Kapitel 9: Remote Procedure Call – RPC.* https://learn.microsoft.com/de

-de/previous-versions/dn151205(v=technet.10)?redirectedfrom=MSDN. Version: 2005. – Letzter Zugriff: 26.02.2023

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

[Friesen 2016] Friesen, Jeff: *Java XML and JSON*. Springer, 2016

[García u. a. 2015] García, Cristian G. ; Espada, Jordán P. ; Bustelo, Begoñ. G. ; Lovelle, Juan Manuel C.: Swift vs. objective-c: A new programming language. In: *IJIMAI* 3 (2015), Nr. 3, S. 74–81

[Góis Mateus u. Martinez 2019] Góis Mateus, Bruno ; Martinez, Matias: An empirical study on quality of Android applications written in Kotlin language. In: *Empirical Software Engineering* 24 (2019), S. 3356–3393

[Hassan u. a. 2020] Hassan, Safwat ; Bezemer, Cor-Paul ; Hassan, Ahmed E.: Studying Bad Updates of Top Free-to-Download Apps in the Google Play Store. In: *IEEE Transactions on Software Engineering* 46 (2020), Nr. 7, S. 773–793. http://dx.doi.org/10.1109/TSE.2018.2869395. – DOI 10.1109/TSE.2018.2869395

[Huber u. a. 2021] Huber, Stefan ; Demetz, Lukas ; Felderer, Michael: Pwa vs the others: A comparative study on the ui energy-efficiency of progressive web apps. In: *Web Engineering: 21st International Conference, ICWE 2021, Biarritz, France, May 18–21, 2021, Proceedings* Springer, 2021, S. 464–479

[Hunt 2021] Hunt, John: *Beginner's Guide to Kotlin Programming*. Springer, 2021

[json.org ] json.org: *Einführung in JSON*. https://www.json.org/json-en.html. – Letzter Zugriff: 26.02.2023

[Kelly u. Silverman 2019] Kelly, Laura ; Silverman, Nathanael: *KotlinConf 2019: Lona: Scaling Server-driven UI by Laura Kelly Nathanael Silverman*. https://www.youtube.com/watch?v=Ir8lq4rSyyc. Version: 2019. – Letzter Zugriff: 26.02.2023

[Lamhaddab u. a. 2019] Lamhaddab, Khalid ; Lachgar, Mohamed ; Elbaamrani, Khalid: Porting mobile apps from iOS to android: A practical experience. In: *Mobile Information Systems* 2019 (2019)

[Lewis u. a. 2012] Lewis, Rory ; McCarthy, Yulia ; Moraco, Stephen M.: *Beginning IOS Storyboarding: Using Xcode*. Apress, 2012

[Li u. a. 2020] Li, Xiaozhou ; Zhang, Boyang ; Zhang, Zheying ; Stefanidis, Kostas: A sentiment-statistical approach for identifying problematic mobile app updates based on user reviews. In: *Information* 11 (2020), Nr. 3, S. 152

[Marchenko 2023] Marchenko, S: JETPACK COMPOSE: NEW APPROACHES TO ANDROID UI DEVELOPMENT. In: *Publishing House "Baltija Publishing"* (2023)

[Maximo 2020] Maximo, Rodrigo: *Backend Driven Development — iOS*. `https://medium.com/movile-tech/backend-driven-development-ios-d1c726f2913b`. Version: 2020. – Letzter Zugriff: 26.02.2023

[Muema 2021] Muema, Linus: *Declarative vs Imperative UI in Android*. `https://www.section.io/engineering-education/declarative-vs-imperative-ui-android/`. Version: 2021. – Letzter Zugriff: 26.02.2023

[Rogers u. Gratch 2022] Rogers, Michael P. ; Gratch, Jonathan: A snapshot of current and trending practices in mobile application development. In: *Journal of Computing Sciences in Colleges* 37 (2022), Nr. 6, S. 54–66

[Rucker 2021] Rucker, Sean: *What is Server-Driven UI?* `https://www.judo.app/blog/server-driven-ui/`. Version: 2021. – Letzter Zugriff: 26.02.2023

[Rupp u. a. 2009] Rupp, Chris ; Simon, Matthias ; Hocker, Florian: Requirements engineering und management. In: *HMD Praxis der Wirtschaftsinformatik* 46 (2009), Nr. 3

[Ryan 2021] Ryan, Mark: Implementing a Design System for Mobile Cross Platform Development. (2021)

[Sarmah u. a. 2018] Sarmah, Upasana ; Bhattacharyya, DK ; Kalita, Jugal K.: A survey of detection methods for XSS attacks. In: *Journal of Network and Computer Applications* 118 (2018), S. 113–143

[Soininen 2021] Soininen, Visa: Jetpack Compose vs React Native–Differences in UI Development. (2021)

[Steinberger 2021] Steinberger, Peter: *The new shiny - On the shift from imperative to declarative UI, and what it might mean for the apps we build today and tomorrow*. `https://increment.com/mobile/the-shift-to-declarative-ui/`. Version: 2021. – Letzter Zugriff: 26.02.2023

[Tandel u. Jamadar 2018] Tandel, Sayali ; Jamadar, Abhishek: Impact of progressive web apps on web app development. In: *International Journal of Innovative Research in Science, Engineering and Technology* 7 (2018), Nr. 9, S. 9439–9444

[Varma u. Varma 2019] Varma, Jayant ; Varma, Jayant: What Is SwiftUI. In: *SwiftUI for Absolute Beginners: Program Controls and Views for iPhone, iPad, and Mac Apps* (2019), S. 1–8

[W3C 2015] W3C: *Extensible Markup Language (XML)*. `https://www.w3.org/XML/`. Version: 2015. – Letzter Zugriff: 26.02.2023

[Wolf u. Horner 2022] Wolf, Thomas ; Horner, Stuart: *A Journey To Building A Server Driven UI Framework*. `https://www.droidcon.com/2022/09/29/a-journey-to-building-a-server-driven-ui-framework/`. Version: 2022. – Letzter Zugriff: 26.02.2023