

Duale Hochschule Baden-Württemberg Karlsruhe  
Fachbereich technische Informatik

# **Entwicklung und Erstellung einer Dating-App auf Basis von Film- und Serienpräferenzen für mobile Endgeräte**

Studienarbeit von

**Leon Gieringer, Robin Meckler & Vincent Schreck**

betreut durch

**Prof. Dr. Kai Becher**

abgegeben am

17. Mai 2021

## **Erklärung**

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema: „Entwicklung und Erstellung einer Dating-App auf Basis von Film- und Serienpräferenzen für mobile Endgeräte“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

---

Ort      Datum

---

Leon Gieringer

---

Ort      Datum

---

Robin Meckler

---

Ort      Datum

---

Vincent Schreck

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Methode . . . . .	1
<b>2 Theoretische Grundlagen</b>	<b>2</b>
2.1 Netzwerkprotokolle . . . . .	2
2.1.1 Schichtenmodell . . . . .	2
2.1.2 HTTP . . . . .	3
2.1.3 HTTPS . . . . .	5
2.2 JavaScript . . . . .	6
2.2.1 Historie . . . . .	6
2.2.2 Wesentliche Programmereigenschaften . . . . .	6
2.2.3 Anwendungsbereiche . . . . .	7
2.3 Node.js . . . . .	8
2.3.1 Architektur . . . . .	8
2.3.2 Module . . . . .	10
2.4 NoSQL-Datenbank . . . . .	17
2.4.1 NoSQL-Datenbanktypen . . . . .	17
2.4.2 BASE . . . . .	19
2.4.3 CAP-Theorem . . . . .	19
2.4.4 MongoDB . . . . .	20
2.5 Firebase . . . . .	27
2.5.1 Firebase Authentifizierung . . . . .	27
2.5.2 Cloud Firestore . . . . .	27
2.5.3 Cloud Storage . . . . .	30
2.5.4 Cloud Functions . . . . .	30
2.5.5 Cloud Messaging . . . . .	31
2.5.6 Google AdMob . . . . .	32
2.6 Anwendungsentwicklung für mobile Endgeräte . . . . .	32
2.6.1 Begriffe . . . . .	33
2.6.2 Plattformspezifische native Apps . . . . .	33
2.6.3 Plattformübergreifende Anwendungen . . . . .	34
2.7 Frameworks zur mobilen, plattformübergreifenden Entwicklung . . . . .	36
2.7.1 React Native . . . . .	37
2.7.2 Flutter . . . . .	42
2.8 Recommender System . . . . .	46
2.8.1 Nutzerinformation . . . . .	46
2.8.2 Content-based filtering . . . . .	47
2.8.3 Collaborative Filtering . . . . .	47
2.8.4 Ähnlichkeit von Objekten und Nutzern . . . . .	48
<b>3 Planung</b>	<b>49</b>
3.1 Konzept . . . . .	49
3.2 Komponenten . . . . .	49

<b>4 Auswahl geeigneter Technologie</b>	<b>52</b>
4.1 Anwendungsframework . . . . .	52
4.2 Server . . . . .	53
4.3 Datenbank . . . . .	54
4.4 Kommunikationsschnittstelle . . . . .	54
4.5 Film-Datenbank . . . . .	54
<b>5 Serverseitige Implementierung</b>	<b>56</b>
5.1 Datenbank . . . . .	56
5.1.1 Bereitstellung der Datenbank . . . . .	56
5.1.2 Importieren der Film- und Städtedaten . . . . .	56
5.2 StreamSwipe-Webserver . . . . .	57
5.2.1 Bereitstellung des Webservers . . . . .	57
5.2.2 Geplante Architektur . . . . .	58
5.2.3 Sichere Kommunikation . . . . .	59
5.2.4 Datenbankverbindung . . . . .	60
5.2.5 Datenbankmodelle und Schemata . . . . .	60
5.2.6 Datenbankzugriff . . . . .	61
5.2.7 Controller . . . . .	69
5.2.8 Routing . . . . .	74
5.2.9 Weitere Backendfunktionalitäten . . . . .	75
5.3 Firebase . . . . .	79
5.3.1 Authentifizierung . . . . .	79
5.3.2 Nutzerdaten . . . . .	80
5.3.3 Chatfunktion . . . . .	81
5.3.4 Sicherheit . . . . .	84
<b>6 Implementierung der mobilen Anwendung</b>	<b>86</b>
6.1 Klassenmodelle . . . . .	86
6.2 Kommunikationschnittstellen . . . . .	87
6.3 Manager . . . . .	89
<b>7 Benutzeroberflächen der mobilen Anwendung</b>	<b>91</b>
7.1 Aspekte von Benutzeroberflächen . . . . .	91
7.2 Barrierefreiheit . . . . .	92
7.2.1 Barrierefreiheit in mobilen Anwendungen . . . . .	92
7.2.2 Barrierefreiheit in Filmen und Serien . . . . .	92
7.2.3 Barrierefreiheit bei StreamSwipe . . . . .	93
7.3 Oberflächen von StreamSwipe . . . . .	95
7.3.1 Login-Screen . . . . .	95
7.3.2 Home-Screen . . . . .	98
7.3.3 Swipe-Screen . . . . .	99
7.3.4 Chat . . . . .	101
7.3.5 Benutzerprofil . . . . .	102
<b>8 Probleme</b>	<b>105</b>
<b>9 Anwendbarkeit</b>	<b>107</b>
<b>10 Ausblick</b>	<b>109</b>
<b>11 Fazit</b>	<b>111</b>

<b>A</b>	<b>Verfasser einzelner Abschnitte</b>	<b>112</b>
<b>B</b>	<b>Sicherheitsregeln</b>	<b>113</b>

## Abbildungsverzeichnis

1	Netzwerkprotokolle: OSI-Schichten . . . . .	2
2	HTTP-Nachrichtenaufbau . . . . .	4
3	JavaScript Objekt . . . . .	7
4	Multithreaded / Blocking I/O . . . . .	8
5	Single Threaded / Non Blocking I/O . . . . .	9
6	Middleware . . . . .	12
7	Graphendatenbank Beispiel . . . . .	17
8	Zeilen- und spaltenorientierte Speicherung . . . . .	18
9	Visualisierung des CAP-Theorems . . . . .	19
10	MongoDB Architektur . . . . .	20
11	MongoDB Replica Set . . . . .	25
12	MongoDBCompass Benutzeroberfläche . . . . .	26
13	Datenmodell in Firebase . . . . .	28
14	Cloud Functions Anwendungsfall Benachrichtigung . . . . .	31
15	Firebase Cloud Messaging Architektur . . . . .	32
16	Google AdMob Anzeigemöglichkeiten . . . . .	33
17	Kategorisierung verschiedener plattformübergreifender Ansätze . . . . .	34
18	Struktur einer hybriden Anwendung . . . . .	35
19	Beliebtheit nach Framework im Bereich der plattformübergreifenden mobilen Entwicklung . . . . .	37
20	Alte und neue Architektur von React Native . . . . .	38
21	Kompatibilität der Dart Plattform . . . . .	42
22	Bibliotheken und Ebenen der Flutter Plattform . . . . .	44
23	Widget Baum einer beispielhaften Anwendung . . . . .	45
24	Deklarative Benutzeroberfläche . . . . .	45
25	Komponentendiagramm . . . . .	51
26	Node.JS Installation . . . . .	57
27	Webserver Architektur . . . . .	59
28	MongoDB - Aufbau der Collections und Beziehungen . . . . .	61
29	Node.js Server - Models Struktur . . . . .	62
30	Node.js Server - Services Struktur . . . . .	62
31	Node.js Server - Controller Struktur . . . . .	69
33	Vereinfachte Architektur der mobilen Anwendung . . . . .	86
34	Screenshots als Beispiele für Barrierefreiheit . . . . .	94
35	Screenshots der Anmeldeseiten . . . . .	97
36	Screenshots des Home-Screens . . . . .	98
37	Screenshots des Swipe-Screen . . . . .	100
38	Screenshots der Chat-Seiten . . . . .	102
39	Screenshots der Profilseite . . . . .	104

## Tabellenverzeichnis

1	Kurzbeschreibung der OSI-Schichten . . . . .	3
2	HTTP Statuscodebeschreibungen . . . . .	5
3	Express.js Module . . . . .	16
4	Aggregation Framework: Pipeline-Operatoren . . . . .	23
5	CRUD-Operatoren . . . . .	23
6	Nutzer-Item Matrix mit Bewertungen. Jede Zelle $r_{u;i}$ steht hierbei für die Bewertung des Nutzers $u$ an der Stelle $i$ . . . . .	46

# Quellcodeverzeichnis

1	Benannter Export von Modulen . . . . .	10
2	Import von Modulen . . . . .	10
3	Einfacher Webserver <sup>1</sup> . . . . .	11
4	Express.json Middleware benutzen . . . . .	12
5	Routinghandler erstellen <sup>2</sup> . . . . .	12
6	Routinghandler benutzen . . . . .	13
7	Mongoose Schema - Beispiel . . . . .	14
8	Model erstellen und exportieren . . . . .	15
9	Model importieren - Objekt instanziieren und persistent speichern . . . . .	15
10	CRUD-Beispielfunktionen eines Mongoose-Models . . . . .	15
11	Mongoose: Verbindung zur Datenbank aufbauen . . . . .	16
12	Mongoose Verbindungsoptionen <sup>3</sup> . . . . .	16
13	JSON - BSON Vergleich . . . . .	21
14	MongoDB Read . . . . .	21
15	MongoDB Read Modifikation . . . . .	21
16	MongoDB Aggregate . . . . .	22
17	MongoDB Create . . . . .	22
18	MongoDB Update . . . . .	23
19	MongoDB Remove . . . . .	24
20	Beschränkung des Zugriffs auf Dokumente der Sammlung <code>cities</code> . . . . .	28
21	Hierarchische Zugriffsbeschränkung . . . . .	29
22	Datenvalidierung für atomare Operationen . . . . .	29
23	Validierung nach Dateigröße . . . . .	30
24	JSX Hello World Element . . . . .	39
25	Native Komponenten . . . . .	39
26	Eigene Komponenten . . . . .	40
27	State mit <code>useState</code> Hook . . . . .	41
28	Datei <code>package.json</code> . . . . .	58
29	Einfache Verbindung . . . . .	59
30	Gesicherte Verbindung . . . . .	59
31	Verbindung zur MongoDB-Datenbank . . . . .	60
32	Swipe Schema und Model . . . . .	61
33	<code>movieService.js</code> - <code>FindMoviesExcept</code> . . . . .	62
34	User Service - <code>CreateUser</code> . . . . .	63
35	User Service - <code>CheckExistence</code> . . . . .	63
36	User Service - <code>CheckExistence</code> . . . . .	64
37	User Service - <code>ChangeCityFromUser</code> . . . . .	64
38	Match Service - <code>CreateUserMatchDocument</code> . . . . .	65
39	Match Service - <code>CreateUserMatchDocument</code> . . . . .	65
40	Match Service - <code>AddNormalMatchToUser</code> . . . . .	65
41	Match Service - <code>SuperMatchMarkAsRemoved</code> . . . . .	66
42	Swipe Service - <code>AddSwipe</code> . . . . .	67
43	Swipe Service - <code>RequestSuperlikeSwipes</code> . . . . .	68
44	Swipe Service - <code>FindAllSwipedMoviesByUserID</code> . . . . .	68
45	City Service - <code>GetAllInhabitedCities</code> . . . . .	68
46	<code>movieController.js</code> Imports und Funktionen . . . . .	69
47	Controller Firebase-Authentifizierung . . . . .	70
48	MovieController - <code>RequestMovie</code> - Excluded Movies . . . . .	70

49	MovieController - RequestMovie - Excluded Movies . . . . .	71
50	UserController - Create User - Transaktionsstart . . . . .	71
51	UserController - Create User - Dokumente erstellen . . . . .	72
52	UserController - Change User . . . . .	72
53	MatchController - RequestMatches . . . . .	73
54	Routing in server.js . . . . .	74
55	Routing in movieRouter.js . . . . .	74
56	Routing in userRouter.js . . . . .	75
57	Routing in matchRouter.js . . . . .	75
58	Routing in swipeRouter.js . . . . .	75
59	Firebase-Service Register . . . . .	76
60	Firebase-Service Register . . . . .	76
61	Match Manager - startMatching - Teil 1: Finde Matches . . . . .	77
62	Match Manager - checkSuperMatches . . . . .	78
63	Match Manager - startMatching - Teil 2: Speichere Matches . . . . .	79
64	Anzeige abhängig ob ein aktueller Nutzer existiert . . . . .	80
65	Sortierung der UIDs . . . . .	82
66	Cloud Functions zur Erstellung von Benachrichtigungen . . . . .	82
67	Movie Modell - JSON Konvertierung . . . . .	86
68	Bad Certificate - Workaround . . . . .	87
69	Movie Service - MoviesRequest . . . . .	88
70	Codeausschnitt in Dart von einem Button mit Semantiken. . . . .	94
71	Sicherheitsregeln Firestore . . . . .	113

## 1 Einleitung

Die Zahl der Scheidungen in Deutschland hat sich während den Einschränkungen durch die COVID-19 Pandemie 2020 verfünfacht [14]. Neben dem erhöhten Ansturm auf Rechtskanzleien haben sich auch unverheiratete Paare zu Zeiten des Lockdowns getrennt und die Paartherapien sind flächendeckend ausgebucht. Die Menschen suchen sich Partner aus, die sie zwar attraktiv finden, mit denen sie jedoch kaum gemeinsame Interessen und Ansichten teilen. Sind diese Leute gezwungen, Zeit miteinander zu verbringen realisieren sie, dass ihre Beziehung nicht passt. Wir wollen diesen gravierenden Fehler in seinem Keim ersticken und revolutionieren das Dating-Game mit einem Verfahren, bei dem persönliche Vorlieben im Vordergrund stehen und das Aussehen zweitrangig ist.

### 1.1 Motivation

Bereits vor tausenden von Jahren haben sich die Menschen Partner gesucht. Mit der ersten Monogamie kam auch die erste Beziehung und dadurch wahrscheinlich die ersten Beziehungsprobleme.

Eine der wichtigsten Grundlagen einer Beziehung sind gleiche Ansichten, Interessen und Vorlieben. Oberflächliche Merkmale wie Aussehen und Geld hingegen sind vergänglich - „Schönheit vergeht und Charakter besteht“. Jedoch ist es nahezu unmöglich einen Partner nach diesen Kriterien auszusuchen, denn man weiß erst, wie gut man zueinander passt, nachdem man sich kennengelernt hat. Viele möglicherweise sehr glückliche Beziehungen finden gar nicht statt, da die Person durch ein eigentlich weniger wichtiges Kriterium herausgefiltert wurde. Sucht man die Ursache dieses Problems, ist man schnell bei der Art des Kennenlernens. Der erste Eindruck ist gewöhnlicherweise optischer Natur. Dementsprechend ist Aussehen in der Realität oft der erste Berührungspunkt zweier Menschen, was durch StreamSwipe an eine spätere Position tritt. Wir bieten die Lösung zu einem jahrtausendealten Problem der Menschheit!

### 1.2 Methode

Gerade in den letzten Jahren genießt das Medium Film und Serie einen immer höheren Stellenwert in der Gesellschaft. Durch Video-on-Demand Plattformen wie Netflix, Disney+ und Amazon Prime Video sind Filme und Serien omnipräsent geworden und das Angebot scheint endlos zu sein. Der Zugriff auf komplette Serien wurde dadurch stark vereinfacht und der Nutzer kann einer Serie oder Filmreihe treu bleiben, da er keine Folge mehr verpassen kann. So kann dieses Medium bereits bei vielen Menschen eine Charaktereigenschaft werden und Charaktereigenschaften vieler Zuschauer passen sich an Filmcharaktere an.

Bereits 2017 haben die 18- bis 39-Jährigen an durchschnittlich 4 Tagen pro Woche eine Serie angeschaut [15]. Aus diesen Vorlieben lässt sich sehr viel auslesen. Bei StreamSwipe wird dies ausgenutzt und über eine Film- und Serienauswahl des Nutzers ein Geschmack berechnet, der über einen Algorithmus mit anderen ähnlichen Geschmäcken gematcht wird. Sobald ein Match entstanden ist, öffnet sich ein privater Chat und die beiden Personen können sich austauschen und verabreden.

OSI-Schicht	TCP/IP-Schicht	Beispiel
Anwendungen (7)	Anwendungen	HTTP, UDS, FTP, SMTP, POP, Telnet, DHCP, OPC UA
Darstellung (6)		TLS, SOCKS
Sitzung (5)		
Transport (4)	Transport	TCP, UDP, SCTP
Vermittlung (3)	Internet	IP (IPv4, IPv6), ICMP (über IP)
Sicherung (2)	Netzzugang	Ethernet, Token Bus, Token Ring, FDDI
Bitübertragung (1)		

Abbildung 1: Netzwerkprotokolle: OSI-Schichten <sup>4</sup>

## 2 Theoretische Grundlagen

### 2.1 Netzwerkprotokolle

#### 2.1.1 Schichtenmodell

Eine der gängigsten Arten der Kommunikation findet heutzutage über das Internet statt. Dabei handelt es sich um ein weltweit verbundenes Netz von Rechnern. Zur Gewährleistung einer effizienten und geregelten Datenübertragung der heterogenen Computer im Internet wurden Regelwerke, die sogenannten Netzwerkprotokolle, benötigt. Um das Jahr 1980 wurden daraufhin von verschiedenen Computerherstellern modularisierte Protokolle entwickelt, die fortan als Standard für die digitale Übertragung innerhalb von Rechnernetzen gelten sollen [Prot1]. Es musste eine Vielzahl von Aufgaben bewältigt und Anforderung bezüglich Zuverlässigkeit, Sicherheit, Effizienz etc. erfüllt werden. Die Aufgaben reichten dabei von der elektronischen Übertragung der Signale bis zur geregelten Reihenfolge der in der Kommunikation abstrakteren Aufgaben [Prot2]. Aus den zu lösenden Problemen und Anforderung kristallisierten sich sieben Schichten bzw. Ebenen heraus. Jede einzelne Schicht setzt dabei separat eine Anforderung um und kann dabei durch verschiedene Protokolle realisiert werden. In dem sich etablierten OSI-Schichtenmodell bauen die einzelnen Schichten aufeinander auf, wobei die unterste Schicht das Fundament ist. Die Open Systems Interconnection (OSI) wurde von der International Organization for Standardization (ISO), der Internationalen Organisation für Normung, als Grundlage für die Bildung von offenen Kommunikationsstandards entworfen. Das Modell wird in Abbildung 1 dargestellt. Die einzelnen Aufgaben werden in der Tabelle 1 beschrieben.

Zusätzlich zum OSI-Modell existiert das in den 1960er-Jahren entwickelte TCP/IP-Referenzmodell. Entwickler dieses Schichtmodells war das Verteidigungsministerium der Vereinigten Staaten, auch bekannt als das Department of Defense (DoD). Dementsprechend trägt das TCP/IP-Referenzmodell auch den Namen DoD-Schichtenmodell [Prot3].

---

<sup>4</sup>Quelle: <https://de.wikipedia.org/wiki/Internetprotokollfamilie>, letzter Zugriff: 06. April 2021

Tabelle 1: Kurzbeschreibung der OSI-Schichten [Prot4]

OSI-Schicht	Aufgabe
Anwendungen	Funktionen für Anwendungen, sowie die Dateneingabe und -ausgabe.
Darstellung	Umwandlung der systemabhängigen Daten in ein unabhängiges Format.
Sitzung	Steuerung der Verbindungen und des Datenaustauschs.
Transport	Zuordnung der Datenpakete zu einer Anwendung.
Vermittlung	Routing der Datenpakete zum nächsten Knoten.
Sicherung	Fehlererkennungsmechanismen / Segmentierung der Pakete in Frames und Hinzufügen von Prüfsummen.
Bitübertragung	Umwandlung der Bits in ein zum Medium passendes Signal und physikalische Übertragung.

**IP:** In der Vermittlungsschicht des OSI-Schichtenmodells findet, unabhängig vom Übertragungsmediums und der genutzten Topologie, die logische Adressierung der Endgeräte statt. Das geläufigste Protokoll dafür ist das Internet Protocol (IP). Jedem am Netz verbundenen Teilnehmer wird eine IP-Adresse zugewiesen. Die bekannteste Notation ist die 32 Bit lange IPv4-Adressen und die IPv6-Adressen mit einer Größe von 128 Bit.

**TCP/UDP:** In der Transportschicht wird eine Ende-zu-Ende-Kommunikation ermöglicht. Sie ist das Bindeglied zwischen den anwendungsorientierten und den transportorientierten Schichten. Die geläufigsten Protokolle sind das User Datagram Protocol (UDP) und das Transmission Control Protocol (TCP). UDP ist verbindungslos und daher unzuverlässiger, ist aber durch weniger Overhead belastet. TCP hingegen ist verbindungsorientiert und dadurch zuverlässiger beim Datentransfer. Jedes netzwerkfähige Gerät enthält eine Vielzahl von Ports, die primär zur Unterscheidung zwischen Datenströmen aus Anwendungen bei Netzwerkverbindungen genutzt werden. Anhand des genutzten Ports bei Netzwerkanfragen wissen Webserver, welches Protokollverfahren genutzt werden soll.

### 2.1.2 HTTP

Das Hypertext Transfer Protocol, kurz HTTP, ist ein zustandloses Protokoll zur Übertragung von Daten auf der Anwendungsschicht.

**Kommunikation:** Unter einer Nachricht versteht man in HTTP die Kommunikationseinheiten zwischen dem Zentralrechner (Server) und dem, der einen Dienst vom Server abruft (Client). Man unterscheidet dabei zwischen der Anfrage (Request) vom Client an den Server und der Antwort (Response) als Reaktion vom Server zum Client.

Eine Nachricht besteht aus dem Nachrichtenkopf (Message Header, kurz Header) und dem Nachrichtenrumpf (Message Body, kurz Body). Der Header enthält generelle Informationen über die Nachricht wie zum Beispiel den Methodentyp, das Datenformat, den genutzten Kompressionsalgorithmus, die Länge der Nachricht oder die verwendete Codierung im Body.

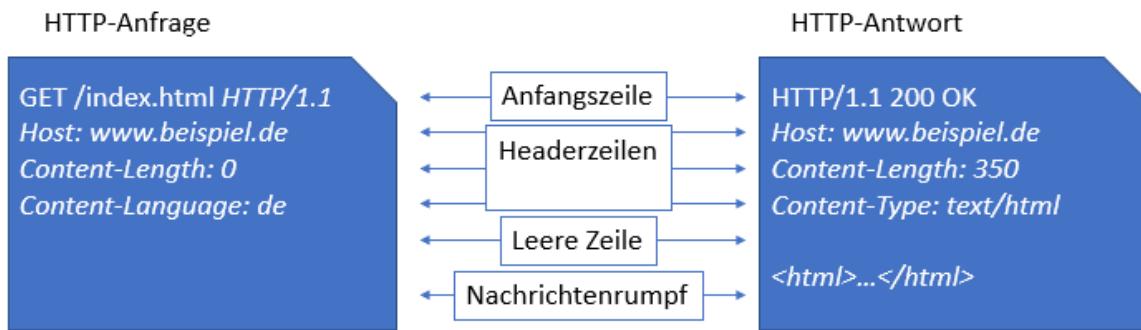


Abbildung 2: HTTP-Nachrichtenaufbau

In Abbildung 2 ist der Aufbau der HTTP-Nachrichten dargestellt. Die erste Zeile des Nachrichtenkopfs ist dreiteilig und besteht bei der Anfrage aus dem Namen der Anfragemethode, dem Pfad zur angeforderten Ressource (Uniform Resource Locator, kurz URL) und der verwendeten HTTP-Version. Die Anfangszeile einer HTTP-Antwort dagegen besteht zunächst aus der verwendeten HTTP-Version, gefolgt von dem zweiteiligen Status-Code. Der Anfangszeile beider Nachrichtentypen folgt eine Reihe von Headerzeilen, wobei jede Zeile aus einem Schlüsselwort/Wert-Paar besteht und die für die Datenübertragung wichtigen Informationen übergibt. Der Nachrichtenrumpf, der mit den Nachrichtenkopf über einen Zeilenumbruch syntaktisch voneinander getrennt wird, enthält schließlich die Nutzdaten.

**Methoden:** HTTP bietet fest definierte Standard-Methoden für Anfragen, die für verschiedene Aufgaben gedacht sind. Im Folgenden werden die wichtigsten Methoden beschrieben:

- *GET* ist die gebräuchlichste Methode. Sie fordert vom Server eine Ressource, die bei Erfolg in der Antwort im Body zurückgegeben wird.
- *POST* ist für die Änderung oder Erzeugung einer Ressource vorgesehen. Dafür werden bei der Anfrage zusätzlich Daten im Body der Nachricht übertragen.
- *PUT* dient dazu, eine Ressource zu verändern, oder bei Nichtexistenz zu erstellen.
- *PATCH* ändert eine bestehende Ressource ohne diese wie bei PUT vollständig zu ersetzen.
- *DELETE* löscht die angegebene Ressource auf dem Server.
- *OPTIONS* liefert eine Liste von Methoden und Merkmale, die vom Server unterstützt werden.

Tabelle 2: HTTP Statuscodebeschreibungen

Typ	Status-Code	Beispiele
Informational	1xx	100 Continue, 101 Switching
Success	2xx	200 OK, 201 Created, 202 Accepted
Redirection	3xx	300 Multiple Choice, 301 Moved Permanently
Client Error	4xx	400 Bad Request, 403 Forbidden
Server Error	5xx	500 Internal Server Error

**Status-Codes:** HTTP-Antworten senden in der Anfangszeile ihrer Nachricht Status-Codes. Die Angabe ist zweiteilig und besteht aus einer standardisierten Statuskennzahl sowie einer kurzen textuellen Beschreibung, die zusammen Auskunft über den Bearbeitungszustand der zugehörigen Anfrage geben. Tabelle 2 stellt die Status-Codes und ihre Beschreibungen anhand von Beispielen dar.

### 2.1.3 HTTPS

Das HTTP-Protokoll hat den großen Nachteil, dass die Nachrichten unverschlüsselt und ungesichert übertragen werden. Die Daten können bei der Übertragung von Dritten empfangen, gelesen und verändert werden. Hypertext Transfer Protocol Secure, kurz HTTPS, soll dem entgegenwirken und die Sicherheit bei der Kommunikation gewährleisten. Dafür dienen zwei Konzepte:

Ersteres ist das Verschlüsseln der Kommunikation von Sender und Empfänger. Die zugrundeliegende Technik nennt sich Transport Layer Security (TLS), ist aber auch als Secure Sockets Layer (SSL) bekannt. Die Idee dahinter ist, dass jeder Teilnehmer der Kommunikation einen öffentlich bekannten Schlüssel (Public Key) und einen geheimen, nicht-öffentlichen Schlüssel (Private Key) besitzt. Über den Public-Key des Empfängers verschlüsselt der Sender seine Nachricht. Diese kann nur über den Private-Key des Empfängers entschlüsselt werden, der vom Empfänger nicht weitergegeben werden sollte.

Das zweite Konzept von HTTPS ist die Webserver-Authentifizierung. Ein Zertifikat, das zu Beginn der Kommunikation an den Webclient gesendet wird, bescheinigt die Vertrauenswürdigkeit des Servers. Dafür vertrauen Browser- und Betriebssystemhersteller bestimmten Zertifizierungsstellen, deren Zertifikate sie in ihrem Browser bzw. Betriebssystem hinterlegen. Die Kommunikation zwischen Webserver und Webclient findet folglich erst nach vollständiger Authentifizierung statt.

## 2.2 JavaScript

In den nächsten Unterkapiteln soll zunächst ein historischer Überblick über die Programmiersprache JavaScript gegeben werden. Im Anschluss wird auf die Bedeutung und Nutzung von JavaScript eingegangen.

### 2.2.1 Historie

Ihren Ursprung findet die Programmiersprache JavaScript im Jahr 1995, als Brendan Eich, ein damaliger Ingenieur des US-amerikanischen Software-Unternehmens „Netscape Communications Corporation“, innerhalb von zehn Tagen diese Sprache für den Browser „Netscape Navigator“ entwickelt hat [JS1]. Das Ziel dabei war es, eine Skriptsprache zu entwickeln, die es Entwicklern möglich machen sollte, auf ihren Webseiten Skripte umzusetzen. Zunächst noch unter dem Namen Mocha und LiveScript änderte sich der Name zu JavaScript aufgrund der Kooperation von Netscape und Sun, der Firma hinter der Programmiersprache Java, und damit verbundenen Marketinggründen. Netscape wollte von der damaligen Popularität von Java profitieren [JS2]. Netscape's Veröffentlichung des Netscape Navigator 2.0, der erste Browser der JavaScript unterstützte, brachte Microsoft dazu, Netscape als ernstzunehmenden Konkurrenten zu sehen. Microsoft antwortete im August 1995 mit der Veröffentlichung des ersten Internet Explorer zusammen mit der Skriptsprache JScript, die einen Dialekt der Sprache JavaScript darstellt. Dies ist ferner als der Beginn der „Browserkriege“ bekannt [JS3].

Im Jahre 1997 reichte Netscape JavaScript bei der European Computer Manufacturers Association (kurz ECMA), einer privaten, internationalen Normungsorganisation zur Normung von Informations- und Kommunikationssystemen und Unterhaltungselektronik, ein. Das Ziel war es, von der ECMA einen einheitlichen Standard für die Sprache schaffen zu lassen, die fortan weiterentwickelt werden und von weiteren Browserherstellern genutzt werden soll. Das resultierende Standard nennt sich ECMAScript, wobei JavaScript die bisher bekannteste Implementierung dieses Standards ist [JS4]. Andere Implementierungen sind zum Beispiel ActionScript von MacroMedia, JScript von Microsoft und ExtendScript von Adobe.

Jährlich wird dieser Standard seit Juni 2015 erweitert. ECMAScript Version 11 beziehungsweise ECMAScript 2020 bildet zum Zeitraum dieser Dokumentation den aktuellen Standard [JS5]. Im Juni 2021 soll die neueste Version ECMAScript 2021 veröffentlicht werden [JS6].

### 2.2.2 Wesentliche Programmereigenschaften

„JavaScript is Not Java“ [JS7]. Die Programmiersprache JavaScript wird aufgrund ihrer Namensgebung oft in falsche Zusammenhänge zu Java gebracht. Das häufigste Missverständnis sei, JavaScript wäre eine vereinfachte Version von Java [JS7].

JavaScript ist eine interpretierte Programmiersprache mit objektorientierten Umsetzungsmöglichkeiten. Interpretation ist in diesem Zusammenhang so zu verstehen, dass der Quellcode zur Laufzeit eines Programms gelesen, übersetzt und ausgeführt wird. Syntaktisch ähnelt JavaScript komplizierten Programmiersprachen wie C, C++ und Java durch gleiche Umsetzung der Kontrollstrukturen wie den Bedingungen, Schleifen oder den booleschen Operatoren [JS8]. Wesentliche Unterschiede sind dagegen, dass JavaScript zum einen eine schwach-typisierte Sprache ist. Durch die schwache Typisierung haben Variablen keinen festen Datentyp und können diesen dynamisch zur Laufzeit ändern. Des Weiteren findet bei JavaScript die Objektorientierung prototypenbasiert statt. Diese Form der Programmierung wird auch klassenlose Objektorientierung bezeichnet. Im Gegensatz zur klassenbasierten Programmierung, werden hier Objekte nicht aus vordefinierten Klassen instanziert, sondern durch das Klonen bereits existierender Objekte erzeugt. Die Objekte, die geklont werden, sind dabei als Prototyp-Objekte zu verstehen.



Abbildung 3: JavaScript Objekt [JS10]

Beim Klonen werden alle Attribute und Methoden des Prototyp-Objekts in das neue Objekt übernommen und können dort überschrieben sowie erweitert werden. Objekte in JavaScript sind eher als Zuordnungslisten, ähnlich wie assoziative Arrays oder Hash-Tabellen, anzusehen, da bei der Eigenschaftszuweisung lediglich ein Mapping eines Schlüsselworts (Key) zu seiner zugehörigen Eigenschaft (Value) stattfindet, wie es in Abbildung 3 zu sehen ist. Ein weiterer Unterschied zu den anderen Programmiersprachen ist, dass alle Funktionen und Variablen außer der primären Datentypen Boolean, Zahl und Zeichenfolge, als Objekte verstanden werden können.

Ein weiterer Unterschied zu den anderen Programmiersprachen ist, dass alle Funktionen und Variablen außer der primären Datentypen Boolean, Zahl und Zeichenfolge, als Objekte verstanden werden können.

### 2.2.3 Anwendungsgebiete

Ursprünglich fand JavaScript seinen Einsatz hauptsächlich darin, dynamische Webseiten im Webbrowser anzuzeigen. Die Verarbeitung erfolgte dabei meist clientseitig durch den Webbrowser (dem sogenannten Frontend) [JS11].

Heutzutage findet sich die Sprache dagegen in wesentlich größeren Einsatzgebieten wieder. Bis vor einigen Jahren war die Serverseite anderen Programmiersprachen wie Java oder PHP vorbehalten. Die Veröffentlichung von Node.js, einer plattformübergreifenden Laufzeitumgebung, die JavaScript außerhalb eines Webbrowsers ausführen kann, führte zu einer immer größeren Verbreitung von serverseitigen Anwendungen (dem Backend), die auf JavaScript basieren. Auf Node.js wird ausführlicher im nächsten Kapitel eingegangen. Ferner findet JavaScript heutzutage aber auch seinen Einsatz in mobilen Anwendungen, Desktopanwendungen, Spielen oder 3D-Anwendungen [JS12].

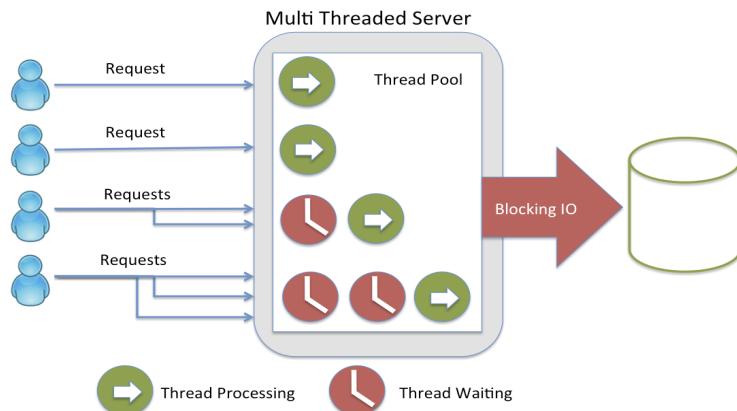


Abbildung 4: Multithreaded / Blocking I/O [Node2]

## 2.3 Node.JS

Im Jahr 2009 veröffentlichte Ryan Dahl das Framework Node.js, das auf Googles V8-Engine<sup>5</sup>, welche auch als JavaScript-Engine in Googles Browser Chrome zum Einsatz kommt, basiert und sich hervorragend für hochperformante, skalierbare und schnelle Webanwendungen eignet. Zudem ermöglicht es Webentwicklern die Entwicklung von serverseitigem JavaScript-Code.

### 2.3.1 Architektur

Eine wesentliche Eigenschaft von Node.js ist die hohe Performance. Im Folgenden soll der Unterschied der Node.js-Architektur zu traditionellen Webserversen und der damit verbundenen höheren Performance dargestellt werden.

Herkömmliche Webserver erstellten zunächst für jede ankommende Anfrage einen neuen Thread. Dieses Vorgehen ist eng mit steigendem Speicher- und Rechenaufwand verbunden. Um sich Rechenzeit zu sparen, die durch die Erstellung und Zerstörung von Threads entstand, wurden Threadpools eingerichtet. Dieser Threadpool enthält mehrere Threads, denen Aufgaben zugewiesen werden können. Nach erfolgreicher Abarbeitung einer Operation kann einem Thread eine weitere Aufgabe zugeordnet werden.

Es bleibt aber ein weiteres Problem: Bei der Anfragenabarbeitung kann es zu einer Form von blockierender Ein- und Ausgabe (Blocking Input/Output, kurz Blocking I/O) kommen: zum Beispiel beim Suchen in einer Datenbank oder dem Laden einer Datei im Dateisystem. Während der Abarbeitung wartet der Thread solange, bis die Operation ein Ergebnis zurückwirft und belegt dabei weiterhin Speicherplatz. Bei hohem Aufkommen von Anfragen kommt es dadurch zu einer hohen Speicherauslastung des Servers. Zudem kosten die Kontextwechsel zwischen den Threads im Betriebssystem weitere Rechenzeit [Node1]. Man spricht bei diesem Architekturkonzept auch vom Multi-Threaded Server. Graphisch dargestellt ist das Konzept in Abbildung 4.

<sup>5</sup><https://v8.dev/>, letzter Zugriff: 03. April 2021

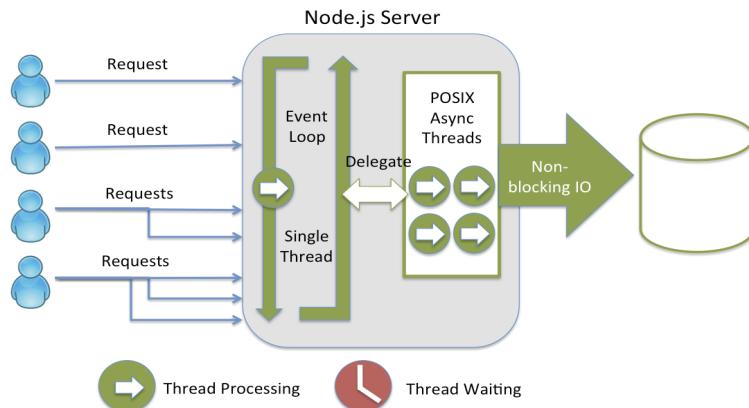


Abbildung 5: Single Threaded / Non Blocking I/O [Node2]

Node.js verfolgt einen anderen Ansatz: Wie in Abbildung 5 dargestellt, werden Anfragen nur in einem einzigen Thread, dem Hauptthread, abgearbeitet und in einer Warteschlange verwaltet. Dadurch bleiben Kontextwechsel zwischen Threads erspart. Hierbei handelt es sich also um einen Single-Threaded Server. Der Hauptthread verwaltet eine Schleife, die sogenannte Event Loop, die permanent Anfragen aus der Event-Warteschlange überprüft und Ereignisse, die von Ein- und Ausgangsoperationen ausgerufen werden, verarbeitet.

Bei Ankommen einer Nutzeranfrage an einen Node.js Server wird zunächst in der Event Loop geprüft, ob diese Anfrage Blocking I/O benötigt. Falls nicht, kann die Anfrage direkt bearbeitet werden und die Antwort an den Nutzer zurückgesendet werden.

Im anderen Fall wird einer von Node.js interner Workern, welche prinzipiell auch Threads sind, aufgerufen, um die jeweilige Operation auszuführen. Dabei wird eine Callback-Funktion mitgegeben, die vom Worker aufgerufen wird, sobald die Operation ausgeführt wurde. Diese Callback-Funktion kann anschließend als Ereignis von der Event Loop registriert werden. Man spricht hierbei auch von ereignisgesteuerter Architektur. [Node6]

Der große Vorteil hierbei ist, dass der Hauptthread trotz der blockierenden Ein- und Ausgabeoperationen nicht anhält, und weitere Anfragen bearbeiten kann. (Non Blocking I/O - Prinzip)

### 2.3.2 Module

Module stellen in Node.js Software-Komponenten dar, die Objekte und Funktionen nach außen hin bereitstellen sollen. Sie können aus einer Skriptdatei oder einem Verzeichnis von Dateien bestehen. Module können als einzelne Default-Komponenten, die den Hauptteil des Moduls repräsentiert, exportiert werden. Bei der anderen Möglichkeit, des sogenannten „benannten Exports“ werden die zu exportierenden Komponenten dagegen explizit angegeben. Letzteres ist im Codebeispiel 1 dargestellt.

```

1 function foo(){}
2 function bar(){}
3
4 //Obige Funktionen exportieren:
5 module.exports.foo = foo;
6 module.exports.bar = bar;
```

Listing 1: Benannter Export von Modulen

Für den Import stehen verschiedene Möglichkeiten zur Verfügung. Im Codebeispiel 2 ist ein Import über die require()-Funktion dargestellt. Mit mitgeliefertem Modul-Pfad als Parameter gibt diese Funktion ein Objekt des Moduls wieder, das die exportierten Objekte (und Funktionen) enthält.

```

1 //Importieren der Funktion einer anderen Datei:
2 const foo = require('./module/path');
3 const bar = require('./module/path');
```

Listing 2: Import von Modulen

Eine wichtige Besonderheit ist, dass importierte Module beim ersten Aufruf gecached werden. Das bedeutet, dass jeder require()-Aufruf innerhalb eines Programms auf ein Modul dasselbe Objekt zurückliefert [Node4].

#### 2.3.2.1 npm

Ehemals als Node Package Manager bekannt, ist npm ein Paketmanager für Node.js, entwickelt 2010 von Isaac Z. Schlueter [Node5]. Es verwaltet ein öffentliches Repository (ein digitales Software-Verzeichnis im Internet) unter dem Name npm Registry. In dem Verzeichnis werden weit über 1 Millionen Pakete (Module) angeboten [Node6]. Der Großteil kann unter freier Lizenz verwendet werden. Mit npm können Module installiert, aktualisiert, entfernt und gesucht werden. Node.js liefert seit seiner Version 0.6.3 npm standardmäßig bei der Installation mit [Node7].

#### 2.3.2.2 Express

„Express ist ein einfaches und flexibles Node.js-Framework von Webanwendungen, das zahlreiche leistungsfähige Features und Funktionen für Webanwendungen und mobile Anwendungen bereitstellt“ [Node8]. Es wurde im November 2010 von Douglas Christopher Wilson und weiteren Entwicklern veröffentlicht und erweitert Node.js unter anderem um das Abarbeiten verschiedener HTTP-Methoden, das separate Abarbeiten von Anfragen mit verschiedenen URL-Pfaden. Im Grunde handelt es sich bei Express um ein Modul, das durch den npm Package Manager heruntergeladen werden kann. Die aktuelle Version zum Zeitpunkt der Dokumentation ist 4.17.1

6.

## Beispiel

Das Erstellen einer einfachen Express-Applikation wird im folgenden Beispiel dargestellt:

```

1 const express = require('express');
2 const app = express();
3 const port = 3000;
4
5 app.get('/', (req, res) => {
6   res.send('Hello World')
7 });
8
9 app.listen(port, () => {
10   console.log("Example app listening on port ${port}!")
11 });

```

Listing 3: Einfacher Webserver<sup>7</sup>

Die require()-Funktion importiert das Express-Modul und gibt ein Express-Objekt zurück. Dieses Objekt als Funktion aufgerufen, gibt wiederum ein Objekt der Express-Applikation zurück, welche traditionell „app“ genannt wird, das Kernstück des Express-Frameworks ist und sämtliche Methoden wie das Weiterleiten von HTTP Anfragen, das Konfigurieren von Middleware oder das Modifizieren des Webserver-Verhaltens beinhaltet [Node9].

Im mittleren Block befindet sich eine Routendefinition. Die app.get() Funktion spezifiziert eine Callback-Funktion, die ein „request“- und „response“-Objekt als Parameter erhält und aufgerufen wird, sobald eine HTTP Anfrage der Methode GET mit dem Pfad ,/‘ empfangen wird. Das Request-Objekt enthält sämtliche Informationen über die HTTP-Anfrage. Das Response-Objekt kann dagegen in der Callback-Funktion mit Informationen gefüllt werden und über die send()-Funktion als HTTP-Antwort an den Sender zurückgesendet werden.

Der unterste Block startet den Webserver auf dem mitgegebenen Port über die Funktion app.listen(). Ihr kann auch eine Callback-Funktion mitgegeben werden, die aufgerufen wird, sobald der Server erfolgreich gestartet ist.

---

<sup>6</sup><https://www.npmjs.com/package/express>, letzter Zugriff: 04. April 2021

<sup>7</sup>Vergleiche [Node9]

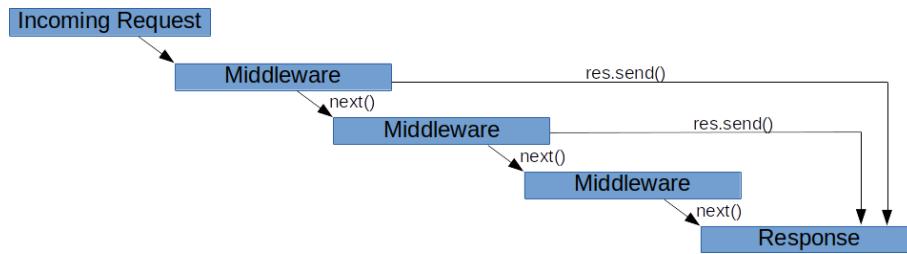


Abbildung 6: Middleware [Node3]

**Middleware:** Express arbeitet nach dem Middleware-Konzept. Darunter versteht man Funktionen, die für die Verarbeitung von Anfragen hintereinandergeschaltet werden können. Jede Middleware hat Zugriff auf das Anfrageobjekt, das Antwortobjekt und die jeweils nächste Middleware-Funktion [Node10]. Dabei kann die HTTP-Request direkt terminiert oder an die nächste Middleware gesendet werden. Die Verkettung der Middleware-Funktionen wird in Abbildung 6 illustriert.

**express.json:** Hierbei handelt es sich um eine in Express eingebaute Middleware, die die in JSON formatierten Daten im Nachrichtenrumpf aus einer eingehenden HTTP-Anfrage grammatisch analysiert. Dabei ist zu beachten, dass der Nachrichtenrumpf nur dann analysiert wird, wenn bei der Anfrage eine Header-Informationen namens „Content-Type“ mit dem entsprechenden JSON-Typ als Wert übergeben wird. Nach erfolgreicher Analyse erstellt die Middleware aus den JSON-Informationen eine neue body-Objekt innerhalb des übergebenen request-Objekts [Node12].

```

1 const express = require('express');
2 const app = express();
3 app.use(express.json());
  
```

Listing 4: Express.json Middleware benutzen

**Router:** Unter dem Begriff Routing (Weiterleitung) versteht man im Kontext von Express „[...] die Definition von Anwendungsendpunkten (URIs) und deren Antworten auf Clientanforderungen.“ [nodejs 2.15]

Die in Express eingebaute Middleware `express.Router` ermöglicht es, modular einbindbare Routenhandler (Weiterleitungsrouter) zu erstellen. Eine Router-Instanz ist als vollständiges Middleware- und Routingsystem zu sehen und wird deshalb auch als „Mini-App“ angesehen. Der Vorteil durch die Modularität ist, dass folglich unterschiedliche Anwendungsendpunkte auf entsprechende Dateien ausgelagert werden können.

```

1 var express = require('express');
2 var router = express.Router();
3
4 // Middleware explizit fuer diesen Router
5 router.use(function timeLog(req,res,next) {
6   console.log('Time: ', Date.now());
7   next();
8 });
9
10 // Homepage Route - Abhandlung
  
```

```
11 router.get('/', function(req,req){  
12   res.send('Birds home page');  
13 });  
14  
15 // About Route - Abhandlung  
16 router.get('/about', function(req,req){  
17   res.send('About birds');  
18 });  
19 module.exports = router;
```

Listing 5: Routinghandler erstellen <sup>8</sup>

In Beispiel 5 wird ein Routerhandler für das Verzeichnis „/birds“ mit eigen implementierter Middleware und zwei Anwendungsendpunkte „/“ (bezieht sich auf das Stammverzeichnis) und „/about“ erstellt. Der Code wird unter der Datei birds.js abgespeichert. Abschließend kann das Routermodul in die Anwendung geladen werden:

```
1 var birds = require('./birds');  
2 ..  
3 app.use('birds', birds);
```

Listing 6: Routinghandler benutzen

---

<sup>8</sup>Express, API-Dokumentation Router. <https://expressjs.com/en/api.html#router>, letzter Zugriff: 05. April 2021

### 2.3.2.3 Mongoose

Mongoose ist ein öffentliches Modul, das zum Zeitpunkt der Dokumentation im npm Package Manager in der Version 5.12.3 zur Verfügung steht<sup>9</sup>. Bei diesem Modul handelt es sich um ein Object-Document Mapper (ODM), der es ermöglicht, asynchron mit einer NoSQL-Datenbank (siehe Kapitel 2.4) zu kommunizieren. Mongoose ist der populärste und am weitesten von MongoDB unterstützte ODM [Node15]. Es unterstützt neben transparenter Persistenz auch die Datenvallidierung, das Erstellen von Abfragen (Queries), das Schreiben von logischem Business Code und die Übertragung zwischen Objekten im Code und der Repräsentation dieser Objekte in der Datenbank.

**Object Document Mapping (ODM):** Object-Relational Mappers (ORM) finden hauptsächlich Einsatz in objektorientierten Anwendungen, dessen Daten in relationalen Datenbanken sind. Dabei werden die Tabellen in persistente Objekte gemappt. Das Mappen ist aber auch für NoSQL-Datenbanken nützlich [Node16]. Die meistverbreiteten NoSQL-Datenbanken basieren auf Dokument-Systemen. Dementsprechend werden für diese Datenbanken Object-Document Mapper für das Mappen zwischen Dokumenten und Objekten genutzt. Einige ODM's sind Mongoose<sup>10</sup>, Morphia<sup>11</sup>, Doctrine<sup>12</sup> und Mandango<sup>13</sup>. NoSQL Mapper nutzen vom Entwickler definierte Datenschemata, die das Objekt beschreiben. Ein daraus abgeleitetes Model-Objekt ermöglicht dann die Kommunikation zwischen dem im Schema beschriebenen Objekt und der entsprechenden Datenbank-Collection.

**Schema:** Mongoose-Schemata definieren die Struktur der gespeicherten Daten einer MongoDB-Collection in der Anwendungsschicht und werden in der JSON-Notation beschrieben. Dokumentenbasierte Datenbanken wie MongoDB enthalten für jede Wurzelentität eine Collection. Mongoose Schemata werden für jede Collection definiert. Innerhalb der JSON-notierten Schemabeschreibung können den einzelnen Eigenschaften bestimmtes Verhalten zugeordnet werden. Zum Beispiel lässt sich explizit der Datentyp angeben (type), eine Eigenschaft verpflichtend (required) oder in Kleinbuchstaben einstellen (lowercase).

```

1 const schema = new Schema({
2   attributeX: {
3     type: String, // Datentyp
4     required: true, // Verpflichtendes Attribut?
5     lowercase: true; // Kleinbuchstaben?
6 });

```

Listing 7: Mongoose Schema - Beispiel

---

<sup>9</sup>npm mongoose, <https://www.npmjs.com/package/mongoose>, letzter Zugriff: 05. April 2021

<sup>10</sup>Mongoose Webpage, <http://mongoosejs.com>, letzter Zugriff: 04. April 2021

<sup>11</sup>Morphia Webpage, <https://github.com/mongodb/morphia>, letzter Zugriff: 04. April 2021

<sup>12</sup>Doctrine Project Webpage, <http://www.doctrine-project.org/>, letzter Zugriff 04.04.2021

<sup>13</sup>Mandango Webpage, <https://mandango.readthedocs.io/en/latest/>, letzter Zugriff: 04. April 2021

**Model:** Ein Model in Mongoose ist ein aus einer Schemadefinition erstellter Konstruktor, aus denen Objekte instanziert werden können. Diese Instanzen werden auch „documents“ genannt. Sie stehen in direkter Verbindung zu den jeweiligen Collections der verbundenen Datenbank und enthalten Methoden für die persistente Speicherung, Bearbeitung oder Löschung. Beispielsweise wird beim Abspeichern einer Mongoose Instanz eines Models die entsprechende Collection in der Datenbank erzeugt, sofern sie noch nicht vorhanden ist. Eine Konvention in Mongoose sieht vor, dass der Name eines Models dem Singular eines Nomens entspricht, während die Collections nach dem Plural dieses Namens beschrieben werden [Node18]. Im Codebeispiel 8 wird ein Model über die Funktion `mongoose.model()` erstellt unter Angabe des Modelnamens und dem zu verwendenden Schema. Dieses Model wird über `module.exports` nach außen zur Verfügung gestellt.

```

1 const mongoose = require('mongoose');
2 const testSchema = new mongoose.Schema({
3   attributeX: {
4     type: String,
5     required: true,
6     lowercase: true
7   }
8 });
9 module.exports = mongoose.model('test', testSchema);

```

Listing 8: Model erstellen und exportieren

An anderer Stelle kann das Model nun importiert werden. Aus dem Model kann ein Objekt instanziert werden, welches über die Funktion `save()` in der Datenbank gespeichert werden kann.

```

1 const testModel = require(test);
2
3 var testInstanz = new testModel();
4 await testInstanz.save();

```

Listing 9: Model importieren - Objekt instanziieren und persistent speichern

Mongoose Models enthalten ohne Instanziierung des Weiteren auch Schnittstellen, um Daten der zugehörigen Collection zu kreieren, abzufragen, zu bearbeiten oder zu löschen. (Create, Receive, Update, Delete oder auch kurz CRUD).

```

1 const testModel = require(test);
2
3 //Create
4 testModel.Insert({attributeX: "abc"})
5 //Receive
6 var testObjects = await testModel.find();
7 var testObject = await testModel.findOne({attributeX: "abc"})
8 //Update
9 await testModel.updateOne({X: "abc"}, {X: "cba"});
10 //Delete
11 await testModel.deleteMany({X: "abc"})

```

Listing 10: CRUD-Beispielfunktionen eines Mongoose-Models

**Verbindung:** Verbindung zur Datenbank kann über die `connect()`-Funktion mit Angabe der genutzten Datenbank und des Datenbankpfads hergestellt werden. Über das Objekt `mongoose.connection` können auf Verbindungsereignisse reagiert werden.

```
1 const mongoose = require('mongoose');
2 await mongoose.connect("mongodb://127.0.0.1:27017/TestDB");
3 mongoose.connection.on('error',(error) => console.log(error));
4 mongoose.connection.on('open',() => console.log('Connected'));
```

Listing 11: Mongoose: Verbindung zur Datenbank aufbauen

Für den Verbindungsauflauf können weitere Optionen übergeben werden. Dafür kann ein Objekt wie in Beispiel 12 erstellt werden, dass die zugehörigen Optionen als Attribute beinhaltet.

```
1 const options = {
2   useNewUrlParser: true,
3   useUnifiedTopology: true,
4   useCreateIndex: true,
5   autoIndex: false,
6   poolSize: 10, // Anzahl der max. Socket Connections
7   serverSelectionTimeoutMS: 5000, // TimeOut bis verbunden
8   socketTimeoutMS: 45000, // Schliesse Socket bei 45s
9   family:4 // Use IPv4
10 }
```

Listing 12: Mongoose Verbindungsoptionen<sup>14</sup>

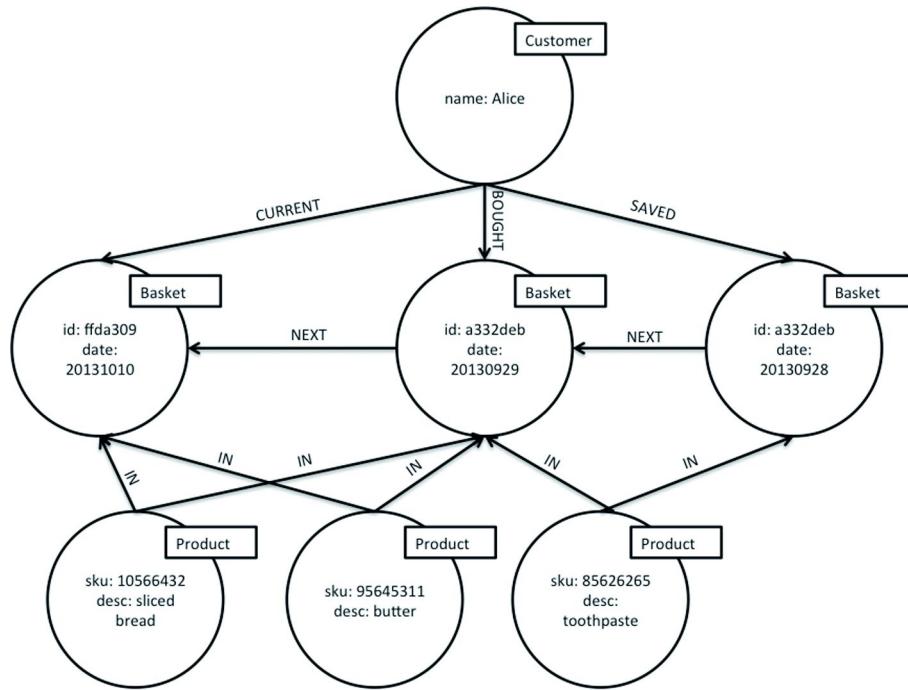
#### 2.3.2.4 Weitere Module

Weitere relevante Module werden in Tabelle 3 beschrieben.

Tabelle 3: Express.js Module

Express-Modul	Beschreibung
fs	Erlaubt die Interaktion mit dem Dateisystem. Zum Beispiel Schreiben/Lesen von Dateien.
http	Ermöglicht Datentransfer über das Protokol HTTP und das Abhören eines Ports.
https	Gesicherte Variante zu HTTP mit SSL. Benötigt Private Key und Zertifikat.
firebase-admin	Ermöglicht die Verbindung zu Google Firebase Cloud.
node-cron	Ermöglicht das Einstellen von sich wiederholenden Aufgaben zu bestimmten Zeitintervallen.

<sup>14</sup>Mongoose Connections, <https://mongoosejs.com/docs/connections.html>, letzter Zugriff: 05. April 2021

Abbildung 7: Graphendatenbank Beispiel<sup>15</sup>

## 2.4 NoSQL-Datenbank

Unter NoSQL („Not only SQL“) werden Datenbanksysteme bezeichnet, die einen nicht-relationalen Ansatz verfolgen. Im Vergleich zu relationalen Datenbanken, welche die Daten in tabellenförmigen Strukturen mit Spalten und Zeilen speichern, nutzt eine NoSQL-Datenbank andere Strukturkonzepte für die Speicherung der Daten wie zum Beispiel Wertpaare, Dokumente, Objekte oder Listen und Reihen. Da NoSQL einige der bekannten Schwächen von relationalen Datenbanken, wie Performance-Schwierigkeiten bei hohem Lastaufkommen oder bei dem Umgang mit großen Datenmengen, vermeidet, erfreut sich diese Technologie in der heutigen Zeit des großen Datenaufkommens immer größerer Beliebtheit [DB1]. Zu den bekanntesten NoSQL-Datenbanken gehören beispielsweise Apache Cassandra, MongoDB und CouchDB.

### 2.4.1 NoSQL-Datenbanktypen

NoSQL-Datenbanken werden hauptsächlich in vier verschiedene Kategorien unterteilt, die unterschiedliche Konzepte verfolgen.

**Graphendatenbanken:** Dieses Datenbankkonzept speichert die Informationen in Netzstrukturen, den sogenannten Graphen ab. Die einzelnen Informationselemente werden durch Knoten mit Eigenschaften repräsentiert. Um die Beziehungen zwischen den Knoten darzustellen, werden Kanten genutzt, die gerichtet und benannt sein können und ebenfalls Eigenschaften besitzen. Das Konzept wird in Abbildung 7 illustriert.

**Dokumentenorientierte Datenbanken:** Im Kontext der dokumentenorientierten Datenbanken sind Dokumente Objekte mit Eigenschaften, die in einer Sammlung gespeichert werden. Während eine Sammlung (Collection) eine Tabelle im relationalen Datenbank wi-

<sup>15</sup><https://entwickler.de/online/datenbanken/wunderbare-welt-der-graphen-114728.html>, letzter Zugriff: 9. Februar 2021

derspiegelt, ist ein Dokument als ein Eintrag beziehungsweise einer Zeile dieser Tabelle gleichzusetzen, mit dem großen Unterschied, dass dokumentenorientierte Datenbanken schemafrei sind und kein bestimmtes Datenschema voraussetzen. Dokumente können weitere Dokumente als Attribut oder in einer Liste einbetten und bieten so die Möglichkeit, komplexe Datenstrukturen zu speichern. In aktuellen Datenbanksystemen wie CouchDB und MongoDB nutzen diese Dokumente Datenformate wie JSON oder XML.

```

1 {
2   "Vorname": "Max",
3   "Nachname": "Mustermann",
4   "Telefon-Nr": "0124567",
5   "Alter": 33,
6   "Adresse": "Musterstrasse 34, Musterstadt",
7   "Kinder": ["Junior", "Elenor"]
8 }
9 \caption{Inhalt eines dokumentenorientieren Datensatzes}

```

**Key-Value-Datenbanken:** Key-Value-Datenbanksysteme bilden mit einer Abbildung der Daten in Schlüssel- und Wertpaare die einfachste NoSQL-Datenbankumsetzung ab. Bei diesem Konzept werden eindeutigen Schlüsselattributen (Key) jeweils ein beliebiger Wert (Value) zugeordnet.

**Spaltenorientierte Datenbanken:** Spaltenorientierte Datenbanken, oder auch „Wide-Column“-Datenbanken genannt, speichern ihre Datensätze in Form von Tabellen. Sie wirken zunächst den Tabellen der relationalen Datenbanksysteme sehr ähnlich, unterscheiden sich grundlegend aber in der Speicherung der Daten, die nicht zeilenorientiert, sondern spaltenorientiert abgelegt werden. Die zeilen- und die spaltenorientierte Speicherung ist in Abbildung 8 dargestellt. Zeilenorientierung bei der Speicherung bietet vor allem Vorteile bei Abfragen, bei denen Informationen aus mehreren Spalten benötigt werden. Spaltenorientierung dagegen eignet sich sehr gut für die Auswertung von Aggregaten.

Artikelnummer	Artikelbezeichnung	Umsatz Tsd.EUR
1	Buntlack RAL 7035	25
2	Heizkörperfarbe	50
3	Grundierfarbe	15

(a)

zeilenorientierte Speicherung

1	Buntlack RAL 7035	25	2	Heizkörperfarbe	50	3	Grundierfarbe	15
---	-------------------	----	---	-----------------	----	---	---------------	----

spaltenorientierte Speicherung

1	2	3	Buntlack RAL 7035	Heizkörperfarbe	Grundierfarbe	25	50	15
---	---	---	-------------------	-----------------	---------------	----	----	----

(b)

Abbildung 8: (a) Spaltenorientierte Datenbank Beispiel (b) Zeilen- und spaltenorientierte Speicherung<sup>16</sup>

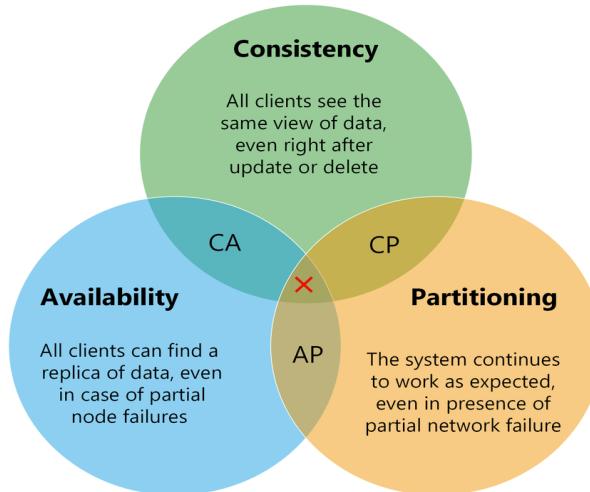


Abbildung 9: Visualization-of-CAP-theorem<sup>17</sup>

#### 2.4.2 BASE

TODO

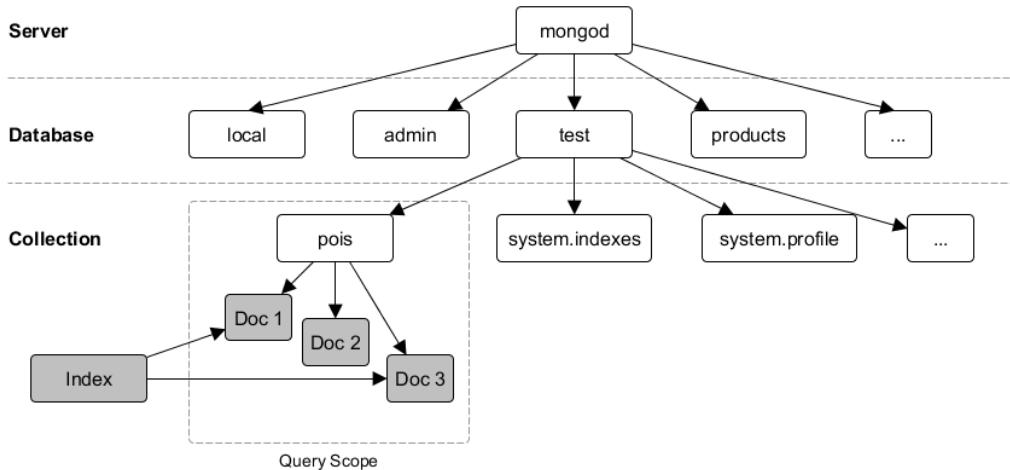
#### 2.4.3 CAP-Theorem

Der Informatiker Eric Brewer von der Universität Berkeley stellte Anfang des Jahres 2000 die Annahme auf, dass ein System nicht gleichzeitig die drei Kerneigenschaften Consistency (Konsistenz), Availability (Verfügbarkeit) und Partition Tolerance (Ausfalltoleranz) abdecken kann [DB2]. Diese Annahme wird in Abbildung 9 dargestellt.

Die **Konsistenz (C)** beschreibt, dass Datenzustände, die in einem verteilten System geändert werden, in jedem zusammenhängenden System gleich sein müssen. Der Zustand der Daten soll somit im gesamten System übereinstimmen. Ein System, das einen ununterbrochenen Betrieb und eine akzeptable Antwortzeit aufweisen kann, besitzt eine hohe **Verfügbarkeit (A)**. Die **Ausfalltoleranz (P)** steht für ein Verhalten, bei dem es bei Ausfallen eines Bestandteiles innerhalb eines Systems zu keinem Gesamtausfall kommt.

<sup>16</sup><https://www.axiom-net.de/die-datenbank-hana-ist-spaltenorientiert>, letzter Zugriff: 9. Februar 2021

<sup>17</sup><https://www.researchgate.net/profile/Hamzeh-Khazaei/publication/282679529/figure/fig2/AS:614316814372880@1523475950595/Visualization-of-CAP-theorem.png>, letzter Zugriff: 10. Februar 2021

Abbildung 10: MongoDB Architektur<sup>18</sup>

## 2.4.4 MongoDB

MongoDB ist ein in C++ geschriebenes, dokumentenorientiertes NoSQL-Datenbanksystem, das im Jahre 2009 von den Entwicklern Horowitz und Merriman als Open-Source Datenbank veröffentlicht wurde und die am weitest-verbreitete NoSQL-Datenbank ist (Stand April 2021) [DB4]. Die Intention der Gründer war es, eine Datenbank mit höherer Skalierbarkeit, Flexibilität und Performance zu entwerfen, die auf einer einfachen Handhabung beruht [DB3]. Gründe der Popularität der Datenbank ist neben den oben erwähnten Eigenschaften die flexible Gestaltungsmöglichkeit der Datenstrukturen sowie die Unterstützung durch zahlreiche Programmiersprachen und Betriebssysteme. Dem Konzept des CAP-Theorems folgend steht MongoDB für Konsistenz und Partitionstoleranz, dafür ordnet sich die Verfügbarkeit den anderen Eigenschaften unter.

### 2.4.4.1 Struktur

Die Abbildung 10 stellt die grundsätzliche Struktur einer MongoDB-Instanz dar. Ein **Server** kann mehrere logische **Datenbanken** verwalten, die ihrerseits einen oder mehrere logische Namensräume enthalten, die sogenannten **Collections**. Eine Collection verwaltet die einzelnen Datensätze, die als **Dokumente** bekannt sind.

**Schema-Freiheit:** Collections sind schemafrei. Dadurch gibt es für die zugehörigen Dokumente kein vorausgesetztes Schema. Aufgrund der Schema-Freiheit dürfen dennoch Architekturentscheidungen bei der Datenmodellierung nicht ignoriert werden. Stattdessen werden diese Entscheidungen des Schema-Managements auf die Anwendungsentwicklung verlagert.

**BSON:** Während MongoDB für Datenaustausch das JSON-Format nutzt, hält es seine Dokumente im Binary JSON-Format (BSON), einer binärcodierter Erweiterung des JSON-Formats. Daten im BSON-Format enthalten zusätzlich Informationen zum Typ und zur Länge der Informationen, wodurch schnelleres Parsen von Daten möglich ist. Des Weiteren ist BSON um zusätzliche Datentypen wie 32- und 64-bit Integer oder das Datum erweitert<sup>19</sup>.

<sup>18</sup><https://www.informatik-aktuell.de/betrieb/datenbanken/mongodb-fuer-software-entwickler.html>, letzter Zugriff: 9. Februar 2021

<sup>19</sup><https://www.mongodb.com/json-and-bson>], letzter Zugriff: 12. Februar 2021

```

1 /* JSON */
2 {"hello": "world"} // "key":"value"
3
4 /* BSON */
5 \x16\x00\x00\x00          // Gesamtgroesse des Dokuments
6 \x02                      // 0x02 = Typ String
7 hello\x00                  // Feldname
8 \x06\x00\x00\x00world\x00  // Feldwert
9 \x00                      // 0x00 = Typ E00 ('end of object')

```

Listing 13: JSON - BSON Vergleich

**ObjectID und Primärschlüssel:** Für die eindeutige Identifikation eines Dokuments vergibt MongoDB automatisch erstellte „\_id“-Felder. Dabei wird bei der Generierung des BSON-Dokuments für den Wert das „\_id“-Feld ein Datentyp „ObjectId“ hinzugefügt. Dieses Identifikationsfeld ist gleichzusetzen mit den Primärschlüsseln aus relationalen Datenbanken [DB6]. Ist eine automatische Generierung des eindeutigen Identifikationsfeldes nicht erwünscht, kann der Anwendungsentwickler dieses Feld auch selbst erzeugen, indem er die Eigenschaft explizit dem zu erstellenden Objekt hinzufügt und mit einem Wert verseht.

#### 2.4.4.2 Datenbankabfrage

Im Vergleich zu relationalen Datenbanken benutzt MongoDB keine Abfragesprache wie SQL. Stattdessen ermöglicht dieses Datenbanksystem drei Arten von Abfragen, wobei eine Abfrage sich immer auf genau eine Collection bezieht. Ein Bezug einer Abfrage zu mehreren Collections, wie es relationale Datenbanken mit „Join“-Operationen erlauben, ist hier nicht gegeben und muss in der Anwendung realisiert werden [?]. Abfragen in MongoDB werden grundsätzlich in Form von Dokumenten formuliert. Dieses Verfahren ermöglicht schnelles Abhandeln komplexer Abfragen von tief geschachtelten Dokumentenstrukturen. Nachfolgend werden die drei Abfragemöglichkeiten erläutert.

**Query-By-Example:** Das folgende Beispiel zeigt einen Aufruf aller Einträge, die als Ort Karlsruhe eingespeichert haben.

```
1 db.pois.find( {"adresse.ort": "Karlsruhe" } )
```

Listing 14: MongoDB Read

Dabei kommt das Prinzip Query-by-Example<sup>20</sup> zum Einsatz, bei dem die als JSON-Dokument beschriebenen Suchkriterien als Filter auf die durchsuchte Collection wirkt. Zusätzlich stehen für diese Art von Abfragemöglichkeit sämtliche logische Verknüpfungen und Vergleichsoperatoren zur Verfügung<sup>21</sup>. Das Ergebnis der obigen `find()` Operation liefert dabei einen Cursor zurück, über den die aufrufende Anwendung durch die einzelnen zurückgelieferten Dokumente iterieren kann. Außerdem kann der Cursor modifiziert werden, um beispielsweise Beschränkungen der Trefferanzahl oder Sortierung vorzunehmen.

```
1 db.pois.find().limit(5).sort({ "adresse.ort": -1 })
```

Listing 15: MongoDB Read Modifikation

<sup>20</sup>Query By Example: [https://de.wikipedia.org/wiki/Query\\_by\\_Example](https://de.wikipedia.org/wiki/Query_by_Example), letzter Zugriff: 12. Februar 2021

<sup>21</sup>Query-Operatoren <https://docs.mongodb.com/manual/reference/operator/query/>, letzter Zugriff: 12. Februar 2021

**MapReduce:** MapReduce ist ein allgemeines Programmiermodell zur verteilten und parallelen Verarbeitung von großen Datenmengen in aggregierte Ergebnisse. Dabei unterteilt sich der Algorithmus im Wesentlichen in zwei Operationen<sup>22</sup>:

- Map: Emittieren der beliebig vielen Key-Value-Paare für jedes Dokument
- Reduce: Zusammenfassen aller Daten, die einem Kriterium entsprechen.

**Aggregation-Framework:** Das Aggregation-Framework bietet als Alternative zum MapReduce-Verfahren den Vorteil der besseren Performance<sup>23</sup>. Dabei wird eine Pipeline genutzt, die das resultierende Dokument einer Operation an die Eingabe der nächsten Operation weiterleitet. Die entsprechende Funktion ist die `Aggregate()`-Operation, deren Parameter als Array von Dokumenten die Pipeline-Operatoren abbilden. Tabelle 4 zeigt die zur Verfügung stehenden Pipeline-Operatoren.

```
1 db.pois.aggregate([
2   {$group: {_id: "$adresse.ort", n: {$sum:1}}},
3   {$sort: {n: -1}}
4 ])
```

Listing 16: MongoDB Aggregate

#### 2.4.4.3 CRUD

MongoDB unterstützt eine Vielzahl an Operationen zum Erzeugen, Lesen, Updaten und Löschen von Daten. Tabelle 5 stellt die Kommandos zur Realisierung der CRUD-Operationen in MongoDB dar.

**Create:** `insertOne()` erzeugt ein einzelnes Dokument in der jeweiligen Collection der Datenbank. Dabei wird ein Dokument im JSON-Format als Parameter mitgegeben. ObjectId wird, wenn nicht als Eigenschaft im Dokument angegeben, automatisch von MongoDB erzeugt. Ebenfalls wird die angesprochene Collection automatisch erzeugt, falls sie nicht bereits existiert. Die Funktion `insertMany()` erlaubt das Hinzufügen mehrerer Datensätze über ein Array von JSON-Dokumenten als Parameter. Die Methode `insert` ist flexibler und bietet beide Parametrisierungsmöglichkeiten<sup>24</sup>.

```
1 db.personen.insertOne({ "vorname" : "Robin"});
```

Listing 17: MongoDB Create

**Read:** Um Datensätze aus der Datenbank zu lesen, wird die Methode `find()` zur Verfügung gestellt. Sie gibt alle Dokumente einer Sammlung zurück. Wie im Kapitel Query-By-Example beschrieben, kann die Funktion mit Filtern versehen werden. Die Funktion `findOne()` bietet die gleiche Funktionsweise, die Dokumentenausgabe ist aber auf ein einzelnes Dokument begrenzt. Ein Beispiel ist dargestellt in Listing 14.

<sup>22</sup>MapReduce Dokumentation <https://docs.mongodb.com/manual/core/map-reduce/>, letzter Zugriff: 10. Februar 2021

<sup>23</sup><https://docs.mongodb.com/manual/meta/aggregation-quick-reference/>, letzter Zugriff: 14. Februar 2021

<sup>23</sup><https://www.informatik-aktuell.de/betrieb/datenbanken/mongodb-fuer-software-entwickler.html>, letzter Zugriff: 9. Februar 2021

<sup>24</sup><https://docs.mongodb.com/manual/reference/method/db.collection.insert/#mongodb-method-db.collection.insert>, letzter Zugriff: 14. Februar 2021

Tabelle 4: Aggregation Framework: Pipeline-Operatoren [MongoDB1.9]

Operator	Beschreibung
\$match	Sucht Dokumente analog zu find(). Sollte idealerweise mindesten 1x zu Beginn der Pipeline ausgeführt werden, um die Ergebnismenge einzuschränken.
\$project	Schränkt auf eine Teilmenge von Feldern ein und verändert die Feldwerte.
\$sort	Sortiert die Dokumente. Analog zu sort() bei find().. Benötigt Private Key und Zertifikat.
\$skip	Überspringt n Dokumente. Analog zu skip() bei find().
\$limit	Begrenzt auf n Dokumente. Analog zu limit() bei find().
\$group	Gruppert nach einem oder mehreren Feldern.
\$unwind	Wird auf ein Array angewendet. Jeder Array-Eintrag generiert dann ein neues Dokument für die nächste Pipeline-Stufe.
\$redact	Filtert Felder des Dokuments in Abhängigkeit vom Inhalte anderer Felder.
\$out	Leitet das Ergebnis der Aggregation in eine Collection um. Kann nur als letzter Operator verwendet werden.

**Update:** Die update Methode ermöglicht es, Änderungen an Datensätzen vorzunehmen. Die Methode erhält zwei Parameter, einen zur Angabe der gesuchten Dokumente und den anderen mit der vorzunehmenden Änderung. Simultan zu den Insert-Methoden gibt es die updateOne() Methode für Änderungen an einem Dokument und updateMany() Methode für mehrere Dokumente.

```
1 db.personen.updateOne({ "vorname" : "Robin" } , currentDate("last_login") );
```

Listing 18: MongoDB Update

Tabelle 5: CRUD-Operatoren

Operation	Beschreibung	MongoDB Methode
Create	Datensätze erstellen	.Insert()
Read	Datensätze lesen	.Find()
Update	Daten aktualisieren	.Update()
Delete	Datensätze entfernen	.Remove()

**Delete:** Für das Löschen von Datensätzen aus einer Collection gibt es die `deleteOne()` und `deleteMany()` Methoden. Über mitgegebenen Parameter können Filter eingestellt werden. Eine ganze Collection kann mit der `drop()` Methode entfernt werden.

```
1 db.personen.deleteOne( {"vorname" : "Robin"} );
```

Listing 19: MongoDB Remove

#### 2.4.4.4 Atomare Operationen

Als atomare Operationen wird ein Verbund von Einzeloperationen bezeichnet, der als logische Einheit betrachtet wird und nur als Ganzes erfolgreich abläuft oder fehlschlägt. In Bezug auf Datenbanken spricht man von Transaktionen, die entweder als Ganzes erfolgreich ablaufen (Commit) oder nach einer fehlerhaften Einzeloperation rückgängig gemacht werden (Rollback). Ohne atomare Operationen können Probleme auftreten, die zu einer Inkonsistenz der Datenzustände führt. Beispielsweise würde ein Abbruch inmitten mehrerer zusammenhängender Operationsabläufe dazu führen, dass nur ein Teil der Operationen ausgeführt wurde, während die restlichen Operationen und somit die verbleibenden Datenänderungen verworfen werden.

Bis vor dem Jahre 2018 unterstützte MongoDB keine Transaktionen. Mit der Veröffentlichung der Version 4.0 wurde der Funktionsumfang von MongoDB stark erweitert. Eines der neuen Erweiterungen war die Möglichkeit, Transaktionen durchführen zu können. Dafür werden sogenannte „Sessions“ genutzt. Sämtlichen CRUD-Operationen, die einer Transaktion zugehören sollen, werden eine Session als zusätzlicher Parameter hinzugefügt. Bei Abschluss der Transaktion kann sie mit der Methode `session.commitTransaction()` bestätigt werden. Andernfalls, sollte ein Fehler aufgetreten sein, kann die Methode `session.abortTransaction()` die Operationen rückgängig machen. Eine wichtige Voraussetzung, um Transaktionen in MongoDB nutzen zu können, ist ein Replica Set aufzusetzen. Darauf wird im weiteren Kapitel eingegangen.

#### 2.4.4.5 Architektur

Bei der Inbetriebnahme einer MongoDB Datenbank spielen zwei Prozesse eine wichtige Rolle. Der „mongod“-Prozess ist der primäre Hintergrundprozess für das Datenbanksystem. Er behandelt Datenabfragen, Datenzugriffe und führt benötigte Hintergrundoperationen durch<sup>25</sup>. Beim Starten des „mongod“-Prozesses können über Flags Konfigurationen vorgenommen werden, beispielsweise ändert „–port 5000“ den Port. Nach erfolgreichem Start des Prozesses kann über den Standard Port der MongoDB (27017) bzw. über den konfigurierten Port eine Verbindung hergestellt werden.

Der Prozess, der als Controller für sich auf mehreren Datenbanken befindenden verteilten Daten dient, nennt sich „mongos“<sup>26</sup>. Dieser Prozess findet seinen Einsatz hauptsächlich in Kombination mit Sharding, auf die im weiteren Unterkapitel eingegangen wird.

**Engine:** Als Storage Engine, oder auch Datenbank-Engine, wird die zugrundeliegende Softwarekomponente eines Datenbanksystems zum Verwalten der Daten bezeichnet. Die gebräuchlichsten Engines in MongoDB sind die MMAPv1-Engine und die WiredTiger-Engine.

Bis vor Version 3.2 war MMAPv1-Engine die Standard-Storage Engine von MongoDB. Eine Kompression der Daten wurde nicht unterstützt und Transaktionen waren nur auf einem Dokument ausführbar. Ab MongoDB 3.2 wird standardmäßig die WiredTiger-Engine eingesetzt. Neben der Kompression der Daten und der einhergehenden Verringerung des

<sup>25</sup>Mongod <https://docs.mongodb.com/manual/reference/program/mongod/>, letzter Zugriff: 15. Februar 2021

<sup>26</sup>Mongos <https://docs.mongodb.com/manual/reference/program/mongos/>, letzter Zugriff: 15. Februar 2021

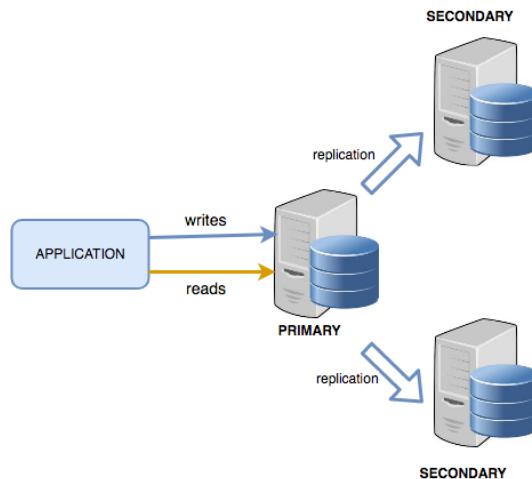


Abbildung 11: MongoDB Replica Set

benötigten Speicherplatzes bietet diese Engine auch eine Verschlüsselung der Daten an. Abhängig des benutzten Kompressionsalgorithmus kann der benötigte Verbrauch um 70% für Daten und um 50% für Indizes reduziert werden [DB8]. Die WiredTiger-Engine skaliert im Vergleich zur MMAPv1 mit der Anzahl der CPU-Kerne und ermöglicht Transaktionen über mehrere Dokumente hinweg. Die genutzte Engine kann in den Konfigurationen eingestellt werden.

**Replica Sets:** Im standardmäßigen Standalone-Modus besteht bei Ausfall des Servers die potenzielle Gefahr des Datenverlusts. Das Problem lässt sich durch Replikation beheben. Hierbei spricht man von der bloßen Herstellung von Mehrexemplaren (Kopien) derselben Daten, die meistens regelmäßig abgeglichen werden [DB9]. MongoDB realisiert die Replikation der Daten über Replica Sets [DB10]. Dabei handelt es sich um eine Gruppe von „mongod“-Prozessen, die dieselben Daten enthalten. Nach dem CAP-Theorem ist MongoDB nicht auf Verfügbarkeit ausgelegt. Dennoch umgeht die Datenbank diesen Nachteil über die Anwendung von Replica Sets. Das Replica Set besteht aus einem primären Knoten und daraus replizierenden sekundären Knoten. Nur der primäre Knoten führt Datensatzändernde Operationen aus. Veränderungen werden in sogenannten 'Oplogs' (Operations logs) gespeichert, die zum Austausch der Daten innerhalb des Replica Sets genutzt wird [DB11]. Bei den Oplog-Dateien handelt es sich um Collections mit festgelegten Speichergrößen, den sogenannten „capped collections“. Während die Oplog-Dateien ähnlich wie ein Logbuch mit allen Veränderungen in den Datensätzen der Datenbank beschrieben werden, werden bei erreichter Maximalgröße der Collection die ältesten Daten von den neuen Daten überschrieben. Dabei enthält jeder Knoten des Replica Sets seine eigene Kopie des Oplogs und gleicht ihn mit dem des primären Knotens ab. Das Konzept der Replica Sets wird in Abbildung 11 dargestellt.

**Sharding:** Hierbei ist die Rede von der Methodik zur Aufteilung der Daten auf mehrere Datenbanken [DB12]. Der Datenbestand wird nach logischen Kriterien in Teilstücke, die sogenannten „Shards“, zerlegt und über mehrere Replica Sets verteilt. In Hinblick auf das Hinzufügen weiterer Server gewährt das Sharding eine horizontale Skalierbarkeit. Des Weiteren werden durch die Tatsache, dass jeder Shard für seinen Bestandteil zuständig ist, Suchanfragen schneller durchgeführt. Sharding ist besonders bei großen Datenmen-

<sup>26</sup>Bild aus [DB11]

The screenshot shows the MongoDB Compass interface for the 'test.books' collection. At the top, it displays 'DOCUMENTS 5' and 'INDEXES 1' with their respective sizes. Below this is a toolbar with 'FILTER', 'ADD DATA', 'VIEW', and search/refresh buttons. The main area lists three documents:

- Document 1: \_id: 7000, author: "Homer", copies: 10
- Document 2: \_id: 7020, title: "Iliad", author: "Homer", copies: 10
- Document 3: \_id: 8645, title: "Eclogues", author: "Dante", copies: 2

At the bottom right, it says 'Displaying documents 1 - 5 of 5'.

Abbildung 12: MongoDBCompass Benutzeroberfläche

gen in Bezug auf Performance von Vorteil. Der „mongos“-Prozess leitet Anfragen an die jeweiligen Shards weiter und dient als Schnittstelle zwischen den Clientanwendungen.

#### 2.4.4.6 Verwaltungswerkzeuge

**Mongo Shell:** Der „mongo“-Prozess ist eine interaktive JavaScript Schnittstelle auf der Kommandozeile. Er bietet umfassende Funktionalitäten für die Systemadministration des Datenbankensystems als auch Zugriff auf die Datensätze<sup>27</sup>. Dazu erhält man eine Eingabeaufforderung, auf dem Befehle in der Sprache JavaScript ausgeführt werden können.

**Treiber:** MongoDB bietet für viele Programmiersprachen bzw. Frameworks Softwarebibliotheken zum Zugriff auf die Datenbank an.<sup>28</sup> Eine offiziell unterstützte ist Mongoose für das Node.js-Framework<sup>29</sup>

**Grafische Oberflächen:** Es gibt einige Anwendungen zur visuellen Darstellung und Bearbeitung der Datenbanken in MongoDB, die eine grafische Benutzeroberfläche bieten. Zum Beispiel MongoDB Compass, Studio3T oder Fang of Mongo. In Abbildung 12 wird die Benutzeroberfläche von MongoDBCompass dargestellt

<sup>27</sup> Mongo <https://docs.mongodb.com/manual/reference/program/mongo/>, letzter Zugriff: 15. Februar 2021

<sup>28</sup> MongoDB Treiber <https://docs.mongodb.com/drivers/>, letzter Zugriff: 01. Mai 2021

<sup>29</sup> Mongoose - Offizielle Webpage <https://mongoosejs.com/docs/>, letzter Zugriff: 01. Mai 2021

## 2.5 Firebase

Firebase ist eine Backend as a Service-Plattform (BaaS) von Google für mobile und Web-Anwendungen. Sie soll es dem Entwickler ermöglichen, einfacher und effizienter Funktionen auf verschiedenen Plattformen bereitzustellen statt Tools und Infrastruktur zur Verfügung. Mit dem Firebase SDK bietet die Plattform API Schnittstellen zu den jeweiligen Tools, welche direkt in die Anwendung integriert werden können, ohne dass serverseitiger Code dafür notwendig ist. Die Firebase Inc. wurde 2011 von James Tamplin und Andrew Lee gegründet und letztendlich 2014 von Google übernommen.<sup>30</sup> Teile der SDK stehen seit der Google I/O 2017 unter der Apache 2.0 Lizenz, sind somit also Open-Source.<sup>31</sup>

Es existieren zwei Kostenmodelle für die Nutzung von Firebase: Ein kostenloses Modell „Spark Plan“ und ein pay-as-you-go „Blaze Plan“. Das kostenlosen Modell beinhaltet die wichtigsten Tools, viele dieser Tools sind jedoch begrenzt durch beispielsweise Bandbreite oder Speicherplatz. Der Pay-as-you-go Plan ist eine Erweiterung des kostenlosen Plans. Er bietet daher das Nutzen von Tools bis zu einem gewissen Limit kostenfrei an; darüber hinaus kostet es jedoch dann pro Nutzung.

Ein Firebase Projekt ist die oberste Ebene in Firebase. Ein Projekt ist letztendlich ein *Google Cloud Projekt*, welches mit speziellen Konfigurationsmöglichkeiten und Services ausgestattet ist. Es beinhaltet die Verknüpfung zu den einzelnen Anwendungen (also bspw. Android-, iOS- oder Webanwendung). Nun können variabel Tools, sog. Firebase products hinzugefügt werden. Diese Produkte lassen sich grundlegend in drei Kategorien einteilen. Die hier relevantesten werden im Folgenden besprochen [13].

### 2.5.1 Firebase Authentifizierung

Die Authentifizierung gehört zu den „Build“Produkten und bietet eine Token-basierte Nutzerauthentifizierung. Hierbei kann zwischen verschiedenen Anmeldeoptionen gewählt werden: klassisch mit E-Mail und Passwort, mit OAuth2.0 Integration für Social Media (Google, Facebook, Twitter, Github, ...) oder per Telefonnummer. Jeder Nutzer erhält eine einzigartige ID und ein zugehöriges Nutzerobjekt in einer NoSQL Datenbank. Grundlegende Werte wie E-Mail Adresse oder Name können hier abgespeichert werden; zusätzliche Informationen müssen über einen weiteren Datenbank Service abgespeichert werden. Für die Verwaltung eines Accounts bietet dieses Tool auch eingebaute E-Mail Aktionen an - bspw. Passwort zurücksetzen oder E-Mail Adresse bestätigen.

Ein Firebase Nutzer Objekt repräsentiert den Account eines Nutzers, welcher sich von einer Anwendung aus beim zentralen Firebase Projekt angemeldet hat. Die Instanz eines Firebase Nutzers ist somit unabhängig von der Authentifizierungsinstanz der Anwendung, also kann eine Anwendung mehrere Nutzer anmelden, jedoch kann sich auch ein Nutzer auf mehreren Anwendungen anmelden. Ist ein Nutzer authentifiziert, erhält die Anwendung eine Referenz des Nutzers, welche so lange existiert, bis er wieder abgemeldet ist [13].

### 2.5.2 Cloud Firestore

Als Datenbank Lösung bietet Firebase zwei unterschiedliche Produkte an: Cloud Firestore und Realtime Database. Firestore ist hier neuer, jedoch ersetzt es Realtime Database nicht.

Cloud Firestore ist eine flexible und auf Skalierung ausgesetzte NoSQL Cloud Datenbank, wel-

<sup>30</sup>[firebase.googleblog.com](https://firebase.googleblog.com), letzter Zugriff: 03. Mai 2021

<sup>31</sup>[opensource.googleblog.com](https://opensource.googleblog.com), letzter Zugriff: 03. Mai 2021



Abbildung 13: Datenmodell in Firebase [13]

che unter anderem die Echtzeitsynchronisierung der Daten zwischen Anwendung und Server ermöglicht. Zusätzlich zu REST und RPC APIs in iOS, Android und web SDKs ist Firestore auch in nativen Node.js, Java, Python und Go SDKs verfügbar.

Das Datenmodell ist hierarchisch aufgebaut, wobei Daten in Dokumenten (documents) und Dokumente in Sammlungen (collections) gespeichert sind. Mithilfe von Sammlungen werden die Daten voneinander abgetrennt und hierüber können Abfragen erstellt werden. Grundlegende Datentypen sind String, Integer und Boolean, jedoch können auch komplexe Datentypen wie Maps, Arrays oder Geopoints. Unter-Sammlungen und darin verstaute Dokumente sind ebenfalls möglich. Das Datenmodell ist in Abbildung 13

Abfragen werden auf Dokumentenebene erstellt, damit nicht eine gesamte Sammlung aufgerufen werden muss. Dies kann über direkte Sortierung, Filter und/oder Limitierung bzw. genaue Auswahl eines Dokumentes bewerkstelligt werden. Bei einer Abfrage erhält man einen *Data Snapshot*, wodurch über Änderungen in Echtzeit informiert und diese angezeigt werden können. Damit es jedoch zu keinen fehlerhaften Daten führt, gelten hier atomare Eigenschaften für Transaktionen. Eine Transaktion ist eine Folge von Datenbankanweisungen, welche entweder alle gemeinsam oder gar nicht ausgeführt werden. Eine Transaktion ist nur dann erfolgreich, wenn alle Anweisungen auf eine Datenbank vollständig geschlossen sind. Ist dies nicht der Fall, werden alle Anweisungen bis zum Stand vor der Transaktion rückgängig gemacht. Das nennt man Rollback.

Die Sicherheit der Daten stellt Cloud Firestore für Mobil- und Webclient-Bibliotheken über die Firestore-Sicherheitsregeln her. Diese bieten sowohl Zugriffsverwaltung und -authentifizierung, jedoch könnte auch Daten hiermit für die Konsistenz der Datenbank validiert werden.

```

1  service cloud.firestore {
2      match /databases/{database}/documents {
3          match /cities/{city} {
4              allow read, write: if request.auth != null;
5          }
6      }
7  }
```

Listing 20: Beschränkung des Zugriffs auf Dokumente der Sammlung `cities`

Im Beispiel 20 wird der Lese- und Schreibzugriff auf ein Dokument der Sammlung `cities` beschränkt. Nur falls der anfragende Nutzer eine valide Authentifizierung besitzt, erhält er Zugriff

auf das angefragte Dokument. Diese simple Darstellung ist jedoch für den wirklichen Produktionsinsatz mit Vorsicht zu nutzen. Oftmals müssen `read` und `write` in detailliertere Vorgänge aufgeteilt werden. Ein `read` wird spezialisiert in `get` und `list`, wobei ein `write` in `create`, `update` und `delete` unterteilt werden kann. Ein `list` ermöglicht es hierbei auf Sammlungen, also die einzelnen Dokumenten IDs lesend zuzugreifen, jedoch nicht auf die Daten einzelner Dokumente. Hierfür wird dann ein `get` benötigt. Mittels `create` erhält man Schreibzugriff auf nicht existierende Dokumente, durch `update` auf bereits vorhandene und Löschechte ganzer Dokumente erhält man über den `delete` Operator.

Sicherheitsregeln werden gleich dem Datenmodell hierarchisch aufgebaut und ermöglichen differenzierte Zugriffsbeschränkungen auf jeder Ebene. In Codebeispiel 21 beinhaltet jedes Dokument (Stadt) der Sammlung `cities` eine Unter-Sammlung `landmarks`. Nun lässt sich der Zugriff auf beide separat regeln. Bei der Sammlung `villages` hingegen wurde der rekursive Platzhalter verwendet. Hiermit sind Zugriffsregeln auf allen tieferen Ebenen gleich. Beim Verschachteln von `match` ist der innere Pfad immer relativ zum äußeren.

Wichtig zu wissen ist hierzu noch, dass falls mehrere `allow` Ausdrücke auf eine Anfrage zutreffen, wird der Zugriff erlaubt sobald **eine** Bedingung wahr, also erfüllt ist.

```

1  service cloud.firestore {
2      match /databases/{database}/documents {
3          match /cities/{city} {
4              allow read, write: if <condition>;
5
6              // Explicitly define rules for the 'landmarks'
7              // subcollection
8              match /landmarks/{landmark} {
9                  allow read, write: if <condition>;
10             }
11             match /villages/{document==*} {
12                 allow read, write: if <condition>;
13             }
14         }
15     }

```

Listing 21: Hierarchische Zugriffsbeschränkung

Wie bereits oben besprochen können diese Regeln auch zur Validierung von Daten genutzt werden, damit die atomare Eigenschaft von Transaktionen bestehen bleibt. Hierzu kann die `getAfter()` Funktion genutzt werden. Mit dieser kann man auf Zustand eines Dokumentes zugreifen und diesen validieren, nachdem einer Folge von Anweisungen ausgeführt, jedoch diese noch nicht auf der Firestore Datenbank abgeschlossen wurde. Im Beispiel 22 existieren zwei Sammlungen: `cities` und `countries`. Jedes `country` Dokument beinhaltet das Feld `last_updated` um zu wissen, welche Stadt innerhalb eines Landes zuletzt aktualisiert wurde. Hierzu wird in den Sicherheitsregeln nach jedem Schreibzugriff auf ein `city` Dokument gleichzeitig auch das Feld des zugehörigen Landes aktualisiert [13].

```

1  service cloud.firestore {
2      match /databases/{database}/documents {
3          // If you update a city doc, you must also
4          // update the related country's last_updated field.

```

```

5     match /cities/{city} {
6         allow write: if request.auth != null &&
7             getAfter(
8                 /databases/${database}/documents/countries/${request.
9                     resource.data.country)
10                ).data.last_updated == request.time;
11            }
12
13        match /countries/{country} {
14            allow write: if request.auth != null;
15        }
16    }

```

Listing 22: Datenvielfältigung für atomare Operationen

### 2.5.3 Cloud Storage

Um Filme, Videos oder andere Nutzer-generierte Inhalte abspeichern zu können, bietet Firebase Cloud Storage an. Durch das Firebase SDK für Cloud Storage können Dateien direkt von Client-Anwendungen hoch- bzw. heruntergeladen werden. Aufgrund von möglicher schlechter Verbindung kann mithilfe von robusten Operationen der Prozess des Hoch- bzw. Herunterladens bei besserer Verbindung an der Stelle weiter geladen werden, an welcher dieser unterbrochen wurde. Ähnlich wie bei Cloud Firestore in Kapitel 2.5.2 bestimmen auch hier Sicherheitsregeln den Zugriff auf bestimmte Dokumente.

Zusätzlich hierzu sind weitere Metadaten verfügbar: `contentType` und `size`. Mit ihnen lassen sich die Dateien beispielsweise validieren. Im Code 23 können Dateien nur hochgeladen werden, falls sie eine Größe kleiner 5 MB besitzen.

```

1  service firebase.storage {
2      match /b/{bucket}/o {
3          match /images/{imageId} {
4              allow read, write: if request.resource.size < 5 * 1024 *
5                  1024
6                  && request.auth != null;
7          }

```

Listing 23: Validierung nach Dateigröße

Außerdem lassen sich durch Cloud Functions aus dem nächsten Kapitel Prozesse automatisieren. Beispielsweise lässt sich beim Upload eines Bildes direkt ein individuelles Thumbnail erstellen lassen [13].

### 2.5.4 Cloud Functions

Da Firebase - bis auf vereinfachte Sicherheitsregeln - eigentlich keinen Backend Code benötigt, jedoch manche Features eben genau diesen brauchen, um beispielsweise Benachrichtigungen an Nutzer zu senden oder Bilder zu komprimieren, existieren Cloud Functions.

Diese ermöglichen es, als Antwort auf ein Event automatisch oder durch HTTPS Anfrage manuell Backend Code auszuführen. Der gesamte Code ist hierbei in der Google Cloud gespeichert

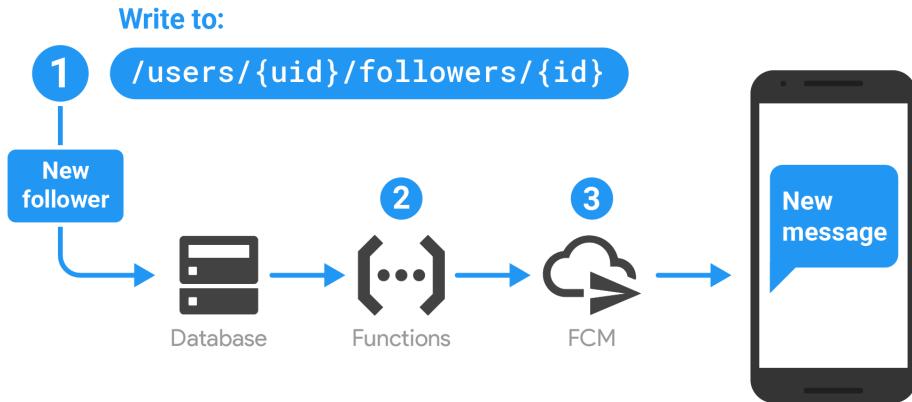


Abbildung 14: Cloud Functions Anwendungsfall Benachrichtigung

und wird in einer verwalteten Umgebung ausgeführt. Als Programmiersprache kann sowohl JavaScript als auch Typescript verwendet werden.

„Google Cloud Functions ist die serverlose Computerlösung von Google zum erStellen ereignisgesteuerter Anwendungen“[13]. Es kann sowohl auf der Google Cloud Platform (GCP) also auch für Firebase genutzt werden. Es ist bei beiden ein Verbindungsglied zwischen Logik und entsprechenden Diensten, welche dadurch mit serverseitigen Code erweitert und kombiniert werden. In Abbildung 14 ist ein typischer Anwendungsfall beschrieben. Ein Event auf der Datenbank wird ausgelöst, hier ein neuer Nutzer folgt einem weiteren Nutzer. Es wird also ein Dokument in der Unter-Sammlung `followers` erzeugt. Diese Unter-Sammlung befindet sich innerhalb des Dokumentes `uid` der Sammlung `users`. Im zweiten Schritt erstellt die Funktion eine Nachricht, welche über Firebase Cloud Messaging (FCM) versendet werden soll. Über abgespeicherte Tokens sendet FCM die Benachrichtigung an das Gerät des Nutzers `uid` [13].

## 2.5.5 Cloud Messaging

Firebase Cloud Messaging ist eine plattformübergreifende Messaging-Lösung zum zuverlässigen Versenden von Nachrichten an Nutzergeräte. In Abbildung 15 ist die Architektur dieses Tools dargestellt. Hierbei wird es grundlegend in das Erstellen, Transportieren und Empfangen der Nachrichten unterteilt [13].

**Erstellen:** Die zu versendenden Nachrichten können, wie in Kapitel 2.5.4 beschrieben, manuell oder automatisiert erzeugt werden. Bei der Automatisierung ist wichtig, dass die Nachrichten in einer vertrauenswürdigen Serverumgebung erstellt werden, damit alle Nachrichtentypen unterstützt werden (Schritt 1). Das FCM Backend akzeptiert dann in Schritt 2 Nachrichtenanfragen, ordnet die Nachrichten verschiedenen Themen zu und erzeugt unter anderem Metadaten für Nachrichten, wie bspw. die Nachricht ID.

**Transportieren:** Die Nachrichten werden hierbei an die entsprechenden Geräte weitergeleitet. Da verschiedene Geräte auf unterschiedlichen Plattformen basieren, muss die Transportschicht auf Plattformebene arbeiten. Hierfür werden folgende Ebenen genutzt:

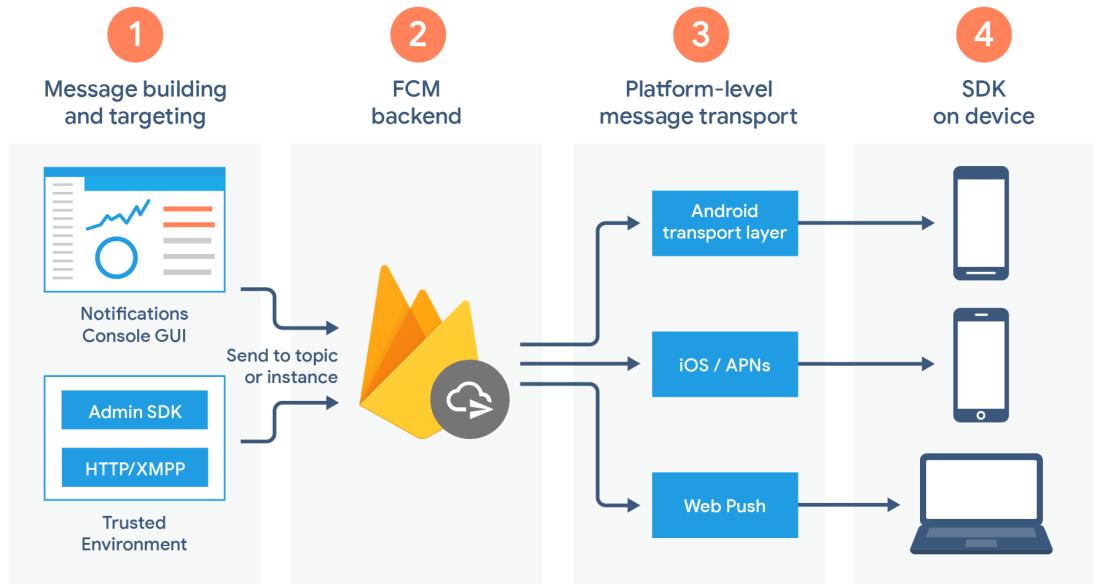


Abbildung 15: Firebase Cloud Messaging Architektur

- Android Transport Layer (ATL) für Android-Geräte mit Google Play-Diensten
- Apple Push Notification Service (APNs) für iOS-Geräte
- Web-Push-Protokoll für Web-Apps

**Empfangen:** Das FCM SDK behandelt die Benachrichtigung oder Nachricht. Dies ist abhängig vom Vorder-/ Hintergrundstatus der Anwendung und der jeweiligen Anwendungslogik.

### 2.5.6 Google AdMob

Google AdMob bietet eine einfache Art, gezielte Werbung innerhalb der Anwendung zu schalten und somit die Anwendung zu monetarisieren. Zusätzlich bietet das Tool in Kombination mit Google Analytics<sup>32</sup> zusätzliche Anwendungsdaten und Analysefähigkeiten.

Werbung lässt sich in unterschiedlicher Weise anzeigen (siehe Abbildung 16) und lässt sich reibungslos in UI Komponenten integrieren. Verschiedene Features sind hier jedoch plattformabhängig. Auf der Android Plattform ist es für Nutzer möglich, beworbene Produkte direkt aus der Anwendung heraus zu kaufen.

Ein weiteres Werbemittel *Google Mobile Ads SDK* ist eine alleinstehende SDK, hingegen Google AdMob bietet einfache Integration in Firebase und weitere Tools.[13]

## 2.6 Anwendungsentwicklung für mobile Endgeräte

Mobile Geräte sind heutzutage ein sehr großer Teil unseres Tagesablaufs. Durchschnittlich verbringen wir 3:54 Stunden pro Tag an mobilen Geräten (hier bezogen auf Bürger der USA). Die meiste Zeit hiervon wird in Apps (ca. 90%).<sup>33</sup> Laut Cisco wird dieser Markt sich jedoch nicht

<sup>32</sup>Ein freies Analysetool, welches über alle Tools hinweg Ereignisse sammelt und diese Werte direkt graphisch darstellt. Da es für die Implementierung nicht weiter relevant ist, wird es nicht detaillierter besprochen. Zusätzliche Informationen unter <https://firebase.google.com/docs/analytics>, letzter Zugriff: 26. Februar 2021

<sup>33</sup><https://www.emarketer.com/content/us-time-spent-with-mobile-2019>, letzter Zugriff: 26. Februar 2021

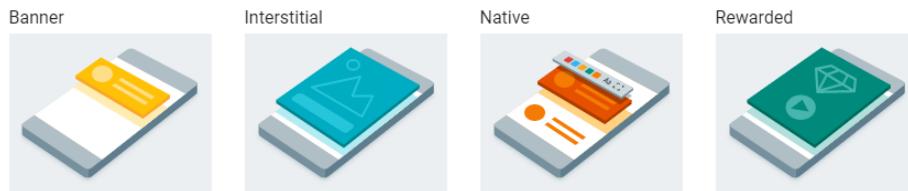


Abbildung 16: Google AdMob Anzeigemöglichkeiten

nur auf Industrieländer beruhen, sondern bis 2023 sollen weltweit 71% der Bevölkerung mobile Konnektivität haben [8]. Diese Entwicklung forcierte viele Firmen immer mehr ihre Anwendungen auch *mobile ready* zu gestalten. Dies kann man bspw. deutlich bei der Anpassung vieler Webseiten an Mobile Seiten- und Größenverhältnisse oder auch dem Anbieten von *Apps*, welche bereits für Desktop o.ä. verfügbar waren, erkennen.

Daher ist es für die Wirtschaft und Entwicklung gleichermaßen wichtig sich ständig weiterzuentwickeln und sich nicht auf (kosten-) ineffizienten Entwicklungsprozessen auszuruhen. Dabei bieten jährliche, wenn nicht sogar halbjährliche Design- und Performanceänderungen von den Geräten selbst oder der Betriebssysteme Herausforderungen an die mobilen Anwendungen - *apps* - und gleichzeitig an deren Programmierumgebung. Trotz einer riesigen Auswahl an *Apps* lassen sich diese allgemein in drei Kategorien eingliedern: Plattformspezifische native Anwendungen, adaptive Webanwendungen und plattformübergreifende Anwendungen.

### 2.6.1 Begriffe

**Eine Plattform:** besteht aus der Hardware (System und zusätzlicher Peripherie, wie Sensoren oder Aktoren), dem Betriebssystem, den spezifischen *Software Development Kits (SDK)* und den jeweiligen Basisbibliotheken. Zusammen bietet eine Plattform die Grundlage um Software für sie zu entwickeln.

**Ein Framework:** definiert eine Architektur für Anwendungen und stellt Komponenten bereit, mit welchen das Entwickeln einer Anwendung erleichtert sein soll [6]. Ein plattformübergreifendes Framework muss somit Anwendungscode für mehrere Plattformen wiederverwenden, jedoch müssen auch plattformspezifische Funktionen, wie Architektur oder Benutzeroberflächen API, bereitgestellt werden. Mehr dazu in Kapitel 2.6.3.

**Eine mobile Anwendung:** ist eine Anwendung, geschrieben für eine Plattform eines mobilen Endgerätes, welche die jeweiligen Features nutzen könnte - dazu zählen Kamera(s), Beschleunigungssensoren oder auch *Global Positioning System (GPS)*. Webseiten als solches sind demnach keine mobilen Anwendungen.

### 2.6.2 Plattformspezifische native Apps

Plattformspezifische oder auch native Anwendungen sind Programme, welche auf eine gewisse Plattform abzielen und in einer der davon unterstützten Programmiersprachen geschrieben wurden. Da diese Art der (mobilen) Anwendung mit plattformspezifischen SDK und *Frameworks* entwickelt wird, ist diese Anwendung an eine Plattform gebunden.

Dies bringt zum einen natürlich Vorteile wie allgemein best mögliche Performance auf der jeweiligen Plattform und direkt vom Hersteller unterstützte Entwicklungsumgebungen/SDKs. Zudem lassen sich plattformspezifische Fähigkeiten oder Einstellungen nutzen - beispielsweise mehrere Kameras oder GPS.

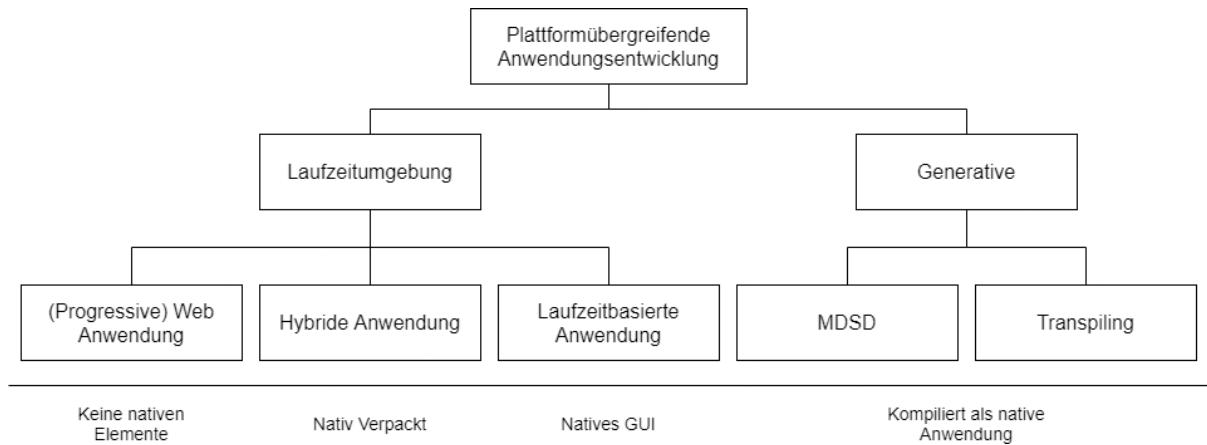


Abbildung 17: Kategorisierung verschiedener plattformübergreifender Ansätze. Erstellt nach [7]

Gleichzeitig beschränkt man sich aber logischerweise auf eine Plattform und deckt mit einer Anwendung nur einen Teil des gesamten Marktes. Dies bringt im Vergleich zu den anderen Möglichkeiten einen deutlich erhöhten Entwicklungs- und Wartungsaufwand mit sich, da für andere Plattformen Programmcode nicht übernommen werden kann. Zusätzlich benötigen Entwickler spezifische Kompetenzen für beide Plattform und Entwicklungsumgebungen.

Zwei der am weitesten verbreiteten Plattformen sind Android von Google und iOS von Apple. Anwendungen für Android können in Kotlin oder Java als Programmiersprache beispielsweise in dem *integrated development environment (IDE)* von Google Android Studio entwickelt werden. Für iOS wird hingegen mit Objective-C und Swift als Programmiersprache primär in der IDE XCode entwickelt.

Beide bieten jeweils Plattform eigene Services an, beispielsweise das direkte Veröffentlichen in den jeweiligen Appstore [2].

### 2.6.3 Plattformübergreifende Anwendungen

Die Entwicklung einer plattformübergreifenden Anwendung zeichnet sich generell durch die Möglichkeit aus, nur einmal Code schreiben zu müssen, diesen jedoch auf mehreren Plattformen ausführen zu können.

Verschiedene Ansätze einer solchen Anwendung sind in Abbildung 17 kategorisiert. Im Folgenden werden jene Entwicklungsmöglichkeiten detaillierter besprochen.

#### 2.6.3.1 (*Progressive*) Web Apps

Eine mobile Webanwendung ist eigentlich eine Webseite, welche sich an die Größe und Auflösung von unterschiedlichen Bildschirmen anpasst - hier speziell an die Bildschirmgrößen der mobilen Geräte. Diese Anwendung ist mit Standard Webentwicklungstools geschrieben (HTML, CSS & JavaScript) und läuft somit theoretisch auf jedem Gerät mit einem Internet Browser [9]. Aufgrund der steigenden Unterstützung von jeglichen APIs in mobilen Browsern, ist es auch möglich geworden auf Geräteeigenschaften, wie bspw. den Standort zuzugreifen.

Jedoch kann diese App logischerweise nicht im jeweiligen *Appstore* heruntergeladen werden, da es sich weiterhin um eine Webseite handelt. Aus gleichem Grund kann hiermit auch kein „natives Design und Leistung“ erzeugt werden.

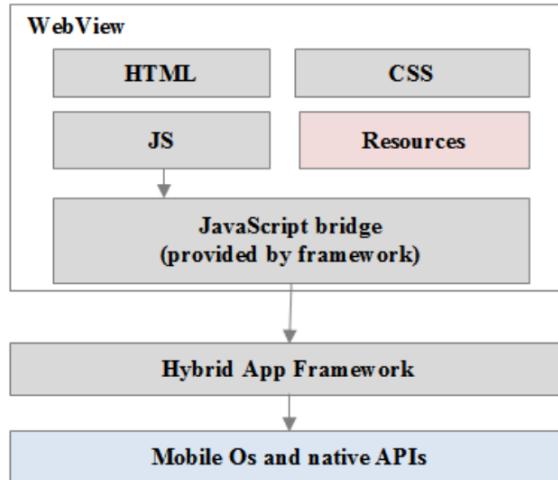


Abbildung 18: Struktur einer hybriden Anwendung<sup>34</sup>

Abhilfe hierfür sorgt jedoch die von Google vorgestellte Design Idee *Progressive Web Apps* (PWA). Sie bietet die Möglichkeit Code in sogenannte *service worker* als Hintergrundthread ausführen zu lassen, ein Webseiten Manifest anzugeben, die App offline bedienen zu können und bieten die Möglichkeit die PWA zu installieren. Gleichzeitig kann mit diesem Design eine zu nativen Apps vergleichbare Leistung erreicht werden [11].

Generell ist der Ansatz sehr simpel, da hiermit plattformübergreifende Anwendungen geschrieben werden können, welche sich auf allen Geräten mit Browser bedienen lassen. Hierfür wird zudem keine zusätzliche Programmiersprache oder wissen über die jeweilige Plattform benötigt. Eine große Schwierigkeit hieran ist weiterhin der Zugriff auf Gerätefeatures, da nicht alle über den Browser verfügbar sind.

### 2.6.3.2 Hybride Anwendungen

Eine hybride Anwendung kombiniert die native Vorgehensweise mit der einer normalen Webseite. In einer nativen WebView ist eine Webanwendung verpackt, welche nun in einer *HTML-Rendering-Engine* gerendert wird. Bei Android und iOS ist das WebKit. Diese WebView funktioniert ähnlich wie ein normaler Browser, jedoch werden Kontrollfenster nicht angezeigt, wie zum Beispiel Adresszeile, Einstellungen oder Lesezeichen. Ähnlich wie bei Web Anwendungen werden über JavaScript APIs Gerätefeatures eingebunden.

Ein sehr frühes Framework für diese Art von Anwendung war Adobe Cordova, eher bekannt als das ursprüngliche PhoneGap von Nitobi. Viele weitere Frameworks basieren auf ihren Anfängen.

Eine hybride Anwendung kann also normal als App im *Appstore* heruntergeladen auf dem Gerät installiert und offline genutzt werden - also sehr ähnlich zu nativen Lösungen. Daher ist dieser Ansatz auch sehr beliebt. Daher liegt auch hier die Leitung der Applikation deutlich hinter der der Nativen [10] [11].

### 2.6.3.3 Runtime basierte Anwendungen

Im Gegensatz zur in Kapitel 2.6.3.2 beschriebenen hybriden Anwendung, nutzen Runtime basierte Anwendungen keinen Browser des Gerätes mit einer WebView, sondern jede App besitzt

<sup>34</sup>Quelle: [10]

eine eigene Runtime Ebene. Jedes Framework muss also eine solche Ebene für alle Plattformen in jeweiliger Programmiersprache mitliefern, damit seine Anwendung hierauf laufen können. Die Anwendungen hingegen sind dann beispielsweise in JavaScript (bspw. React Native oder NativeScript), C# (Xamarin) oder sonstigen Programmiersprachen (bspw. Qt) geschrieben.

Jedem Framework-Entwickler ist die Freiheit gegeben, wie man die Anbindung an native Funktionen regelt. Bei hybriden Anwendungen ist dies durch die WebView Cordova festgelegt. Typisch für Anwendungen dieser Art jedoch ist ein Plug-In-basiertes *bridging System*. Es ermöglicht den Aufruf von fremden Funktionsinterfaces in plattformspezifischem Code. Somit können beispielsweise React Native und NativeScript mit Sprachinterpretoren (bspw. JavaScriptCore und V8) auf den Geräten Auszeichnungssprache (hier HTML (Hypertext Markup Language)) interpretieren und plattformspezifische Komponenten der Benutzeroberflächen erzeugen.

Ein großer Nachteil dieser Strategie ist jedoch zugleich ihr Vorteil: Jedes Framework besitzt seine eigene Architektur. Dadurch sind Plug-ins des einen Frameworks trotz gleicher Anwendungssprache nicht unbedingt funktionstüchtig im anderen. Bei projektspezifischen Plug-ins macht es einen späteren Systemwechsel daher besonders schwer, da nicht nur Benutzeroberfläche und Businesslogik neu geschrieben werden müssen, sondern auch jeweilige Plug-ins [11].

#### 2.6.3.4 Model-driven Software Entwicklung

Die Grundsätze der modellgetriebenen Softwareentwicklung beschäftigen sich mit der Abstraktion des Modells als (Teil eines) System, von welchem die eigentliche Software abgeleitet wird [12].

Das bedeutet in der Realität, dass eine höhere Abstraktion als Quellcode in Form von textuellen oder grafischen domänenspezifischen Sprachen oder universell einsetzbaren Modellierungssprachen (Unified Modeling Language(UML)) zum beschreiben der Software verwendet wird. Codegeneratoren übersetzen diese Modelle nun jeweils in Programmiersprachen der gewählten Zielplattform, auf welcher sie kompiliert werden.

Theoretisch kann hierdurch der komplette Funktionsumfang wie bei einer nativen Anwendung erreicht werden. Bekannte Frameworks dieser Methode sind zum Beispiel *MD<sub>2</sub>*, MAML, WebRatio Mobile, BiznessApps und Bubble.

Der große Nachteil hieran ist, dass Entwickler sehr selten modellgetriebene Entwicklung verwenden, sondern Quellcode-basierte Programmiermethoden bevorzugen [11].

#### 2.6.3.5 Kompilierte Anwendungen

Kompilierte plattformübergreifende Anwendungen basieren auf einer einzigen Codebasis und können für mehrere Plattformen vollständig kompiliert werden. Dies kann entweder von der Codebasis einer nativen Anwendung für mindestens eine andere Plattform (bspw. J2ObjC), oder von einer unabhängigen Codebasis direkt für mehrere Plattformen (bspw. Flutter) geschehen. Hierbei ist Flutter für diesen Anwendungsfall am interessantesten und wird in Kapitel ?? näher behandelt.

Ein Hindernis dieser Art ist die erhöhte Komplexität der einzelnen Frameworks.

### 2.7 Frameworks zur mobilen, plattformübergreifenden Entwicklung

In den folgenden Kapiteln werden einzelne Frameworks zur mobilen, plattformübergreifenden Entwicklung vorgestellt.

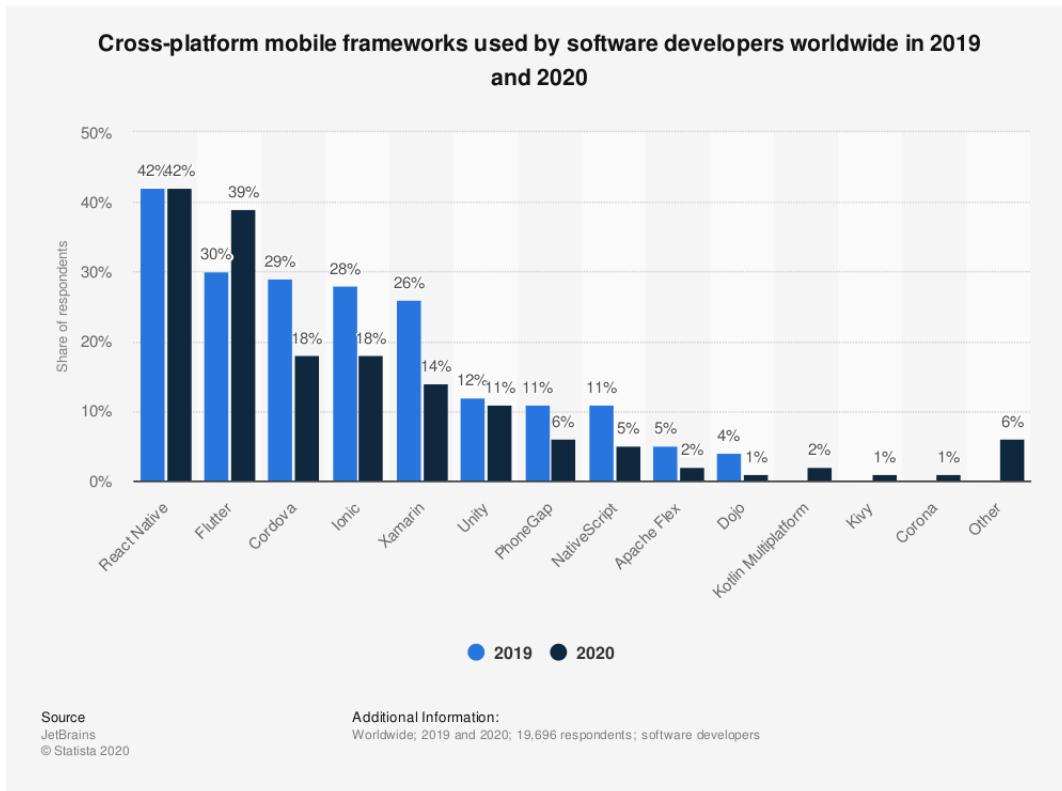


Abbildung 19: Beliebtheit nach Framework im Bereich der plattformübergreifenden mobilen Entwicklung<sup>35</sup>

In der Statistik 19 aus dem Jahr 2020 ist React-Native das beliebteste Framework, dicht gefolgt von Flutter. Wie man deutlich hier auch sehen kann, haben andere, bisher auch sehr erfolgreiche Frameworks einen enormen Rückgang von teilweise über einem Drittel ihrer Nutzer erleben müssen.

Aus diesen Gründen werden im weiteren Verlauf nur die Frameworks React-Native und Flutter weiter besprochen.

### 2.7.1 React Native

React Native ist ein open source Framework, welches von Facebook 2015 veröffentlicht wurde. Es basiert auf dem bekannten Web-Framework *React* (ebenfalls von Facebook) und bringt daher den deklarativen und Komponenten-basierten Stil mit sich. Die Programmiersprache ist aus diesem Hintergrund auch logischerweise JavaScript. Das Framework an sich ist in verschiedenen Sprachen implementiert: JavaScript, Swift, Objective-C, C++ und Python.

Allgemein bietet das Framework die Möglichkeit plattformübergreifende Apps für iOS, Android und für Windows zu schreiben. Hierbei wird der geschriebene Code in einer JavaScript Laufzeitumgebung ausgeführt (React Native selbst verwendet generell JSC (JavaScriptCore), seit neuestem kommt auch Hermes zum Einsatz, jedoch sind auch andere bekannte Umgebungen denkbar - bspw. V8 in Chrome) es lässt sich zu den Runtime-basierten Anwendungen in Kapitel

<sup>35</sup>Quelle: <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>, letzter Zugriff: 22. April 2021

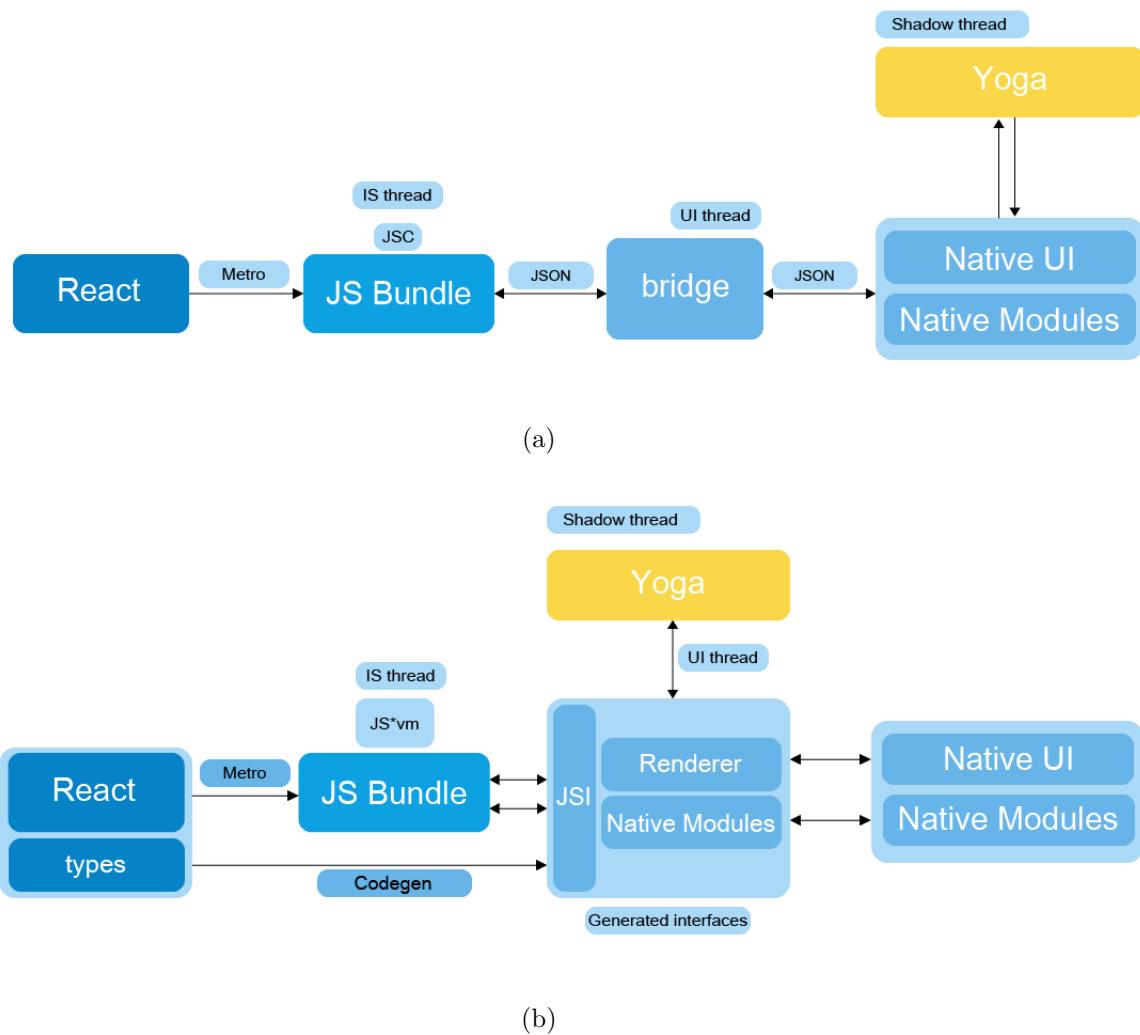


Abbildung 20: (a) Alte und (b) neue Architektur von React Native<sup>36</sup>

2.6.3.3 zuordnen [3].

### 2.7.1.1 Architektur

Grundlegend wurde React Native als Plattform-agnostisch designet. Entwickler schreiben also plattformunabhängigen JavaScript React Code, während das Framework den erstellten React Baum in Plattform-spezifischen Code umschreibt. Hierbei wurde 2013 (noch intern) die Web-Technologie React mit nativen Plattformen (nur interne) vereint, jedoch war dieses Design aufgrund eines einzigen Threads sehr langsam. Um dies zu verbessern basierte das Framework lange Zeit auf drei unterschiedlichen Threads, welche über eine Brücke verbunden sind.

- *JavaScript Thread*. Hier wird der gesamte JavaScript Code abgelegt und interpretiert. Alles wird über die JSC Engine ausgeführt.
- *Native Thread*. Die Benutzeroberfläche und Kommunikation mit dem JavaScript Thread steht hier im Mittelpunkt. Der gesamte native Code wird hier ausgeführt. Die Benutzeroberfläche besteht aus **Native UI** und **Native Modules**.

<sup>36</sup>Quelle: <https://litslink.com/blog/new-react-native-architecture>, letzter Zugriff: 15. April 2021

berfläche wird dann aktualisiert, sobald die eben ein Änderung vom JS Thread vermittelt wird.

- *Shadow Thread*. Hier wird das gesamte Layout der Anwendung berechnet. Zugrunde liegt die Facebook-eigene layout engine „Yoga“.

Ein Hauptproblem dieses Ansatzes ist, dass die Brücke grundlegend eine asynchrone Warteschlange ist, da der JS Thread und der native Thread unabhängig voneinander arbeiten. Zusätzlich werden während der gesamten Datenübertragung die Daten im JSON Format serialisiert und deserialisiert . Daher kann es zu Performance-Einbrüchen und somit zu schlechter Nutzererfahrung, durch bspw. Eingabeverzögerung, kommen.

Nach der Ankündigung 2018 veröffentlichte Facebook im Juli 2020 die neue Architektur. Mit ihr wurde der Bottleneck (die Brücke) ersetzt durch das JavaScript Interface. Es ermöglicht nicht nur die komplette Synchronisierung der beiden Threads, sondern auch die direkte Kommunikation untereinander - vor allem das Konzept von „shared ownership“ ist hier tragend, weshalb auch keine Serialisierung mehr nötig ist. Zudem ist man nun nicht mehr an JSC gebunden, sondern kann auch jegliche hoch-performante JavaScript Engines als Laufzeitumgebung verwenden. Native Module werden nun nur noch bei Bedarf geladen anstatt alle beim Start der App.

Weiter wurde veralteter Legacy-Code aus dem Kern von React Native entfernt und nicht-essentielle Teile aus dem Kern ausgelagert. Die aktuelle Architektur von React Native ist in Abbildung 20b zu sehen.<sup>37</sup>

### 2.7.1.2 JSX mit nativen Komponenten

React (Native) verwendet als Programmiersprache JSX. Diese ist eine syntaktische Erweiterung von JavaScript (**J**ava**S**cript **eXtension**), welche zur fundamentalen Beschreibung der Nutzeroberfläche dient. JSX wird in normale JavaScript Objekte kompiliert, weshalb es nicht zwingend ist.

```
1 const element = <h1>Hello, world!</h1>;
```

Listing 24: JSX Hello World Element

Mithilfe dieser losen Kopplung von UI-Code und dazugehöriger Logik schlägt React eine optionale Lösung zur *Separation of Concerns* vor. Anstelle dessen ist es auch möglich die Technologien in Markup- und Logik-Dateien aufzuteilen.

Außerdem kann argumentiert werden, dass JSX nur eine weitere Template-Sprache sei, ähnlich HTML oder XAML. Dies ist nicht korrekt, da wie oben bereits erwähnt JSX lediglich eine syntaktische Erweiterung von JavaScript ist, also inmitten von JSX Objekten JavaScript geschrieben werden kann.

Weiterhin ist interessant, dass JSX Cross Site Scripting vorbeugt, indem der React DOM alle eingesetzten Werte zunächst in einen einfachen String konvertiert [4].

```
1 import React from 'react';
2 import { Text } from 'react-native';
3
4 const Cat = () => {
5   return (
6     // <Text> as native component
7     <Text>Hello, I am your cat!</Text>
```

<sup>37</sup>Quelle: React Native's re-architecture (2020): <https://medium.com/swlh/react-natives-re-architecture-in-2020-9bb82659792c>, letzter Zugriff: 17. April 2021

```

8     );
9 }
10
11 export default Cat;

```

Listing 25: Native Komponenten

In dem Codebeispiel 25 wird ein Element `Cat` erzeugt, welches als Beschreibung dessen dient, was letztendlich auf dem Bildschirm angezeigt wird. In diesem einfachen Beispiel wird die native Komponente `<Text>...</Text>` verwendet. Native Komponenten sind in nativem Code (Kotlin oder Java für Android, bzw. Swift oder Objective-C für iOS) implementierte Komponenten und können in JavaScript Code aufgerufen werden. Diese werden dann während der Laufzeit für die jeweilige Plattform erstellt.

React Native bringt die wichtigsten Komponenten mit sich, die **Core Components**. Zusätzlich erlaubt das Framework jedoch auch eigene Komponenten nativ zu implementieren, welche dem jeweiligen Anwendungsfall angepasst werden können.

### 2.7.1.3 Komponenten

Gleichzeitig erlaubt React Native aber auch wiederverwendbare Komponenten in JavaScript aus den Kernkomponenten zusammenzustellen. Hierzu lassen sich einzelne Komponente ineinander verschachteln, um hier im Beispiel 26 einen `Text` innerhalb einer `View`<sup>38</sup> anzeigen zu lassen. Zusätzlich ist es möglich sogenannte „props“ also Eigenschaften (engl.: properties), ähnlich einer normalen Funktion mitzugeben. In dem betrachteten Beispiel entspricht das den Namen einzelner Katzen, welche als Text angezeigt werden [3].

```

1 import React from 'react';
2 import { Text, View } from 'react-native';
3
4 // configurable props
5 const Cat = (props) => {
6   return (
7     <View>
8       <Text>Hello, I am {props.name}!</Text>
9     </View>
10   );
11 }
12
13 const Cafe = () => {
14   return (
15     <View>
16       // reusable
17       <Cat name="Maru" />
18       <Cat name="Jellylorum" />
19       <Cat name="Spot" />
20     </View>
21   );

```

<sup>38</sup>Eine `View` ist die Basiskomponente einer Benutzeroberfläche. In einer `View` wiederum können weitere Views verschachtelt sein.

```
22 }
23
24 export default Cafe;
```

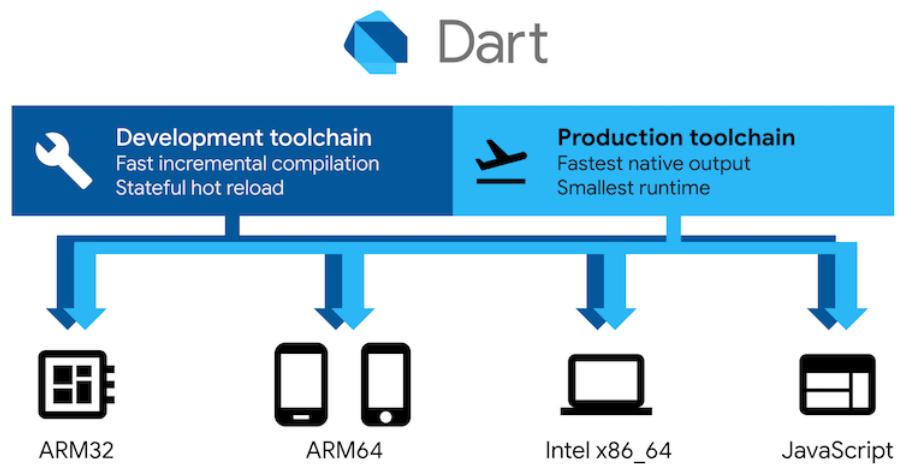
Listing 26: Eigene Komponenten

#### 2.7.1.4 State

Für eine interaktive Benutzeroberfläche fehlt jedoch noch das Kernprinzip eines deklarativen UI. Ein sogenannter *State* wird verwendet um die Daten, welche sich mit der Zeit oder über Nutzerinteraktion ändern, auf der Oberfläche anzuzeigen. Dieses Konzept wird ebenfalls von dem zweiten Framework verwendet und ist in Abbildung 24 visualisiert.

Seit v14.8 ermöglicht React (und auch React Native) durch Hooks einer Funktion einen *State* hinzuzufügen. Hooks sind Funktionen, welche es Entwicklern ermöglichen sich in React Features einzuhaken. Bisher wurde dies über Klassenkomponenten ermöglicht, wodurch es jedoch komplizierter ist *stateful logic* zwischen Komponenten wiederzuverwenden. Daher entkoppelt man diese Logik von den Komponenten und erlaubt unter anderem das separate Testen.

```
1 import React, { useState } from "react";
2 import { Button, Text, View } from "react-native";
3
4 const Cat = (props) => {
5   // make isHungry stateful
6   const [isHungry, setIsHungry] = useState(true);
7
8   return (
9     <View>
10    <Text>
11      // show text dependent on isHungry state
12      I am {props.name}, and I am {isHungry ? "hungry" : "full"}!
13    </Text>
14    <Button
15      onPress={() => {
16        setIsHungry(false);
17      }}
18      disabled={!isHungry}
19      title={isHungry ? "Pour me some milk, please!" : "Thank you!"}
20    }
21    />
22  );
23}
24
25 const Cafe = () => {
26   return (
27     <>
28       <Cat name="Munkustrap" />
29       <Cat name="Spot" />
30     </>
31   );
}
```

Abbildung 21: Kompatibilität der Dart Plattform <sup>40</sup>

```

32 }
33
34 export default Cafe;

```

Listing 27: State mit useState Hook

Im Beispiel 27 wird in Zeile 6 der `useState`-Hook verwendet. Diese Funktion erzeugt eine State-Variablen mit dem Initialwert `true` und erstellt gleichzeitig eine Funktion zur Änderung des States (`setIsHungry`). Daraufhin wird, abhängig ob die Katze hungrig ist, dies im Text angezeigt, der Knopf zum füttern (de-)aktiviert bzw. auch hier den Text verändert [3].

## 2.7.2 Flutter

Flutter ist eine open-source SDK entwickelt von Google und ist geschrieben in C, C++ und Dart. Auf Basis eines Codes erlaubt Flutter es gleichzeitig Anwendungen für Android, iOS, Web und Desktop zu erstellen und ist zudem die primäre Methode für Google Fuchsia, Googles Betriebssystem<sup>39</sup>. Als 2D Grafikbibliothek verwendet Flutter Skia, welche auch von Chrome, Firefox und Android verwendet wird. Zudem basiert Flutter auf der Dart-Plattform welche das Kompilieren auf 32-bit und 64-bit ARM Prozessoren, auf Intel x64 Prozessoren und in JavaScript ermöglicht (siehe Abbildung 21). Daher ist Flutter eine wie in Kap. 2.6.3.5 beschriebene kompilierte, plattformübergreifende Anwendung.

Während der Entwicklung werden Flutter-Apps in einer virtuellen Maschine (VM) gestartet, welche *stateful hot reload* ermöglicht. Bei Änderungen muss die App so nicht komplett neu kompiliert werden, sondern sie erneuert sich während der Laufzeit. Bei der Veröffentlichung der App, wird sie in den Maschinencode der beschriebenen Plattformen übersetzt.

### 2.7.2.1 Architektur

Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has privileged access to

<sup>39</sup>Quelle: <https://fuchsia.dev/>

<sup>40</sup>Quelle: <https://github.com/flutter/flutter>, letzter Zugriff; 12. April 2021

the layer below, and every part of the framework level is designed to be optional and replaceable<sup>41</sup>.

Grundlegend ist das Framework in drei Prozesseinheiten gegliedert. Diese bestehen wiederum jeweils aus, für sie charakteristischen APIs und Bibliotheken:

**Flutter embedder:** Der Einstiegspunkt in die jeweilige Plattform. Er koordiniert Zugriffe auf Services des Betriebssystems; er ist also zuständig für bspw. die Kommunikation mit dem Input Method Editor (IME) und den Lifecycle Events der App. Daher ist der Embedder in der, von der Plattform unterstützten Programmiersprache geschrieben: derzeit wird Java und C++ für Android, Objective-C/Objective-C++ für iOS und macOS, und C++ für Windows und Linux verwendet.

**Flutter Engine:** Der Kern von Flutter, geschrieben hauptsächlich in C und C++, ist die *low-level* Implementierung der Flutter Kern Programmierschnittstelle (API). Daher ist sie zuständig für das graphische Darstellen (Rasterisierung) des Codes sobald ein neuer *Frame* angezeigt werden muss. Im Flutter Framework wird die *Engine* als dart:ui Bibliothek offengelegt - der zugrundeliegende C++ Code wird in Dart Klassen eingefügt.

**Flutter Framework:** Das Framework, mit welchem der Entwickler schlussendlich meistens arbeiten wird. Es ist in Dart geschrieben und bietet sogenannte *Layer* für Animationen, Layout und Widgets. Widgets werden von Flutter als Einheit der Komposition von Benutzeroberflächen verwendet und sind als einzelne Bausteine zu verstehen, welche zusammengefügt ein Objekt oder sogar einen kompletten Bildschirm ergeben.

Bei der Entwicklung mit Flutter wird ein Baum von Widgets erzeugt, welcher als Bauplan der Applikation angesehen werden kann. Nach diesem Plan wird mithilfe von States der einzelnen Widgets schlussendlich das User Interface (UI) gerendert [5].

### 2.7.2.2 Dart

Während der Entwicklung von Flutter standen sicherlich mehrere Sprachen zur Auswahl: Wie in Kapitel 2.6.3 gelernt, gibt es viele unterschiedliche Ansätze mit beispielsweise webbasierteren Sprachen wie JavaScript, mit nativen Sprachen wie Java oder Swift, oder auch mit anderen objektorientierten Sprachen wie C#. Wieso wurde also genau Dart als Programmiersprache und Runtime ausgewählt?

Die Programmiersprache Dart ist in allgemeinen Zügen der Sprache C ähnlich, sodass vielen Entwicklern das Lesen erleichtert wird. Es ist eine objektorientierte Sprache und besitzt einen Garbage Collector.

Dart ist designet als eine client-fokussierte Sprache, welche gleichermaßen Entwicklung (sub-second stateful hot reload) und Produktion in allen möglichen Zielplattformen (Web, Mobile und Desktop) priorisiert. Dadurch erhält man eine effizienzoptimierte Entwicklungsphase, sowie ebenfalls die Möglichkeit eine Code-Basis in unterschiedliche Plattformen zu kompilieren (siehe Abbildung 21).

Sie bietet zudem auch *sound-null-safety*, was bedeutet dass Werte nicht null sein können, außer sie werden so festgelegt. Hiermit werden Null-Exceptions während der Laufzeit durch statische Codeanalyse vorgebeugt.

---

<sup>41</sup><https://flutter.dev/docs/resources/architectural-overview>

<sup>42</sup>Quelle: <https://github.com/flutter/flutter>, letzter Zugriff: 17. März 2021

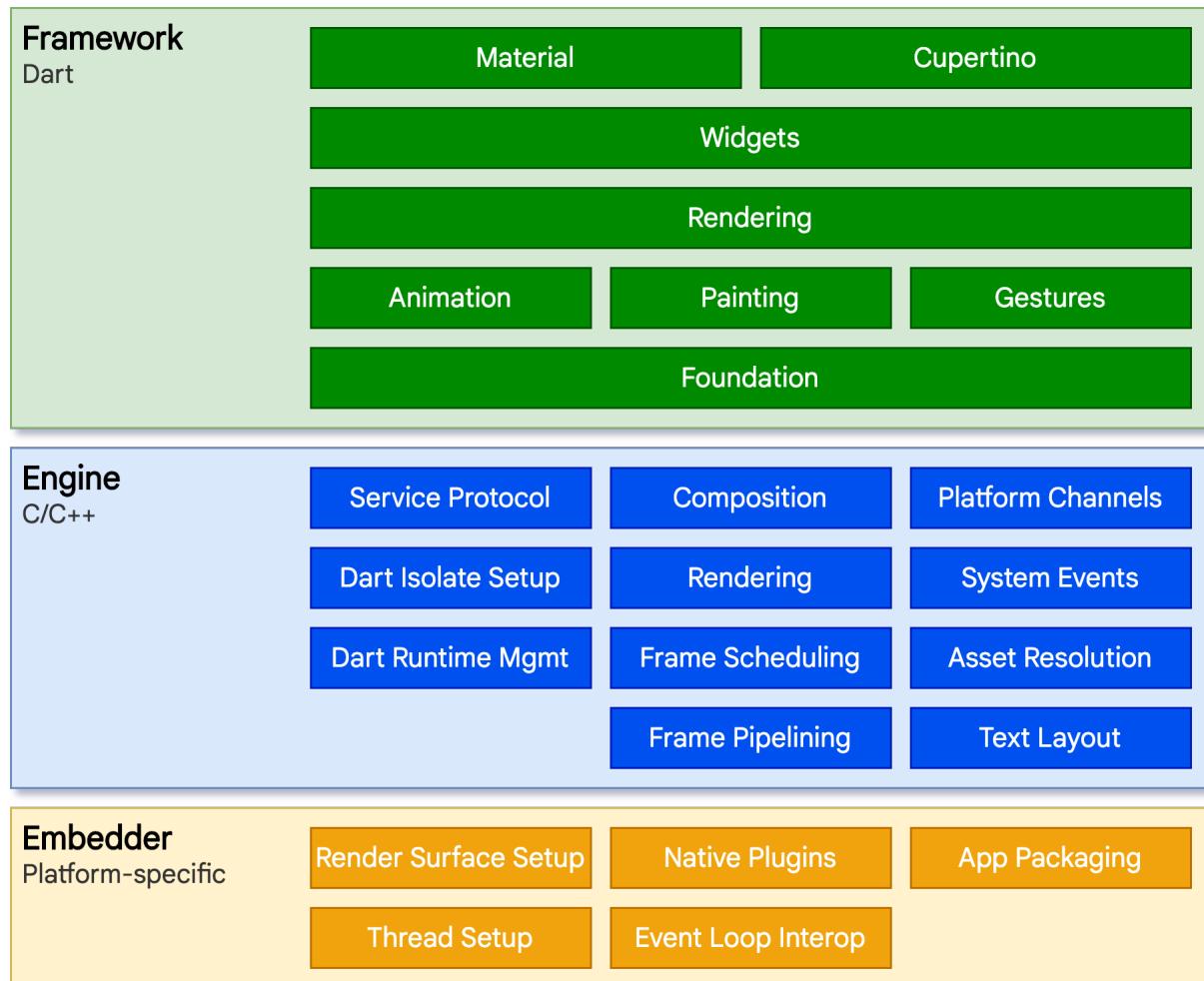


Abbildung 22: Bibliotheken und Ebenen der Flutter Plattform <sup>42</sup>

### 2.7.2.3 Widgets

Wie bereits beschrieben sind Widgets wiederverwendbare Kompositionsbauusteine, mit welchen Benutzeroberflächen in Flutter zusammengebaut werden. Jedes einzelne ist ein *immutable declaration* eines Teils der Benutzeroberflächen - also ein konstanter Bestandteil.

Flutter arbeitet mit der Devise:

***Everything is a widget.***

Auf diesem Satz baut die Einfachheit von Flutter auf. Jedes Objekt, jede Animation, jede Reihe, einfach alles ist ein Widget. Somit baut man eine App von der Wurzel aus auf und beschreibt die einzelnen Abzweigungen exakt. Die Anordnung von Widgets ist daher hierarchisch aufgebaut. Ein Widget wird also immer in einem Elternteil verschachtelt sein und erhält bei seiner Erstellung den *build context* übergeben. Das „äußerste“ Widget, also die Wurzel, enthält somit die gesamte App. Typischerweise ist das ein *MaterialApp* oder *CupertinoApp* Widget.

OEM Widgets, also Widgets von und für eine spezifische Plattform werden von Flutter gemieden. Hierfür erzeugt Flutter eigene Widgets mithilfe der oben genannten, eigener Rendering Plattform. Da diese Widgets jedoch komplett individualisierbar sind, bietet man somit native

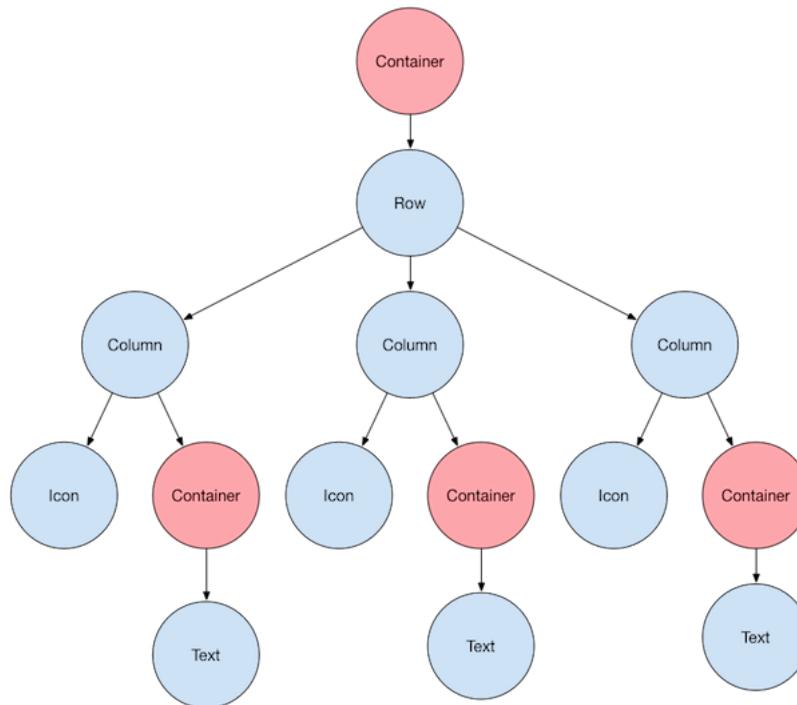


Abbildung 23: Widget Baum einer beispielhaften Anwendung<sup>43</sup>



Abbildung 24: Deklarative Benutzeroberfläche<sup>44</sup>

Möglichkeiten für jegliche Stile. Es gibt auch Pakete, welche Plattform-ähnliche Widgets zur Verfügung stellen.

#### 2.7.2.4 States

Flutter ist deklarativ - das bedeutet, die Benutzeroberfläche wird anhand von dem aktuellen *State* der App erzeugt. Im Gegensatz dazu muss der Entwickler beim imperativen Stil die Übergänge der einzelnen *States*. Die Abbildung 24 stellt eine deklarative Benutzeroberfläche dar. Hier wird dies durch das Framework gelöst.

<sup>43</sup>Quelle: <https://flutter.dev/docs/development/ui/layout>, letzter Zugriff: 21. März 2021

<sup>44</sup>Quelle: <https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative>, letzter Zugriff: 12. April 2021

Tabelle 6: Nutzer-Item Matrix mit Bewertungen. Jede Zelle  $r_{u;i}$  steht hierbei für die Bewertung des Nutzers  $u$  an der Stelle  $i$

		Items					
		1	2	...	i	...	m
Users	1	2		1			3
	2	4			5		
	...			1			4
	u		4		5		1
		2				3	
	n		4		3		

## 2.8 Recommender System

Auf der Webseite Youtube allein werden minütlich mehr als 500 Stunden Videomaterial hochgeladen. (<https://blog.youtube/press/>, 10.02.2021) Um bei einer solch unvorstellbaren Menge an Daten (allein auf einer Webseite) den Überblick als Endnutzer behalten zu können, ist ein personalisierten Filtersystems unausweichlich.

Solche Filtersysteme, auch Recommendation System genannt, nutzt bisher gesammelte Daten um Nutzern potentiell interessante Objekte jeweils individuell vorzuschlagen. Ein sogenannter *Candidate Generator* ist hierbei ein Recommendation System, welches die Menge  $M$  als Eingabe erhält und für jeden Nutzer eine Menge  $N$  ausgibt. Hierbei umfasst  $M$  alle Objekte und gleichzeitig gilt  $N \subset M$ .

Die Bestimmung einer solchen Menge  $N$  beruht grundlegend auf zwei Informationsarten. Erstens die sogenannten Nutzer-Objekt Interaktionen, also beispielsweise Bewertungen oder auch Verhaltensmuster; Und zweitens die Attributwerte von jeweils Nutzer oder Item, also beispielsweise Vorlieben von Nutzern oder Eigenschaften von Items [1]. Systeme, welche zum Bewerten ersteres benutzen, werden *collaborative filtering* Modelle genannt. Andere, welche zweiteres verwenden, werden *content-based filtering* Modelle genannt. Wichtig hierbei ist jedoch, dass *content-based filtering* Modelle ebenfalls Nutzer-Objekt Interaktionen (v.a. Bewertungen) verwenden können, jedoch bezieht sich dieses Modell nur auf einzelne Nutzer - *collaborative filtering* basiert auf Verhaltensmustern von allen Nutzern bzw. allen Objekten.

Ein solches Recommendation System kann im einfachsten Fall wie in 6 als Matrix dargestellt werden.

### 2.8.1 Nutzerinformation

Damit ein *Recommender System* einem Nutzer Vorschläge bereitstellen kann, benötigt es Nutzerinformationen. Das Design des jeweiligen Systems hängt auch, wie oben beschrieben, von der Art der Information und von der Art der Beschaffung dieser ab.

#### 2.8.1.1 Explizite Nutzerinformation

Bei der expliziten Methode muss der Nutzer individuelle Informationen aktiv über sich preisgeben. Dies kann über konkrete Fragestellungen zu beispielsweise Geburtsdatum, Geschlecht oder Interessen geschehen. Diese Art der Information beschreiben einen Nutzer konkret.

Eine andere Art der Information sind Bewertungen von Objekten. Diese lassen sich beispielsweise Intervall basiert darstellen. Hierbei werden geordnete Zahlen in einem Intervall als Indikator genutzt, ob ein Objekt gut oder schlecht war - zum Beispiel eine Bewertung eines Produktes von 0 bis 5 Sternen bei Amazon. Diese Information beschreiben die Vorlieben eines Nutzers konkret.

Je größer diese Skala ist, desto differenzierter ist auch das Meinungsbild, da jeder Nutzer sich genau ausdrücken kann. Jedoch desto komplizierter und unübersichtlich wird auch das Bewertungsverfahren an sich, da man einen zu großen Entscheidungsraum für den Nutzer darbietet.

### 2.8.1.2 Implizite Nutzerinformation

Um implizit Nutzerinformationen zu erfassen, muss ein System die Verhaltensmuster seiner Kunden als Daten abspeichern. Beispielsweise könnte das System von YouTube erfassen, ob Videos frühzeitig abgebrochen oder ganz angeschaut werden. Anklicken von Webseiten und die darauf verbrachte Zeit könnte ebenfalls als Bewertung gespeichert und zur Generierung von Vorschlägen genutzt werden.

### 2.8.2 Content-based filtering

Unter *content-based filtering* versteht man das Betrachten von Ähnlichkeiten zwischen Objekten anhand von Schlüsselwörtern (Eigenschaften) und daraus dann das Vorhersagen der Nutzer-Objekt Kombination für ein bestimmtes Objekt. Nimmt man an, Film 1 und Film 2 haben ähnliche Eigenschaften (gleiches Genre, gleiche Schauspieler, ...) und Nutzer A mag Film 1, so wird das System Film 2 vorschlagen.

Das System ist also unabhängig von anderen Nutzerdaten, da die Vorschläge nur auf Präferenzen eines einzelnen Nutzers basieren. Dies bietet im Hinblick auf eine App auch gute Skalierungsmöglichkeiten. Zudem kann auf Nischen-Präferenzen gut eingegangen werden, da nicht mit anderen Nutzerdaten verglichen wird, sondern nur ein Nutzer für sich betrachtet wird.

Gleichzeitig schlagen *content-based filtering* Systeme aber eher offensichtliche Objekte vor, da Nutzer oft unzureichend genaue "Beschreibungen", also Vorlieben mit sich bringen. Dadurch, dass nur basierend auf Schlüsselwörter neue Objekte vorgeschlagen und andere Nutzerwertungen nicht miteinbezogen werden, sind die Vorschläge sehr wahrscheinlich oftmals ähnlich bis gleich - man "verfängt" quasi in eine Richtung [1].

### 2.8.3 Collaborative Filtering

Unter *collaborative filtering* versteht man das Betrachten von Ähnlichkeiten im Verhalten von Nutzern anhand von Bewertungen und Präferenzen, bzw. anhand der Ähnlichkeiten von Objekten.

Generell wird bei [1] in zwei Typen unterschieden:

1. *Memory-based Methoden*: Es wird, wie oben beschrieben, aus gesammelten Daten Ähnlichkeit herausgearbeitet und Nutzer-Objekt Kombinationen durch eben diese vorhergesagt. Daher wird dieser Typ auch *neighborhood-based collaborative filtering* genannt. Man unterscheidet weiter in:
  - (a) *User-based*: Ausgehend von einem Nutzer A werden andere Nutzer mit ähnlichen Nutzer-Objekt Kombinationen gesucht, um Vorhersagen für Bewertungen von A zu treffen. Ähnlichkeitsbeziehungen werden also über die Reihen der Bewertungsmatrix berechnet.
  - (b) *Item-based*: Hierbei werden ähnliche Objekte gesucht und diese genutzt um die Bewertung eines Nutzers für ein Objekt vorherzusagen. Es werden somit Spalten für die Berechnung der Ähnlichkeitsbeziehungen verwendet.
2. *Model-based Methoden*: Machine Learning und Data Mining Methoden werden verwendet um Vorhersagen über Nutzer-Objekt Kombinationen zu treffen. Hierbei sind auch gute Vorhersagen bei niedriger Bewertungsdichte in der Matrix möglich.

Vereinfacht gesagt: Wenn Nutzer A ähnliche Bewertungen verteilt wie Nutzer B, und B den Film 1 positiv bewertet hat, wird das System Film 1 auch Nutzer A vorschlagen. Das selbe gilt auch umgekehrt (*Item-based*).

Diese Art leidet sehr unter dem *sparsity* Problem, also dass die Nutzer zu wenige Bewertungen von Objekten ausüben. Daher sind Vorhersagen über Ähnlichkeit von Nutzern aufgrund unzureichender Datensätze nicht sinnvoll möglich. Dieses Problem wird *Cold-Start Problem* genannt.

#### 2.8.4 Ähnlichkeit von Objekten und Nutzern

Sowohl bei *collaborative filtering*, als auch bei *content-based filtering* wird jedes Objekt und jeder Nutzer als ein Vektor im Vektorraum-Modell  $E = \mathbb{R}^d$  (englisch *embedding space*) erfasst. Sind Objekte beispielsweise ähnlich, haben sie eine geringe Distanz voneinander.

Ähnlichkeitsfunktionen sind Funktionen  $s : E \times E \rightarrow \mathbb{R}$  welche aus zwei Vektoren beispielsweise von einem Objekt  $q \in E$  und einem Nutzer  $x \in E$  ein Skalar berechnen, welches die Ähnlichkeit dieser zwei beschreibt  $s(q,x)$ .

Hierfür werden mindestens eine der folgenden Funktionen verwendet:

**Cosinus-Funktion:** Hier wird einfach der Winkel zwischen beiden Vektoren berechnet:  $s(q,x) = \cos(q,x)$

**Skalarprodukt:** Je größer das Skalarprodukt, desto ähnlicher sind sich die Vektoren.  $s(q,x) = q \circ x = \sum_{i=1}^d q_i x_i$

**Euklidischer Abstand:**  $s(q,x) = \|q - x\| = [\sum_{i=1}^d (q_i - x_i)^2]^{\frac{1}{2}}$

Dadurch, dass die Ähnlichkeit des Skalarprodukts deutlich mit der Größe der Norm eines Vektors steigt. Da populäre Objekte, beispielsweise beliebte YouTube Videos, öfter in einem Satz von Trainingsdaten auftauchen, besitzen diese eine höhere Norm. Somit würden über das Skalarprodukt auch eher populäre Objekte vorgeschlagen werden. Je nach Anwendungsfall muss also die Ähnlichkeitsfunktion gewählt sein.<sup>45</sup>

---

<sup>45</sup>Quelle: <https://developers.google.com/machine-learning/recommendation/>, letzter Zugriff: 07. Januar 2021

## 3 Planung

### 3.1 Konzept

Aus Perspektive der Benutzer ist die Idee hinter StreamSwipe trivial. Dem User werden nacheinander Filme und Serien vorgeschlagen, die er bewerten kann. Auf Basis seiner Präferenzen erhält er Matches, mit denen er über eine Chatfunktionen kommunizieren kann. Jeder dieser Schritte muss möglichst schnell und unkompliziert erfolgen.

Aus Sicht des Anbieters ist die Realisierung wesentlich komplexer als es dem User erscheint, da eine Zusammenstellung an Komponenten benötigt wird. Diese besteht aus einer Smartphone-App, einer Userdatenbank, einer Datenbank mit den Filminformationen und einem Server, der das Matching ausführt. Zusätzlich wird eine mögliche Komponente für den Messenger benötigt. Die Auswahl dieser Komponente wird jeweils über individuell gestellte Anforderungen getroffen.

Bei der Smartphone-App muss bereits während der Entwicklung auf Benutzerfreundlichkeit und Barrierefreiheit geachtet werden. Sie sollte intuitiv bedienbar, attraktiv design und ressourcensparend für leistungsschwache Smartphones programmiert sein. Um ein möglichst großes Publikum ansprechen zu können, muss sie für iOS und Android erhältlich sein.

Die Datenbank mit den Benutzerinformationen muss jederzeit erreichbar sein und eine gewisse Form von Sicherheit und Verschlüsselung bieten, da da hier persönliche Angaben gespeichert werden.

Die Datenbank mit den Filminformationen muss sehr umfangreich sein, also auch ältere Filme und Nischenfilme enthalten, und ständig aktualisiert werden. Neben den offensichtlichen Informationen wie Filmtitel und Filmposter muss sie auch weitere Daten zu den Filmen enthalten wie Erscheinungsdatum, Handlung, Regie, etc. trotzdem sollte der Zugriff schnell sein und wenig kosten, sodass die Nutzung der App möglichst preiswert ist.

Der Server muss durchgehend laufen und über eine Internetverbindung erreichbar sein. Er muss leistungsstark genug sein um alle Nutzeranfragen bedienen zu können, sollte jedoch so klein gehalten werden, dass die Anschaffungs- und Unterhaltskosten minimal sind.

Der Messenger muss ebenfalls über einen zentralen Punkt gesteuert werden, da sich nur zwischen Personen ein Chat öffnen darf, die gematcht wurden. Es sollte keine Zugriffsbegrenzung auf den Nachrichtenserver geben, um einen unbegrenzten Chatverlauf garantieren zu können. Nachrichten müssen gespeichert werden, um versendete Nachrichten auch nach dem Schließen der App zustellen zu können. Dieses System muss ebenfalls verschlüsselt sein und benötigt Zugriff auf Benutzerinformationen wie ID, Name und Profilbild.

### 3.2 Komponenten

In StreamSwipe ist für jede dieser Komponenten eine Lösung gefunden worden. Bei der Entwicklung der App wurden Benutzerfreundlichkeit und Barrierefreiheit beachtet, wie in Abschnitt 7. Wie in Abschnitt 2.7 beschrieben wird, können die genannten Anforderungen durch eine geschickte Wahl des Frameworks erfüllt werden. Als Benutzerdatenbank kann Firebase verwendet werden, da wie in Abschnitt 2.5 beschrieben, die benötigten Funktionen vorhanden sind. Es ist ebenfalls möglich den Messenger mit den Firebasefunktionen zu implementieren. Als Quelle der Filminformationen wird die API von „The Movie Database“ (TMDb) verwendet, auf welche in Abschnitt 4.5 genauer eingegangen wird. Der verwendete Server besteht aus einem Raspberry Pi, jedoch mehr dazu in Kapitel 4.2. Wie diese Komponenten aufeinander zugreifen wird in Abbildung 25 verdeutlicht.

Über die Smartphone-App werden die Benutzerdaten an Firebase weitergeleitet. Wird ein neuer Account angelegt, so werden diese Daten dort gespeichert und bei einem Anmeldevorgang mit den bestehenden Benutzerdaten verglichen. Nur wenn der User bereits vorhanden ist, kann

eine Anmeldung stattfinden. So wird sichergestellt, dass nur angemeldete Benutzer auf die eigentlichen Funktionen der App Zugriff erhalten. Jede Aktion in der App wird auf diese Weise einem Benutzerkonto zugewiesen und kann später darüber identifiziert werden. Innerhalb der App können alle Funktionen frei genutzt werden weshalb es wichtig ist, dass ausschließlich eingeloggte User auf die Screens der App zugreifen können.

Der Server erhält die Film-IDs aller Filme aus der TMDb-API und erstellt somit eine eigene Film-Datenbank. Diese IDs werden an das Smartphone weitergeleitet und die App erhält über die IDs die Filminformationen direkt von der TMDb-API. Es werden gleichzeitig nur eine geringe Anzahl an Filminformationen aus der TMDb-API auf das Smartphone geladen, sodass dieser Vorgang möglichst wenig Ressourcen benötigt und unbemerkt im Hintergrund ablaufen kann. Hierzu zählen Filmposter, Rating, Besetzung, Veröffentlichungsdatum, übersetzte Sprachen und viele mehr. Noch bevor der User über alle diese Filme abgestimmt hat, werden neue Filminformationen geladen, sodass es für den User keine Unterbrechungen gibt und es ihm wie ein einzelner unendlicher Fluss an Daten erscheint.

Die Filmbewertung wird mit den Film-IDs vom Smartphone an den Server geschickt, der diese beiden Informationen miteinander verknüpft. Das Rekommendationsverfahren auf dem Server verarbeitet diese Präferenzen und sucht wie in Kapitel 2.8 beschrieben nach Übereinstimmungen bei den anderen Benutzern. Da hierbei alle Präferenzen aller User miteinander verglichen werden, ist dieser Schritt sehr aufwändig und sollte nicht nach jeder neuen Präferenz durchgeführt werden. Bei StreamSwipe wird dieses Matching deshalb automatisch in regelmäßigen Zeitabständen durchgeführt, optimalerweise zu einer Tageszeit, an der die Benutzeraktivität gering ist. Die Anzahl der verglichenen Nutzer kann mit einer Filterung durch Geschlechterpräferenzen und Wohnort reduziert werden um das Matchingverfahren zu beschleunigen.

Werden zwei User gematcht, wird ihnen dies jeweils in der App angezeigt und sie können ein Nachrichtenfenster öffnen um miteinander zu schreiben. Die Textkommunikation findet ebenfalls über Firebase statt. aus den beiden User-IDs wird eine einzigartige Chat-ID erstellt, welche jeweils in der Nutzerdatenbank der beteiligten User hinterlegt wird, sodass nur diese beiden User auf den Chat Zugriff erhalten.

In Firebase wird für jeden Chat eine weitere Sammlung erstellt. Durch den hierarchischen Aufbau von Firebase, welcher in Kapitel 2.5.2 erklärt ist, können innerhalb und unterhalb von Sammlungen Daten gespeichert werden. Innerhalb der Sammlung eines Chats sind die beiden Nutzernamen, die beiden Nutzer-IDs und die Chatroom-ID gespeichert. Unter dieser Sammlung werden die Nachrichten mit den Informationen wer von beiden die Nachricht geschickt hat (*sendBy*), wer sie empfangen hat (*sendTo*), dem Nachrichteninhalt und einem Zeitstempel gespeichert. Mithilfe der Daten *sendBy* und *sendTo* kann später im Chatverlauf entschieden werden auf welcher Seite eine Nachricht angezeigt wird, wie in Abbildung 38e zu sehen ist, und wer der beiden Beteiligten bei neuen Nachrichten eine Benachrichtigung erhält.

Wie bereits erwähnt wird die Chat-ID bei beiden Usern gespeichert. Abhängig davon ob ein Chat angenommen wurde oder nicht, ist die Chat-ID unter *chatrooms* oder unter *pendingChatrooms* gespeichert. Für den User, der die Unterhaltung beginnt, wird die Chat-ID unter *chatrooms* hinterlegt, für den anderen ist dies solange unter *pendingChatrooms* bis auf die Nachricht geantwortet wird. Sobald beide User die Unterhaltung akzeptiert haben, können sie jeweils die Profilseite des anderen besuchen.

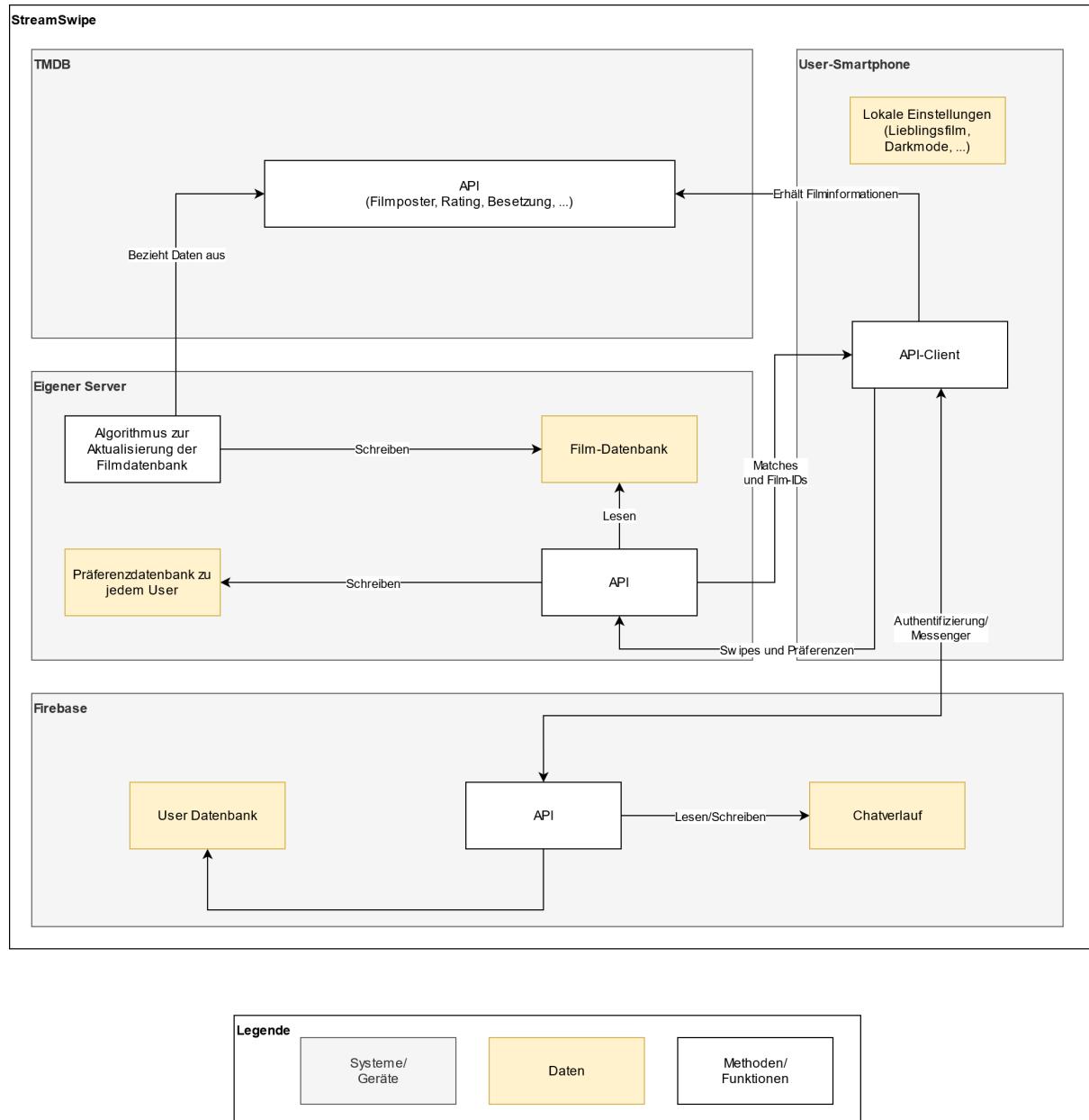


Abbildung 25: Komponentendiagramm

## 4 Auswahl geeigneter Technologie

Den Anforderungen der Entwicklung entsprechend wurde zunächst entschieden, welche Technologien zum Einsatz kommen.

### 4.1 Anwendungsframework

Bei der Auswahl des Framework zur Programmierung der eigentlichen Anwendung gibt es, wie in Kapitel 2.6 erläutert, eine Vielzahl von möglichen Herangehensweisen. Die Anwendung ist letztendlich der Teil unseres Systems, welches direkt mit dem Nutzer in Berührung kommt. Daher sind vor allem die Anforderungen an Performance, Aussehen und Benutzbarkeit der Oberfläche essentiell wichtig. Gleichzeitig ist bei der Entwicklung auf die verwendete Programmiersprache, die Entwicklungsumgebung, welche das Framework mit sich bringt und die Kenntnisse aller Entwickler zu achten.

Wie bereits in Abbildung 19 gezeigt, können sich Marktanteile verschiedener Frameworks sehr schnell ändern. Dies hängt auch deutlich mit den unterliegenden Plattformen und deren Update-Zyklen zusammen. Google beispielsweise veröffentlicht beinahe jährlich eine neue Version von Android<sup>46</sup>, weshalb sich eben Programmierumgebungen immer mitentwickeln. Die Popularität eines Frameworks spiegelt somit unter anderem wieder, wie gut es mit den neuesten Features und Programmierkonzepten umgeht. Daher wurde sich bei den plattformübergreifenden Frameworks 2.7 auf React Native und Flutter beschränkt. Beide bieten vordefinierte, nativ implementierte Benutzeroberflächenkomponenten - welche im Falle von Flutter sogar auf dem Design System *Material Design* beruht.

Das Konzept aus Kapitel ?? schlägt vor, dem Nutzer die Bewertung einzelner Filme über eine Wisch-Funktion bereitzustellen. Allein dieser Bildschirm muss zunächst einmal eine Reihe von Filmen inklusive Titelbild und Informationen über HTTP Anfragen von unserer Datenbank laden, die Animationen und Logik hinter dem Bewertungssystem abarbeiten und zusätzlich die Bewertung wieder zurück an unseren Server senden. Aufgrund dieses enorm hohen Performance-Anspruchs sind nativ implementierte Komponente unabdingbar. Da sowohl Flutter als auch React Native es ermöglichen, die UI Komponenten in der plattformspezifischen Sprache zu implementieren, erweist sich der Leistungsvergleich beider Frameworks als schwierig. Grundlegend kann jedoch angenommen werden, dass Flutter eher weniger CPU-Nutzung beansprucht, jedoch dazu tendiert, mehr Speicher vor der eigentlichen Anwendung anzufordern [11].

Somit kristallisierten sich drei Entwicklungsmöglichkeiten per Ausschlussverfahren heraus:

- Native Anwendung für jeweils Android und iOS
- Plattformübergreifend mit Flutter
- Plattformübergreifend mit React Native

Mit einer nativen Anwendung für jede Plattform erhält man nicht nur doppelten Entwicklungsaufwand, sondern gleichzeitig auch doppelten Wartungsaufwand. Zusätzlich gegen die native Möglichkeit spricht, dass für die Entwicklung einer iOS Anwendung die Entwicklungsumgebung XCode benötigt wird, welche eine Desktopanwendung ausschließlich für das Betriebssystem macOS ist. Daher ist die Entwicklung nicht nur in Sachen Codezeilen ein deutlicher Mehraufwand, welcher sich aber bei der Größe des Projektes nicht bezahlt macht. Bei den plattformübergreifenden Lösungen entfällt dieser Aspekt, jedoch ist das veröffentlichen einer iOS Anwendung im Vergleich zu Android weiterhin deutlich komplizierter - hierauf wird im weiteren Verlauf der Arbeit noch eingegangen.

---

<sup>46</sup><https://engineerbabu.com/blog/evolution-of-android-versions/>, letzter Zugriff: 20. April 2021

Beim direkten Vergleich zwischen React Native und Flutter ist der deutlichste Unterschied, dass React Native die Entwicklung einer Laufzeit-basierten Anwendung bietet und Flutter die einer komplizierten Anwendung. Hierbei werden unterschiedliche Programmiersprachen verwendet: Flutter beruht auf der Google eigenen Sprache Dart. Da diese sich bisher noch nicht etablieren konnte, ist diese Sprache unbekannt und muss neu erlernt werden. JavaScript hingegen ist weit verbreitet, jedoch bringt die spezielle Erweiterung JSX bei React Native ebenfalls zusätzlichen Lernaufwand mit sich.

Während der Entwicklung bieten beide Frameworks eine „Hot Reload“Funktion an, welche vor allem bei der Erstellung von Benutzeroberflächen deutliche Zeitersparnisse ermöglicht. Flutter punktet jedoch vor allem hierbei durch die direkte Unterstützung von UI Bibliotheken, welche React Native ausschließlich über externe Bibliotheken bezieht. Gleichzeitig bringt Flutter allgemein mehr „out of the box“mit sich, weshalb bei React Native eher Probleme durch Abhängigkeiten von Drittanbieter entstehen können.

Der ausschlaggebende Punkt ist jedoch tatsächlich die BaaS-Plattform (Backend-as-a-Service). Es existiert zwar ein großer Konkurrent zu Firebase, nämlich AWS Amplify. Hierbei handelt es sich ebenfalls um ein Backend-Service für Mobil- und Webanwendungen. Aufgrund der auf Skalierung ausgesetzten Tools mit besseren Echtzeit-Features ist Firebase jedoch speziell für Nachrichtenaustausch besser geeignet. Gleichzeitig spielt die Vertrautheit mit dieser BaaS Plattform und die breit gefächerten Tools eine große Rolle bei der Auswahl des Services. Letztendlich wurde die Entscheidung getroffen, das allgemeines Nutzermanagement und Chat-Funktion der Anwendung mit Firebase zu realisieren.

Da AWS Amplify erst seit Beginn 2021 eine offizielle Unterstützung für Flutter anbietet<sup>47</sup>, wurde Flutter als Frontend-Framework und Firebase als BaaS-Plattform ausgewählt.

## 4.2 Server

Der Webserver ist jener Dienst, der die zugrundeliegenden Funktionalitäten bezüglich der Filmabfragen, der Matching-Logik und der Filmempfehlung bietet. Anforderungen an die genutzte Webservertechnologie sind zum einen eine grundlegend hohe Performance, um mit einer hohen Anzahl an Serveranfragen umzugehen. Des Weiteren sollte die Technologie Skalierbarkeit in Bezug auf die Performance und wachsenden Ressourcen wie Hardware aufweisen und eine Unterstützung, Dokumentation und umfassende Funktionalitäten des Frameworks bieten. Ein Paketmanager, der umfassende Kern-Funktionalitäten zur Verfügung stellt, sollte vorhanden sein, damit diese nicht neu implementiert werden müssen. Um offen für das genutzte Zielbetriebssystem zu bleiben, sollten mehrere Betriebssysteme unterstützt werden. Im folgenden werden keine proprietären Webserver-technologien betrachtet.

Nach genauerer Recherche kamen drei Webservertechnologien in die engere Betrachtung:

- PHP
- Django
- Node.js

Im Hinblick auf die Performance sticht Node.js aufgrund seiner ereignisgesteuerter Architektur und dem Non-Blocking I/O-Mechanismus heraus und verspricht eine bessere Ressourcennutzung.

<sup>47</sup><https://aws.amazon.com/de/about-aws/whats-new/2021/02/announcing-general-availability-amplify-flutter-data-authentication-support/>

Große Firmen wie Uber<sup>48</sup>, Ebay und Netflix<sup>49</sup> haben ihre Systeme bereits auf Node.js umgestellt. Die Wahl als Webservertechnologie fällt auf Node.js, da es breite Unterstützung erfährt, die auch durch den mächtigen Paketmanager npm ergänzt wird und eine hohe Performance errichtet.

### 4.3 Datenbank

Im Hinblick auf die Speicherung der potenziell hohen Anzahl an Nutzern, deren Swipe-Entscheidungen und deren Matches untereinander sowie der Anzahl von über 500.000 Filmen<sup>50</sup> wird ein performanter Umgang der Datenbank mit vielen Datensätzen notwendig sein. Um massive Daten speichern zu können sind relationale Datenbanken nicht die passende Wahl. Es hat sich gezeigt, dass je größer die Menge an Daten ist und je mehr Tabellen in einer Anfrage enthalten sind, desto größer ist der Performanceverlust durch SQL. [?]

Die Verwendung von dokumentenbasierten Datenbanken führt dagegen zu einer strukturlosen Zusammensetzung an Daten, bei denen ein Dokument ein einzelnes Objekt repräsentieren kann. Somit muss für die Wiedergabe eines Objekts nur ein Dokument angefragt werden. Die fehlenden Möglichkeiten zur Normalisierung können jedoch zu Redundanzen in den Daten führen, wodurch die Entwicklung der aufrufenden Anwendung komplexer werden kann. Die Redundanz wird jedoch in Kauf genommen, um schnelle Abfragen zu ermöglichen und eine hohe Performance zu erhalten. Da Datensätze in dokumentenbasierten NoSQL-Datenbanken schemalos als JSON-Objekte abgelegt werden, begünstigt dies den generischen Dokumentenaufbau in der Entwicklung.

Einige NoSQL-Datenbanken wie MongoDB verfolgen einen nicht-relationalen Ansatz und werden mit JavaScript-fähigen Schnittstellen bereitgestellt. Durch die Kommunikation im JSON-Format eignen sich ein optimaler Einsatz mit Node.js gegeben. MongoDB ist über seine horizontale Skalierbarkeit darauf ausgelegt, in einem kurzen Zeitraum sehr viele Daten zu verarbeiten [?]. Während relationale Systeme vertikal im Sinne von neuen Tabelleneinträgen skalieren, werden Dokumente hingegen werden in einer Kollektion, die horizontal erweitert werden können, indem Datenmengen im Sinne des Shardings auf mehreren Systemen verteilt werden, anstatt ein einzelnes System zu verwenden.

Als Datenbank für die Backend-Implementierung wurde aufgrund von Performance, der guten Einbindung an Node.js und der Skalierbarkeit MongoDB ausgewählt. Um die Vorteile der Datenkompression sowie der Transaktionen über mehrere Dokumente hinweg zu nutzen, ist die WiredTiger-Engine die Wahl für das genutzte Storage-Engine.

### 4.4 Kommunikationsschnittstelle

Als Kommunikationsschnittstelle wird eine WebApi entwickelt, dessen Kommunikation auf HTTP-Nachrichten basiert, deren Informationen im JSON-Format übergeben werden. Diese Technologie bietet eine einfache und dennoch effiziente Form der Kommunikation zwischen Server und Client.

### 4.5 Film-Datenbank

Um die Informationen der einzelnen Filme anzeigen zu können, werden zunächst einmal Informationen benötigt. Hierfür existieren mehrere Anbieter, wobei die Anbindungen und auch die Datensätze selbst von unterschiedlicher Qualität sind.

<sup>48</sup>siehe <https://eng.uber.com/uber-tech-stack-part-two/>, letzter Zugriff: 18. März 2021

<sup>49</sup>siehe <https://entwickler.de/online/javascript/7-gruende-node-js-579924149.html>, letzter Zugriff: 18. März 2021

<sup>50</sup>siehe <https://www.themoviedb.org/faq/general>, letzter Zugriff: 18. März 2021

Für die API der OMDb (Open Movie Database) ist leider keine gute Dokumentation frei verfügbar. Gleichzeitig ist der Zugriff über 1000 Anfragen pro Tag und eine Poster API zur Anzeige der Filmposter nur verfügbar, falls man eine Spende über Patreon hinterlässt<sup>51</sup>.

Die API von IMDb (Internet Movie Database) ist hingegen sehr gut dokumentiert und bietet auch die wichtigsten Daten (Titel, Beschreibung und Bild) an. Problem hieran ist jedoch, dass es zwar einen kostenlosen Zugang gibt, dieser jedoch auf 100 Anfragen täglich beschränkt ist<sup>52</sup>. Gleichzeitig hat diese API eine Verzögerung von knapp 1 s, was zu schlechterer Benutzerfahrung führen kann<sup>53</sup>.

Letztendlich wurde die Version 3 der API von TMDb (The Movie Database) gewählt. Sie ist außerordentlich gut dokumentiert und bietet mehr Möglichkeiten als für diese Anwendung überhaupt notwendig. Beispielsweise lassen sich Filme bewerten, nach Genre Filme suchen oder auch Trailer anzeigen. Sie ist gratis für jeden zur Nutzung verfügbar, solange in der Anwendung angegeben ist, dass diese Film-Datenbank verwendet wurde. Die Nutzung wiederum ist unbegrenzt, also es existiert keine Abfragenlimitierung<sup>54</sup>.

---

<sup>51</sup>Quelle: <https://www.omdbapi.com/>, letzter Zugriff: 20. März 2021

<sup>52</sup>Quelle: <https://imdb-api.com/>, letzter Zugriff: 20. März 2021

<sup>53</sup>Quelle: <https://rapidapi.com/amrelrafie/api/movies-tvshows-data-imdb>, letzter Zugriff: 20. März 2021

<sup>54</sup>Quelle: <https://developers.themoviedb.org/3/>, letzter Zugriff: 20. März 2021

## 5 Serverseitige Implementierung

Unter dem Begriff „Backend“ versteht man Komponenten eines digitalen Systems, die den Betrieb des Programms ermöglichen und vom Benutzer nicht ersichtlich sind.

Eine Backend-Anwendung kann direkt mit dem Frontend interagieren und dessen Benutzerdienste unterstützen, sowie die Schnittstellen mit allen erforderlichen Ressourcen anbieten.

### 5.1 Datenbank

#### 5.1.1 Bereitstellung der Datenbank

Über die offizielle Seite der MongoDB-Hersteller wird das Community-Installationspaket heruntergeladen und ausgeführt<sup>55</sup>. Außerdem wird die Kommandozeilenanwendung MongDBShell<sup>56</sup> und die graphische Benutzeroberfläche MongoDBCompass installiert<sup>57</sup>.

Nach erfolgreichem Initiieren eines Replica-Sets<sup>58</sup> kann der mongod-Prozess unter Angabe des zu verwendenden Replica-Sets, des Ports und des Datenspeicherpfads gestartet werden.

Standardmäßig wird bei Installation von MongoDB eine Konfigurationsdatei erstellt, dessen Name und Verzeichnis abhängig vom benutzten Betriebssystem sind. Auf diese Datei wird bei fehlender Angabe im mongod-ausführendem Kommando zugegriffen<sup>59</sup>.

Sie ermöglicht das Konfigurieren der zu nutzenden Storage-Engine. Dementsprechend wird als Storage Engine die WiredTiger-Engine eingestellt. Über die graphische Benutzeroberfläche MongoDBCompass wird nach erfolgreicher Verbindung mit dem MongoDB-Datenbanksystem eine neue Datenbank hinzugefügt namens 'StreamSwipeDatabase'. Ihr werden die benötigten Collections hinzugefügt:

- movies
- users
- matches
- swipes

#### 5.1.2 Importieren der Film- und Städtedaten

Um Nutzer innerhalb einer Stadt miteinander matchen zu können, werden Daten der realen Städtenamen auf der Datenbank benötigt. Auf der Webseite „datenbörse.net“ wird eine Liste deutscher Städtenamen zur Verfügung gestellt<sup>60</sup>. Die von uns genutzte Filmdatenbank TMDb bietet eine Liste der datenbankseitig vorhandenen Filme. Sie kann über die Filmdatenbankanbieter-API heruntergeladen werden<sup>61</sup>. Über MongoDBCompass werden die heruntergeladenen Städtenamen in die Datenbank als Collection 'cities' und die Filminformationen in die 'movies'-Collection importiert.

<sup>55</sup> MongoDB Download: <https://www.mongodb.com/try/download/community>

<sup>56</sup> MongoDBShell Download: <https://www.mongodb.com/try/download/shell>

<sup>57</sup> MongoDBCompass Download: <https://www.mongodb.com/try/download/compass>

<sup>58</sup> Vergleiche Replica Set: <https://docs.mongodb.com/manual/tutorial/deploy-replica-set/>

<sup>59</sup> MongoDB Konfiguration: <https://docs.mongodb.com/manual/reference/configuration-options/>

<sup>60</sup> Städteverzeichnis: [https://www.datenbörse.net/item/Liste\\_von\\_deutschen\\_Städtenamen\\_.csv](https://www.datenbörse.net/item/Liste_von_deutschen_Städtenamen_.csv)

<sup>61</sup> TMDB-API Daily File Exports: <https://developers.themoviedb.org/3/getting-started/daily-file-exports>

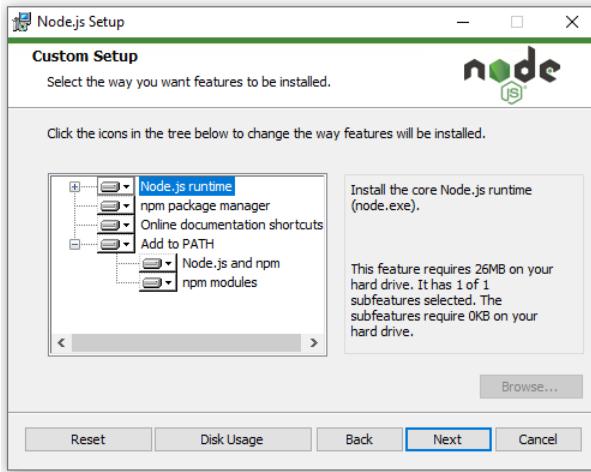


Abbildung 26: Node.JS Installation

## 5.2 StreamSwipe-Webserver

Die Serverzuständigkeiten des StreamSwipe-Webservers lassen sich in folgende Aspekte zusammenfassen:

- Empfangen und Beantworten der Clientanfragen
- Routing der Anfragen zu den entsprechenden Abhandlungsroutinen
- Überprüfen der Identität des Clients
- Kommunikation zur Datenbank für die persistente Speicherung der Zustände der Nutzer und ihrer Präferenzen, Swipes und Matches.
- Matching-Algorithmus

In den folgenden Unterkapiteln wird auf die Einrichtung des Node.js-Webservers und der MongoDB-Datenbank, sowie auf die Erstellung der sicheren Kommunikationsschnittstelle und auf die Implementierung der serverzuständigen Funktionalitäten eingegangen. Der StreamSwipe-Webserver wird im weiteren Text in der Kurzform als Webserver bezeichnet.

### 5.2.1 Bereitstellung des Webservers

Zunächst wird das Node.js-Installationspaket aus der offiziellen Seite der Hersteller heruntergeladen und ausgeführt, wie in Abbildung 26 dargestellt. Hierbei werden sowohl die Laufzeitumgebung für Node.js, als auch der npm package manager installiert (siehe nächste Abbildung). Zusätzlich wird bei der Installation ausgewählt, dass Node.js sowie npm und dessen Module zu den Umgebungsvariablen hinzugefügt werden. Dabei werden Variablen unter ihrem Applikationsnamen gespeichert und ihre entsprechenden Datei-Pfade hinterlegt. Über die Benutzung dieser Umgebungsvariablen ist ein schneller Zugriff über ein Terminal beziehungsweise einer anderen Applikation gewährleistet.

Nach der Installation von Node.js (siehe Abbildung 26) kann das Projekt mithilfe des Befehls „npm init“ im Terminal initialisiert werden. Hier werden nacheinander Input für relevante Projektaspekte wie dem Projektnamen, der Initialversion, der Startprogrammdatei oder dem GIT-Repository abgefragt. Im Anschluss wird im aktuellen Verzeichnis eine Datei „package.json“ erstellt, bei der es sich um eine Manifest-Datei im JSON-Format handelt, die unter anderem die benötigten Pakete sowie dessen Version, als auch projektspezifische Meta-Informationen wie

den Projektnamen, die Projektversion, die Projektbeschreibung und den Autor enthält. Im Anschluss an die Initialisierung werden die benötigten Pakete installiert. Dafür wird der Befehl „npm install“ in Kombination mit dem angeforderten Modul genutzt. Nach der ersten Installation eines Moduls wird im Hauptverzeichnis des Projekts automatisch ein Ordner „node\_modules“ erzeugt. Dieser enthält die Quelldaten der Node.js-Module.

Da die Funktionalität, die nodemon bietet, nur in der Entwicklung benötigt wird, wird in der Datei „package.json“ ein Entwicklungsskript „devStart“ definiert. Skripte erlauben das automatische Starten von anderen Applikationen. Über „npm run“ in Kombination mit dem auszuführenden Skript wird die Hauptapplikation über die Datei, die im package.json unter „main“ hinterlegt ist, zusammen mit den Applikationen, welche im package.json unter dem entsprechenden Skript aufgezählt sind, gestartet.

Als Applikationsstartpunkt wird die Datei „server.js“ erzeugt und im package.json unter main hinterlegt.

```

1  {
2      "name": "StreamSwipeServer",
3      "version": "1.0.0",
4      "description": "Our Backend-Server for the StreamSwipe Mobile
5          Application",
6      "main": "server.js",
7      "scripts": {
8          "start": "",
9          "devStart": "nodemon"
10     },
11     "author": "Robin Meckler, Vincent Schreck, Leon Gieringer",
12     "license": "-",
13     "dependencies": {
14         "express": "^4.17.1",
15         "firebase-admin": "^9.5.0",
16         "mongoose": "^5.11.17",
17         "node-cron": "^2.0.3",
18         "dotenv": "^8.2.0"
19     },
20     "devDependencies": {
21         "nodemon": "^2.0.7"
22     }
}

```

Listing 28: Datei package.json

### 5.2.2 Geplante Architektur

Die Software für den StreamSwipe-Server wird in Komponenten/Module aufgeteilt. Vorteile dieser Modularisierung sind, dass die einzelnen Module schneller verstanden und dementsprechend leichter überarbeitet werden können. Außerdem können Codeduplizierungen vermieden werden und Softwarekomponenten an verschiedenen Punkten im Code wiederverwendet werden. Abbildung 27 stellt die Architektur des Webservers dar. Die Route-Komponenten dienen als Empfangsschnittstelle für HTTPS-Anfragen. Die tatsächlichen Abhandlungsroutinen finden in den Controller-Komponenten statt, welche zum Zugriff auf die Datenbank auf die Service-Komponenten zugreifen und die HTTPS-Antworten zurückschicken.

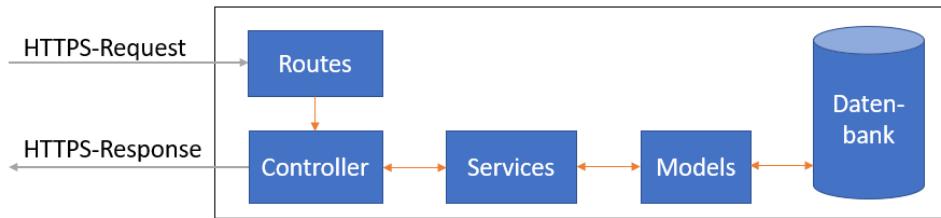


Abbildung 27: Webserver Architektur

### 5.2.3 Sichere Kommunikation

Das http-Modul ermöglicht eine Kommunikation über das http-Protokoll.

```

1  {
2    const app = express();
3    app.use(express.json());
4    var httpServer = http.createServer(app);
5    httpServer.listen(process.env.HTTP_PORT, () =>
6      console.log("HTTP-Server started on " + process.env.HTTP_PORT));
7  }
  
```

Listing 29: Einfache Verbindung

Dabei werden jedoch die Daten unverschlüsselt versendet. Um ausreichend Datenschutz zu gewährleisten, wird stattdessen das https-modul genutzt<sup>62</sup>.

Benötigt für einen HTTPS-Server werden ein Sicherheitszertifikat und ein privater Schlüssel, die zunächst mithilfe des Tools OpenSSL erzeugt werden. Dabei ist zu beachten, dass während der Entwicklungsphase das Zertifikat nicht von einer zuständigen Zertifikatsstelle signiert wurde und somit von anderen Gegenstellen nicht akzeptiert wird.

In der Anwendung wird zunächst ein Objekt 'httpsOptions' erzeugt, das unter dem Attribut 'cert' das generierte Sicherheitszertifikat und unter dem Attribut 'key' den privaten Schlüssel enthält. Anschließend wird über die Funktion 'createServer' des https-Objekts der https-Server gestartet, woraufhin ein Objekt vom Typ https.Server zurückgegeben wird<sup>63</sup>. Diesem Server-objekt wird über seine Methode 'listen' aufgefordert, auf eingehende Nachrichten in dem als Parameter übergebenen Port einzugehen.

```

1  {
2    ...
3    const https = require("https");
4    const httpsOptions = {
5      cert: fs.readFileSync('sslcert/server.crt', 'utf8'),
6      key: fs.readFileSync('sslcert/server.key', 'utf8')
7    }
8    var httpsServer = https.createServer(httpsOptions, app);
9    httpsServer.listen(process.env.HTTPS_PORT, () => {console.log(
  "HTTPS - Server started on " + process.env.HTTPS_PORT)});
  
```

<sup>62</sup>Siehe Dokumentation: <https://nodejs.org/api/https.html>, letzter Zugriff: 24. April 2021

<sup>63</sup>Siehe Dokumentation: [https://nodejs.org/api/https.html#https\\_class\\_https\\_server](https://nodejs.org/api/https.html#https_class_https_server), letzter Zugriff: 24. April 2021

10 }

Listing 30: Gesicherte Verbindung

### 5.2.4 Datenbankverbindung

Wie bereits erwähnt, wird das Modul „mongoose“ für die Verbindung mit der MongoDB-Datenbank verwendet, wie im Beispiel 31 dargestellt. Da der Quellcode in anderen Dateien hinterlegt ist, muss für den Zugriff auf dessen Funktionalitäten das entsprechende Modul zunächst inkludiert werden. Dazu wird die require-Methode aufgerufen, die ein Objekt zurückgibt, dass die aus dem Modul exportierten Methoden enthält und im Folgenden als Variabel mit dem Namen „mongoose“ gespeichert wird.

Über die connect-Methode des zurückgelieferten Objekts wird nun bei Parameterübergabe der Datenbank-URL versucht, eine Verbindung aufzubauen. Dabei wird unter der Objekt-Membervariable „connection“ ein Objekt vom Typ „Connection“ hinterlegt, über das bei erfolgreicher Verbindung mit der Datenbank kommuniziert werden kann und das nachfolgend unter der Variable „database“ abgespeichert ist.

```

1  {
2    const mongoose = require('mongoose');
3    let database = null;
4
5    async function startDatabase() {
6      await mongoose.connect(process.env.DATABASE_URL,
7        {useNewUrlParser: true,
8         useUnifiedTopology: true});
9      database = mongoose.connection;
10     database.on('error',(error) => console.log(error));
11     database.on('open',(error) => console.log('Connected to DB'))
12   }
13
14  async function getDatabase() {
15    if (!mongoose.connection) await startDatabase();
16    return database;
17  }
18
19  module.exports = {
20    getDatabase,
21    startDatabase
22  }

```

Listing 31: Verbindung zur MongoDB-Datenbank

### 5.2.5 Datenbankmodelle und Schemata

Ein Model in Mongoose ist ein aus einer Schemadefinition erstellter Konstruktor, aus denen Objekte instanziert werden können. Diese Instanzen stehen in direkter Verbindung zu den jeweiligen Collections der verbundenen Datenbank und enthalten Methoden für die persistente Speicherung, Bearbeitung oder Löschung.

Listing 32 zeigt den Aufbau des Schemas für die Swipe-Collection.

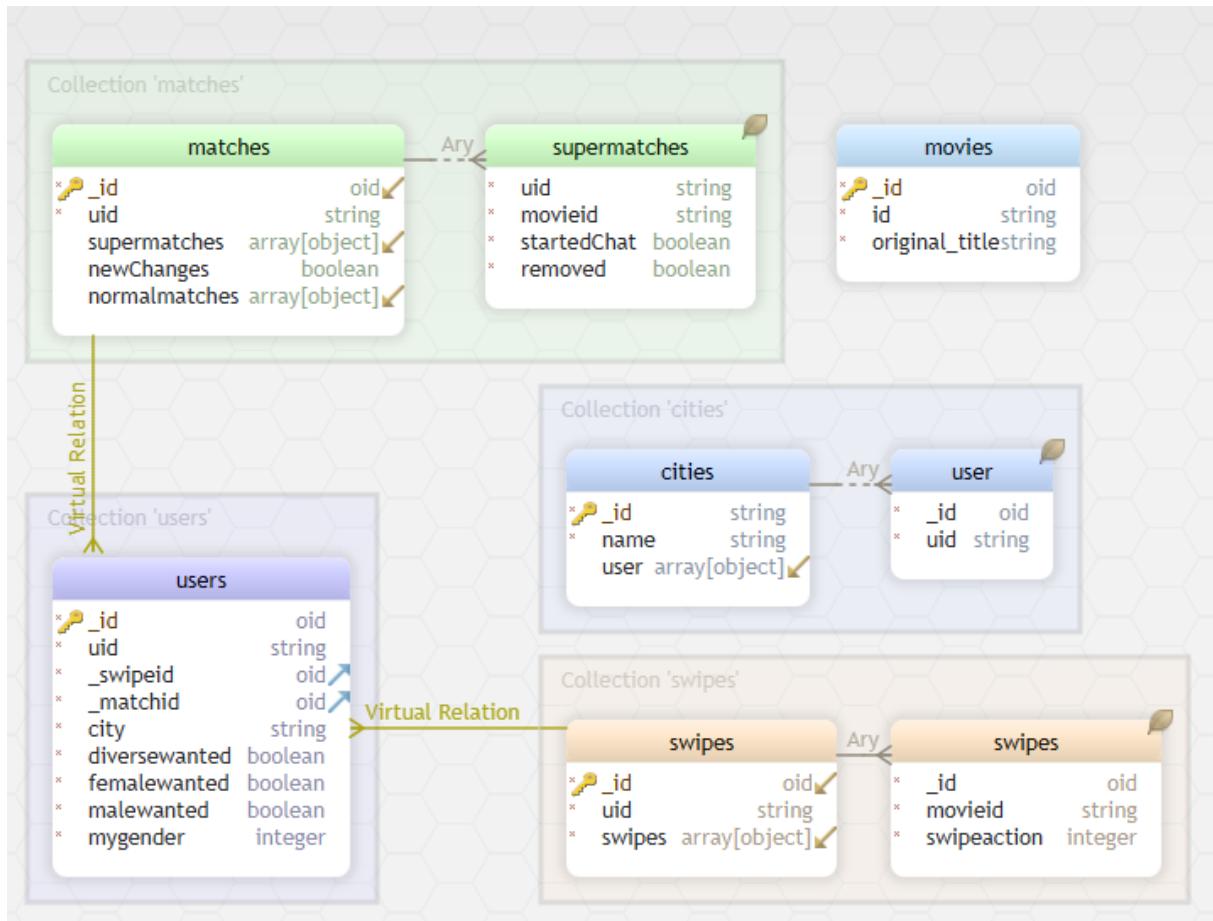


Abbildung 28: MongoDB - Aufbau der Collections und Beziehungen

```

1 const mongoose = require('mongoose')
2
3 const swipeSchema = new mongoose.Schema({
4   uid: {
5     type: String,
6     required: true
7   },
8   swipes : [
9     { movieid: { type: String },
10       swipeaction: {type: Number}}]
11 }
12
13 module.exports = mongoose.model('Swipe', swipeSchema)

```

Listing 32: Swipe Schema und Model

Die einzelnen Schemata wurden nach dem in Abbildung 28 dargestellten Aufbau für jede Collection in separate Dateien unter dem Verzeichnis '/database/models' erstellt (siehe Abbildung 29). Jede Datei exportiert dabei das aus dem zugehörigen Schema erzeugte Model.

### 5.2.6 Datenbankzugriff

Für den Datenzugriff auf die Datenbank wurden zu jeder Collection Service-Module unter dem Verzeichnis '/services' erstellt, die entsprechenden Zugriff gewähren. Dafür wurden innerhalb



Abbildung 29: Node.js Server - Models Struktur



Abbildung 30: Node.js Server - Services Struktur

der Service-Module die benötigten Zugriffsfunktionen implementiert (siehe Abbildung 30).

#### 5.2.6.1 Movie Service

Die Funktionen des Moduls movieService.js bieten innerhalb der Projektumgebung den Zugriff auf bestimmte Operationen, die auf die Collection movies angewandt werden. Das Funktionspektrum begrenzt sich für diesen Service auf die Funktion 'FindMovieExcept', die in Listing 33 zu sehen ist.

**FindMoviesExcept:** Diese Funktion erhält als Parameter 'excludedMovies' eine Liste von MovieID's und als Parameter 'amount' einen Integerwert. Über die Find-Funktion des importierten Movie-Models wird eine über den Wert von 'amount' begrenzte Anzahl an Movie-Dokumenten ausgelesen, deren IDs nicht in der übergebenen 'excludedMovies'-Liste vorhanden sind. Für das Filtern wird die 'nin'-Operation verwendet<sup>64</sup>.

Sollte die Datenbank bei der Datenauslese einen Fehler zurückgeben, wird dieser über den try-catch-Block gefangen und an im Aufrufstack liegende Funktion über das Schlüsselwort throw weitergeleitet.

```

1 const Movie = require('../database/models/movie')
2
3 async function FindMoviesExcept(excludedMovies, amount) {
4     var movies;
5
6     try{
7         movies = await Movie.find({ id: { $nin: excludedMovies } })
8             .limit(amount);
9     }
10    catch(err){throw err;}
11
12    return movies;
13}

```

<sup>64</sup>Siehe Dokumentation: <https://docs.mongodb.com/manual/reference/operator/query/nin/>, letzter Zugriff: 26. April 2021

```
14 module.exports.FindMoviesExcept = FindMoviesExcept;
```

Listing 33: movieService.js - FindMoviesExcept

### 5.2.6.2 User Service

Die Funktionen des Moduls userService.js bieten innerhalb der Projektumgebung den Zugriff auf bestimmte Operationen, die auf die Collection User angewandt werden. Dieser Service import das User-Model.

**CreateUser:** Die in Listing 34 dargestellte Funktion erhält sämtliche Eigenschaften, die im User-Schema beschrieben sind, als Parameter. Diese Funktion wird in der Projektumgebung in Zusammenhang mit mindestens einer weiteren datenbankzugreifenden Funktion aufgerufen. Im Sinne einer Transaktion müssen sie als atomare Operationen ausgeführt werden, um den Datenbestand konsistent zu halten. Daher wird ein Session-Objekt als Parameter mitgeliefert. Innerhalb der Funktion wird über die Create-Funktion des importierten User-Models ein neuer Eintrag in der User-Collection der Datenbank erstellt.

```
1 async function CreateUser(uid, swipeid, matchid, city, malewanted
  , femalewanted, diversewanted, mygender, session) {
2   try {
3     return (await User.create([
4       _id: mongoose.Types.ObjectId(),
5       uid: uid,
6       _swipeid: swipeid,
7       _matchid: matchid,
8
9       city: city,
10      malewanted: malewanted,
11      femalewanted: femalewanted,
12      diversewanted: diversewanted,
13      mygender: mygender
14    ])
15    , { session: session })) [0];
16  }
17  catch (Exception) {
18    throw Exception;
19  }
20}
```

Listing 34: User Service - CreateUser

**CheckExistence:** Wie in Listing 36 zu sehen prüft diese Funktion, ob innerhalb der User-Collection ein User mit entsprechendem Wert für die Eigenschaft 'uid', die als Parameter übergeben wird, existiert und gibt entsprechend den boolschen Wert 'true' bei Vorhandensein beziehungsweise 'false' bei Nicht-Vorhandensein zurück.

```
1 async function CheckExistence(uid) {
2   return await User.exists({ uid: uid });
3 }
```

Listing 35: User Service - CheckExistence

**GetCityFromUser:** Diese Funktion gibt den Eintrag der Eigenschaft 'city' eines Dokuments zurück, dessen 'uid'-Attribut mit dem übergebenen 'uid'-Parameter übereinstimmt, wie in Listing 36 zu sehen ist.

```

1  async function GetCityFromUser(uid) {
2      try {
3          var user = await User.findOne({ 'uid': uid });
4          if (user) {
5              return user.city;
6          }
7          else throw { message: "No user Found" + uid };
8      }
9      catch (err) { console.log(err); throw err; }

```

Listing 36: User Service - CheckExistence

**ChangeCityFromUser:** Die in Listing 37 dargestellte Funktion führt ein Update auf einem User-Dokument aus, dessen 'uid'-Eigenschaft mit dem gleichnamigen übergebenem Parameter übereinstimmt. Dabei wird die 'city'-Eigenschaft innerhalb des User-Dokuments auf den Wert des gleichnamigen Parameters aktualisiert.

```

1  async function ChangeCityFromUser(uid, city, session) {
2      try {
3          if (CheckExistence(uid)) {
4              await user.UpdateOne(
5                  { 'uid': uid },
6                  { city: city },
7                  { session: session });
8          }
9          else throw { message: "No User Found" + uid }
10     }
11     catch (err) { console.log(err); throw err; }

```

Listing 37: User Service - ChangeCityFromUser

**ChangeGenderWantedFromUser:** Diese Funktion erfüllt die gleiche Funktionalität wie die ChangeCityFromUser-Funktion mit dem Unterschied, dass statt der 'city'-Eigenschaft die Eigenschaften 'malewanted', 'femalewanted' und 'diverswanted' aktualisiert werden.

**ChangeGenderFromUser:** Die Funktion ChangeGenderFromUser erfüllt die gleiche Funktionalität wie die ChangeCityFromUser-Funktionalität mit dem Unterschied, dass statt der 'city'-Eigenschaft die 'mygender'-Eigenschaft aktualisiert wird.

### 5.2.6.3 Match Service

Die Funktionen des Moduls matchService.js bieten innerhalb der Projektumgebung den Zugriff auf bestimmte Operationen, die auf die Collection matches angewandt werden. Dieser Service import das Match-Model.

**CreateUserMatchDocument:** Die in Listing 39 dargestellte Funktion überprüft zunächst, ob ein Match-Dokument mit übergebener 'uid'-Eigenschaft bereits existiert. Ist dies nicht der Fall, wird ein neues Match-Dokument in der matches-Collection erzeugt.

```

1  async function CreateUserMatchDocument(uid, session) {
2      if (!(await Match.exists({ uid: uid }))) {
3          try {
4              return (await Match.create([
5                  _id: mongoose.Types.ObjectId(),
6                  uid: uid,
7                  swipes: []
8              ], { session: session })) [0];
9          }
10         catch (Exception) {
11             throw Exception;
12         }
13     }
14     else
15         throw { message: "Match already exists for " + uid };
16
17 }
```

Listing 38: Match Service - CreateUserMatchDocument

**CheckMatchExists:** Wie in Listing 39 zu sehen, überprüft diese Funktion ob ein Match-Dokument existiert, welches den übergebenen 'uid'-Eigenschaftswert hat sowie innerhalb seiner 'supermatches'- oder 'normalmatches'-Liste die übergebene 'matchedUid' enthält. Zurück wird ein entsprechender boolescher Wert geschickt.

```

1  async function CheckMatchExists(uid, matchedUid, session) {
2      try {
3          var match = await Match.findOne({ uid: uid, 'supermatches.
4              .uid': matchedUid }).session(session)
5
6          if (match) { return true; }
7          else {
8              var match = await Match.findOne({ uid: uid, '.
9                  normalmatches.uid': matchedUid }).session(session)
10
11             if (match) return true;
12             else return false;
13         }
14     }
15     catch (err) { console.log(err); throw err; }
16 }
```

Listing 39: Match Service - CreateMatchDocument

**AddNormalMatchToUser:** Die Funktion AddNormalMatchToUser, welche in Listing 40 zu sehen ist, fügt der 'normalmatches'-Liste eines Match-Dokuments ein neues Objekt mit den Eigenschaften 'uid', 'matchUid' und 'movieid', die ihren Wert über die übergebenen Funktionsparameter erhalten. Des Weiteren wird das Attribut 'startedChat' und 'removed' jeweils mit dem booleschen Standardwert 'false' hinzugefügt. Außerdem wird die Eigenschaft 'newChanges' des Match-Dokuments auf true gesetzt.

```

1  async function AddNormalMatchToUser(uid, matchedUid, movieid,
2                                     session) {
```

```

2  try {
3    if (Match.exists{ 'uid': uid }) {
4      await Match.findOneAndUpdate(
5        { 'uid': uid },
6        { newChanges: true,
7          $push: { normalmatches: { uid: matchedUid,
8            movieid: movieid,
9            startedChat: false,
10           removed: false } }},
11        { session: session });
12    } else throw { message: "No Match Found" };
13  catch (err) { console.log(err); throw err; }
14 }

```

Listing 40: Match Service - AddNormalMatchToUser

**AddSuperMatchToUser:** Die Funktion unterscheidet sich von 'AddNormalMatchToUser' nur in dem Aspekt, dass ein neuer Eintrag in die 'supermatches'- statt der 'normalmatches'-Liste hinzugefügt wird.

**GetMatches:** Die Funktion empfängt eine 'uid' als Parameter und gibt ein entsprechendes Match-Dokument zurück, sofern es existiert.

**SuperMatchMarkAsRemoved:** Innerhalb der Funktion wird, wie in Listing 41 dargestellt, anhand des übergebenen 'uid' und 'matchUid' der Eintrag in der 'supermatches'-Liste angepasst. Dabei wird der Wert für 'removed' auf true gesetzt. Außerdem wird 'newChanges' des betroffenen Match-Dokuments auf true gesetzt.

```

1  async function SuperMatchMarkAsRemoved(uid, matchUid, session) {
2    try {
3      var match = await Match.findOneAndUpdate({ uid: uid, "
4        supermatches.uid": matchUid },
4        { "$set": { newChanges: true,
5          "supermatches.$..removed": true }},
6        { session: session });
7      if (!match) {
8        throw { message: "No Match Found: " +
9          uid + "matching: " + matchUid }};
10    } catch (err) { console.log(err); throw err; }
11  }

```

Listing 41: Match Service - SuperMatchMarkAsRemoved

**NormalMatchMarkAsRemoved:** Die Funktion unterscheidet sich von 'SuperMatchMarkAsRemoved' nur in dem Aspekt, dass der Eintrag in der 'supermatches'- statt der 'normalmatches'- Liste angepasst wird.

**MatchesReceived:** Anhand einer übergebenen 'uid' und 'matchUid' wird die Eigenschaft 'newChanges' eines entsprechenden Match-Dokuments in der Match-Collection auf den booleschen Wert false gesetzt.

#### 5.2.6.4 Swipe Service

Die Funktionen des Moduls swipeService.js bieten innerhalb der Projektumgebung den Zugriff auf bestimmte Operationen, die auf die Collection swipes angewandt werden. Dieser Service import das Swipe-Model.

**CreateUserSwipeDocument:** Die Funktion erfüllt die gleiche Funktionalität wie die 'CreateUserMatchDocument'-Funktionalität des Match-Services mit dem Unterschied, dass anstelle eines Match-Dokuments ein Swipe-Dokument in der swipes-Collection erstellt wird.

**AddSwipe:** Die in Listing 42 dargestellte Funktion erstellt zunächst ein 'swipe'-Objekt mit den Eigenschaften 'movieid' und 'swipeaction' und entnimmt die Werte dafür aus den gleichnamigen übergebenen Parametern. Wenn ein Swipe-Dokument mit übergebener 'uid' existiert, wird überprüft ob das Dokument in der 'swipes'-Liste bereits ein Eintrag mit entsprechender movieid enthält. Ist dies der Fall, wird die 'swipeaction'-Eigenschaft auf den Wert des übergebenen gleichnamigen Parameters aktualisiert. Ansonsten wird der Liste das zu Beginn erstellte 'swipe'-Objekt hinzugefügt. Letzlich wird das Swipe-Dokument über die Save-Funktion des Save-Models gespeichert.

```

1  async function AddSwipe(uid, movieid, swipeaction) {
2    var swipe = { movieid: movieid, swipeaction: swipeaction };
3    var dbSwipe;
4    if (Swipe.exists({ 'uid': uid })) {
5      try {
6        dbSwipe = await Swipe.findOne({ 'uid': uid });
7
8        //Überprüfe Vorhandensein des Swipes
9        var index = dbSwipe.swipes.findIndex(x => x.movieid ===
10          movieid);
11
12        if (index >= 0) {
13          if (dbSwipe.swipes[index].swipeaction != swipeaction)
14            dbSwipe.swipes[index].swipeaction = swipeaction; }
15          else { dbSwipe.swipes.push(swipe); }
16          dbSwipe.save();
17        } catch (err) { throw err; }
18      } else { throw "No Swipe available for this uid " + uid; }
19      dbSwipe.save();
20      return swipe;
21    }

```

Listing 42: Swipe Service - AddSwipe

**RequestSwipes:** Die Funktion empfängt eine 'uid' als Parameter und gibt ein entsprechendes Match-Dokument zurück, sofern es existiert.

**RequestSuperlikeSwipes:** Innerhalb dieser in Listing 43 dargestellten Funktion kommt es zum Einsatz einer Aggregation. Dabei kommt es zum Einsatz mehrerer Pipeline-Operatoren. [] Über den 'unwind'-Operator wird gesetzt, dass für jeden Listeneintrag innerhalb der 'swipes'-Liste neue Dokumente erzeugt werden, die in die nachfolgende Pipeline-Stufen weitergeleitet werden. Anhand des 'match'-Operators werden dann die neuen Dokumente gefiltert. Letzlich wird eine Liste von Swipes zurückgeschickt, die eine 'swipeaction' von 2 (entsprechend Superli-

ke) aufweisen.

```

1  async function RequestSuperlikeSwipes(uid) {
2    var dbSwipe;
3    if (Swipe.exists({ 'uid': uid })) {
4      try {
5        dbSwipe = await (await Swipe.aggregate([
6          { $unwind: '$swipes' },
7          { $match: { uid: uid,
8                  'swipes.swipeaction': 2 } },
9          { $group: { _id: '_id',
10                  swipes: { $push: { movieid: "$swipes.movieid",
11                                swipeaction: "$swipes.
12                                swipeaction"
13                            } } } } ]));
14      if (dbSwipe.length > 0 && dbSwipe[0]) {
15        return dbSwipe[0].swipes;
16      } catch (err) { throw err; }
17    } else {throw { message: "Swipe uid not existing " + uid }; }
}

```

Listing 43: Swipe Service - RequestSuperlikeSwipes

**FindAllSwipedMoviesByUserID:** Innerhalb dieser in Listing 44 dargestellten Funktion werden für eine übergebene 'uid' sämtlich 'movieid's zurückgeben, die in der 'swipes'-Liste des entsprechenden Swipe-Dokuments vorhanden sind.

```

1 var swipedMovieIDs = [];
2
3 if (Swipe.exists({"uid": uid)) {
4   try { var dbSwipe = await FindOne(uid);
5     await dbSwipe.swipes.forEach(x => swipedMovieIDs.push(x.movieid
6       ));
6   } catch (err) { throw err; }
7 return swipedMovieIDs;
}

```

Listing 44: Swipe Service - FindAllSwipedMoviesByUserID

### 5.2.6.5 City Service

Die Funktionen des Moduls cityService.js bieten innerhalb der Projektumgebung den Zugriff auf bestimmte Operationen, die auf die Collection cities angewandt werden. Dieser Service import das Swipe-Model.

**GetAllInhabitedCities:** Diese Funktion wird im Matching-Algorithmus aufgerufen. Wie in Listing 45 zu sehen liefert er sämtliche Städte, mit mindestens zwei Nutzern.

```

1 async function GetAllInhabitedCities() {
2   var cities;
3   try{ cities = await City.find({ user : {$exists:true}, $where:
4     'this.user.length>1' }) }

```

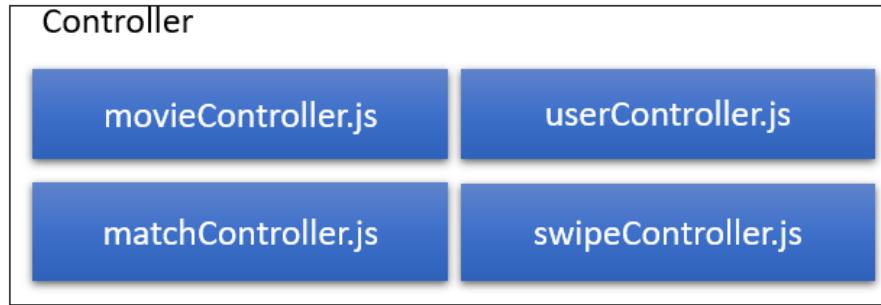


Abbildung 31: Node.js Server - Controller Struktur

```

4     catch (err) { throw err; }
5     return cities;
6 }
  
```

Listing 45: City Service - GetAllInhabitedCities

**AddUserToCity:** Diese Funktion fügt einem 'City'-Dokument eine 'uid' hinzu.

**RemoveUserFromCity:** Diese Funktion entfernt eine 'uid' aus einem 'City'-Dokument.

### 5.2.7 Controller

Die Controller enthalten die Abhandlungs routinen für die HTTPS-Anfragen (siehe Abbildung 31). Die einzelnen Funktionen empfangen jeweils das Request- und das Response-Objekt der Anfrage. Das Füllen und Zurück schicken des Response-Objekts ist ebenfalls Aufgabe der Controller. Zum Zugriff auf die Datenbank greifen sie auf die Services zu.

#### 5.2.7.1 Movie Controller

Der Movie-Controller aus Listing 46 nutzt den Movie-Service zum Zugriff auf die Datenbank.

```

1 const SwipeService = require('../services/swipeService')
2 const MovieService = require('../services/movieService')
3 const FirebaseService = require('../services/firebaseService')
4
5 exports.RequestMovies = async function(req, res){
6   ...
7 }
  
```

Listing 46: movieController.js Imports und Funktionen

Er enthält die Funktion 'RequestMovies', welche in engem Kontakt zum SwipeManager des Frontends steht und es Nutzern ermöglichen sollen, neue Filminformationen, die vom Nutzer noch nicht empfangen wurden, abzufragen.

Wie in Listing 47 zu erkennen ist, erwartet die Funktion im 'body'-Objekt des als Parameter übergebenen Request-Objekts eine Eigenschaft 'uidtoken', dessen Wert eine aus Firebase generierte Token-Referenz zur eindeutigen Authentifizierung des Nutzers ist. Der Token wird an die 'GetUID'-Funktion des Firebase-Services weitergeleitet, die bei erfolgreicher Authentifizierung die entsprechende 'uid' zurück schickt. Bei einem aufgetretenen Fehler, wie beispielsweise

einem ungültigen Token, wird das Response-Objekt mit dem Statuscode 401 sowie der aufgetretenen Fehlermeldung zurückgeschickt und die Funktion beendet. Der folgende Code kommt in weiteren Funktionen anderer Controller ebenfalls zum Einsatz, wenn eine Authentifizierung benötigen.

```

1  var uid;
2  const uidToken = req.body.uidtoken;
3  try{ uid = await FirebaseService.GetUID(uidToken); }
4  catch(Exception)
5  { res.status(401).json({title: "TOKEN ERROR", message:
    Exception}); return; }
```

Listing 47: Controller Firebase-Authentifizierung

Nach erfolgreicher Authentifizierung des Firebase-Tokens werden weitere Eigenschaftswerte aus dem Request-Body als Variablen gespeichert. Erwartet wird ein Zahlenwert 'amount', der die Anzahl der abgefragten Filme darstellt. Über die lokale Funktion 'RestrictAmount' wird geprüft, dass die begrenzende Zahl den Wert 10 nicht übersteigt. Damit soll sichergestellt werden, dass die Datenbankabfrage mit den weit über 500.000 Filmen nicht ausgelastet wird. Des Weiteren werden in der Variable 'alreadyRequestedMovieIDs' eine Liste von eindeutigen Identifizierern aus der Movie-Collection erwartet. Die Werte sollen jene Film-ID's wiederspiegeln, die bereits vom Nutzer abgefragt, aber noch nicht über einen Swipe-Request in der Datenbank hinterlegt wurden. Die vom Nutzer bereits getätigten Swipes werden über die 'FindAllSwipesByUserID'-Funktion des SwipeServices abgefragt. Zurückgegeben wird eine Liste von Film-ID's, die zusammen mit der Liste der 'alreadyRequestedMovieIDs' in die Variabel 'excludedMovieIDs' gespeichert werden.

```

1  var amount = RestrictAmount(req.body.amount);
2  var alreadyRequestedMovieIDs = req.body.
    alreadyRequestedMovieIDs;
3  var excludedMovieIDs = [];
4  var newMovies;
5
6  // Frage bereits geswipete Filme ab
7  try{ var swipedMovieIDs = await SwipeService.
    FindAllSwipesByUserID(uid) }
8  catch(err){ res.status(400).json({message: err.message});
    return; }
9  excludedMovieIDs.push(...swipedMovieIDs); }
10
11 // Speichere bereits abgefragte Filme ab
12 if(alreadyRequestedMovieIDs !== undefined &&
    alreadyRequestedMovieIDs != null)
13 { await alreadyRequestedMovieIDs.forEach(element =>
    excludedMovieIDs.push(element)); }
```

Listing 48: MovieController - RequestMovie - Excluded Movies

Letztlich wird bei vorhandenen zu exkludierenden Filmen die Funktion 'FindMoviesExcept' des Movie-Services aus Listing 49 aufgerufen. Ist die Liste 'excludedMovieIDs' dagegen leer, so wird die 'FindExactAmount'-Funktion aufgerufen. Bei Erfolg wird das Response-Objekt mit dem Statuscode 200 und den Movie-Dokumenten als JSON-Objekt im Body der Antwort zurückgeschickt.

```

1  if(excludedMovieIDs.length > 0)
2  {
3      try{ newMovies = await MovieService.FindMoviesExcept(
4          excludedMovieIDs,amount) }
5      catch(err){ res.status(400).json({message: err.message});
6          return; }
7  } else {
8      try{ newMovies = await MovieService.FindExactAmount(
9          amount); }
10     catch(err){ res.status(400).json({message: err.message});
11         return; }
12 }
13
14 res.status(200).json(newMovies);

```

Listing 49: MovieController - RequestMovie - Excluded Movies

### 5.2.7.2 User Controller

Der User-Controller bietet Funktionen, die die users-Collection der Datenbank betreffen. Sie greift dafür auf das User-Service zu. Die folgenden Funktionen greifen teils auch auf andere Collections zu. Daher werden auch die entsprechenden weiteren Services importiert.

**CreateUser:** Die Funktion erstellt ein neues User-Dokument in der users-Collection. Dafür werden gleichnamige Eigenschaften des User-Models im Request-Body der eingehenden Anfrage erwartet. Nach erfolgreicher Authentifizierung des Firebase-Tokens und Überprüfung über die 'CheckExistence'-Funktion des User-Services, ob ein User-Dokument mit der gleichen 'uid' bereits existiert, wird für die weiteren Datenbankabfragen eine Transaktion gestartet.

```

1 // 1. Starte Transaktion!
2 const session = await mongoose.startSession();
3 await session.startTransaction();

```

Listing 50: UserController - Create User - Transaktionsstart

Das dafür genutzte 'session'-Objekt wird in den weiteren Service-Funktionen übergeben. In Listing 51 wird:

- ein Swipe-Dokument über die 'CreateUserSwipeDocument'-Funktion des Swipe-Services erstellt.
- ein Match-Dokument über die 'CreateUserMatchDocument'-Funktion des Match-Services erstellt.
- ein User-Dokument über die 'CreateUser'-Funktion des User-Services mit entsprechender Parametrisierung erstellt.
- die 'uid' der entsprechenden Stadt über die 'AddUserToCity'-Funktion des City-Services hinzugefügt.

Nur wenn alle Operationen erfolgreich ausgeführt wurden, wird die Transaktion über die 'commitTransaction'-Methode ausgeführt. Damit soll sichergestellt sein, dass einzelne, zusammenhängende Dokumente und Informationen nur im Ganzen erstellt werden. Bei Misserfolg einer Operation wird die komplette Transaktion über die 'abortTransaction'-Methode abgebrochen.

```

1 try {
2     // 2. Erstelle SWIPE-Dokument
3     var createdSwipe = await SwipeService.
4         CreateUserSwipeDocument(uid, session);
5
6     // 3. Erstelle MATCH-Dokument
7     var createdMatch = await MatchService.
8         CreateUserMatchDocument(uid, session);
9
10    // 4. Erstelle USER-Dokument
11    var createdUser = await UserService.CreateUser(uid,
12        createdSwipe._id, createdMatch._id, city, malewanted,
13        femalewanted, diversewanted, mygender, session);
14
15    // Fuege User zu City hinzu
16    if (createdUser._id)
17        await CityService.AddUserToCity(uid, city, session);
18
19} catch (Exception) {
20    res.status(501).json({
21        title: "Server-User Creation Error", message: Exception
22    });
23    // Fehler => Transaktion abbrechen
24    await session.abortTransaction(); }
25
session.endSession();

```

Listing 51: UserController - Create User - Dokumente erstellen

**ChangeUser:** Diese Funktion erlaubt einem Nutzer, seine Eigenschaften innerhalb der Datenbank zu aktualisieren. Die Schritte sind in Listing 52 zu sehen. Dafür wird nach erfolgreicher Authentifizierung eine Transaktion gestartet. Im folgendem Try-Block werden die einzelnen Operationen dargestellt, die für eine erfolgreiche Transaktion ausgeführt werden. Über das User-Service werden die Methoden 'ChangeGenderWantedFromUser' und 'ChangeGenderFromUser' aufgerufen. Anschließend muss der Stadtseintrag angepasst werden, welcher an mehreren Stellen in der Datenbank geändert werden muss. So muss zunächst die 'uid' aus dem alten 'city'-Dokument entfernt (Schritt 3 und 4) und dem Dokument hinzugefügt werden, dass dem aktualisierten Stadtwert entspricht (Schritt 5). Letzlich wird der Wert der 'city'-Eigenschaft über die 'ChangeCityFromUser' des User-Services angepasst.

```

1 //1. Änderung an GenderWanted
2 await UserService.ChangeGenderWantedFromUser(uid, malewanted,
3     femalewanted, diversewanted, session);
4
5 //2. Änderung an Gender
6 await UserService.ChangeGenderFromUser(uid, mygender, session);

```

```

7 //3. Frage vorherige Stadt ab
8 var oldCity = await UserService.GetCityFromUser(uid, session);
9
10 //4. Loesche Nutzer aus vorheriger Stadt
11 await CityService.RemoveUserFromCity(uid, oldCity, session);
12
13 //5. Fuege Nutzer zu neuer Stadt hinzu
14 await CityService.AddUserToCity(uid, newCity, session);
15
16 //6. Aktualisiere den Stadteintrag beim Nutzer
17 await UserService.ChangeCityFromUser(uid, newCity, session);
18
19 await session.commitTransaction();
20 res.status(200).json();

```

Listing 52: UserController - Change User

**InfoUser:** Diese Funktion dient dazu, nach erfolgreicher Authentifizierung die gespeicherten Eigenschaft und ihre Werte des abgefragten User-Dokuments zu erhalten. Dafür wird die 'GetInfoFromUser'-Methode des User-Services aufgerufen.

### 5.2.7.3 Match Controller

Der Match-Controller bietet Funktionen, die die matches-Collection der Datenbank betreffen. Sie greift dafür vorrangig auf den Match-Service zu.

**RequestMatches:** Das Frontend erlaubt es, Matches anzeigen zu lassen. Dafür bietet die in Listing 53 dargestellte Funktion Informationen über das Match-Dokument des jeweiligen Nutzers. Nach erfolgreicher Authentifizierung wird das zugehörige Match-Dokument über die 'GetMatches'-Methode abgefragt. Das Dokument enthält zwei Listen: supermatches und normalmatches. Beide enthalten jeweils eine Eigenschaft 'removed'. Ist diese auf true gesetzt, so soll impliziert werden, dass der Nutzer das Match entfernt hat. Folglich soll das gelöschte Match nicht mehr angezeigt werden. Daher werden die beiden Listen über die 'filter'-Funktion nach den nicht entfernten Matches gefiltert. Bei Erfolg wird ein JSON-Objekt mit beiden Listen und der Eigenschaft 'newChanges' aus dem Match-Dokument zurückgesendet.

```

1 var match = await MatchService.GetMatches(uid);
2 var filteredSupermatches = match.supermatches.filter(match =>
3   match.removed == false)
4 var filteredNormalmatches = match.normalmatches.filter(match =>
5   match.removed == false)
6 res.status(200).json({ newChanges: match.newChanges,
7                     supermatches: filteredSupermatches,
8                     normalmatches: filteredNormalmatches } );

```

Listing 53: MatchController - RequestMatches

**DeleteSupermatch:** Hier wird die 'SuperMatchMarkAsRemoved'-Methode des Match-Services aufgerufen, um die 'removed'-Eigenschaft des entsprechenden Supermatches auf true zu setzen. Dieser Supermatch wird folglich nicht mehr über die 'RequestMatches'-Funktion zurückgegeben.

**DeleteNormalmatch:** Gleiches Prinzip wie 'DeleteSupermatch' mit Normalmatches.

**Received:** Hier wird die 'newChanges'-Eigenschaft auf false gesetzt. Es wird impliziert, dass der Nutzer den aktuellsten Stand der Matches hat.

**Trigger:** Diese Funktion ruft MatchManager.startMatching() auf. Sie ist vorerst nur für die Entwicklung gedacht, und soll es ermöglichen, über das Frontend den Matching-Algorithmus im Backend zu starten.

#### 5.2.7.4 Swipe Controller

Der Swipe-Controller bietet eine Funktion, die die swipes-Collection der Datenbank betrifft. Hierfür greift sie auf den Swipe-Service zu. Sie bietet lediglich die Funktion **CreateSwipe** an. Sie ruft die SwipeService.AddSwipeToDB-Methode auf.

#### 5.2.8 Routing

In der Server.js werden dem Express-Objekt 'app' die einzelnen Routen für die Weiterleitung der HTTPS-Anfragen an die entsprechenden Controller hinzugefügt. Dabei werden die zu den Anfragen gehörenden Request- und Response-Objekte als Parameter an die Controller übergeben.

```

1 //Movies
2 const moviesRouter = require('./routes/movies')
3 app.use('/movies', moviesRouter)
4
5 //Users
6 const usersRouter = require('./routes/users')
7 app.use('/users', usersRouter)
8
9 //Matches
10 const matchesRouter = require('./routes/matches')
11 app.use('/matches', matchesRouter)
12
13 //Swipes
14 const swipesRouter = require('./routes/swipes')
15 app.use('/swipes', swipesRouter)
```

Listing 54: Routing in server.js

#### 5.2.8.1 Movie Router

Innerhalb des Movie Routers wird die '/movies/request' an die Funktion RequestMovie des Movie-Controller weitergeleitet.

```

1 // Importiere benoetigte Controllermodul.
2 const MovieController = require('../controllers/movieController')
3
4 // Leite Anfragen weiter an MovieController
5 router.post('/request', MovieController.RequestMovies)
```

Listing 55: Routing in movieRouter.js

### 5.2.8.2 User Router

Der User Router leitet '/users/create', '/users/change' und '/users/info' an die entsprechenden Funktionen des User-Controllers weiter.

```

1 // Importiere benoetigtes Controllermodul.
2 const UserController = require('../controllers/userController')
3
4 // Leite Anfragen weiter an UserController
5 router.post('/create', UserController.CreateUser)
6 router.post('/change', UserController.ChangeUser)
7 router.post('/info', UserController.InfoUser)

```

Listing 56: Routing in userRouter.js

### 5.2.8.3 Match Router

Innerhalb des Match Routers werden die unten dargestellten URL's an die Funktion des Match-Controller weitergeleitet.

```

1 // Importiere benoetigtes Controllermodul.
2 const MatchController = require('../controllers/matchController')
3
4 // Leite Anfragen weiter an MatchController
5 router.post('/request', MatchController.RequestMatches)
6 router.post('/deleteSupermatch', MatchController.DeleteSupermatch)
7 router.post('/deleteNormalmatch', MatchController.
8     DeleteNormalmatch)
9 router.post('/received', MatchController.Received)
9 router.post('/trigger', MatchController.Trigger)

```

Listing 57: Routing in matchRouter.js

### 5.2.8.4 Swipe Router

Der Swipe Router leitet '/swipes/create' an die entsprechende Funktionen des Swipe-Controllers weiter.

```

1 // Importiere benoetigtes Controllermodul.
2 const SwipeController = require('../controllers/swipeController')
3
4 // Leite Anfragen weiter an SwipeController
5 router.post('/create', SwipeController.CreateSwipe )

```

Listing 58: Routing in swipeRouter.js

## 5.2.9 Weitere Backendfunktionalitäten

Nachfolgend werden weitere Systemfunktionalitäten des Backends dargestellt.

### 5.2.9.1 Firebase-Service

Um unberechtigte Zugriffe zu vermeiden, findet für die nutzerbezogenen Anfragen eine Authentifizierung statt. Die erste Authentifizierung des Nutzers findet über den Login des Frontends in Firebase statt. Nachträglich muss bei Anfragen an den Webserver sichergestellt werden, dass

der Nutzer weiterhin authentifiziert ist. Ohne diesen Vorgang könnte man sich über das Schicken einer willkürlich übermittelten Nutzer-Uid fälschlicherweise als anderer Nutzer ausgeben. Für den Zugriff auf die Firebase-Authentifizierungsfunktionen wird in die firebaseService.js-Datei das Modul 'firebase-Admin' importiert<sup>65</sup>.

**Register:** Um die Anwendung bei Firebase zu registrieren, wird die Funktion 'initializeApp' des Firebase-Moduls ausgeführt. Ein aus Firebase generierter Authentifizierungsschlüssel wird dabei für den Zugriff auf die StreamSwipe-Umgebung mit übergeben. Die Register-Methode wird anschließend nach außen exportiert und zu Beginn des Serverstarts in der Server.js-Datei ausgeführt<sup>66</sup>.

```

1 var serviceAccount = require("../sslcert/streamswipe-firebase-
  adminsdk-uicyci-80bc08a5b2.json");
2 firebaseAdmin.initializeApp({
3   credential: admin.credential.cert(serviceAccount),
4   databaseURL: "https://streamswipe.firebaseio.com"
5 });

```

Listing 59: Firebase-Service Register

**UID/TokenID-Dictionary:** Um Zugriffszeiten auf die Firebase-Schnittstelle, werden in einem lokalen Dictionary aus Schlüsselwertpaaren der Zusammenhang zwischen TokenID und den UID samt ihrem Ablaufdatum zwischengespeichert.

**GetUID:** Die Funktion erwartet einen Firebase Token als Parameter 'uidtoken', welcher an die Funktion 'verifyIdToken' des FirebaseAuth-Objekts weitergeleitet wird. Zurück wird ein Objekt gegeben, dass unter anderem die 'uid' des zum Token zugehörigen Nutzers und die Ablaufzeit schickt. Nach erfolgreichem Überprüfen, ob die 'uid' tatsächlich ein Wert übermittelt bekommen hat, wird das Paar aus UidToken und Uid samt Ablaufzeit in der UID/TokenID-Dictionary gespeichert.

```

1 verifiedUid = await firebaseAdmin.auth().verifyIdToken(uidToken);
2 uid = verifiedUid.uid;
3 expireTime = verifiedUid.exp;
4 if(uid === undefined || uid == null) { throw {message: "No uid
  returned!"}; }
5 TokenIDDict[uidToken] = {uid, new Date(expireTime*1000)};
6 return uid;
7 ... //Ende Try-Catch-Block

```

Listing 60: Firebase-Service Register

**RefreshList:** Diese Funktion wird aufgerufen, um abgelaufene Token in der UID/TokenID-Dictionary zu löschen. Sie wird über das Modul TimedEvents periodisch aufgerufen. Dabei wird zu jedem Paar die aktuelle Uhrzeit und die Ablaufszeit verglichen. Stellt die Ablaufszeit ein größeren Wert dar, wird das Schlüsselwertpaar aus der Dictionary entfernt.

<sup>65</sup>Siehe Dokumentation: <https://firebase.google.com/docs/admin/setup>, letzter Zugriff: 3. April 2021

<sup>66</sup>Siehe Dokumentation: <https://firebase.google.com/docs/admin/setup#initialize-without-parameters>, letzter Zugriff: 3. April 2021

### 5.2.9.2 Timed Events

Über das 'node-cron'-Modul<sup>67</sup> können zeitlich definierte und periodische Funktionen ausgeführt werden. Dafür wird das 'node-cron'-Modul in die TimedEvents.js-Datei importiert. Die Funktionalität des Moduls wird beispielsweise für das periodische Aktualisieren der movies-Collection, das periodische Ausführen des Matching-Algorithmus und das Aufrufen der 'firebaseService.RefreshList'-Funktion zum Aktualisieren der UID/TokenID-Dictionary verwendet.

### 5.2.9.3 MatchManager

**StartMatching - Teil 1:** Die aktuelle Implementierung des Matching-Algorithmus sucht für jede Stadt Nutzerpaare, die einen gleichen Film mit einem Superlike versehen haben. Dafür werden zunächst über die 'GetAllInhabitedCities'-Funktion des City-Services die Städte in einer Liste gespeichert, die mindestens zwei Nutzer aufweisen. Folglich finden eine Verschachtelung von Iterationsabläufen zum Ausführen von Programmcode auf jedem Element einer Liste statt. In der ersten Iterationsstufe wird durch die Städte iteriert. Die zweite Iterationsstufe vom ersten bis zum vorletzten Nutzereintrag ('uid') innerhalb der aktuell iterierten Stadt. Innerhalb des zugehörigen Codeblocks wird die 'RequestSuperlikeSwipes'-Funktion des SwipeServices aufgerufen mit der 'uid' des aktuell iterierten Nutzers als Parameterübergabe. Die zurückgehaltene Liste enthält sämtliche Film-ID's, die vom Nutzer mit einem Superlike versehen wurden. Die Liste wird samt dem Index des nächsten Users in der Liste, der aktuell iterierten Stadt und dem aktuell iterierten User. Außerdem wird eine Referenz auf das 'foundSupermatches'-Objekt, dass später mit Informationen zu den errechneten Supermatches gefüllt wird, mitgegeben.

```

1 var cities = await CityService.GetAllInhabitedCities();
2
3 // City - 1. Iterationsstufe
4 for (let cityIterator = 0; cityIterator < cities.length;
5     cityIterator++) {
6     // Fuer jeden Nutzer ausser den letzten
7     var prelastSupermatchIndex = cities[cityIterator].user.length -
8         2;
9
10    // User - 2. Iterationsstufe
11    for (let user1Iterator = 0; user1Iterator <=
12        prelastSupermatchIndex; user1Iterator++) {
13        var matchingUserNextIndex = user1Iterator + 1;
14        var User1 = cities[cityIterator].user[user1Iterator];
15
16        //SUPERMATCH-CHECK
17        var user1SuperlikedMovies = await SwipeService.
18            RequestSuperlikeSwipes(User1.uid);
19        if (user1Superlikes) {
20            // Ueberpruefe Superlikes mit den nachfolgenden Usern
21            await checkSuperMatches(matchingUserNextIndex, cities[
22                cityIterator], user1SuperlikedMovies, foundSupermatches,
23                User1);
24        }
25    }
26}

```

<sup>67</sup>Siehe Dokumentation: <https://www.npmjs.com/package/node-cron>, letzter Zugriff: 26. April 2021

```
19
20     //NORMALMATCH-CHECK
21     var user1swipes = await SwipeService.RequestSwipes(User1.
22         uid);
23     if (user1swipes) {
24         await checkNormalMatches(matchingUserNextIndex, cities[
25             cityIterator], user1swipes, normalmatches, User1);
26     }
27 }
28 ... //end Try-Catch-Block
```

Listing 61: Match Manager - startMatching - Teil 1: Finde Matches

**checkSuperMatches:** Die Funktion wird im Matching-Algorithmus von der 'findMatches'-Methode aufgerufen. Es wird ausgehend vom übergebenen 'matchingUserNextIndex', welches der Index des nächsten Users nach dem aktuell iterierten Users der 2. Iterationsstufe ist, durch die restliche Nutzerliste der übergebenen Stadt 'currentCity' iteriert. Dies entspricht der 3. Iterationsstufe. Hierbei wird zunächst die lokale Funktion 'checkGenderPreference' aufgerufen, die anhand der übergebenen Nutzerobjekte prüft, ob jeweils das Geschlecht des einen Nutzers und das für das Matchen preferierte Geschlecht des anderen Nutzers übereinstimmen. Ist dies der Fall, so werden die Superlikes des zweiten Users angefragt. Abschließend werden die Superlikes-Listen beider Nutzer verglichen. Dabei wird überprüft, ob eines der MovieID's der einen Liste in der anderen vorhanden ist. Bei einem Treffer wird ein Objekt mit den 'uid' der bei Nutzer und die 'movied' des zugehörigen Films in die 'supermatches'-Liste hinzugefügt.

```

22         continue;
23     }}}}}}
```

Listing 62: Match Manager - checkSuperMatches

**checkNormalMatches:** Diese Funktion ist zum Zeitpunkt der Dokumentation nicht implementiert

**StartMatching - Teil 2:** Die 'startMatching'-Funktion wird beendet, nachdem die gefundenen Matchinformationen in den entsprechenden Match-Dokumenten gespeichert werden. Es wird durch die Liste 'foundSupermatches' durchiteriert. Dafür wird zunächst geprüft, dass ein Match der zwei Nutzer noch nicht in der Datenbank hinterlegt ist. Anschließend werden beiden zugehörigen Match-Dokumenten die gegenseitigen 'uid' in die 'supermatches'-Liste hinzugefügt. Der Vorgang wird für die 'normalmatches'-Liste wiederholt mit entsprechender Speicherung in die 'normalmatches'-Listen.

```

1 await session.startTransaction();
2
3 for (let i = 0; i < foundSupermatches.length; i++) {
4     // Ueberpruefe, ob Match vorhanden ist
5     if (!(await MatchService.CheckMatch(foundSupermatches[i].matchid1, foundSupermatches[i].matchid2, session))) {
6         // Fuege Match zu User1's Match-Dokument hinzu
7         await MatchService.AddSuperMatchToUser(foundSupermatches[i].matchid1, foundSupermatches[i].matchid2, foundSupermatches[i].movieid, session);
8
9         // Fuege Match zu User2's Match-Dokument hinzu
10        await MatchService.AddSuperMatchToUser(foundSupermatches[i].matchid2, foundSupermatches[i].matchid1, foundSupermatches[i].movieid, session);
11    }
12
13    await session.commitTransaction();
```

Listing 63: Match Manager - startMatching - Teil 2: Speichere Matches

### 5.3 Firebase

Firebase bietet viele verschiedene Werkzeuge zur Entwicklung und Überwachung von Mobil- und Webanwendungen. Nach dem Konzept nach Kapitel ?? beschränkt sich der Aufgabenbereich des Firebase Backends auf Nutzerauthentifizierung, Verwaltung der Nutzerdaten und der Chatfunktion. Die hierfür genutzten Werkzeuge werden im Folgenden besprochen.

Grundlegend muss zunächst eine Anwendung erstellt, ein Firebase Projekt aufgesetzt und diese zusammen verknüpft werden. Sobald dieser Prozess abgeschlossen ist, muss sichergestellt sein, dass Firebase in der Anwendung initialisiert wird.

#### 5.3.1 Authentifizierung

Um Nutzern eine sichere Registrierung, bzw. An- und Abmeldung ermöglichen zu können bietet Firebase das Authentifizierungswerkzeug. Dieser Backendservice verfügt über unterschiedliche Authentifizierungsmethoden. Um es zu nutzen, muss nur die jeweilige Methode ausgewählt werden.

In unserem Fall wurde die E-Mail und Passwort Authentifizierung gewählt, um unabhängig von Drittanbietern zu sein. Nun muss das SDK `firebase_auth` integriert werden und mithilfe der Funktionen `createUserWithEmailAndPassword()` bzw. `signInWithEmailAndPassword()` ein Nutzer registriert und angemeldet werden. Hierbei können mithilfe von Fehlercodes geeignete Fehlermeldungen erstellt werden. Zusätzlich besitzt die Klasse `User` das Feld `emailVerified` (Boolean) und die Methode `sendEmailVerification()`, wodurch eine Verifizierung der E-Mail über Nachrichtenvorlage durchgeführt wird. Ist ein Nutzer einmal authentifiziert, muss unterschieden werden, welchen Bildschirm er sehen darf. Hierzu wird darauf geachtet ob ein aktueller Nutzerobjekt existiert. Ist dies nicht der Fall, wird der Anmeldebildschirm angezeigt; ansonsten der Hauptbildschirm.

```

1 // Globale Instanz des Authentifizierungsservice
2 final AuthService auth = Provider.of(context).auth;
3 return FutureBuilder<User>(
4   future: auth.getCurrentUser(),
5   builder: (BuildContext context, AsyncSnapshot<User> snapshot)
6   {
7     // Existiert ein Nutzer, wird der Hauptbildschirm angezeigt
8     if (snapshot.hasData) {
9       return MessageHandler();
10    }
11    // Zeige ansonsten den Login-Bildschirm
12    else {
13      return SignUpScreen(authFormType: AuthFormType.signIn);
14    }
15  );

```

Listing 64: Anzeige abhängig ob ein aktueller Nutzer existiert

### 5.3.2 Nutzerdaten

Bei der Authentifizierung eines Nutzers wird mittels des Auth-Services ein Nutzerobjekt erstellt. Dieses kann folgende Felder besitzen:

- Einzigartige Identifikation (UID)
- E-Mail Adresse
- Namen
- Bild-URL

Es ist jedoch nicht möglich über diesen Service weitere Eigenschaften abzuspeichern. Diese müssen über zusätzliche Speicherwerkzeuge, wie beispielsweise Cloud Firestore (siehe 2.5.2) gesichert werden.

Hierzu wurde die Sammlung „users“ erstellt und bei der Registrierung für jeden Nutzer ein Dokument mit der selben ID, wie die Nutzer UID erstellt. Dadurch ist sichergestellt, dass es ein einzigartiges Dokument und der Zugriff einfach geregelt ist. Hier werden nun weitere Eigenschaften gesichert, welche teilweise ausschließlich zu Anzeigezwecken in Firebase doppelt abgespeichert werden.

- Anzeigename
- E-Mail
- Wohnort, welcher aus den wichtigsten Städten Deutschlands bestehen
- Geschlechter, nach welchen gesucht wird
- Eigenes Geschlecht
- Lieblingsfilm

Unser Nutzer jedoch speichert seine Profil und Hintergrundbilder weder im Auth-Service Nutzerobjekt noch im Cloud Firestore Dokument. Bei Auth-Service lässt sich lediglich die URL zu einem einzigen Bild hinterlegen. Bei Firestore ist es zwar möglich eine theoretisch unbegrenzte Menge an Bild URLs abzuspeichern, jedoch muss der Nutzer die Möglichkeit haben seine eigenen Bilder hochzuladen und nicht nur eine URL auf ein bereits hochgeladenes Bild abspeichern.

Hierfür bietet Firestore das Werkzeug Storage. Wie in Kapitel 2.5.3 beschrieben, können hier Nutzerinhalte hoch- und heruntergeladen werden. Dazu wird beim ersten Hochladen ein Ordner für jeden Nutzer erstellt. Darin werden daraufhin die Bilder unter dem Namen `profile-picture` oder `background-picture` jeweils abgespeichert und eventuell überschrieben. Mit einer Funktion kann über einen Boolean-Parameter entschieden werden, welches Bild somit angezeigt wird.

```

1 Future<String> getPictureFromStorage(String uid, bool
2     isProfilePicture) async {
3     try {
4         // Die Referenz auf Storage
5         Reference storage = FirebaseStorage.instance.ref();
6         // Ordner UID mit Datei profile-picture oder background-
7         // picture
8         Reference ref = storage.child(uid)
9             .child(isProfilePicture ? '/profile-picture' : '/
10                background-picture');
11        // Gebe die URL zum Download zurueck
12        return await ref.getDownloadURL();
13    } on Exception catch (e) {
14        // Fehlerbehandlung eine Ebene oberhalb
15        throw e;
16    }
17 }
```

### 5.3.3 Chatfunktion

Damit Nutzer bei einem erfolgreichen Match sich unterhalten und vielleicht auch verabreden können, muss eine Anwendung dieser Art eine Chatfunktion bieten. Diese Funktion jedoch beschränkt sich auf eine 1-zu-1 Kommunikation, es werden also keine Gruppenchats benötigt. Aus welchem Grund wird hierfür jedoch Firebase verwendet?

Da Google weltweit über Server verfügt, ist diese Chatanwendung mithilfe von Firebase direkt auch global verfügbar. Zudem würde der Backend Server, welcher die Filmdaten bereitstellt und Nutzer über Matching Algorithmen zusammenführt, zusätzlich durch Netzwerkverkehr der Chatanwendung belastet. Hierfür bietet Firebase extra auf Skalierung ausgelegte Werkzeuge, damit

sich Entwickler nicht zwingend mit diesen Problematiken auseinandersetzen müssen. Zusätzlich müsste mit einem separaten Server die Sicherheit und Wartung behandelt werden; dies wird bei Firebase direkt gemacht. Ein großer Nachteil ist jedoch, dass mit Firebase die Ende-zu-Ende Verschlüsselung nicht direkt gegeben ist und nachträglich eigenhändig oder über externe Bibliotheken eingefügt werden muss (weiterführende Informationen in Kapitel 5.3.4). Bei einem separaten Server ist dies zwar ebenfalls nicht gegeben, aber kann über das Design des Features im Voraus geregelt werden.

Grundsätzlich benötigt man für die Chaträume eine Sammlung in Firebase, welche die beteiligten Nutzer beinhaltet. Ein Dokument dieser Sammlung bekommt eine einzigartige ID bestehend aus den jeweiligen UIDs der Nutzer und zusammengefügt durch eine Unterstrich. Damit diese ID wirklich einzigartig ist und nicht zwei Chaträume mit den IDs „Nutzer1\_Nutzer2“ und „Nutzer2\_Nutzer1“ entstehen können, werden bei der Erstellung beide UIDs alphabetisch verglichen und entsprechend angeordnet (siehe Code 65).

```

1 // Vergleicht die Strings, setzt den im Alphabet vorher
  kommenden zuerst
2 if (firstUID.compareTo(secondUID) <= 0)
3   roomID = firstUID + "_" + secondUID;
4 else
5   roomID = secondUID + "_" + firstUID;
```

Listing 65: Sortierung von UIDs

In diesem Dokument werden nicht nur die UIDs der einzelnen Nutzer abgespeichert, sondern aus Anzeigegründen auch die Nutzernamen und die Raum-Identifikation selbst. Zusätzlich benötigt dieser Raum auch eine Möglichkeit Nachrichten abzuspeichern. Hierzu existiert eine Untersammlung `messages`, in welcher einzelne Nachrichten als Dokumente gespeichert werden. Diese wiederum beinhalten den eigentlichen Nachrichtentext, die UID des Senders, die des Empfängers und einen Zeitstempel. Die UIDs werden einerseits dazu benötigt, um in der Oberfläche die Nachricht links- oder rechtsbündig anzuzeigen (siehe Abbildung 38e). Andererseits werden sie für die Benachrichtigungen benötigt, also welcher Nutzer auf seinem Gerät eine Benachrichtigung erhalten soll.

Für die Benachrichtigungen wird das Werkzeug Cloud Functions verwendet. Es bietet die Ausführung von serverseitigem Code unter bestimmten Bedingungen. Um eine solche Funktion nun ausführen zu können, muss eine weitere Sammlung `unreadMessages` existieren. In diese wird beim Versenden einer Nachricht das selbe Nachrichtendokument wie in der Untersammlung `messages` gespeichert. Dieser Schreibvorgang löst nun die Funktion von Cloud Functions aus, welche eine Benachrichtigung zusammenbaut (siehe Code 66). Diese Funktion muss jedoch wissen an welches Gerät die Benachrichtigung versendet werden muss. Hierzu wird in dem Dokument des Nutzers (in der `users` Sammlung) eine Untersammlung `tokens` abgespeichert. Dessen Dokumente beinhalten die jeweilige Plattform, das zum Gerät des Nutzers passende Token und ein Erstellungsdatum. Pro Gerät, auf dem der Nutzer sich mindestens einmal angemeldet hat, wird also ein Token abgespeichert.

```

1 // Wird bei document.create in unreadMessages aufgerufen
2 export const sendToDevice = functions.firebaseio
3   .document("unreadMessages/{unreadMessage}")
4   .onCreate(async (snapshot) => {
5     // Nehme aktuelle Nachricht
```

```

6   const message = snapshot.data();
7
8   // Untersammlung tokens des Empfängers
9   const querySnapshot = await db
10    .collection("users").doc(message.sendTo)
11    .collection("tokens").get();
12   // Dokument des Senders
13   const sender = await db
14    .collection("users").doc(message.sendFrom)
15    .get();
16
17   // Erstelle Map von allen Tokens
18   const tokens = querySnapshot.docs.map((snap) => snap.id);
19
20   // Payload mit Name, Nachricht und clickAction
21   const payload: admin.messaging.MessagingPayload = {
22     notification: {
23       title: sender.get("name"),
24       body: message.message,
25       clickAction: "FLUTTER_NOTIFICATION_CLICK",
26     },
27   };
28   // Senden der Benachrichtigung
29   return fcm.sendToDevice(tokens, payload);
30 });

```

Listing 66: Cloud Functions zur Erstellung von Benachrichtigungen

Das Doppelte Abspeichern der Nachrichten ist zwingend notwendig, da die Erstellung einer Nachricht nicht für jeden Chatraum anders überprüft werden kann. Gleichzeitig wird aber auch die Sammlung `chatroom` benötigt, da diese zur Anzeige aller Räume eines Nutzers und aller Nachrichten eines Raumes benötigt werden. Für das Anzeigen wird in das Nutzerdokument eine Untersammlung mit Chaträumen abgespeichert. Dies muss aus Anzeigegründen eine Sammlung und kann kein einfaches Array oder Liste sein - pro Dokument werden Nutzernamen, NutzerIDs und Raum ID gespeichert.

Falls nun ein Nutzer mit einem anderen Nutzer gematched wird (siehe Abbildung 36b), kann dieser einen Chatraum erstellen. Dabei wird der Chatraum in die Untersammlung `chatRooms` wie beschrieben eingefügt. Dadurch erscheint er in der Liste seiner Chaträume. Gleichzeitig wird dieser Raum auch in die Untersammlung `pendingChatRooms` des Kommunikationspartners eingetragen. Nun erscheint die Chatanfrage beim Kommunikationspartner auf dem Hauptbildschirm (siehe Abbildung 36a oben). Er kann jetzt bereits mit ihm eine Konversation führen, jedoch ist es beiden nicht erlaubt die Profile des jeweils anderen zu sehen. Dies dient als Schutzmechanismus gegenüber oberflächlicher Betrachtung des Partners. Es steht dem angefragten Nutzer nun die Wahl ob er den Gesprächspartner annehmen (Abbildung 38c) oder ablehnen (Abbildung 38d) will. Beim Annehmen wird der Chat in die Untersammlung `chatRooms` verschoben und beide können die Profile einander sehen, Beim Ablehnen hingegen wird der Chat gelöscht und eine neutrale Informationsnachricht für den Kommunikationspartner in den Chat geschrieben.

### 5.3.4 Sicherheit

#### 5.3.4.1 Sicherheitsregeln

Um unbefugte Zugriffe nicht zuzulassen, müssen die Sicherheitsregeln korrekt gewählt werden. Diese können mittels der „Emulator Suite“ und Unit Tests (hier das Test Framework „mocha“) auf ihre Korrektheit getestet werden<sup>68</sup>.

Da jeder letztendlich die Profilbilder sehen darf, ist hier nur als Bedingung gegeben, dass der Sender der Anfrage angemeldet sein und die Datei kleiner als 5 MB sein muss (siehe Codebeispiel 23). Dies wurde gewählt, da ab einer Menge von 5 GB verbrauchter Speicherplatz Kosten in Höhe von \$ 0.026 pro GB anfallen.<sup>69</sup>

Bei den Firestore Regeln ist dies jedoch etwas komplizierter - diese sind im Anhang als Codebeispiel 71 zu finden. Die Zeilenabgabe ist bei den folgenden Erklärungen am Ende des Satzes zu finden. Grundsätzlich ist für Nutzer essentiell wichtig, dass Zugriffe nur gewährt werden, falls man authentifiziert ist. Ein Nutzer soll logischerweise Schreibrechte auf seinen eigenen Eintrag in der Datenbank haben (5). Ein Fremder hingegen darf dieses Lesen, falls er einen Eintrag in der Datenbank besitzt, damit keine Fehler auf der Oberfläche entstehen, falls veraltete Accounts (ohne Eintrag in der Datenbank) auf etwas zugreifen wollen (6). Auf die eigene Untersammlung `tokens` darf nur der Nutzer selbst zugreifen, damit zum Beispiel Benachrichtigungen nicht auch fälschlicherweise auf anderen Geräten angezeigt werden kann (7).

Die eigenen Chaträume darf ebenfalls nur der Nutzer selbst sehen und verändern (11). Die eingehenden Chatanfragen in der Untersammlung `pendingChatrooms` darf jeder Lesen, der einen existenten Eintrag in der Datenbank besitzt, da die Funktion `isPartOfChat()` für unseren Anwendungsfall Fehler liefern würde (14). Der Ursprung liegt bei der Überprüfung der Chat-Anwendung, ob ein Nutzer das Profil des anderen sehen darf oder nicht. Es wird also überprüft, ob der Nutzer in der eingehenden Chatanfrage des anderen Nutzers ist. Es tritt ein Fehler auf, welcher in der Testumgebung bisher nicht reproduziert werden konnte. Gleichzeitig schreiben darf nur jemand, der Teil des Chats ist oder eben der Nutzer selbst (15).

Auf die Sammlung `unreadMessage` hat jeder angemeldete Nutzer mit Datenbankeintrag Zugriff, damit jeder Benachrichtigungen an Personen versenden kann (18).

Ein Chatraum darf jeder valide Nutzer erstellen (22) und die generellen Informationen über ihn auch lesen (24). Das Lesen hatte gleiche Hintergründe, wie bei (14). Schreibzugriff durch beispielsweise Namensänderungen besitzt jeder, der Teil des Chats ist (23). Um einzelne Nachrichten im Chat lesen und schreiben zu dürfen, muss der Nutzer auf jeden Fall Teil des Chats und valide sein.

Da auf Nutzerdaten, welche im Auth-Service abgespeichert werden sowieso nur der eigentliche Nutzer zugreifen darf, gibt es hier auch keine Regeln.

#### 5.3.4.2 Ende-zu-Ende Verschlüsselung

Ende-zu-Ende Verschlüsselung bedeutet die Verschlüsselung von Daten, die bei einer Kommunikation nur von den Teilnehmern als Klartext gelesen werden kann. In Firebase sieht die Struktur des implementierten Chats wie in Abbildung 32a aus. Die Nachricht ist zwar bei der Verbindung von den Geräten zum Server über das Protokoll HTTPS und über eine lokale Verschlüsselung zum Abspeichern auf dem Server gesichert, jedoch sind diese Daten als Klartext sichtbar, während sie in die Frontend und Backend Server verarbeitet werden. Zudem haben Fi-

<sup>68</sup><https://firebase.google.com/docs/rules/unit-tests>, letzter Zugriff: 05. Mai 2021

<sup>69</sup><https://firebase.google.com/pricing>, letzter Zugriff: 05. Mai 2021

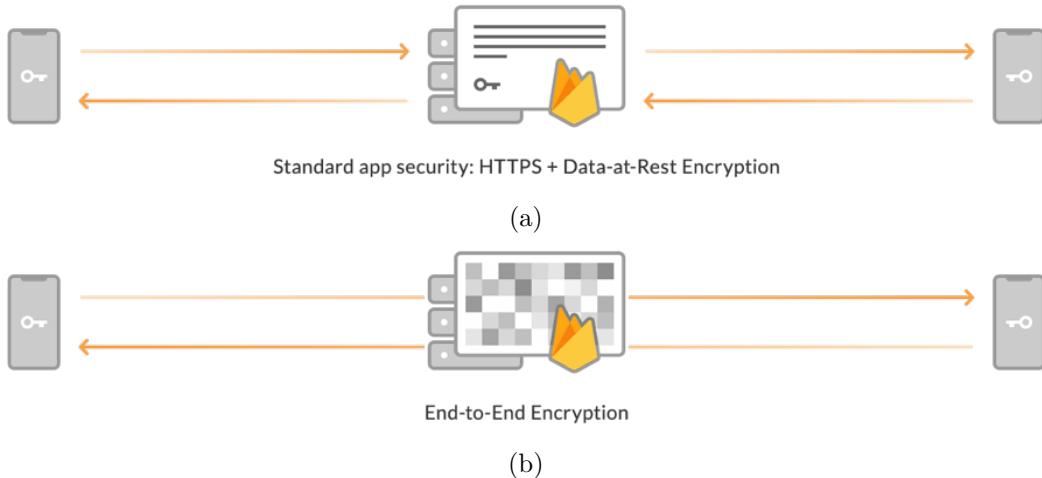


Abbildung 32: Firebase (a) ohne / (b) mit Ende-zu-Ende Verschlüsselung<sup>71</sup>

rebase Administratoren und Entwickler ebenfalls vollen Zugriff auf unverschlüsselte Nachrichten und könnten alles mitlesen. Daher wäre Ende-zu-Ende Verschlüsselung eine zusätzliche Schutzschicht, sowohl für Text, als auch Dateien. Die Struktur hierzu ist in Abbildung 32b beschrieben.

Um dies selbst zu implementieren, müssten beide Kommunikationspartner die öffentlichen Schlüssel des jeweils anderen kennen um die eigenen Nachrichten asymmetrisch verschlüsselten zu können. Einfacher geht das über eine externe Bibliothek. Zum Beispiel bietet Virgil Security hierzu das E3Kit als Backend-unabhängige Lösung. Da diese jedoch noch nicht zum Zeitpunkt der Dokumentation für die Programmiersprache Dart verfügbar ist, ist dieses Feature auch noch nicht implementiert.<sup>70</sup>

<sup>70</sup>Quelle: <https://virgilsecurity.com/blog/e3kit-for-firebase>, letzter Zugriff: 10. Mai 2021

<sup>71</sup>Quelle: <https://virgilsecurity.com/blog/e3kit-for-firebase>, letzter Zugriff: 10. Mai 2021

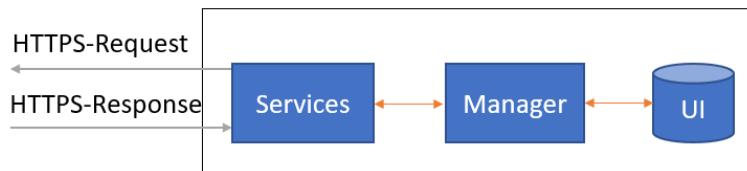


Abbildung 33: Vereinfachte Architektur der mobilen Anwendung

## 6 Implementierung der mobilen Anwendung

Die Software für die mobile Anwendung wird ebenfalls in Komponenten/Module aufgeteilt. Abbildung 33 stellt die vereinfachte Architektur dar. Die Service-Komponenten dienen als Kommunikationsschnittstelle zum Senden für HTTPS-Anfragen. Die Manager-Komponenten nutzen diese Schnittstellen und führen im Hintergrund des Programms verschiedene Tätigkeiten aus. Das User Interface (UI) stellt die Benutzeroberflächen dar und kann auf die Funktionen und Daten der Manager zurückgreifen.

### 6.1 Klassenmodelle

Nachfolgend werden die Klassenmodelle innerhalb der mobilen Anwendung dargestellt. Die Filmdaten, die der Nutzer für das Swipen auf der Oberfläche erhält, werden aus dem Webserver bezogen. Der Informationsgehalt eines Films beschränkt sich dabei auf den eindeutigen Identifikator (ID) und den Titel des Films. Im weiteren Schritt werden anhand der Film-ID's die weiteren benötigten Informationen aus dem TMDB-Server angefragt.

**Movie Modell:** Zum Konvertieren des JSON-Objekts aus dem Webserver und zum Speichern der Daten in die Umgebung der mobilen Anwendung wird die Klasse 'Movie'<sup>72</sup>.

```

1 Movie movieFromJson(String str) => Movie.fromJson(json.decode(str));
2 String movieToJson(Movie data) => json.encode(data.toJson());
3
4 class Movie {
5   Movie({
6     this.original_title,
7     this.id
8   });
9
10  String original_title;
11  String id;
12
13  factory Movie.fromJson(Map<String, dynamic> json) => Movie(
14    original\_title: json["original_title"],
15    id: json["id"],
16  );
17
18  Map<String, dynamic> toJson() => {
19    "original\_title": original_title,
20    "id": id,
  
```

<sup>72</sup>Zum Konvertieren von JSON Objekten in Dart-Code wurde die Online-Konvertierung <https://app.quicktype.io/> genutzt.

```
21 } ;
22 }
```

Listing 67: Movie Modell - JSON Konvertierung

**TMDB-Movie Modell:** Die Klasse 'TMDBMovie' ist eine Zusammenstellung mehrerer Daten. Sie enthält jeweils ein Objekt der folgenden aufgelisteten Klassen:

- **TmdbMovieDetails:** Enthält filmbezogene Informationen wie Titel, Posterpfad, Rating..
- **TmdbMovieCredits:** Enthält Informationen zu den mitwirkenden Personen wie Schauspieler und Regisseure.
- **TmdbMovieProvider:** Enthält Informationen bezüglich anbietenden Streaming-Providern.
- **TmdbMovieTranslations:** Enthält Informationen zu Sprachen zum Film.

**Match Modell:** Nutzer werden ihre Matches vom Webserver abfragen. Hierbei erhalten erhalten sie zwei Listen 'supermatches' und 'normalmatches' sowie die Informationen 'newChanges', die Auskunft darüber gibt, ob es seit der letzten Anfrage zu Änderungen kam. Die Listen sind jeweils mit den entsprechenden Matchinformationen gefüllt. Der Informationsgehalt begrenzt sich auf die 'uid' der Nutzer und die 'id' des Films/der Filme, über das beide Nutzer gematcht sind. Für die Konvertierung und Speicherung dieser Informationen wurde die Klasse 'PartInformationMatches' erstellt.

Für die Darstellung eines Matches werden jedoch weitere Informationen wie der Titel oder der Posterpfad des Filmes und dem Namen des gematchten Nutzers. Die Klasse 'FullInformationMatches' enthält Eigenschaft für die Speicherung dieser zusätzlichen Daten.

## 6.2 Kommunikationschnittstellen

Die Kommunikationsanfragen an den Webserver und den Filmdatenbankserver TMDB werden auf einzelne Service-Komponenten aufgeteilt, die jeweils das von Dart zur Verfügung gestellte HTTP-Package<sup>73</sup> implementieren.

Da der Webserver in der Entwicklungsphase einen selbstsigniertes Sicherheitszertifikat hat, wird für die HTTPS-Kommunikation die 'badCertificateCallback'-Funktion des 'HttpClient'-Objekts im HTTP-Package umgeschrieben. Es sei zu erwähnen, dass dieser Workaround nur in der Entwicklung genutzt werden soll.

```
1 class DevHttpOverrides extends HttpOverrides {
2   @override
3   HttpClient createHttpClient(SecurityContext context) {
4     return super.createHttpClient(context)
5       ..badCertificateCallback =
6         (X509Certificate cert, String host, int port) => true;
7   }
8 }
```

Listing 68: Bad Certificate - Workaround

Die Implementierung soll im Weiteren anhand der 'MoviesRequest'-Funktion des Movie-Services dargestellt werden. Die Funktionen der User-, Match- und Swipe-Services sind im ähnlichen Stil gehandhabt.

<sup>73</sup>Offizielle Homepage : <https://pub.dev/packages/http>

**Movie Service:** Der Movie-Service bietet die Funktion 'MoviesRequest', welche einen Post-Anfrage über HTTPS an die Serveradresse (URL) und dem dazugehörigen relativen Pfad '/movies/request' sendet. Dafür wird zunächst ein Objekt 'data' erstellt, das mit den als Parameter übergebenen Daten gefüllt wird, die für die serverseitige Abhandlung der Anfrage benötigt werden. Über die 'post'-Funktion wird die Anfrage an die übergebene URL gesendet. Dem Header muss bei Übergabe von JSON-Daten der Eigenschaft 'Content-Type' der Wert 'application/json' hinzugefügt werden. Dem Body wird das aus 'data' in JSON codierte Objekt übergeben. Nach Ankunft der Antwort wird über den Statuscode geprüft, ob die Anfrage problemfrei ausgeführt werden konnte. Letztlich wird eine Liste von Movie-Objekten, die aus den im Body der Antwort befindlichen JSON-Daten konvertiert wurden, erstellt und zurückgegeben.

```

1 static Uri _url_Request = Uri.parse('https://'+ URL + ':3001/
2   movies/request');
3
4 static Future<List<Movie>> MoviesRequest(final List<String>
5   internMovieList,
6   final int movieAmount,final bool includeSwiped) async {
7   var _responseBody;
8   String _uidtoken = await UserManager().getUserToken();
9
10  //Body mit JSON fuellen
11  Map data = { 'alreadyRequestedMovieIDs': internMovieList,
12    'includeSwiped': includeSwiped,
13    'amount': movieAmount,
14    'uidtoken': _uidtoken };
15
16  var body = json.encode(data);
17  // Daten senden
18  try { var response = await http.post(_url_Request,
19    headers: {"Content-Type": "application/json"},  

20    body: body).timeout(Duration(seconds: 10),onTimeout: (){  

21      throw Exception(); });
22
23  _responseBody = response.body;
24  if(response.statusCode != 200)
25  { throw(_responseBody); }
26
27  List collection = json.decode(_responseBody);
28  return collection.map((json) => Movie.fromJson(json)).toList();
29 }
```

Listing 69: Movie Service - MoviesRequest

**User Service:** Dieser Service enthält die User-bezogenen Funktionen 'CreateUser' und 'ChangeUser', die entsprechende Anfragen an den Webserver schicken.

**Match Service:** Dieser Service enthält die Match-bezogenen Funktionen 'MatchesRequest', 'DeleteSupermatch' und 'DeleteNormalmatch'. Ausserdem wird für die Entwicklungsphase die 'Trigger'-Funktion angeboten. Die Funktionen des Match-Services senden die dem Namen entsprechenden Anfragen an den Webserver.

**Swipe Service:** Dieser Service enthält lediglich die Funktion 'CreateSwipe', die die entsprechende Anfrage an den Webserver schickt.

**TMDB Service:** Der TMDB-Server bietet eine Programmierschnittstelle, auch Application Programming Interface (API) genannt, zum Abfragen von Daten<sup>74</sup>. Die nachfolgend aufgelisteten Funktionen fragen entsprechend dem benötigten Informationsgehalt unterschiedliche URL'S an. Dabei wird jeweils der URL eine ID eines Filmes mitgegeben. Die in der Antwort zurückgesendeten Daten beziehen sich auf die übergebene Film-ID. Dem Namen der Funktion entsprechend wird ein Objekt der Klasse 'TMDBMovieDetails', 'TMDBMovieCredits', 'TMDBMovieProviders' oder 'TMDBMovieTranslations' zurückgegeben.

- Request Details
- Request Credits
- Request Providers
- Request Translations

### 6.3 Manager

Die Manager-Klassen regeln die Backend-Funktionalitäten, die vom Nutzer nicht direkt ersichtlich sind.

**Swipe Manager:** Der Swipe Manager verwaltet sowohl die Filme, die in der Benutzeroberfläche geswipedet werden können, als auch getätigten Swipes des Nutzers. Er agiert im Hintergrund des Programms. Dabei pflegt er eine Liste von Filmen, dessen Anzahl er periodisch überprüft. Wird eine bestimmte Mindestanzahl erreicht, frägt er den Webserver nach neuen Filmen, die der Nutzer weder bereits geswipedet, noch geladen hat. Anschließend werden für jeden erhaltenen Film sämtliche Funktion des TMDB-Services mit der jeweiligen Film-ID aufgerufen und die zurückgehaltenen Objekte als TMDBMovie-Objekte in einer weiteren Liste gespeichert. Die daraus generierte Liste kann nun an das User-Interface zum Swipen übergeben werden. Ein Swipen hat zur Folge, dass der davon betroffene Film aus der Liste entfernt wird und der Webserver die Informationen über diesen Swipe-Vorgang erhält. Für die entsprechenden Anfragen an den Webserver nutzt der Swipe Manager den Movie- und den Swipe-Service.

**User Manager:** Dieser Manager verwaltet userbezogene Daten der mobilen Anwendung. Sie ist ausserdem für die Verwaltung des Firebase-Tokens zuständig und frägt bei abgelaufenem oder invaliden Token einen neuen ab.

**Match Manager:** Der Match Manager delegiert die Match-bezogenen Anfragen an den Match-Service weiter. Ausserdem verwaltet er bereits abgefragte Matches. Innerhalb der 'RequestMatches'-Funktion ruft er neben der Delegierung an den Match-Service weitere Funktionen aus. Zum einen werden anhand der mitüberlieferten Film'IDs Anfragen über den TMDB-Service an den TMDB-Server gesendet, mit dem Zweck, den zugehörigen Filmtitel und Posterpfad abzufragen. Zum Anderen wird eine Anfrage an Firebase zum Erhalt der 'uid' des gematchten Nutzers geschickt. Zur Verhinderung von redundanten Abfragen der Zusatzinformationen zwischen aufeinanderfolgenden Match-Abfragen wird innerhalb der Funktion 'requestMovies' zunächst auf die erhaltene 'newChanges'-Eigenschaft geprüft. Nur wenn dessen Wert dem booleschen Wert

---

<sup>74</sup>Offizielle Seite der API: <https://www.themoviedb.org/documentation/api>

true gleicht, werden die Anfragen an TMDB und Firebase ausgeführt. Andernfalls wird die gleiche Liste, die aus vorherigen Match-Anfrageresultaten im Match-Manager zwischengespeichert wurden, zurückgegeben.

## 7 Benutzeroberflächen der mobilen Anwendung

Die Benutzeroberfläche einer Software muss im Grunde genommen nur einen Informationsfluss in zwei Richtungen erzeugen. Die eine Richtung liefert Informationen an den User und über die andere kann der User Informationen an das System weitergeben. Um auf dem heutigen Markt Fuß fassen zu können, sollte eine Oberfläche jedoch wesentlich mehr Aspekte erfüllen.

### 7.1 Aspekte von Benutzeroberflächen

Die Vielschichtigkeit einer Benutzeroberfläche kann ausschlaggebend für den Erfolg einer Applikation sein, abhängig davon welche Erfahrungen der User mit der Oberfläche macht und welche Eindrücke sie hinterlässt. Hieraus resultiert wie lange ein User auf der App bleibt und wie oft er zurück kommt. Neben der Nutzungszeit erhöht eine positive User Experience die Weiterempfehlungsrate.

Bei erfolgreicher Software besteht ein großer Teil der Entwicklung in der Planung der Oberfläche, da die User Experience nicht zu umgehen ist. Auf die eine oder andere Art erlebt der User immer eine Erfahrung. Neben den offensichtlichen Aus- und Eingabefunktionen werden beispielsweise folgende Kriterien betrachtet:

**Simpel:** Ausgegebene Information kann zum Beispiel durch Icons, Farben oder Symbole vereinfacht werden. Eine Oberfläche sollte weder überladen sein, noch sollten alle Ein- und Ausgaben auf verschiedenen Screens verteilt sein. Bei der Entwicklung wird eine gesunde Mischung aus maximaler Funktionalität und einfacher, übersichtlicher Darstellung angestrebt.

**Einheitlich:** Die Bedienung und das Lesen von Applikationen kann erheblich vereinfacht werden wenn einheitliche Bedien- oder Ausgabeelemente verwendet werden. Nicht nur innerhalb einer App ist es sinnvoll konsistente Elemente in der Oberfläche zu verwenden, auch Funktionen von anderen Apps können die Bedienung vereinfachen. Bekannte Funktionen bei Smartphone-Applikationen sind zum Beispiel die Vergrößerung mit zwei Fingern oder das „Daumen nach oben“-Symbol als positive Rückmeldung. Durch das Einbauen solcher Features wird eine App intuitiv und ohne Einführung bedienbar.

**Benutzergesteuert:** Alle ausgeführten Aktionen sollten vom Benutzer ausgehen. Ein gutes Interface unterstützt den User lediglich bei seiner Bedienung, schränkt ihn aber nicht ein. Mit der heutigen Technologie ist die Verführung groß viele Funktionen automatisch ablaufen zu lassen. Was eigentlich der Sinn einer Applikation ist, kann jedoch auch negative Folgen haben. Zu viel Automatisierung verursacht das Gefühl von Kontrollverlust und Unsicherheit, was sich negativ auf das Vertrauen und somit auf die Nutzungszeit von dem User auswirkt.

**Klarheit:** Eine mobile App muss ohne Anleitung bedienbar sein. Sobald Unklarheiten beim User entstehen und Funktionen oder Ausgaben nicht erkannt werden können, verliert die Anwendung auf dem freien Markt.

Der User sollte zu jeder Zeit wissen welche Optionen ihm zur Verfügung stehen und welche Folgen seine Aktionen haben. Besonders wichtig ist das Feedback infolge einer Aktion. Auch wenn diese Aspekte offensichtlich erscheinen, können sie bei der Entwicklung einer App leicht übersehen werden. Verwendet werden einfache und für den User bekannte Funktionen, wie die Beschriftung aller Buttons oder das haptische, akustische oder optische Feedback beim drücken eines dieser Buttons.

**Benutzerfreundlich/Barrierefreiheit:** Die Bedienung der App sollte für Menschen mit Einschränkungen im vollen Umfang möglich sein. In Abschnitt 7.2 wird auf dieses Thema tiefer eingegangen. Aber auch Benutzer ohne Einschränkungen erwarten eine einfache und übersichtliche Bedienung, die auch beispielsweise Eingabefehler mit mehreren Versuchen verzeiht.

**Ästhetik:** Das Design spielt bei dieser Betrachtung gleich mehrere wichtige Rollen. Es sollte eine angenehme Arbeitsumgebung für den User erstellen, Ein- und Ausgaben verdeutlichen und gleichzeitig mithilfe eines eigenen Stils ein einzigartiges Image für die App schaffen (sogenanntes Branding) um deren Individualität und Wiedererkennungswert zu steigern. Das Design erschafft ein Erlebnis während der Benutzung und weckt unterbewusst Gefühle im User.

Gerade weil viele dieser Aspekte unterbewusst wirken, ist eine ausgiebige Betrachtung unumgänglich.

Eine Schwierigkeit, die sich bei der Entwicklung ergibt sind die zwei unterschiedlichen Ziele. Einerseits sollten bestehende Design- und Bedienelemente übernommen werden um die Bedienung intuitiv und übersichtlich zu gestalten, andererseits aber auch neue Ideen und Innovationen eingebracht werden, um sich von anderen Apps abzuheben und bleibenden Wiedererkennungswert aufzubauen.

## 7.2 Barrierefreiheit

Barrierefreiheit im Allgemeinen bedeutet, dass ein Gegenstand, eine Einrichtung oder Informationsquelle für Menschen mit Behinderung ohne Unzulänglichkeiten nutzbar, zugänglich oder auffindbar ist ([17], §4). In der Softwareentwicklung versteht man darunter Applikationen für Menschen mit Einschränkungen zugänglich und bedienbar zu machen. Bezogen auf die Entwicklung von mobilen Apps gilt es dabei den akustischen, optischen oder motorischen Einschränkungen der Benutzer entgegenzuwirken.

### 7.2.1 Barrierefreiheit in mobilen Anwendungen

Mit der Verbreitung von Smartphones ist die Benutzung mobiler Apps stark angestiegen und mittlerweile in nahezu jedem Haushalt aufzufinden. Obwohl etwa 9,5% aller in Deutschland lebenden Menschen einen Schwerbehindertenausweis besitzen (Stand 24.06.2020)[16] was etwa 7,9 Millionen Menschen entspricht, ist die Implementierung von barrierefreier Bedienung nicht selbstverständlich. Gerade Programmierern/innen aus dem privaten Sektor sind diese Funktionen oft nicht bekannt, es besteht kein Interesse oder sie werden schlichtweg vergessen. Software, die für öffentliche Einrichtungen entwickelt wird, ist durch das Behindertengleichstellungsgesetz von 2002 dazu verpflichtet ihr Softwareangebot bis spätestens dem 23. Juni 2021 barrierefrei zu gestalten ([17], §12a Abs.1). Hierzu zählen sämtliche Webseiten sowie mobile Anwendungen.

### 7.2.2 Barrierefreiheit in Filmen und Serien

Auch die Zugänglichkeit von Filmen und Serien für Menschen mit eingeschränkter Wahrnehmung wurde in den letzten Jahren stark verbessert. Hierbei lässt sich zwischen optischer und akustischer Einschränkung differenzieren. Für hörgeschädigte Personen werden bereits seit mehreren Jahrzehnten Untertitel eingesetzt. Was früher für vereinzelte Filme durch eine Funktion des Teletextes erreicht wurde, wird heutzutage durch eine integrierte Funktion des Videoplayers

verwirklicht. Immer mehr Videos werden mit Untertiteln veröffentlicht. Manche Anbieter wie beispielsweise die Internetplattform YouTube bieten durch Spracherkennung automatisch generierte Untertitel an, was eine flächendeckende Untertitelung ermöglicht.

Auch für Menschen mit eingeschränktem Sehvermögen werden Filme und Serien mithilfe von Audiodeskriptionen vermehrt zugänglich gemacht. Hierbei wird die bereits vorhandene Tonspur mit Bildbeschreibungen und Kommentaren versehen. Was bis vor wenigen Jahren noch etwas Besonderes war und nur für ausgewählte Filme bestimmt war, ist heutzutage Standard. Größere Video-On-Demand-Plattformen wie Netflix oder Amazon Prime bieten diese Möglichkeit bei nahezu allen Eigenproduktionen an. Zusätzlich werden bestehende Filme neu mit Audiodeskriptionen versehen.

Hieraus lässt sich leicht erkennen, dass Filme und Serien heutzutage auch von Menschen mit Einschränkungen genutzt werden. Was auf den ersten Blick vielleicht nicht bedacht wird oder als unwichtig abgestempelt wird, kann einen nicht unerheblichen Vergrößerungsfaktor für den Kundenstamm bewirken. Für die Entwicklung einer mobilen App, bei der Filme und Serien bewertet werden, spielt also die Barrierefreiheit eine wichtige Rolle und darf auf keinen Fall vernachlässigt werden.

### 7.2.3 Barrierefreiheit bei StreamSwipe

Bei der Entwicklung von StreamSwipe werden mehrere mögliche Einschränkungen der User betrachtet und entsprechend reagiert. Ziel ist es, dass sowohl der Kunde sowie der Anbieter maximal davon profitieren. Hierfür soll die App für ein möglichst großes Publikum zugänglich gemacht werden, jedoch auch sogenanntes Over-Engineering vermieden werden, da zu viele Funktionen eine App unübersichtlich, teuer und langsamer werden lassen.

Allgemein wird Leserlichkeit durch große Schriftgrößen, hohe Farbkontraste, große Schaltflächen oder universelles Design erreicht. Alleine in Deutschland tragen 44,5 Millionen Menschen regelmäßig eine Brille oder Kontaktlinsen und benötigen somit Sehhilfen [18]. Unterstützung auf Seiten der App kann hierfür durch vergrößerbaren Text geschehen. Da aber davon ausgegangen werden kann, dass Personen, die sich auf Sehhilfen verlassen, bereits eine Brille oder Kontaktlinsen besitzen, wird die Textgröße vorerst nicht variabel gehalten. Außerdem gibt es bei Android- und Apple-Smartphones bereits eingebaute Vergrößerungsfeatures, die Bildausschnitte vergrößert darstellen können. Aus diesem Grund wird in diesem Projekt kein Fokus auf dieses Feature gelegt.

Farbblindheit kann jedoch in vielen Formen auftreten. Um der bekannten Farbfehlsicht entgegenzuwirken, werden Farben aus Problembereichen wie Rot und Grün nicht nebeneinander benutzt. Allgemein wird ein schlichtes Design gewählt und Farben nur zu Akzentuierung und als Stilmittel benutzt, statt als Informationsträger wie beispielsweise in den Abbildungen 34a erkennbar ist. Geringe Sehschärfe durch Achromatopsie kann wie weiter oben beschrieben umgangen werden.

Ist die Sehkraft noch weiter eingeschränkt oder gar nicht mehr vorhanden, werden Semantiken eingesetzt. Hierbei erhält jedes Element auf dem Bildschirm eine Beschreibung, die vorgelesen werden kann. Bei Zahlen und Texten werden diese vorgelesen, sofern keine weitere Information hinterlegt ist. Besonders hilfreich ist dies jedoch bei Abbildungen. Ausgeführt wird das Auslesen von einem Screenreader. Mobile Geräte haben diese Funktion bereits standardmäßig eingebaut (VoiceOver bei Apple und TalkBack bei Android) und wandeln die Semantiken mittels Sprachsynthese in akustische Signale um. Bei Desktopanwendungen wie z.B. JAWS für Windows können diese Informationen zusätzlich auch durch eine Braillezeile wiedergegeben werden.

Bei Flutter ist das Hinzufügen von Semantiken bereits eingebaut. Hierfür kann ein String dem

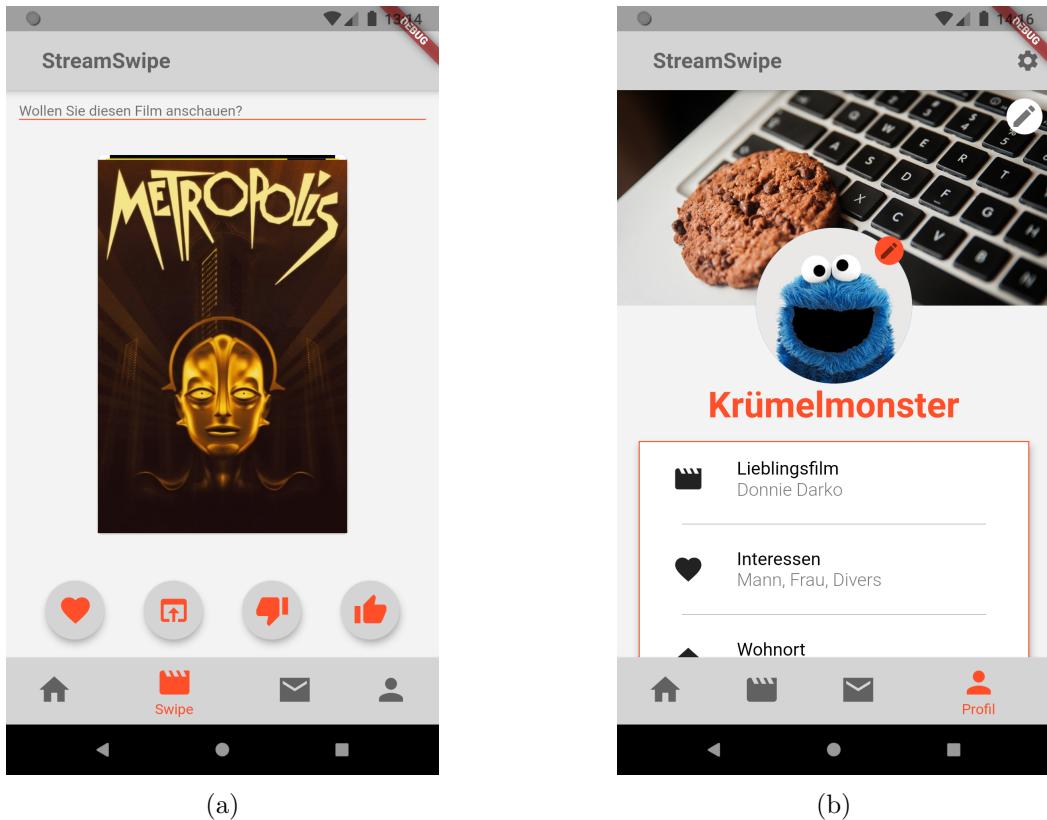


Abbildung 34: Screenshots aus der App StreamSwipe als Beispiele zu (a) schlichem Design, bei dem farbige Akzente nicht der Informationenübertragung dienen um die Zugänglichkeit für farbblinde Menschen zu verbessern und für einen Icon in (b), welcher sonst durch sehgeschädigte Menschen nicht wahrnehmbar ist, wird exemplarisch eine Semantik implementiert.

jeweiligen Bereich zugeordnet werden. In Beispiel 70 ist hierfür der Code des Buttons, der zu den Einstellungen führt. In Abbildung 34b ist dieser Button ganz rechts oben im Eck zu sehen. Die Funktion `GestureDetector()` erkennt Interaktionen mit dem Touchscreen, wobei hier nur auf Antippen reagieren soll, deshalb die Funktion `onTap: () {}`, die auf den Einstellungsbildschirm leitet. Diese Implementierung ist hier aber nicht von Relevanz und wird übersprungen. In dem `GestureDetector()` ist ein Icon eingebettet, von der Form *Settings*, was einem Zahnräder entspricht. Dieses Icon erhält eine Farbe und anschließend eine Semantik aus allem was in den Anführungszeichen steht. Ein Screenreader kann AE erkennen und ihn als den Umlaut Ä aussprechen.

So wird im kompletten Programm für jedes relevante Element vorgegangen. Teilweise müssen den Semantiken Variablen übergeben werden, da sich die vorzulesende Information ändert wie beispielsweise bei den Filmtiteln.

```

1 GestureDetector(
2   onTap: () {
3     ...
4   },
5   child: Icon(
6     Icons.settings,
7     color: Provider.of(context).colors.textSmall,
8     semanticLabel: "Einstellungen. Zum Auswählen doppeltippen.",
9   )

```

10 ) ,

Listing 70: Codeausschnitt in Dart von einem Button mit Semantiken.

Bei einer sauberen Implementierung wird auf diese Weise vorgegangen und eine bereits vorhandene Funktion verwendet. Dies vereinfacht nicht nur die Leserlichkeit des Codes, sondern bietet auch die höchste Modularität, da hierbei normalerweise standardisierte Schnittstellen für Betriebssysteme oder andere Anwendungen verwendet werden. In diesem Fall müssen die Screenreader von Android und Apple damit arbeiten können.

Um für Personen mit eingeschränktem Hörvermögen oder vollständiger Gehörlosigkeit die App zugänglich zu machen, wird auf akustisches Feedback als notwendige Informationsübertragung verzichtet. Innerhalb der App werden keine Geräusche erzeugt, außer der oben beschriebenen Funktion der Semantiken. Beim Erhalten einer neuen Nachricht oder eines neuen Matches kann weiterhin optional eine akustische Benachrichtigung erhalten werden. Hierbei wird die betriebssystemeigene Funktion übernommen, sodass in der App keine neuen Einstellungen vorgenommen werden müssen.

Auch feinmotorische Einschränkungen werden versucht zu umgehen. Die Navigation und die Filmbewertung in StreamSwipe können durch großflächige Wischbewegungen ausgeführt werden. Wo diese Lösung nicht möglich ist, werden verhältnismäßig große Buttons eingesetzt. Lediglich beim Registrieren und Einloggen werden feine Bewegungen erforderlich. Hierbei öffnet sich allerdings die als Standard eingestellte digitale Tastatur, die in vielen Fällen eine Spracheingabe besitzt, sodass die sehr kleinen Tasten nicht benutzt werden müssen.

Sollte sich in Zukunft jedoch Kritik in Form von negativen Nutzerbewertungen herauskristallisieren, kann eines der noch nicht implementierten Features über ein Update nachgerüstet werden.

## 7.3 Oberflächen von StreamSwipe

Die Smartphone-App lässt sich in mehrere Bereiche aufteilen, die sich in ihren Funktionen unterscheiden. Auf Basis der oben beschriebenen Grundlagen wurden diese Bereiche entworfen und werden in diesem Kapitel analysiert. Auch wenn manches davon als gewöhnlich oder naheliegend erscheint, so ist jedes Element mit Bedacht gewählt, erstellt und angepasst worden.

### 7.3.1 Login-Screen

Bei erstmaliger Benutzung der App öffnet sich der Login-Screen. An diesem Punkt wird der erste Eindruck für den Benutzer gesetzt, wobei bei StreamSwipe ein schlichtes Design gewählt wurde. Man sieht helle Grautöne mit einem Akzentfarbton, welche sich durch alle Bildschirme der App ziehen werden. Abhängig davon, ob der User in den Systemeinstellungen den dunklen Modus gewählt hat, werden anstatt den hellen Grautönen, dunkle bis schwarze Farben dargestellt, siehe auch Abbildungen 36c und 37e.

Auf dem Login-Screen (siehe Abbildung 35a) sind neben einer Überschrift mehrere beschriftete Textfelder und Buttons zu sehen, welche allesamt mit Semantiken versehen wurden, um durch einen Screenreader erkannt und identifiziert werden zu können. Die gewählte Anordnung wird universell bei Apps, Programmen und Webseiten benutzt, sodass die Felder auch ohne die eingetragenen Hinweistexte korrekt ausgefüllt werden könnten. Beim Antippen der Textfelder, öffnet sich die Standardtastatur des Betriebssystems. Sind alle Felder korrekt ausgefüllt, wird der User in die eigentliche App weitergeleitet, ansonsten wird durch individualisierte Fehlermeldung auf eventuelle Falscheingaben hingewiesen. Nach Erstellen eines neuen Accounts, durchläuft der User einen ähnlich aufgebauten Bildschirm (siehe Abbildung 35b) und wird danach aufgefordert

weitere Informationen zur Profilvervollständigung einzugeben (siehe Abbildung 35c und 35d). Auch hierbei werden bekannte Bedienelemente wie Textfelder, Dropdownmenüs und Checkboxen verwendet, wie in der Abbildung 35e beispielhaft dargestellt ist. Falls der User in den Systemeinstellungen des Smartphones den Nachtmodus aktiviert hat, wird das Appdesign angepasst, siehe Abbildung 35f.

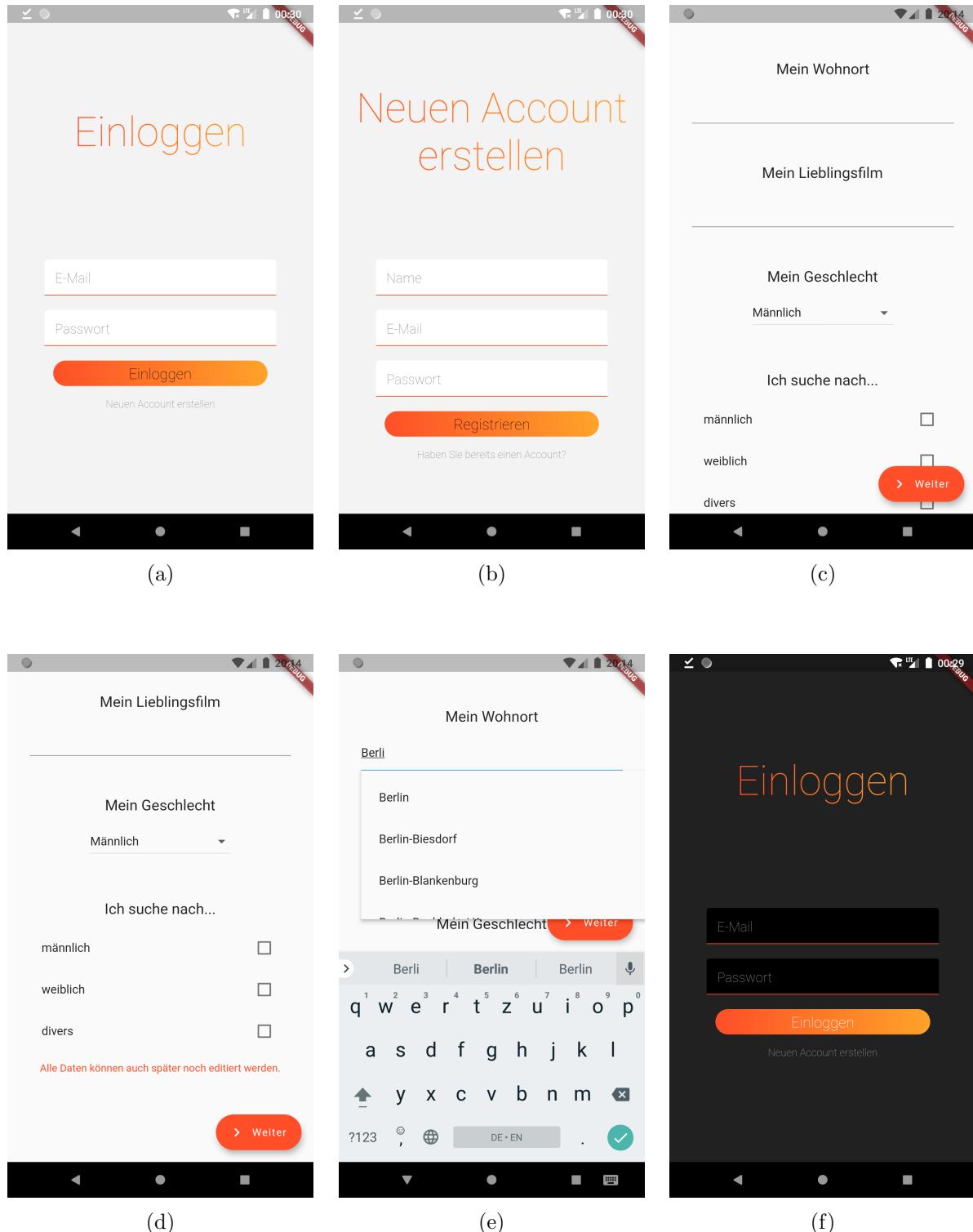


Abbildung 35: Die Anmeldeseiten von StreamSwipe und alle damit zusammenhängenden Screens. Man sieht (a) das Einloggen bei bestehendem Account, (b) das Erstellen eines Accounts, (c) und (d) das Formular für die benötigten Profildaten, (e) ein Texteingabefeld mit Auto vervollständigung als Dropdownmenü und (f) das Farbschema der Anmeldeseiten im Dark-mode am Beispiel des Login-Screens.

### 7.3.2 Home-Screen

Da davon ausgegangen wird, dass der User sich nicht nach jeder Nutzung ab- und wieder anmeldet, erscheint im alltäglichen Gebrauch der in Abbildung 36 dargestellte Bildschirm zuerst. Demnach bietet es sich an Ereignisse wie neue Matches und neue Nachrichten hier anzuzeigen. Diese werden wie Abbildungen 36a und 36b zeigen klar strukturiert in Abschnitte eingegliedert, welche mit Überschriften kenntlich gemacht sind. Die einzelnen Matches befinden sich mit allen dazugehörigen Funktionen und Informationen jeweils auf einer Karte. Durch diese Karten kann mit einer von anderen Apps bekannten horizontalen Swipemechanik navigiert werden. Weiterführende Funktionen wie das Starten eines Chats oder das Löschen des Matches werden durch Antippen von allgemein verständlichen Icons ausgeführt.

Auch auf diesem Screen findet sich einerseits das bereits eingeführte Farbschema wieder und es werden andererseits ebenfalls Semantiken verwendet. Bei den neuen Nachrichten wird jeweils der Benutzername vorgelesen und bei den neuen Matches je nach ausgewähltem Bereich der Filmname, die Icons oder der Text dazwischen.

Am unteren Bildschirmrand ist eine sogenannte Bottom-Navigation-Bar zu sehen. Sie ermöglicht eine kompakte und anschauliche Navigation durch die relevanten Bildschirme. Außerdem zeigt sie an welcher Bildschirm aktuell ausgewählt ist, wobei diese Information wie in Abschnitt 7.2.3 erarbeitet nicht ausschließlich auf einer Farbänderung basieren sollte und deshalb das ausgewählte Icon durch Hinzufügen von Text hervorgehoben wird. Liest der Screenreader die Semantik hiervon, gibt er die Bezeichnung des aktuellen Bildschirms sowie die Anzahl der weiteren Möglichkeiten an.

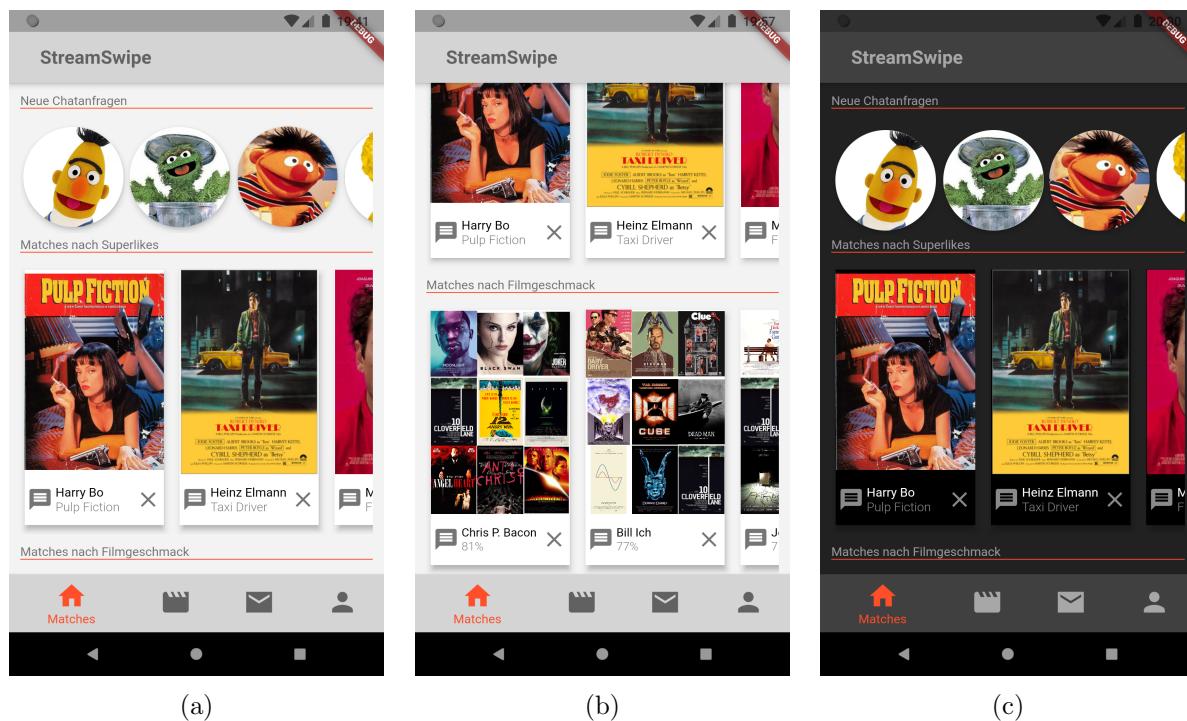


Abbildung 36: Der Home-Screen, der beim Öffnen der App zuerst gezeigt wird und Neuigkeiten wie neue Nachrichten und Matches zusammenfasst. Um den gesamten Inhalt dieser Seite sehen zu können, wird in (a) der obere Abschnitt und in (b) der untere Abschnitt gezeigt. Hat der User in den Systemeinstellungen den dunklen Modus aktiviert, so wird (c) der Home-Screen wie alle anderen Screens angepasst.

### 7.3.3 Swipe-Screen

Auf dem Swipe-Screen (Abbildungen 37) findet die Bewertung der Filme statt. Durch das hier verwendete Matchingsystem mithilfe des Filmgeschmacks unterscheidet sich StreamSwipe von anderen Apps und erhält so einen innovativen, individuellen Charakter, womit diese Seite das Herzstück der App bildet.

Das zuvor eingeführte Farbschema bleibt auch hier erhalten, wie Abbildung 37a zeigt. Eine Überschrift im selben Stil wie bereits aus Abschnitt 7.3.2 bekannt, verdeutlicht durch eine Frage nach welcher Motivation die Filmauswahl getroffen werden soll. Zentral im Bild ist eine Liste von Postern der zu beurteilenden Filme. Wie bereits durch die Datingapp Tinder verbreitet, werden die vier Antwortmöglichkeiten durch eine Swipe-Bewegung in eine der vier Richtungen ausgewählt. Abhängig von der Position des Fingers auf dem Touchscreen bewegt sich das Filmposter innerhalb des Bildschirms, was den Effekt einer frei beweglichen Karte hervorruft. Um klarzustellen welche Swipe-Richtung für welche Entscheidung steht, verfärbt sich der jeweilige Indikator in der unteren Reihe bei Verschiebung des Filmposter. Beide diese Animationen sind in Abbildung 37d zu sehen. Die Indikatoren sind mit Icons versehen, zeigen aber durch Drücken welche Entscheidung sie repräsentieren und in welche Richtung der User dafür swipen muss, wie Abbildung 37c am Beispiel des rechten Indikators zeigt.

Durch Antippen des Filmposters werden weitere Informationen zu dem jeweiligen Film dargestellt, wie in Abbildung 37b zu sehen. Gleichfalls wird durch ein einfaches Antippen wieder zurück zu den Postern gewechselt. Eine Rotations-Animation verdeutlicht die Illusion der Karten.

Alle diese für die Bedienung der App grundlegenden Steuerungen verlangen keine feinmotorischen Eingaben und können problemlos von Personen mit motorischen Einschränkungen genutzt werden. Auch dieser Bildschirm ist vollkommen mit Semantiken ausgestattet. Anstelle des Filmposters wird der Name des Films ausgelesen und für die vier Indikatoren am unteren Rand werden jeweils deren Funktion und durch welche Swipe-Richtung sie erreicht werden vorgelesen. Sämtliche Textfelder können ebenfalls problemlos von einem Screenreader gelesen werden.

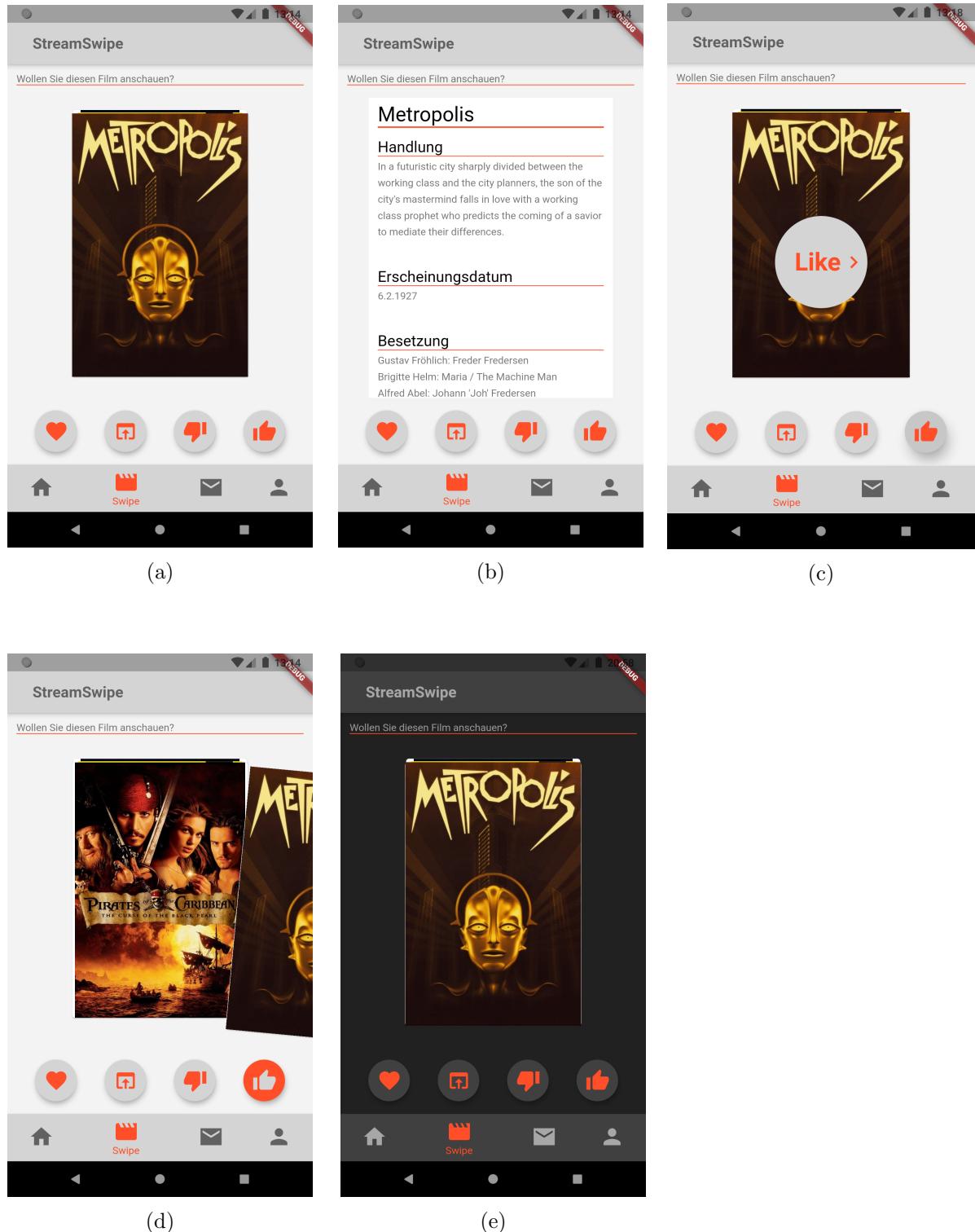


Abbildung 37: Darstellungen und Funktionen des Swipe-Screens mit (a) der Standarddarstellung, (b) weiteren Filminformationen, (c) einer Animation beim Drücken einer der Indikatoren und (d) der Swipe-Animation. Hat der User in den Systemeinstellungen den dunklen Modus aktiviert, so wird (e) der Swipe-Screen wie alle anderen Screens angepasst.

### 7.3.4 Chat

Die Chatseite ist in eine Liste aus aktiven Chats und eine Warteliste aufgeteilt, siehe 38a und 38b. Um zwischen diesen beiden Listen zu wechseln werden Tabs eingesetzt, wie sie aus Windowsanwendungen bekannt sind. Zwischen diesen Tabs kann entweder gewechselt werden, indem ein anderes Tabfenster angetippt wird, oder der gesamte Bildschirm mit einer Geste zur Seite gewischt wird. Bei jedem Element der Chatliste ist der jeweilige Benutzername und die neueste Nachricht zu sehen, dazu wird falls vorhanden entweder ein Profilbild oder eine einfarbige Fläche mit dem Anfangsbuchstaben des Namens angezeigt. Chat Requests können jeweils durch das Schieben nach links angenommen oder nach rechts ablehnt werden, wie in den Abbildungen 38c, bzw. 38d zu sehen ist. Diese Mechanik wird häufig in Email-Apps zum Löschen oder Verschieben der Mails benutzt.

Durch Antippen eines Matches, öffnet sich der Chatverlauf, welcher in Abbildung 38e zu sehen ist. Die Anordnung der Nachrichten innerhalb des Chatverlaufs ist wie aus anderen Messenger bereits bekannt, aber in den Stilfarben von StreamSwipe. Am oberen Bildschirmrand wird der Profilname des Matches angezeigt und rechts davon befindet sich der Button zu dessen Profilseite, auf der genauere Details über diese Person zu finden sind. Die Profilseiten werden in Kapitel 7.3.5 genauer vorgestellt.

Dem User wird durch das ihm bereits vorgestellte Design und der ausschließlichen Nutzung von bekannter Mechanik ein vertrautes Umfeld geboten. Wie auf jedem Screen passt sich auch hier das Farbschema automatisch an, falls in den Systemeinstellungen des Smartphones das dunkle Design gewählt wurde, wie beispielsweise in Abbildung 38f dargestellt. Neben der Benutzerfreundlichkeit wird auch die Barrierefreiheit beachtet, indem alle Elemente, die nicht bereits aus einem Text bestehen, mit Semantiken ausgestattet werden. Zusätzlich werden keine feinmotorischen Bewegungen zur Navigation durch die Bildschirme benötigt. Bis auf die Eingabe über die Tastatur kann alles über große Flächen oder Wischmechaniken bedient werden. Im Chatverlauf wird die Standardtastatur des Systems verwendet, mit der der Benutzer bereits vertraut ist.

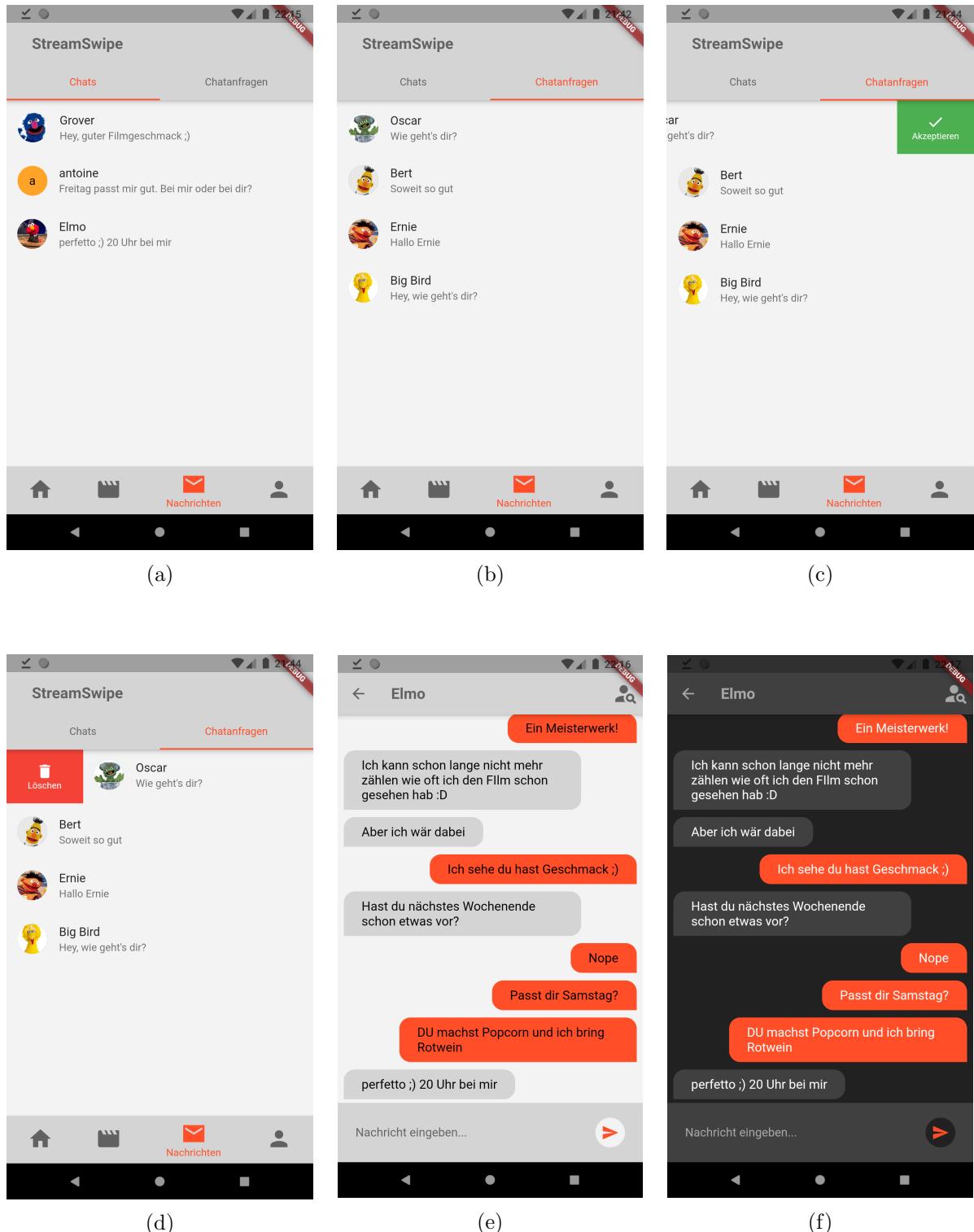


Abbildung 38: Darstellungen und Funktionen der Chat-Screens mit (a) den aktiven Chats, (b) den Chats auf der Warteliste, (c) und (d) angenommene, bzw. abgelehnte Chats auf der Warteliste, sowie (e) einem Chatverlauf im hellen und (f) in dunklen Modus.

### 7.3.5 Benutzerprofil

Auf der Profilseite werden ein Profilbild, ein Hintergrundbild und für das Matching relevante persönliche Informationen dargestellt. Es gibt eine Version, die nur von anderen Nutzern sichtbar

ist, mit denen ein Match stattgefunden hat, und eine Version, die über die Bottom-Navigation-Bar erreichbar werden kann. Die Letztere wird in Abbildung 39 dargestellt und unterscheidet sich von der Version für andere Nutzer darin, dass Profil- und Hintergrundbild bearbeitet werden können.

Das Farbschema und das Design wurden an die bisherigen Seiten angepasst. Um die Oberfläche simpel und selbsterklärend zu halten, wird jede dargestellte Information mit einem passenden Icon und einem Hinweis versehen (siehe Abbildung 39a). Die Icons zum Bearbeiten der Bilder sind wie auch in vielen anderen Apps platziert und designt. Sie öffnen die systemeigene Bildergalerie des Smartphones um den User aus einem bekannten Umfeld Bilder auswählen lassen zu können.

Beim initialen Öffnen einer Profilseite sollen Namen, Profilbild und ein Hintergrundbild ins Auge springen. Sie stellen die ersten Informationen dar, die dem Betrachter wichtig sind, weshalb sie wie in Abbildung 39a deutlich sichtbar ist beim Öffnen mehr als die Hälfte des Bildschirms einnehmen. Anschließend wird der Fokus auf detailliertere Informationen gerichtet. Auf der Profilseite von StreamSwipe wird hierfür heruntergescrollt um den Block mit den Profildaten sehen zu können. Bei dieser Aktion blendet eine Animation das Profilbild aus und verschmälert das Hintergrundbild. Der Benutzername wird ebenfalls aus dem Fokus gezogen, bleibt aber wie in Abbildung 39b zu sehen mit dem verbleibenden Hintergrundbildausschnitt erhalten. Dies hilft dem Betrachter unterbewusst bei dem Fokuswechsel und schafft ein modernes, responsives Feedback bei der User Experience.

Um das durchgängig schlichte Design der App zu erhalten ist der Zugang zu den Einstellungen ausschließlich auf der Profilseite zu finden. Hierfür ist im rechten oberen Bildschirmbereich das repräsentative Icon. Der hierdurch erreichbare Bildschirm (Abbildung 39c) ist gleich aufgebaut wie die Informationeneingabe nachdem ein neuer Account erstellt wurde (Abbildungen 35c und 35d). Die dort angegebenen Informationen können hier wieder angepasst werden.

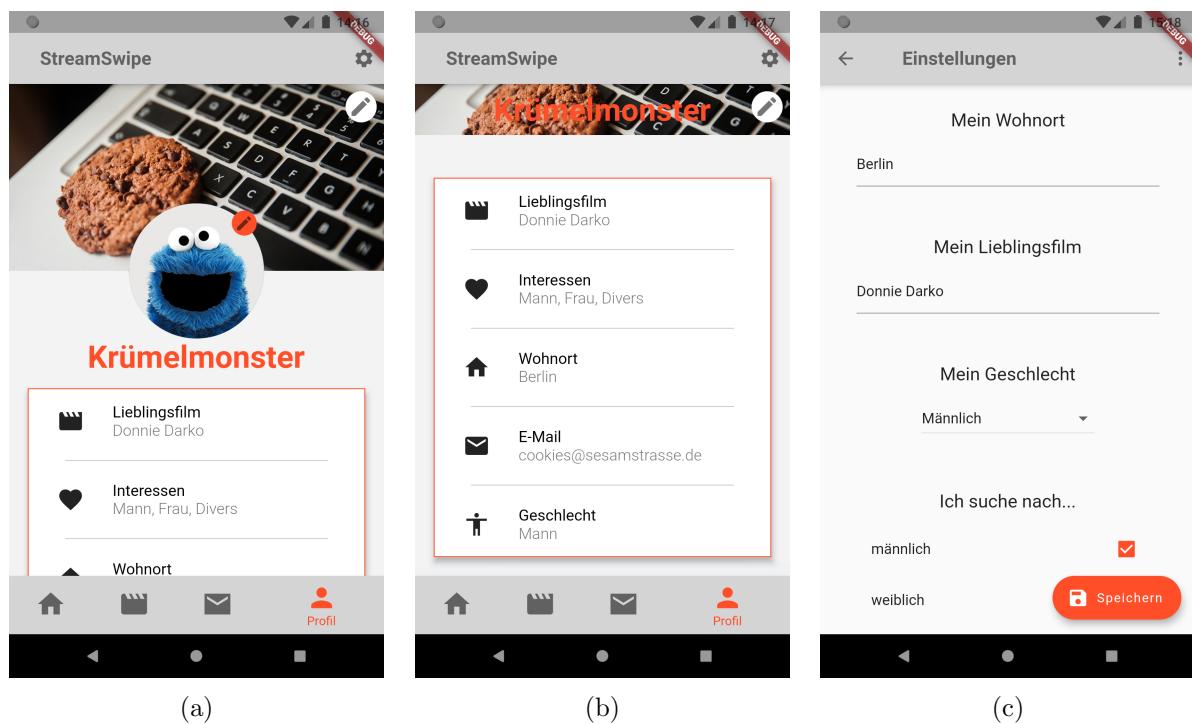


Abbildung 39: Profilseite wie sie für den Nutzer selbst angezeigt wird (a) im normalen Zustand und (b) nach vollständigem Einklappen des Profilkopfes durch eine Animation während dem Herunterscrollen. Mit den von hier aus erreichbaren Einstellungen (c) können die anfänglich gegebenen Profilangaben abgepasst werden.

## 8 Probleme

In der Planungsphase eines Projektes werden oft große Meilensteine gesetzt. Sobald das Thema bekannt ist, werden durch Brainstorming und andere Methoden der Ideenfindung in kurzer Zeit viele Ziele gesteckt und ein in der Theorie fertiges Projekt ausgearbeitet. Eine Begrenzung wird nur durch die Fantasie der Beteiligten gezogen, die jedoch später in der Umsetzung schnell erreicht wird. Aber auch bei realistischen Zielen wird oft eine sehr spezielle Vorstellung angestrebt, die mit den gegebenen oder gewählten Mittel nur bedingt umsetzbar ist. Neben den Arbeitsmitteln zählen aber auch Zeit und Geld zu den einschränkenden Ressourcen.

Wie unzählige Projekte vor uns, konnten wir ebenfalls nicht alle zu Beginn gesteckten Ziele zu unserer vollsten Zufriedenheit abschließen. Auch wenn alle grundlegend wichtigen Kriterien erfüllt sind und funktionieren, gibt es auch nicht erreichte Ziele. Diese bestehen aus erkennbaren und nicht erkennbaren Schwächen in der App. Ersteres sind Unsauberkeiten, die der User bei der Benutzung in manchen Situationen bemerken kann. Hierzu zählen beispielsweise Designfehler, die bei der Programmierung nicht erkannt wurden. Nicht erkennbare Schwächen sind Ideen und Ziele, die im vorgegebenen Rahmen nicht mehr eingebaut werden konnten. Diese fallen nur dem Programmierer auf. In unserem Fall ist das Fehlen mancher Funktionen hauptsächlich auf Zeitmangel zurückzuführen. Bei einem zeitlich begrenzten Bearbeitungsrahmen, während dem noch einige andere Projekte, Vorlesungen, Prüfungen und eine Arbeitsstelle in Vollzeit belegt werden, können nicht alle beabsichtigten Features eingebaut werden. Alle geplanten Erweiterungen wurden jedoch bereits durchdacht und werden nach Abgabe der schriftlichen Arbeit implementiert, aber hierzu mehr in Kapitel 11.

Da StreamSwipe in Deutschland entwickelt und programmiert wurde, liegt es nahe Deutsch als In-App-Sprache zu verwenden. Außer den auf den Oberflächen angezeigten Texten müssen auch Fehlermeldungen, Semantiken und Auto vervollständigungen angepasst werden. Wie auf den Screenshots im ganzen Kapitel 7.3 zu sehen ist, wird dies in der App konsequent durchgesetzt. Die Filmdatenbank, aus der die dargestellten Informationen erhalten werden, liefert diese jedoch nur auf Englisch, was an der Handlungsausgabe in Abbildung 37b zu sehen ist. Alle Datenbanken, die in Kapitel 4.5 betrachtet werden, bieten ihre Informationen nur in einer Sprache an. Eine Lösung dieses Problems wird in Kapitel 10 beschrieben.

Bei einer sauberen Implementierung werden die Elemente auf den Bildschirmen in Abhängigkeit der Gesamtgröße des Screens angeordnet, sodass sie sich auf unterschiedlichen Geräten entsprechend anordnen und ihre Größe anpassen können. Man spricht hierbei von responsivem Design, auf welches bei StreamSwipe ebenfalls Wert gelegt wurde. Je nach Verhältnis von Breite und Höhe der Bildschirmgrößen unterschiedlicher Geräte können so in Grenzfällen jedoch ungewollte Verzerrungen auftreten. Das Aussehen einer Benutzeroberfläche kann außerdem auch von dem verwendeten Betriebssystem abhängen, da unterschiedliche Generationen unterschiedliche Standards verwenden. Durch die begrenzte Entwicklungszeit und den limitierten Zugang zu Ressourcen konnte die App nur auf einer kleinen Anzahl von Endgeräten getestet und angepasst werden. Es kann somit leider keine Garantie für eine saubere Darstellung auf älteren Geräten gegeben werden. Längere Testphasen könnten dieses Problem minimieren, jedoch ist der Markt von Smartphones mittlerweile so unübersichtlich groß, dass es nahezu unmöglich ist jede Variation zu testen.

Das wahrscheinlich größte Manko ist das Ausbleiben der Veröffentlichung der App am Abgabetermin der schriftlichen Arbeit. Ursprünglich geplant war eine Veröffentlichung in Google Play für Androidgeräte und im App Store für iPhones, weshalb wie in Kapitel 2.7.2 beschrieben Flut-

ter als Framework verwendet wurde. Bei einer Veröffentlichung in Google Play wird die App vor Release einer umfangreichen Überprüfung unterzogen, was voraussichtlich eine Woche und in manchen Fällen auch länger dauern kann [19]. Außerdem muss das vom Webserver benutzte Sicherheitszertifikat zur Kommunikation über HTTPS von einer offiziellen Zertifizierungsstelle signiert werden. Wie bereits in Kapitel 5.2.3 angesprochen, benutzen wir ein selbstsigniertes Zertifikat. Zu den Kosten dieser Zertifizierung wird noch eine Mitgliedschaft als Google Play Developer von einmalig \$ 25 benötigt [21]. Für eine Veröffentlichung im App Store ist eine Mitgliedschaft im Apple Developer Programm notwendig, die jährlich \$ 99 kostet [20]. Diese Investitionen werden erst sinnvoll, wenn ein Monetarisierungsplan der App ausgearbeitet wurde..

Wie in Kapitel 3.2 bereits erwähnt, werden die Verteilung der Film-IDs und die Auswertung der Filmbewertungen auf einem Server ausgeführt. Dieser Server besteht zur Zeit aus einem Raspberry Pi. Für den Gebrauch während der Entwicklungsphase ist die somit erreichte Rechenkapazität völlig ausreichend, da nur maximal drei Personen gleichzeitig darauf zugegriffen haben. Sollte die App jedoch veröffentlicht werden, werden die Nutzerzahlen unvorhersehbar ansteigen, sodass nicht vorausgesagt werden kann, ab wann der Server in diesem Aufbau überlastet sein wird. Um den Benutzern eine positive User Experience bieten zu können und eventuelle Hardwareschäden an dem Raspberry Pi zu vermeiden, muss mit der Veröffentlichung der App gewartet werden bis eine leistungsstärkere Lösung gefunden wurde.

## 9 Anwendbarkeit

Die theoretische Planung einer App weicht oft von der späteren praktischen Anwendung ab. Dies kann unterschiedliche Gründe haben, die hier zum Teil beleuchtet werden sollen.

Bei der Entwicklung einer App werden Funktionen für gutmütige Benutzer implementiert. Auch wenn dieses Vorgehen unbewusst auftritt, ändert es nichts daran, dass es sich hierbei um eine naive Herangehensweise handelt. Die eingebauten Funktionen werden in der Regel so ausgelegt, dass sie bei sachgemäßer Benutzung problemlos funktionieren. Wie sie jedoch in der Praxis angewendet werden, wird hierbei oft nicht bedacht. Die unsachgemäße Benutzung muss nicht mutwillig durch die User geschehen, kann jedoch durch einfache Bedienfehler Probleme verursachen. Beim Programmieren der Features sollte deshalb auf verschiedene Problemherde eingegangen werden.

**Falscheingaben:** Hierzu zählen freiwillige und unfreiwillige Falscheingaben. Unfreiwillige Angaben sind beispielsweise Tippfehler oder falsche Filmpräferenzierung. Ersteres wird teilweise bei der Eingabe überprüft, wie etwa die E-Mail-Adresse, die eine bestimmte Form aufweisen muss, und andere Angaben können später in den Einstellungen korrigiert werden. Falsche Filmbewertungen können in unserem Fall jedoch nicht rückgängig gemacht werden, sollten aber auf zwischenmenschlichem Wege nach einem Matching vom User gelöst werden können.

Unter freiwilligen Falscheingaben versteht man falsche Datenangaben, was bei einem falschen Namen zur Anonymisierung führt. Das Matchingverfahren wird über den angegebenen Wohnort gemacht, sodass durch einen Wohnortwechsel der potentielle Personenkreis geändert wird. Da StreamSwipe von einer hohen Userdichte pro Wohnort profitiert, wirkt sich dies negativ auf den vom User verlassenen Wohnort aus. Es ist möglich den Userstandort per GPS auszulesen, jedoch können so User in einem schwach besiedelten Gebiet nicht in die nächstgrößere Stadt wechseln und die Filterung bei der Matchberechnung wäre dann wesentlich feinmaschiger.

**Schließen der App:** Falls die App beispielsweise während einer Datenangaben geschlossen wird, können unerwartete Fehler auftreten. Die Auswirkungen variieren, abhängig davon, an welchem Zeitpunkt die App geschlossen wird. Im schlimmsten Fall sind die Benutzerdaten anschließend beschädigt und verursachen einen Absturz der App beim nächsten Start.

**Funktionenmissbrauch:** Ist der genaue Vorgang hinter einer Funktion bekannt, kann diese schnell missbraucht werden. Bei dem Bewertungsverfahren von StreamSwipe wird über eine große Anzahl von Filmen ein Präferenzprofil erstellt und eine Übereinstimmung mit anderen Profilen über einem gewissen Prozentsatz gesucht. Wird jedoch ein Film mit 'Superlike' bewertet, so matcht das System alle User, die diesen Film ebenfalls mit 'Superlike' bewertet haben. Außerdem fließt die Bewertung 'Superlike' ebenfalls in die Erstellung des Präferenzprofils. Wenn also ein User ausschließlich Filme superliket, unabhängig ob ihm der Film gefällt, kann er eine sehr hohe Anzahl an Matches erreichen. Jedoch wird durch diese Art des Matchings nicht das ursprünglich geplante Ziel aus Kapitel 1 erreicht und User, die die App ernsthaft für einen gelungenen Filmeabend benutzen, könnten sich betrogen fühlen und die App wieder deinstallieren.

Es ist bereits geplant die Anzahl der gespeicherten Superlikes pro User zu begrenzen, sodass ab einer gewissen Anzahl alte Superlikes gelöscht werden wenn Neue hinzukommen. Nur die aktuell gespeicherten Superlikes werden auch zum Matching verwendet. Die Implementierung dieses Updates wird jedoch erst nach Abgabe dieser Dokumentation geschehen, da es in der Regel selbst nach der Veröffentlichung einer App eine Weile dauert

bis solche Schlupflöcher gefunden werden.

Die Praxis zeigt jedoch, dass auch bei guten anfänglichen Überlegungen nicht alle Probleme beseitigt werden können, da diese teilweise zu vielschichtig sind oder sich mit anderen gewollten Features überschneiden. Trotz umfangreicher Maßnahmenergreifung kann nie vorausgesagt werden wie sich etwas entwickelt. Über wurde beispielsweise ursprünglich als Limousinenservice gegründet und ist heute eines der größten Taxiunternehmen weltweit. Deshalb sollte bei der Planung bedacht werden was die User eigentlich wollen und welche Funktionen in der Praxis wirklich genutzt werden. Bei einer Dating-App geht es darum möglichst einfach mit anderen Personen Kontakt aufzunehmen. Das Swipen von Filmpostern sollte in den meisten Fällen kein Hindernis darstellen. Ein großer Prozentsatz der User einer Dating-App verfolgt bei der Benutzung ein sehr spezielles Ziel, auf das bei StreamSwipe keinen Fokus gelegt wurde. Jedoch existiert hierbei bereits zu Beginn des Matches zwischen den beteiligten Personen ein gemeinsames Thema und ein Filmabend ist für diese Absicht wahrscheinlich zielführender als manch anderer Vorwand. Sicher lockt StreamSwipe auch Personen an, die gerne neue Filme und Personen mit ähnlichem Geschmack kennenlernen würden. Hierfür bietet sich StreamSwipe optimal an, sobald Filmempfehlungen implementiert werden, mehr dazu in Kapitel 10.

Ebenso relevant wie die bestehenden Bedürfnisse der Nutzer, sind die bisherigen Lösungen, die der Markt bietet. Hierzu müssen alle Apps betrachtet werden, die die selben Funktionen wie StreamSwipe anbieten. Schränkt man die Funktionen auf das wesentliche ein, sodass der Fokus auf dem Bewerten von Filmen ist wodurch neue Leute kennengelernt werden, ist der Markt noch absolut frei. Keine andere App bietet diese Funktionen an, jedoch existieren ähnliche Konzepte, die aber alle für Einzelpersonen oder bestehende Gruppen Filmvorschläge auf Basis der bewerteten Filme bieten. Das Matching mit neuen Leuten ist bisher noch nicht vertreten, was die perfekte Basis für einen Marktstart bietet.

## 10 Ausblick

Ausblick:

- Bilder zwischenspeichern
- Matches nach Filmgeschmack
- Angabe bevorzugter Genres
- Serien
- Algorithmus zur Aktualisierung der Datenbank
- Filme ausfiltern(doppelungen etc..)
- Filme filtern: Filme nach verfügbarer Sprache/genre filtern
- Superlikes begrenzen
- Trailer abspielen in der App
- ...

Die Entwicklung und Programmierung der App war ein wichtiger und großer Schritt, jedoch ist er nicht der letzte. Für die nun kommende Zeit stehen bereits einige Aufgaben fest um das Projekt StreamSwipe aufrechtzuerhalten.

Wie bereits in Kapitel 8 erwähnt, konnten nicht alle gewünschten Features eingebaut werden. Sie waren während der Entwicklungsphase entweder von zu geringer Priorität, oder sind erst im späteren Stadium aufgetreten. Beispielsweise werden die Filmdaten bisher nur auf Englisch erhalten. In einer sonst deutschsprachigen App ist diese Umstellung jedoch schon in Planung. Es existieren deutsche Filmdatenbanken wie beispielsweise OFDb, aus denen die übersetzten Daten erhalten werden können. Diese ist leider nicht so umfangreich wie die bisher genutzte Datenbank, sodass dann von beiden parallel Daten erhalten und abgeglichen werden um die User Experience nicht zu mindern. In diesem Zuge ist auch eine optionale In-App-Sprache geplant, die der Benutzer auswählen kann, da nun die Informationen auf Deutsch und auf Englisch vorliegen. Um die Sprache der App ändern zu können, müssen alle Textelemente durch Variablen ersetzt werden, die in einer zentralen Tabelle beschrieben werden. Für eine andere Sprache werden diese Variablen dann mit den übersetzten Formulierungen überschrieben. Ist die App angepasst und der Datenfluss der beiden Datenbanken synchronisiert, können auch mit weniger Aufwand auch andere Sprachen in das System aufgenommen werden.

Ein Update, bei dem Filmempfehlungen eingeführt werden, ist bereits geplant. Hierbei werden auf Basis der berechneten Filmpräferenz und den Überschneidungen innerhalb eines Matches eine Liste mit Filmvorschlägen generiert. Dieses Feature ist von großer Bedeutung wenn man bedenkt wie viel Zeit in die Suche eines geeigneten Filmes gesteckt wird.

Auch das Ressourcenproblem des Servers kann in absehbarer Zeit behoben werden. Da die Skripte auf dem Raspberry Pi bereits geschrieben sind, ist lediglich eine leistungsstärkere Hardware nötig um einen stabilen Server zu realisieren. Solange die Benutzerzahlen und somit auch die benötigte Leistung überwacht werden, kann frühzeitig ohne Datenverlust oder Performanceeinbußen auf ein größeres System umgestellt werden.

Ist die App dann für die Öffentlichkeit zugänglich, lohnt es sich ein Feedbacksystem zu nutzen, durch welches die User Lob und Kritik äußern können. Oft treten kleine Bugs nur in sehr speziellen Situationen auf, die während der Testphase nicht bedacht oder nicht realisiert werden. Der Wartungsaufwand nach Release einer Software wird oft im Verhältnis zum Entwicklungsaufwand als gering erachtet, sollte aber nicht unterschätzt werden. Werbung spielt

einen entscheidenden Faktor in der Kundengewinnung. Mithilfe gezielter Produktplatzierungen, möglichst großflächiger Marketingstrategien und zufriedenen Bestandskunden kann innerhalb von wenigen Wochen ein beachtlicher Kundenstamm aufgebaut werden. Timing ist in diesem Fall sehr wichtig, da die angeworbenen Kunden möglichst gleichzeitig mit der App bekannt gemacht werden sollten. StreamSwipe basiert auf einem möglichst flächendeckenden Kundenkreis, da nur lokal gematcht wird und ein User ohne Matches nicht lange gehalten werden kann. Entsprechend ergibt es Sinn gezielte Werbung in einer lokalen Umgebung einzusetzen und lieber eine hohe Benutzerdichte, als eine große Reichweite aufzubauen. Wird die App in diesem Umfeld genutzt und somit Matches generiert, verbreitet sie sich automatisch auch in umliegende Städte und beginnt so zu wachsen. Mit dem Benutzerradius sollte auch der Radius der geschalteten Werbung sukzessiv vergrößert werden.

## 11 Fazit

Die Vision einer Dating-App, die Matches auf Basis des Film- und Seriengeschmacks erstellt, konnte in dem vorgegebenen Zeitrahmen erfolgreich erstellt werden. Alle grundlegenden Funktionen wurden implementiert und das Backend bildet eine saubere Einheit mit dem Frontend wodurch ein angenehmes Benutzererlebnis erzeugt wird.

Die daraus erwartete Revolutionierung des Onlinedatings kann jedoch noch nicht demonstriert werden, da die App noch nicht veröffentlicht wurde, beziehungsweise außer von den beteiligten Entwicklern noch kein Benutzerfeedback zur Auswertung vorliegt. Auch wenn die App alle notwendigen Funktionen bereits erfüllt, sollten wie in Kapitel 10 erarbeitet vor der Veröffentlichung noch ein paar Features hinzugefügt werden. Neben den notwendigen Aktualisierungen wurden auch einige neue Ideen angesprochen, mit welchen die App weiter wachsen kann. Solche Implementierungen können jedoch nicht als negativer Aspekt gewertet werden, da diese lediglich zum Fortschritt und zur Optimierung einer App beitragen. Dieser Wandel gehört zum natürlichen Entwicklungsprozess und ist auch bei populären und professionell erstellten Apps zu beobachten. Kritiken, neue Ideen und Verbesserungsvorschläge von Benutzern und Entwicklern lassen auf eine regelmäßige und aktive Benutzung einer App schließen.

Unter diesen Gesichtspunkten zeigt StreamSwipe ein gelungenes Konzept mit einem enormen Marktpotential.

## A Verfasser einzelner Abschnitte

Kapitel / Abschnitt	Verfasser
1. Einleitung	Vincent Schreck
1.1 Motivation	Vincent Schreck
1.2 Methode	Vincent Schreck
2. Theoretische Grundlagen	Leon Gieringer & Robin Meckler
2.1 Netzwerkprotokolle	Robin Meckler
2.2 JavaScript	Robin Meckler
2.3 Node.JS	Robin Meckler
2.4 NoSQL-Datenbank	Robin Meckler
2.5 Firebase	Leon Gieringer
2.6 Anwendungsentwicklung für mobile Endgeräte	Leon Gieringer
2.7 Frameworks zur mobile, plattformübergreifenden Entwicklung	Leon Gieringer
2.8 Recommender Systems	Leon Gieringer
3. Planung	Vincent Schreck
3.1 Konzept	Vincent Schreck
3.1 Komponenten	Vincent Schreck
4. Auswahl geeigneter Technologie	Leon Gieringer & Robin Meckler
4.1 Anwendungsframework	Leon Gieringer
4.2 Server	Robin Meckler
4.3 Datenbank	Robin Meckler
4.4 Kommunikationsschnittstelle	Robin Meckler
4.5 Film-Datenbank	Robin Meckler
5. Serverseitige Implementierung	Leon Gieringer & Robin Meckler
5.1 Firebase	Leon Gieringer
5.2 Datenbank	Robin Meckler
5.3 StreamSwipe-Webserver	Robin Meckler
6. Implementierung der mobilen Anwendung	Robin Meckler
6.1 Klassenmodelle	Robin Meckler
6.2 Kommunikationsschnittstellen	Robin Meckler
6.3 Manager	Robin Meckler
7. Benutzeroberfläche	Vincent Schreck
7.1 Aspekte von Benutzeroberflächen	Vincent Schreck
7.2 Barrierefreiheit	Vincent Schreck
7.3 Oberflächen von StreamSwipe	Vincent Schreck
8. Probleme	Vincent Schreck
9. Anwendbarkeit	Vincent Schreck
10. Ausblick	Vincent Schreck
11. Fazit	Vincent Schreck

## B Sicherheitsregeln

```

1  rules_version = '2';
2  service cloud.firestore {
3      match /databases/{database}/documents {
4          match /users/{user} {
5              allow write: if isAuthenticated() && request.resource.id
6                  == request.auth.uid;
7              allow read: if isAuthenticated() && userExists();
8              match /tokens/{token} {
9                  allow read, write: if isAuthenticated() && user ==
10                     request.auth.uid;
11             }
12             match /chatRooms/{chatroom} {
13                 allow read, write: if isAuthenticated() && user ==
14                     request.auth.uid;
15             }
16             match /pendingChatRooms/{chatroom} {
17                 allow read: if isAuthenticated() && userExists();
18                 allow write: if isPartOfChat(request.resource.id,
19                     request.auth.uid) || user == request.auth.uid;
20             }
21             match /unreadMessages/{unreadMessage} {
22                 allow read, write: if isAuthenticated() && userExists();
23             }
24             match /chatroom/{chatRoomId} {
25                 allow create: if isAuthenticated() && userExists();
26                 allow write: if isPartOfChat(chatRoomId, request.auth.uid
27                     );
28                 allow read: if isAuthenticated() && userExists();
29             }
30             /****** Hilfsfunktionen *****/
31             // Ueberprueft, ob der Anfragende angemeldet ist
32             function isAuthenticated() {
33                 return request.auth != null;
34             }
35             // Ueberprueft, ob der Nuter einen Eintrag in Firestore
36             // besitzt
37             function userExists() {
38                 return exists(userRef(request.auth.uid));
39             }
40             // Ueberprueft, ob das Array "users" in einem Chatraum die
41             // UID des Anfragenden enthaelt
42             function isPartOfChat(chatRoomId, userId) {

```

```
41     return (get(chatroomRef(chatRoomId)).data.users[0] ==  
42             userId)  
43     || (get(chatroomRef(chatRoomId)).data.users[1] == userId)  
44  
45     /***** Hilfsreferenzen *****/  
46     function chatroomRef(id){  
47         return /databases/$(database)/documents/chatroom/${id};  
48     }  
49     function userRef(id) {  
50         return /databases/$(database)/documents/users/${id};  
51     }  
52 }  
53 }
```

Listing 71: Sicherheitsregeln Firestore

## Abkürzungsverzeichnis

Abkürzung	Bedeutung
2D	Zweidimensionalität
ATL	Android Transport Layer
API	Application Programming Interface
APN	Apple Push Notification Service
App	Applikation
BaaS	Backend-as-a-Service
CAP	Consistency, Availability, Partitioning
CSS	Cascading Style Sheets
CRUD	Create, Read, Update, Delete
DOD	(United States) Department of Defense
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
GCP	Google Cloud Platform
GPS	Global Positioning System
FCM	Firebase Cloud Messaging
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input / Output
ID	(eindeutiger) Identifier
IDE	Integrated Development Environment
IME	Input Method Editor
IMDB	Internet Movie Database
IP	Internet Protocol
ISO	International Organization for Standardization
JS	JavaScript
JSC	JavaScriptCore
JSON	JavaScript Object Notation
JSX	JavaScript Expression
MB	MegaBytes
NoSQL	Not only SQL
ODM	Object Document Mapper
OEM	Original Equipment Manufacturer
OMDB	Open Media Database
OpLog	Operation log

Abkürzung	Bedeutung
ORM	Object Relational Mapping
OSI	Open Systems Interconnection
PWA	Progressive Web App
SDK	Software Development Kit
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TMDB	The Movie Database
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UI	User Interface
uid	Userbezogene ID
UML	Unified Modeling Language
VM	Virtuelle Maschine
XAML	Extensible Application Markup Language

## Literaturverzeichnis

- [Prot1] Wikipedia. Netzwerkprotokolle. <https://de.wikipedia.org/wiki/Netzwerkprotokolle>, letzter Zugriff: 06.04.2021
- [Prot2] Wikipedia. OSI-Modell. <https://de.wikipedia.org/wiki/OSI-Modell>, letzter Zugriff: 06.04.2021
- [Prot3] Wikipedia. DoD-Schichtenmodell. <https://de.wikipedia.org/wiki/DoD-Schichtenmodell>. letzter Zugriff: 06.04.2021
- [Prot4] Elektronik Kompendium. OSI-Schichtenmodell. <https://www.elektronik-kompendium.de/sites/kom/0301201.htm>, letzter Zugriff: 06. April 2021
- [JS1] Saternos, Casimir (2014). Client-Server Web Apps with JavaScript and Java. O'Reilly Media. Seite 32f. ISBN 978-1449369330
- [JS2] Philip Ackerman (2018). JavaScript Das umfassende Handbuch. Rheinwerk Computing. Pp 45. ISBN 978-3-8362-5696-4
- [JS3] Ingo Pakalski, 15 Jahre WWW: Die Browserkriege - Golem.de. <http://www.golem.de/0805/59377.html>, letzter Zugriff: 03. April 2021
- [JS4] Ben Ilegbodu, History of ECMAScript. <https://www.benmvp.com/blog/learning-es6-history-of-ecmascript/>, letzter Zugriff: 03. April 2021
- [JS5] ECMA International, ECMAScript®2020 Language Specification <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>, letzter Zugriff: 03. April 2021
- [JS6] Paul Krill, ECMAScript 2021 spec for JavaScript nears the finish line. <https://www.infoworld.com/article/3613948/ecmascript-2021-spec-for-javascript-nears-the-finish-line.html>, letzter Zugriff: 03. April 2021
- [JS7] David Flanagan, JavaScript The Definitive Guide, Seite 1, ISBN 858-1-1222-6666-3
- [JS8] David Flanagan, JavaScript The Definitive Guide, Seite 2, ISBN 858-1-1222-6666-3
- [JS9] Wikipedia, JavaScript, [https://en.wikibooks.org/wiki/JavaScript/Relation\\_to\\_other\\_languages](https://en.wikibooks.org/wiki/JavaScript/Relation_to_other_languages), letzter Zugriff: 03. April 2021
- [JS10] Programiz, JavaScript object properties, <https://www.programiz.com/javascript/object>, letzter Zugriff: 03. April 2021
- [JS11] Dipl.-Ing. (FH) Stefan Luber, Stephan Augsten. <https://www.dev-insider.de/was-ist-javascript-a-586580/>, letzter Zugriff: 03. April 2021
- [JS12] Philip Ackerman, JavaScript Das umfassende Handbuch (2018), Rheinwerk Computing, Seite 46ff, ISBN 978-3-8362-5696-4
- [Node1] Basarat Ali Syed , Beginning Node.js 2014 Seite 23ff. ISBN 978-1-4842-0187-9
- [Node2] StrongLoop, What Makes Node.js Faster Than Java? <http://strongloop.com/strongblog/node-js-is-faster-than-java/>, letzter Zugriff: 03. April 2021
- [Node3] Philip Ackerman (2018). JavaScript Das umfassende Handbuch. Rheinwerk Computing. Seite 868ff. ISBN 978-3-8362-5696-4

- [Node4] Node.js documentation. [https://nodejs.org/api/modules.html#modules\\_caching](https://nodejs.org/api/modules.html#modules_caching), letzter Zugriff: Stand 05. April 2021
- [Node5] Wikipedia. NPM. [https://de.wikipedia.org/wiki/Npm\\_\(Software\)](https://de.wikipedia.org/wiki/Npm_(Software)), letzter Zugriff: 05. April 2021
- [Node6] Ahmad Nassri, So long, and thanks for all the packages! <https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages.html>, letzter Zugriff: 03. April 2021
- [Node7] Alexander Neuman, Node.js 0.6.3 integriert npm <https://www.heise.de/developer/meldung/Node-js-0-6-3-integriert-npm-1386141.html>, letzter Zugriff: 03. April 2021
- [Node8] <https://expressjs.com/de/>, letzter Zugriff: Stand 04. April 2021
- [Node9] [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs/Introduction](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction), letzter Zugriff: 04. April 2021
- [Node10] Philip Ackerman (2018). JavaScript Das umfassende Handbuch. Rheinwerk Computing. Seite 895ff. ISBN 978-3-8362-5696-4
- [Node11] Lee Brandt, Build and Understand Express Middleware through Examples <https://developer.okta.com/blog/2018/09/13/build-and-understand-express-middleware-through-examples>, letzter Zugriff: 04. April 2021
- [Node12] Express, API-Dokumentation. <https://expressjs.com/de/api.html>, letzter Zugriff: Stand 05. April 2021
- [Node13] Weiterleitung (Routing). <https://expressjs.com/de/guide/routing.html>, letzter Zugriff: Stand 05. April 2021
- [Node14] Binariks, Express.js Mobile App Development: Pros and Cons for Developers. <https://binariks.com/blog/tools/express-js-mobile-app-development-pros-cons-developers/>, letzter Zugriff: 05. April 2021
- [Node15] Basarat Ali Syed, Beginning Node.js, Seite 275, ISBN 978-1-4842-0187-9
- [Node16] U. Störl, T. Hauf, M. Klettke and S. Scherzinger, SSchemaless nosql data stores—Object-NoSQL mappers to the rescue?”, Proc. Datenbanksyst. Bus. Technologie Web (BTW), Seiten 579-599
- [Node17] Simon Holmes. Mongoose for Application Development. Seite 124f. ISBN 978-1782168195
- [Node18] Tim Ambler, Nicholas Cloud. JavaScript Frameworks for Modern Web Dev. Seite 321. ISBN 978-1-4842-0663-8
- [DB1] Andreas Meier, Grundlagen, Systeme und Nutzungspotenziale, Seite 149, ISBN: 9783658115890
- [DB2] Andreas Meier, Big Data, Grundlagen, Systeme und Nutzungspotenziale, Seite 34, ISBN: 9-783-6581-1589-0
- [DB3] S. G. Edward und N. Sabharwal, Practical MongoDB. Berkeley, 2015, Seite 26, ISBN 9-781-4842-0648-5
- [DB4] <https://db-engines.com/de/ranking>, letzter Zugriff: 10. Februar 2021
- [DB5] <https://www.mongodb.com/json-and-bson>, letzter Zugriff: 10. Februar 2021

- [DB6] S. G. Edward und N. Sabharwal, Practical MongoDB. Berkeley, 2015, Seite 32, ISBN 9-781-4842-0648-5
- [DB7] S. G. Edward und N. Sabharwal, Practical MongoDB. Berkeley, 2015, Seite 83, ISBN 9-781-4842-0648-5
- [DB8] S. G. Edward und N. Sabharwal, Practical MongoDB. Berkeley, 2015, Seite 160, ISBN 9-781-4842-0648-5
- [DB9] [https://de.wikipedia.org/wiki/Replikation\\_\(Datenverarbeitung\)](https://de.wikipedia.org/wiki/Replikation_(Datenverarbeitung)), letzter Zugriff: 10. Februar 2021
- [DB10] <https://docs.mongodb.com/manual/replication/>, letzter Zugriff: 10. Februar 2021
- [DB11] <https://docs.mongodb.com/manual/core/replica-set-oplog/>, letzter Zugriff: 10. Februar 2021
- [DB12] <https://docs.mongodb.com/manual/sharding/>, letzter Zugriff: 11. Februar 2021
- [Tech1] <https://entwickler.de/online/javascript/7-gruende-node-js-579924149.html>, letzter Zugriff: 18. März 2021
- [Tech2] J. Batra und S. Batra, „MONGODB Versus SQL: A Case Study on Electricity, Seite 297-298
- [Tech3] G. Roden, Node.js und Co: Skalierbare hochperformante und echtzeitfähige Webanwendungen professionell in JavaScript entwickeln, ISBN 9-783-8986-4829-5
- [Tech5]
- [1] Aggarwal, C. C. (2016). Recommender systems (Vol. 1). Cham: Springer International Publishing.
  - [2] Fentaw, A. E. (2020). Cross platform mobile application development: a comparison study of React Native Vs Flutter.
  - [3] Facebook Inc. (2021) React Native documentation. [Online] Verfügbar: <https://reactnative.dev/docs/getting-started>, letzter Zugriff: 13. April 2021
  - [4] Facebook Inc. (2021) React documentation. [Online] Verfügbar: <https://reactjs.org/docs/getting-started.html>, letzter Zugriff: 13. April 2021
  - [5] Flutter (2021) Flutter architectural overview [Online] Verfügbar: <https://flutter.dev/docs/resources/architectural-overview>, letzter Zugriff: 04. März 2021
  - [6] Johnson R. E. & Foote B. “Designing Reusable Classes.” Journal of ObjectOriented Programming 1, 2 (June/July 1988). Page 22-35.
  - [7] Majchrzak TA, Ernsting J, Kuchen H (2015) Achieving business practicability of model-driven crossplatform apps. OJIS 2(2):3–14
  - [8] Cisco (2020) Cisco Annual Internet Report (2018–2023) White Paper [Online] Verfügbar: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>
  - [9] Charland, A. & Leroux, B. (2011). Mobile application development: Web vs. native. Communications of the ACM, 54(5):49–53.

- [10] Lachgar, M., & Abdelmounaim, A. (2017). Decision Framework for Mobile Development Methods. International Journal of Advanced Computer Science and Applications, 8.
- [11] Biørn-Hansen, A., Rieger, C., Grønli, TM. et al. (2020) An empirical investigation of performance overhead in cross-platform mobile development frameworks. Empir Software Eng 25, 2997–3040. <https://doi.org/10.1007/s10664-020-09827-6>
- [12] Stahl T, Volter M (2006) Model-driven software development. Wiley, Chichester
- [13] Firebase Inc. (2021) Firebase documentation. [Online] Verfügbar: <https://firebase.google.com/docs>, letzter Zugriff: 25. April 2021
- [14] Meinungsforschungsinstituts Civey (2020): <https://www.presseportal.de/pm/145489/4627304>, letzter Zugriff: 13. Mai 2021
- [15] Splendid Research (2017): <https://www.springerprofessional.de/konsumforschung/marketingstrategie/konsumenten-auf-der-serien-welle/15146374>, letzter Zugriff: 14. Mai 2021
- [16] Statistisches Bundesamt (2020): <https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Behinderte-Menschen/Tabellen/schwerbehinderte-alter-geschlecht-quote.html;jsessionid=885260788D4FFC7F670576B72E5089F4.live741>, letzter Zugriff: 17. April 2021
- [17] Behindertengleichstellungsgesetz (2002): <https://www.gesetze-im-internet.de/bgg/BGG.pdf>, letzter Zugriff: 19. April 2021
- [18] Institut für Demoskopie Allensbach (2019): *Untersuchung zum Sehbewusstsein der Deutschen*, <https://www.sehen.de/presse/pressemitteilungen/zahlen-fakten/allensbach-brillenstudie-201920/>, letzter Zugriff: 11. Mai 2021
- [19] Google Support (2021): *App veröffentlichen*, <https://support.google.com/googleplay/android-developer/answer/9859751?hl=de>, letzter Zugriff: 14. Mai 2021
- [20] Apple Support (2021): *Infos zum Apple Developer Program*, <https://developer.apple.com/de/support/compare-memberships/>, letzter Zugriff: 14. Mai 2021
- [21] Digital Guide IONOS (2020): <https://www.ionos.de/digitalguide/websites/web-entwicklung/die-eigene-app-entwickeln-android-app-veröffentlichen/>, letzter Zugriff: 14. Mai 2021