StreamSwipe
Tinder für Filme

Leon Gieringer, Robin Meckler,Vincent Schreck

Studienarbeit

11. Mai 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

1.1 ..

1.2 Aufbau der Arbeit

2 Motivation

3 Theoretische Grundlagen

3.1 Netzwerkprotokolle

3.1.1 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. [1.0] 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. [1.05] 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.

Zusätzlich zum OSI-Modell existiert das in den 1960er-Jahren entwickelte TCP/IP-Referenzmodell. Entwickler dieses Schichtmodells war das das Verteidigungsministerium der Vereinigten Staaten, auch bekannt als das Departments of Defense (DoD). Dementsprechend trägt das TCP/IP-Referenzmodell auch den Namen DoD-Schichtenmodell. [1.06]

OSI-Schicht	TCP/IP-Schicht	Beispiel
Anwendungen (7)	Anwendungen	HTTP, UDS, FTP, SMTP, POP, Telnet, DHCP, OPC UA
Darstellung (6)		
Sitzung (5)		TLS, SOCKS
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: OSILAYER [1.1]

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.

Tabelle 1: Kurzbeschreibung der OSI-Schichten [1.2]

IP
In der Vermittlungsschicht des OSI-Schichtenmodells findet, unabhängig des Übertragungsmediums und der genutzten Topologie, die logische Adressierung der Endgeräte statt. Das geläufigste Protokoll dafür ist das Internet Procotol (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 verbindungslose, unzuverlässige, aber weniger Overhead belastete User Datagram Protocol (UDP) und das verbindungsorientierte und datentransferzuverlässige Transmission Control Protocol (TCP). 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.

3.1.2 HTTP

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

Kommunikation
Unter einer Nachricht versteht man die 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 Kompressi-

onsalgorithmus, die Länge der Nachricht oder die verwendete Kodierung im Body.

Die erste Zeile des Nachrichtenkops ist dreiteilig und besteht bei der Anfrage aus dem Namen der Anfragemethode, dem Pfad zur angeforderten Ressource (Uniform Resource Locator, kurz URL) und der verwendet HTTP-Version. Die Anfangszeile einer HTTP-Antwort dagegen besteht zunächst aus der verwendeten HTTP-Version, gefolgt von dem zweiteiligem Status-Code. Anschließend 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. TODO BILD

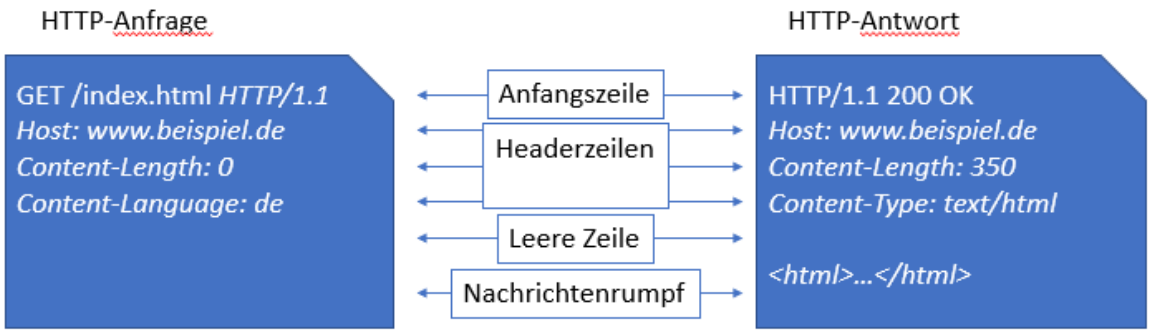

Abbildung 2: HTTP-Nachrichtenaufbau

Methoden

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

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

Statuscodes

HTTP-Antworten senden in der Anfangszeile ihrer Nachricht Statuscodes. 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.

The following are the list of HTTP response status codes		
Type	Status Codes	Examples
Informational	1xx	100 Continue, 101 Switching Protocols
Success	2xx	200 - OK , 201 - Created, 202 Accepted
Redirection	3xx	300 Multiple Choices, 301 Moved Permanently, 302 - Found
Client Error	4xx	400 Bad Request, 403 - Forbidden , 404 - Not Found , 422 - Unprocessable Entity
Server Error	5xx	500 - Internal Server Error , 503 - Service Unavailable

Abbildung 3: HTTP-Statuscodes

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, 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) enthalten. 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, dass zu Beginn der Kommunikation an den Webclient gesendet wird, bescheinigt die Vertrauenswürdigkeit des Senders. 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.

3.2 Framework

Hier steht mein Framework Text

3.2.1 Node.JS

Im Jahr 2009 veröffentlichte Ryan Dahl das Framework Node.js, das auf Googles V8-Engine, 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. - [NodeJS 1.0]

- Architektur TODO Aufzählung Eine wesentliche Eigenschaft von Node.js ist die hohe Performance. Im Folgenden soll der Unterschied der Node.js-Architektur zu traditionellen Webservern 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, die durch die Erstellung und Zerstörung von Threads entstanden, zu sparen, 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 Anfragen-aufkommen kommt es dadurch zu einer hohen Speicherauslastung des Servers. Zudem kosten die Kontextwechsel zwischen den Threads im Betriebssystem weitere Rechenzeit. [Node.js 1.05] Man spricht bei diesem Architekturkonzept auch vom Multi-Threaded Server.

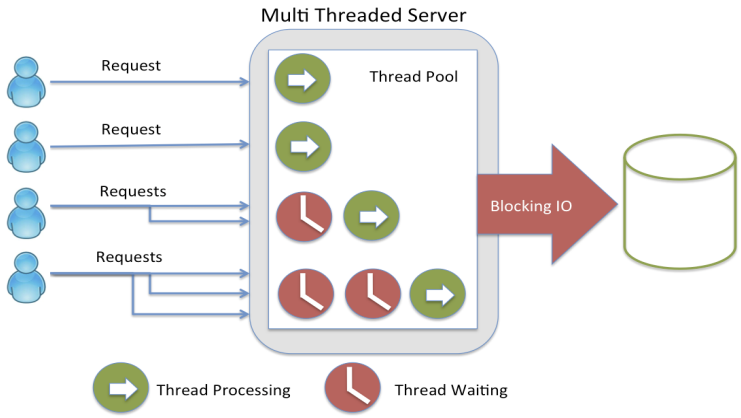

Abbildung 4: Multithreaded / Blocking I/O [Nodejs 1.1]

Node.js verfolgt einen anderen Ansatz: Anfragen werden 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 ist kann anschließend als Ereignis von der Event Loop registriert werden. Man spricht hierbei auch von ereignisgesteuerter Architektur. [1.4]

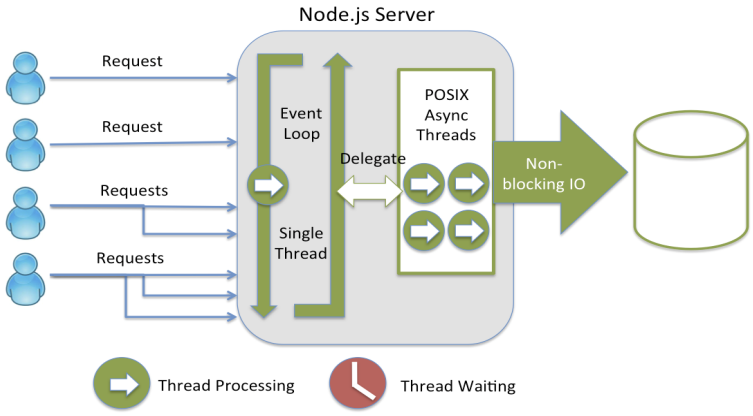

Abbildung 5: Single Threaded / Non Blocking I/0 [Nodejs 1.1]

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)

- Module TODO Aufzählung

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-Komponente, 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 in nachfolgender Abbildung dargestellt.

```
function foo() {}
function bar() {}

// Obige Funktionen exportieren:
module.exports.foo = foo;
module.exports.bar = bar;
```

Abbildung 6: Benannter Export von Modulen

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

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

Abbildung 7: Import von Modulen

Caching TODO Modules are cached after the first time they are loaded. This means (among other things) that every call to require('foo') will get exactly the same object returned, if it would resolve to the same file.
Provided require.cache is not modified, multiple calls to require('foo') will not cause the module code to be executed multiple times. This is an important feature. With it, partially doneöbjects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles. [nodejs 1.21]

- npm TODO Aufzählung Ehemals als Node Package Manager bekannt, ist npm ein Paketmanager für Node.js, entwickelt 2010 von Isaac Z. Schlueter [nodejs 1.3] 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. [1.4] 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. [1.5]

Express TODO Aufzählung „Express ist ein einfaches und flexibles Node.js-Framework von Webanwendungen, das zahlreiche leistungsfähige Features und Funktionen für Webanwen-

dungen und mobile Anwendungen bereitstellt.“ [nodejs 1.6] Es wurde im November 2010 von Douglas Christopher Wilson und weiteren Entwicklern veröffentlicht und erweitert Node.js um das Abarbeiten verschiedener HTTP-Methoden, das separate Abarbeiten von Anfragen mit verschiedenen URL-Pfaden sowie weiterer nützlicher Möglichkeiten. Im Grunde handelt es sich bei Express um ein Modul, dass durch den npm Package Manager heruntergeladen werden kann. Die aktuelle Version zum Zeitpunkt der Dokumentation [??] ist 4.17.1. [nodejs 1.65]

Beispiel

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

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

app.get('/', (req, res) => {
  res.send('Hello World!')
});

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

Abbildung 8: Einfacher Webserver [nodejs 1.8]

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 app-Verhaltens beinhaltet. [nodejs 1.8]

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.

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. [nodejs 1.9] Dabei kann die HTTP-Request direkt terminiert oder an die nächste Middleware gesendet werden. Die Verkettung der Middleware-Funktionen wird in folgender Abbildung illustriert:

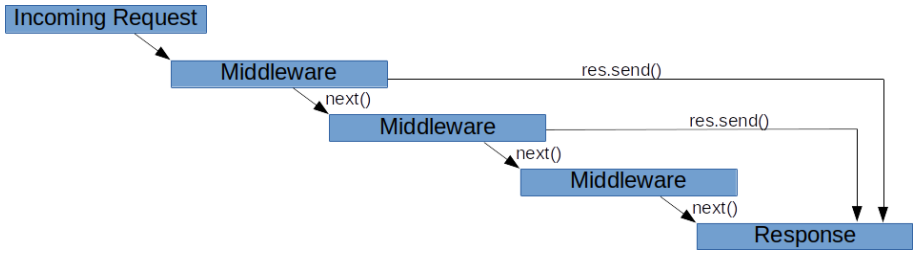

Abbildung 9: Middleware [nodejs 2.0]

Middleware: 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 neues body-Objekt innerhalb des übergebenen request-Objekts. [nodejs 2.1]

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

Abbildung 10: Express.json Middleware benutzen

Middleware: 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 (Weiterleitungsroutinen) zu erstellen. Eine Router-Instanz ist als vollständiges Middleware- und Routingsystem zu sehen und wird deshalb auch als „Mini-App“ angesehen. Der sich durch die Modularität herausziehende Vorteil ist, dass folglich unterschiedliche Anwendungsendpunkte auf entsprechende Dateien ausgelagert werden können.

```
var express = require('express');
var router = express.Router();

// middleware that is specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

Abbildung 11: Routinghandler erstellen[nodejs 2.2]

In oberem Beispiel 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:

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

Abbildung 12: Routinghandler benutzen[nodejs 2.2]

- Mongoose TODO Aufzählung Mongoose ist ein öffentliches Modul, dass zum Zeitpunkt der Dokumentation[??] im npm Package Manager in der Version 5.12.3 zur Verfügung steht. [nodejs 2.4] Bei diesem Modul handelt es sich um ein Object-Document Mapper (ODM), der es ermöglicht, asynchron mit einer NoSql-Datenbank zu kommunizieren. Mongoose ist der populärste und am weitest von MongoDB unterstützte ODM. [nodejs 2.55] Es unterstützt neben transparenter Persistenz auch die Datenvalidierung, das Erstellen von Abfragen (Queries), das Schreiben von logischem Business Code und die Übertragung zwischen Objektem im Code und der Repräsentierung dieser Objekte in der Datenbank.

Object Document Mapping (ODM)
Object-Relational Mappers (ORM) finden hauptsächlich Einsatz in objektorientieren Anwendungen, dessen Daten in relationalen Datenbanken sind. Dabei werden die Tabellen in persistente Objekte gemappt. Das Mappen ist aber auch für NoSQL-Datenbanken nützlich. [nodejs 2.56] 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[nodejs 2.7], Morphia [nodejs 2.8], Doctrine[nodejs 2.9] und Mandango[nodejs 3.0]. 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 und jede Nichtwurzel-entität[??] 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).

```
const schema = new Schema({
    attributeX: {
      type: String,   //Datentyp
      required: true, //Verpflichtend?
      lowercase: true //Kleinbuchstaben?
    }
  });
```

Abbildung 13: Mongoose Schema - Beispiel

Model

Ein Model in Mongoose ist ein aus einer Schemadefinition erstellter Konstruktor, aus denen Objekte instanziiert 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. [nodejs 3.2] Im folgenden Beispiel wird ein Model über die mongoose.model()-Funktion erstellt unter Angabe des Modelnamens und dem zu verwendenden Schema. Dieses Model wird über module.exports nach außen zur Verfügung gestellt.

```
const mongoose = require('mongoose')
const testSchema = new  mongoose.Schema({
    attributeX: {
      type: String,   //Datentyp
      required: true, //Verpflichtend?
      lowercase: true //Kleinbuchstaben?
    }
  });

module.exports = mongoose.model('test',testSchema)
```

Abbildung 14: Model erstellen und exportieren

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

```
const testModel = require(test);

var testX = new testModel(); //Erstellt neue Instanz
await testX.save(); // Speichert in die Datenbank
```

Abbildung 15: Model importieren, Objekt instanziieren und persistent speichern

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

```
const testModel = require(test);

//Create
testModel.save({ attributeX: "abc"});
//Receive
var testObjects = await testModel.find();
var testObject = await testModel.findOne({ attributeX: "abc"});
//Update
await testModel.updateOne({ attributeX: "abc"},  {attributeX: "cba"});
//Delete
await testModel.deleteMany({ attributeX: "abc"});
```

Abbildung 16: CRUD-Besipielfunktionen eines Mongoose-Models

Verbindung

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

```
const mongoose = require('mongoose')
await mongoose.connect("mongodb://127.0.0.1:27017/TestDatenbank");
mongoose.connection.on('error',(error) => console.log(error));
mongoose.connection.on('open',() => console.log('Connected to DB'));
```

Abbildung 17: Mongoose: Verbindung zur Datenbank aufbauen

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

```
const options = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useCreateIndex: true,
  useFindAndModify: false,
  autoIndex: false, // Don't build indexes
  poolSize: 10, // Maintain up to 10 socket connections
  serverSelectionTimeoutMS: 5000, // Keep trying to send operations for 5 seconds
  socketTimeoutMS: 45000, // Close sockets after 45 seconds of inactivity
  family: 4 // Use IPv4, skip trying IPv6
};
mongoose.connect(uri, options);
```

Abbildung 18: Mongoose Verbindungsoptionen[nodejs 3.3]

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.

Tabelle 2: Module

- Weitere Module TODO Aufzählung TODO warum Tabelle oben?

3.3 Language

Hier steht mein Language Text.

3.3.1 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.

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. [1] 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 Marketinggründen. Netscape wollte von der damaligen Popularität von Java profitieren. [1.05]

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. [1.06]

Im Jahre 1997 reichte Netscape JavaScript an die European Computer Manufacturers Association (kurz ECMA[ABK]), 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. [1.07] 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. [1.08] Im Juni 2021 soll die neueste Version ECMAScript 2021 veröffentlicht werden. [1.09]

Wesentliche Programmiereigenschaften „JavaScript is Not Java“ [1.091 ??]. 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. [1.091]

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 kompilierten Programmiersprachen wie C, C++ und Java durch gleiche Umsetzung der Kontrollstrukture wie den Bedingungen, Schleifen oder den booleschen Operatoren. [1.1] Wesentliche Unterschiede sind dagegen, dass JavaScript zum einen eine schwach-typisierte Sprache ist. Durch die schwache Typisierung haben Variablen keinen festen Dateityp 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. Anders als bei der klassenbasierten Programmierung, bei der Objekte aus vordefinierten Klassen instanziiert werden, werden hier Objekte durch Klonen bereits existierender Objekte erzeugt. Die Objekte, die geklont werden, sind dabei als Prototyp-Objekte zu versteh-

en. 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 Namens (dem Key) zu seiner zugehörigen Eigenschaft (dem Value) stattfindet.

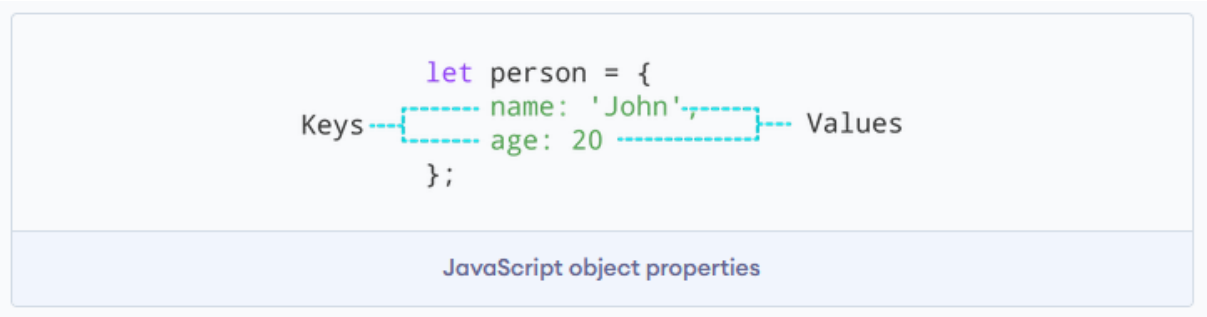

Abbildung 19: JavaScript Objekt [1.29]

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.

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). [1.3]

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. [1.4]]

3.4 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 aus Bürger der USA). Die meiste Zeit hiervon wird in Apps (ca. 90%). [10] Laut Cisco wird dieser Markt sich jedoch nicht 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-) ineffiziente Entwicklungsprozesse 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.

3.4.1 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 3.4.3.

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.

3.4.2 Plattformspezifische native Apps

Plattformspezifische oder auch native Anwendungen sind Programme, welche auf eine gewisse Plattform abzielen und in einer der davon unterstützen 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.

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

[10] `https://www.emarketer.com/content/us-time-spent-with-mobile-2019`, zuletzt aufgerufen: 26.02.2021

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]

3.4.3 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 20 kategorisiert. Im Folgenden werden jene Entwicklungsmöglichkeiten detaillierter besprochen.

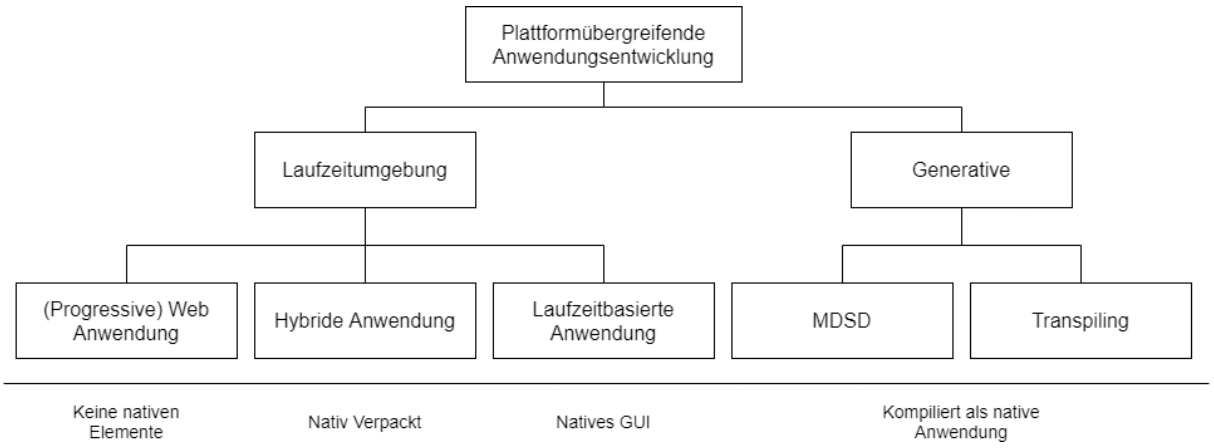

Abbildung 20: Kategorisierung verschiedener plattformübergreifender Ansätze. Erstellt nach [7]

3.4.3.1 *(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.

Abhilfe hierfür sorgt jedoch die von Google vorgestellte Design Idee *Progressive Web Apps* (PWA). Sie bietet die Möglichkeit Code in sog. *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

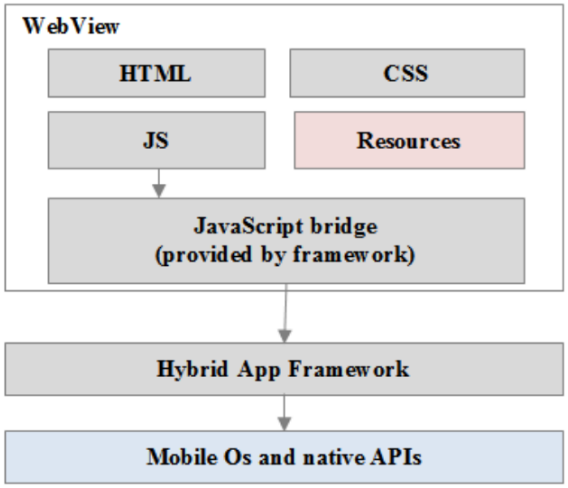

Abbildung 21: Struktur einer hybriden Anwendung [11]

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.

3.4.3.2 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]

3.4.3.3 Runtime basierte Anwendungen

Im Gegensatz zur in Kapitel 3.4.3.2 beschriebenen hybriden Anwendung, nutzen Runtime basierte Anwendungen keinen Browser des Gerätes mit einer WebView, sondern jede App besitzt 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

[11] Quelle: [10]

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 Sprachinterpretern (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]

3.4.3.4 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 dadurch der komplette Funktionsumfang wie bei einer nativen Anwendung erreicht werden. Bekannte Frameworks dieser Methode sind zum Beispiel MD_2,MAML, WebRatio Mobile, BiznessApps und Bubble.

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

3.4.3.5 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 3.5.2 näher behandelt.
Ein Hindernis dieser Art ist die erhöhte Komplexität der einzelnen Frameworks.

3.5 Frameworks zur mobilen, plattformübergreifenden Entwicklung

In den folgenden Kapiteln werden einzelne Frameworks zur mobilen, plattformübergreifenden Entwicklung vorgestellt.
In der Statistik 22 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.

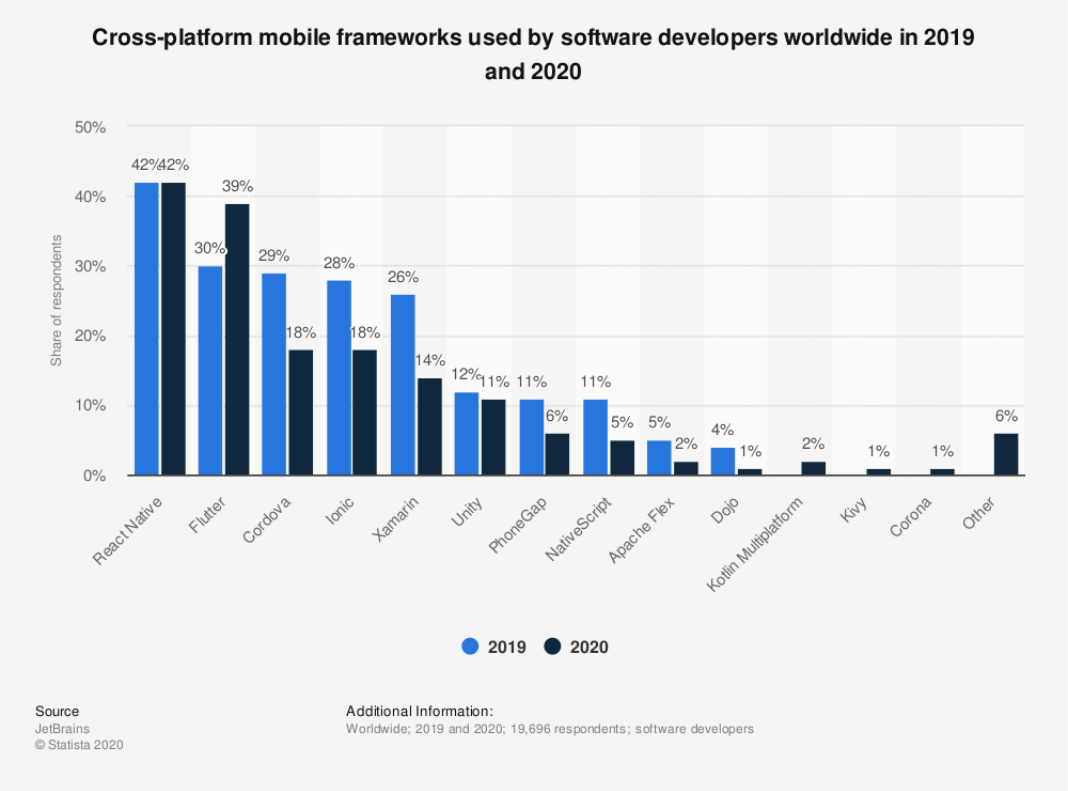

Abbildung 22: Beliebtheit nach Framework im Bereich der plattformübergreifenden mobilen Entwicklung [13]

3.5.1 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 3.4.3.3 zuordnen. [3]

[13] Quelle: www.statista.com, zuletzt aufgerufen am 22.04.2021

3.5.1.1 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.

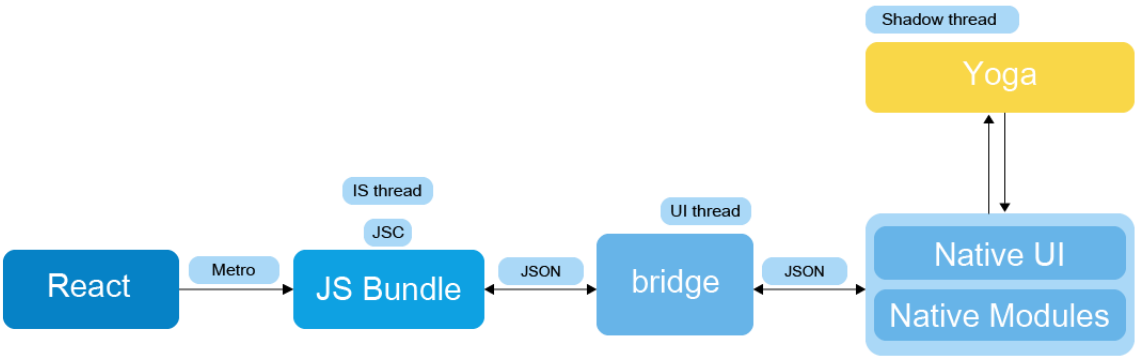

Abbildung 23: Alte Architektur von React Native [14]

- *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 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.

[14] Quelle: React Native Re-Architecture, zuletzt aufgerufen am 15.04.2021
[15] Quelle: React Native Re-Architecture, zuletzt aufgerufen am 15.04.2021

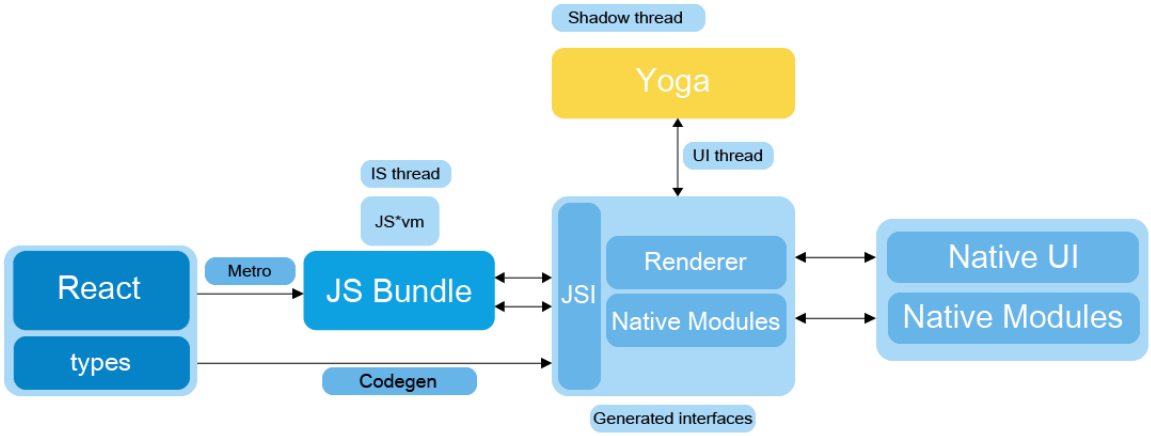

Abbildung 24: Neue Architektur von React Native [15]

Weiter wurde veralteten Legacy-Code aus dem Kern von React Native entfernt und nicht-essenzielle Teile aus dem Kern ausgelagert. Dadurch zeigt sich die aktuelle Architektur von React Native in Abbildung 24.[16]

3.5.1.2 JSX mit nativen Komponenten

React (Native) verwendet als Programmiersprache JSX. Diese ist eine syntaktische Erweiterung von JavaScript (**J**ava**S**cript e**X**tention), 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 1: 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 könnte man argumentieren, das JSX einfach eine weitere Template-Sprache sei - ähnlich HTML oder XAML. Jedoch ist dies falsch, da (wie oben bereits erwähnt) JSX lediglich eine syntaktische Erweiterung von JavaScript ist, also man inmitten von JSX Objekten JavaScript schreiben kann.
Weiterhin ist interessant, dass JSX Cross Site Scripting vorbeugt, indem der React DOM alle eingesetzte Werte zunächst als normalen 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>
```

[16] Quelle: React Native's re-architecture in 2020

```
8       );
9     }
10
11    export default Cat;
```

Listing 2: Native Komponenten

In dem Codebeispiel 2 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 zum speziellen Anwendungsfall passen.

3.5.1.3 Komponenten

Gleichzeitig erlaubt React Native aber auch wiederverwendbare Komponenten in JavaScript aus den Kernkomponenten zusammenstellen. Hierzu lassen sich einzelne Komponente jedoch nicht nur in einander verschachteln, um hier beispielsweise einen `Text` innerhalb einer `View`[17] anzeigen zu lassen. Zusätzlich ist es möglich sogenannte „props“also Eigenschaften (engl.: properties), ähnlich einer normalen Funktion mitzugeben. In diesem Beispiel sind das hier die 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   );
```

[17] Eine `View` ist die Basiskomponente einer Benutzeroberfläche. In einer `View` wiederum können wieder Views verschachtelt sein.

```
}

export default Cafe;
```

Listing 3: Eigene Komponenten

Für eine interaktive Benutzeroberfläche fehlt jedoch noch das Kernprinzip eines deklarativen UI:

3.5.1.4 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 28 gut visualisiert.

Um einer Funktion einen *State* hinzuzufügen, ermöglicht React (und auch React Native) seit v16.8 dies durch Hooks. Hooks sind Funktionen, welche es Entwicklern ermöglicht sich in React Features einzuhaken. Bisher wurde dies über Klassen Komponenten ermöglicht, dadurch ist es jedoch komplizierter *stateful logic* zwischen Komponenten wiederzuverwenden. Daher entkoppelt man diese Logik von den Komponenten und erlaubt unter anderem das separate Testen.

```
  import React, { useState } from "react";
  import { Button, Text, View } from "react-native";

  const Cat = (props) => {
    // make isHungry stateful
    const [isHungry, setIsHungry] = useState(true);

    return (
    <View>
    <Text>
    // show text dependent on isHungry state
    I am {props.name}, and I am {isHungry ? "hungry" : "full"}!
    </Text>
    <Button
    onPress={() => {
        setIsHungry(false);
    }}
    disabled={!isHungry}
    title={isHungry ? "Pour me some milk, please!" : "Thank you!"
       }
    />
    </View>
    );
  }

  const Cafe = () => {
    return (
    <>
    <Cat name="Munkustrap" />
    <Cat name="Spot" />
    </>
```

```
31      );
32    }
33
34    export default Cafe;
```

Listing 4: State mit `useState` Hook

Im Beispiel 4 wird in Zeile 6 der `useState` Hook verwendet. Die Funktion erzeugt eine State Variable 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]

3.5.2 Flutter

Flutter ist eine open-source SDK entwickelt von Google und ist geschrieben in C, C++ und Dart. Flutter erlaubt es Anwendungen für Android, iOS, Web und Desktop basierend auf einem Code zu erstellen und ist zudem die primäre Methode für Google Fuchsia, Googles Betriebssystem.[18] Flutter verwendet Skia als 2D Grafikbibliothek, welche auch von Chrome, Firefox und Android verwendet wird. Zudem basiert Flutter auf der Dart Plattform welche das Compilieren auf 32-bit und 64-bit ARM Prozessoren, auf Intel x64 Prozessoren und in JavaScript ermöglicht (siehe Abbildung 25). Daher ist Flutter eine, wie in Kap. 3.4.3.5 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 also nicht komplett neu kompiliert werden. Wird die App nun veröffentlicht, wird sie in die Maschinencode der beschriebenen Plattformen übersetzt.

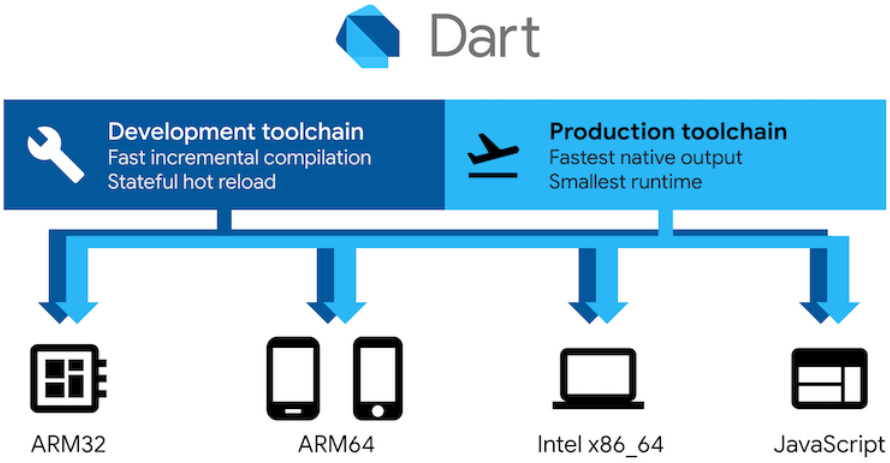

Abbildung 25: Kompatibilität der Dart Plattform [19]

[18] Quelle: `https://fuchsia.dev/`
[19] Quelle: `https://github.com/flutter/flutter`

3.5.2.1 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 the layer below, and every part of the framework level is designed to be optional and replaceable.[20]

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]

3.5.2.2 Dart

Während der Entwicklung von Flutter standen sicherlich mehrere Sprachen zur Auswahl:
Wie in Kapitel 3.4.3 gelernt, gibt es viele unterschiedliche Ansätze mit beispielsweise webbasierten 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?

Dart allgemein ist C ähnlich, also für viele Entwickler leicht(er) leserlich. Es ist eine objektorientierte Sprache und besitzt einen Garbage Collector.
Dart ist designet als eine Client-fokussierte Sprache, welche gleichermaßen Entwicklung (subsecond stateful hot reload) und Produktion in allen möglichen Zielplattformen (Web, Mobile und Desktop) priorisiert. Dadurch erhält man mit dieser Sprache eine Effizienz optimierte Entwicklungsphase, sowie ebenfalls die Möglichkeit eine Code-Basis in unterschiedliche Plattformen zu kompilieren (siehe Abbildung 25).
Es bietet zudem auch *sound-null-safety* - Werte können also nicht null sein, außer man legt dies fest. Damit kann es null Exceptions während der Laufzeit durch statische Code Analyse vorbeugen.

[20] `https://flutter.dev/docs/resources/architectural-overview`
[21] Quelle: `https://github.com/flutter/flutter`

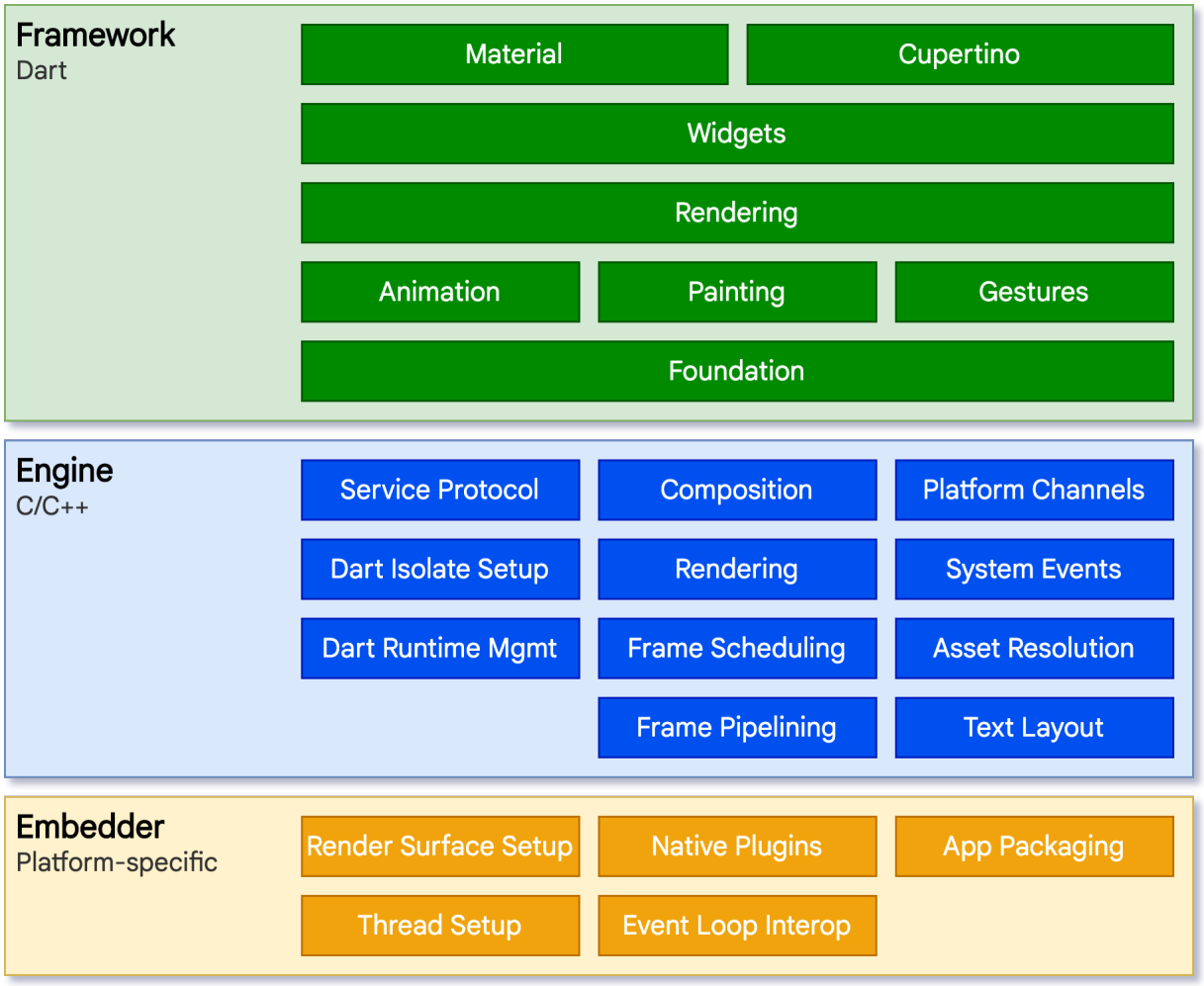

Abbildung 26: Bibliotheken und Ebenen der Flutter Plattform [21]

3.5.2.3 Widgets

Wie bereits beschrieben sind Widgets wiederverwendbare Kompositionsbausteine, 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 wir 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

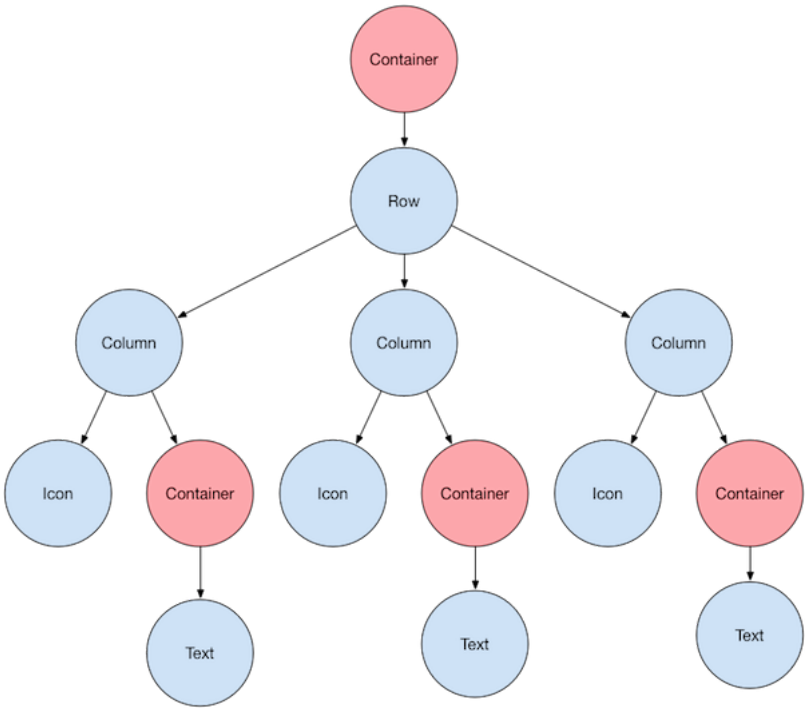

Abbildung 27: Widget Baum einer beispielhaften Anwendung [22]

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

3.5.2.4 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*. Hier wird dies durch das Framework gelöst.

Abbildung 28: Deklarative Benutzeroberfläche [23]

3.6 IDE

Hier steht mein IDE Text.

[22] Quelle: `https://flutter.dev/docs/development/ui/layout`
[23] Quelle: `https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative`

3.7 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 Datenbank, 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. [NoSQL 1.1] Zu den bekanntesten NoSQL-Datenbanken gehören beispielsweise Apache Cassandra, MongoDB und CouchDB.

3.7.1 NoSQL-Datenbanktypen

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

Graphendatenbanken Dieses Datenbankenkonzept 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.

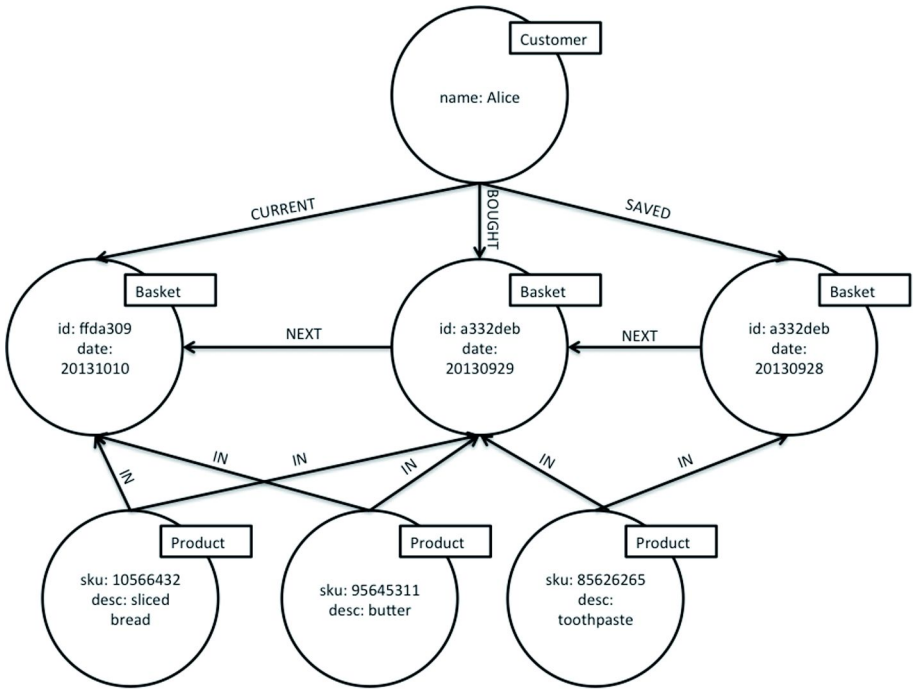

Abbildung 29: Graphikdatenbank Beispiel [NoSql 1.2]

Dokumentenorientierte Datenbanken
Im Kontext der dokumentenorientierte Datenbanken sind Dokumente Objekte mit Eigenschaften, die in einer Sammlung gespeichert werden. Während eine Sammlung eine Tabelle im relationalen Datenbank widerspiegelt, 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 Lsite 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.

```
{
"Vorname" : "Max" ,
"Nachname" : "Mustermann" ,
"Telefon-Nr." : "0123456",
"Adresse" : "Musterstraße 34, Musterstadt" ,
"Kinder" : ["Musterkind1" , "Musterkind2" ] ,
"Alter" : 33
...
}
```

Abbildung 30: Dokumentenorientiert JSON Beispiel [NoSql 1.4]

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.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Abbildung 31: Key-Value Beispiel [NoSql 1.5]

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.

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

Abbildung 32: Spaltenorientierte Datenbank Beispiel [NoSql 1.5]

Nachfolgend ist die zeilenoriente Speicherung dargestellt. Sie bietet vorallem Vorteile bei Abfragen, bei denen Informationen aus meheren Spalten benötigt werden.

1	Buntlack RAL 7035	25	2	Heizkörperfarbe	50	3	Grundierfarbe	15

Abbildung 33: Key-Value Beispiel [NoSql 1.5]

Folglich wird die spaltenorientiere Speicherung abgebildet. Sie eignen sich sehr gut für die Auswerttung von Aggregaten

1	2	3	Buntlack RAL 7035	Heizkörperfarbe	Grundierfarbe	25	50	15

Abbildung 34: Key-Value Beispiel [NoSql 1.5]

3.7.2 BASE

s

3.7.3 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 [NoSQL 1.6].

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.

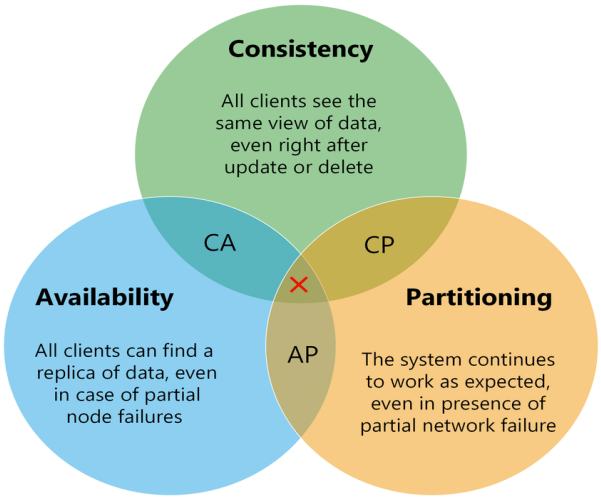

Abbildung 35: Visualization-of-CAP-theorem [NoSql 1.67]

3.7.4 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-verbreiteste NoSQL-Datenbank. (Stand April 2021) [MongoDB1.7] Die Intention der Gründer war es, eine Datenbank mit höherer Skalierbarkeit, Flexibilät und Performance zu entwerfen, die auf auf einer einfachen Handhabung beruht. [MongoDB1.65]

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.

2.2.4.3 Aufbau/Struktur

2.2.4.3 Dokumente

Während MongoDB für Datenaustausch das JSON-Format nutzt, hält es seine Dokumente im Binary JSON-Format (BSON), einer binärcodiertem 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 werden ist BSON um zusätzliche Datentypen wie 32- und 64-bit Integer oder das Datum erweitert. [MongoDb1.8 https://www.mongodb.com/json-and-bson]

CRUD

Im Vergleich zu relationalen Datenbanken benutzt MongoDB keine Abfragesprache wie SQL, sondern

Wired Tiger? Storage Engine Replica Sets Oplog Sharding

Technische Grundlagen / Replica Transaktionen

Verwaltungswerkzeuge Mongo Shell Treiber Grafische Oberflächen

```
/* JSON */
{"hello": "world"}  // „key“:“value“

/* BSON */
\x16\x00\x00\x00           // total document size
\x02                       // 0x02 = type String
hello\x00                  // field name
\x06\x00\x00\x00world\x00  // field value
\x00                       // 0x00 = type EOO ('end of object')
```

Abbildung 36: Graphikdatenbank Beispiel [NoSql 1.2]

3.8 Representational State Transfer - Application Programming Interface

Hier steht mein Database Text.

Es gibt verschiedene Datenbankmodelle.

3.9 Firebase

Firebase ist eine Backend-as-a-Service (BaaS) Plattform von Google für mobile oder Web-Anwendungen. Sie soll es dem Entwickler ermöglichen, einfacher und effizienter Funktionen auf verschiedenen Plattformen bereitzustellen stellt 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.[24] Teile der SDK stehen seit der Google I/O 2017 unter der Apache 2.0 Lizenz, sind somit also Open-Source.[25]

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]

3.9.1 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.

3.9.2 Firestore

Als Datenbank Lösung bietet Firebase zwei unterschiedliche Produkte an: Firestore und Realtime Database. Firestore ist hier neuer, jedoch ersetzt es Realtime Database nicht.
Firestore ist eine flexible und auf Skalierung ausgesetzte NoSQL Cloud Datenbank, welche 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.

[24] firebase.googleblog.com, zuletzt aufgerufen am 03.05.2021
[25] opensource.googleblog.com, zuletzt aufgerufen am 03.05.2021
[27] Quelle: [13]

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.

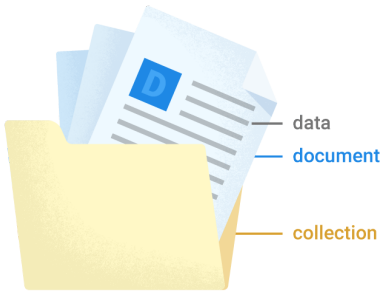

Abbildung 37: Datenmodell in Firebase [27]

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 könne 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 5: Beschränkung des Zugriffs auf Dokumente der Sammlung `cities`

Im Beispiel 5 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 Produktionseinsatz 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 Löschrechte ganzer Dokumente erhält man über den `delete` Operator.

Sicherheitsregeln werden gleich dem Datenmodell hierarchisch aufgebaut und ermöglichen differenzierte Zugriffsbeschränkungen auf jeder Ebene. In Codebeispiel 6 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.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /cities/{city} {
      allow read, write: if <condition>;

      // Explicitly define rules for the 'landmarks'
         subcollection
      match /landmarks/{landmark} {
        allow read, write: if <condition>;
      }
    }
    match /villages/{document=**} {
      allow read, write: if <condition>;
    }
  }
}
```

Listing 6: 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 7 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]

```
service cloud.firestore {
  match /databases/{database}/documents {
    // If you update a city doc, you must also
    // update the related country's last_updated field.
    match /cities/{city} {
      allow write: if request.auth != null &&
      getAfter(
      /databases/$(database)/documents/countries/$(request.
         resource.data.country)
      ).data.last_updated == request.time;
    }

    match /countries/{country} {
      allow write: if request.auth != null;
    }
  }
}
```

Listing 7: Datenvalidierung für atomare Operationen

3.9.3 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 3.9.2 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 8 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 write: if request.resource.size < 5 * 1024 * 1024
5        && request.resource.contentType.matches('image/.*');
6      }
7    }
```

Listing 8: Validierung nach Dateigröße

Außerdem lassen sich die

3.9.4 Cloud Functions

3.9.5 Analytics

3.9.6 Google AdMob, Google Ads

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

Tabelle 3: Nutzer-Item Matrix mit Bewertungen. Jede Zelle $r_{u;i}$ steht hierbei für die Bewertung des Nutzers u an der Stelle i

3.10 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 3.10 als Matrix dargestellt werden.

3.10.1 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.

3.10.1.1 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.

3.10.1.2 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.

3.10.2 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 "verfängtßich quasi in eine Richtung.[1]

3.10.3 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 unterscheidet man in zwei Typen:[1]

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.

3.10.4 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 $s : E \times E \to \mathbb{R}$ 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
- Skalarprodukt
- Euklidischer Abstand

3.10.4.1 Cosinus-Funktion

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

3.10.4.2 Skalarprodukt

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

3.10.4.3 Euklidischer Abstand

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

`https://dl.acm.org/doi/pdf/10.1145/3383313.3412488` `http://www.microlinkcolleges.net/elib/files/undergraduate/Photography/504703.pdf`

4 Konzept?

Test..

5 Auswahl geeigneter Technologie

Hier steht mein TechnologieAuswahl Text...

5.1 Server

Anforderungen sind ...

5.2 Datenbank

Es gibt verschiedene Datenbankmodelle. Bei grossen Datenbank spielt .. eine wichtige Rolle..

5.3 Kommunikationsschnittstelle

Verschiedene Modelle für die Kommunikation verteilter Systeme...

6 Backend-Implementierung

Unter Backend versteht man... , Sie differenziert durch... von den ... .

6.1 Server

Hier steht mein BackendServer Text.

6.1.1 Einrichtung

Zunächst...

6.1.2 Sicherheit

Den Server gilt es zu schützen.

6.1.3 Webserver

Den Server gilt es zu schützen.

6.2 Datenbank

Hier steht mein BackendDatenbank Text.

6.2.1 Einrichtung

Hier steht mein Imlementierung Text.

6.3 Kommunikationsschnittstelle

Hier steht mein BACKENDKommunikationschnittstelle Text.

6.3.1 Implementierung

7 Funktionen/Komponenten

7.1 Swipe/Aussuchen/Voting

7.2 Matches/Chat

7.3 Film-/Serienvorschläge

7.4 Gruppenorgien

7.5 Gespeicherte Filme/Filmliste

7.6 Zugänglichkeit/Behindertenfreundlichkeit

8 Benutzeroberflächen

8.1 Home-Screen

8.2 Gruppen

8.3 Chat

8.4 Filmliste

9 CodeBeispiele

10 Probleme

11 Fazit

Literaturverzeichnis

[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`, Zuletzt aufgerufen am: 13.04.2021

[4] Facebook Inc. (2021) React documentation. [Online] Verfügbar: `https://reactjs.org/docs/getting-started.html`, Zuletzt aufgerufen am: 13.04.2021

[5] Flutter (2021) Flutter architectural overview [Online] Verfügbar: `https://flutter.dev/docs/resources/architectural-overview`, Aufgerufen am: 04.03.2021

[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-interr 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`, Aufgerufen am: 25.04.2021