

# **Progressive Web App**

## **Studienarbeit**

des Studiengangs IT Automotive  
an der Dualen Hochschule Baden-Württemberg Stuttgart

von

**Emmelie Beitlich, Finn Freiheit**

Oktober 2020

**Bearbeitungszeitraum**  
**Matrikelnummer, Kurs**  
**Gutachter**

12 Wochen  
2533282, ITA19  
Dipl.-Ing. (FH) Peter Pan

## **Erklärung**

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema: *Progressive Web App* 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.

Stuttgart, Oktober 2020

---

Emmelie Beitlich, Finn Freiheit

## **Abstract**

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>VI</b>
<b>Abbildungsverzeichnis</b>	<b>VII</b>
<b>Tabellenverzeichnis</b>	<b>IX</b>
<b>Listings</b>	<b>X</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Problemstellung . . . . .	2
1.3. Begriffserklärung . . . . .	2
1.4. Aufbau der Arbeit . . . . .	3
<b>I. Theoretische Grundlagen</b>	<b>5</b>
<b>2. Architektur</b>	<b>6</b>
<b>3. Angular</b>	<b>8</b>
3.1. Strukturdirektiven . . . . .	8
3.2. Interpolation . . . . .	9
3.3. Property Binding . . . . .	9
3.4. Event Binding . . . . .	9
3.5. Backend Anbindung . . . . .	10
<b>4. Node.js</b>	<b>11</b>
4.1. Überblick . . . . .	11
4.2. Charakteristika . . . . .	11
4.2.1. Single-Thread-Architektur . . . . .	11
4.2.2. Ereignisschleife . . . . .	11
4.3. Vorteile durch die Nutzung von JavaScript . . . . .	12
4.4. Node Package Manager . . . . .	12
4.4.1. NPM Aufgabenautomatisierung . . . . .	12
4.5. Deskriptordatei package.json . . . . .	13
4.6. Node Express . . . . .	13
4.6.1. Vorteile . . . . .	13
4.6.2. Routing . . . . .	14
4.6.3. Middleware . . . . .	15

<b>5. Datenbanken</b>	<b>18</b>
5.1. SQL . . . . .	18
5.1.1. ACID . . . . .	18
5.2. NoSQL . . . . .	18
5.2.1. Modelle . . . . .	19
5.3. SQL vs NoSQL . . . . .	20
<b>6. MongoDB</b>	<b>22</b>
6.1. Überblick . . . . .	22
6.2. Speicherung der Daten . . . . .	22
6.2.1. JSON-basierte Speicherung . . . . .	22
6.2.2. BSON . . . . .	23
6.2.3. Identifier . . . . .	23
6.2.4. Dateiupload . . . . .	24
<b>7. Progressive Web App</b>	<b>25</b>
7.1. Web-App-Manifest . . . . .	25
7.1.1. short_name . . . . .	26
7.1.2. icons . . . . .	26
7.1.3. start_url . . . . .	26
7.1.4. display . . . . .	26
7.1.5. theme_color . . . . .	27
7.1.6. Debugging des Manifests . . . . .	27
7.2. Service Workers . . . . .	28
7.2.1. Der Lebenszyklus eines Service Workers . . . . .	28
7.3. Geräteschnittstellen . . . . .	30
7.4. Push-Notifications . . . . .	31
7.4.1. Push-Registrierung . . . . .	33
7.4.2. Informationsaustausch . . . . .	34
7.4.3. Zusammenfassung Push-Notifikation . . . . .	34
7.5. Progressive Web Apps in Angular . . . . .	35
7.6. Endgerät-Emulation . . . . .	36
<b>II. Praktische Umsetzung</b>	<b>40</b>
<b>8. Funktionalitäten</b>	<b>41</b>
<b>9. Layout</b>	<b>42</b>
9.1. Layout Webanwendung . . . . .	42
9.1.1. CSS-Grid . . . . .	43
<b>10. Node.js und Node Express</b>	<b>45</b>
10.1. Routing . . . . .	45
10.1.1. Endpunkte . . . . .	45

10.1.2. Request-Funktionen . . . . .	48
10.1.3. Verwendung der Endpunkte . . . . .	50
<b>11. MongoDB</b>	<b>51</b>
11.1. Mongoose . . . . .	51
11.2. Schemata . . . . .	51
11.3. Dateiupload . . . . .	52
<b>12. Installation der PWA</b>	<b>54</b>
<b>13. Verwendung von Geräteschnittstellen</b>	<b>58</b>
<b>14. Push-Notifikation</b>	<b>59</b>
14.1. Registrierung . . . . .	59
14.2. Push-Nachrichten im Backend . . . . .	61
<b>15. Untersuchung der Plattformunabhängigkeit</b>	<b>63</b>
15.1. Google Chrome und Android . . . . .	63
15.2. Apple Safari und IOS . . . . .	64
<b>Literatur</b>	<b>66</b>
<b>Anhang</b>	<b>70</b>
.1. HTML . . . . .	70
.1.1. Attribute . . . . .	71
.2. CSS . . . . .	71
.2.1. CSS in die HTML-Datei einbinden . . . . .	72
.3. JavaScript . . . . .	73
.3.1. JavaScript in die HTML-Datei einbinden . . . . .	73

# Abkürzungsverzeichnis

<b>PWAs</b>	Progressive Web Apps
<b>PWA</b>	Progressive Web App
<b>JSON</b>	JavaScript Object Notation
<b>URL</b>	Uniform Resource Locator
<b>API</b>	Application Programming Interface
<b>DOM</b>	Document Object Model
<b>SPA</b>	Single Page Application
<b>SQL</b>	Structured Query Language
<b>CRUD</b>	create, read, update and delete
<b>HTTP</b>	Hypertext Transfer Protocol
<b>HTML</b>	Hyper Text Markup Language
<b>CSS</b>	Cascading Style Sheets
<b>REST</b>	Representational State Transfer
<b>NoSQL</b>	Not only SQL
<b>CLI</b>	Command Line Interface
<b>AVD Managers</b>	Android Virtual Device Manager
<b>IETF</b>	Internet Engineering Task Force
<b>VAPID</b>	Voluntary Application Server Identification

# Abbildungsverzeichnis

2.1.	Architektur der gesamten Anwendung . . . . .	6
5.1.	CAP-Theorem [Vet+22] . . . . .	19
5.2.	Modelle einer NoSQL Datenbank [Vet+22] . . . . .	20
6.1.	Aufbau von MongoDB [ES15, S.30] . . . . .	23
7.1.	display in „Standalone“-Einstellung . . . . .	26
7.2.	display in „Minimal User Interface“-Einstellung . . . . .	26
7.3.	Theme Color auf Weiß geändert . . . . .	27
7.4.	Entwicklereinstellungen Web-App-Manifest . . . . .	27
7.5.	Sequenzdiagramm für die Ereignisse einer Anmeldung, eines Push-Nachrichttransfers und einer Abmeldung [BT16] . . . . .	32
7.6.	Emuliertes Android Handy in Android Studio . . . . .	37
7.7.	Emuliertes iPhone in XCode . . . . .	38
9.1.	CSS-Grid Raster der Hauptanwendung. . . . .	43
12.1.	Installation der PWA aus dem Google Chrome Browser durch die Auswahl des Download-Buttons in der URL-Suchleiste des Browsers. . . . .	54
12.2.	Die installierte PWA als Desktop-Anwendung . . . . .	55
12.3.	PWA im Apple Safari Browser in einem IOS System . . . . .	56
12.4.	Installation der PWA über die Auswahl <i>Zum Home-Bildschirm</i> . . . . .	56
12.5.	Eingabe einer Bezeichnung für die PWA-App . . . . .	57
12.6.	Die installierte PWA-App auf dem Home-Bildschirm eines IOS-Systems . .	57
14.1.	Nutzerabfrage um Push-Nachrichten versenden zu können. Abfrage erfolgt erst nachdem der Button <b>SUBSCRIBE TO PUSH</b> in der oberen Navigationsleiste ausgewählt wurde. . . . .	60
14.2.	Push-Nachricht, mit dem Payload aus Listing 14.2 . . . . .	62
15.1.	Google Chrome Developer Tools zum Einsehen der im Cache gespeicherten Dateien . . . . .	64
15.2.	Apple Safari verfügt nicht über ein Push-Manager und ist somit nicht in der Lage Push-Nachrichten zu erhalten . . . . .	65
.3.	HTML Grundgerüst . . . . .	71

.4. Grüne Überschrift . . . . .	72
---------------------------------	----

# Tabellenverzeichnis

1.1. Plattformen und die dazu benötigten Programmiersprachen . . . . .	1
10.1. User Endpunkte . . . . .	46
10.2. Comments Endpunkte . . . . .	46
10.3. Posts Endpunkte . . . . .	47
10.4. Likes Endpunkte . . . . .	47
11.1. User-Schema . . . . .	51
11.2. Post-Schema . . . . .	52
11.3. Comment-Schema . . . . .	52
11.4. Like-Schema . . . . .	52

# Listings

3.1. Interpolation im Template . . . . .	9
3.2. Property Binding . . . . .	9
3.3. Event Binding . . . . .	10
4.1. Express Modul installieren . . . . .	13
4.2. Routing in Express . . . . .	14
4.3. Routing-Funktion mit zwei Callback-Funktionen . . . . .	14
4.4. modulare Routing Methode [Exp22d] . . . . .	15
4.5. Verwendung der Instanz . . . . .	16
4.6. Error-Handling-Middleware . . . . .	16
4.7. Module von Drittanbietern integrieren [Exp22e] . . . . .	17
7.1. Registrierung des Service Workers . . . . .	29
7.2. Push-Registrierung unter Berücksichtigung der Konzepte des Progressive Enhancements . . . . .	33
7.3. Angular <i>ngsw-config.json</i> -Datei zur Angabe der Ressourcen, die durch den Service Worker in den Cache gespeichert werden sollen . . . . .	36
9.1. HTML-Grundgerüst der Webanwendung . . . . .	42
9.2. Realisierung des Grid-Rasters in CSS . . . . .	44
10.1. GET-Request . . . . .	48
10.2. POST-Request . . . . .	48
10.3. PATCH-Request . . . . .	49
10.4. DELETE-Request . . . . .	49
10.5. Verwendung der API-Routen . . . . .	50
11.1. Herstellen der Verbindung zu MongoDB . . . . .	51
11.2. Erstellen eines neuen Schemas . . . . .	52
11.3. Zuweisen der URL der Datenbank . . . . .	52
11.4. Festlegen der zugelassenen Dateiformate . . . . .	53
11.5. Festlegen des Dateinamens mit Zeitstempel . . . . .	53
11.6. Speicherort des Bildes . . . . .	53
14.1. Funktion zur Registrierung beim Push-Dienst . . . . .	61
14.2. Mindestanforderung an ein Payload, für eine Push-Nachricht . . . . .	61

1.	Grundgerüst einer HTML-Seite . . . . .	70
2.	HTML Attribute . . . . .	71
3.	Die generelle Syntax für CSS-Eigenschaften . . . . .	71
4.	Grüne Überschrift CSS . . . . .	72
5.	CSS-Datei in HTML verlinken . . . . .	72
6.	CSS-Datei in HTML einbinden . . . . .	73
7.	JavaScript in HTML einbinden . . . . .	74

# 1. Einleitung

## 1.1. Motivation

Folgendes Szenario soll einen Einblick in die Vorteile von Progressive Web Apps ([PWAs](#)) geben.

Ein junges Startup aus IT-Studenten hat eine Idee für eine Applikation und möchte diese umsetzen. Ihr Ziel ist es, diese Applikation an so viele Nutzer wie möglich zu verbreiten. Die Applikation soll demnach für folgende Plattformen verfügbar sein - siehe Tabelle 1.1.

Plattform	Programmiersprache
IOS	Swift
Android	Java oder Kotlin
MacOS	Swift
native Windows	C++ oder C#
Webbrowser	JavaScript, HTML und CSS

Tabelle 1.1.: Plattformen und die dazu benötigten Programmiersprachen

Wie man anhand der Tabelle sehen kann, kommt eine Vielzahl an unterschiedlichen Programmiersprachen zum Einsatz, um die Applikation über mehrere Plattformen zu verbreiten. Jedoch verfügt das junge Startup nicht über die benötigten Kapazitäten, um die Applikation in jeder dieser Programmiersprachen implementieren und warten zu können.

Damit die Applikation dennoch an möglichst viele Kunden gelangt, sucht das Startup nach einer anderen Lösung, um die aufwendige Implementierung zu umgehen. Dabei kommt die Idee auf, eine Progressive Web App ([PWA](#)) zu entwickeln. Eine PWA ist eine Webanwendung mit erweiterten Funktionen. Die Besonderheit liegt darin, dass sie - einmal implementiert - auf sämtliche Plattformen installiert werden kann. Dadurch ist sie plattformunabhängig und die Verwendung sämtlicher Programmiersprachen könnte umgangen werden.

## 1.2. Problemstellung

Die These, dass Progressive Web Apps plattformunabhängig und vollumfänglich eingesetzt werden können, soll im Verlauf dieser Arbeit untersucht werden.

Hierfür wird eine Applikation entwickelt, die die Grundfunktionen einer PWA implementiert. Die Anwendung soll

- installierbar sein,
- auf Geräteschnittstellen zugreifen können,
- offline verfügbar sein und
- Push-Benachrichtigungen ermöglichen.

Im Anschluss wird die Anwendung auf jenen Plattformen ausgeführt, die in Tabelle 1.1 aufgeführt sind. Dabei wird untersucht, ob alle beschriebenen Funktionalitäten vorhanden und voll ausführbar sind.

## 1.3. Begriffserklärung

Der Begriff *Progressive Web App* setzt sich aus den Begriffen *Web App* und *Progressive Enhancement* zusammen. Eine *Web App* (zu deutsch: Webanwendung) ist eine mithilfe von JavaScript, HTML und CSS entwickelte Applikation. Der zweite Begriff entstand durch Steve Champeon im Jahre 2003 in seiner Publikation mit dem Titel *progressive enhancement and the future of web design* [Cha].

Unter dem Begriff *Progressive Enhancement* (zu deutsch: Progressive Verbesserung) verbirgt sich das Ziel, Webseiten so zur Verfügung zu stellen, dass jeder Webbrowser in der Lage ist, die grundlegendste Form einer Webseite darzustellen. Hierbei ist es unabhängig davon, über welche Version der Browser oder das Endgerät verfügt. Alle zusätzlichen Funktionalitäten, die sich nur unter Verwendung moderner Browser und Endgeräte nutzen lassen, werden erst im Anschluss in Form von Skripten eingebunden.

Um PWAs nutzen zu können, werden die neusten Funktionen moderner Webbrowers benötigt, darunter *service workers*(Abschitt 7.2) und *web app manifests*(Abschnitt 7.1).

Google hat das Konzept von PWAs im Jahr 2015 vorgestellt und ist seitdem maßgeblich an der Entwicklung beteiligt. Das Ziel bei der Entwicklung von PWAs liegt darin, die Vorteile Nativer Applikationen mit denen der Webanwendung zu kombinieren.

Native Applikationen beziehungsweise plattformspezifische Applikationen sind sehr funktionsreich und zuverlässig. Weitere Vorteile sind, dass sie :

- netzwerkunabhängig funktionieren,
- lokale Dateien aus dem Dateisystem lesen und schreiben können,
- auf Hardwareschnittstellen wie USB und Bluetooth zugreifen können,
- mit Daten des Geräts interagieren können, wie zum Beispiel Fotos oder aktuell spielende Musik.

Webapplikationen wiederum sind sehr gut erreichbar, sie können verlinkt, über Suchmaschinen gefunden und geteilt werden.

Mithilfe von PWAs können Applikation erzeugt werden, die:

- installierbar sind, Abschnitt [7.1](#)
- auf Geräteschnittstellen zugreifen können, Abschnitt [7.3](#)
- netzwerkunabhängig funktionieren, Abschnitt [7.2](#) und Abschnitt [7.5](#)
- Push-Notifications versenden können, Abschnitt [7.4](#)

PWAs sind laut Sam Richard und Pete LePage das Beste aus zwei Welten [\[Sam\]](#). Mithilfe von progressiver Verbesserung werden die modernen Funktionen von Browsern genutzt, um die Vorteile von plattformspezifischen Anwendungen nutzen zu können. Sind die dafür benötigten Funktionen wie beispielsweise *service workers* nicht vorhanden, können die Grundfunktionen der Anwendung dennoch im Web genutzt werden.

### 1.4. Aufbau der Arbeit

Ziel der Arbeit ist es, eine Progressive Webanwendung zu entwickeln und anhand dieser die Umsetzung und Reife ihrer Charakteristika zu bewerten. Dazu müssen zunächst die theoretischen Grundlagen geschaffen werden. Hier gilt es zwischen dem Frontend, dem Backend und den PWA-spezifischen theoretischen Inhalten zu unterscheiden.

Für das Frontend wird das Web-Framework Angular verwendet, dessen Eigenschaften und Funktionsweisen zunächst erläutert werden sollen. Das Backend enthält eine MongoDB, in der alle Daten von Userdaten über hochgeladene Bilder bis hin zu Kommentardaten gespeichert werden sollen. Im Laufe der Beschreibung des Aufbaus und der Funktionsweise der Datenbank wird außerdem erläutert, wieso man sich für eine nicht-relationale Datenbank und speziell für MongoDB entschieden hat.

## *1. Einleitung*

---

Um die Verbindung von Frontend und Backend herzustellen, benötigt man einen Server, der das Routing von Anfragen übernimmt. Dazu kommt Node.js zum Einsatz. Speziell das Modul Express soll beschrieben werden, da über jenes das Routing – teils auch über Middleware – übernommen wird. Nachdem der aktuelle technische Stand des Aufbaus einer Webpage erläutert wurde, soll im Anschluss darauf eingegangen werden, wie man aus einer normalen Webanwendung eine PWA entwickeln kann. Web App Manifests, Service Workers und Push Notifications sind dabei wichtige Elemente, die beschrieben werden müssen, um alle Grundlagen einer PWA zusammenzufassen.

Nach Schaffen der technischen Grundlagen sollen diese praktisch umgesetzt werden. Dazu soll zunächst bestimmt werden, was die Webanwendung können soll. Dazu gehört das Layout der Anwendung, die Optionen, welche man während der Anwendung ausführen kann sowie die Reaktionen des Backends. Um die Antworten des Backends auf die Anfragen des Frontends umzusetzen, müssen die zugehörigen Requests und Callback-Funktionen im Backend erstellt werden. Um die Webanwendung später als eine PWA installieren, das heißt sie offline verwenden und Daten speichern zu können, müssen im Anschluss Service Workers und Web App Manifests eingebunden werden.

Nach Fertigstellen der App soll diese auf verschiedenen Geräten und in verschiedenen Browsern installiert und bewertet werden. Ein kritischer Blick muss vor allem auf die Offline-Funktionalitäten, das Caching von Daten und die Push-Nachrichten geworfen werden.

# **Teil I.**

# **Theoretische Grundlagen**

## 2. Architektur

Die grundlegende Architektur einer Webanwendung ist in Abbildung 2.1 beschrieben. Man spricht von einer Client-Server-Architektur. Die Nutzer einer Webanwendung interagieren mit einem Browser, dem Client. Innerhalb des Browsers wird die Webseite - das Frontend (Kapitel 3) - dargestellt. Die Inhalte einer Webseite werden mithilfe von Hyper Text Markup Language (**HTML**) (siehe Abschnitt .1) und das Layout mithilfe von Cascading Style Sheets (**CSS**) (siehe Abschnitt .2) definiert. Auf Nutzerinteraktionen kann dynamisch unter Verwendung von JavaScript reagiert werden. Ein Browser kann somit die drei Skript-Sprachen HTML, CSS und JavaScript interpretieren.

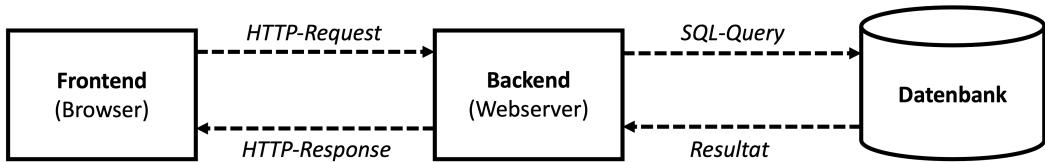


Abbildung 2.1.: Architektur der gesamten Anwendung

Die Webseiten liegen auf einem Webserver - dem Server - oder werden von einem Webserver bereitgestellt. In den Browser wird eine Uniform Resource Locator (**URL**) eingegeben, welche die Adresse eines Webservers und den Namen der sich darauf befindlichen Webseite enthält. Die Kommunikation zwischen Browser und Webserver erfolgt mittels Hypertext Transfer Protocol (**HTTP**). Die Eingabe einer URL in den Browser bewirkt einen sogenannten *HTTP-Request* an den Webserver. Typischerweise wird eine GET-Anfrage an den Webserver gestellt, um diese zu laden. Der Webserver beantwortet diesen Request mit einer *Response*, indem er die angefragte Seite an den Browser sendet. Diese wird im Browser dargestellt. Wird innerhalb der Webseite auf einen Hyperlink geklickt, entspricht das in der Regel einer weiteren Anfrage an den (oder einen anderen) Webserver und eine neue Seite wird übermittelt und dargestellt.

Es kann jedoch sein, dass der Webserver die angefragte Webseite noch nicht vollständig (*statisch*) bereitstellen kann. Eine solche Webseite muss erst auf dem Webserver *dynamisch* zusammengestellt werden. Dies ist z.B. der Fall, wenn Suchanfragen durch den Nutzer gestellt und die Ergebnisse der Suche zunächst aus einer Datenbank extrahiert und im Anschluss in eine Webseite eingebunden werden. Der Webserver kommuniziert in einem solchen Fall mit der an den Webserver angebundenen Datenbank mittels Structured Query Language (**SQL**). Insbesondere, wenn die an den Browser zu übertragende Webseite erst

durch den Webserver „zusammengebaut“ werden muss, wird der Webserver auch als *Backend* betitelt.

Die Kommunikation zwischen Browser und Webserver beinhaltet jedoch nicht nur GET-Anfragen, mit denen der Browser eine Ressource (Webseite) von dem Webserver anfordert. Darüber hinaus kann die Kommunikation auch das Senden von Daten an den Webserver beinhalten. Dies ist beispielsweise der Fall, wenn eine Webseite ein Formular enthält, in das Daten eingeben werden können. Diese Daten sollen entweder in die Datenbank gespeichert oder aber als neue Daten zur Aktualisierung bereits vorhandener Daten in einer Datenbank verwendet werden. Neben den GET-Anfragen sind demnach im HTTP-Standard auch sogenannte POST-, zum Senden neuer Daten, PUT-, zum Aktualisieren von Daten und DELETE-Anfragen, zum Löschen von Daten vorgesehen. Betrachtet man die Möglichkeiten zur Manipulation einer Datenbank, dann gibt es vier verschiedene Operationen, die auf einer Datenbank möglich sind:

- *Create* : das Hinzufügen neuer Daten(sätze),
- *Read* : das Lesen einer oder mehrerer Daten(sätze),
- *Update* : das Aktualisieren einer oder mehrerer Daten(sätze) sowie
- *Delete* : das Löschen einer oder mehrerer Daten(sätze).

Diese vier Operationen werden gerne unter dem Begriff *create, read, update and delete (CRUD)* zusammengefasst. Die oben genannten HTTP-Anfragen lassen sich somit gut auf diese CRUD-Operationen abbilden:

- GET-Anfrage für das Lesen (read),
- POST-Anfrage für das Erstellen (create),
- PUT-Anfrage für das Aktualisieren (update) und
- DELETE-Anfrage für das Löschen (delete)

einer Ressource (Daten). Dieses „Mapping“ bildet die Grundlage für eine Representational State Transfer (**REST**)-Schnittstelle. REST stellt eine Beschreibung von konkreten HTTP-Anfragen an konkrete Ressourcen auf dem Webserver (oder der Datenbank) dar. Es verbindet eine HTTP-Anfrage mit einer Ressource und beschreibt somit eindeutig, was mit dieser Ressource geschehen soll. In der vorliegenden Arbeit wird eine REST-Schnittstelle mithilfe von *Node.js* implementiert, siehe Abschnitt 4. Mithilfe der Schnittstellen werden Daten der Not only SQL (**NoSQL**)-Datenbank names *MonogDB* manipuliert, siehe Abschnitt 5.

# 3. Angular

Für die Implementierung der Webanwendung wird ein Framework verwendet. Ein Framework ist ein Programmiergerüst, das verwendet werden kann, um modulare, skalierbare und gut wartbare Applikationen zu entwickeln. In der Webentwicklung ist Angular neben React.js und Vue.js eines der beliebtesten Frameworks [sta21].

Mit Angular werden komponentenbasierte Single Page Applikation (**SPA**) erstellt. Bei einer *Single Page Application* wird immer nur eine Seite im Browser geladen. Der Inhalt dieser Seite ändert sich je nach Nutzerinteraktion. Dies hat unter anderem einen performanten Vorteil, da nur die benötigten Inhalte berechnet werden müssen. Mithilfe des Frameworks können sehr große Webanwendung entwickelt werden. Um die Übersichtlichkeit zu erhöhen, werden die Funktionen der Anwendung in Komponenten aufgeteilt. Die Komponenten sind die Grundbausteine einer Angular-Anwendung.

Eine Komponente setzt sich aus einem *Template* und einer *TypeScript-Klasse* zusammen. Das Template ist für die Darstellung von Inhalten verantwortlich und besteht aus einer HTML-Datei. Die TypeScript-Klasse verwaltet und manipuliert die Daten, die im Template angezeigt werden. TypeScript ist eine Obermenge von JavaScript und unterstützt eine typsichere und objektorientierte Programmierung [Typ21].

Neben dem Template und der TypeScript-Klasse verfügt eine Komponente über eine Datei, um CSS-Eigenschaften zu deklarieren. Bei der zu erstellenden Angular-Anwendung handelt es sich um *SCSS-Dateien*. SCSS ist eine Stylesheet-Sprache, welche die Funktionalitäten von CSS erweitert [Sas21].

Angular bietet zusätzliche Funktionen, die den Entwickler bei der Implementierung von Webanwendung unterstützen sollen. Einige dieser Funktionen werden im Laufe der Arbeit verwendet und demnach im Folgenden erläutert.

## 3.1. Strukturdirektiven

*Strukturdirektiven* erweitern die Funktionalität von HTML-Elementen. Sie werden im Template verwendet und sind durch einen voranstehenden Stern \* markiert. Die **\*ngIf**-Direktive ist ein Vertreter der Strukturdirektiven. Die Direktive wird im HTML-Tag angegeben und somit diesem HTML-Element zugeordnet. Dieses HTML-Element kann

durch die Direktive ein- bzw. ausgeblendet werden. Dafür muss der Direktive ein `boolean` zugewiesen werden, dessen Wahrheitswert darüber bestimmt [MHK20][Ang21b]. Weitere Strukturdirektiven sind unter anderem `*ngFor` und `*ngSwitchCase`. Dabei erlaubt die Strukturdirektive `*ngFor` ein wiederholtes Einfügen eines HTML-Elementes in den HTML-Code, während die Direktive `*ngSwitchCase` - ähnlich wie `*ngIf` - eine Alternative formuliert.

## 3.2. Interpolation

Durch *Interpolation* können Daten (Werte) aus der TypeScript-Klasse in das Template eingebunden werden. Dies geschieht syntaktisch durch zwei geschweifte Klammern. Die Klammern umschließen die Variable aus der TypeScript-Klasse, siehe Listing 3.1. Dadurch wird der Wert der Variablen in der Webanwendung angezeigt [MHK20][Ang21d].

```
1 <p>{{Variable}}</p>
```

Listing 3.1: Interpolation im Template

## 3.3. Property Binding

Mithilfe von *Property Bindings* können variable Werte aus der TypeScript-Klasse als HTML-Attribut einem HTML-Element zugeordnet werden. Diese Eigenschaft kann z.B. dazu verwendet werden, das `href`-Attribut eines Links zu ändern [MHK20][Ang21c]. Syntaktisch wird bei einem Property Binding die entsprechende Eigenschaft (Property) von eckigen Klammern umschlossen und mit einem Wert in Hochkomma durch das Gleichheitszeichen verknüpft, siehe Listing 3.2.

```
1 <a [href]="Variable"> Link </a>
```

Listing 3.2: Property Binding

## 3.4. Event Binding

Mit *Event Bindings* kann auf Ereignisse im Template reagiert werden. Mögliche Ereignisse sind das Betätigen (engl. `click`) eines Knopfes (engl. `Button`) oder das Betätigen einer bestimmten Eingabetaste (engl. `Key`). Diese Ereignisse können mit einer Funktion in

der TypeScript-Klasse verknüpft werden. Somit stellen Event Bindings den Datenfluss von dem Template zu der TypeScript-Klasse dar. Dies macht sie zum Gegenpart der Property Bindings. Im Listing 3.3 wird gezeigt, wie bei dem Betätigen des Buttons die `clickFunktion()` in der TypeScript-Klasse aufgerufen wird [MHK20][Ang21a].

```
1 <button (click)="clickFunktion()"> Click me </button>
```

Listing 3.3: Event Binding

## 3.5. Backend Anbindung

Die Anbindung an das Backend wird in Angular typischerweise in einem *Service* implementiert. Ein *Service* in Angular ist eine TypeScript-Klasse, die einem konkreten Zweck dient. Ein Service sollte möglichst genau eine Sache erledigen. Ein Service kann typischerweise von allen beziehungsweise vielen Komponenten verwendet werden. In einen solchen Service, der die Anbindung an das Backend implementiert, muss in Angular der `HttpClient`-Service per *dependency injection* injiziert werden. Dieser Service wird durch das Modul `HttpClientModule` bereitgestellt, welches in die `app.module.ts` importiert werden muss. Der `HttpClient`-Service stellt die Funktionen `get()`, `put()`, `post()` und `delete()` in Äquivalenz zu den entsprechenden HTTP-Anfragen bzw. den REST-Endpunkten bereit.

# 4. Node.js

## 4.1. Überblick

Node.js ist eine hochskalierbare Low-Level-Plattform. Unter Verwendung einer JavaScript-Laufzeitumgebung können Netzwerkanwendungen aufgebaut werden [Per16, S.1]. Node.js verwendet JavaScript als serverseitige Programmiersprache [Per16, S.3] und arbeitet asynchron und ereignisgesteuert [Nod22]. Außerdem setzt es standardmäßig Betriebssystem-Threads nie gleichzeitig ein, sondern arbeitet zu jedem Zeitpunkt mit nur einem Haupt-Thread. Dies bedeutet für Projekte, dass es keine Gefahr der Entstehung von Deadlocks gibt, da sich die Threads nicht gegenseitig blockieren können [Nod22]. Zudem beinhaltet Node.js Frameworks, mit deren Einsatz eine Echtzeit-Kommunikation von Client und Server möglich ist [Per16, S.3].

## 4.2. Charakteristika

### 4.2.1. Single-Thread-Architektur

Eine Charakteristik von Node.js ist die Single-Thread-Architektur. Jeder initiierte Prozess erhält eine Haupt-Thread-Instanz. Ein Prozess kann somit nicht mit einer Mehrzahl von Threads arbeiten, die sich gegenseitig blockieren könnten. Hierbei wird die asynchrone Verarbeitung deutlich. Jene Funktionen von Node.js, die als asynchron gelten, arbeiten als nichtblockierende Ein- und Ausgaben. Sollte beispielsweise eine Anwendung eine Datei lesen, wird in diesem Zusammenhang die CPU nicht blockiert. Dadurch kann diese Anwendung gleichzeitig andere Aufgaben bearbeiten und ausführen, die von anderen Nutzern angefordert werden [Per16, S.1].

### 4.2.2. Ereignisschleife

Node.js arbeitet ereignisorientiert. Muss in einem Moment keine Aufgabe ausgeführt werden, schlafst die JavaScript-Laufzeitumgebung und wartet auf ein neues Ereignis. Dabei arbeitet Node.js nicht mit Ereignissen in Form von Mausklicks oder Tastaturanschlägen, sondern mit Datenbankverbindungen oder dem Öffnen von Dateien. Für diesen Prozess ist die

Ereignisschleife verantwortlich, eine Endlosschleife, die in jeder Iteration das Vorhandensein von neuen Ereignissen prüft. Sie muss erkennen, wann eine neue Aufgabe ausgeführt werden muss. Sobald ein Ereignis ausgelöst wird, führt die Schleife die zugehörige Aufgabe. Währenddessen kann jeder beliebige Logik in die Callback-Funktion schreiben [Per16, S.3]. Neben der Ereignisschleife gibt es den Worker Pool, der sich um den restlichen Workload kümmert. Während die Ereignisschleife außer für Callback-Funktionen nur für Anfragen verantwortlich ist, die nicht blockieren können, laufen innerhalb des Worker Pools Aufgaben, die nicht in jedem Fall nicht-blockierend sind. (@misnode.js, title=Don't block the event loop (or the worker pool), url=https://nodejs.org/en/docs/guides/dont-block-the-event-loop/, journal=Node.js, author=Node.js )

## 4.3. Vorteile durch die Nutzung von JavaScript

Durch die Verwendung von JavaScript als serverseitige Programmiersprache ergeben sich einige Vorteile. Zum einen lässt sich das erstellte Projekt sehr leicht pflegen, sollte man die Sprache beherrschen. Die Sprache auf Server-Seite ist keine andere als diejenige auf Client-Seite. Dies hat den Vorteil, dass man auf die Verwendung serverseitiger Betriebssystem-Sprachen verzichten kann. Zudem benötigt Node.js keine zusätzlichen Frameworks, um JSON-Objekte parsen zu können. Gerade für Projekte, in deren Rahmen eine Verbindung zu Datenbanken wie beispielsweise MongoDB aufgebaut werden soll, ist dies ein hervorzuhebendes Argument, da diese Daten in Form von JSON-Objekten speichern [Per16, S.3].

## 4.4. Node Package Manager

Node.js verfügt über einen Package Manager – Kurzform NPM – mit dessen Hilfe Node-Module leicht verwaltet werden können. NPM beweist sich in der Verwendung als recht einfach und übernimmt beispielsweise die Verwaltung von Abhängigkeiten des Projekts oder das Installieren neuer Module [Per16, S.10].

### 4.4.1. NPM Aufgabenautomatisierung

Mit dem NPM können Aufgaben automatisch ausgeführt werden. Ergänzt man in der Deskriptordatei das Attribut `scripts`, kann man verschiedenen Befehlen einen kürzeren Namen geben und sie damit ausführen. Beispielsweise kann der Befehl `npm run node app.js` auf `npm run start` gekürzt werden [Per16, S.12].

## 4.5. Deskriptordatei package.json

Alle Projekte, die mithilfe von Node.js erstellt werden, bezeichnet man als Module. Erstellt man ein Projekt, wird zeitgleich dazu eine Deskriptordatei der Module mitgeliefert, die den Namen `package.json` trägt. Diese Datei ist für ein Projekt von großer Bedeutung, da sie wichtige Schlüsselattribute enthält. Jene Attribute werden sowohl von Node.js als auch von dem Package Manager gelesen. Bei Unordentlichkeiten innerhalb der Datei können daraus Fehler in der Ausführung resultieren. Zu den Schlüsselattributen zählen beispielsweise `name`, `description`, `author` und `version`. Mit `name` als Hauptschlüssel wird der Name des erstellten Projekts bestimmt, über den das Projekt später aufgerufen werden kann, zum Beispiel bei der Verwendung von `npm run`. Mit der `description` kann eine genauere Erläuterung des Projekts folgen und unter `author` wird der Name des Autors des Projekts gespeichert. Das Attribut `version` spielt im Zusammenhang mit der Installation eine wichtige Rolle, da es empfohlen wird, bei dieser in jedem Fall eine Version anzugeben [Per16, S.10f.].

## 4.6. Node Express

Node Express ist ein flexibles, minimalistisches Node.js Web Framework, womit sich Webanwendungen sowie mobile Anwendung leicht entwickeln lassen. Mithilfe des Moduls ist es möglich, Anwendung in unterschiedlicher Größe und Komplexität zu erstellen [Per16, S.16]. Express macht es außerdem einfach, durch die Vielzahl der verwendbaren http-Methoden und die zur Verfügung stehende Middlewares eine robuste API zu bauen [Exp22c]. Mit folgendem Befehl lässt sich das Express Modul installieren, siehe Listing 4.1.

```
1 $ npm install express -save
```

Listing 4.1: Express Modul installieren

### 4.6.1. Vorteile

Zu den Vorteilen von Node Express gehört das Routing. Da viele http-Request-Methoden wie GET, POST, PUT etc. zur Verfügung stehen und eine Großzahl an Middleware in der Lage ist, auf diese Anfragen zu reagieren, lässt sich das Routing als robust bezeichnen. Außerdem benötigt man für die Verwendung des Moduls nur minimalistischen Code. Express beinhaltet keine unflexiblen Konventionen, was Entwicklern die Möglichkeit gibt, ihren Code frei zu gestalten. Zudem können Entwickler Standards und Best Practices

von REST in ihren Code einbeziehen und damit bereits vorhandenes Wissen in ihren Anwendungen umsetzen [Per16, S.16f.].

## 4.6.2. Routing

Beim Routing wird festgelegt, wie eine Anwendung auf eine Anforderung des Clients an einen bestimmten Endpunkt reagieren soll. Dieser ist durch einen URI oder einen Pfad sowie eine http-Request-Methode dargestellt [Exp22a]. Einfaches Routing hat folgende Struktur, siehe Listing 4.2.

```
1 app .METHOD(PATH ,  HANDLER)
```

Listing 4.2: Routing in Express

App ist hierbei eine Instanz von Express. METHOD bezieht sich auf die http-Request-Methoden, welche in Kleinbuchstaben geschrieben werden müssen. PATH ist der Pfad des Endpunkts auf dem Server und HANDLER ist eine Funktion, die ausgeführt wird, sollte die Route zutreffen. Diese wird entweder als Handler- oder als Callback-Funktion bezeichnet [Exp22a]. Die Anwendung wartet auf Anfragen des Clients. Bei Übereinstimmung dieser Anfrage mit der Route und der Request-Methode, wird die zugeordnete Handler-Funktion ausgeführt [Exp22d]. Jede Route kann dabei mehrere Handler-Funktionen haben, die je nach Anfrage ausgeführt werden [Exp22a].

Außerdem gibt es die Möglichkeit, dass die Argumente einer Routing-Funktion mehrere Callback-Funktionen enthalten, die bei Zutreffen der Anfrage nacheinander ausgeführt werden. Um dies umsetzen zu können, muss jedoch das zusätzliche Argument „next“ an die Callback-Funktion übergeben und in dieser mit next() aufgerufen werden. Nur unter Verwendung dieses Befehls kann die nächste Callback-Funktion ausgeführt werden [Exp22d]. Ansonsten bleibt die Anfrage in dieser Funktion hängen und wird nicht weiterbearbeitet [Exp22e].

Beispielsweise könnte eine Routing-Funktion mit zwei Callback-Funktionen folgendermaßen aussehen, siehe Listing 4.3

```
1 app.get('/example' ,  (req ,  res ,  next) => {
2     console.log('this is the first callback function')
3     next()
4 }, (req ,  res) => {
5     res.send('this is the secound callback function')
6});
```

Listing 4.3: Routing-Funktion mit zwei Callback-Funktionen

In obigem Beispiel kann man die Response-Methode `res.send()` sehen. Damit lässt sich eine Antwort verschiedener Typen senden. Neben dieser gibt es weitere Methoden, die in Callback-Funktionen verwendet werden können. Beispielsweise lässt sich mit `res.json()` eine Antwort in JSON-Format und mit `res.sendStatus()` der Status-Code der Antwort senden. Der Status-Code bezieht sich auf http-Codes wie beispielsweise Code 200 OK und Code 400 Bad Request.

Des Weiteren gibt es die Möglichkeit verkettete Routen-Handler zu erstellen unter der Verwendung von `app.route()`. Dabei wird der Pfad an einem Punkt definiert. Im Anschluss lassen sich die Callback-Funktionen für verschiedene Request-Methoden spezifizieren. Das definieren modularer Routen hat den Vorteil, dass sich Redundanzen vermeiden lassen und der Code minimalistisch gehalten werden kann. Beispielsweise kann eine modulare Routing-Methode so aussehen, siehe Listing 4.4

```

1  app.route('/user')
2    .get((req, res) => {
3      res.send('Get user')
4    })
5    .post((req, res) => {
6      res.send('Add user')
7    })
8    .patch((req, res) => {
9      res.send('Update user')
10 })

```

Listing 4.4: modulare Routing Methode [Exp22d]

### 4.6.3. Middleware

Als Middleware wird eine Funktion bezeichnet, die vor dem endgültigen Request-Handler aufgerufen wird. Sie steht zwischen einem rohem Request und der letztendlich verwendeten Route [Exp22b]. Eine Express-Anwendung ist eine Serie von Middleware-Funktionsaufrufen. Diese Funktionen haben Zugriff auf die Request- und Response-Objekte sowie die in der Serie nächste Middleware-Funktion. Middleware-Funktionen können verschiedene Aufgaben ausführen. Dazu gehört die Ausführung jedes beliebigen Codes und die Möglichkeit, Änderungen an Request- und Response-Objekten vornehmen zu können. Außerdem kann eine Funktion die Nächste aufrufen unter Verwendung von `next()`. Wird jener Befehl nicht aufgerufen, kann die Kontrolle nicht an die nächste Middleware-Funktion übergeben werden. Dies bedeutet, dass der Request-Response-Zyklus der Anwendung beendet wird. Insgesamt gibt es fünf verschiedene Arten von Middleware, die Express verwenden kann, die im folgenden beschrieben werden.

## Application-Level-Middleware

Application-Level-Middleware wird im Zusammenhang mit Instanzen eines App-Objekts genutzt. Durch die Verwendung von `app.use()` oder `app.METHOD()` kann diese Art der Middleware verwendet werden. Bisher gezeigte Beispiele sind ebenfalls Beispiele für die Verwendung von Application-Level-Middleware.

## Router-Level-Middleware

Dieser Typ von Middleware funktioniert generell gleich wie der Application-Level. Der Unterschied beider Typen liegt in der Verwendung der Instanz. Während der Application-Level an eine Instanz eines App-Objekts gebunden ist, verwendet der Router-Level eine Instanz von `express.Router()`. Nachdem man das Router-Objekt definiert, kann die Middleware ebenfalls mit `router.use()` und `router.METHOD()` verwendet werden.

Die Instanz wird folgendermaßen verwendet, siehe Listing 4.5

```
1 const router = express.Router()
```

Listing 4.5: Verwendung der Instanz

## Error-Handling-Middleware

Ebenfalls lässt sich mit Middleware-Funktionen ein Error-Handling durchführen. Wichtig dabei ist, dass immer vier Argumente angeboten werden müssen, damit eine Middleware als Error-Handling-Middleware erkannt wird. Diese vier Argumente sind “err, req, res, next,“. Eine Error-Handling-Funktion könnte folgendermaßen aussehen, siehe Listing 4.6

```
1 app.use((err, req, res, next) => {
2   console.error(err)
3   res.status(400).send('Bad Request!')
4 })
```

Listing 4.6: Error-Handling-Middleware

## Built-in Middleware

Express verfügt über drei integrierte, vordefinierte Middleware-Funktionen. Diese sind:

- **Express.static**: dient statischen Inhalten wie beispielsweise Bildern und HTML-Dateien
- **Express.json**: parst Anfragen, die JSON-Payloads enthalten
- **Express.urlencoded**: parst Anfragen, die URL-kodierte Daten enthalten

### Third-party Middleware

In Express gibt es die Möglichkeit, Module von Drittanbietern zu integrieren. Dazu muss das gewünschte Node.js-Modul installiert und später in die Anwendung entweder auf Application- oder Router-Level geladen werden. Beispielsweise gibt es eine Drittanbieter-Middleware für das Parsen von Cookies. Diese könnte folgendermaßen geladen werden, siehe Listing 4.7.

```
1 const express = require('express')
2 const app = express()
3 const cookieParser = require('cookie-parser')
4 app.use(cookieParser())
```

Listing 4.7: Module von Drittanbietern integrieren [Exp22e]

# 5. Datenbanken

Um Daten zu speichern, gibt es verschiedene Lösungsansätze. Generell unterscheidet man zwischen den SQL- und NoSQL-Datenbankansätzen. Die Abkürzung „SQL“ steht dabei für die „Sequel Query Language“, die zur Abfrage von relationalen Datenbanken genutzt wird. Im Gegensatz dazu umfasst der Begriff „NoSQL“ – ausgeschrieben „Not only SQL“ – jene Datenbanksysteme, die nicht den relationalen Ansätzen entsprechen [Mei18].

## 5.1. SQL

Spricht man von SQL-Datenbankansätzen, ist die Rede von Relationalen-Datenbank-Management-Systemen (RDBMS). Innerhalb eines RDBMS werden die Daten strukturiert in Tabellen gespeichert, wobei jede Spalte über einen spezifischen Typ verfügt. Systeme relationaler Datenbanken basieren auf dem Konzept sogenannter ACID-Transaktionen.

### 5.1.1. ACID

ACID ist die Abkürzung vierer Worte, die den Aufbau und die Funktionsweise der RDBMS beschreiben: atomar, konsistent, isoliert und dauerhaft (eng.: Atomic, Consistent, Isolated, Durable). Unter atomar versteht man die Aussage, dass entweder alle Änderungen, die während einer Transaktion vorgenommen werden, übernommen werden oder keine dieser Änderungen. Nachdem die Transaktion durchgeführt wurde, befinden sich die Daten in einem konsistenten Zustand. Damit erhalten alle Abfragen, die bestimmte Daten abrufen möchten, dasselbe aktualisierte Ergebnis. Die Transaktionen, die an einem Datensatz vorgenommen werden, sind außerdem isoliert und damit unabhängig von weiteren Transaktionen. Zuletzt sind Änderungen, die innerhalb des Systems passieren, permanent und gehen im Falle eines Versagens desjenigen Systems nicht verloren.

## 5.2. NoSQL

NoSQL-Datenbankansätze basieren nicht auf dem Konzept der ACID-Transaktionen. Ihre Charakteristik liegt in der Speicherung von semi- bis hin zu unstrukturierten Daten. Dadurch verfügen sie über ein flexibles Datenmodell und werden gerne für große Datenmengen

im Internet verwendet [ES15, S.13]. Anstelle des ACID-Konzepts liegt die Grundlage der nichtrelationalen Datenbanksysteme in dem BASE-Konzept. Um dieses jedoch verstehen zu können, muss zuerst ein Überblick über das CAP-Theorem verschafft werden. Das Kürzel CAP steht für die Bedingungen Konsistenz, Verfügbarkeit und Teilungstoleranz (eng: Consistency, Availability, Partition Tolerance). Die Konsistenz bedeutet auch hier den einheitlichen Zustand der Daten nach Durchführung einer Transaktion. Unter der Verfügbarkeit versteht man die durchgängige Erreichbarkeit des Systems. Die Teilungstoleranz sagt aus, dass das System auch weiterfunktionieren kann, sollte es auf Gruppen von Servern aufgeteilt werden, die nicht in der Lage sind, miteinander zu kommunizieren, siehe Abbildung 5.1.

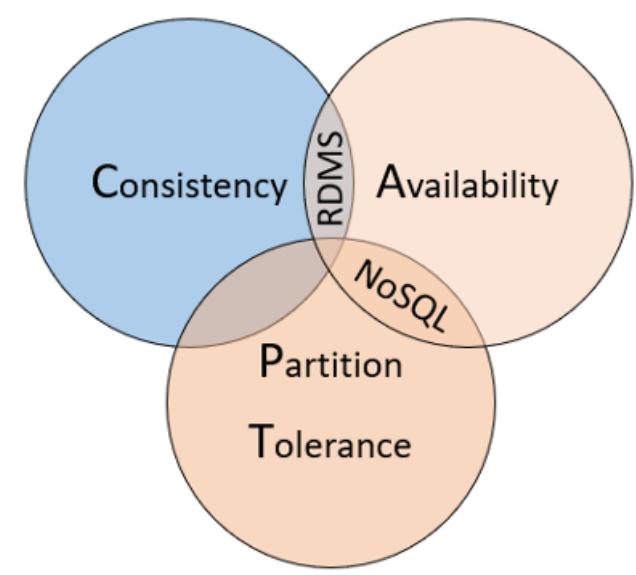


Abbildung 5.1.: CAP-Theorem [Vet+22]

Laut dem Theorem können jedoch immer nur zwei der drei Bedingungen gleichzeitig erfüllt werden [ES15, S.15f]. NoSQL möchte vor allem immer verfügbar sein. Im Notfall verzichtet eine NoSQL-Datenbank zuerst auf die Konsistenz der Daten, bevor die Verfügbarkeit eingeschränkt wird. SQL-Datenbanken dagegen stellen die Konsistenz in den Vordergrund. Bevor diese verloren geht, wird auf die Verfügbarkeit verzichtet.

### 5.2.1. Modelle

NoSQL umfasst im Allgemeinen vier verschiedene Modelle, die einen möglichen Aufbau einer NoSQL-Datenbank zeigen, siehe Abbildung ??.

Auf dem ersten Bild erkennt man den Dokumentspeicher. In diesem Modell werden Daten in Dokumenten hierarchisch in einer Datenbank angeordnet und abgespeichert. Diese Dokumente sind JSON-basiert, was heißt, dass die dort abgespeicherten Daten im

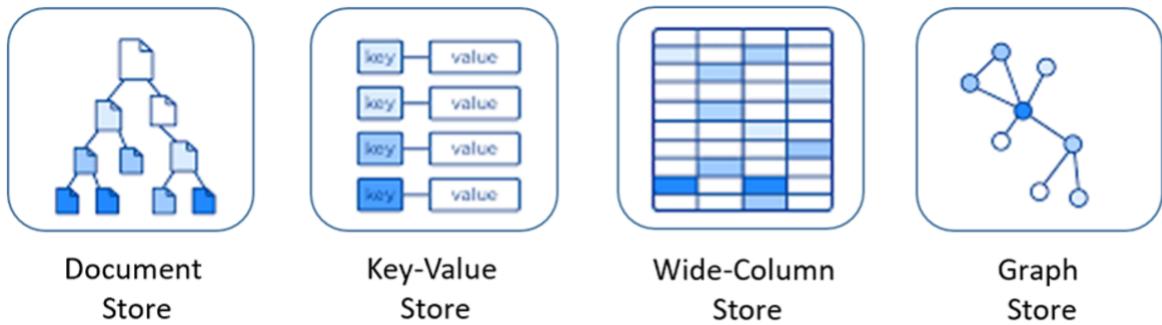


Abbildung 5.2.: Modelle einer NoSQL Datenbank [Vet+22]

JSON-Format vorliegen. Eine zweite Möglichkeit, Daten abzuspeichern, ist ein Schlüssel-Wert-Speicher, was die einfachste Speicherform ist. Dabei werden Daten in Form von Schlüssel-Wert-Paaren abgespeichert. Dies ist die einzige Ordnung, über die Datenbanken, die dieses Modell verwenden, verfügen. Das dritte Speicher-Modell ist der Wide-Column Store. Für diese Form der Speicherung werden die Schlüssel-Wert-Paare verknüpfter Daten gruppiert. In diesen entstandenen Gruppen werden sie in einer einzelnen Spalte einer Tabelle abgespeichert. Die letzte Speicherform ist der Graphspeicher. Hierbei werden Daten in Form von Knoten, Kanten und Dateneigenschaften in einer Graphstruktur gespeichert [Vet+22].

### 5.3. SQL vs NoSQL

Einen großen Unterschied der beiden Ansätze sieht man in der Flexibilität der Schemata. RDBMS weisen nur eine geringe Flexibilität in ihrem Design auf. Sie sind zeilenorientiert, sodass schon das Hinzufügen einer Spalte zu einer Tabelle zu Problemen führen kann, vor allem, wenn die Tabelle bereits mit Daten gefüllt sein sollte. Oft kann ein solches Hinzufügen dazu führen, dass neue Tabellen erstellt werden. Die Komplexität des Systems wird dadurch erhöht, da es Beziehungen zwischen den Tabellen geben kann. Spaltenorientierte Datenbanken haben den Vorteil, dass es möglich ist – falls nötig – Spalten ohne Probleme hinzuzufügen. Auch Dokumente mit semistrukturierten Daten schaffen keine größeren Komplexitäten und sind in der Wahl des Schemas sehr flexibel [ES15, S.19]. Dadurch ist die Arbeit mit allen möglichen Datentypen möglich. Änderungen, die eine Aktualisierung des Datenbankschemas bedeuten, können somit ohne weiter Probleme umgesetzt werden [ES15, S.18]. Einen weiteren Vergleichspunkt findet man in der Verwendung komplexer Abfragen. Durch den standardisierten Aufbau von RDBMS entstehen komplexe JOIN-Abfragen, die sowohl schwer zu implementieren und zu pflegen sind als auch erhebliche Ressourcen für die Ausführung benötigen. NoSQL-Datenbanken verfügen weder über Beziehungen noch

Fremdschlüssel, wodurch keine komplexen Abfragen entstehen. Jedoch müssen mehrere Abfragen ausgeführt werden, um über eine Tabelle hinaus abfragen zu können [ES15, S19f.]. Die einfachen Datenmodelle führen dennoch zu einer einfachen Verwaltbarkeit [ES15, S.18]. Ebenfalls wichtig zu betrachten ist die Skalierbarkeit. Hierbei gilt es, die beiden Ansätze von Scale-up und Scale-out zu erläutern. SQL-Datenbanken nutzen den Scale-up-Ansatz. Dabei erfolgt die Erweiterung vertikal mithilfe von kostspieligen, qualitätsvollen Servern. Dieser Ansatz versagt jedoch, sobald die Transaktionsraten und Anforderungen an schnelle Reaktionen steigen. In Momenten wie diesen kommt der Scale-out-Ansatz zum Einsatz. Dabei findet eine horizontale Erweiterung unter Verwendung von günstigen Commodity-Servern statt [ES15, S.18]. Diesen Ansatz nutzen NoSQL-Datenbanken, wodurch sie eine bessere Skalierbarkeit bieten, als auch Daten preiswert speichern und verarbeiten können [ES15, S.19f.]. Ein letzter wichtiger Vergleichspunkt ist die Synchronisation der Daten. Das Aktualisieren der Daten über die Tabellen der RDBMS hinweg erweist sich als sehr komplex. Die Updates müssen dabei auf die verschiedenen Knoten des Systems verteilt werden. Unterstützen Systeme ein gleichzeitiges Schreiben auf verschiedenen Knoten nicht, kann es dabei zu Fehlern und zu höheren Latenzzeiten kommen. Um diese Latenzzeiten gering zu halten, bieten dagegen NoSQL-Anwendungen gute Synchronisierungsmöglichkeiten. Beispielsweise MongoDB kann eine Aktualisierung gleichzeitig auf mehreren Knoten ausführen und die Konsistenz innerhalb des Systems in einer akzeptablen Latenzzeit sicherstellen [ES15, S.19f.]. Für die Arbeit wird mit der Verwendung der Datenbank MongoDB ein NoSQL-Ansatz gewählt, da die Vorteile vor allem in Bezug auf die Flexibilität der Schemata überwiegen. Ebenso andere Begründungen zur Nutzung von NoSQL-Ansätzen, die in diesem Kapitel beschrieben werden, sind für die Arbeit von Vorteil [ES15, S.17-20].

# 6. MongoDB

## 6.1. Überblick

MongoDB ist eine NoSQL-Datenbank, die über einen JSON-basierten Dokumentenspeicher verfügt. Abfragen in MongoDB basieren auf den Schlüsseln innerhalb der Dokumente. Jene Dokumente können auf verschiedenen Servern verteilt sein. Führt man eine Abfrage aus, sucht in seinen Dokumenten nach Resultaten, die er als Ergebnis ausgibt. Dadurch erreicht man eine lineare Skalierbarkeit und eine verbesserte Performance. Außerdem verfügt MongoDB über eine Primär-Sekundär-Replikation, wobei die Primärseite die Schreibanforderungen annimmt. Will man die Schreibleistung verbessern, kann man das sogenannte Sharding einsetzen. Dabei werden die Daten auf mehrere Rechner verteilt, die jeweils unterschiedliche Teile der Datensätze aktualisieren können. In MongoDB wird Sharding automatisch ausgeführt. Fügt man einen neuen Rechner hinzu, werden die Daten automatisch verteilt [ES15, S.26].

## 6.2. Speicherung der Daten

### 6.2.1. JSON-basierte Speicherung

Eine Datenbank enthält Kollektionen, die Dokumente umfassen, siehe Abbildung ???. Diese Dokumente enthalten die zu speichernden Daten [ES15, S.30] in Form von Schlüssel-Wert-Paaren [ES15, S.27]. Zur Speicherung wird Binary JSON – kurz BSON – eingesetzt. JSON ist die JavaScript Object Notation und gilt als einer der Standardsprachen für den Datenaustausch im heutigen, modernen Web. Das Format von JSON ist sowohl für Maschinen als auch für den Menschen lesbar. Zudem werden alle Standarddatentypen wie beispielsweise Strings, Zahlenwerte, Boolean-Werte und Arrays unterstützt und können verwendet werden [ES15, S.31]. Ein Vorteil von JSON ist die Möglichkeit, zusammengehörige Daten gruppiert an einem Platz zu speichern. Dadurch sind gewisse Daten einfacher auffindbar, was die Performance der Abfragen verbessert. Außerdem ist es möglich, schemalose Modelle zu verwenden, wodurch Änderungen an einem Schema nie zu Problemen führen können. Im Gegenteil zu RDBMS kann für eine Speicherung ein dynamisches Schema gewählt werden. Dies bedeutet, dass Dokumente über unterschiedliche Strukturen oder Felder verfügen oder aber auch ähnliche Felder unterschiedliche Daten speichern können [ES15, S.26f.]. In

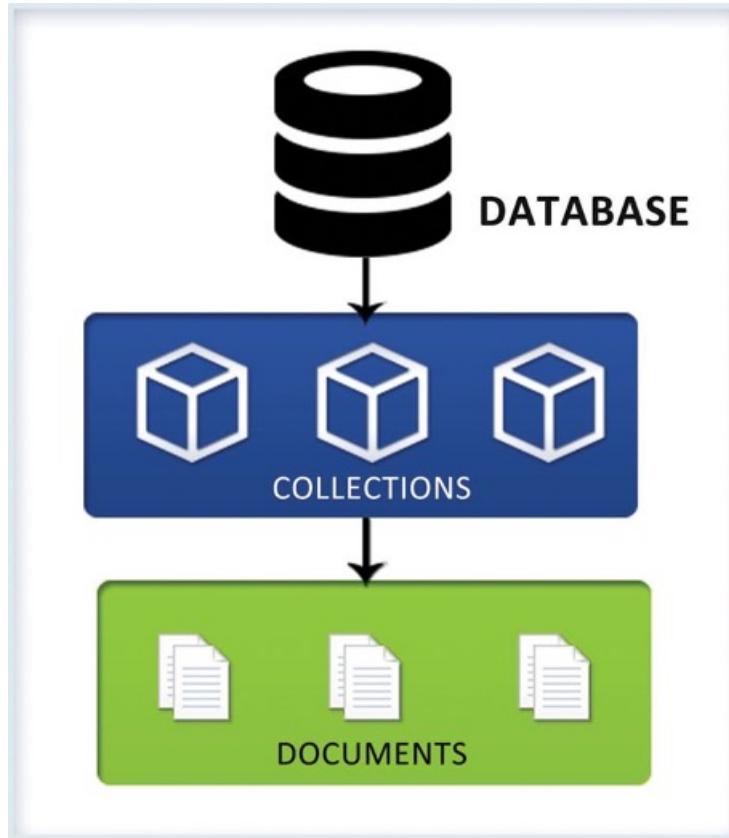


Abbildung 6.1.: Aufbau von MongoDB [ES15, S.30]

Fällen, in denen eine Kollektion Dokumente unterschiedlicher Typen beinhaltet, spricht man von einem polymorphen Schema [ES15, S.32].

### 6.2.2. BSON

Ein JSON-Dokument enthält die tatsächlichen Daten. Dennoch werden jene Daten nicht in dieser Form abgespeichert, sondern zuvor binär verschlüsselt. Die Implementierung eines BSON-Dokuments in MongoDB ist schnell und leichtgewichtig. Außerdem ermöglicht es MongoDB, Indizes in Objekten zu erstellen und diese mit den Abfragen abzugleichen [ES15, S.31].

### 6.2.3. Identifier

Wie auch die Primärschlüssel in RDBMS werden Identifier in Dokumenten benötigt, um diese identifizieren zu können. Identifier werden mit dem Schlüssel `id` bezeichnet. Den Wert der ID kann man – sofern es gewollt ist – selbst zuordnen. Dafür kann jeder beliebige Datentyp verwendet werden, außer Arrays. Ordnet man selbst der ID keinen Wert zu, übernimmt dies MongoDB automatisch [ES15, S.32].

### 6.2.4. Dateiupload

MongoDB ist in der Lage, Daten bis zu einer Größe von 16Mb in Binärformat zu speichern. Um dennoch größere Dateien wie beispielsweise Bilder speichern zu können, müssen diese in *chunks* zerlegt und in dieser Form gespeichert werden. Dazu enthält MongoDB eine Spezifikation namens GridFS [Gri22]. Bilder sind FormData-Objects im `multipart/form-data`-Format. Um Objekte dieser Form einsetzen zu können, verwendet man eine Middleware, die als Multer bezeichnet wird [npm].

Zur gemeinsamen Verwendung von Multer und GridFS werden folgende drei Pakete benötigt:

- multer
- multer-gridfs-storage
- gridfs-stream

Um Daten zu speichern, verwendet GridFS zwei Collections, `fs.files` und `fs.chunks`. In der ersten Collection sind die Metadaten der Datei enthalten wie beispielsweise der Name oder die Größe. Die chunks-Collection enthält die tatsächliche Datei, die in Chunks der Größe von 255kB zerlegt wurde. Diese werden als durchnummerierte separate Dokumente abgespeichert und später der Reihe nach zusammengesetzt, möchte man die Datei wieder abrufen [Gan18].

# 7. Progressive Web App

Wie in Kapitel 1.3 erwähnt, verfügt eine PWA unter anderem über folgende Funktionen:

- Installierbar,
- Zugriff auf Geräteschnittstellen,
- Netzwerkunabhängig,
- Push-Notifications.

Im Folgenden werden die theoretischen Grundlagen erläutert, die benötigt werden, um diese Funktionalitäten zu realisieren.

## 7.1. Web-App-Manifest

*The web app manifest is a JSON file that defines how the PWA should be treated as an installed application, including the look and feel and basic behavior within the operating system [Dev22].*

Eine PWA kann auf ein Endgerät wie zum Beispiel ein Desktop oder Handy installiert werden. Um diese Funktion zu realisieren, müssen zusätzliche Informationen wie beispielsweise der Name und das Icon der installierten Applikation in einer Datei festgehalten werden.

Bei dieser Datei handelt es sich um das sogenannte Web-App-Manifest. Die Informationen sind im JavaScript Object Notation ([JSON](#))-Format<sup>1</sup> angegeben.

Ohne das Manifest ist die Applikation nicht installierbar, was die Datei zu einer zwingenden Voraussetzung für eine PWA macht. Das Manifest muss mindestens einen `name`-Schlüssel und einen `String`-Wert aufweisen. Neben dem Namen der Applikation kann ein Manifest über folgende Informationen verfügen:

---

<sup>1</sup> durch Komma getrennte Schlüssel-Wert-Paare

### 7.1.1. short\_name

Unter `short_name` kann ein kurzer Name der Applikation angegeben werden. Dieser Name wird verwendet, falls das Endgerät nicht über genügend Platz verfügt, um den originalen Namen anzuzeigen.

### 7.1.2. icons

Unter `icons` wird ein Array<sup>1</sup> mit Bildobjekten gespeichert. Ein Bildobjekt besteht aus einem Dateipfad, unter dem das anzuzeigende Bild gespeichert ist, einer Typbeschreibung des Bildes, zum Beispiel *png* oder *svg*, einer Informationen über die Auflösung des Bildes sowie optional einer Angabe, welchem Zweck das Bild dient.

Die gespeicherten Bilder werden als App-Icon auf dem Desktop oder Handy angezeigt.

### 7.1.3. start\_url

Die angegebene `start_url` ist jene [URL](#) die geöffnet wird, sobald der Nutzer das installierte Icon auswählt und damit die Applikation startet. Wird keine explizite Startadresse angegeben, so wird die URL verwendet, von der die PWA installiert wurde.

### 7.1.4. display

Bei Auswählen des installierten Icons wird die PWA in einem neuem Fenster geöffnet. Unter `display` kann angegeben werden, wie das Betriebssystem das Fenster darstellen soll. Es kann zwischen **Fullscreen**, **Standalone** und **Minimal User Interface** unterschieden werden. Der Unterschied zwischen den einzelnen Auswahlmöglichkeiten liegt bei den Navigationselementen, siehe Abbildung 7.1 und 7.2.

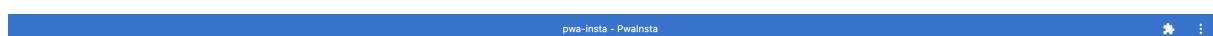


Abbildung 7.1.: display in „Standalone“-Einstellung

---

<sup>1</sup> Datentyp, das mehrere Werte speichern kann



Abbildung 7.2.: display in „Minimal User Interface“-Einstellung

### 7.1.5. theme\_color

Mit Hilfe dieser Einstellung kann die Farbe der oberen Navigationsleiste angepasst werden, wie in Abbildung 7.3 dargestellt ist. Hierbei ist jedoch darauf zu achten, dass die Applikation nicht den Meta-Tag `theme-color` definiert.



Abbildung 7.3.: Theme Color auf Weiß geändert

### 7.1.6. Debugging des Manifests

Neben den oben aufgezählten Grundeinstellungen sind viele weitere Optionen möglich. Um nachzuvollziehen, ob alle Einstellungen den Anforderungen entsprechen, kann das Manifest mithilfe der Browser-Entwicklerwerkzeuge untersucht werden. Unter Google Chrome kann unter dem Reiter *Application* das Manifest ausgewählt werden. Daraufhin erhält man folgende Ansicht, siehe Abbildung 7.4.

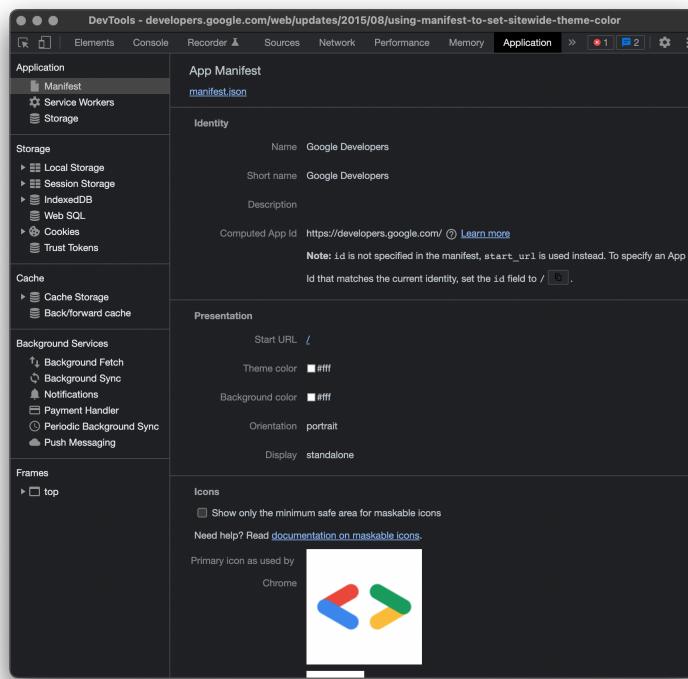


Abbildung 7.4.: Entwicklereinstellungen Web-App-Manifest

## 7.2. Service Workers

*Service workers are a fundamental part of a PWA. They enable fast loading (regardless of the network), offline access, push notifications, and other capabilities [Dev].*

Der *Service Worker* ist ein wichtiger Grundbaustein, um Funktionalitäten wie Push-Notifications und Hintergrund-Synchronisation zu realisieren sowie die Möglichkeit, die Anwendung auch offline auszuführen.

Bei einem *Service Worker* handelt es sich um ein Script, das im Hintergrund des Browsers - unabhängig von der Webanwendung - läuft [Gau]. Entstanden sind die Service Workers aus der Verwendung des Application Caches. Die Service Worker Applikation Programming Interface (API) wächst kontinuierlich und bietet zunehmend weitere Funktionalitäten.

Bei der Verwendung eines Service Workers sollten folgenden Eigenschaften berücksichtigt werden:

- Ein Service Worker kann zwar nicht direkt das Document Object Model (DOM) einer Seite manipulieren, kann aber auf Requests der Seite mit Responses reagieren. Die Seite selbst kann darunter ihr DOM ändern.
- Ein Service Worker ist ein „programmierbarer“ Proxy, der steuert, wie Requests von der Webseite behandelt werden.
- Service Workers verwenden die IndexDB-API, um client-seitig strukturierte Daten persistent zu speichern.
- Service Workers verwenden Promises.

### 7.2.1. Der Lebenszyklus eines Service Workers

Der Service Worker wird von einem Browser in einem eigenen Thread unabhängig von der Applikation ausgeführt. Dementsprechend besitzt der Service Worker einen eigenen Lebenszyklus, der im Folgenden beschrieben wird.

### Registrieren

Der Lebenszyklus eines Service Workers beginnt mit der Registrierung. Dies erfolgt mithilfe der `register()`-Methode in der `serviceWorker`-Eigenschaft des `navigator`-Objektes [doc], siehe Listing 7.1 Zeile 2. Um dem Konzept des Progressive Enhancements zu folgen (Abschnitt ??), sollte vor der Registrierung überprüft werden, ob die Service-Worker-Schnittstellen vorhanden sind. Hierfür wird die `serviceWorker`-Eigenschaft innerhalb des `navigator`-Objekts abgefragt, siehe Listing 7.1 Zeile 1.

```
1 if ('serviceWorker' in navigator){  
2     navigator.serviceWorker.register('./sw.js')  
3     .then(registration => console.log(registration))  
4     .catch(error => console.error(error));  
5 }
```

Listing 7.1: Registrierung des Service Workers

Das Aufrufen der `register()`-Methode gibt einen Promise zurück, welcher in Zeile 3 und 4 behandelt wird. Im Erfolgsfall (`then`-Methode) wird ein Objekt des Typs `ServiceWorkerRegistration` übergeben. Das Objekt verfügt unter anderem über folgende Schnittstellen:

- mit der `update()`-Methode wird die Aktualisierung des Service Workers ausgelöst,
- die `unregister()`-Methode hebt die Registrierung des Service Workers auf, welches im Anschluss durch den Browser gelöscht wird,
- die Eigenschaft `updatefound` wird aufgerufen, sobald der Browser eine neue Version des Service Workers gefunden hat.

### Installieren

Mit der Registrierung erfolgt ebenfalls die Installation des Service-Worker-Skripts, welches bei der Registrierung übergeben wurde, siehe Listing 7.1 Zeile 2. In der Installationsphase lädt der Service Worker Ressourcen von dem Webserver und speichert diese lokal ab. Wichtig ist hierbei, dass die Installationsphase erst dann beendet werden darf, wenn alle benötigten Ressourcen heruntergeladen wurden. Um dies zu realisieren, steht die `waitFor()`-Methode zur Verfügung, die eine Promise-Kette abarbeitet. Diese Methode öffnet den zugehörigen Cache. Im Cache werden durch die `addAll()`-Methode alle benötigten Ressourcen gespeichert, die normalerweise bei dem Öffnen der Applikation durch den Webserver angefordert werden. Dies ermöglicht, dass die Applikation in Zukunft auch ohne Netzwerkverbindung aus dem lokalen Cache geladen werden kann.

Die im Cache gespeicherten Dateien können mithilfe der Google Chrome Developer Tools unter dem Reiter *Application* eingesehen werden.

### Aktivieren

Nach der Installation geht der Service Worker in den aktiven Zustand über. Ein aktiver Service Worker ist in der Lage, funktionale Ereignisse zu behandeln wie zum Beispiel das Beantworten von HTTP(S) Anfragen oder das Entgegennehmen von Pushbenachrichtigungen. Der Service Worker übernimmt somit die Kontrolle der Applikation. Dies geschieht jedoch erst, wenn ein neuer Browser-Kontext geöffnet wird. Aktiviert man Service Worker während der Laufzeit der Webanwendung, kontrolliert er alle ihrer HTTP(S)-Anfragen, darunter auch jene, deren Ressourcen noch nicht im Cache gespeichert wurden.

### Überschreiben

Sobald ein neuer Service Worker für einen bestimmten Scope aktiviert wurde, terminiert der Webbrowser das alte Skript. Der alte Service Worker geht in den Zustand *redundant* und wird in dem weiteren Verlauf entfernt.

Unter Apple Safari wird ein registrierter Service Worker automatisch deinstalliert, sobald die Webseite des Service Workers mehrere Wochen nicht aufgerufen wurde.

## 7.3. Geräteschnittstellen

Die Eigenschaft von PWAs auf Geräteschnittstellen zugreifen zu können wird anhand der Geolocation-API gezeigt. Die Geolocation-Schnittstelle ist über das Navigator-Objekt erreichbar, welches bereits beim Service Worker zum Einsatz kam. Die Schnittstelle stellt drei Methoden zur Verfügung:

- `getCurrentPosition()`: Ruft einmalig die Position des Anwenders ab und gibt im Erfolgsfall ein Objekt mit der Position des Anwenders zurück.
- `watchPosition()`: Die Methode registriert eine Callback-Funktion, die aufgerufen wird, sobald sich die Anwenderposition verändert.
- `clearWatch()`: löscht die unter `watchPosition()` beschriebene registrierte Callback-Funktion.

## 7.4. Push-Notifications

Eine Notification ist eine Nachricht, die auf dem Endgerät des Nutzers angezeigt wird. Dies kann als Reaktion auf eine Nutzereingabe geschehen. Es kann jedoch auch unabhängig von dem Nutzer eine Nachricht von einem Server „gepusht“ werden. Die ist auch ohne das Aktivsein der Applikation möglich. Um Push-Notifications zu erzeugen, werden zwei APIs benötigt. Die *Notifications API* visualisiert die Nachricht für den Nutzer. Die *Push API* erlaubt es wiederum dem Service Worker (Kapitel 7.2) Nachrichten zu verwalten, die durch den Server gesendet wurden, während die Applikation inaktiv war.

Neben Frontend, Backend und Service Worker sind an der Push-Kommunikation zudem der Push-Service sowie der Push-Dienst beteiligt. Jeder Browserhersteller hat einen eigenen Push-Dienst (eng. *unser agent*) implementiert. Google Chrome verwendet *Firebase Cloud Messaging*, Mozilla Firefox den *Mozilla Push Service* und Microsoft Edge die *Windows Push Notification Services*. Der Push-Dienst nimmt die Pushnachrichten des Push-Services entgegen und leitet diese an den Service Worker weiter. Die Kommunikation mit den Push-Diensten ist durch die Internet Engineering Task Force ([IETF](#)) spezifiziert [[TDR16](#)]. Somit müssen keine gesonderten Implementierungen für die einzelnen Push-Dienste vorgenommen werden.

Wie die einzelnen Kommunikationspartner im Genauen miteinander interagieren ist in dem Sequenzdiagramm in Abbildung 7.5 dargestellt und wird im Folgendem erläutert.

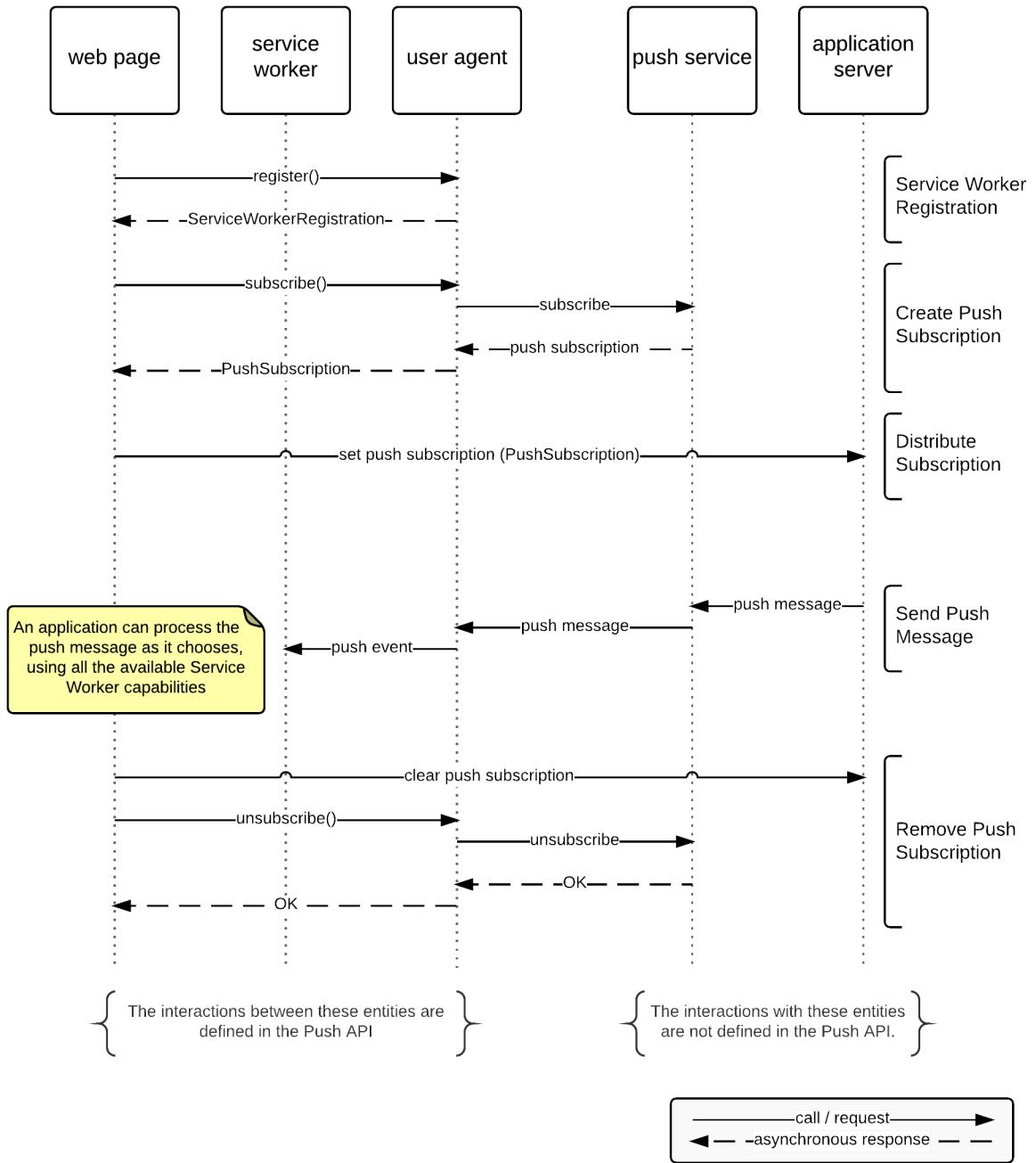


Abbildung 7.5.: Sequenzdiagramm für die Ereignisse einer Anmeldung, eines Push-Nachrichtentransfers und einer Abmeldung [BT16]

### 7.4.1. Push-Registrierung

Um Push-Nachrichten empfangen zu können, muss die Webanwendung über den Webbrowsr eine Push-Registrierung bei dem Push-Dienst beantragen. Diese Beantragung erfolgt über die PushAPI, die über den ServiceWorker - genauer die **ServiceWorkerRegistration** - zur Verfügung gestellt wird. Aus diesem Grund ist es zwingend notwendig, dass der Service Worker registriert ist. Bei der Registrierung sollte das Konzept des *Progressive Enhancements* berücksichtigt werden, indem die Browserkompatibilität vor der Registrierung abgefragt wird, siehe Listing 7.2.

```
1  if('serviceWorker' in navigator){  
2      navigator.serviceWorker.register('sw.js');  
3      navigator.serviceWorker.ready.then(registration => {  
4          if('PushManager' in window){  
5              registerForPush(registration.pushManager);  
6          }  
7      });  
8  }
```

Listing 7.2: Push-Registrierung unter Berücksichtigung der Konzepte des Progressive Enhancements

Falls der Browser über die nötigen Funktionen verfügt, kann die `subscribe()`-Methode der **ServiceWorkerRegistration** aufgerufen werden, welches ein Objekt des Typs **PushSubscription** zurückgibt. Die **PushSubscription** verfügt über ein Konfigurationsobjekt mit zwei Eigenschaften. Die erste Eigenschaft trägt den Namen `userVisibleOnly` und ist eine boolsche Eigenschaft, die beschreibt, ob Pushnachrichten immer zu einer für den Anwender sichtbaren Meldung führen. Diese hat standardmäßig den Wert `false`. Innerhalb des Browsers Google Chrome führt das jedoch zu Komplikationen, da der Browser aus Gründen der Datensicherheit keine sogenannte *Silent Push* Nachrichten zulässt. Daher sollte diese Eigenschaft auf den Wert `true` geändert werden.

Die zweite Eigenschaft des Konfigurationsobjekts bezeichnet man als `applicationServerKey`. Diese Eigenschaft wird benötigt, um die Nachrichten zwischen dem Push-Dienst und der Anwendung zu verschlüsseln. Zur Verschlüsselung wird das Voluntary Application Server Identification (**VAPID**)-Protokoll verwendet, das von der IETF spezifiziert wurde [TB17]. Die Verschlüsselung benötigt einen öffentlichen und einen privaten Schlüssel. Der öffentliche Schlüssel ist unter der `applicationServerKey`-Eigenschaft gespeichert. Der private Schlüssel wird wiederum in dem Backend der Anwendung hinterlegt.

Um die Registrierung abzuschließen, muss der Anwender sein Einverständnis zum Empfangen von Push-Benachrichtigungen geben. Für die Abfrage öffnet der Browser einmalig ein Dialogfenster. Wird kein Einverständnis vom Nutzer erteilt, so wird auf diesem speziellen

System, in keinem Fall, erneut danach gefragt. Der Anwender müsste die Berechtigung manuell über das zugehörige Webseitenmenü erteilen, sollte er sich entscheiden.

Nachdem die Berechtigung durch den Nutzer erteilt wurde, ist die Registrierung abgeschlossen. Die Anwendung ist nun in der Lage Push-Nachrichten zu empfangen.

### 7.4.2. Informationsaustausch

Nach einer erfolgreichen Registrierung erhält der Anwender ein Subscription-Objekt vom Typ `PushSubscription`. Das Subscription-Objekt wird dem Backend übergeben. Jeder Anwender, der die Pushbenachrichtigung abonniert hat, besitzt ein eigenes Subscription-Objekt, welches im Backend der Anwendung verwaltet wird. Die Subscription-Objekte sind für jeden Anwender einzigartig, da das Subscription-Objekt über die Eigenschaft `endpoint` verfügt. Hinter dieser Eigenschaft, befindet sich die URL des API-Endpunktes vom Push-Dienst. Der Browserspezifische Push-Dienst versendet die Push-Nachricht an den Service Worker und somit an den Anwender.

Die API-Endpunkte der Push-Dienste sind standardisiert. Über den Endpunkt kann nun eine Benachrichtigungen durch das Backend angestoßen werden. Hierbei empfiehlt es sich eine Web-Push-Bibliothek zu verwenden, um die standardisierte Kommunikation über den Endpunkt mit dem Push-Dienst zu erleichtern.

### 7.4.3. Zusammenfassung Push-Notifikation

Im folgenden Szenario werden die einzelnen Schritte zusammengefasst, um eine Push-Nachricht zu erhalten. Hierfür wird von einer Zeitschrift-Webseite ausgegangen. Der Nutzer besucht die Webseite zum ersten mal über den Browser *Chrome*. Automatisch wird der Service Worker im Hintergrund installiert. Der Nutzer wird nach der Einwilligung zum erhalten von Push-Nachrichten gefragt, und bestätigt diese. Durch die Bestätigung, wird ein Objekt vom Typ `PushSubscription` erzeugt und an das Backend der Zeitschrift-Webseite gesendet. Das `PushSubscription` Objekt verfügt über die URL eines API-Endpunktes. Der Endpunkt ermöglicht die Kommunikation zum Push-Dienst von Google mit dem Namen *Firebase Cloud Messaging*. Im Backend wird das `PushSubscription`-Objekt in einer Datenbank abgelegt.

Die Zeitschrift-Firma möchte nun alle Abonnenten darüber informieren, dass sie einen neuen Artikel erstellt haben. Dafür wird im Backend über alle in der Datenbank gespeicherten `PushSubscription`-Objekte iteriert. Über das `Web-Push-Protokoll` und dem im

PushSubscription-Objekt hinterlegten Endpunkten wird die Nachricht, das ein neuer Artikel hochgeladen wurde an die Push-Dienste gesendet.

Die Push-Dienste leiten diese Nachricht an den Nutzer über den Service Worker weiter oder speichern diese, falls der Nutzer nicht erreichbar ist.

## 7.5. Progressive Web Apps in Angular

Angular (Abschnitt 3) ist geeignet, um eine PWA zu entwickeln, da sowohl das Framework als auch die Progressive Web Apps Entwicklungen von Google sind. Um aus einer Angular-Anwendung eine PWA zu erstellen, muss das Paket `@angular/pwa` über das Command Line Interface (CLI) hinzugefügt werden. Durch das Hinzufügen des Pakets werden im Hintergrund folgende Veränderungen an dem Projekt durchgeführt:

- Das Paket `@angular/service-worker` wird zu dem Projekt hinzugefügt,
- der Build Support für den Service Worker wird in der Angular CLI aktiviert,
- das `ServiceWorkerModule` wird in dem `AppModule` importiert,
- die Datei `index.html` wird um einen Link zu dem Web App Manifest (Abschitt 7.1) und um relevante Meta-Tags ergänzt,
- Icon-Dateien werden erzeugt und verlinkt,
- die Konfigurationsdatei `ngsw-config.json` für den Service Worker wird erzeugt.

Im Anschluss kann die Applikation alle Grundfunktionen einer PWA ausführen. Die Anwendung kann installiert werden. Innerhalb des Web App Manifests steht der Projektname und das unter `assets` abgespeicherte Angular-Icon wird als App-Icon verwendet. Nach dem erstmaligen Öffnen der Applikation in dem Browser wird der Service Worker installiert.

Angular unterstützt den Entwickler beim Cachen von Daten und Dateien durch eine Konfigurationsdatei mit dem Namen `ngsw-config.json`. Die Grundeinstellung wird in Listing 7.3 beschrieben. Unter `assetGroups` werden Ressourcen angegeben, die teil der Angular-Anwendung sind und im Cache gespeichert werden. Aus der Datei kann entnommen werden, dass die `index.html`, das `webmanifest`, alle `css` und `JavaScript`-Dateien sowie alle Inhalte aus dem `assets`-Verzeichnis im Cache gespeichert werden. Somit kann die Applikation auch ohne Netzwerkverbindung gestartet werden, da alle benötigten Ressourcen im Cache hinterlegt sind.

Zusätzlich zu den Ressourcen der Angular-Anwendung können auch die Antworten von API-Anfragen im Cache gespeichert werden. Hierfür werden unter der Eigenschaft `dataGroups`

URL-Pfade von API-Endpunkten angegeben. Sobald der Service Worker eine API-Anfrage des Nutzer weiterleitet, der mit einem URL-Pfad aus `dataGroups` übereinstimmt, wird die Antwort im Cache gespeichert. Somit ist der Service Worker in der Lage auch ohne Netzwerkverbindung die API-Anfrage aus dem Cache zu beantworten.

Es kann zwischen zwei Cache-Strategien unterschieden werden. Bei der `freshness`-Strategie wird jede API-Anfrage an den entsprechenden Endpunkt weitergeleitet. Die Daten werden erst dann aus dem Cache bezogen, wenn der Endpunkt nicht erreichbar ist. Diese Strategie wird gewählt, wenn sich die Angefragten Daten regelmäßig ändern. Bei der `performance`-Strategie werden die Daten immer aus dem Cache geladen. Die Daten aus dem Cache werden in regelmäßigen Abständen aktualisiert. Das Laden aus dem Cache ist deutlich schneller als ein Netzwerzugriff. Diese Strategie eignet sich besonders für Daten die nicht oft geändert werden.

```

1   "assetGroups": [
2     {
3       "name": "app",
4       "installMode": "prefetch",
5       "resources": {
6         "files": [
7           "/favicon.ico",
8           "/index.html",
9           "/manifest.webmanifest",
10          "/*.css",
11          "/*.js"
12        ]
13      }
14    },
15    {
16      "name": "assets",
17      "installMode": "lazy",
18      "updateMode": "prefetch",
19      "resources": {
20        "files": [
21          "/assets/**",
22          "/*.(eot|svg|curl|jpg|png|webp|gif|otf|ttf|woff|woff2|
23            ani)"
24        ]
25      }
26    }
27  ]
28 }
```

Listing 7.3: Angular `ngsw-config.json`-Datei zur Angabe der Ressourcen, die durch den Service Worker in den Cache gespeichert werden sollen

## 7.6. Endgerät-Emulation

Um eine PWA lokal auf Endgeräten zu testen, können diese emuliert werden.

### Android

Um ein Android-Handy zu emulieren, kann das Programm *Android Studio* verwendet werden. In *Android Studio* kann mithilfe des *Android Virtual Device Manager* ([AVD Managers](#)) ein virtuelles Handy gestartet werden, Abbildung [7.6](#). In dem emuliertem Handy kann der Browser geöffnet werden. Um auf die lokale IP-Adresse des Rechners zugreifen zu können, muss innerhalb des Browsers auf dem Handy die IP-Adresse 10.0.2.2:8080 eingegeben werden.

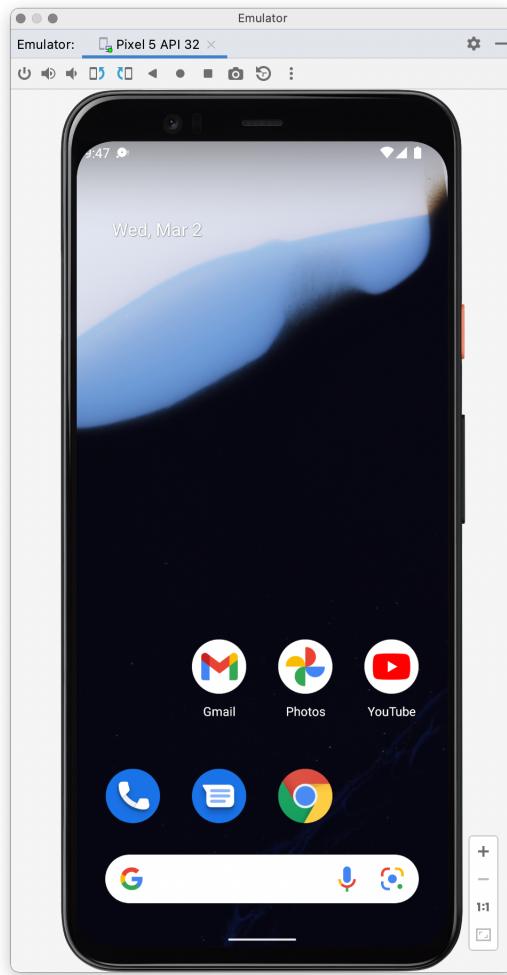


Abbildung 7.6.: Emuliertes Android Handy in Android Studio

### iOS

Um ein iOS-Gerät zu emulieren, wird das Programm XCode benötigt, welches jedoch nur auf einem MacOS Gerät installiert werden kann. In XCode kann ein einfaches `HelloWorld` iOS-Projekt erstellt werden. Wird dieses ausgeführt öffnet sich automatisch ein emuliertes iPhone, siehe Abbildung [7.7](#).

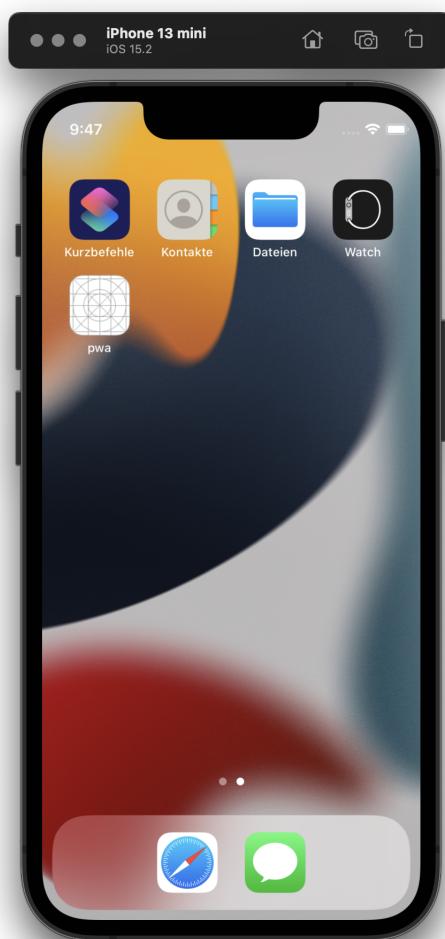


Abbildung 7.7.: Emulierte IPhone in XCode

## **ngrok**

Das das Werkzeug ngrok ist es auch möglich die PWA auf dem eigenen Endgerät zu Simulieren. Ngrok stellt lokale Endpunkte wie `localhost` unter eine öffentlich erreichbare Adresse zur Verfügung.

# **Teil II.**

## **Praktische Umsetzung**

## 8. Funktionalitäten

Startet man die Progressive Web App gelangt man zunächst zum Login. Hier muss man den eigenen Benutzernamen sowie das Passwort eingeben, um weiter zu gelangen. Hat man noch keinen User angelegt, kann man sich neu registrieren. Bei Klicken des `Create an account`-Buttons wird man auf eine neue Seite geleitet, um dort einen User anzulegen. Dafür muss ein Benutzername und ein Passwort gewählt werden. Nachdem das Passwort wiederholt wurde, kann man sich anmelden und hat einen neuen User registriert.

Auf der Hauptseite befindet sich in der Mitte ein Bild, das von einem User in die PWA hochgeladen wurde. Darunter kann man über das Herz den Post liken sowie die Anzahl der Likes, die das Bild bereits erhalten hat, sehen. Daneben befindet sich eine Sprechblase, über die man den Bereich der Kommentare erreichen kann. Klickt man darauf, wird man zu einer neuen Seite geleitet, auf der die Kommentare anderer User zu diesem Post zu sehen sind. Ebenso kann man unten in einem Schreifeld selbst einen Kommentar eingeben und über `send` posten. Klickt man erneut auf die Sprechblase, gelangt man zurück zur Hauptseite. Hier kann man mit den Pfeilen, die sich am linken und rechten Rand des Bildes befinden, zu anderen Posts wechseln und sich neue Bilder anzeigen lassen.

Außerdem kann man rechts oben über den `Subscribe to push`-Button Push-Nachrichten aktivieren.

Zuletzt kann man - sofern man angemeldet ist - selbst Posts hochladen und sie anderen anzeigen lassen. Dazu wählt man das Upload-Icon im unteren rechten Rand der PWA aus und gelangt auf eine neue Seite. Dort lässt sich per Drag and Drop ein Bild reinziehen, das im Anschluss gepostet werden kann.

# 9. Layout

Im Folgendem wird beschrieben, wie die Applikation aufgebaut ist. Und wie durch ein Responsive Design die Anwendung auch unabhängig von der Bildschirmgröße übersichtlich dargestellt werden kann.

## 9.1. Layout Webanwendung

Die Abwendung verfügt über folgende von einander Abgrenzbare Elemente:

- obere Navigationsleiste
- Hauptfeld
- Fußzeile

Die obere Navigationsleiste beinhaltet den Title der Anwendung, ein Knopf um Push-Nachrichten zu Abonnieren, und einen um zur Anmeldeseite zu gelangen. Im Hauptfeld werden wahlweise Fotos oder Kommentare angezeigt. Im unterem Teil des Hauptfeldes befinden sich zwei Icons. Das Herz-Icon kann vom Nutzer ausgewählt werden, was zu einer Inkrementierung des Zählers führt, der wiederum in der oberen rechten Ecke des Herz-Icons zu sehen ist. Das Kommentar-Icon dient als Navigationsobjekt um zwischen der Anzeige von Bildern und Kommentaren zu wechseln. Das HTML-Grundgerüst der Anwendung ist in Listing 9.1 dargestellt.

```
1  <div class="pageContainer">
2      <div class="navbar">
3          <app-navbar></app-navbar>
4      </div>
5      <div class="insta">
6          <app-inst-a-container></app-inst-a-container>
7      </div>
8      <div class="footer">
9      </div>
10 </div>
```

Listing 9.1: HTML-Grundgerüst der Webanwendung

### 9.1.1. CSS-Grid

Das Anordnen der Elemente und das Verhalten bei unterschiedlichen Bildschirmbreiten wird mithilfe von CSS-Grid umgesetzt. Mithilfe von CSS-Grid kann ein Raster aus Zeilen und Spalten erstellt werden. Die HTML-Elemente werden im Anschluss dem Raster zuteilt. Das Raster der Anwendung und dem im Listing aufgeführtem HTML-Grundgerüst (Listing 9.1) ist in Abbildung 9.1 dargestellt.

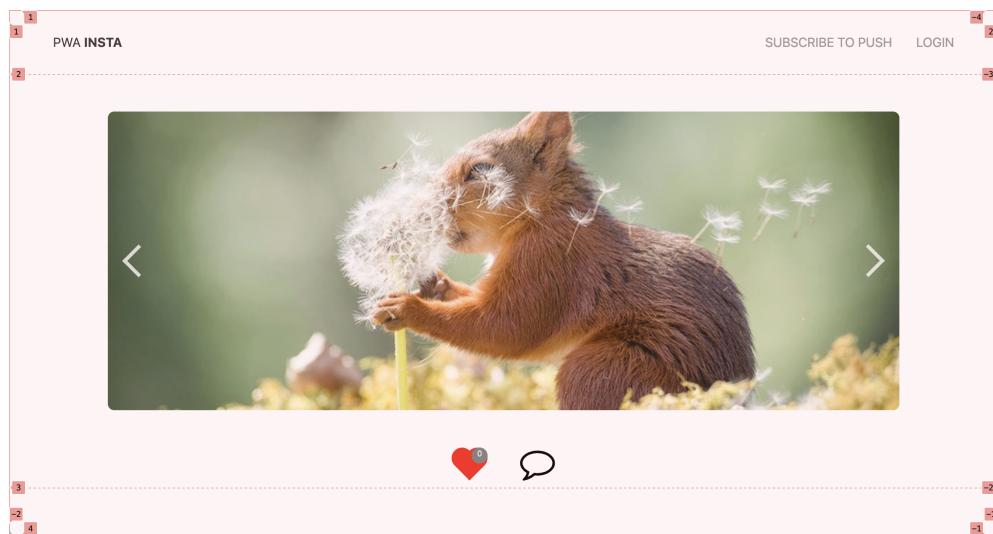


Abbildung 9.1.: CSS-Grid Raster der Hauptanwendung.

Die Realisierung des Rasters in CSS ist in Listing 9.2 aufgeführt. Die CSS-Klassen beziehen sich auf die, aus dem HTML-Grundgerüst 9.1.

```
1 .pageContainer{  
2     width: 100%;  
3     height: 100%;  
4     display: grid;  
5     grid-template-areas:  
6         "navbar"  
7         "insta"  
8         "footer";  
9     grid-template-rows: 80px 1fr 60px;  
10 }  
11  
12 .navbar {  
13     grid-area: navbar;  
14     width: 100%;  
15     height: 100%;  
16 }  
17  
18 .insta{  
19     width: 100%;  
20     height: 100%;  
21     grid-area: insta;  
22 }  
23  
24 .footer{  
25     width: 100%;  
26     height: 100%;  
27     grid-area: footer;  
28 }
```

Listing 9.2: Realisierung des Grid-Rasters in CSS

# **10. Node.js und Node Express**

## **10.1. Routing**

Unter Verwendung von Node Express soll das Routing umgesetzt werden. Zunächst müssen die Endpunkte mitsamt ihrer Callback-Funktionen definiert werden, bevor festgelegt wird, wie man sie verwenden kann.

### **10.1.1. Endpunkte**

Mithilfe des Routings kann auf die Daten innerhalb der MongoDB zugegriffen werden. Dazu werden verschiedene Endpunkte benötigt. Die verwendeten Endpunkte sind in den folgenden Tabellen definiert.

<b>Method</b>	<b>URL</b>	<b>Description</b>
GET	/users	get all users
GET	/users/:id	get user by id
GET	/users/:id/username	get username of user specified by id
GET	/users/:id/password	get password of user specified by id
GET	/users/:id/pictureID	get pictureID of user specified by id
GET	/users/login	together with the path a json file is sent, that includes the username and password given at login, afterwards it is checked whether there is one user with this username and password
POST	/users	create new user
PATCH	/users/:id	update user by id
DELETE	/users/:id	delete user specified by id

Tabelle 10.1.: User Endpunkte

<b>Method</b>	<b>URL</b>	<b>Description</b>
GET	/comments	get all comments
GET	/comments/:id	get comment by id
GET	/comments/:id/username	get username of user who wrote the comment specified by id
GET	/comments/:id/message	get message of comment specified by id
POST	/comments	create new comment
DELETE	/comments/:id	delete comment specified by id

Tabelle 10.2.: Comments Endpunkte

Method	URL	Description
GET	/posts	get all posts
GET	/posts/:id	get picture by id
GET	/posts/:id/caption	get caption of post specified by id
GET	/posts/:id/picture	get picture of post specified by id
GET	/posts/:id/geoloc	get geolocation of post specified by id
POST	/posts	create new post
PATCH	/posts/:id	update post by id
DELETE	/posts/:id	delete post specified by id

Tabelle 10.3.: Posts Endpunkte

Method	URL	Description
GET	/likes	get all likes
GET	/likes/:id	get like by id
GET	/likes/:id/userid	get id of user who gave the like specified by id
GET	/likes/:id/pictureid	get id of picture the like specified by id is given to
POST	/likes	create new like
DELETE	/likes/:id	delete like specified by id

Tabelle 10.4.: Likes Endpunkte

### 10.1.2. Request-Funktionen

Innerhalb eines Routing-Files werden alle Responses auf mögliche http-Anfragen des Clients festgelegt. Grundlegend sehen die jeweiligen GET-, POST-, PATCH- und DELETE-Funktionen gleich aus, weshalb im Folgenden zu jeder Anfrage ein Beispiel erläutert wird.

```
1   router.get('/users/:id', async(req, res) => {
2     try {
3       const user = await User.findOne({ _id: req.params.id });
4       console.log(req.params);
5       res.send(user);
6     } catch {
7       res.status(404);
8       res.send({
9         error: "User does not exist!"
10      });
11    }
12  });

```

Listing 10.1: GET-Request

Mit der obigen Beispiel-Funktion wird ein User über seine ID unter Verwendung der http-Request-Methode GET aus der Datenbank ausgelesen. Der zugehörige Endpunkt ist `/users/:id`. Unter `:id` kann der Client später den Identifier des Users einfügen, um anhand dessen die Daten dieses Users herauszufinden. Nach dem Pfeil wird die Callback-Funktion definiert. Sie beschreibt, wie der Server auf die Anfrage reagieren soll. Dabei soll er zunächst probieren, den der ID zugeordneten User zu finden. Findet der Server diesen User in der Datenbank, wird er mit allen enthaltenen Daten zurückgegeben. Ist dies nicht der Fall, wird ein Error geworfen und die Konsole enthält die Information `User does not exist!`.

```
1   router.post('/users', async(req, res) => {
2     const newUser = new User({
3       _id: req.params.id,
4       username: req.body.username,
5       password: req.body.password,
6       pictureId: req.body.pictureId,
7     })
8     await newUser.save();
9     res.send(newUser);
10  });

```

Listing 10.2: POST-Request

In obigem Code kann man eine POST-Anfrage an den Endpunkt `/users` erkennen. Im Zuge dieser Anfrage wird ein JSON-Objekt gesendet, dessen Inhalte über `req.body.*` aufgerufen

werden können. Mit den enthaltenen Daten wird ein neuer User in der Datenbank angelegt. Zusätzlich erstellt MongoDB automatisch eine ID und weiß sie dem neuen User zu.

```

1   router.patch('/users/:id', async(req, res) => {
2     try {
3       const user = await User.findOne({ _id: req.params.id })
4
5       if (req.body.username) {
6         user.username = req.body.username
7       }
8
9       if (req.body.password) {
10        user.password = req.body.password
11      }
12
13      await User.updateOne({ _id: req.params.id }, user);
14      res.send(user)
15    } catch {
16      res.status(404)
17      res.send({ error: "User does not exist!" })
18    }
19  });

```

Listing 10.3: PATCH-Request

In diesem Beispiel kann man eine PATCH-Anfrage erkennen, mit der die Daten eines Users über seine ID aktualisiert werden können. Mit der Anfrage an den Endpunkt `/users/:id` wird ebenfalls ein JSON-Objekt gesendet, welches die neuen Daten enthält. Findet der Server in diesem Objekt neue Daten zu einem Schlüssel, wird dessen alter Wert überschrieben. Ist der Server nicht in der Lage einen User mit der eingegebenen ID zu finden, wird ein Error geworfen.

```

1   router.delete('/users/:id', async(req, res) => {
2     try {
3       await User.deleteOne({ _id: req.params.id })
4       res.status(204).send()
5     } catch {
6       res.status(404)
7       res.send({ error: "User does not exist!" })
8     }
9   });

```

Listing 10.4: DELETE-Request

In dem letzten Beispiel wird eine DELETE-Anfrage beschrieben. Mit dieser wird über den Endpunkt `/users/:id` ein User anhand seiner ID gelöscht. Findet der Server diesen User in der Datenbank, kann er ihn löschen und den Success-http-Status 204 zurücksenden. Im Fehlerfall wird ein Error geworfen und der Error-Status 404 gesendet.

### 10.1.3. Verwendung der Endpunkte

Die Requests und Callback-Funktionen sind bereits definiert. Nun muss der Server angewiesen werden, wie er diese zu verwenden hat. Dazu wird in einer separaten Datei `server.js` zunächst festgelegt, wo die gebauten Routen zu finden sind. Später können diese Routen mit `app.use()` aufgerufen werden und sind für den Client verfügbar. Ebenso muss in dieser Datei der Port festgelegt, über den die Endpunkte zu erreichen sind, sowie die Verbindung zu der Datenbank aufgebaut werden.

```
1 const routes = require('./routes/routes');
2 app.use('/', routes);
```

Listing 10.5: Verwendung der API-Routen

# 11. MongoDB

## 11.1. Mongoose

Die Verbindung der Datenbank mit der Express-Webanwendung kann mithilfe des Moduls Mongoose hergestellt werden, einem objektorientierten JavaScript-Framework, das eine einfach zu verwendete Schnittstelle zwischen Node.js und MongoDB darstellt. Die Verbindung wird in der `server.js`-Datei hergestellt und sieht folgendermaßen aus:

```
1  mongoose.connect('mongodb://127.0.0.1:27017', { useNewUrlParser:  
    true, useUnifiedTopology: true });
```

Listing 11.1: Herstellen der Verbindung zu MongoDB

Anstelle von `127.0.0.1` lässt sich auch `localhost` verwenden. Dabei können jedoch Probleme auftreten, sollt die Adresse geändert worden sein.

Mongoose arbeitet schema-basiert, weshalb sich die Verwendung von Schemata in Objekten zur Datenspeicherung anbietet.

## 11.2. Schemata

Für die Speicherung der Daten in der MongoDB werden vier Schemata entwickelt. Ein Schema ist ein JSON-Objekt, das die Struktur der zu speichernden Daten vorgibt. Für die Erstellung der PWA werden Schemata für einen User, einen Post, einen Kommentar und einen Like benötigt. Diese sollen folgende Daten enthalten:

User	
username	string
password	string
pictureID	string[]

Tabelle 11.1.: User-Schema

Unter Verwendung von

### Post

caption	string
geoloc	string

Tabelle 11.2.: Post-Schema

### Comment

userID	string
message	string
pictureID	string

Tabelle 11.3.: Comment-Schema

### Like

userID	string
pictureID	string

Tabelle 11.4.: Like-Schema

```
1 const schema = new mongoose.Schema({})
```

Listing 11.2: Erstellen eines neuen Schemas

lässt sich ein neues Schema erstellen. Die Definition der Schlüssel-Wert-Paare findet innerhalb der geschwungenen Klammern statt.

## 11.3. Dateiupload

Um Bilder in die MongoDB laden zu können wird zunächst eine Middleware benötigt. Dazu verwendet man Multer. In der Middleware wird festgelegt, an welchem Ort die Datei gespeichert werden soll und welche Formate angenommen werden. In diesem Fall sind das eine URL der Datenbank sowie die Bildformate png und jpeg. Wichtig ist, dass die Dateinamen einzigartig sind. Um dies zu bewahren, werden den Original-Dateinamen Zeitstempel hinzugefügt.

```
1 url: 'mongodb://127.0.0.1:27017',
```

Listing 11.3: Zuweisen der URL der Datenbank

```
1 const match = ["image/png", "image/jpeg"];
```

Listing 11.4: Festlegen der zugelassenen Dateiformate

```
1 filename: '${Date.now()}-${file.originalname}';
```

Listing 11.5: Festlegen des Dateinamens mit Zeitstempel

Die Middleware wird aufgerufen, sobald der Client einen POST-Request zum Hochladen eines Bildes ausführt. Dabei wird überprüft, ob die Anfrage ein passendes File enthält. Ist dies der Fall erhält man im Rückgabewert die URL des Speicherorts des Bildes.

```
1 const imgUrl = 'http://localhost:4000/download/${req.file.filename}';
```

Listing 11.6: Speicherort des Bildes

Ebenso lassen sich bereits gespeicherte Bilder aus der Datenbank downloaden. Dafür wird zunächst ein GET-Request an die Datenbank gesendet, der den Filenamen enthält. Dieser wird in der GridFS-Collection `fileupload` gesucht. Findet der Cursor den Filenamen, kann der Download gestartet werden. Da das Bild zuvor in einzelne Chunks zerlegt wurde, um in der MongoDB gespeichert werden zu können, müssen diese zuerst zusammengesetzt werden. Außerdem kann ein Bild über einen DELETE-Request gelöscht werden. Auch hier wird über den Filenamen geprüft, ob das Bild in der Datenbank enthalten ist. Falls ja kann es gelöscht werden.

# 12. Installation der PWA

Dadurch, das die Webanwendung über ein Service Worker und ein Web-Manifest verfügt, kann diese auf dem Endgerät installiert werden. Das von Angular erzeugte Web-Manifest wurde in den Eigenschaften `theme_color` und `background_color` angepasst, beide Eigenschaften wurden von Blau auf Weiß gesetzt.

Die Installation über Google Chrome erfolgt über den Download-Button in der URL-Suchleiste, siehe Abbildung 12.1

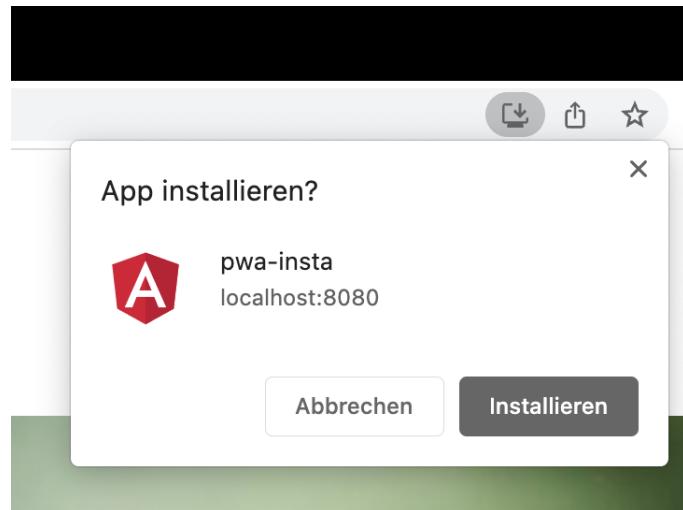


Abbildung 12.1.: Installation der PWA aus dem Google Chrome Browser durch die Auswahl des Download-Buttons in der URL-Suchleiste des Browsers.

Die PWA kann nicht über den Apple Safari Browser auf den Rechner installiert werden. Möchte man die PWA auf einem MacOS system installieren muss somit ein zusätzlicher Browser wie zum Beispiel Google Chrome installiert werden.

Nachdem die PWA unter Chrome installiert wurde, kann die Applikation unter *Programme* geöffnet werden. Die Desktop-Anwendung der PWA ist in Abbildung 12.2 dargestellt.

Obwohl die Installation eine PWA über Safari auf MacOS nicht möglich ist, kann diese über Safari auf IOS installiert werden. Hierfür kann unter dem *Teilen*-Button die Auswahl *Zum Home-Bildschirm* ausgewählt werden. Nachdem ein Name für die App eingegeben wurde, wird die Anwendung als App zu dem Bildschirm hinzugefügt. In der Folgenden Bilderserie sind die einzelnen Schritte aufgezeigt.

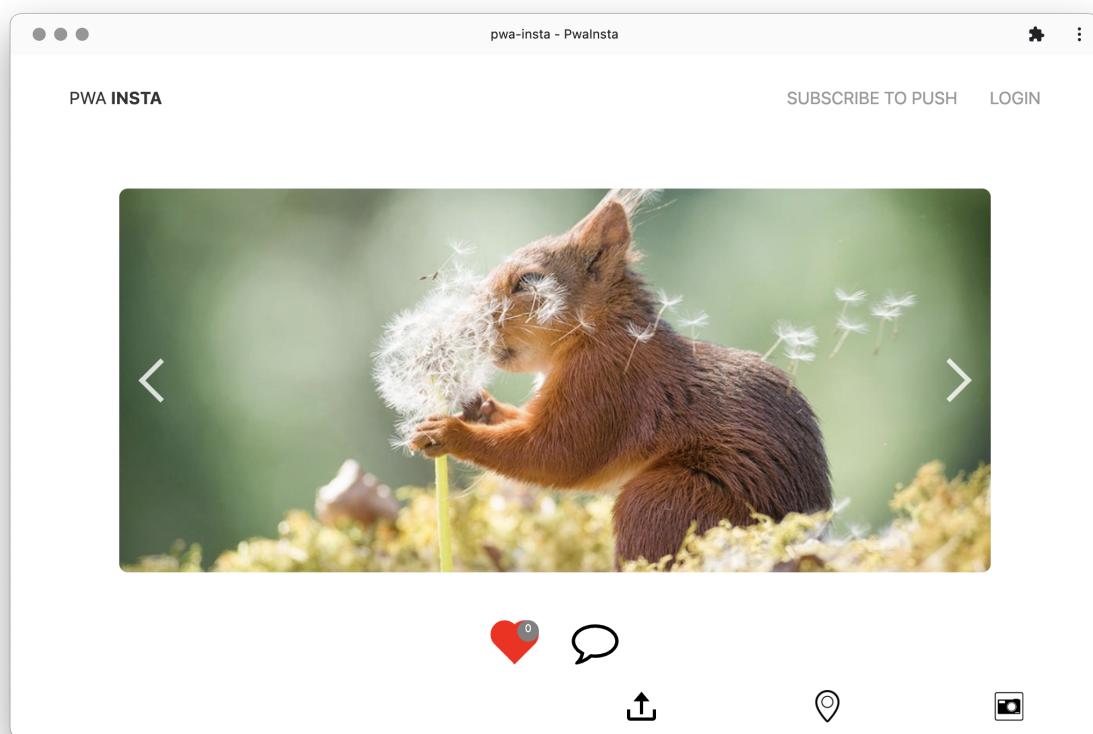


Abbildung 12.2.: Die installierte PWA als Desktop-Anwendung

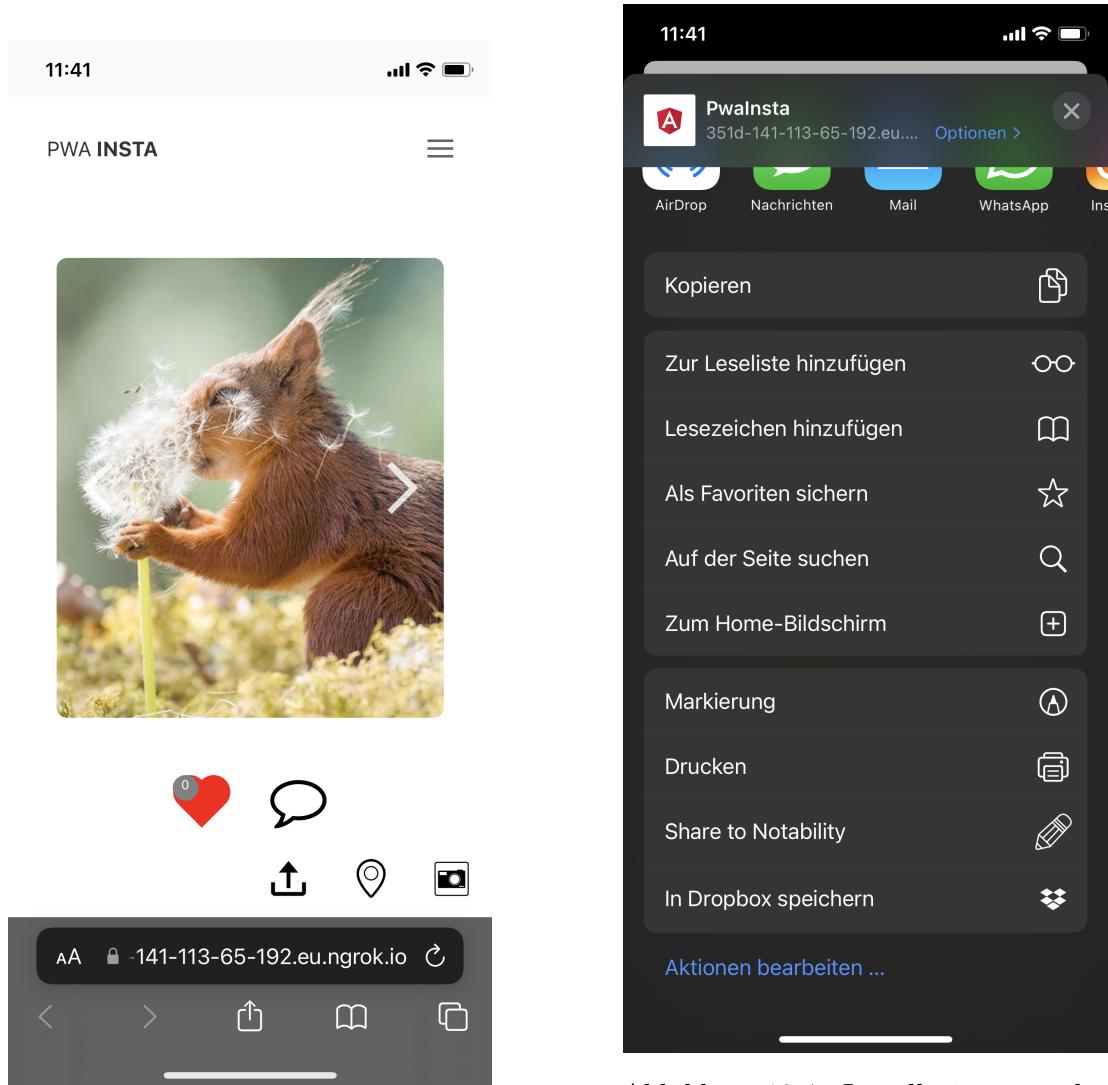


Abbildung 12.3.: PWA im Apple Safari Browser in einem IOS System

Abbildung 12.4.: Installation der PWA über die Auswahl *Zum Home-Bildschirm*

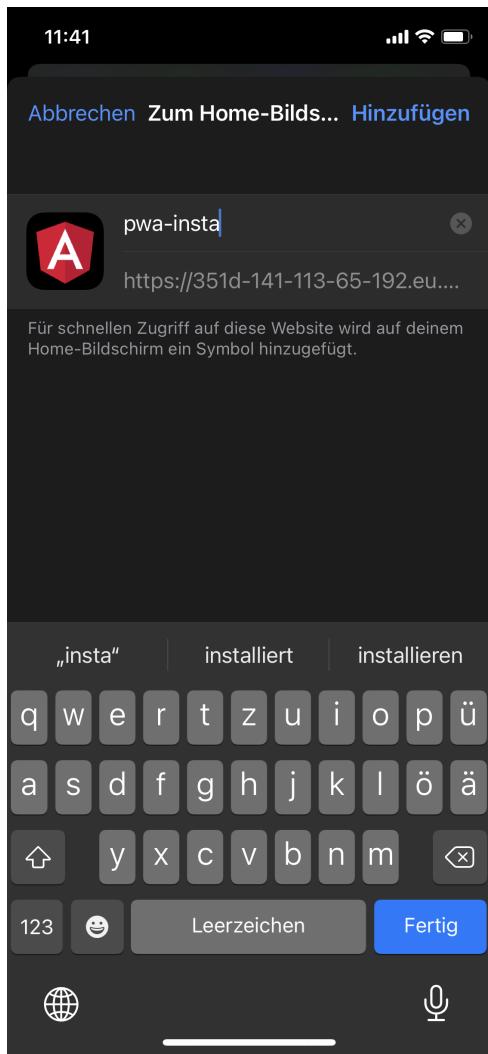


Abbildung 12.5.: Eingabe einer Bezeichnung für die PWA-App

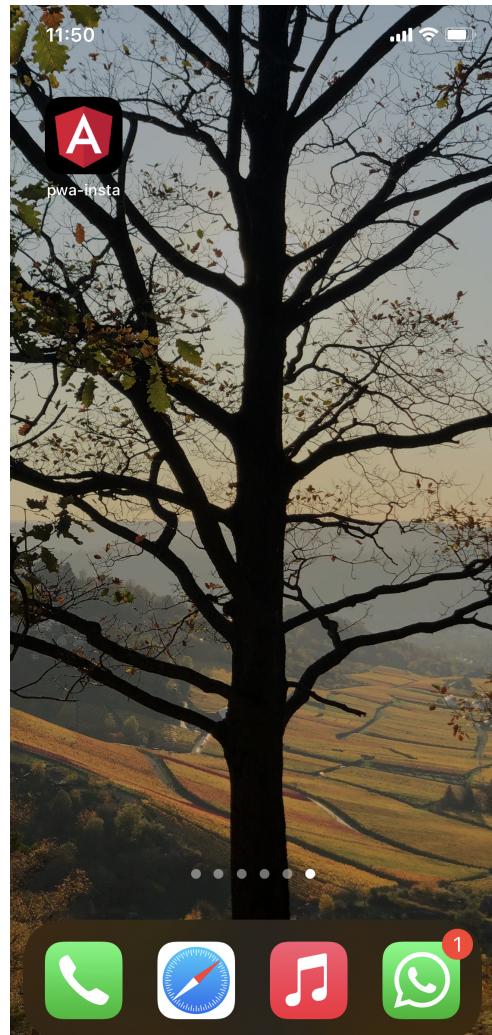


Abbildung 12.6.: Die installierte PWA-App auf dem Home-Bildschirm eines IOS-Systems

# **13. Verwendung von Geräteschnittstellen**

# 14. Push-Notifikation

## 14.1. Registrierung

Damit die Webanwendung Push-Nachrichten empfangen kann, muss sie sich vorab beim Push-Dienst registrieren. Die Registrierung erfordert das Einverständnis des Anwenders. Um das Anwendererlebnis zu verbessern, wird dieser nicht direkt danach gefragt sondern kann über den Button `SUBSCRIBE TO PUSH` die Registrierung anstoßen, siehe Abbildung 14.1. Um das Konzepte von Progressive Enhancement einzuhalten, wird der Button nur den Nutzern angezeigt, dessen Browser über die nötigen Funktionalitäten verfügt um Push-Nachrichten empfangen zu können. Hierfür wird der Button über eine `*ngIf`-Strukturdirektive ein- bzw ausgeblendet (Abschnitt 3.1), je nach dem, ob der Browser über ein `PushManager` verfügt oder nicht.

Der Button ist mithilfe eines Event Bindings (Abschnitt 3.4) an die Funktion `subscribeToNotifications()` verknüpft. Da es sich bei der Anwendung um ein Angular Projekt handelt, wird die Registrierung über die `SwPush`-Klasse durchgeführt (Listing 14.1 Zeile 2) die mithilfe von dependency injection über den Konstruktor geladen wurde. Wie in Abschnitt 7.4.1 beschrieben wird die Push-Kommunikation durch das VAPID-Protokoll verschlüsselt. Hierfür muss bei der Registrierung der öffentliche Schlüssel übergeben werden (Listing 14.1 Zeile 3). Anders als in Abschnitt 7.4.1 beschrieben, ist der Standartwert der Eigenschaft `userVisibleOnly`, aus dem Konfigurationsobjekt, `true`. Diese Einstellung wird von der Klasse `SwPush` vorgenommen.

Ist die Registrierung erfolgreich wird ein Objekt von typ `PushSubscription` zurückgegeben. Das Objekt verfügt über folgende Eigenschaften:

- **endpoint**: Hinter dieser Eigenschaft befindet sich eine einzigartige URL, die für die Kommunikation mit dem Push-Dienst verwendet wird.
- **expirationTime**: Durch diese Eigenschaft können Nachrichten versendet werden, die nur für einen bestimmten Zeitraum gelten.
- **keys**: Enthält Informationen, um die Nachrichten zu entschlüsseln.

Dieses Objekt muss im Anschluss über ein HTTP-Request an das Backend gesendet werden (Listing 14.1 Zeile 7-8). Hierbei wird die Kommunikation mit dem Backend in einem `Service` Ausgelagert.

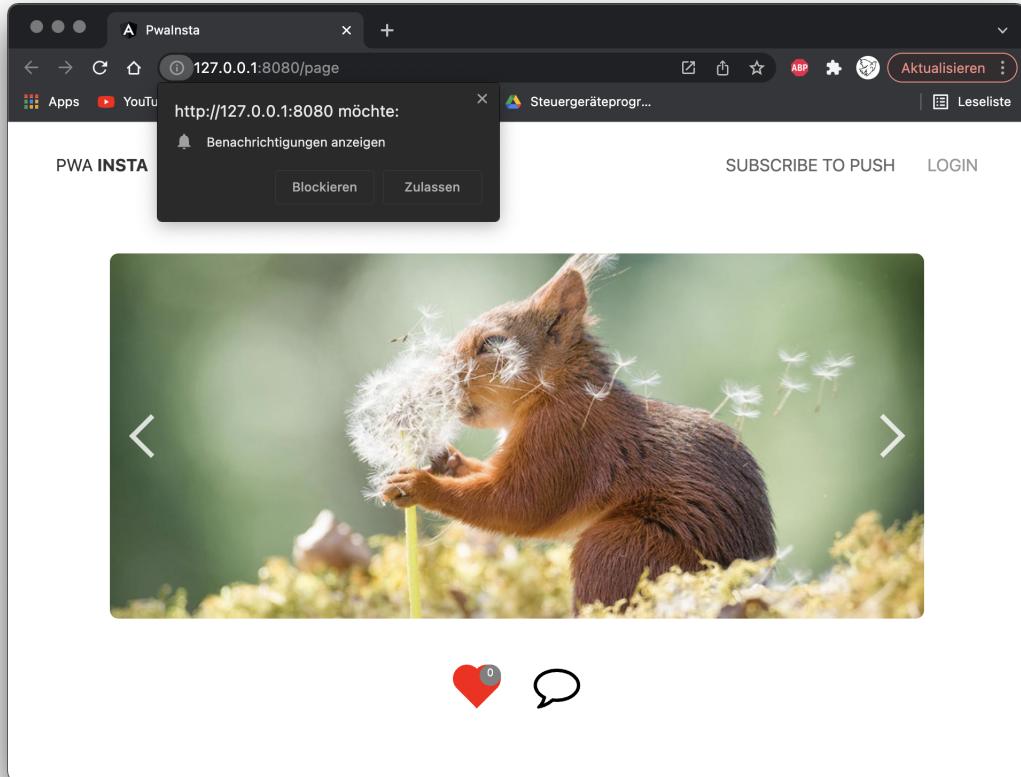


Abbildung 14.1.: Nutzerabfrage um Push-Nachrichten versenden zu können. Abfrage erfolgt erst nachdem der Button `SUBSCRIBE TO PUSH` in der oberen Navigationsleiste ausgewählt wurde.

Falls die Registrierung erfolglos war, wird in der `catch`-Methode des Promises der Fehler auf der Konsole ausgegeben (siehe Listing 14.1 Zeile 11).

Nach der Registrierung und dem Versenden des `PushSubscription` Objektes an das Backend müssen keine weiteren Vorkehrungen im Frontend vorgenommen werden.

```
1 subscribeToNotifications(){
2     this.swPush.requestSubscription({
3         serverPublicKey: this.VAPID_PUBLIC_KEY
4     })
5     .then(subscribeObj => {
6         console.log(subscribeObj);
7         this.pushService.addPushSubscriber(subscribeObj).subscribe(
8             res => {
9                 console.log('Backend\u2014Response\u2014Push\u2014Subscription' + res);
10            });
11     .catch(err => console.error("Could\u2014not\u2014subscribe\u2014to\u2014
12 notifications", err));
13 }
```

Listing 14.1: Funktion zur Registrierung beim Push-Dienst

## 14.2. Push-Nachrichten im Backend

Im Backend werden zwei `post`-Endpunkte implementiert. Der Subscription Endpunkt wird verwendet, um das `PushSubScription` Objekt entgegenzunehmen. Das Objekt wird in einer Datenbank gespeichert.

Der zweite Endpunkt wird verwendet um Push-Notifikations anzustoßen. Das Backend benutzt die Web-Push Bibliothek um allen Registrierten Klienten eine Push-Nachricht zu senden. Die `sendNotifikation`-Methode der Web-Push Bibliothek benötigt ein Payload und ein `PushSubScription`-Objekt. Im Payload werden die Informationen angegeben, die in der Push-Nachricht angezeigt werden, hierbei muss mindestens ein Title und eine Nachricht vorhanden sein, siehe Listing 14.2.

```
1 const notificationPayload = {
2     notification: {
3         title: 'Neue\u2014Notification',
4         body: 'Das\u2014ist\u2014der\u2014Inhalt\u2014der\u2014Notification'
5     },
6 }
```

Listing 14.2: Mindestanforderung an ein Payload, für eine Push-Nachricht

Die Web-Push Methode, verschlüsselt die Nachricht vor dem Versenden, hierfür müssen die VAPID-Schlüsselpaare der Methode `webpush.setVapidDetails()` übergeben werden. Wird nun eine HTTP-Post Anfrage an diesen Endpunkt gesendet, iteriert die Funktion über alle in der Datenbank gespeicherten `PushSubscription`-Objekte und sendet über die `sendNotifikation`-Methode den verschlüsselten Payload. Alle registrierten Klienten erhalten folgende Mitteilung, siehe Abbildung 14.2.

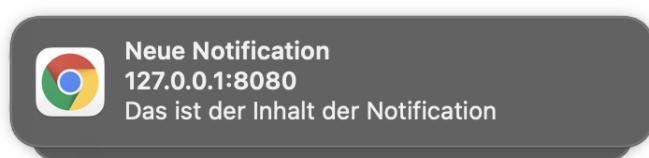


Abbildung 14.2.: Push-Nachricht, mit dem Payload aus Listing 14.2

# 15. Untersuchung der Plattformunabhängigkeit

Um die These,dass Progressive Web Apps plattformunabhängig und vollumfänglich eingesetzt werden können, aus Abschnitt 1.2 zu untersuchen, wird die Entwickelte Anwendung auf den gängigsten Plattformen ausgeführt und die Funktionen:

- installierbar,
- offline verfügbar,
- zugriff auf Geräteschnittstellen und
- Push-Benachrichtigungen

werden untersucht.

## 15.1. Google Chrome und Android

### Installierbar

Um die PWA in dem Browser Google Chrome zu installieren, muss der Download-Button in der Suchleiste des Browsers ausgewählt werden, siehe Abbildung ??.

### offline verfügbar

ToDo

siehe Abbildung 15.1. Da es sich in der Abbildung um eine Angular-Anwendung handelt, können die in der *ngsw-config.json*-Datei angegebene Ressourcen wiedergefunden werden.

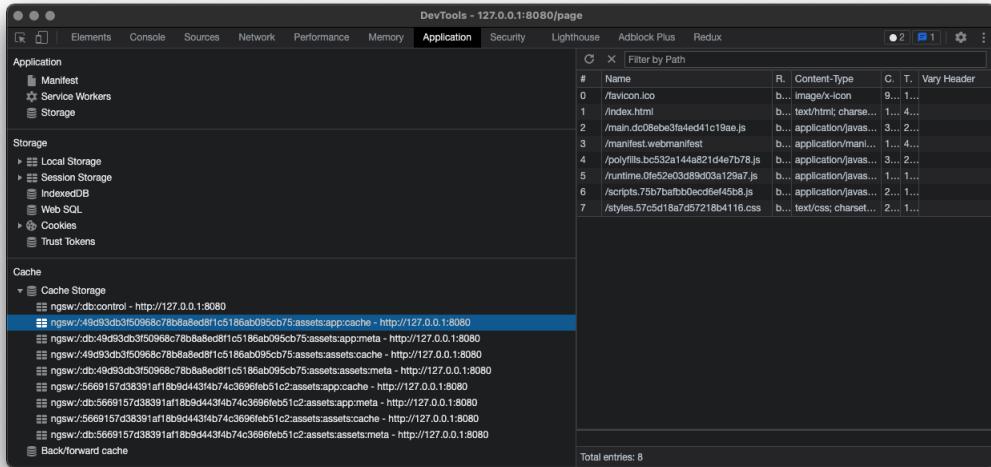


Abbildung 15.1.: Google Chrome Developer Tools zum Einsehen der im Cache gespeicherten Dateien

## 15.2. Apple Safari und iOS

Apple unterstützt die Entwicklung von PWAs nicht. PWAs stellen eine Möglichkeit dar, um Apps für das iPhone zu installieren, die sich nicht in dem App Store befinden. Im Jahr 2020 hat Apple mit dem App Store einen Umsatz von 643 Milliarden US-Dollar erwirtschaftet [Kir]. Apple erhält 30% Provision, wenn Apps gekauft oder In-App-Käufe abgeschlossen werden. Durch den Einsatz von PWAs könnte diese Einnahmen reduziert werden.

Durch zusätzliche Einstellungen können PWAs auf dem Homescreen von IOS Geräten installiert werden. In dem `head` der `index.html` muss der Dateipfad zu dem verwendeten Icon angegeben werden, da dieser nicht aus dem Web-App-Manifest ausgelesen wird. Des weiteren muss in einem `Meta-Tag` angegeben werden, dass die Webanwendung als App genutzt werden kann. Diese zusätzlichen Angaben ermöglichen es, die Applikation zu installieren.

Apple unterstützt weder im Browser noch im IOS-Betriebssystem die Push-API. Im Browser Safari ist der Push-Manager nicht implementiert, siehe Abbildung 15.2, somit sind Push-Benachrichtigungen wie unter Google Chrome nicht möglich.

Der Browser Safari bietet als Alternative *Safari Push Notifications*. Um Safari Push Notification versenden zu können, muss das Backend mit dem Apple Push Notification service ([APNs](#)) kommunizieren. Die Kommunikation erfolgt jedoch **nicht** über das standardisierte Web-Push-Protokoll.



Abbildung 15.2.: Apple Safari verfügt nicht über ein Push-Manager und ist somit nicht in der Lage Push-Nachrichten zu erhalten

Die APNs Kommunikation erfordert, dass der Entwickler sich beim Apple Developer Programm registriert und eine Push-ID für die Webanwendung anlegt. Anschließend muss der Entwickler ein Transport Layer Security ([TLS](#))-Clientzertifikat anfordern. In der Safari Umgebung wird anstelle des `PushSubscription`-Objektes ein `Device-Token` erzeugt, um Pushbenachrichtigungen zu versenden.

Es ist somit ein erhöhter Programmieraufwand nötig, um Push-Notifications für Apple-Umgebungen zu realisieren.

Apple beschränkt jedoch die Nutzung des vollen Funktionsumfangs der PWAs. Durch ein Softwareupdate hat Apple eine Zwangslösung für lokal beschreibbare Speicherfunktionen eingeführt [[t3n](#)]. Eine PWA nutzt diesen lokalen Speicher um Daten zu sichern.

# Literatur

- [Ang21a] Angular. *Event Binding*. 2021. URL: [Event%20binding](#).
- [Ang21b] Angular. *NgIf*. 2021. URL: <https://angular.io/api/common/NgIf>.
- [Ang21c] Angular. *Property binding*. 2021. URL: <https://angular.io/guide/property-binding>.
- [Ang21d] Angular. *Text interpolation*. 2021. URL: <https://angular.io/guide/text-interpolation>.
- [BT16] Peter Beverloo und Martin Thomson. *Push API*. März 2016. URL: <https://www.w3.org/TR/push-api/#abstract>.
- [Cha] Steve Champeon. *PROGRESSIVE ENHANCEMENT AND THE FUTURE OF WEB DESIGN*. URL: [http://www.hesketh.com/progressive\\_enhancement\\_and\\_the\\_future\\_of\\_web\\_design.html](http://www.hesketh.com/progressive_enhancement_and_the_future_of_web_design.html).
- [Dev] Google Developers. *Service workers*. URL: <https://web.dev/learn/pwa/service-workers/>.
- [Dev22] Google Developers. *Web app manifest*. 27. Feb. 2022. URL: <https://web.dev/learn/pwa/web-app-manifest/>.
- [doc] mdm web docs. *Navigator*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Navigator>.
- [ES15] Shakuntala Gupta Edward und Navin Sabharwal. *Practical MongoDB: Architecting, Developing, and Administering MongoDB*. Apress, 2015.
- [Exp22a] Express. *Basic routing*. 2022. URL: <https://expressjs.com/en/starter/basic-routing.html>.
- [Exp22b] Express. *Express glossary*. 2022. URL: <https://expressjs.com/en/resources/glossary.html>.
- [Exp22c] Express. *Node.js web application framework*. 2022. URL: <https://expressjs.com/>.
- [Exp22d] Express. *Routing*. 2022. URL: <https://expressjs.com/en/guide/routing.html>.
- [Exp22e] Express. *Using Express middleware*. 2022. URL: <https://expressjs.com/en/guide/using-middleware.html>.

- [Gan18] Kavita Nambissan Ganguli. *Uploading and retrieving a file from gridfs using Multer*. Nov. 2018. URL: <https://medium.com/@kavitanambissan/uploading-and-retrieving-a-file-from-gridfs-using-multer-958dfc9255e8>.
- [Gau] Matt Gaunt. *Service Workers: an Introduction*. URL: <https://developers.google.com/web/fundamentals/primers/service-workers>.
- [Gri22] GridFS. *GridFS - MongoDB Manual*. 2022. URL: <https://www.mongodb.com/docs/manual/core/gridfs/>.
- [Kir] Julius Kirchenbauer. *Apple Entwickler:innen stellen sich den Herausforderungen während der Pandemie und steigern die Verkäufe und Umsätze, die das App Store-Ökosystem ermöglicht hat, um 24 Prozent auf 643 Milliarden US-Dollar im Jahr 2020*. URL: <https://www.apple.com/de/newsroom/2021/06/apple-developers-grow-app-store-ecosystem-billings-and-sales-by-24-percent-in-2020/#:~:text=Apple%20hat%20heute%20bekannt%20gegeben,Prozent%20im%20Vergleich%20zum%20Vorjahr..>
- [Mc21] Mozilla und individual contributors. *Webtechnologien für Entwickler*. 2021. URL: <https://developer.mozilla.org/de/docs/Web>.
- [Mei18] Andreas Meier. *Werkzeuge der digitalen Wirtschaft: Big Data, NoSQL und Co.: Eine Einführung in relationale und nicht-relationale Datenbanken*. Springer Vieweg, 2018.
- [MHK20] F. Malcher, J. Hoppe und D. Koppenhagen. *Angular: Grundlagen, fortgeschritten Themen und Best Practices – inkl. RxJS, NgRx und PWA*. dpunkt.verlag, 2020. ISBN: 9783969100820. URL: <https://books.google.de/books?id=0e8BEAAAQBAJ>.
- [Nod22] Node.js. *About*. 2022. URL: <https://nodejs.org/en/about/>.
- [npm] npm. *Multer*. URL: <https://www.npmjs.com/package/multer>.
- [Per16] Caio Ribeiro Pereira. *Building APIs with Node.js*. Apress, 2016.
- [Sam] Sam Richard,Pete LePage. *What are Progressive Web Apps?* URL: <https://web.dev/progressive-web-apps/>.
- [Sas21] Sass. *Sass Basics*. 2021. URL: <https://sass-lang.com/guide>.
- [sta21] stackoverflow. *Web frameworks survey*. 2021. URL: <https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks>.
- [t3n] t3n. *Artikel merkenUnter dem Deckmantel des Guten: Apples neuer Safari-Browser behindert die Entwicklung von progressiven Web-Apps*. URL: <https://t3n.de/news/deckmantel-guten-apples-neuer-1266784/>.

- [TB17] Martin Thomson und Peter Beverloo. *Voluntary Application Server Identification (VAPID) for Web Push*. RFC 8292. Nov. 2017. doi: [10.17487/RFC8292](https://doi.org/10.17487/RFC8292). URL: <https://www.rfc-editor.org/info/rfc8292>.
- [TDR16] Martin Thomson, Elio Damaggio und Brian Raymor. *Generic Event Delivery Using HTTP Push*. RFC 8030. Dez. 2016. doi: [10.17487/RFC8030](https://doi.org/10.17487/RFC8030). URL: <https://www.rfc-editor.org/info/rfc8030>.
- [Typ21] TypeScript. *TypeScript for the New Programmer*. 2021. URL: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>.
- [Vet+22] Rob Vettor u.a. *Vergleich der relationalen und NoSQL-Daten*. 2022. URL: <https://docs.microsoft.com/de-de/dotnet/architecture/cloud-native/relational-vs-nosql-data>.
- [W3S21a] W3Schools. *CSS Tutorial*. 2021. URL: <https://www.w3schools.com/css/default.asp>.
- [W3S21b] W3Schools. *HTML Tutorial*. 2021. URL: <https://www.w3schools.com/html/default.asp>.
- [W3S21c] W3Schools. *JavaScript Tutorial*. 2021. URL: <https://www.w3schools.com/js/default.asp>.



# Anhang

## .1. HTML

Mit der [HTML](#) wird das Grundgerüst einer Internetseite aufgebaut. Dafür werden Textelemente von einem HTML-Tag jeweils geöffnet (`<html>`) und geschlossen (`</html>`). Auch HTML-Tags können Einfluss auf die Position und die Darstellung der Textelemente im Browser ausüben. Zum Beispiel verursacht der HTML-Tag `<h1>Text</h1>`, dass das Textelement als Überschrift angezeigt wird [W3S21b].

Eine HTML-Datei verfügt im Allgemeinen über folgendes Grundgerüst.

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Seitentitel</title>
5     </head>
6     <body>
7         <h1> Das ist eine Grosse Ueberschrift </h1>
8         <p> Das ist ein Absatz </p>
9     </body>
10 </html>
```

Listing 1: Grundgerüst einer HTML-Seite

Der Tag `<!DOCTYPE html>` gibt dem Browser an, dass es sich um eine HTML-Datei handelt. Der `<head>`-Bereich beinhaltet Metadaten über das Webdokument, wie zum Beispiel den Titel, der im Browser-Reiter angezeigt wird. Neben dem Titel können im Head noch weitere Metadaten wie Schlüsselwörter, Autor und Zeichenkodierung angegeben werden. Wird das Aussehen einer Webseite in einer separaten Datei festgelegt, muss diese Datei auch im Head der HTML-Datei verlinkt werden. Die Inhalte, die vom Browser dargestellt werden, sind im `<body>`-Bereich angegeben. Im Code-Beispiel 1 ist das eine Überschrift und ein Absatz und wird, wie in Abbildung .3 dargestellt, im Browser angezeigt.

# Das ist eine Grosse Ueberschrift

Das ist ein Absatz

Abbildung .3.: HTML Grundgerüst

## .1.1. Attribute

HTML-Tags können durch Attribute erweitert werden. Die Attribute werden innerhalb der spitzen Klammern angegeben, siehe Listing 2. Besonders wichtig sind globale Attribute, die für alle HTML-Elemente verwendet werden können. Mithilfe des globalen Attributs `class` können mehrere HTML-Tags in einer Kategorie beziehungsweise Klasse zusammengefasst werden. Die Eigenschaften dieser Klasse können im Anschluss in der CSS-Datei beeinflusst werden, siehe das folgende Kapitel.

```
1 <h1 class="Ueberschrift">
2     Das ist eine Grosse Ueberschrift
3 </h1>
```

Listing 2: HTML Attribute

## .2. CSS

CSS werden verwendet, um

- dem HTML-Dokument einen ansprechenden Stil zuzuweisen,
- das Layout des HTML-Dokumentes zu definieren und
- das Layout so zu gestalten, dass sich der Inhalt automatisch an die Bildschirmgröße anpasst.

Generell gilt, dass mit HTML die Inhalte definiert werden und mit CSS das Aussehen. Die Syntax, um CSS-Eigenschaften für HTML-Elemente zu definieren, ist wie folgt aufgebaut.

```
1 selektor {
2     Eigenschaften : Wert;
3 }
```

Listing 3: Die generelle Syntax für CSS-Eigenschaften

Selektoren können HTML-Elemente wie `<h1>` und `<p>`, oder Klassen und IDs, wie bereits im Kapitel .1 angesprochen, sein. Wird eine Klasse als Selektor verwendet, muss ein Punkt vor den Namen geschrieben werden, bei einer ID eine Raute [W3S21a].

Um die Überschrift aus Listing 2 grün zu färben, würde die CSS-Syntax wie in Listing 4 dargestellt, aussehen. Das Ergebnis ist in Abbildung .4 dargestellt.

```
1  <!-- eine Klasse als Selektor -->
2  .Ueberschrift {
3      color : green;
4  }
5
6  <!-- ein HTML-Element als Selektor-->
7  h1 {
8      color : green;
9  }
```

Listing 4: Grüne Überschrift CSS

## Das ist eine Grosse Ueberschrift

Das ist ein Absatz

Abbildung .4.: Grüne Überschrift

### .2.1. CSS in die HTML-Datei einbinden

Wie im Kapitel .1 bereits erwähnt, ist es möglich, eine CSS-Datei im HTML-Head zu verlinken, siehe Listing 5. Darin wird die Datei `style.css` eingebunden.

```
1 <head>
2     <link rel="stylesheet" href="style.css">
3 </head>
4 <body>
5     <h1 class="Ueberschrift">
6         Das ist eine Grosse Ueberschrift
7     </h1>
8 </body>
```

Listing 5: CSS-Datei in HTML verlinken

Das Attribut `href` gibt den Pfad der verlinkten Ressource an. Die Beziehung des verknüpften Dokuments zum aktuellen Dokument wird mit dem Attribut `rel` angegeben [Mc21]. CSS Deklarationen können auch direkt in das HTML Dokument geschrieben werden, indem man den HTML-Tag `<style>` verwendet, Listing 6.

```
1 <head>
2   <style>
3     .Ueberschrift {
4       color : green;
5     }
6   </style>
7 </head>
8 <body>
9   <h1 class="Ueberschrift">
10    Das ist eine Grosse Ueberschrift
11  </h1>
12 </body>
```

Listing 6: CSS-Datei in HTML einbinden

## 3. JavaScript

JavaScript ist eine Programmiersprache, mit der man komplexe Funktionen für Webseiten realisieren kann. JavaScript findet immer dann Anwendung, wenn eine Internetseite mehr als nur statische Informationen darstellt [Mc21].

Mithilfe von JavaScript können:

- Informationen in Variablen gespeichert,
- Operationen auf Webinhalte, wie zum Beispiel Texte ausgeführt und
- auf Ereignisse reagiert werden, wie zum Beispiel auf einen Maus-Klick.

### 3.1. JavaScript in die HTML-Datei einbinden

JavaScript kann, genauso wie CSS, entweder unter Verwendung des HTML-Tags `<script>` direkt in das HTML-Dokument geschrieben werden, oder der Code wird als externe Datei eingebunden. Dafür wird auch der HTML-Tag `script` verwendet, mit dem Attribut `src1`, und dem Pfad der JavaScript Datei [W3S21c], siehe Listing 7.

---

<sup>1</sup> source (Quelle)

```
1  <head>
2      <!--direkte Einbindung -->
3      <script>
4          // JavaScript Code
5      </script>
6
7      <!-- als externe Datei Einbinden -->
8      <script src="dateiname.js"></script>
9  </head>
```

Listing 7: JavaScript in HTML einbinden