



hochschule mannheim

**Konzipierung und Entwicklung einer
Progressive Web App zum Herunterladen,
Verwalten und Abspielen von Audio-Medien
zur Offlinenutzung mit Angular**

Martin Schalter

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Informatik

Fakultät für Informatik

Hochschule Mannheim

19.08.2020

Betreuer

Prof. Dr. Thomas Specht, Hochschule Mannheim

Christian Perian, biblepool gUG

Schalter, Martin:

Konzipierung und Entwicklung einer Progressive Web App zum Herunterladen, Verwalten und Abspielen von Audio-Medien zur Offlinenutzung mit Angular / Martin Schalter. – Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2020. 57 Seiten.

Schalter, Martin:

Developement of a progressive web app to download, manage and play audio files for offline use with Angular / Martin Schalter. –

Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2020. 57 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 19.08.2020

Martin Schalter

Abstract

Konzipierung und Entwicklung einer Progressive Web App zum Herunterladen, Verwalten und Abspielen von Audio-Medien zur Offlinenutzung mit Angular

Developement of a progressive web app to download, manage and play audio files for offline use with Angular

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Zielsetzung	2
1.3	Aufbau	2
2	Grundlagen	3
2.1	Service Worker	3
2.2	Progressive Web App	4
2.3	Angular und Typescript	5
2.4	Publish Subscribe Pattern	6
2.5	CROSSLOAD	7
3	Anforderungsanalyse	11
3.1	Vorgehensweise und Randbedingungen	11
3.2	Szenarien	14
3.2.1	Predigt im Auto anhören	14
3.2.2	Inhalt für die Reise vormerken	15
3.2.3	Inhalt herunterladen sobald eine WLAN Verbindung besteht	15
3.3	Anforderungen	16
4	Konzeption	19
4.1	Lokale Datenspeicherung	19
4.1.1	Web Storage	19
4.1.2	File System API	20
4.1.3	Web SQL	22
4.1.4	IndexedDB	22
4.1.5	Cache API	23
4.1.6	Maximale Datenmenge	24
4.1.7	Mögliche Architekturen	25
4.2	Herunterladen der Audiodaten	35
4.2.1	Fetch API	35
4.2.2	HttpClient	36
4.2.3	Background Sync	36
4.2.4	Background Fetch	38

4.2.5	Entscheidung	38
4.3	Verbindungsstatus auslesen	39
4.3.1	Online / Offline	39
4.3.2	Verbindungsart	40
5	Implementierung	41
5.1	Offline Metadaten	41
5.1.1	Detailseite	41
5.1.2	Suche	43
5.2	Lokale Datenspeicherung	44
5.3	Herunterladen der Audiodaten	45
5.4	Grafische Oberfläche	45
6	Evaluation und Reflexion	51
6.1	Kann durch die PWA eine native APP ersetzt werden?	51
6.2	Nutzerfreundlichkeit	52
7	Zusammenfassung und Ausblick	55
7.1	Zusammenfassung	55
7.2	Ausblick	56
	Abkürzungsverzeichnis	vii
	Tabellenverzeichnis	ix
	Abbildungsverzeichnis	xi
	Literatur	xiii

Kapitel 1

Einleitung

In diesem Kapitel wird zuerst die Motivation und Zielsetzung der vorliegenden Thesis erläutert, damit der Leser sich einen ersten Eindruck verschaffen kann. Anschließend gibt es einen Überblick über alle nachfolgenden Kapitel.

1.1 Motivation

Das Webportal CROSSLOAD bietet viele christliche Medien zum Download an. Zur Nutzergruppe gehören Leute, die es gewohnt sind mit dem Computer oder Smartphone zu arbeiten, wie zum Beispiel Studenten. Aber auch ältere Menschen, die erst seit ein paar Monaten ein Smartphone besitzen und somit wenig Erfahrung im Umgang mit technischen Geräten haben, gehören zur Zielgruppe. Das Portal wird sowohl von daheim im eigenen WLAN als auch unterwegs mit mobilen Daten genutzt.

Durch die mancherorts schlechte Mobilfunkabdeckung oder das beschränkte Datenvolumen möglicher Nutzer, könnte die Nutzung des Portals an vielen Stellen nicht möglich sein. Bei dem Streamen von Inhalten im Auto oder im Zug könnte es wegen Verbindungsunterbrechungen zu Verzögerungen kommen. Wenn man die Medien unterwegs abrufen möchte, müsste man sie sich vorher herunterladen und anschließend auf dem Gerät suchen. Für einige Nutzer ist das eine zu große Hürde und muss deswegen vereinfacht werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist es für CROSSLOAD ein Konzept zur Offline Nutzung zu entwickeln und anschließend zu implementieren. Inhalte die Offline genutzt werden sollen sind Predigten im Audio-Format, die bis zu 90 Minuten lang sind. Allen Nutzern soll eine komfortable Nutzung des Webportals angeboten werden, sowohl auf Desktop-Computern, Tablets oder Smartphones.

Dafür werden verschiedene Funktionen und Application Programming Interfaces (APIs) von Webbrowsern untersucht und verglichen. Die passendsten Funktionen werden ausgewählt und prototypisch in das vorhandene Portal eingebunden.

Schließlich stellt sich noch die Frage, ob die Funktionen und APIs der Webbrowser ausreichen, um alle Anforderungen zu erfüllen. Ist eine Progressive Web App (PWA) in der Lage für CROSSLOAD eine native App zu ersetzen?

1.3 Aufbau

Zuerst werden die Anforderungen an das Webportal im Blick auf die Offline Nutzung herausgearbeitet. Danach folgt die Konzeption zur Datenspeicherung und zum Herunterladen der Daten. Anschließend wird auf die Implementierung ausgewählter Funktionen eingegangen. In Kapitel 6 folgt die Evaluation und kritische Beurteilung. Zuletzt werden noch einige Funktionen beleuchtet, die in Zukunft umgesetzt werden könnten, um eine noch bessere Nutzerfreundlichkeit zu bieten.

Kapitel 2

Grundlagen

Alle relevanten Grundlagen zum Verständnis dieser Thesis werden in diesem Kapitel erklärt. Dabei geht es um verwendete Begriffe, Technologien sowie Frameworks. Zuletzt wird CROSSLOAD, das zu erweiternde Webportal beschrieben und gezeigt.

2.1 Service Worker

Service Worker laufen in einem Browser und können JavaScript ausführen (Sheppard 2017). Sie sind im Hintergrund aktiv, sogar wenn die Webseite geschlossen wurde, haben aber keinen Zugriff aufs Document Object Model (DOM) (Sheppard 2017). Ein Service Worker ist wie ein Mittler zwischen App und Internet, dabei wird meistens auf die Cache API zurückgegriffen (Sheppard 2017). Abbildung 2.1 zeigt die Architektur von Service Workern. Wenn die Website eine Netzwerkanfrage schickt kann der Service Worker diese Anfrage abfangen. Die abgefangene Abfrage kann nun verändert werden, weitergeleitet werden oder mithilfe der Anfrage eine Antwort aus dem Cache geholt werden.

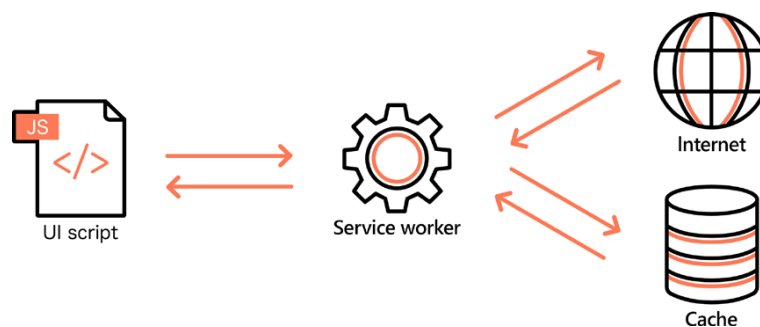


Abbildung 2.1: Service Worker Architektur (Sheppard 2017)

2.2 Progressive Web App

Eine PWA ist eine Anwendung, die in modernen Webbrowsern läuft und dem Nutzer dabei eine Erfahrung ähnlich zu bekannten nativen Apps bietet (Sheppard 2017) (Rojas 2020). Die wichtigsten Funktionen dabei sind:

- **Schnell:** Die App startet schnell und bietet schnell die Möglichkeit zur Interaktion (Hajian 2019) (Sheppard 2017)
- **Zuverlässig:** auf älteren Geräten funktionsfähig und auch ohne Internetverbindung nutzbar. Außerdem passt sich das Design an verschiedene Bildschirmgrößen an (responsive design) (Hajian 2019) (Sheppard 2017)
- **Installierbar:** nach dem Installieren wird ein Icon auf dem Homescreen angezeigt (Hajian 2019) (Sheppard 2017) (Rojas 2020)
- **Benachrichtigungen:** der Nutzer kann Benachrichtigungen empfangen, obwohl er die App nicht offen hat (Hajian 2019) (Sheppard 2017). Eine Messenger-App könnte zum Beispiel eine Benachrichtigung schicken, sobald der Nutzer eine Nachricht bekommt. Dafür ist es nicht erforderlich, dass der Bildschirm an ist oder die App geöffnet ist.
- **Native-like Funktionen:** Zugriff auf Hardware wie zum Beispiel die Kamera (Hajian 2019)

Bei der Nutzung bzw. der Entwicklung einer PWA entstehen viele Vorteile für die Entwickler und Nutzer. Webtechnologien sind weit verbreitet und für jedes Betriebssystem sind moderne Browser verfügbar, dadurch kann die Anwendung sehr leicht an eine große Nutzergruppe ausgeliefert werden (Rojas 2020). Das Ausliefern an die Nutzer ist auch sehr einfach, weil diese nur eine URL benötigen (Khan, Al-Badi und Al-Kindi 2019) und eine neue Version der App automatisch beim Start der App heruntergeladen wird (Rojas 2020). Außerdem ist die Entwicklung mit Webtechnologien weit verbreitet, man findet viele Ressourcen und Tools, die bei der Entwicklung helfen (Rojas 2020). Beim Vergleichen von nativen Apps zu einer PWA fällt zudem auf, dass die PWA sehr wenig Speicherplatz nach der Installation benötigt (Biørn-Hansen, Majchrzak und Grønli 2017) (Khan, Al-Badi und Al-Kindi 2019). Die Geschwindigkeit einer PWA auf Android kann im Vergleich zu anderen Cross-Plattform Ansätzen mithalten oder ist sogar schneller (Biørn-Hansen, Majchrzak und Grønli 2017).

Eine PWA hat nicht nur Vorteile, sondern auch Nachteile. Einer davon ist die Limitierung auf die Funktionen die der Webbrowser zur Verfügung stellt. Über den Webbrowser ist es zum Beispiel nicht möglich Sensoren wie den Beschleunigungssensor auszulesen oder direkt auf das Dateisystem zuzugreifen. Zum Tracken von Fitnessaktivitäten ist eine PWA auch nicht geeignet, weil diese nur eine kurze Zeit im Hintergrund laufen kann. Hinzu kommt, dass einige Funktionen in den Webbrowsern noch nicht standardisiert sind und / oder noch nicht von allen gängigen Browsern unterstützt werden (Majchrzak, Biørn-Hansen und Grønli 2018) (Biørn-Hansen, Majchrzak und Grønli 2017). Wenn die Performance der App eine große Rolle spielt, wie etwa bei Spielen, ist eine PWA nicht sehr gut geeignet (Biørn-Hansen, Majchrzak und Grønli 2017).

Damit aus einer Website eine PWA wird sind mindestens zwei Dinge erforderlich. Erstens wird eine Manifest-Datei benötigt, die Informationen über das Icon, den Namen und vielen mehr enthält, um die App installieren zu können (Hajian 2019) (Rojas 2020). Als zweites wird ein Service Worker benötigt, der unter anderem für die Offline-Fähigkeit verantwortlich ist (Rojas 2020). Der Service Worker cached alle nötigen Dateien einer Website und stellt diese zur Verfügung wenn keine Internetverbindung besteht (Rojas 2020).

2.3 Angular und Typescript

Angular ist ein Framework zum Entwickeln von Webanwendungen für alle Plattformen (Google LLC 2020). Es ist Open-Source, getrieben von einer großen Community aber auch von Firmen wie Google weiterentwickelt und selbst viel genutzt (Google LLC 2020). Angularanwendungen bestehen unter anderem aus Modulen, Komponenten und Services (*Angular Getting started* 2020). Module fassen mehrere Komponenten zusammen, um diese zu verwalten (*Angular Getting started* 2020). Komponenten sind kleine Bestandteile der Anwendung, die sich vor allem um die Darstellung eines bestimmten Bereichs auf der Seite kümmern (*Angular Getting started* 2020). In Services liegt die Logik der Anwendung (*Angular Getting started* 2020). Dort werden zum Beispiel Netzwerkanfragen gesendet oder Daten zwischengespeichert. Als Programmiersprache ist Typescript vorgesehen.

Typescript erweitert Javascript um einige Funktionen wie zum Beispiel Typisierung (Microsoft 2020b). Die Flexibilität von Javascript bleibt trotzdem erhalten, weil der

Nutzer selbst entscheiden kann wann er typisieren möchte oder nicht (Microsoft 2020b). Typescript kann vor dem Ausliefern in reines Javascript kompiliert werden und ist somit in allen gängigen Browsern oder auch auf Servern mit Node.js ausführbar (Microsoft 2020b).

Die Kombination aus Angular und Typescript bietet eine sehr gute Basis zum Entwickeln von großen, schnellen und skalierbaren Anwendungen (Google LLC 2020). Durch die statische Typisierung sind zum Beispiel Refactorings mit den vielen verfügbaren Tools schnell und sicher durchzuführen (Microsoft 2020b) (Google LLC 2020). In der großen Community findet man auf sehr viele Fragen sofort eine Antwort.

2.4 Publish Subscribe Pattern

In Angular wird sehr oft mit Observables gearbeitet und damit das Publish-Subscribe Pattern oder auch Observer-Pattern genannt umgesetzt.

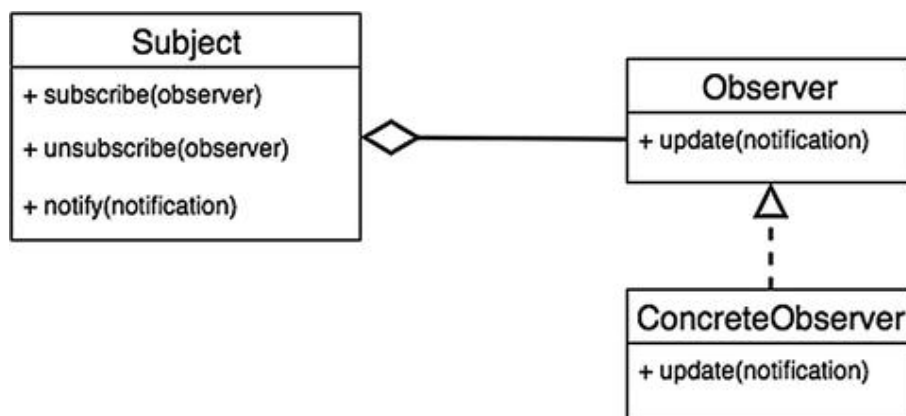


Abbildung 2.2: Publish-Subscribe Pattern UML (Mezzalira 2018)

Abbildung 2.2 zeigt den Aufbau des Publish-Subscribe Pattern. Es gibt ein *Subject* welches eine Liste an Observern hält (Mezzalira 2018). Ein Observer kann über *subscribe(observer)* hinzugefügt werden und über *unsubscribe(observer)* auch wieder entfernt werden (Mezzalira 2018). Sobald beim Subject die Methode *notify(notification)* aufgerufen wird, wird bei allen Observern die *update(notification)* Methode aufgerufen (Mezzalira 2018). Üblicherweise gibt es ein Interface Observer und dazu mindestens eine Implementierung (Mezzalira 2018).

Dieses Pattern bietet den Vorteil, dass mehrere Werte an mehrere Subscriber gesendet werden können.

2.5 CROSSLOAD

CROSSLOAD ist ein Webportal das christliche Medien, im Moment hauptsächlich Predigten, zur Verfügung stellt (biblepool gUG 2020). Diese Medien können durchsucht, angehört und heruntergeladen werden. Die Entwicklung befindet sich im Moment in der Beta-Phase (<https://beta.crossload.org>) und wird vor allem von Ehrenamtlichen vorangetrieben. In Abbildung 2.4 ist die mobile Seite mit den neuesten Inhalten auf CROSSLOAD zu sehen. Abbildung 2.5 zeigt die Detailansicht einer Predigt.

Im Frontend wird Angular mit Typescript zum Entwickeln verwendet und das Backend besteht aus mehreren Services, die mit unterschiedlichen Sprachen und Frameworks entwickelt werden. Abbildung 2.3 zeigt die Architektur. Das Frontend ist in drei Teile aufgeteilt:

- Components zeigen dem Nutzer Inhalte an. In Components werden keine Berechnungen durchgeführt oder Daten manipuliert
- Pages sind für die Datenaufbereitung zuständig, bekommen Daten von Services und geben Daten an Components weiter
- Services sind die Schnittstelle zu externen APIs um Daten abzurufen

Viele Bilder von CROSSLOAD werden von Unsplash zur Verfügung gestellt. Für die Suche wird Solr verwendet und Dateien werden in Amazon S3 abgelegt.

Das Webportal erfüllt bereits alle Anforderungen einer PWA: Es besitzt eine Manifest-Datei und kann somit auf den Startbildschirm des Smartphones hinzugefügt werden. Statische Inhalte werden mit Hilfe eines Service Workers gecached und sind somit auch offline verfügbar.

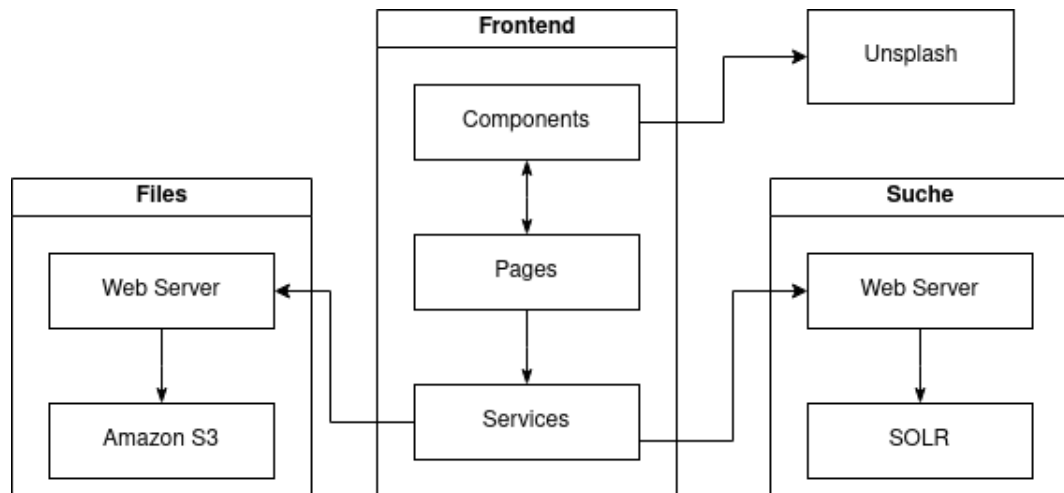


Abbildung 2.3: Architektur von CROSSLOAD

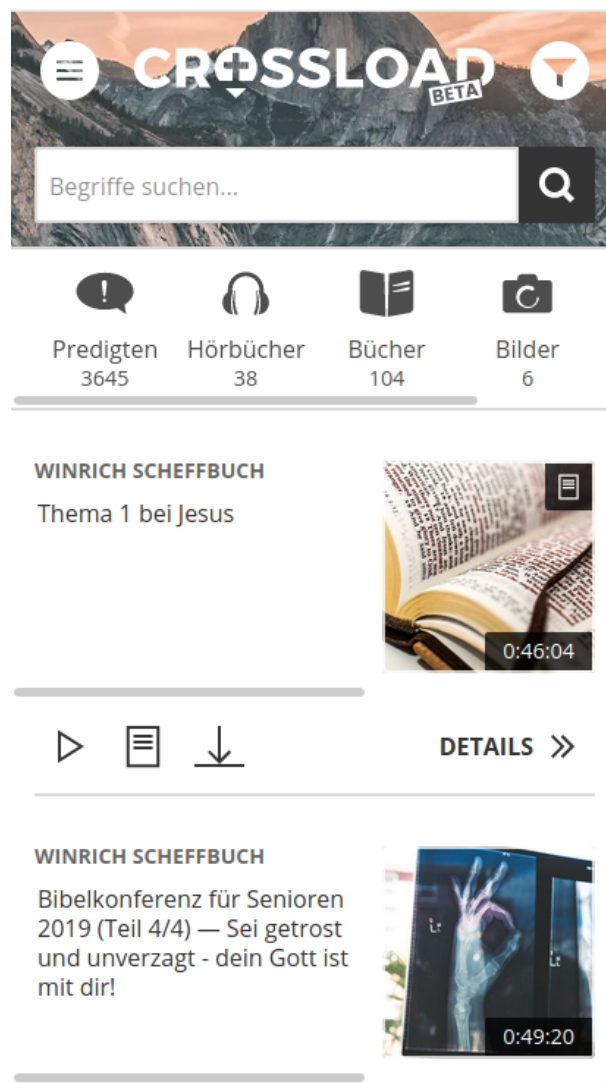



Abbildung 2.4: CROSSLOAD - Mobile Ansicht der neusten Inhalte


WINRICH SCHEFFBUCH

Thema 1 bei Jesus


14.04.2019


 Markus 1,15


▼




0:00:00 | 0:46:04

 Download

 Feedback

 Teilen

 Details

Downloads

→ MITSCHRIFT

▼

→ DATEIEN

▼

Mitschrift




Abbildung 2.5: CROSSLOAD - Detailansicht einer Predigt

Kapitel 3

Anforderungsanalyse

Die Anforderungen die in diesem Kapitel herausgearbeitet werden, bilden die Grundlage für die Entscheidungen in den nächsten Kapiteln. Zuerst werden grundlegende Bedingungen und die Vorgehensweise erklärt. Anschließend werden einige Szenarien zur Benutzung der Offline-Funktion beschrieben. Diese sollen beim Verstehen der Anforderungen, die im darauf folgenden Kapitel aufgelistet werden, helfen.

3.1 Vorgehensweise und Randbedingungen

Da CROSSLOAD fast nur ehrenamtliche Mitarbeiter hat, gibt es niemanden der genaue Anforderungen an Features vorschreibt. Vielmehr werden die Anforderungen gemeinsam im Team erarbeitet und diskutiert. In einem ersten Brainstorming wurden die gewünschten Funktionalitäten anhand einiger Beispielszenarien diskutiert und dokumentiert. In den Kapiteln 3.2 und 3.3 werden diese näher ausgearbeitet und beschrieben.

Es gibt zwei Randbedingungen, die Weiterentwicklung der PWA relevant sind:

Erstens muss das Framework Angular benutzt werden, weil das bisherige Webportal auch in Angular geschrieben ist. Eine andere Technologie zu verwenden, würde die Integration in die bestehende Plattform sehr schwer gestalten.

Als zweites stellt sich die Frage welche Browser unterstützt werden müssen. CROSSLOAD befindet sich in der Beta-Phase und hat noch keine große Nutzerbasis. Für die ersten zwanzig Tage im Juni wurden 800 Besucher gezählt. Die Verteilung der Browser dieser 800 Nutzer ist in Abbildung 3.1 und Abbildung 3.2 zu sehen. Auf

Mobilgeräten dominieren eindeutig Safari und Chrome mit einem Anteil von über 90%. Auf Desktopgeräten teilt sich die Nutzung fast gleichmäßig zwischen Safari, Firefox und Chrome auf. Chrome hat etwas mehr Anteil und Firefox etwas weniger. Der Browser Edge wird noch sehr selten benutzt.

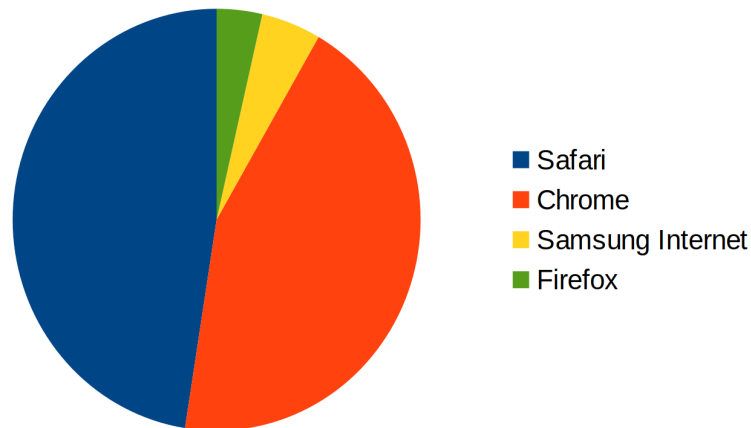


Abbildung 3.1: Nutzung von CROSSLOAD nach Webbrowser auf Mobilgeräten

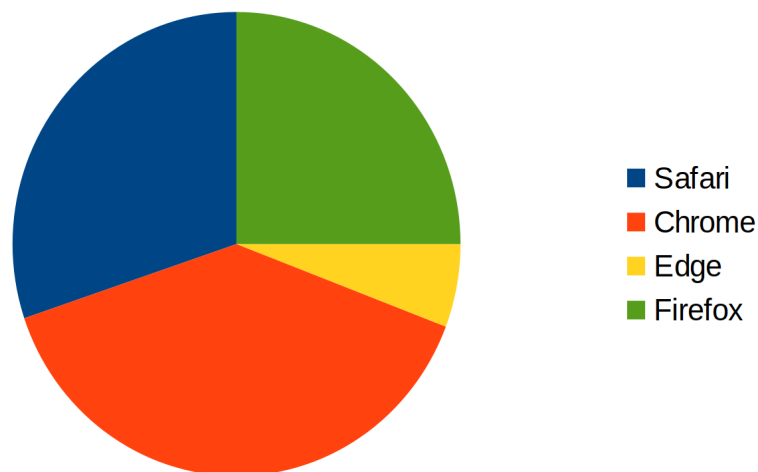


Abbildung 3.2: Nutzung von CROSSLOAD nach Webbrowser auf dem Desktop

Da diese Daten noch nicht sehr aussagekräftig sind, weil eine zu geringe Nutzerbasis vorhanden ist, wird die durchschnittliche Browsernutzung in Deutschland betrachtet. Dabei kann zwischen der Nutzung der Browser auf PCs wie in Abbildung 3.4 zu sehen und auf Mobilgeräten wie in Abbildung 3.3 zu sehen unterschieden werden.

Der führende Browser auf Mobilgeräten ist Chrome mit über 50% Marktanteil. Gefolgt von Safari und Samsung Internet. Andere Browser spielen eine sehr untergeordnete Rolle. Im Vergleich zu den Daten von CROSSLOAD ist Samsung Internet öfter genutzt dagegen Safari weniger oft.

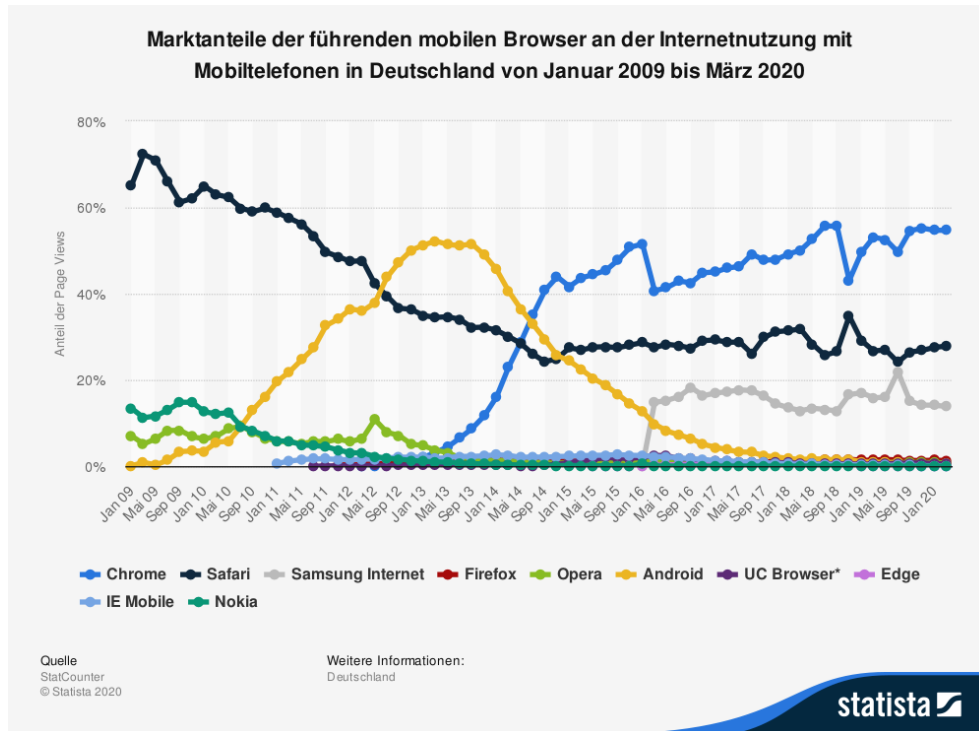


Abbildung 3.3: Internetnutzung nach mobilen Browsern in Deutschland (StatCounter 2020b)

Der am häufigst genutzte Browser auf dem PC in Deutschland ist Chrome mit über 40%. Firefox steht an Platz zwei. Safari erreicht noch einen Wert von 10%. Zu beobachten ist auch der Abwärtstrend vom Internet Explorer, der die letzten Jahre viel Marktanteile verlor. In den Daten von CROSSLOAD hat Safari mehr Anteile und Chrome etwas weniger.

Auch wenn die Offline-Funktionen des Webportals für dessen Nutzung nicht essenziell sind, sollen diese möglichst vielen Nutzern zur Verfügung stehen. Dies bedingt zumindest die Unterstützung der am weitesten verbreiteten Web Browser Google Chrome, Firefox, Safari und Samsung Internet auf Android und iOS. Dabei wird nur die aktuelle Version des jeweiligen Browsers unterstützt, weil alle gängigen Browser über eine automatische Updatefunktion verfügen. Dadurch erhalten die meisten Nutzer sehr zeitnah die neuste Version.

Samsung Internet basiert intern auf Chromium und teilt somit die allermeisten Funktionen mit Chrome (Appelquist 2019). Für den Edge-Browser wurde bereits 2018 angekündigt Chromium als Basis zu verwenden (Microsoft 2020a). In späteren Entscheidungen wird deswegen Samsung-Internet und Edge nur erwähnt wenn es von der Funktionalität im Chrome Browser abweicht.

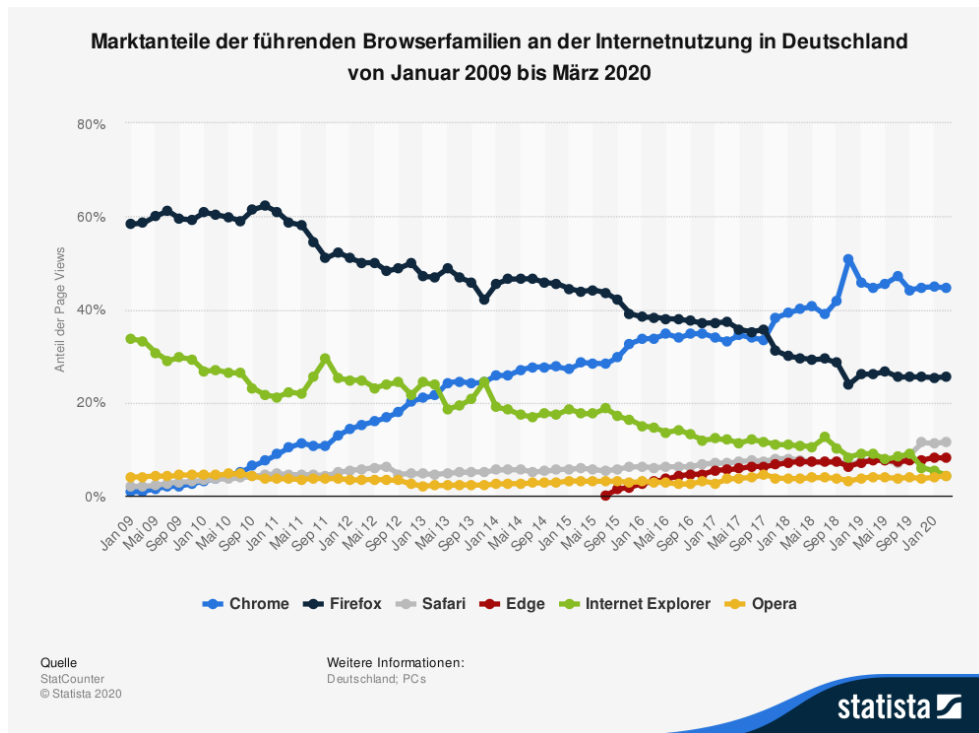


Abbildung 3.4: Internetnutzung auf PCs nach Browsern in Deutschland (StatCounter 2020a)

Plattform	unterstützte Browser
Android	Chrome, Samsung Internet
iOS	Safari, Chrome
Windows, Mac, Linux	Chrome, Firefox, Edge

Tabelle 3.1: Unterstützte Browser und Plattformen

In Tabelle 3.1 ist zusammengefasst welche Browser unter welchen Plattformen unterstützt werden sollen.

3.2 Szenarien

Jedes Szenario beschreibt eine konkrete Interaktion mit dem Webportal ohne Sonderfälle abzubilden. Sie dienen dazu die Anforderungen besser zu verstehen. Im folgenden werden drei Szenarien beschrieben.

3.2.1 Predigt im Auto anhören

Akteur: Benutzer

Ablauf:

1. Benutzer ist zu Hause und durchsucht Inhalte auf CROSSLOAD
2. Benutzer favorisiert sich mehrere Inhalte
3. Der Download der Inhalte startet
4. Sobald der Download fertig ist, wird das dem Benutzer angezeigt
5. Benutzer geht außer Haus in sein Auto
6. Benutzer bekommt alle favorisierten Inhalte angezeigt
7. Benutzer wählt einen favorisierten Inhalt aus und sieht die Übersichtsseite des Inhalt
8. Benutzer hört sich die Predigt an und benötigt dafür kein Internet

3.2.2 Inhalt für die Reise vormerken

Akteur: Benutzer

Ablauf:

1. Benutzer ist am Flughafen und steht kurz vor einem Flug
2. Benutzer favorisiert sich einen Inhalt auf CROSSLOAD
3. CROSSLOAD fragt den Benutzer, ob er den Inhalt auch über mobiles Internet herunterladen möchte
4. Der Benutzer bestätigt diese Anfrage
5. Der Download beginnt
6. Sobald der Download fertig ist, wird das dem Benutzer angezeigt
7. Benutzer ist im Flugzeug und hört sich die favorisierte Predigt an

3.2.3 Inhalt herunterladen sobald eine WLAN Verbindung besteht

Akteur: Benutzer

Ablauf:

1. Benutzer ist unterwegs und bekommt einen Inhalt auf CROSSLOAD empfohlen
2. Benutzer favorisiert diesen Inhalt
3. CROSSLOAD fragt den Benutzer, ob er den Inhalt auch über mobiles Internet herunterladen möchte
4. Der Benutzer verneint diese Anfrage
5. Der Benutzer kommt nach Hause und ist mit dem eigenen WLAN verbunden
6. Der Download der favorisierten Predigt beginnt
7. Sobald der Download fertig ist, wird das dem Benutzer angezeigt
8. Der Benutzer hört sich die favorisierte Predigt an. Obwohl er eine WLAN-Verbindung besitzt, werden die heruntergeladenen Inhalte zum Abspielen der Predigt genutzt

3.3 Anforderungen

Aus den Diskussionen mit Mitarbeitern von CROSSLOAD haben sich die Anforderungen herausgestellt, die in diesem Kapitel aufgelistet sind. Für diese Thesis sind nur Audioinhalte auf CROSSLOAD relevant.

1. Die Länge der Audioinhalte liegt zwischen wenigen Minuten bis zu 90 Minuten. Eine Audiodatei ist bis zu 100 Megabyte (MB) groß.
2. Der Benutzer kann sich einen Inhalt vormerken, auch favorisieren genannt. Dabei gibt es keine Begrenzung wie viele Inhalt insgesamt favorisiert werden können.
3. Der vorgemerkte Inhalt wird automatisch anhand der Verbindungsart heruntergeladen. Der genaue Ablauf ist in einem Aktivitätsdiagramm in Abbildung 3.5 zu sehen.
 - Wenn eine WLAN-Verbindung besteht wird der Download sofort gestartet
 - Wenn eine mobile Datenverbindung besteht wird der Benutzer gefragt, ob er den Inhalt jetzt herunterladen möchte

- Wenn der Benutzer dies verneint wird auf eine WLAN-Verbindung gewartet und der Download gestartet, sobald diese besteht

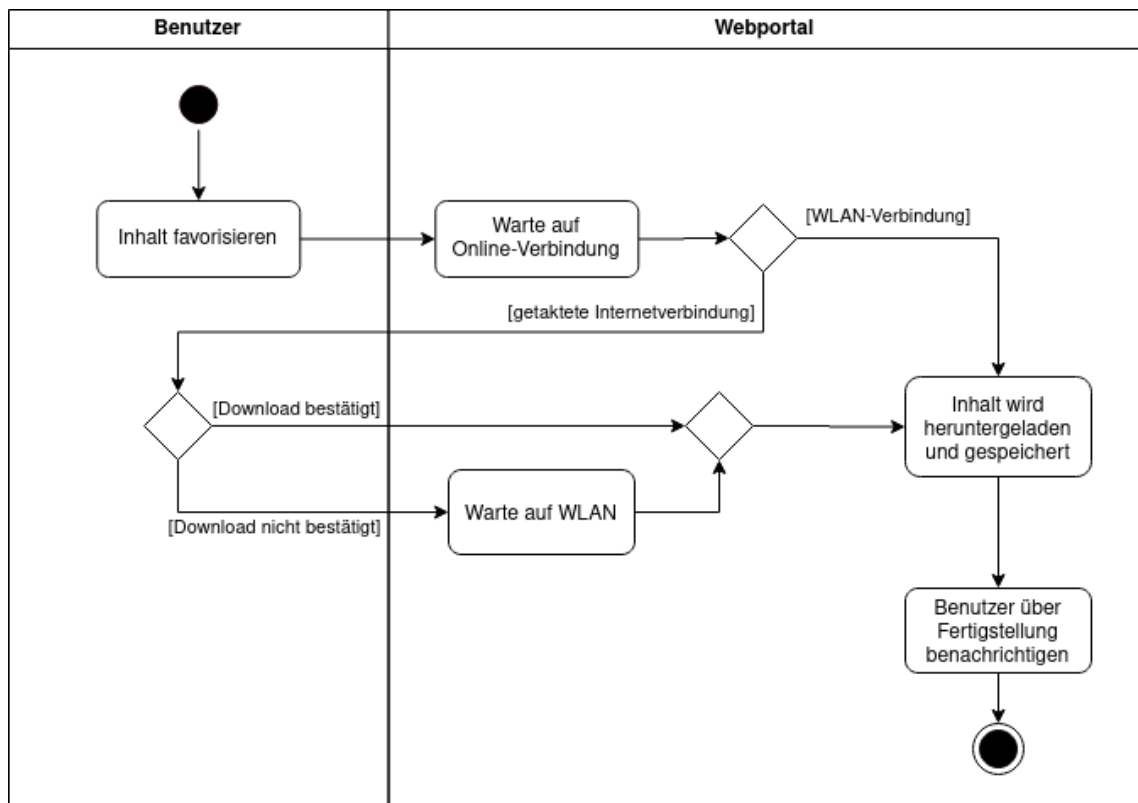


Abbildung 3.5: Aktivitätsdiagramm: Download des Inhalt abhängig von der Verbindungsart

4. Ein Download findet im Hintergrund statt ohne die Bedienung des Portals einzuschränken wie zum Beispiel durch einfrieren der Oberfläche
5. Der Benutzer wird über den Fortschritt des Downloads informiert. Folgende Status sind möglich: gestartet, wird heruntergeladen, heruntergeladen, Fehler
6. Das Portal erkennt wenn die Internetverbindung unterbrochen wird und zeigt dem Benutzer einen Hinweis an
7. Die zurzeit heruntergeladenen Inhalte können in einer Übersicht gesehen werden
8. Die Inhalte sollen gefiltert oder sortiert werden könnten. Zum Beispiel nach Download-Datum
9. Die zurzeit heruntergeladenen Inhalte können gelöscht werden. Einzeln oder alle auf einmal

10. Es soll möglich sein für einen bestimmten Inhalt herauszufinden ob dieser Offline verfügbar ist
11. Die Inhalte der Detailseite eines heruntergeladenen Inhalts werden gespeichert. Dazu gehören zum Beispiel: Id, Titel, Autor oder Vorschaubild. Diese Daten sind pro Inhalt bis zu 50 Kilobyte (kB) groß.
12. Der heruntergeladene Inhalt kann angehört werden
 - Wenn ein Inhalt angehört werden soll, der bereits heruntergeladen ist, sollen die heruntergeladenen Daten verwendet werden, um Netzwerkverkehr zu vermeiden

Abbildung 3.6 zeigt alle Use Cases in einem Diagramm. Netzwerk-Service steht für eine API zum Herunterladen eines Inhalts. Welche API das sein wird, wird in den nächsten Kapiteln erarbeitet. Auch welche API zur Datenverwaltung benutzt wird steht noch nicht fest. Inhalte können vorgemerkt und dadurch heruntergeladen werden. Außerdem gespeichert, abgerufen, durchsucht, gelöscht und angehört werden.

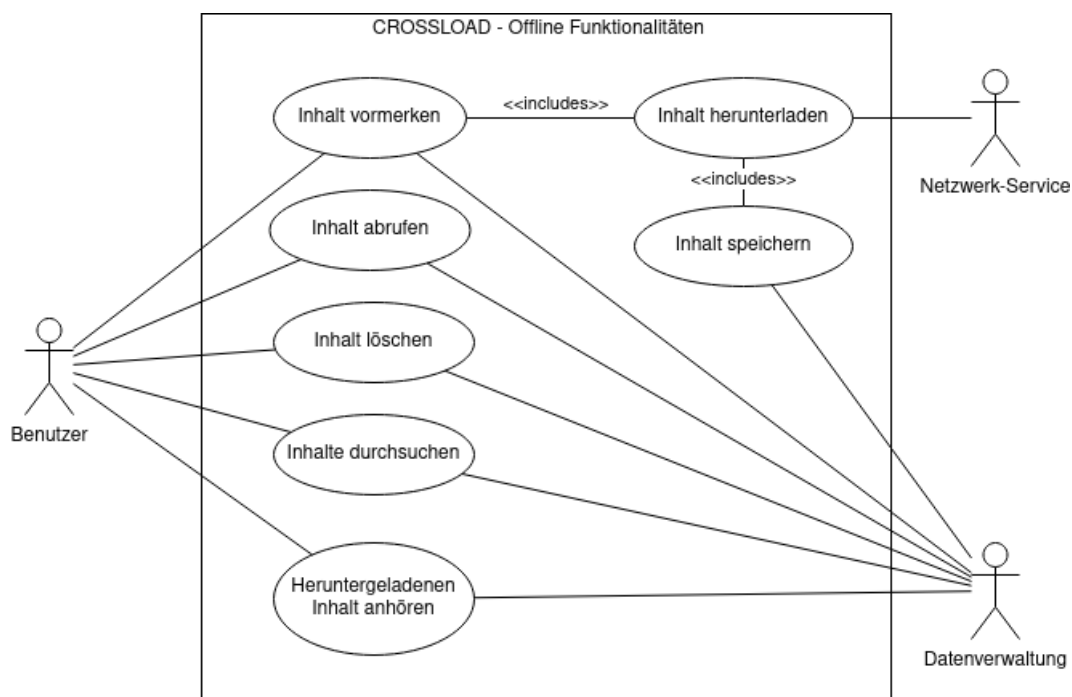


Abbildung 3.6: Use Case Diagramm der Offline-Funktionalitäten von CROSSLOAD

Kapitel 4

Konzeption

4.1 Lokale Datenspeicherung

Moderne Webbrowser stellen viele verschiedene Lösungen zum lokalen Speichern von Daten zu Verfügung. In diesem Kapitel werden diese vorgestellt und miteinander verglichen. Zuletzt wird eine Entscheidung für CROSSLOAD getroffen.

Vergleichskriterien zur lokalen Datenspeicherung ergeben sich aus den Anforderungen. Hinter jedem Kriterium steht in Klammern die Nummer der Anforderung aus Abschnitt 3.3:

1. Ist die API asynchron oder nur synchron nutzbar? Eine nur synchron nutzbare Lösung könnte zum Einfrieren der Nutzeroberfläche führen (Anforderung 4)
2. Maximal zulässige Speichermenge (Anforderung 1, 2, und 11)
3. Mögliche Datentypen. Können Bilder, Audiodateien und Text abgespeichert werden? (Anforderung 1 und 11)
4. Wie gut wird das Durchsuchen, Filtern und Löschen von Inhalten unterstützt? (Anforderung 7, 8, 9 und 10)
5. Welche Browser unterstützen diese Funktionalitäten?

4.1.1 Web Storage

Die Web Storage API bietet die Möglichkeit Schlüssel / Wert Paare im Browser zu speichern (*Web Storage API* 2020). Werte werden immer mit einem Schlüssel gespeichert oder über einen Schlüssel geladen (*Web Storage API* 2020).

Alle relevanten Browser implementieren diese API (*Web Storage API* 2020), es gibt aber auch einige Einschränkungen:

- Zugriffe sind nur synchron möglich (Hajian 2019). Das heißt: wenn große Datenmengen gespeichert oder geladen werden, kann das Browserfenster für eine Zeit lang einfrieren
- Als Schlüssel und Werte können jeweils nur Strings gespeichert werden (Hajian 2019). Sollen JavaScript-Objekte gespeichert werden müssen diese in JSON umgewandelt werden.
- Es können maximal 5 MB an Daten gespeichert werden (*Web Storage API* 2020)

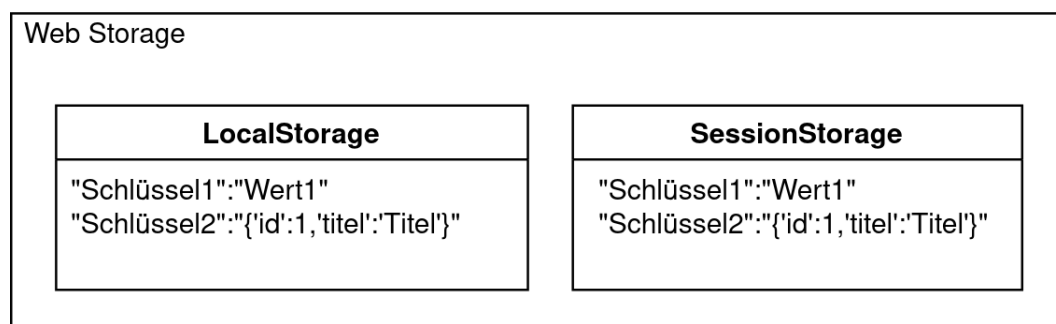


Abbildung 4.1: Aufbau der Web Storage API

Abbildung 4.1 zeigt die Web Storage API mit SessionStorage und LocalStorage. Diese beiden Speicher unterscheiden sich nur darin wie lange Daten gespeichert werden (Hajian 2019) (*Web Storage API* 2020). Die Daten vom SessionStorage werden gelöscht sobald die Sitzung auf der Website vorbei ist, das heißt der Browser oder der Tab geschlossen wird (Hajian 2019) (*Web Storage API* 2020). Der LocalStorage bleibt über mehrere Sitzungen erhalten und wird nicht automatisch gelöscht (Hajian 2019) (*Web Storage API* 2020). In beiden Speichern können Schlüssel-Wert Paare als Text gespeichert werden (Hajian 2019) (*Web Storage API* 2020).

4.1.2 File System API

Die File System API und FileWrite API bieten dem Browser die Möglichkeit Dateien in ein virtuelles Dateisystem abzulegen und von dort wieder zu laden (Hajian 2019) (LePage 2020). Das Mozilla Developer Network (MDN) nennt diese API Fi-

le and Directory Entries API (*Introduction to the File and Directory Entries API* 2019).

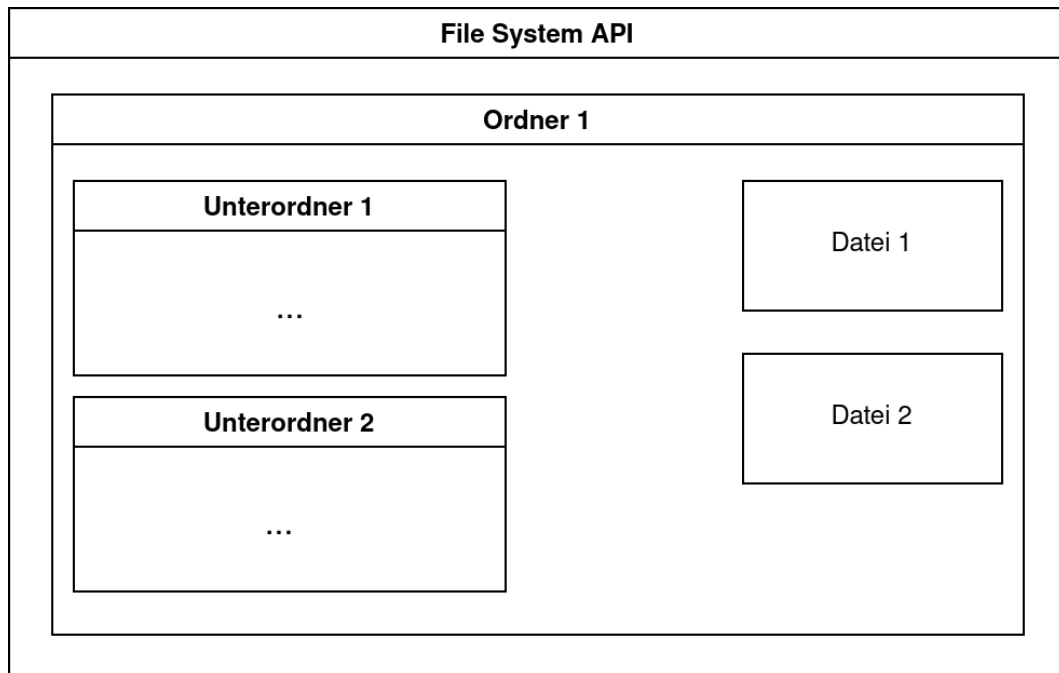


Abbildung 4.2: Struktur der File System API

In Abbildung 4.2 ist die Struktur der File System API zu sehen. Es gibt Ordner und Dateien. Ein Ordner kann Dateien und Unterordner enthalten. Die API erlaubt eine Navigation in den Ordnern, wie zum Beispiel: gehe in den übergeordneten Ordner (*Introduction to the File and Directory Entries API* 2019). Außerdem lassen sich alle Dateien und Unterordner eines Ordners auflisten (*Introduction to the File and Directory Entries API* 2019). Dateien können geladen und gespeichert werden (*Introduction to the File and Directory Entries API* 2019). Diese Dateien und Ordner liegen nur in einem virtuellen Dateisystem und sind in dieser Ordnerstruktur nicht auf der Festplatte des Benutzers wiederzufinden (*Introduction to the File and Directory Entries API* 2019).

Der Vorteil dieser API ist der Umgang mit binary large objects (blobs), wie zum Beispiel Audiodateien. Diese können leicht gespeichert, geladen und manipuliert werden (*Introduction to the File and Directory Entries API* 2019). Es können ebenso andere Datentypen wie Strings gespeichert werden (*Introduction to the File and Directory Entries API* 2019). Außerdem gibt es eine API für synchrone und asynchrone Zugriffe (Hajian 2019).

Es gibt keinen offiziellen Standard (*Introduction to the File and Directory Entries API* 2019) (*Filesystem & FileWriter API* 2020), deswegen unterstützen noch nicht sehr viele Browser diese API. Bisher ist in Chrome und in allen Chromium basierten Browser diese Funktion verfügbar (*Filesystem & FileWriter API* 2020).

4.1.3 Web SQL

Web SQL ist eine API, die es erlaubt Daten in einer Datenbank zu speichern und diese Daten mit einer SQL ähnlichen Sprache zu durchsuchen (Hickson 2010). Alle Anfragen sind asynchron (Hajian 2019).

Diese API wurde nie von allen Browsern implementiert und ist mittlerweile deprecated, soll also nicht mehr verwendet werden (Hajian 2019).

Aufgrund der Tatsache, dass Web SQL deprecated ist, wird diese Technologie auch keinen Einsatz bei CROSSLOAD finden und hier nicht weiter erläutert.

4.1.4 IndexedDB

IndexedDB ist eine key-value NoSQL objekt-orientierte Datenbank zum Speichern von großen und vielen Dateien (Hajian 2019). Konkret heißt das, dass Objekte mit einem Schlüssel in die Datenbank abgelegt werden können und diese über den angegebenen Schlüssel wieder auffindbar sind. Dabei werden viele unterschiedliche Datentypen (boolean, number, string, date, object, array, regexp, undefined, null, blob) (*IndexedDB Grundlagen* 2020), Transaktionen und Indexe zum schnelleren Durchsuchen und Filtern unterstützt (Sheppard 2017).

Abbildung 4.3 zeigt den Aufbau von IndexedDB. Es können mehrere Datenbanken erstellt werden und jede Datenbank hat mehrere Object stores (*IndexedDB Grundlagen* 2020). Ein Object store ist mit einer Tabelle in einer relationalen Datenbank vergleichbar. Transaktionen sind innerhalb einer Datenbank über mehrere Object stores möglich (*IndexedDB Grundlagen* 2020). Jeder Object store kann key-value Paare speichern (Sheppard 2017). Als Werte können JavaScript Objekte, Zahlen, Texte, Bilder, Audiodateien und vieles mehr gespeichert werden (*IndexedDB Grundlagen* 2020). Wenn Werte JavaScript Objekte sind kann auf ein Attribut ein Index gelegt werden (*IndexedDB Grundlagen* 2020). In diesem Beispiel könnte das Attribut *id* oder *titel* oder auch beide indiziert werden.

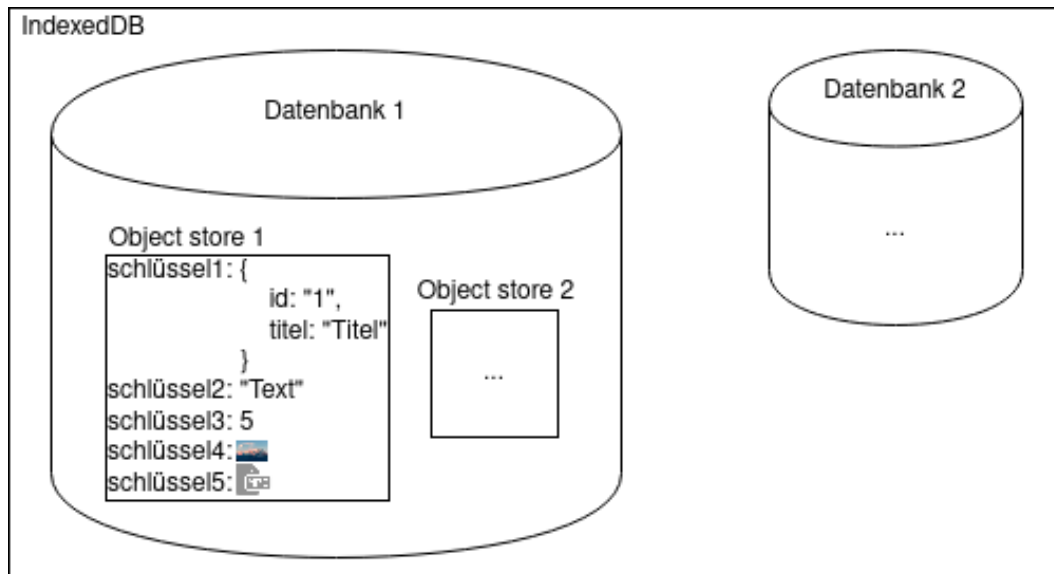


Abbildung 4.3: Architektur der IndexedDB

IndexedDB kann asynchron verwendet werden (Hajian 2019) (*IndexedDB Grundlagen* 2020) und alle relevanten Browser unterstützen die API (*IndexedDB* 2020). Die maximale Speicherkapazität der Datenbank hängt vom Browser ab und wird in Kapitel 4.1.6 näher behandelt.

Anfragen an die Datenbank können direkt eine Sortierung oder Suchkriterien enthalten (*IndexedDB Grundlagen* 2020). Dabei ist zu beachten, dass die Textsuche limitiert ist und keine Anfragen zum Finden eines einzelnen Wortes in einem ganzen Text unterstützt (*IndexedDB Grundlagen* 2020).

Das Programmieren mit der API von IndexedDB ist komplex, weil mit Events gearbeitet wird und nicht wie in modernen APIs mit Promises (Hajian 2019). Deswegen existieren viele Bibliotheken, die eine Verwendung der API erleichtern, wie zum Beispiel LocalForage oder Dexie.js (Hajian 2019) (*IndexedDB Grundlagen* 2020).

4.1.5 Cache API

Die Cache API bietet die Möglichkeit Netzwerkanfragen zwischenspeichern (*Cache - Web APIs* 2020). Über Request-Objekte können Response-Objekte gespeichert und geladen werden (*Cache - Web APIs* 2020).

Abbildung 4.4 zeigt den Aufbau der Cache API: Es können mehrere Caches angelegt werden (*Cache - Web APIs* 2020). In einem Cache können mehrere Request-

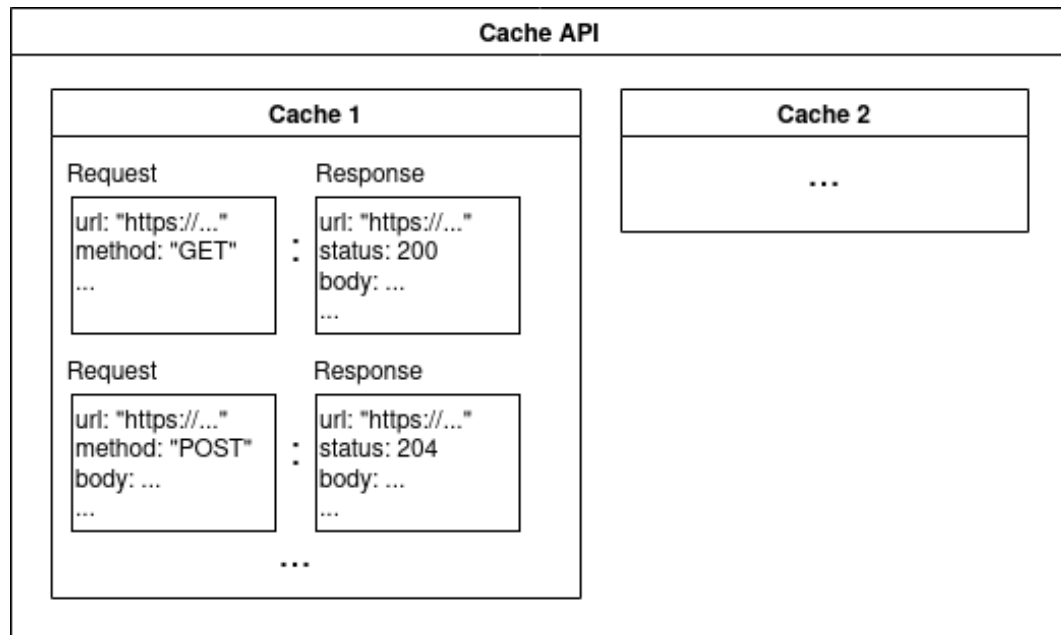


Abbildung 4.4: Aufbau der Cache API

Response Paare gespeichert werden (*Cache - Web APIs* 2020). Ein Request-Objekt enthält die URL, die Methode (GET, POST, PUT, ...), einen body und optional noch mehr Parameter (*Request* 2020). Das Response-Objekt ist ähnlich aufgebaut, enthält aber zum Beispiel auch einen Statuscode (*Response* 2020).

Die API ist in allen relevanten Browsern verfügbar (*Cache - Web APIs* 2020). Oft wird die Cache API in Verbindung mit Service Workern verwendet. Die maximale Speicherkapazität des Caches hängt vom Browser ab und wird in Kapitel 4.1.6 näher erläutert.

4.1.6 Maximale Datenmenge

Es gibt keine einheitliche Regelung wie viel Speicher von einer Webseite genutzt werden darf, deswegen hat jeder Browser seine eigenen Regeln (LePage 2020).

Chrome und Firefox verfolgen die gleiche Strategie beim vergeben von Speicherplatz: Es gibt einen shared pool, den sich alle Websites teilen (LePage 2020) (*Browser storage limits and eviction criteria* 2019). Also eine maximale Datenmenge, die der Browser generell speichert. Außerdem gibt es dann noch ein group limit, wobei jede Top Level Domain (TLD) eine eigene Gruppe darstellt (*Browser storage limits and eviction criteria* 2019). Zu einer Gruppe würden also hs-mannheim.de,

Browser	shared pool	group limit
Chrome	bis zu 60% der Festplattengröße	100% des shared pools
Firefox	bis zu 50% der Festplattengröße	20% vom shared pool aber maximal 2 GB
Safari	nicht bekannt	bis zu 1 GB, durch Nutzerbestätigung erweiterbar

Tabelle 4.1: Speicherlimits der Browser

www.hs-mannheim.de und bib.hs-mannheim.de gehören. Eine weitere Gruppe wäre zum Beispiel uni-mannheim.de. Das group limit hängt unter anderem vom shared pool ab. Die genauen Limits sind in Tabelle 4.1 angegeben.

Safari erlaubt eine Nutzung von bis zu 1 Gigabyte (GB), wenn dieses Limit erreicht ist, wird der Nutzer gefragt, ob er mehr Speicher erlauben möchte (LePage 2020).

Die StorageManager API kann verwendet werden, um den verfügbaren Speicherplatz abzufragen und den bisher genutzten Speicherplatz abzufragen (LePage 2020). Diese Angaben müssen nicht genau sein, sondern bieten nur einen Richtwert (LePage 2020). Diese Funktion wird aber nicht von allen Browsern unterstützt. Im Moment wird die API von Firefox und Chrome aber nicht von Safari unterstützt (Storage API 2019).

4.1.7 Mögliche Architekturen

In Tabelle 4.2 werden die vorgestellten Möglichkeiten zum Speichern von Daten nach bereits genannten Kriterien verglichen: maximale Datenmenge, mögliche Datentypen, synchron / asynchron, Browsersupport, Aufwand für Suchen und Filtern. Bei fehlenden Feldern in Web SQL wurde nicht weiter recherchiert, weil diese Technologie veraltet ist und nicht mehr verwendet werden soll.

Technologie	Datenmenge	Datentypen	(a)synchron	Browsersupport	Unterstützung für Suchen und Filtern	Anmerkung
<i>Web Storage API</i>	max. 5 MB	String	nur synchron	alle	nicht vorhanden	
<i>File System API</i>	max. Speicherkapazität des Browsers	blobs und weitere	synchron und asynchron	nur Chrome	nicht vorhanden	
<i>Web SQL</i>	-	.	synchron und asynchron	teilweise	vorhanden	veraltet
<i>IndexedDB</i>	max. Speicherkapazität des Browsers	einfache Datentypen, Objekte, blobs	asynchron	alle	vorhanden	
<i>Cache API</i>	max. Speicherkapazität des Browsers	Request- und Response-Objekte	asynchron	alle	nicht vorhanden	

Tabelle 4.2: Vergleich der APIs zur lokalen Datenspeicherung

Web SQL ist deprecated und soll deswegen in CROSSLOAD keine Anwendung finden. Außerdem ist das Speichern von großen Datenmengen dringend erforderlich, weshalb jede Architektur mindestens eine der folgenden Technologien benutzen muss: File System API, IndexedDB, Cache API.

Es werden nun mögliche Architekturen vorgestellt und miteinander verglichen. Als Kriterien gelten die gleichen wie bisher: maximale Datenmenge, mögliche Datentypen, synchron / asynchron, Browsersupport, Aufwand für Suchen und Filtern.

Cache API und Web Storage

Die erste Architektur ist in Abbildung 4.5 zu sehen. Es wird die Cache API zum Speichern der Audio-Dateien und den Web Storage zum Speichern der Metadaten verwendet. Metadaten sind in diesem Fall alle Information, die zu einem Inhalt gehören abgesehen von der Audio-Datei. Der Web Storage speichert alle ids der bisher favorisierten Inhalte in einem Array. Außerdem hat jede id einen eigenen Eintrag im Web Storage, der die Metadaten enthält. Dieser Ansatz hat den Vorteil, dass schnell überprüft werden kann, ob ein Inhalt favorisiert ist oder nicht. Das Abrufen aller favorisierten Inhalte dauert hingegen lange, weil viele verschiedene Einträge aus dem Web Storage geladen werden müssen. Als alternativen Ansatz könnte man den in Listing 4.1 beschriebenen umsetzen. Hier werden alle Metadaten in einem einzigen Array gespeichert. Das Laden aller Inhalte ist jetzt schneller. Das Speichern oder Laden eines einzelnen Inhalts dauert dagegen länger, weil immer das gesamte Array geladen oder gespeichert werden muss.

```
favoriten: [
  {
    id: "id1",
    titel: "Titel des Inhalts",
    erstelltAm: "05.06.2020",
    // weitere Metadaten
  },
  {
    id: "id2",
    // weitere Metadaten
  }
]
```

Listing 4.1: Speichern der Metadaten in einem Array

In Abbildung 4.6 ist der Ablauf dargestellt wenn ein Nutzer einen Inhalt favorisiert. Sobald der Nutzer einen Inhalt favorisiert werden die Metadaten abgerufen und di-

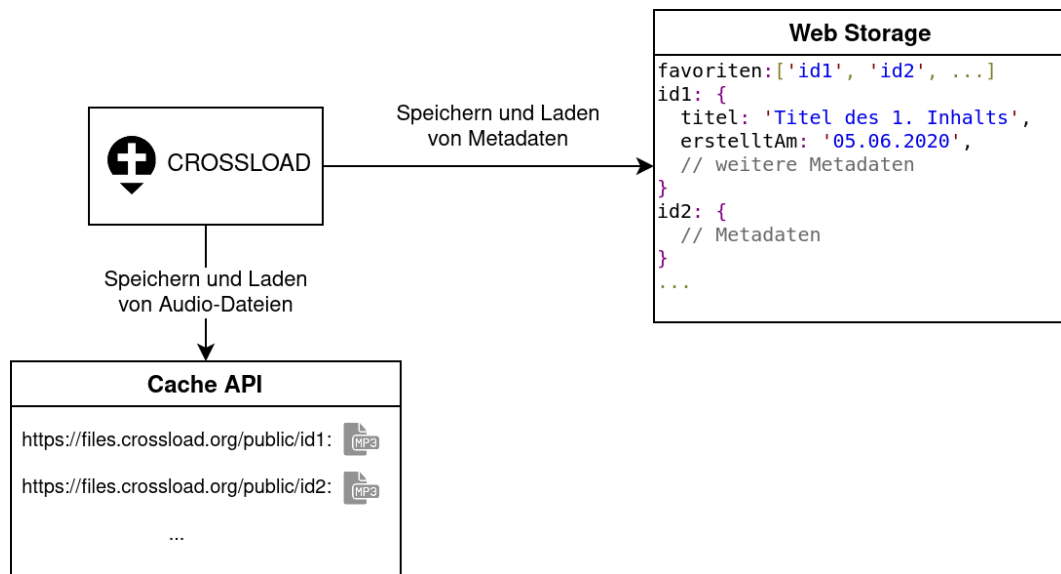


Abbildung 4.5: Architektur mit Cache API und Web Storage

rekt abgespeichert. Anschließend beginnt der Download der Audio-Datei, welche in der Cache API gespeichert wird. Sobald das Herunterladen und Speichern erfolgreich war, wird dies in den Metadaten aktualisiert und der Nutzer benachrichtigt.

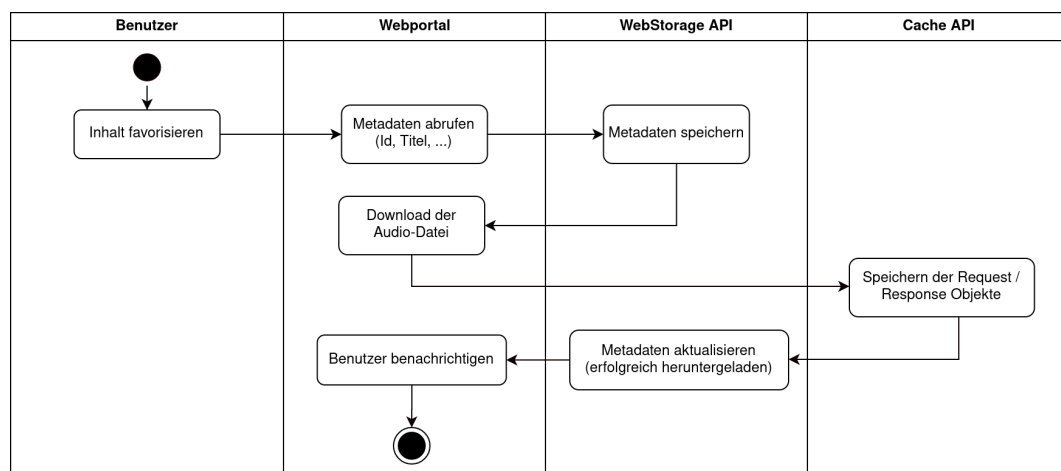


Abbildung 4.6: Ablauf für das Favorisieren mit Cache API und Web Storage

In Abbildung 4.7 wird der Ablauf für das Abspielen eines Inhaltes gezeigt. Damit der Nutzer auf den abzuspielenden Inhalt kommt, lässt er sich alle favorisierten Inhalte anzeigen. Die Metadaten aller Inhalte müssen dementsprechend aus dem Web Storage geladen werden und anschließend angezeigt werden. Der Nutzer wählt einen dieser Inhalte aus. Die Audio-Datei wird nun aus der Cache API geladen und abgespielt.

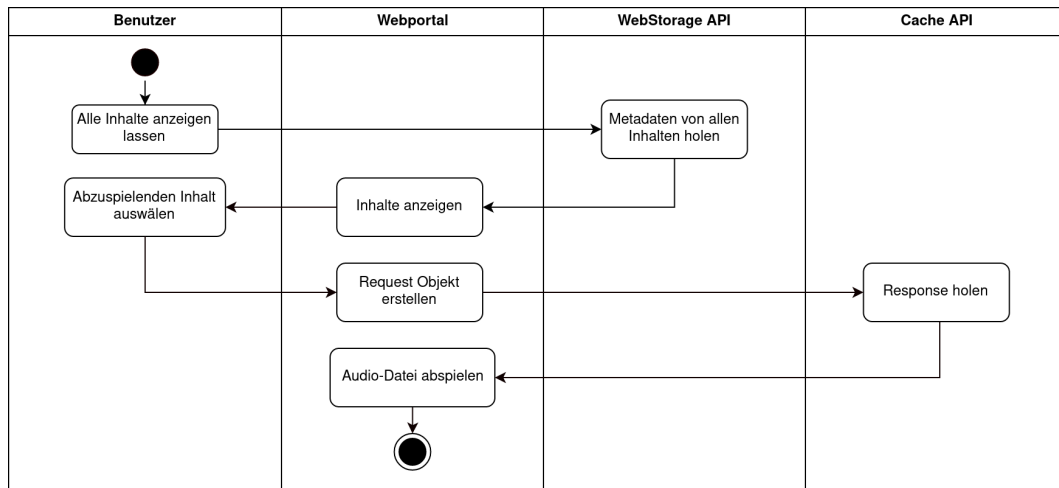


Abbildung 4.7: Ablauf für das Abspielen mit Cache API und Web Storage

Der Vorteil dieser Architektur liegt in der Einfachheit der benutzten APIs. Sowohl die WebStorage API, sowie die Cache API sind sehr einfach zu verstehen und zu verwenden. Außerdem sind beide Technologien in allen relevanten Browsern verfügbar. Die Nachteile dieser Architektur sind:

- Der Zugriff auf den Web Storage ist synchron. Bei vielen Daten kann dies zu längeren Ladezeiten kommen und währenddessen ist das Portal nicht bedienbar.
- Durch die Verwendung des Web Storages werden entweder Aktualisierungen der Metadaten oder das Abrufen aller Metadaten ineffizient.
- Das Speicherlimit des Web Storages beträgt 5 MB. Bei Metadaten bis zu 50 kB können bis zu 100 Einträge gespeichert werden.
- Wenn Inhalte durchsucht oder gefiltert werden müssen, muss dies manuell nach dem Laden aller Metadaten erfolgen.

Cache API und IndexedDB

Abbildung 4.8 zeigt die Architektur mit Cache API und IndexedDB. Die Cache API wird zum Speichern der Audio-Dateien verwendet und in der IndexedDB werden alle Metadaten gespeichert.

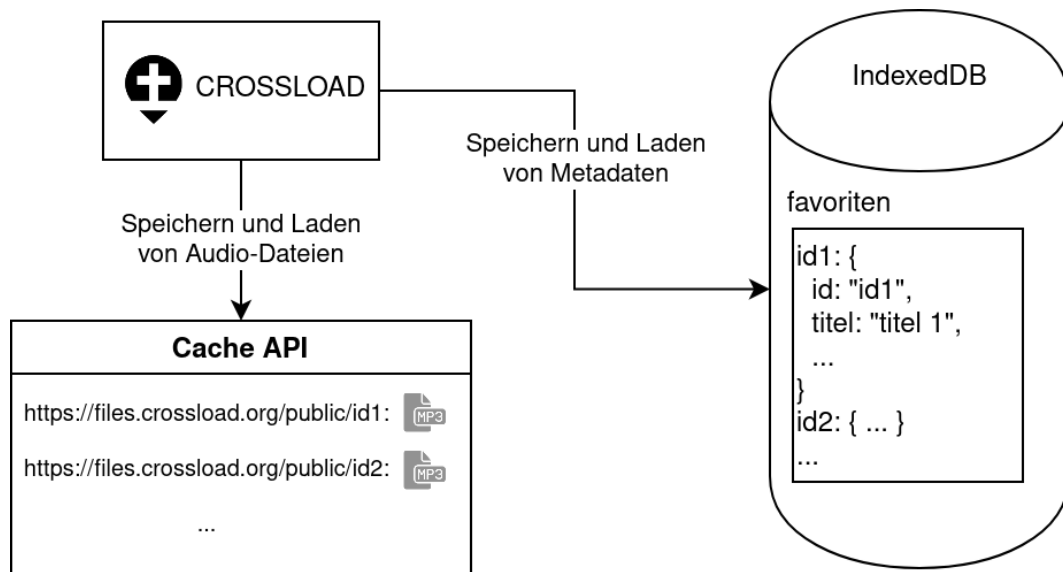


Abbildung 4.8: Architektur mit Cache API und IndexedDB

Der Ablauf zum Favorisieren und Abspielen eines Inhaltes bleibt wie in Abbildung 4.6 und Abbildung 4.7 beschrieben mit dem kleinen Unterschied, dass der Web Storage durch IndexedDB ersetzt wird.

Durch die Verwendung von IndexedDB ist das Abrufen und Speichern der Metadaten nicht synchron, sondern asynchron. Außerdem bietet IndexedDB die Möglichkeit Inhalte zu sortieren und zu filtern. IndexedDB ist in allen relevanten Browsern verfügbar

File System API und IndexedDB

In dieser Architektur, welche in Abbildung 4.9 beschrieben ist, werden Audio-Dateien mit der File System API abgespeichert. Metadaten werden in der IndexedDB gespeichert, wo sie auch sortiert und gefiltert werden können. In dem virtuellem Dateisystem wird ein Ordner *favoriten* angelegt. In diesem Ordner gibt es Unterordner für jeden Inhalt, welche die heruntergeladene Audio-Datei enthalten.

Der Nachteil dieser Architektur ist die Browserkompatibilität. Die File System API wird bisher nur in Chrome unterstützt.

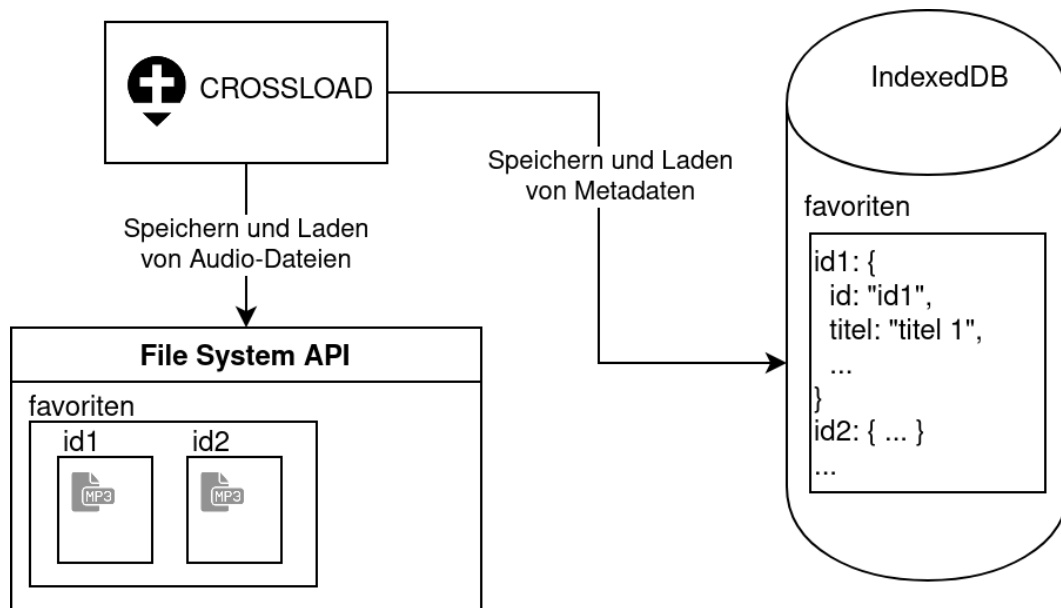


Abbildung 4.9: Architektur mit File System API und IndexedDB

IndexedDB

Abbildung 4.10 zeigt die Architektur nur mit IndexedDB zum Speichern der Daten. Sowohl Metadaten als auch die Audio-Datei werden in der Datenbank abgespeichert. Es werden zwei object stores verwendet. Einer speichert alle Metadaten und einer speichert alle Audio-Dateien. Es wäre auch möglich die Audio-Dateien mit im object store *favoriten* zu speichern, das wäre jedoch schlecht für die Geschwindigkeit der Seite wenn nur die Metadaten ohne die Audio-Datei benötigt werden.

Abbildung 4.11 zeigt dies am Beispiel für das Abspielen eines Inhaltes auf CROSSLOAD. Wenn der Nutzer sich alle Inhalte anzeigen lässt werden zuerst nur die Metadaten benötigt und sobald ein Inhalt ausgewählt wurde wird eine Audio-Datei zu einem Inhalt benötigt.

In Abbildung 4.12 ist der Ablauf zum Favorisieren eines Inhaltes gezeigt. Sowohl Metadaten als auch die Audio-Datei wird in der IndexedDB gespeichert.

Das Abrufen und Speichern aller Daten ist nicht synchron, sondern asynchron. Außerdem bietet IndexedDB die Möglichkeit Metadaten sortiert oder gefiltert abzurufen.

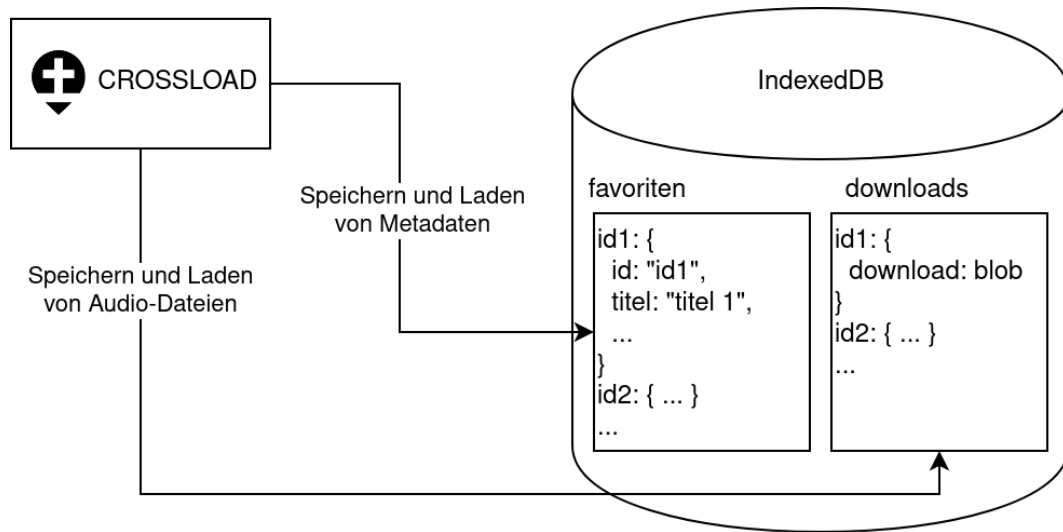


Abbildung 4.10: Architektur mit IndexedDB

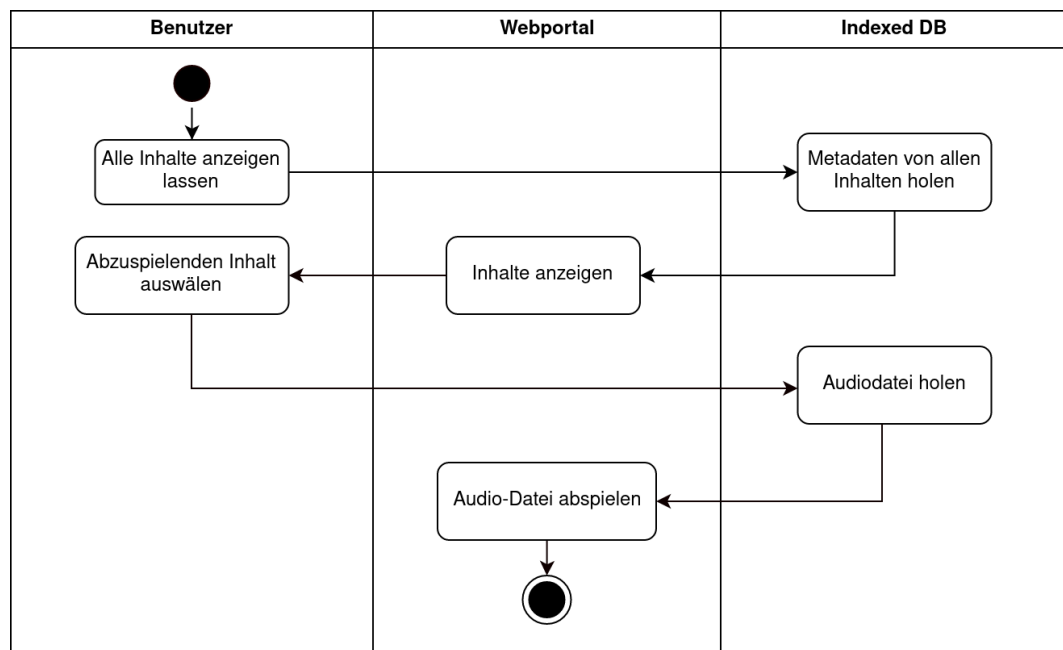


Abbildung 4.11: Ablauf für das Abspielen mit IndexedDB

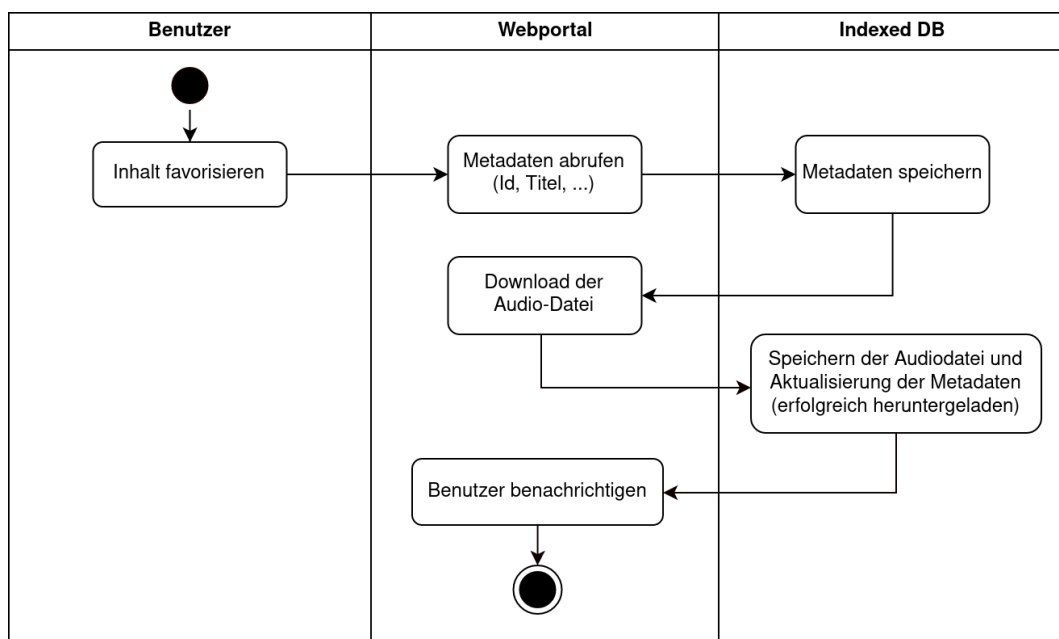


Abbildung 4.12: Ablauf für das Favorisieren mit IndexedDB

Architektur	Datenmenge	Datentypen	(a)synchron	Browsersupport	Unterstützung für Suchen und Filtern
Cache API und Web Storage	max. 5 MB Metadaten	Metadaten müssen in einen String umgewandelt werden	Metadaten nur synchron	alle	nicht vorhanden
Cache API und IndexedDB	max. Speicherkapazität des Browsers	blobs und weitere	synchron und asynchron	alle	vorhanden
File System API und IndexedDB	max. Speicherkapazität des Browsers	blobs und weitere	synchron und asynchron	teilweise	vorhanden
IndexedDB	max. Speicherkapazität des Browsers	blobs und weitere	asynchron	alle	vorhanden

Tabelle 4.3: Vergleich der Architekturen zur lokalen Datenspeicherung

Entscheidung

In Tabelle 4.3 werden alle vorgestellten Architekturen miteinander verglichen. Die Architektur *Cache API und IndexedDB* sowie *IndexedDB* erfüllen alle Kriterien: Es wird die maximal zulässige Speicherkapazität des Browsers ausgenutzt, alle relevanten Datentypen können asynchron geladen und gespeichert werden, alle relevanten Browser unterstützen die APIs und das Filtern / Sortieren wird unterstützt.

Alle anderen Architekturen erfüllen mindestens ein Kriterium nicht und werden deswegen nicht verwendet.

Für CROSSLOAD wird sich für die Architektur *IndexedDB* entschieden. Die Verwendung von *Cache API und IndexedDB* wäre auch möglich, verkompliziert die Anwendung jedoch unnötig. Das Verwenden von zwei APIs macht den Code für andere Entwickler komplizierter aber bringt keinen Vorteil gegenüber der Verwendung von *IndexedDB* alleine.

4.2 Herunterladen der Audiodaten

Favorisierte Inhalten, sollen für den Offline-Gebrauch heruntergeladen werden. Dafür gibt es verschiedene Möglichkeiten. In diesem Kapitel werden verschiedene Funktionen vorgestellt und miteinander verglichen. Zum Vergleichen dienen die Kriterien: Mit welchen Browsern ist die API kompatibel, wie ist die Lebensdauer des Downloads (nur wenn der Tab geöffnet ist, Lebensdauer eines Service Workers, ...) und wie leicht ist die Technologie in die bestehende Anwendung integrierbar.

4.2.1 Fetch API

Die Fetch API bietet die Möglichkeit XMLHttpRequests zu verschicken, also Ressourcen von einem Server anzufragen (Rojas 2020). Verwendet wird die API mit Promises, ist also asynchron und bietet eine einfachere Fehlerbehandlung (Rojas 2020) (*Fetch API* 2020).

Außerdem unterstützt jeder relevante Browser diese Funktion (*Fetch API* 2020). Die Fetch API kann sowohl im DOM als auch in Service Workern verwendet werden (*Fetch API* 2020). Das heißt, dass in einem Angular Service oder in Komponenten der Anwendung eine Netzwerkanfrage mit der Fetch API versendet werden kann.

Wenn die Fetch API in einem Service Worker verwendet wird gibt es ein paar Vorteile. Ein Service Worker kann begrenzt im Hintergrund weiterlaufen auch wenn die Webanwendung geschlossen ist. Jedoch kann ein Service Worker vom Browser zu jeder Zeit beendet werden, selbst wenn ein Event noch nicht beendet wurde (Russell u. a. 2020). Die Ausführungsdauer für ein Event sollte sehr kurz sein und ist nicht für langanhaltende Downloads gedacht (Russell u. a. 2020).

Bis jetzt bietet Angular keinen Weg einen eigenen Service Worker zu schreiben der gut in die Anwendung integrierbar ist. Also zum Beispiel Funktionen eines Services aufrufen kann.

4.2.2 HttpClient

In Angular können Netzwerkanfragen mit dem HttpClient verschickt werden (*HttpClient* 2020). Bei einem Request kann die URL, Header, Parameter und vieles mehr angepasst werden (*HttpClient* 2020). Die Antwort eines Request wird als Observable zurück gegeben, auch mit der Option zwischendurch einen Fortschritt des Requests abzufragen (*HttpClient* 2020). Dadurch, dass der HttpClient ein Teil von Angular ist funktioniert er in allen relevanten Browsern und ist sehr leicht in die bestehende Anwendung integrierbar.

Der HttpClient kann in allen Angular Komponenten und Services per Dependencie Injection genutzt werden (*HttpClient* 2020). In Service Worker können keine angularspezifische Teile verwendet werden, somit ist auch der HttpClient in Service Workern nicht nutzbar (*Angular service worker introduction* 2019).

4.2.3 Background Sync

Mit Background Sync können Netzwerk Anfragen abgeschickt werden, selbst wenn keine Internetverbindung besteht. Die Anfrage wird gespeichert, und sobald eine Internetverbindung besteht abgeschickt (Josh Karlin 2020) (Rojas 2020).

Abbildung 4.13 zeigt den Ablauf wenn ein Sync-Event gesendet wird. Wenn bereits eine Netzwerkverbindung besteht wird das Event sofort ausgeführt. Ansonsten wird auf eine Netzwerkverbindung gewartet. Das Ausführen des Event kann zum Beispiel durch einen Netzwerkfehler scheitern. Wenn nach mehrmaligen Versuchen das Event nicht geklappt hat wird ein *Last-Chance-Event* gefeuert (Rojas 2020).

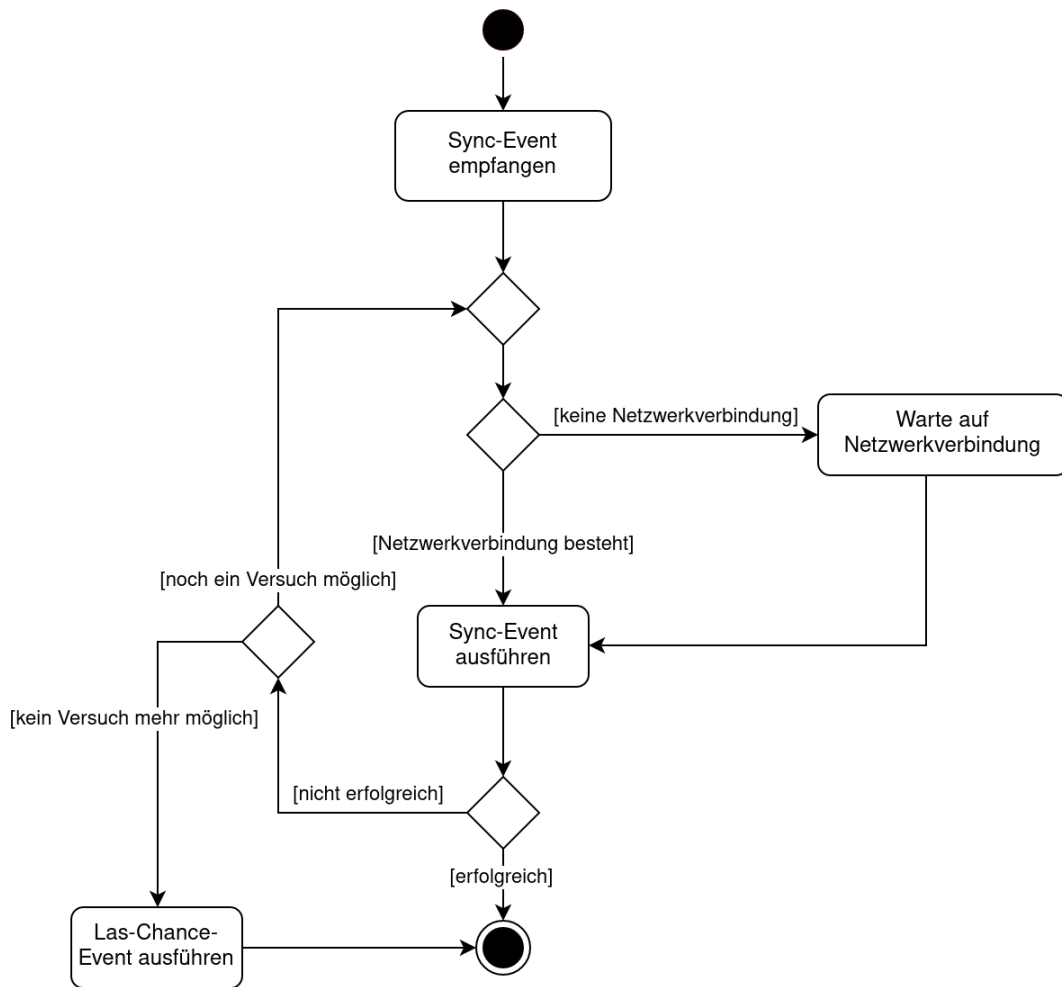


Abbildung 4.13: Ablauf für Background Sync

Diese Funktionalität ist Teil von Service Workern und funktioniert solange ein Service Worker läuft (Josh Karlin 2020). Auch das Event für Background Sync hat nur eine begrenzte Lebensdauer und kann jederzeit vom Browser beendet werden (Russell u. a. 2020). Auch diese Funktionalität von Service Workern ist noch nicht in Angular integriert. Bei der Verwendung dieser Funktionalität können also keine Angular Komponenten / Funktionen zum Einsatz kommen.

Im Moment wird diese Funktion nur von Chrome unterstützt (*Background Sync API* 2020), in Firefox ist sie aber schon in Entwicklung (*Firefox Platform Status* 2020).

4.2.4 Background Fetch

Diese API bietet die Möglichkeit große Dateien hoch- oder herunterzuladen und zeigt dem Nutzer dabei Informationen über den Fortschritt (Archibald 2018). Der Unterschied zu Background Sync liegt auch darin, dass der Download / Upload funktioniert wenn der Browser geschlossen wird und kein Service Worker aktiv ist (Archibald 2018). Abbildung 4.14 zeigt die Fortschrittsanzeige auf einem Android Smartphone. Viele nützliche Funktionen, wie pausieren und fortsetzen des Downloads bei Verbindungsunterbrechungen werden ebenso von Background Fetch übernommen (Archibald 2018).

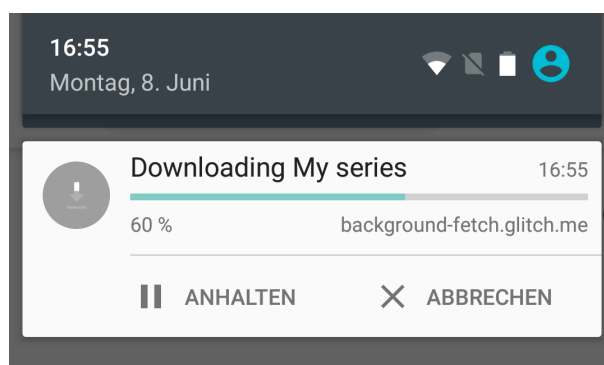


Abbildung 4.14: Benachrichtigung auf Android bei Benutzung der Background Fetch API

Diese API ist im Moment nur in Chrome verfügbar (Archibald 2018).

Auch Background Fetch ist noch nicht in Angular integriert und Bibliotheken zur Verwendung konnten auch nicht gefunden werden. Somit ist die Integration in die bestehende Anwendung kompliziert.

4.2.5 Entscheidung

Der HttpClient und die Fetch API laufen in allen relevanten Browsern. Die Fetch API und Background Sync erlauben einen längeren Download im Hintergrund solange der Service Worker noch aktiv ist, selbst wenn der Tab nicht geöffnet ist. Der HttpClient ist dagegen leichter zu integrieren.

Als Technologie wird für CROSSLOAD der HttpClient ausgewählt. Dieser ist am einfachsten zu integrieren und läuft in allen Browsern. Die längere Lebensdauer der Fetch API ist nur minimal, weil ein Service Worker keine langen Berechnungen / Downloads durchführen soll.

Technologie	Browserkompatibilität	Lebensdauer	Integrierbarkeit
<i>Fetch API</i>	alle	Lebensdauer eines Service Workers	in Service Workern schlecht integrierbar, ansonsten gut
<i>HttpClient</i>	alle	Solange die Anwendung geöffnet ist	sehr gut
<i>Background Sync</i>	Chrome	Lebensdauer eines Service Workers	schlecht
<i>Background Fetch</i>	Chrome	Bis der Download abgeschlossen ist	schlecht

Tabelle 4.4: Vergleich der Technologien zum Herunterladen von Dateien

Als Ergänzung kann später Background Fetch implementiert werden, weil damit auch Inhalte heruntergeladen werden können wenn die Website geschlossen ist. Dies ist jedoch nicht Teil dieser Thesis.

4.3 Verbindungsstatus auslesen

In Anforderung 3 und 6 in Abschnitt 3.3 wurden Anforderungen definiert, die ein Auslesen des Verbindungsstatus erfordern. Dazu gehört die Frage ob überhaupt eine Internetverbindung besteht und auch ob der Nutzer über Mobile Daten oder Wifi verbunden ist. Außerdem soll auf eine Änderung dieser Werte möglichst schnell reagiert werden können.

In diesem Kapitel wird untersucht, ob Funktionen zum Auslesen dieser Daten existieren und welche Browser diese unterstützen.

4.3.1 Online / Offline

Mit *navigator.onLine* gibt es in allen relevanten Browsern (*Online/offline status* 2020) eine Funktion, um zu überprüfen ob eine Internetverbindung besteht oder nicht (Sheppard 2017) (*Online and offline events* 2019). Diese API arbeitet nicht ganz genau. Wenn eine Verbindung zu einem Netzwerk besteht, in diesem Netzwerk aber kein Internetzugriff möglich ist gibt *navigator.onLine* in manchen Fällen true zurück und signalisiert eine Internetverbindung (Sheppard 2017). Es kann aber nicht vorkommen, dass der Nutzer offline ist und die API true zurück liefert. Dies ist eine zu vernachlässigende Ungenauigkeit, weil dieser Fall nur sehr selten auftritt.

Wenn eine zuverlässigere Methode benötigt wird kann ein einfacher Request abgeschickt werden. Wenn ein Ergebnis zurück kommt existiert eine Verbindung, wenn ein Fehler geworfen wird, existiert keine Verbindung.

Des weiteren gibt es auch eine Möglichkeit einen Verbindungsstatuswechsel mitzubekommen. Auf dem `<body>` Element werden zwei Events gefeuert: *online* wenn der Browser eine Internetverbindung herstellen konnte und *offline* wenn eine bestehende Internetverbindung abbricht (*Online and offline events* 2019). Auch hier treffen die Limitierungen von *navigator.onLine* zu.

4.3.2 Verbindungsart

Zusätzlich ist es erforderlich die aktuelle Verbindungsart des Gerätes auslesen zu können. Das könnte zum Beispiel Wifi oder Mobilfunk sein. In Anforderung 3 aus Abschnitt 3.3 wird mit einem Download so lange gewartet bis eine Wifi-Verbindung besteht oder der Nutzer zugestimmt hat. Mit *navigator.connection.type* gibt es eine API dies zu erreichen (*NetworkInformation.type* 2019). Der Wert *cellular* wird zurückgegeben wenn eine Mobilfunkverbindung besteht (*NetworkInformation.type* 2019). Es ist auch möglich auf ein Event zu hören sobald sich dieser Status ändert (*Network Information API* 2020). Diese API ist aber noch nicht in allen relevanten Browsern verfügbar. Nur Nutzer die Chrome verwenden können von dieser API profitieren (*NetworkInformation.type* 2019).

Kapitel 5

Implementierung

Die zuvor herausgearbeiteten Konzepte werden nun im bereits vorhandenem Webportal umgesetzt. Dabei wird der bisherige Aufbau der Anwendung analysiert und erweitert. Einzelne wichtige Aspekte werden in diesem Kapitel herausgegriffen und erläutert.

5.1 Offline Metadaten

Jeder Inhalt auf CROSSLOAD besitzt Eigenschaften, die hier als Inhalt oder Metadaten bezeichnet werden. Dazu gehören zum Beispiel die id, Titel, Bibelstellen, Ort oder Datum. Diese Daten werden benötigt, um Suchergebnisse oder die Detailseite einer Predigt anzuzeigen. In diesem Kapitel werden Strategien beschrieben wie diese Inhalte gecached werden, um eine bestmögliche Nutzererfahrung zu bieten.

5.1.1 Detailseite

Im Webportal CROSSLOAD wird bereits ein Service Worker verwendet, der einerseits statische Inhalte zwischenspeichert aber auch dynamische Requests cached. Für dynamische Request wurde bislang die Cache Strategie freshness verwendet. Das heißt, dass der Cache nur verwendet wird wenn der Netzwerk-Request fehlschlägt (*Service worker configuration* 2020). Angular Service Worker unterstützten noch eine zweite Cache Strategie, die immer den Wert aus dem Cache zurück gibt wenn er verfügbar ist (*Service worker configuration* 2020). Die momentane Lösung ist nicht optimal. Sie ermöglicht zwar eine Offlineverfügbarkeit der Daten aber bie-

tet nicht die optimale Geschwindigkeit für den Nutzer. In vielen Fällen wird sich der Inhalt im Cache nicht mit der Antwort aus der Netzwerkanfrage unterscheiden, trotzdem muss der Nutzer auf die Antwort vom Netzwerk warten. Eine bessere Lösung ist folgende:

Wenn der Nutzer einen Inhalt möchte, wird direkt im Cache nachgeschaut ob der Inhalt existiert. Wenn er existiert wird er dem Nutzer angezeigt. Gleichzeitig wird eine Netzwerkanfrage geschickt. Sobald die Antwort vom Netzwerk da ist wird die Antwort mit dem Wert im Cache verglichen. Wenn sich die Werte unterscheiden wird der Cache aktualisiert und dem Nutzer die aktuelleren Daten angezeigt. Diese Strategie ist mit Service Workern nicht so einfach zu lösen, weil ein Service Worker nur ein Mittelsmann in einem Request ist. Das heißt er kann nur einen Wert zurückgeben und nicht einen zweiten etwas zeitverzögert. Deswegen wird der Offline Inhalt nicht im Service Worker gecached, sondern in einem Angular-Service.

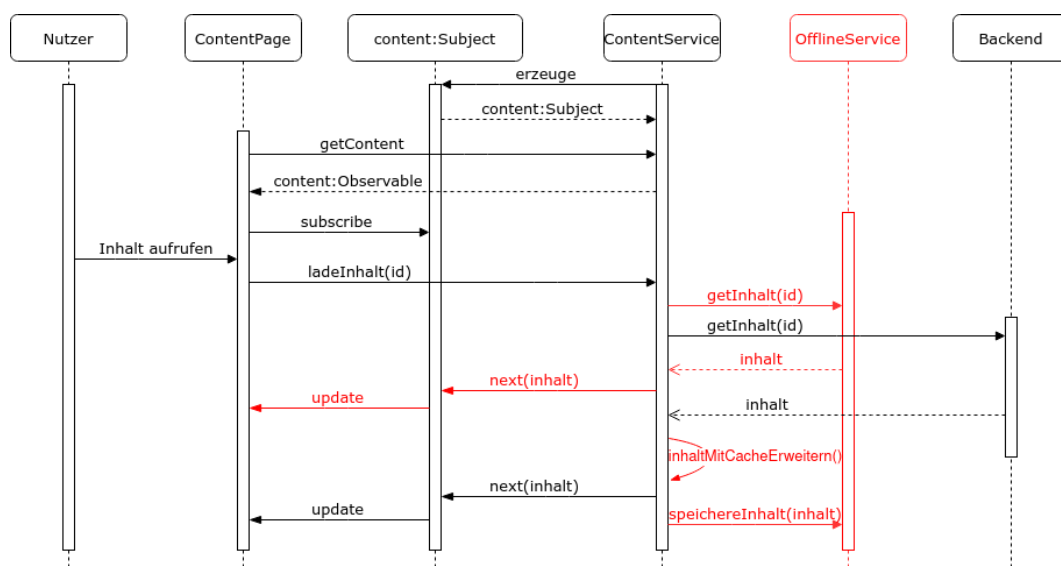


Abbildung 5.1: Sequenzdiagramm für den Abruf der Metadaten eines Inhalts

In Abbildung 5.1 ist die bereits beschriebene Cache-Strategie gezeigt. Alle Teile in Schwarz, sind bereits vorhanden. Die Teile in Rot sind Ergänzungen im Rahmen dieser Arbeit. Im ContentService werden bereits Subjects und Observables verwendet. Durch das Publish-Subscribe Pattern, das dadurch erzeugt wird ist es möglich mehrere Subscriber (in diesem Fall nur ResultPage) zu benachrichtigen sobald sich ein Inhalt ändert. Darüber ist es auch möglich etwas zeitversetzt einen aktualisierten Inhalt auszuliefern. Der OfflineService verwaltet den Cache, dessen Implementierung wird in Abschnitt 5.2 genauer beschrieben. Manche Eigenschaften des Inhalts

werden nur lokal gespeichert. Dazu gehört die Eigenschaft, ob ein Inhalt favorisiert ist und ob die entsprechende Audio-Datei offline verfügbar ist. Deswegen wird der Inhalt vom Backend mit dem Inhalt aus dem OfflineService erweitert.

5.1.2 Suche

Auf CROSSLOAD ist es möglich Inhalte anhand verschiedener Kriterien zu suchen. Die Suche wird nicht gecached, was auch weiterhin so bleiben soll. Die Suchergebnisse können sich oft ändern, zum Beispiel wenn neue Inhalte hinzukommen, dass sich ein cachen nicht lohnt. Dem Nutzer soll aber trotzdem angezeigt werden, ob ein Inhalt, der in der Suche angezeigt wird, offline verfügbar ist bzw. favorisiert ist. Dafür muss für jeden Inhalt von den Suchergebnissen im Offline Speicher geschaut werden, ob ein Inhalt verfügbar ist. Diese Überprüfung kann je nach Anzahl der Inhalte einige Millisekunden dauern. Um die bestmögliche Nutzererfahrung bieten zu können wir eine ähnliche Strategie wie für die Detailseite verwendet:

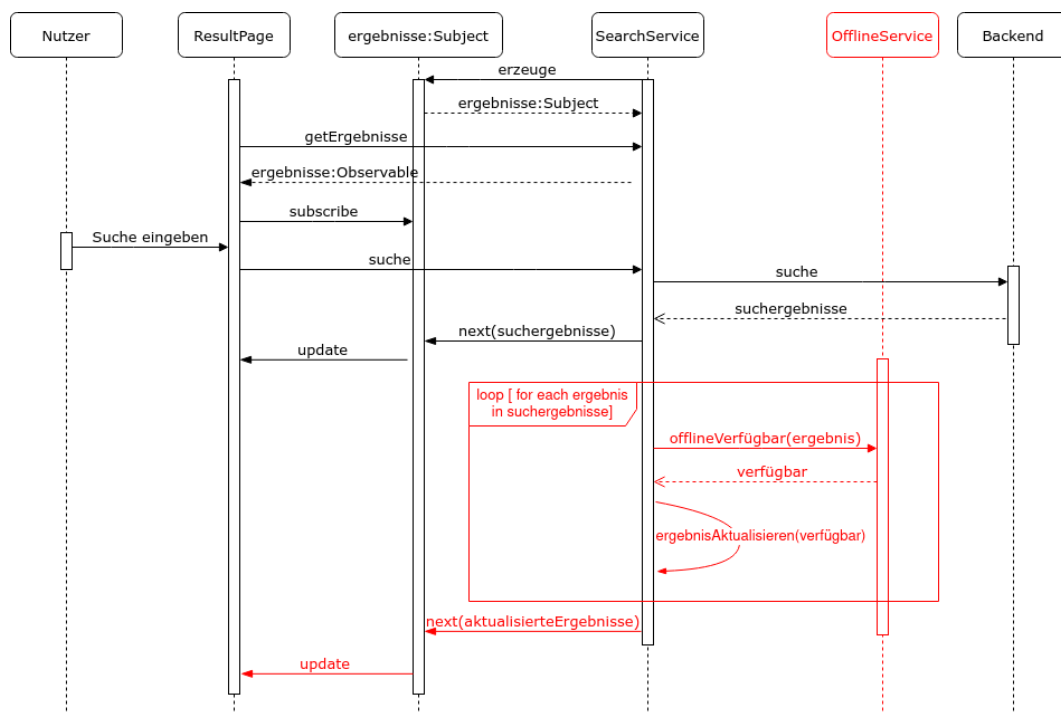


Abbildung 5.2: Sequenzdiagramm für die Suche nach Inhalten

In Abbildung 5.1 wird der Ablauf für das Suchen auf CROSSLOAD dargestellt. Alle Teile in Schwarz, sind bereits vorhanden und die Teile in Rot sind Ergänzungen im Rahmen dieser Arbeit. Sobald ein Nutzer Inhalte auf CROSSLOAD sucht

wird eine Netzwerkanfrage gesendet. Sobald die Antwort der Anfrage da ist, werden die Inhalte dem Nutzer angezeigt. Gleichzeitig wird für jeden gefundenen Inhalt überprüft, ob er offline verfügbar ist. Dieser Status wird dann in das Suchergebnis geschrieben. Sobald alle Inhalte überprüft wurden, werden die aktualisierten Suchergebnisse dem Nutzer angezeigt. Dafür wird wieder das Publish-Subscribe Pattern verwendet.

5.2 Lokale Datenspeicherung

Für die Speicherung von Offlinedaten wurde sich bereits für IndexedDB entschieden. Die Verwendung von IndexedDB ist nicht trivial, weil mit Events gearbeitet wird und nicht wie in moderneren APIs mit Promises (*IndexedDB Grundlagen* 2020). Es gibt aber einige Bibliotheken, die das Arbeiten mit IndexedDB erleichtern. Für diese Arbeit wurde Dexie ausgewählt. Dexie ist ein minimaler Wrapper für IndexedDB, zum Arbeiten mit Promises (Fahlander 2020). Es gibt auch noch andere Bibliotheken wie LocalForage. Dexie wurde ausgewählt, weil mehrere Indexe in der Datenbank unterstützt werden und generell ist die Bibliothek sehr flexibel und trotzdem klein.

Die ganze Logik befindet sich in der Angularanwendung in einem Service. Dieser Service abstrahiert alle Datenzugriffe, sodass falls später notwendig Dexie auch durch eine andere Bibliothek ausgetauscht werden kann.

IndexedDB besitzt kein Schema wie von relationalen Datenbanken bekannt. Es können beliebige Objekte gespeichert werden, die nicht alle die gleiche Struktur haben. Wenn man diese Objekte aber durchsuchen möchte, sollte man einen Index auf die durchsuchten Eigenschaften legen. Diese Indexe muss man beim Erstellen der Tabellen angeben. Für diese Anwendung werden zwei Tabellen angelegt: *content* und *downloads*. Siehe auch Abbildung 4.10 zur Erklärung.

Die Tabelle *content* enthält alle Metadaten zu einem Inhalt. Für diese Daten besteht schon ein Datenobjekt, welches in die Datenbank gespeichert wird. Als Index wird vorerst nur die *id* des Inhalts festgelegt. Später wenn auch nach anderen Kriterien gesucht oder gefiltert werden soll, kann ein weiterer Index hinzugefügt werden.

5.3 Herunterladen der Audiodaten

Auch das Herunterladen der Audiodaten wird in einem Angular-Service umgesetzt. Der Service ist dafür zuständig abzufragen welche APIs auf dem aktuellen Gerät unterstützt werden und startet dann den Download. Zuerst wird das Herunterladen mittels dem HttpClient von Angular implementiert. Eine zusätzliche Implementierung von Background Fetch ist in dieser Arbeit nicht vorgesehen aber für zukünftige Entwicklungen von CROSSLOAD vorgemerkt.

Abbildung 5.3 zeigt die Benutzung des *DownloadService* zusammen mit anderen Komponenten. Dazu gehört die *ContentPage*, welche dafür zuständig ist, einen Inhalt anzuzeigen. Über die *ContentPage* kann ein Inhalt favorisiert werden und heruntergeladen werden. Der Downloadfortschritt soll an mehreren Stellen in der Anwendung verfügbar sein: Der Download soll auf der entsprechenden *ContentPage* angezeigt werden. Außerdem gibt es eine *DownloadComponent*, welche alle laufenden Downloads mit Fortschritt anzeigt.

Auch der *DownloadService* setzt wieder auf Subjects und Observables, um das Publish-Subscribe Pattern umzusetzen. Zuerst wird ein Subject erzeugt, über das später der Downloadfortschritt an andere Komponenten geschickt wird. Immer wenn ein Fortschritt vom *HttpService* kommt, wird dem Subject ein neuer Wert geschickt. Die *ContentPage* abonniert Ergebnisse auf dem Subject, die etwas mit dem Inhalt zu tun hat, der gerade angezeigt wird. Die *DownloadComponent* dagegen abonniert alle Fortschritte. Sobald ein Download fertiggestellt wurde, wird ein letztes mal ein Fortschritt versendet und das Ergebnis über den *OfflineService* in die Datenbank abgelegt.

5.4 Grafische Oberfläche

Der Designentwurf für die Funktion *unterwegs anhören* wurde gemeinsam diskutiert und herausgearbeitet.

In der Desktop-Ansicht wird in den Player ein Switch hinzugefügt mit dem ein Inhalt Offline verfügbar gemacht werden kann. In Abbildung 5.4 ist dieser Player mit dem Button zu sehen. Außerdem gibt es einen Button *Meine Inhalte* mit dem man auf die Übersichtsseite mit allen heruntergeladenen Inhalten kommt.

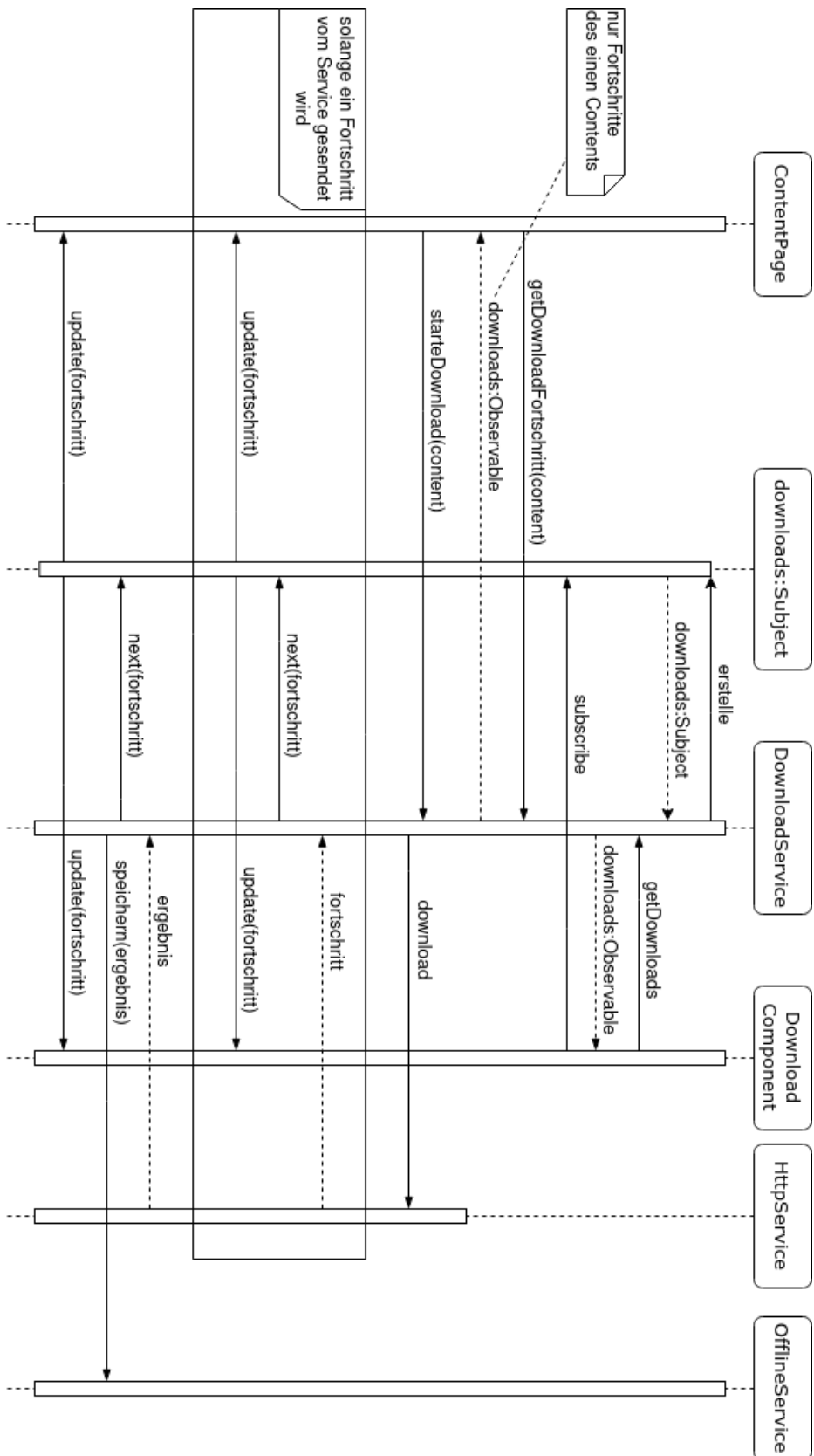


Abbildung 5.3: Sequenzdiagramm für den Download von Dateien



Abbildung 5.4: Player der Desktop-Ansicht

Da der Player in der Mobile-Ansicht nicht genügend Platz bietet wird der Switch zum Herunterladen unter dem Player dargestellt. Dies ist in Abbildung 5.5 zu sehen. Wenn der Player vergrößert wird erscheint auch der Button *Meine Inhalte* wie in Abbildung 5.6 zu sehen.

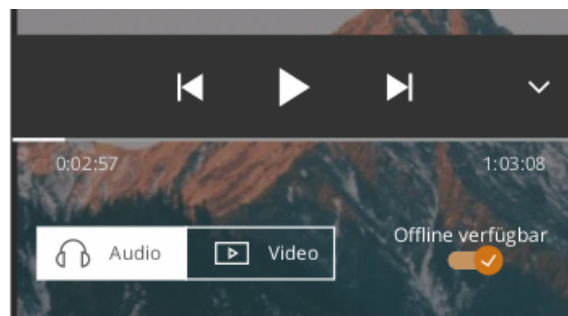


Abbildung 5.5: Design: Player der Mobilen-Ansicht

Zuletzt zeigt Abbildung 5.6 eine Liste mit allen Inhalten die heruntergeladen wurden. Diese können einzeln oder alle auf einmal gelöscht werden. Außerdem ist es möglich einen Inhalt abzuspielen oder alle Inhalte zu durchsuchen.

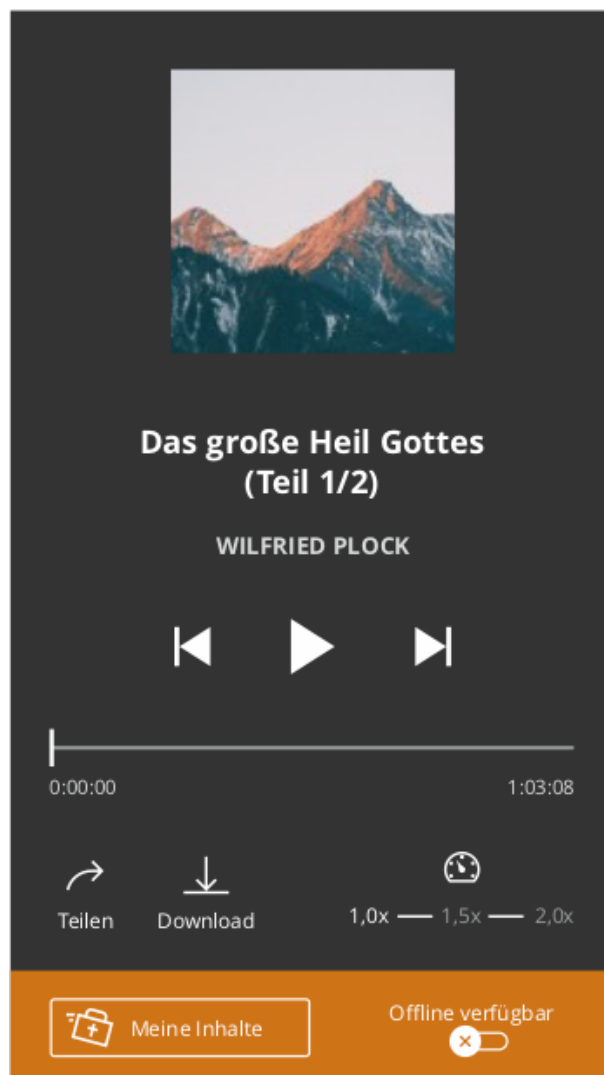


Abbildung 5.6: Design: Player der Mobilen-Ansicht

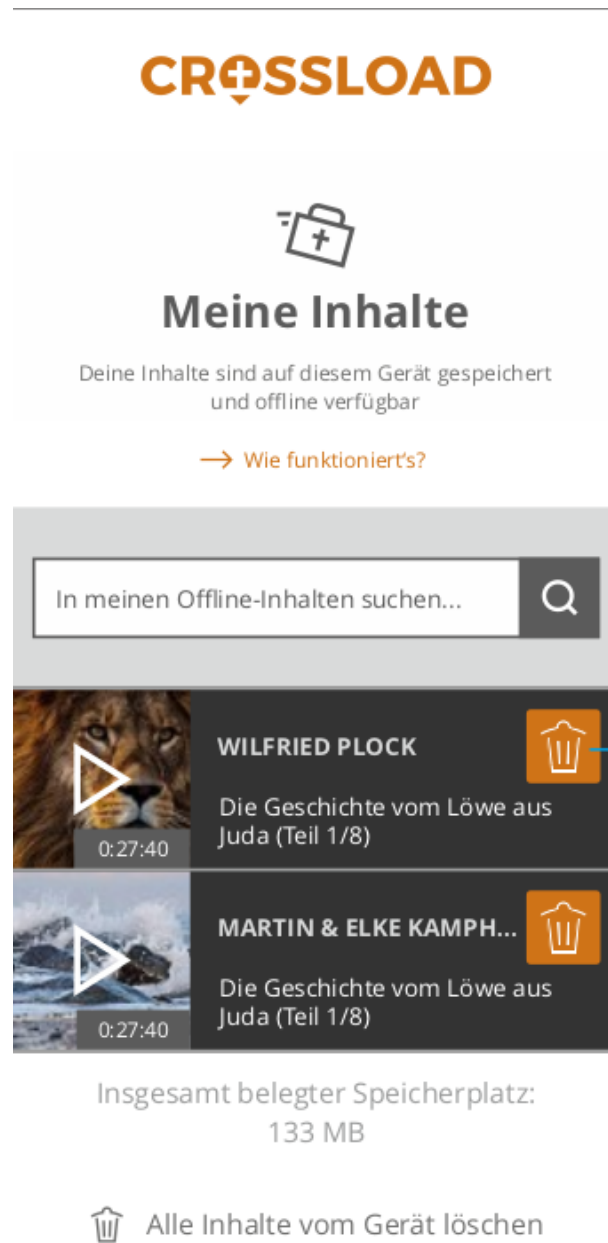


Abbildung 5.7: Design: Liste mit allen heruntergeladenen Inhalten

Kapitel 6

Evaluation und Reflexion

In diesem Kapitel wird die Arbeit kritisch reflektiert. Welche Anforderungen konnten erfüllt werden und wo ist noch Verbesserungspotenzial.

6.1 Kann durch die PWA eine native APP ersetzt werden?

Moderne Webtechnologien können in vielen Fällen eine eigene Native App für Android, iOS oder andere Systeme ersetzen. Ob dies für CROSSLOAD auch der Fall ist wird in diesem Kapitel untersucht. Dabei wird nur die Offline-Funktionalität berücksichtigt.

Durch das Überprüfen aller Anforderungen aus Abschnitt 3.3 kann festgestellt werden, ob eine PWA für CROSSLOAD ausreicht oder nicht. Im Nachfolgendem wird jede Anforderung einzeln bewertet. Folgende Bewertungen gibt es:

- erfüllt: Die Anforderung ist ohne Einschränkungen erfüllt
- teilweise erfüllt: Teile der Anforderung können erfüllt werden oder nur in manchen Browsern
- nicht erfüllt: Es ist mit einer PWA nicht möglich diese Anforderung zu erfüllen.

Wenn durch die Verwendung einer nativen App Vorteile für den Nutzer entstehen würden werden diese auch genannt. Tabelle 6.1 zeigt diesen Vergleich.

Bis auf die Anforderung 3 konnten alle Anforderungen erfüllt werden. Eine native App bietet in vielen Fällen eine bessere Nutzererfahrung ist aber für die reine Funk-

tionalität nicht notwendig. Für CROSSLOAD ist eine PWA eine gute alternative zu einer nativen App. Der Aufwand für jede Plattform eine eigene App zu schreiben ist nicht angemessen, um Anforderung 3 für alle Nutzer zu erfüllen.

6.2 Nutzerfreundlichkeit

Generell bieten die neuen Funktionen, welche in dieser Thesis entwickelt wurden, einen Mehrwert für die Nutzer. Ob diese Funktion angenommen und wie sie genutzt wird, wird sich erst in ein paar Monaten zeigen.

Es gibt einen Punkt der noch zu verbessern ist und wofür noch keine gute Lösung gefunden wurde. Die hier entwickelten Offline-Funktionalitäten speichern Audio-Dateien im Browser in einer Datenbank. Auf CROSSLOAD gibt es aber auch die Möglichkeit die Audio-Datei als mp3-Datei auf das Endgerät herunterzuladen. Die heruntergeladene Audio-Datei kann dann mit anderen geteilt werden. Das Speichern im Browser hat den Vorteil der Übersichtlichkeit: Alle verfügbaren Inhalte sind direkt einsehbar und auch verwaltbar. Weil beide Funktionen Vorteile haben, sollen beide Funktionen erhalten bleiben. Für den Nutzer ist es auf den ersten Blick aber sehr schwer zu verstehen was der Unterschied der beiden Optionen ist. Für diesen Zweck wurde eine Erklärseite eingeführt, die dem Nutzer diese Funktion erklärt wenn er sich dafür interessiert. Es besteht jedoch das Problem dass viele Nutzer diese Erklärung nicht lesen und deswegen die neuen Offline-Funktionalitäten nicht kennenlernen und davon profitieren können.

Deswegen ist es sehr wichtig das Verhalten der Nutzer auf der Seite zu analysieren und gegebenenfalls eine andere Möglichkeit zu finden, dem Nutzer die neuen Offline-Funktionalitäten zu erklären.

Anforderung	Erfüllt	Anmerkung	Verbesserung durch native App
1	erfüllt	-	-
2	erfüllt	-	Der verfügbare Speicher im Browser ist begrenzt und nutzt nicht die ganze Festplatte. Bei sehr geringem freien Speicher könnte eine native App mehr Inhalte speichern.
3	teilweise	Die Verbindungsart kann nicht in allen Browsern ausgelesen werden	In einer nativen App können alle Netzwerkinformationen ausgelesen werden
4	erfüllt	-	Es gibt in einer PWA Einschränkungen, die eine native App nicht hat: Das Browserfenster muss geöffnet bleiben wenn ein Download stattfindet. Durch die Verwendung von Background Fetch, was aber nur in Chrome verfügbar ist, wird dieser Nachteil wieder ausgeglichen.
5	erfüllt	-	-
6	erfüllt	-	-
7	erfüllt	-	-
8	erfüllt	Eine performante Volltextsuche ist nicht möglich	In einer nativen App ist auch eine performante Volltextsuche möglich
9	erfüllt	-	-
10	erfüllt	-	-
11	erfüllt	-	Eine native App würde hier den Vorteil einer konfigurierbaren Benachrichtigung bieten. Über diese Benachrichtigung könnte man dann die Predigt stoppen oder zum nächsten Inhalt springen.

Tabelle 6.1: Erfüllung der Anforderungen

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel gibt eine Zusammenfassung der ganzen Arbeit. Zuletzt werden auch noch einige Punkte genannt, die in Zukunft angedacht werden könnten, um CROSSLOAD weiter zu verbessern.

7.1 Zusammenfassung

Das Ziel dieser Arbeit war es das bereits vorhandene Webportal CROSSLOAD um eine Offline-Funktionalität zu erweitern. Außerdem wurde dabei untersucht welche Webtechnologien dafür zur Verfügung stehen und ob diese ausreichen alle Anforderungen zu erfüllen. Dem Nutzer soll es möglich sein Inhalte des Portals im Browser zu speichern und diese Inhalte auch nutzen zu können wenn keine oder eine schlechte Internetverbindung besteht. Moderne Browser bieten die Möglichkeit Daten in einer Datenbank der IndexedDB zu speichern, die auch ein Filtern und Durchsuchen erlaubt. In dieser Datenbank werden Inhalte und deren Metadaten gespeichert. Der Download der Inhalte findet in einem Angular Service statt. Es gibt andere Technologien wie Background Fetch, die dem Nutzer eine bessere Erfahrung bieten würden, diese sind aber nicht für alle Browser verfügbar. Zum Bereitstellen der Inhalte und des Downloadfortschrittes wird immer auf das Publish-Subscribe Pattern gesetzt. Dadurch ist es sehr flexibel möglich Daten an mehrere Komponenten weiterzugeben. Die grafische Oberfläche für diese Funktionen wurde noch nicht fertig entwickelt, sondern nur die Machbarkeit gezeigt. Entwürfe für das endgültige Design sind für spätere Entwicklungen vorhanden. Zusammenfassend kann gesagt werden, dass Webtechnologien und PWAs sehr gut geeignet sind um dem

Nutzer auch eine Offlinenutzung zu ermöglichen. Fast alle Anforderungen konnten ohne Kompromisse umgesetzt werden. Nur eine Anforderung ist noch nicht in allen Browsern umsetzbar. Die Entwicklung wird dadurch erschwert, dass nicht jeder Browser alle Funktionen implementiert. Dazu gehört zum Beispiel die Möglichkeit die Verbindungsart wie Wifi oder Mobile Daten auszulesen, um den Download von Inhalten zu steuern. Außerdem bieten PWAs noch nicht ausreichend die Möglichkeit Berechnungen im Hintergrund auszuführen. Ein Download von Inhalten im Hintergrund, der Browser also geschlossen / minimiert ist, ist deswegen nicht auf allen Geräten möglich. Diese Funktionalität kann einigen Nutzern zur Verfügung gestellt werden, eine Alternative für nicht unterstützte Geräte muss aber trotzdem entwickelt werden. Wie die neuen Funktionalitäten vom Nutzer angenommen werden ist noch zu beobachten und Aufgrund dieser Erkenntnisse ist es wichtig das Portal immer wieder anzupassen.

7.2 Ausblick

Nach dem Abschluss dieser Arbeit gibt es noch viele Möglichkeiten CROSSLOAD zu verbessern. Dazu gehört zu erst die Implementierung der erarbeiteten Designs. Die Designs müssen auch regelmäßig überdacht und evaluiert werden und dementsprechend das Portal angepasst werden. Außerdem ist eine Sortierung, Filterung und Suche der Offline-Inhalte noch nicht implementiert. Dies ist vor allem für Nutzer mit sehr vielen Inhalten nützlich.

Beim Beobachten einiger Nutzer ist aufgefallen, dass Inhalte schnell heruntergeladen werden aber dann vergessen werden zu löschen. Dies führt dann dazu dass der Speicher knapp wird und evtl. sogar dazu dass keine neuen Inhalte mehr heruntergeladen werden können. Dafür könnte eine Strategie entwickelt werden wann Inhalte automatisch gelöscht werden oder dem Nutzer selbst die Möglichkeit zu geben einzustellen wann Inhalte automatisch gelöscht werden sollen. Das könnte in verschiedene Kriterien wie *wurde die Predigt schon angehört* oder *wann wurde der Inhalt heruntergeladen* unterteilt werden. Mit diesen Kriterien kann eine Bewertung abgegeben werden und die Inhalte mit der geringsten Bewertung werden gelöscht sobald Speicher benötigt wird.

Zuletzt sollte eine Synchronisation der heruntergeladenen Inhalte über mehrere Geräte entwickelt werden. Ein Nutzer möchte seine Inhalte vielleicht auf dem Laptop

auswählen und dann unterwegs auf seinem Smartphone anhören. Der Nutzer legt sich dafür einen Account bei CROSSLOAD an oder meldet sich über einen anderen Dienst wie Google oder Facebook an. Die vorgemerkten Inhalte werden dann auf einem Server von CROSSLOAD für jeden Nutzer gespeichert. Bei jedem Start des Portals, auf dem Smartphone oder Laptop, wird die Liste der vorgemerkten Inhalte geladen und gegebenenfalls neue Inhalte heruntergeladen.

Abkürzungsverzeichnis

API	Application Programming Interface
blob	binary large object
DOM	Document Object Model
GB	Gigabyte
kB	Kilobyte
MB	Megabyte
MDN	Mozilla Developer Network
TLD	Top Level Domain
PWA	Progressive Web App

Tabellenverzeichnis

3.1	Unterstützte Browser und Plattformen	14
4.1	Speicherlimits der Browser	25
4.2	Vergleich der APIs zur lokalen Datenspeicherung	26
4.3	Vergleich der Architekturen zur lokalen Datenspeicherung	34
4.4	Vergleich der Technologien zum Herunterladen von Dateien	39
6.1	Erfüllung der Anforderungen	53

Abbildungsverzeichnis

2.1	Service Worker Architektur (Sheppard 2017)	3
2.2	Publish-Subscribe Pattern UML (Mezzalana 2018)	6
2.3	Architektur von CROSSLOAD	8
2.4	CROSSLOAD - Mobile Ansicht der neusten Inhalte	8
2.5	CROSSLOAD - Detailansicht einer Predigt	9
3.1	Nutzung von CROSSLOAD nach Webbrowser auf Mobilgeräten	12
3.2	Nutzung von CROSSLOAD nach Webbrowser auf dem Desktop	12
3.3	Internetnutzung nach mobilen Browsern in Deutschland (StatCounter 2020b)	13
3.4	Internetnutzung auf PCs nach Browsern in Deutschland (StatCounter 2020a)	14
3.5	Aktivitätsdiagramm: Download des Inhalt abhängig von der Verbindungsart	17
3.6	Use Case Diagramm der Offline-Funktionalitäten von CROSSLOAD	18
4.1	Aufbau der Web Storage API	20
4.2	Struktur der File System API	21
4.3	Architektur der IndexedDB	23
4.4	Aufbau der Cache API	24
4.5	Architektur mit Cache API und Web Storage	28
4.6	Ablauf für das Favorisieren mit Cache API und Web Storage	28
4.7	Ablauf für das Abspielen mit Cache API und Web Storage	29
4.8	Architektur mit Cache API und IndexedDB	30
4.9	Architektur mit File System API und IndexedDB	31
4.10	Architektur mit IndexedDB	32
4.11	Ablauf für das Abspielen mit IndexedDB	32
4.12	Ablauf für das Favorisieren mit IndexedDB	33
4.13	Ablauf für Background Sync	37
4.14	Benachrichtigung auf Android bei Benutzung der Background Fetch API	38
5.1	Sequenzdiagramm für den Abruf der Metadaten eines Inhalts	42
5.2	Sequenzdiagramm für die Suche nach Inhalten	43
5.3	Sequenzdiagramm für den Download von Dateien	46

5.4	Player der Desktop-Ansicht	47
5.5	Design: Player der Mobilen-Ansicht	47
5.6	Design: Player der Mobilen-Ansicht	48
5.7	Design: Liste mit allen heruntergeladenen Inhalten	49

Literatur

Angular Getting started (2020). URL: <https://angular.io/start> (besucht am 15.06.2020).

Angular service worker introduction (26. Aug. 2019). URL: <https://angular.io/guide/service-worker-intro> (besucht am 14.07.2020).

Appelquist, Daniel (9. Okt. 2019). *Samsung Internet 10.2 Beta*. URL: <https://medium.com/samsung-internet-dev/samsung-internet-10-2-beta-d741ea15906d> (besucht am 23.05.2020).

Archibald, Jake (Dez. 2018). *Introducing Background Fetch*. URL: <https://developers.google.com/web/updates/2018/12/background-fetch> (besucht am 08.06.2020).

Background Sync API (2020). URL: <https://caniuse.com/#feat=background-sync> (besucht am 08.06.2020).

biblepool gUG (2020). *Crossload - Deine Tankstelle für Wachstum im Glauben*. URL: <https://crossload.org/> (besucht am 20.05.2020).

Biørn-Hansen, Andreas, Tim A Majchrzak und Tor-Morten Grønli (2017). „Progressive Web Apps: The Possible Web-native Unifier for Mobile Development.“ In: *WEBIST*, S. 344–351.

Browser storage limits and eviction criteria (16. Mai 2019). URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria (besucht am 04.06.2020).

Cache - Web APIs (15. Mai 2020). URL: <https://developer.mozilla.org/en-US/docs/Web/API/Cache> (besucht am 04.06.2020).

Fahlander, David (2020). *Dexie.js*. URL: <https://dexie.org/> (besucht am 15.06.2020).

- Fetch API* (19. Apr. 2020). URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (besucht am 08.06.2020).
- Filesystem & FileWriter API* (2020). URL: <https://caniuse.com/#feat=filesystem> (besucht am 29.05.2020).
- Firefox Platform Status* (29. Apr. 2020). URL: <https://platform-status.mozilla.org/#background-sync> (besucht am 08.06.2020).
- Google LLC (2020). *Angular*. URL: <https://angular.io/> (besucht am 20.05.2020).
- Hajian, Majid (2019). „Safety Service Worker“. In: *Progressive Web Apps with Angular: Create Responsive, Fast and Reliable PWAs Using Angular*. Berkeley, CA: Apress, S. 283–288. DOI: 10.1007/978-1-4842-4448-7_11.
- Hickson, Ian (18. Nov. 2010). *Web SQL Database*. URL: <https://www.w3.org/TR/webdatabase/> (besucht am 29.05.2020).
- HttpClient* (19. Mai 2020). URL: <https://angular.io/api/common/http/HttpClient> (besucht am 14.07.2020).
- IndexedDB* (18. Feb. 2020). URL: https://developer.mozilla.org/de/docs/Web/API/IndexedDB_API (besucht am 03.06.2020).
- IndexedDB Grundlagen* (12. Jan. 2020). URL: https://developer.mozilla.org/de/docs/Web/API/IndexedDB_API/Grundkonzepte_hinter_IndexedDB (besucht am 30.06.2020).
- Introduction to the File and Directory Entries API* (24. Sep. 2019). URL: https://developer.mozilla.org/en-US/docs/Web/API/File_and_Directory_Entries_API/Introduction (besucht am 29.05.2020).
- Josh Karlin, Marijn Kruisselbrink (28. Mai 2020). *Web Background Synchronization*. URL: <https://wicg.github.io/background-sync/spec/> (besucht am 08.06.2020).
- Khan, Asharul Islam, Ali Al-Badi und Mahmood Al-Kindi (2019). „Progressive Web Application Assessment Using AHP“. In: *Procedia Computer Science* 155, S. 289–294. DOI: <https://doi.org/10.1016/j.procs.2019.08.041>. URL: <http://www.sciencedirect.com/science/article/pii/S187705091930955X>.

- LePage, Pete (27. Apr. 2020). *Storage for the web*. URL: <https://web.dev/storage-for-the-web/> (besucht am 29. 05. 2020).
- Majchrzak, Tim A, Andreas Biørn-Hansen und Tor-Morten Grønli (2018). „Progressive web apps: the definite approach to cross-platform development?“ In:
- Mezzalana, Luca (2018). „Reactive Programming“. In: *Front-End Reactive Architectures: Explore the Future of the Front-End using Reactive JavaScript Frameworks and Libraries*. Berkeley, CA: Apress, S. 65–96. DOI: 10.1007/978-1-4842-3180-7_3. URL: https://doi.org/10.1007/978-1-4842-3180-7_3.
- Microsoft (6. März 2020a). *Häufig gestellte Fragen (FAQs) für IT-Experten*. URL: <https://docs.microsoft.com/de-de/microsoft-edge/deploy/microsoft-edge-faq> (besucht am 23. 05. 2020).
- (2020b). *Typescript - JavaScript that scales*. URL: <https://www.typescriptlang.org/> (besucht am 20. 05. 2020).
- Network Information API* (14. Mai 2020). URL: https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API (besucht am 17. 07. 2020).
- NetworkInformation.type* (23. März 2019). URL: <https://developer.mozilla.org/en-US/docs/Web/API/NetworkInformation/type> (besucht am 17. 07. 2020).
- Online and offline events* (23. März 2019). URL: https://developer.mozilla.org/en-US/docs/Web/API/NavigatorOnline/Online_and_offline_events (besucht am 09. 06. 2020).
- Online/offline status* (2020). URL: <https://www.caniuse.com/#search=navigator.online> (besucht am 09. 06. 2020).
- Request* (1. Feb. 2020). URL: <https://developer.mozilla.org/en-US/docs/Web/API/Request> (besucht am 24. 06. 2020).
- Response* (24. Jan. 2020). URL: <https://developer.mozilla.org/en-US/docs/Web/API/Response> (besucht am 24. 06. 2020).
- Rojas, Carlos (2020). *Building Progressive Web Applications with Vue.js : Reliable, Fast, and Engaging Apps with Vue.js*. Berkeley, CA: Apress. DOI: 10.1007/978-1-4842-5334-2.

- Russell, Alex u. a. (8. Juli 2020). *Service Workers Nightly*. URL: <https://w3c.github.io/ServiceWorker/> (besucht am 14. 07. 2020).
- Service worker configuration* (8. Juni 2020). URL: <https://angular.io/guide/service-worker-config> (besucht am 12. 06. 2020).
- Sheppard, Dennis (2017). *Beginning Progressive Web App Development: Creating a Native App Experience on the Web*. Berkeley, CA: Apress. DOI: <https://doi.org/10.1007/978-1-4842-3090-9>.
- StatCounter (24. Apr. 2020a). *Marktanteile der führenden Browserfamilien an der Internetnutzung in Deutschland von Januar 2009 bis März 2020*. URL: <https://de.statista.com/statistik/daten/studie/13007/umfrage/marktanteile-der-browser-bei-der-internetnutzung-in-deutschland-seit-2009/> (besucht am 09. 05. 2020).
- (24. Apr. 2020b). *Marktanteile der führenden mobilen Browser an der Internetnutzung mit Mobiltelefonen in Deutschland von Januar 2009 bis März 2020*. URL: <https://de.statista.com/statistik/daten/studie/184297/umfrage/marktanteile-mobiler-browser-bei-der-internetnutzung-in-deutschland-seit-2009/> (besucht am 09. 05. 2020).
- Storage API* (18. Mai 2019). URL: https://developer.mozilla.org/en-US/docs/Web/API/Storage_API (besucht am 04. 06. 2020).
- Web Storage API* (8. Feb. 2020). URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API (besucht am 29. 05. 2020).