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. 39 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. 39 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

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, aber auch solche mit wenig Erfahrung im Umgang mit technischen Geräten. 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. 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. Allen Nutzern soll eine komfortable

Nutzung des Webportals angeboten werden, sowohl auf Desktop-Computern, Tables oder Smartphones.

Dafür werden verschiedene Funktionen und Application Programming Interfaces (APIs) von Webbrowsern untersucht und verglichen. Die passendsten Funktionen werden ausgewählt und in das vorhandene Portal eingebunden. Dabei muss noch kein fertiges Produkt entstehen, aber die Machbarkeit gezeigt werden.

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 ersetzten?

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 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 Internet 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 bekommen, obwohl er die App nicht offen hat und aktiv benutzt (Hajian 2019) (Sheppard 2017)
- 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. 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 (Hajian 2019) (Rojas 2020). Als zweites wird ein Service Worker benötigt, der unter anderem für die Offline-Fähigkeit verantwortlich ist (Rojas 2020).

2.2 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 mehre-

re 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.

2.3 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 2.1 ist die mobile Seite mit den neusten Inhalten auf CROSSLOAD zu sehen. Abbildung 2.2 zeigt die Detailansicht einer Predigt.

Im Frontend wird Angular mit Typescript zum Entwickeln verwendet und das Backend besteht aus vielen verschiedenen Services, die mit unterschiedlichen Sprachen und Frameworks entwickelt werden. Für diese Arbeit ist nur die Technologie für das Frontend interessant.

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.

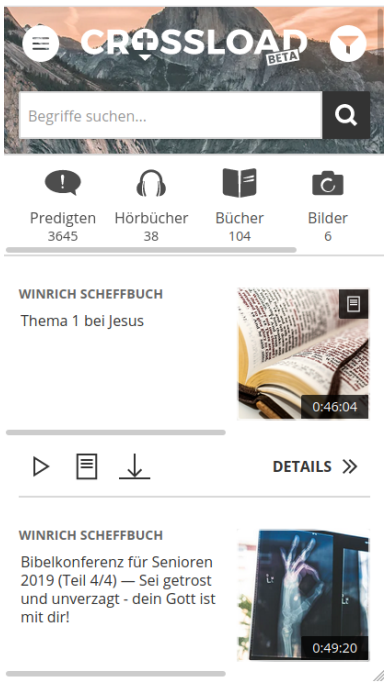

Abbildung 2.1: CROSSLOAD - Mobile Ansicht der neusten Inhalte

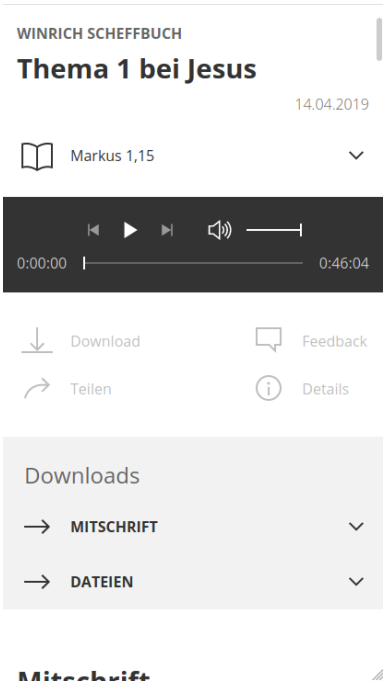

Abbildung 2.2: CROSSLOAD - Detailansicht einer Predigt

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 zusammen im Team diskutiert. Auch für die Offline Funktionalitäten gab es ein Brainstorming mit Mitarbeitern von CROSSLOAD. Daraus haben sich einige Anforderungen herausgestellt. Zusätzlich werden sich einige Szenarien überlegt, die verschiedene Nutzungen der Offline Funktionalität beleuchtet. Die Szenarien helfen beim Verstehen der Anforderungen.

Es gibt zwei Randbedingungen, die für diese Thesis 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, deswegen gibt es noch nicht genügend Daten darüber, welche Browser von den Nutzern von CROSSLOAD verwendet werden. Als Anhaltspunkt dient deswegen die durchschnittliche Browsernutzung in Deutschland. Dabei kann zwischen der

Nutzung der Browser auf PCs wie in Abbildung 3.1 zu sehen und auf Mobilgeräten wie in Abbildung 3.2 zu sehen unterschieden werden.

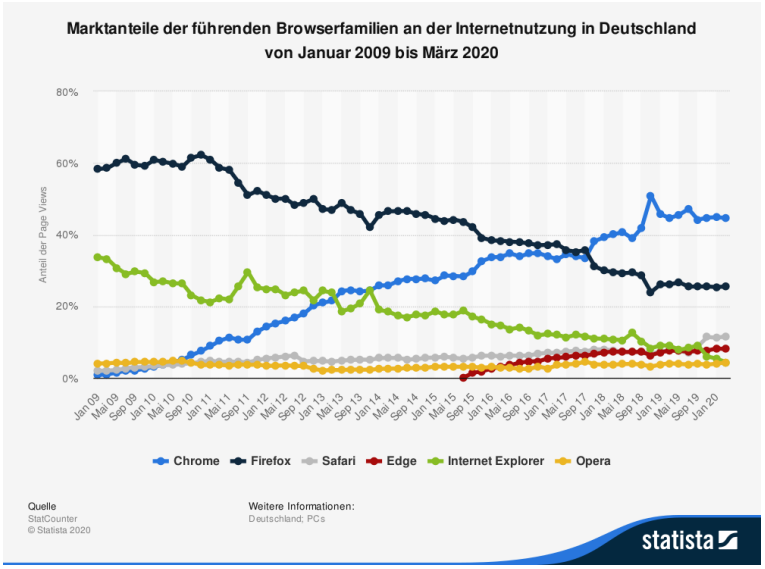

Abbildung 3.1: Internetnutzung nach Browsern in Deutschland (StatCounter 2020a)

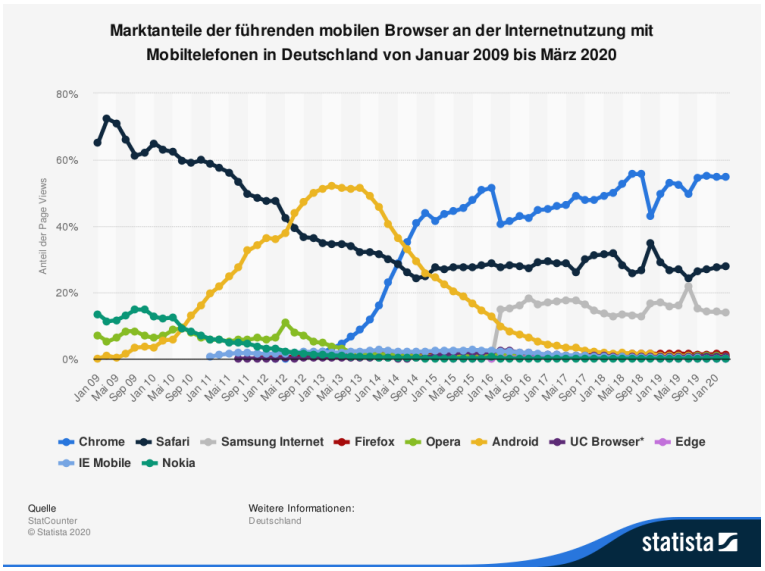

Abbildung 3.2: Internetnutzung nach mobilen Browsern in Deutschland (StatCounter 2020b)

Die Offline-Funktionen für CROSSLOAD, die in dieser Thesis erarbeitet werden, sind nicht essentiell zum Benutzen des Portals. Deswegen kann auch in kauf genommen werden, dass manche Funktionen für Nutzer mit gewissen Browsern nicht zur Verfügung stehen. Natürlich ist es besser, wenn viele Nutzer von den neuen Funktionalitäten profitieren. Wichtig ist nur, dass iOS und Android unterstützt werden.

Dabei wird nicht berücksichtigt, welche Browserversion die Nutzer verwenden. Alle gängigen Browser verfügen über eine automatische Updatefunktion, dadurch erhalten die meisten Nutzer sehr zeitnah veröffentlichte Updates.

Bei den Browsern für Mobilgeräte ist der Marktanteil von Chrome, Safari und Samsung Internet zusammengerechnet über 96%. Außerdem basiert Samsung Internet intern auf Chromium und teilt somit die allermeisten Funktionen mit Chrome (Appelquist 2019). In späteren Entscheidungen wird deswegen Samsung-Internet nur erwähnt wenn es von der Funktionalität im Chrome Browser abweicht.

Für PCs gibt es mehr Browserfamilien, die relevant sein könnten. Insgesamt lassen sich alle relevanten Browser auf drei Gruppen reduzieren: Safari und Firefox haben jeweils eine eigene Render-Engine. Opera setzt schon seit einigen Jahren auf Chromium auf (Opera Software 2013) und auch für den Edge-Browser wurde 2018 angekündigt Chromium als Basis zu verwenden (Microsoft 2020a). Der Internet Explorer wird immer weniger verwendet und von Microsoft nicht mehr für den normalen Endnutzer empfohlen (Microsoft 2020a), deswegen wird der Internet Explorer hier auch nicht berücksichtigt. In den nachfolgenden Kapiteln wird der Browser Edge und Opera mit Chrome gleichgesetzt und nicht extra erwähnt, solange keine relevanten Unterschiede vorhanden sind. Die Browser Safari, Firefox und Chrome (inklusive der Chromium basierten Browser) haben einen Marktanteil von über 94%.

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.

- Der Benutzer kann sich einen Inhalt vormerken, auch favorisieren genannt
- Der vorgemerkte Inhalt wird automatisch anhand der Verbindungsart heruntergeladen. Der genaue Ablauf ist in einem Aktivitätsdiagramm in Abbildung 3.3 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
- Der Benutzer wird über den Fortschritt des Downloads informiert
- Das Portal erkennt wenn ein Benutzer keine Internetverbindung hat und leitet ihn auf eine spezielle Seite weiter
- Die zurzeit heruntergeladenen Inhalte können in einer Übersicht gesehen werden

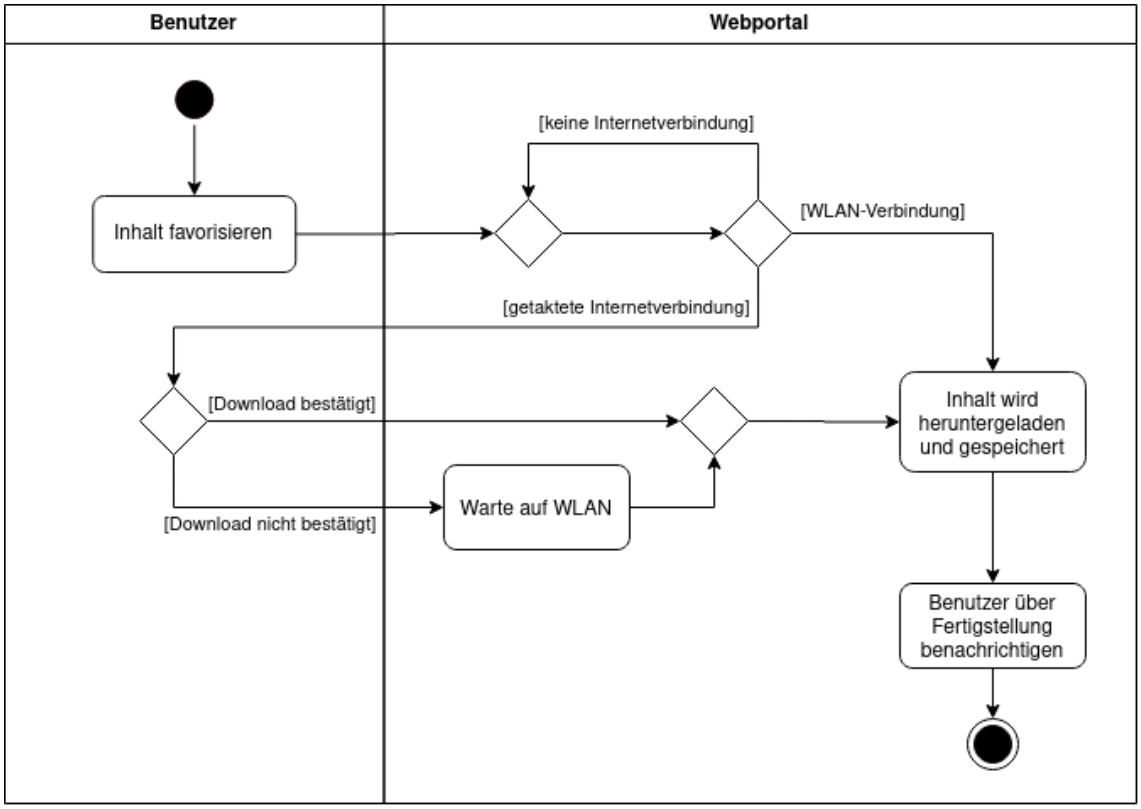

Abbildung 3.3: Aktivitätsdiagramm: Download des Inhalt abhängig von der Verbindungsart

  - Schön wäre es wenn die Inhalte auch gefiltert oder sortiert werden könnten. Zum Beispiel nach Download-Datum. Diese Anforderung muss aber nicht erfüllt werden.

- Die zurzeit heruntergeladenen Inhalte können gelöscht werden
- Es soll möglich sein, für einen bestimmten Inhalt herauszufinden, ob dieser Offline verfügbar ist
- Die Inhalte der Detailseite eines heruntergeladenen Inhalts werden gespeichert. Dazu gehören zum Beispiel: Id, Titel, Autor oder Vorschaubild
- 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

Kapitel 4

Konzeption

4.1 Lokale Datenspeicherung

Moderne Webbrowser stellen viele verschiedene Lösungen zum lokalen Speichern von Daten zu Verfügung. In diesem Kapitel werden einige davon mit ihren Stärken und Schwächen vorgestellt. Zuletzt wird eine passende Lösung für CROSSLOAD ausgewählt und begründet.

4.1.1 Web Storage

Der Web Storage besitzt zwei unterschiedliche Arten Daten zu speichern: LocalStorage und SessionStorage (Hajian 2019) (*Web Storage API* 2020). Die beiden Speicher unterscheiden sich nur in der Dauer der Speicherung der Daten. 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).

Die Web Storage API bietet die Möglichkeit Schlüssel / Wert Paare im Browser abzulegen (*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)
- Als Schlüssel und Werte können jeweils nur Strings gespeichert werden (Hajian 2019)

- Die API ist nicht von Web Workern aufrufbar (Hajian 2019). Service Worker haben zum Beispiel keinen Zugriff darauf
- Es können maximal 5 Megabyte (MB) an Daten gespeichert werden (*Web Storage API* 2020)

Aufgrund der Synchronität und des geringen Datenvolumens das gespeichert werden darf, eignet sich die Web Storage API nur zum Speichern von wenig Daten. Größere Daten und insbesondere binäre Dateien, die nicht sehr gut in eine textuelle Form übertragen werden können sollten nicht abgespeichert werden.

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). Dieses virtuelle Dateisystem unterstützt viele gängige Funktionen wie Ordner und Dateimanipulationen, wie man sie von herkömmlichen Dateisystem gewohnt ist (*Introduction to the File and Directory Entries API* 2019). Mozilla Developer Network (MDN) nennt diese API File and Directory Entries API (*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). Außerdem gibt es eine API für synchrone und asynchrone Zugriffe und ist ebenso in Web Workern verfügbar (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). Die

Anfragen sind asynchron funktionieren aber nicht in einem Web Worker (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. Als Datenbank in Browsern hat sich IndexedDB etabliert.

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 unterstützt (Sheppard 2017).

Einschränkungen in der Verwendung der API gibt es nicht, da IndexedDB auch in Web Worker zur Verfügung steht und asynchron verwendet werden kann (Hajian 2019) (*IndexedDB Grundlagen* 2020). Alle relevanten Browser unterstützen IndexedDB (*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).

Die Verwendung von IndexedDB ist komplex, deswegen existieren viele Bibliotheken, die eine Verwendung der API erleichtern wollen, wie zum Beispiel LocalForage oder Dexie.js (Hajian 2019) (*IndexedDB Grundlagen* 2020).

IndexedDB ist ein sehr flexibler Speicher für den Client, weil in einer Anfrage direkt gesucht und sortiert werden kann. Außerdem können simple Datentypen, Objekte und sogar blobs (binäre Dateien) gespeichert werden. Ein Nachteil ist die Komplexität in der Verwendung der API.

4.1.5 Cache API

Die Cache API bietet die Möglichkeit Netzwerkanfragen zwischenzuspeichern (*Cache - Web APIs* 2020). Über Request-Objekte können Response-Objekte gespeichert und gefunden werden (*Cache - Web APIs* 2020). Die API kann über das window-Ojekt und in Web-Workern verwendet werden und ist in allen relevanten Browsern verfügbar (*Cache - Web APIs* 2020).

Oft wird die Cache API in Verbindung mit Service Workern verwendet. Service Worker laufen im Hintergrund, sogar wenn die Webseite geschlossen wurde, haben aber keinen Zugriff aufs 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 4.1 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.

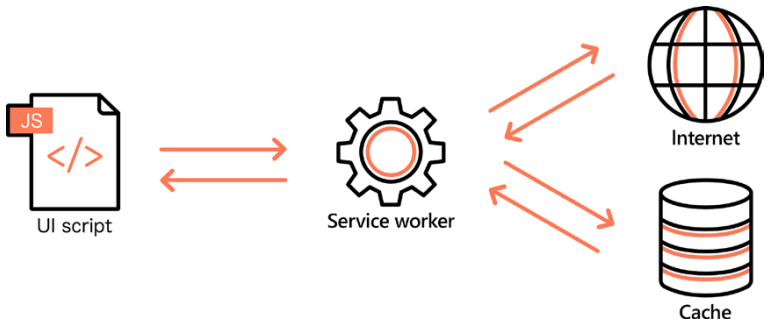

Abbildung 4.1: Service Worker Architektur (Sheppard 2017)

Die maximale Speicherkapazität des Caches hängt vom Browser ab und wird in Kapitel 4.1.6 näher erläutert.

Zusammengefasst ist die Cache API eine gute Möglichkeit Netzwerkanfragen zu cachen und somit die Website offline-fähig zu machen. Außerdem kann dadurch auch die Geschwindigkeit der Seite verbessert werden und eine gute Nutzererfahrung gewährleistet werden. Ein Nachteil ist, dass nur Request- und Response-Objekte gespeichert werden können.

Tabelle 4.1: Speicherlimits der Browser

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

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, www.hs-mannheim.de und bib.hs-mannheim.de gehören. Eine weiter 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 folgenden Kriterien verglichen: maximale Datenmenge, mögliche Datentypen, synchron / asynchron, Browsersupport, in Web Worker aufrufbar. Bei fehlenden Feldern in Web SQL wurde nicht weiter recherchiert, weil diese Technologie veraltet ist und nicht mehr verwendet werden soll.

Durch die bisherigen Nachforschungen ist klar, dass in CROSSLOAD entweder IndexedDB oder die Cache API verwendet werden sollte. Es ist auch möglich beides zu verwenden. Alle anderen vorgestellten Technologien werden entweder nicht von allen kompatiblen Browsern unterstützt oder können nur eine geringe Datenmenge speichern. Diese Technologien können als Ergänzung eingesetzt werden, zum Speichern der großen Datenmengen sind sie jedoch nicht geeignet.

Es werden nun mögliche Architekturen vorgestellt und miteinander verglichen.

Cache API und Web Storage

Die erste Architektur benutzt die Cache API zum Speichern der Audio-Dateien und den Web Storage zum Speichern der Metadaten. Metadaten sind in diesem Fall alle Information, die zu einem Inhalt gehören abgesehen von der Audio-Datei. In Abbildung 4.2 ist der Ablauf dargestellt wenn ein Nutzer einen Inhalt favorisiert oder sich alle Inhalte anschaut und einen davon auswählen möchte.

Beim Speichern der Daten im Web Storage gibt es zwei verschiedene Ansätze. Der erste Ansatz speichert alle Daten in einem Array wie in Listing 4.1 zu sehen. Der zweite Ansatz speichert alle favorisierten Ids in einem Array und jede Id ist ein Schlüssel im Web Storage über den dann die eigentlichen Metadaten abgerufen werden können. Dies ist in Listing 4.2 veranschaulicht. Der erste Ansatz ist besser wenn alle Daten auf einmal angefragt werden. Wenn aber nur geprüft werden soll, ob ein Element favorisiert ist, muss das ganze Array durchsucht werden. Umgekehrt ist der zweite Ansatz sehr schlecht wenn es darum geht alle Daten zu holen, weil sehr viele Anfragen benötigt werden. Das Anfragen eines einzelnen Inhalts ist jedoch sehr schnell. Außerdem ist es beim zweiten Ansatz effizienter einen Inhalt zu aktualisieren, weil nicht das ganze Array erneut gespeichert werden muss sondern nur ein Objekt.

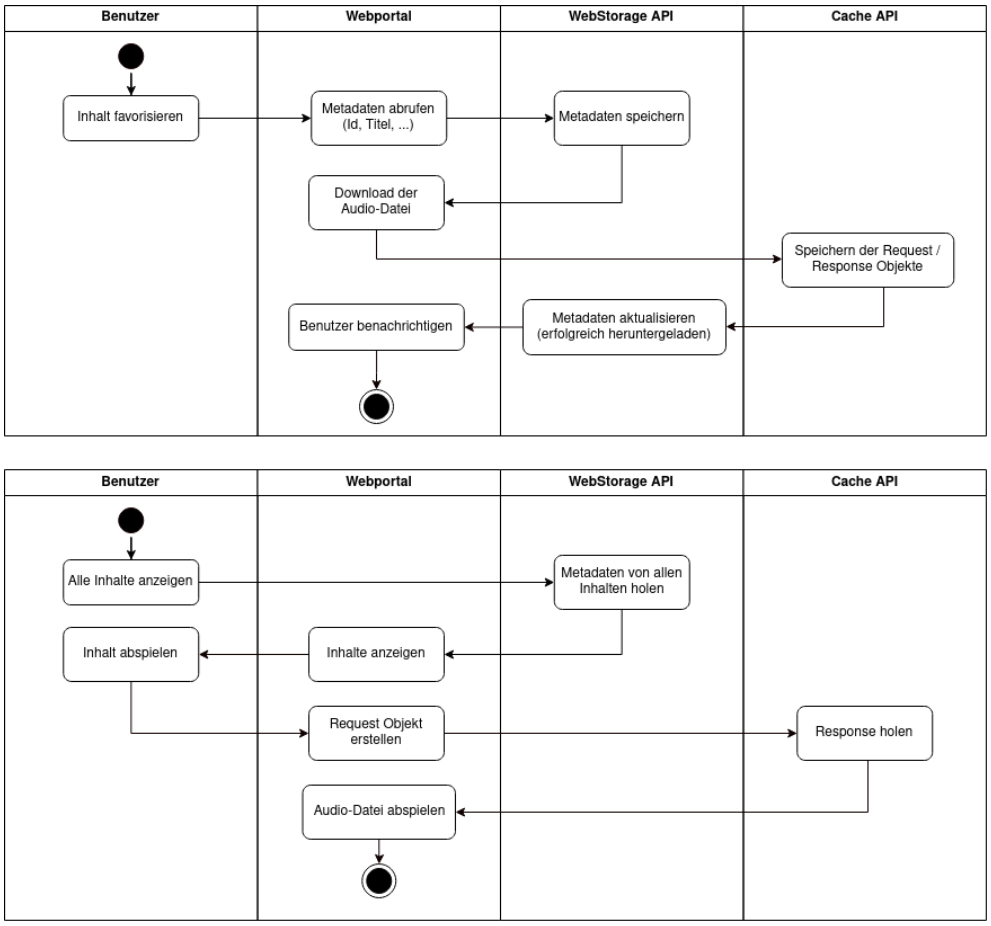

Abbildung 4.2: Architektur mit Cache API und Web Storage

Tabelle 4.2: Vergleich der APIs zur lokalen Datenspeicherung

Technologie	Datenmenge	Datentypen	(a)synchron	Browsersupport	Web Worker	Anmerkung
Web Storage API	max. 5 MB	String	nur synchron	alle	nein	
File System API	max. Speicherkapazität des Browsers	blobs und weitere	synchron und asynchron	nur Chrome	Ja	
Web SQL	-	.	synchron und asynchron	teilweise	nein	veraltet
IndexedDB	max. Speicherkapazität des Browsers	einfache Datentypen, Objekte, blobs	asynchron	alle	Ja	Inhalt durchsuch- und filterbar
Cache API	max. Speicherkapazität des Browsers	Request- und Response-Objekte	asynchron	alle	Ja	

```
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

```
favoriten: ["id1", "id2"]
id1: {
  titel: "Titel des 1. Inhalts",
  erstelltAm: "05.06.2020",
  // weitere Metadaten
}
id2: {
  // Metadaten
}
```

Listing 4.2: Speichern der Ids in einem Array

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. Die Nachteile dieser Architektur überwiegen jedoch:

- 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.

Als Alternative zum Web Storage kann IndexedDB verwendet werden, was in der nächsten Architektur vorgestellt wird.

Cache API und IndexedDB

Die Architektur bleibt wie in Abbildung 4.2, mit dem Unterschied, dass Web Storage durch IndexedDB ersetzt wird. Durch das Ersetzen vom Web Storage durch IndexedDB, fallen viele Nachteile weg. Das Abrufen und Speichern der Metadaten

ist nicht synchron, sondern asynchron. Außerdem bietet IndexedDB die Möglichkeit Inhalte zu sortieren und zu filtern, was für CROSSLOAD auch interessant ist. Der Nachteil ist die hinzugekommene Komplexität.

Die Komplexität kann verringert werden, indem weniger verschiedene APIs eingesetzt werden. Auf IndexedDB soll nicht verzichtet werden, weil diese Vorteile wie Sortierung und Filterung bietet.

IndexedDB

Abbildung 4.3 zeigt die Architektur nur mit IndexedDB zum Speichern der Daten. Sowohl Metadaten als auch die Audio-Datei werden in der Datenbank abgespeichert.

Durch das Ersetzen von der Cache API durch IndexedDB sind keine Nachteile entstanden. Die Komplexität des Webportals wird nicht erhöht, sondern eher etwas verringert, weil nicht mehr so viele verschiedene APIs verwendet werden.

Entscheidung

Zum Speichern der Daten wird allein auf IndexedDB gesetzt. Die zusätzliche Verwendung von der Cache API bietet keine weiteren Vorteile und die Verwendung vom Web Storage wird aufgrund der Synchronität abgelehnt.

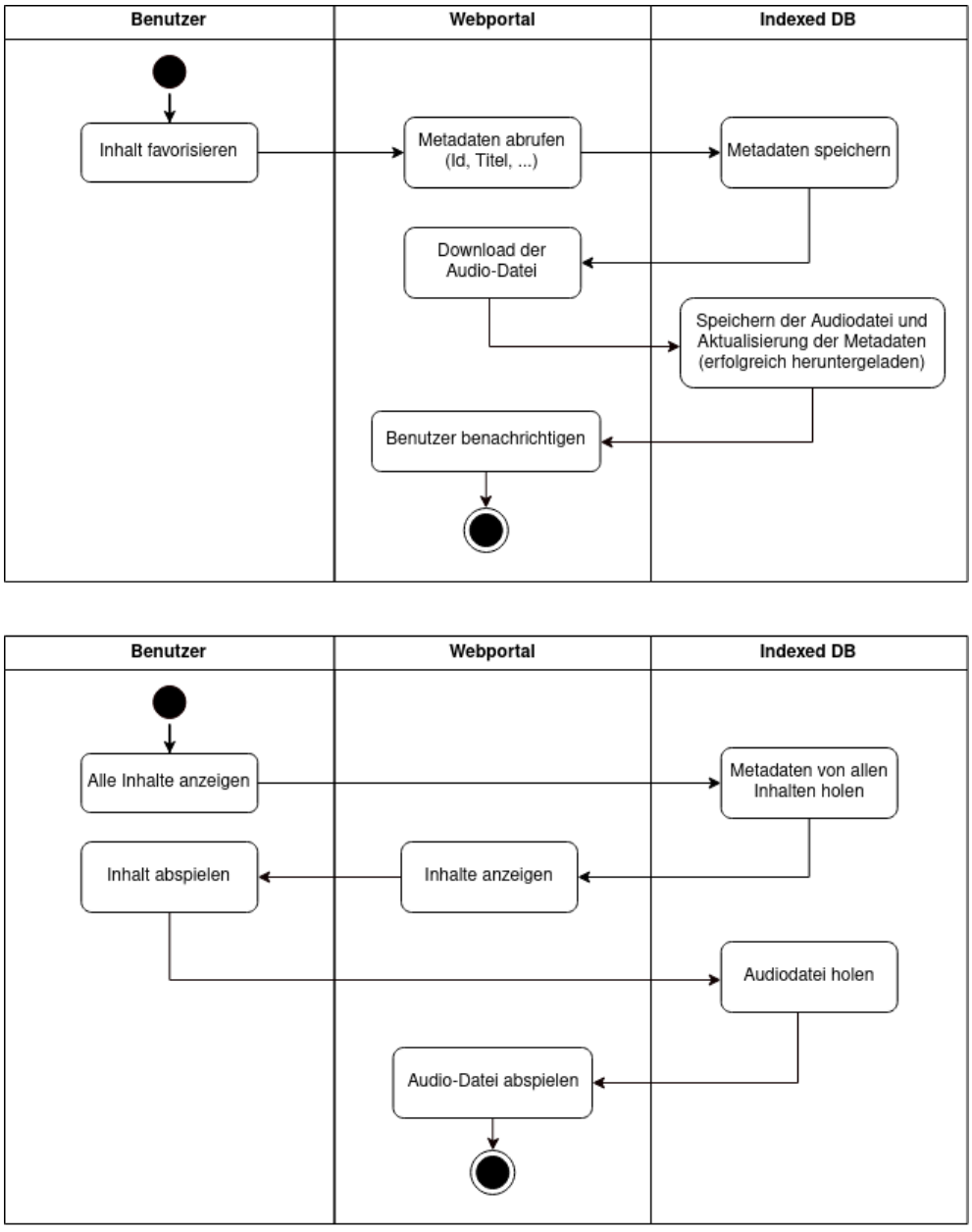

Abbildung 4.3: Architektur nur mit IndexedDB

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.

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).

Diese API ist sehr flexibel und kann deswegen sehr gut in service workern oder mit der Cache API verwendet werden (*Fetch API* 2020). Außerdem unterstützt jeder relevante Browser diese Funktion (*Fetch API* 2020).

Damit eine Netzwerkfrage abgeschickt und empfangen werden kann muss das Browserfenster aber geöffnet sein. Es ist nicht möglich eine Anfrage zu schicken und dann den Browser zu verlassen.

In Angular werden Netzwerkanfragen mit dem HttpService verschickt. Dies ist in diesem Vergleich mit der Fetch API gleichzusetzen, weil die gleichen Funktionen vorhanden sind und auch die gleichen Einschränkungen gelten.

4.2.2 Background Sync

Mit Background Sync können Netzwerk Anfragen abgeschickt werden, selbst wenn keine Internetverbindung besteht. Die Anfrage wird gespeichert, bis eine Internetverbindung besteht und dann abgeschickt (Josh Karlin 2020). Diese Funktionalität ist Teil von Service Workern und funktioniert solange ein Service Worker läuft (Josh Karlin 2020).

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).

4.2.3 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 4.4 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).

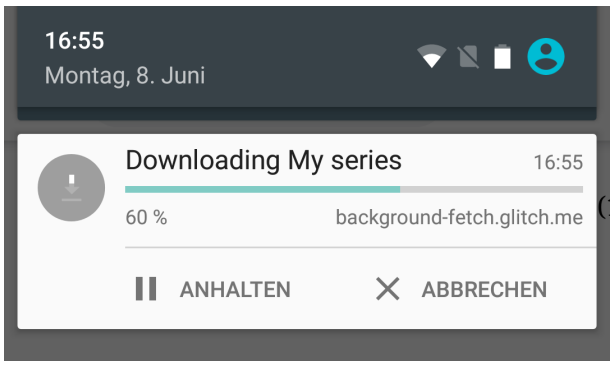

Abbildung 4.4: Benachrichtigung auf Android bei Benutzung der Background Fetch API

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

4.2.4 Entscheidung

Die einzige API, die von allen relevanten Browsern unterstützt wird, ist die Fetch API und wird deswegen auch verwendet. Als Ergänzung kann später Background Sync oder Background Fetch implementiert werden, da diese auch wenn die Website geschlossen ist Inhalte herunterladen können. Background Sync wird wahrscheinlich schneller eine größere Browserunterstützung finden und sollte deswegen bevorzugt werden.

4.3 Verbindungsstaus auslesen

Um eine gute Nutzererfahrung bieten zu können, muss man zu jeder Zeit wissen, ob gerade eine Internetverbindung besteht oder nicht. Außerdem möchte man so schnell wie möglich eine Änderung der Verbindung mitbekommen.

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 kann einen einfachen Request abschicken. 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.

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.

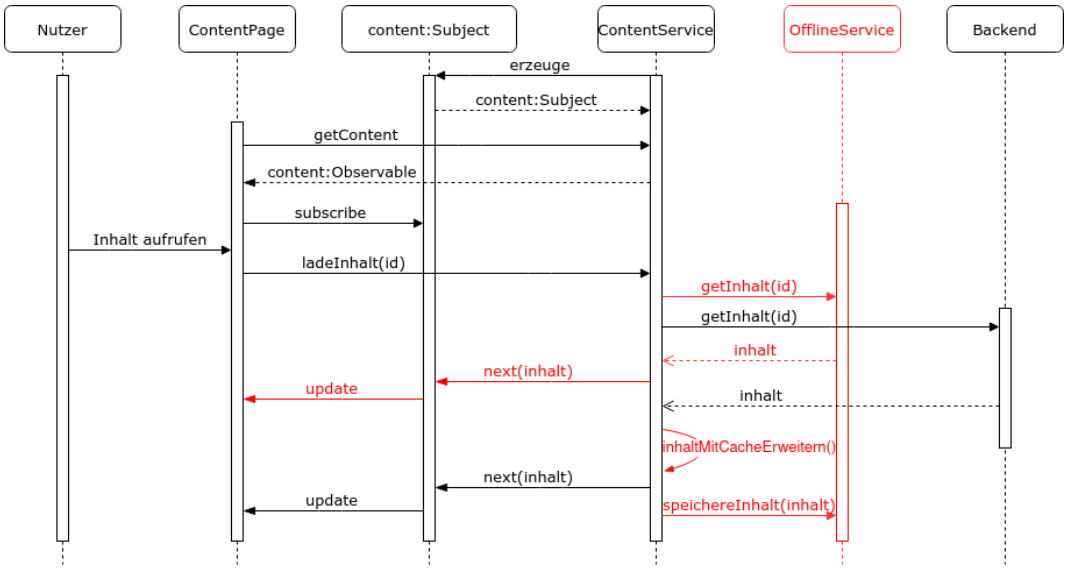

Abbildung 5.1: Sequenzdiagramm für den Abruf der Metadaten eines Inhalts

In Abbildung 5.1 ist die bereits beschriebene Cache-Strategie beschrieben. 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-Subsribe 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 Eigenschaf-

ten 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:

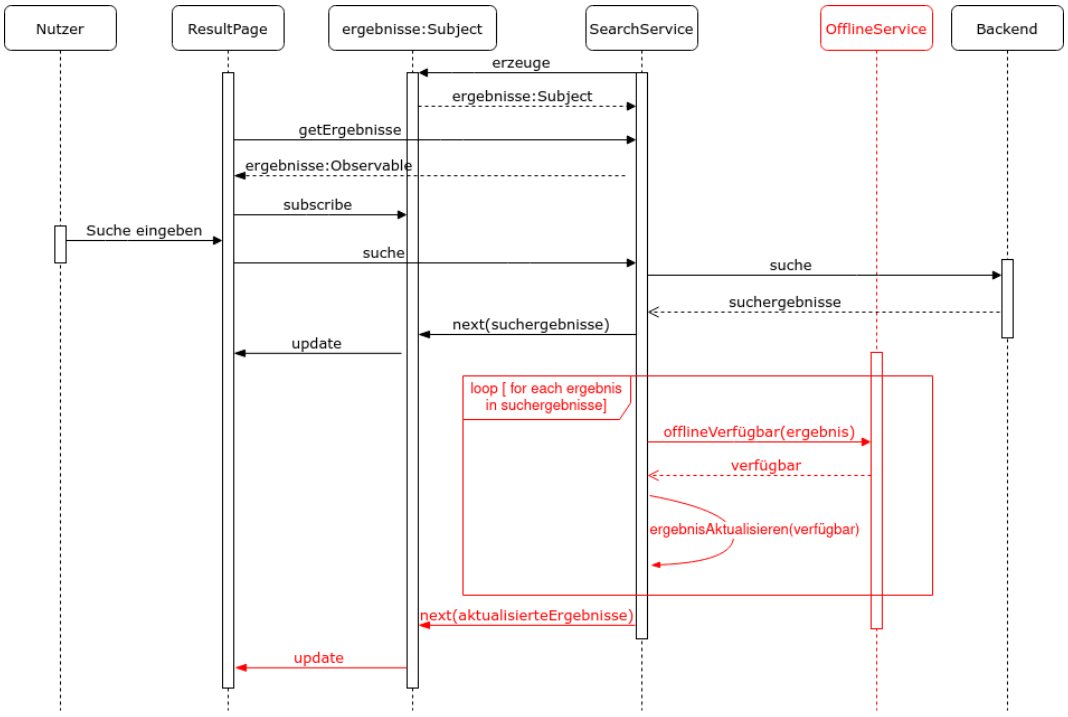

Abbildung 5.2: Sequenzdiagramm für die Suche nach Inhalten

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 Ergänzun-

gen 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 gefunden 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*.

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.

Die Tabelle *downloads* speichert eine id und die Audiodatei. Es ist nicht möglich zu einer id mehrere Dateien zu speichern. Es wäre auch möglich gewesen die Audi-

odatei mit dem content-Objekt zu speichern. Dies hätte aber den Nachteil das jedes mal die große Audiodatei aus der Datenbank geladen werden muss wenn die Metadaten zu einem Inhalt geladen werden. In IndexedDB gibt es keine Möglichkeit nur einen Teil eines Objektes zu laden. Durch die Trennung von Metadaten und Audiodaten wird die Anwendung also schneller. Der Nachteil dabei ist, dass dadurch eine Anfrage mehr and die Datenbank gestellt werden muss, um herauszufinden ob eine Audio-Datei für einen Inhalt verfügbar ist.

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 Fetch API bzw. dem HttpService von Angular implementiert. Eine zusätzliche Implementierung von der Background Sync API 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.

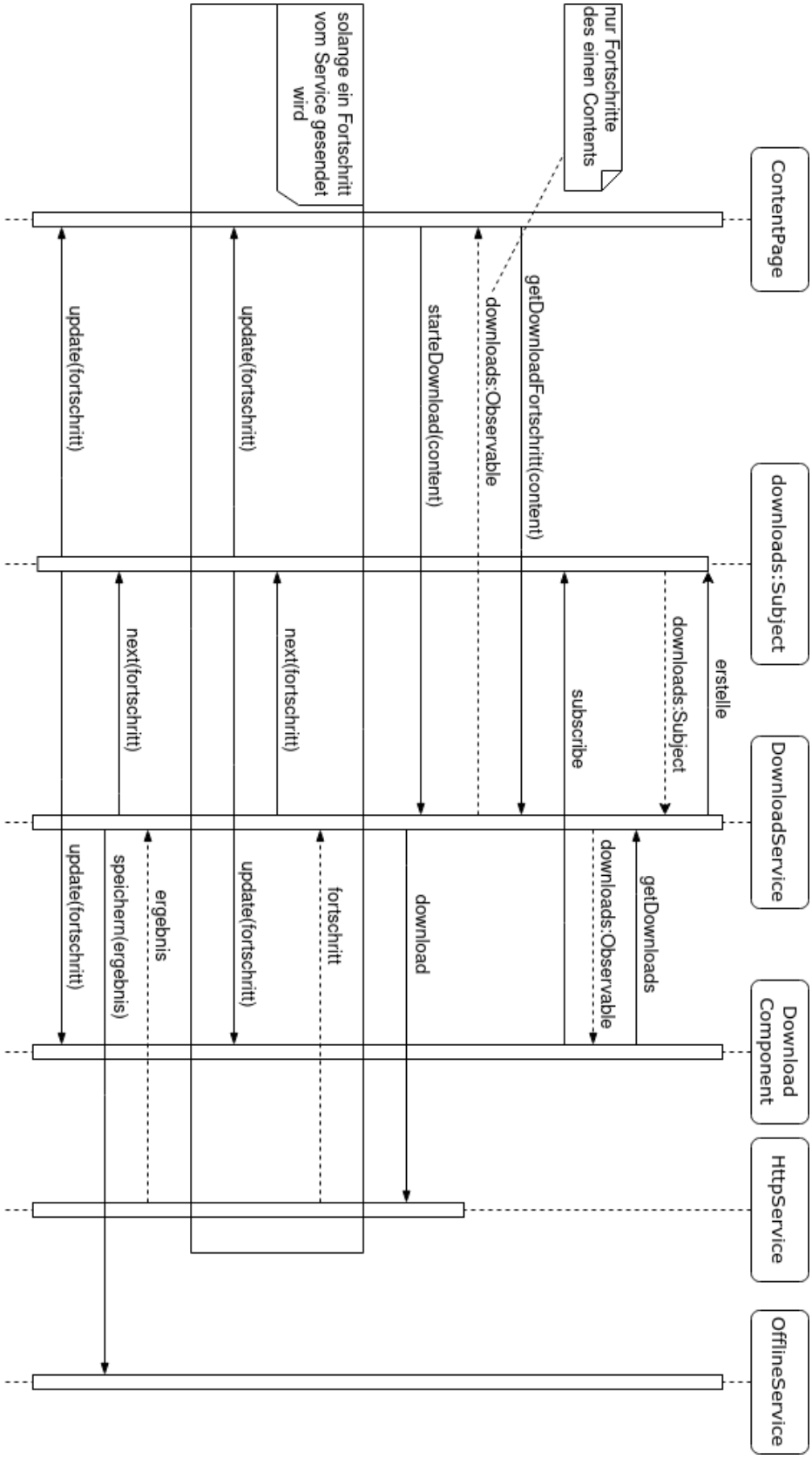

Abbildung 5.3: Sequenzdiagramm für den Download von Dateien

5.4 Grafische Oberfläche

Kapitel 6

Evaluation und Reflexion

6.1 Kann durch die PWA eine native APP ersetzt werden?

6.2 Nutzerfreundlichkeit

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

7.2 Ausblick

- Bei anderen Nutzern wurde beobachtet, dass Predigten nicht manuell gelöscht werdne, wenn man die mp3-Dateien heruntergealden hat. Evtl. automatisches Löschen angehörtet Predigten

- Heruntergeladene Inhalte sortieren / filtern / suchen

- Die Favoriten sollen über einen Account synchronisiert werden können

Abkürzungsverzeichnis

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

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

Angular Getting started (2020). URL: https://angular.io/start (besucht am 15. 06. 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).

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:

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).

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).

Opera Software (13. Feb. 2013). *Opera gears up at 300 million users*. URL: https://press.opera.com/2013/02/13/opera-gears-up-at-300-million-users/ (besucht am 23. 05. 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.

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).