Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

SEBASTIAN KNAUER
MASTER THESIS

OFFLINE EXPERIENCE FOR WEB APPLICATIONS

Eingereicht am 22. Oktober 2018

Betreuer: Prof. Dr. Dirk Riehle, M.B.A., Andreas Kaufmann, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 22. Oktober 2018

License

Erlangen, 22. Oktober 2018

Abstract

In the recent years, the popularity of Web based applications, in contrast to conventional desktop applications, has risen significantly. The advantages include platform independence, easy collaboration possibilities, decreased maintenance costs and more. Never the less, being web based results in a major disadvantage. To successfully use the application, an active connection to the internet is required. This makes working in offline environments or with intermediate long-lasting connection losses impossible. This thesis presents a possible solution to this problem at the example of the web application QDAcity[1]. The solution is based on the Service Worker API that is a new technology used in Progressive Web Apps. The core functionality of this application can now be reliably used even without an active internet connection or during reoccurring connection losses. After reconnecting, the changes the user made while being offline, are synchronized so that the data on the server and the client contain the most recent version. This solution improves web applications by making them more resilient against connection losses, that can occur in mobile office environments or on travel and makes them significantly more attractive compared to desktop applications.

[1] qdacity.com

Zusammenfassung

Die Popularität von Webanwendungen ist gegenüber konventionellen Desktop Anwendungen in den letzten Jahren deutlich gestiegen. Die Vorteile beinhalten Plattformunabhängigkeit, einfache Möglichkeiten zur Kollaboration, verringerter Wartungsaufwand und viele weitere. Dennoch ergibt sich bei webbasierten Anwendungen ein großer Nachteil: für die erfolgreiche Benutzung der Anwendung ist ein aktive Internetverbindung nötig. Das macht das Arbeiten mit der Anwendung in Offline-Umgebungen, bzw. bei immer wiederkehrenden, auch länger andauernden Verbindungsverlusten, unmöglich. Diese Arbeit stellt eine mögliche Lösung für dieses Problem, am Beispiel der Webanwendung QDAcity[2], vor. Die Lösung basiert auf der Service Worker API, einer neuen Technologie, die in Progressive Web Apps benutzt wird. Die Webanwendung kann nun ohne Internetverbindung bzw. bei auftretendem Verbindungsverlust in ihren Kernfunktionalitäten zuverlässig benutzt werden. Bei Wiederherstellung der Verbindung werden die Änderungen, die der Benutzer ohne Verbindung durchgeführt hat, synchronisiert, sodass die Daten des Clients und des Servers auf dem aktuellsten Stand sind. Diese Lösung verbessert Webanwendungen, indem sie sie widerstandsfähiger gegenüber Verbindungsverlusten macht. Verbindungsverluste sind in mobilen Arbeitsumgebungen oder auf Reisen keine Seltenheit. Dadurch werden Webanwendungen gegenüber konventionellen Desktop Anwendungen deutlich attraktiver.

[2] qdacity.com

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Die Zahl der Nutzer, die mit einem mobilen Endgerät internetbasierte Inhalte konsumieren, belief sich in Deutschland im ersten Quartal 2018 auf 87% aller Internetnutzer in Deutschland (Statistisches Bundesamt, 2018), Tendenz steigend. Folglich ist auch in den letzten Jahren die Nachfrage nach Technologien, die den Markt auf mobilen Plattformen bedienen können, deutlich gestiegen. Aufgrund des hohen Aufwands, um native mobile Anwendungen für unterschiedliche Plattformen wie iOS und Android zu entwickeln, ist die Cross-Platform Entwicklung sehr attraktiv (Malavolta et al, 2015). Es existieren dabei verschiedene, populäre Lösungen, wie PhoneGap[1], Xamarin[2] oder React Native [3]. Mit diesen lassen sich hybride, interpretierte und native Anwendungen erstellen. Im Bezug auf die Nutzerfreundlichkeit auf mobilen Geräten waren die, mit diesen Werkzeugen entwickelten Anwendungen, klassischen Webanwendungen unter vielen Gesichtspunkten voraus. Progressiv Web Apps (PWA), ein von der Google Web Fundamentals Group entwickelter Ansatz, soll die durch Webanwendungen gebotenen Nutzererfahrung nativer werden lassen (Osmani, 2015). Unter den Hauptargumenten finden sich folgende Eigenschaften: fortschrittlich (progressive), zugänglich (responsive), verbindungsunabhängig (connectivity independent), *app*-like, aktuell (fresh), sicher (safe), installierbar (installable).

Dabei kann der Ansatz von PWAs auch auf klassische Webanwendungen angewendet werden, um diese nativer (*App*-like) werden zu lassen. Ein Ansatz von PWAs, der nicht zwangsläufig an Webanwendungen gebunden ist, die auf mobile Plattformen ausgerichtet sind, ist der Aspekt der Offline Nutzung der Anwendung (connectivity independent). Nutzer, die regelmäßig eine Anwendung an Orten mit schlechter Internetverbindung (z.B. in Zügen) nutzen, sind darauf angewiesen, dass diese auch hier ordnungsgemäß funktioniert.

Eine dieser Webanwendungen, die davon profitieren kann ist QDAcity. QDAcity ist ein Werkzeug, das von Forschern zur qualitativen Datenanalyse genutzt wird. Es wird zurzeit aktiv am Lehrstuhl für Open Source Software an der Friedrich-

[1] https://phonegap.com/
[2] https://visualstudio.microsoft.com/xamarin/
[3] http://reactnative.com/

Alexander-Universität Erlangen Nürnberg entwickelt.

Diese Arbeit behandelt den Aspekt der Offline Nutzung der Webanwendung. Dabei wurde eine Erweiterung mittels eines Service Workers implementiert, sodass einige Kernfunktionalitäten der Anwendung, auch bei Unterbrechung der Verbindung zum Server, verfügbar sind.

Die Arbeit ist folgendermaßen strukturiert: *Kapitel 2* beschreibt die Grundlagen eines Service Workers. Die Anforderungen, die diesem Projekt zugrunde liegen werden in *Kapitel 3* erläutert. *Kapitel 4* stellt die grobe Architektur, sowie für die Implementierung notwendige Designentscheidungen vor. Diese wird folgend in *Kapitel 5* behandelt. In *Kapitel 6* werden die Erfüllungen der Anforderungen evaluiert. Schließlich wird in *Kapitel 7* ein Fazit gezogen und auf die künftige Arbeit eingegangen.

2 Grundlagen

2.1 Service Worker

Um die Offline Funktionalität zur Verfügung zu stellen wird ein Service Worker benutzt. Die Service Worker API ist eine recht junge Technologie, deren Spezifikation von der Service Worker Working Group entworfen wurde (Alex Russell, Jungkee Song, Jake Archibald, Marijn Kruisselbrink, 2018b). Die Service Worker Working Group ist Teil des World Wide Web Consortium (W3C) und wird geleitet von Jake Archibald und Jatinder Mann, die Mitarbeiter des Google Chrome Teams bzw. Microsoft sind. Trotz der sich in Entwicklung befindenden API werden dennoch die in dieser Arbeit genutzten Features von den meisten modernen Browsern unterstützt (MDN, 2018).

Funktionalität	**Chrome**	**Firefox**	**Edge**	**Safari**	**Opera**
Basic support	40	44	17	N/A	24
install/activate events	40	44	17	N/A	Ja
fetch event/request/`respondWith()`	40	44	17	N/A	N/A
caches/cache	42	39	17	N/A	N/A
MessageEvent	57	55	N/A	N/A	N/A
NavigationPreloadManager	59		N/A	N/A	N/A

Tabelle 2.1: Kompatibilität der Funktionalitäten des Service Workers in unterschiedlichen Browsern. Die Werte sind die Versionsnummer des Browser, ab der er diese Funktionalität unterstützt.

Die Service Worker API wird als Nachfolger der zuvor für Offline Verfügbarkeit von Webseiten benutzten Application Cache Api (AppCache)[1] betrachtet. Diese wird aufgrund zahlreicher Mängel in den Designentscheidungen dieser API als veraltet betrachtet. Unter diesen Mängeln befinden sich unter anderem Folgende:

- Dateien werden immer vom AppCache geliefert, selbst wenn man online ist

[1] https://www.w3.org/TR/2011/WD-html5-20110525/offline.html#appcache

- Der AppCache wird nur aktualisiert, wenn sich der Inhalt des Manifests ändert
- Nicht gecachte Ressourcen werden auf einer gecachten Seite nicht geladen

Es wird daher empfohlen, stattdessen auf die Service Worker API zurückzugreifen (Steering Group, 2018).

Standardmäßig ist JavaScript single-threaded, läuft also in einem einzelnen Thread innerhalb des Browsers. Damit beispielsweise rechenintensiver Programmcode die Benutzeroberfläche nicht blockiert, was zu einem schlechten Anwendererlebnis führen kann, kann dieser in einen WebWorker ausgelagert werden. Der WebWorker läuft dann in einem eigenen Thread neben dem main thread und kann diesen somit nicht mehr blockieren. Der Service Worker ist ein spezieller WebWorker, wird also auch in einem separaten Thread ausgeführt. Er verhält sich wie ein Proxy zwischen dem Netzwerk und der eigentlichen Webanwendung. Das bedeutet, nachdem die Anwendung den Service Worker registriert hat, werden alle HTTP Requests erst vom Service Worker entgegengenommen. Dieser kann die Requests dann bearbeiten, weiterleiten, die Antwort behandeln und wieder zurück zur Anwendung schicken. Dieser Vorgang geschieht transparent für den Client, der Code im Client kann also unverändert bleiben. Abbildung 2.1 veranschaulicht diesen Ablauf.

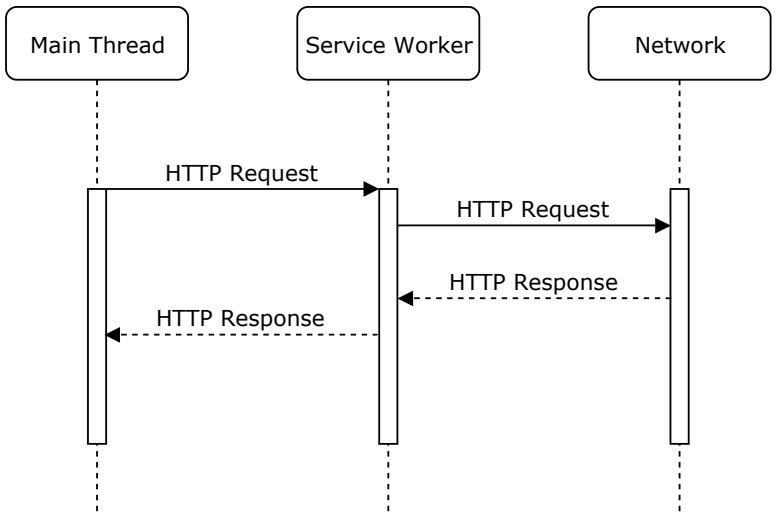

Abbildung 2.1: Service Worker als Proxy bei aktiver Verbindung

Der Service Worker arbeitet event-basiert. Nachdem der Service Worker das Fetch Event *fetch* registriert hat, werden alle HTTP Requests abgefangen. Falls das Durchreichen an das Netzwerk nicht funktioniert hat und dadurch auch keine Antwort vom Netzwerk zurück kommt, kann der Service Worker entsprechend eine eigene Antwort an den Client liefern. Hierdurch entsteht das gewünschte

Verhalten in einer Offline-Umgebung. Dieses Verhalten wird in Abbildung 2.2 dargestellt.

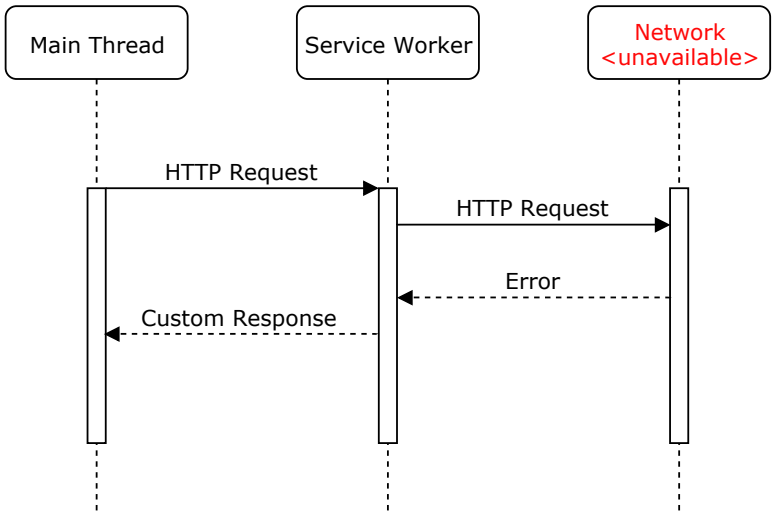

Abbildung 2.2: Service Worker als Proxy bei unterbrochener Verbindung

Dafür kann der Service Worker auf verschiedene APIs zugreifen, unter anderem Fetch API, Cache API, Indexed Database API. (siehe Abschnitt 4.1.1)

2.1.1 Lebenszyklus

Um das Verhalten eines Service Workers zu kennen, muss sein Lebenszyklus verstanden werden.

Nachdem der Service Worker registriert wurde, befindet er sich in dem Zustand *waiting*. Sobald alle Service Worker, die die Seite kontrollieren, nicht mehr aktiv sind, geht er in den Zustand *activated* über. Ein Service Worker kontrolliert eine Seite genau dann, wenn er von dieser registriert wurde und sich im Zustand *activated* befindet. So verhält es sich auch bei einem Update des Service Workers. Bei der Registrierung eines Service Workers wird jedesmal versucht diesen erneut herunterzuladen. Seit der Chrome Version 68 wird der Service Worker standardmäßig nicht mehr gecached, selbst wenn er vom Backend mit einem anderen Header als `Cache-Control: max-age=0` ausgeliefert wird (Jeff Prosnick, 2018). Falls die neu heruntergeladene und die aktuelle Version sich (bitweise) unterscheiden, geht der neue Service Worker in den *waiting*-State solange, bis der alte terminiert. Der Lebenszyklus wird in Abbildung 2.3 als Zustandsdiagramm dargestellt.

Ist der Service Worker aktiv, fängt er nur Requests von Klienten ab, die im Sichtbarkeitsbereich des Service Workers liegen. Der Sichtbarkeitsbereich eines Service Workers beinhaltet alle Adressen, die unter der des Service Workers selbst

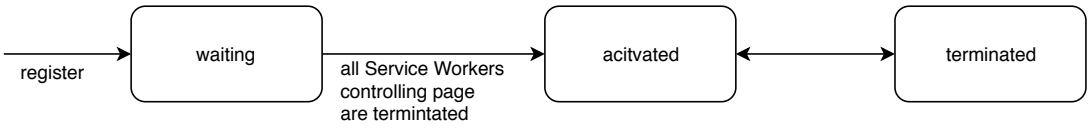

Abbildung 2.3: Lebenszyklus des Service Workers

liegen. Wird der Service Worker gegen den Sichtbarkeitsbereich `'/'` registriert, fängt er alle Requests ab, die von der Ursprungsadresse der Anwendung aus gesendet werden. Weiterhin kann ein Service Worker nur registriert werden, wenn er über *https* ausgeliefert wird. Die Ursprungsadresse `localhost` bildet hier eine Ausnahme. Um die Entwicklung zu vereinfachen ist in diesem Fall kein *https* nötig (Matt Gaunt, 2018).

3 Anforderungen

Das Hauptziel dieser Arbeit besteht darin, die Benutzbarkeit der Webanwendung QDAcity bei Verbindungsverlust zu gewährleisten. Das bedeutet, eine definierte Menge von Funktionalitäten muss auch ohne aktive Internetverbindung erfolgreich vom Benutzer der Anwendung ausgeführt werden können. Ist er wieder online, soll er, nachdem seine Änderungen synchronisiert wurden, wie erwartet weiterarbeiten können. Um diese grobe Anforderung genauer betrachten zu können, wurden folgende, detailliertere Anforderungen erstellt. Mit diesen Anforderungen kann sichergestellt werden, dass die Software ihre Funktion erfüllt und angemessen entwickelt wurde. Tabelle 3.1 erklärt die in den Anforderungen benutzten Terme.

3.1 Funktionale Anforderungen

Die hier gelisteten funktionale Anforderungen folgen der Schablone FunktionsMASTER der SOPHISTen (Chris Rupp, Rainer Joppich, 2013). Abbildung 3.1 beschreibt den Aufbau einer einzelnen Anforderung. Die Knoten, deren Bezeichnung in spitzen Klammern dargestellt ist, sind Platzhalter und werden mit tatsächlichen Ausdrücken ersetzt. Die Schlüsselwörter *muss*, *sollte* und *wird* werden in Tabelle 3.2 definiert.

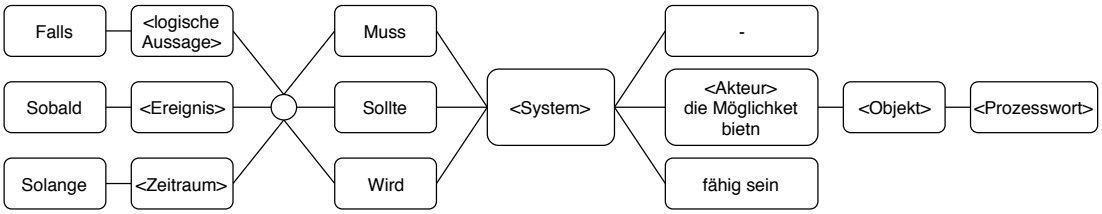

Abbildung 3.1: Schablone FunktionsMASTER

Term	Semantische Bedeutung
Muss	Die Anforderung muss erfüllt werden, damit die Software ihre Funktion erfüllt
Sollte	Die Anforderung ist wichtig, die Erfüllung stellt aber keine notwendige Bedingung für die Funktionalität der Software dar
Wird	Die Anforderung ist nice-to-have

Tabelle 3.2: Definition der Schlüsselworte

3.1.1 Allgemein

REQ-01: Sobald der Benutzer offline ist, muss die Anwendung dem Benutzer dies kenntlich machen. Wenn er wieder online ist, muss das ebenfalls in der Anwendung dem Benutzer ersichtlich sein.

3.1.2 Lokale Aktionen

REQ-02: Sobald der Benutzer eine lokale, nicht unterstützte Aktion ausführt, muss die Anwendung dem Benutzer kenntlich machen, dass die Aktion offline nicht unterstützt wird.

REQ-03: Sobald der Benutzer eine lokale, unterstützte Aktion ausführt, muss die Anwendung mithilfe des Service Workers die entstandenen Operationen ausführen. Die Ausführung der Operationen und die daraus resultierende Antwort muss identisch mit der Antwort des Backends sein.

REQ-04: Sobald der Benutzer eine lokale, unterstützte Aktion ausführt, muss dem Benutzer kenntlich gemacht werden, dass diese Aktion nur lokal ausgeführt wurde.

REQ-05: Sobald der Benutzer mindestens eine lokale, unterstützte Aktion ausgeführt hat, muss dem Benutzer eine Liste aller lokalen Aktionen angezeigt werden.

3.1.3 Synchronisation

REQ-06: Sobald der Benutzer eine lokale, unterstützte Aktion ausführt, muss sich die Anwendung diese für die Synchronisation merken.

REQ-07: Sobald der Benutzer, nachdem er offline war, wieder online ist, muss er erneut authentifiziert werden.

REQ-08: Sobald der Benutzer, nachdem er offline war, wieder online ist, muss die Synchronisation gestartet werden.

REQ-09: Aktionen, die durch den RTCS unterstützt werden, müssen auch während der Synchronisation mit diesem ausgeführt werden.

REQ-10: Sobald eine Operation für die Synchronisierung gespeichert wird, wird die Liste der Operationen vereinfacht. Vereinfacht bedeutet, dass Operationen, deren Änderung durch folgende Operationen überschrieben werden, entfernt werden.

REQ-11: Nachdem die Synchronisation erfolgreich beendet wurde, müssen die aktuellen Ressourcen in die Frontend Anwendungen geladen werden.

3.1.4 Konfliktbehandlung

REQ-12: Sobald bei der Synchronisation ein Konflikt entsteht, muss die Anwendung diesen erkennen.

REQ-13: Sobald bei der Synchronisation ein Konflikt entsteht, muss dem Benutzer die Möglichkeit gegeben werden, den Konflikt beheben zu können, indem entweder a) die aktuelle Synchronisationsoperation mit der eigenen Änderung ausgeführt wird, oder b) der neuen Stand im Backend akzeptiert wird

REQ-14: Sobald ein Konflikt erkannt wird, die Änderungen aber einfach zusammengeführt werden können, sollte dies als dritte Option der Konfliktbehandlung verfügbar sein.

3.1.5 Unterstützte Aktionen

Die folgende Anforderung folgt nicht dem bisherigen Schema FunktionsMaster. Die in den Aktionen verwendeten Begriffe sind im Anhang in der Tabelle 7.1 definiert.

REQ-15: Die in Liste 3.2 gelisteten Aktionen sind Elemente der Menge der *unterstützten Aktionen*.

1. Hinzufügen eines Codings
2. Entfernen eines Codings
3. Bearbeiten eines Codes
4. Hinzufügen eines Codes
5. Entfernen eines Codes
6. Verschieben eines Codes
7. Hinzufügen eines Dokumentes
8. Entfernen eines Dokumentes

Abbildung 3.2: Liste der unterstützten Aktionen, absteigend priorisiert

3.2 Nicht-Funktionale Anforderungen

Die Liste der nicht-funktionalen Anforderungen orientiert sich an der Norm ISO/IEC 25010:2011 (ISO, 2011). Tabelle 3.3 beschreibt eine Teilmenge der Anforderungen, die hier Verwendung finden.

Der Aufbau der Anforderungen folgt der Schablone EigenschaftsMASTER und UmgebungsMASTER der SOPHISTen (Chris Rupp, Rainer Joppich, 2013) und ist in Abbildung 3.4, sowie Abbildung 3.4 dargestellt.

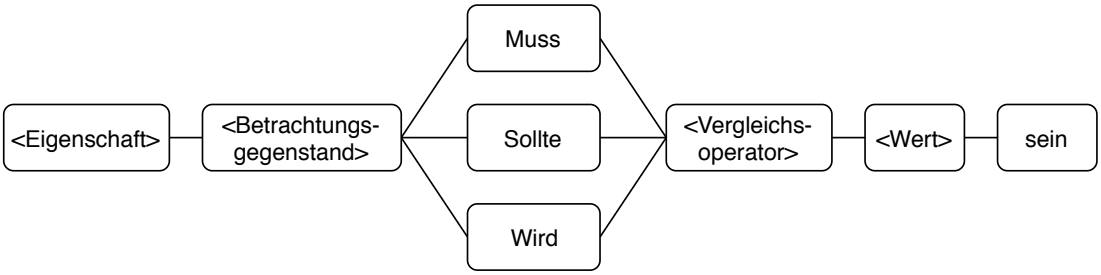

Abbildung 3.3: Schablone EigenschaftsMASTER

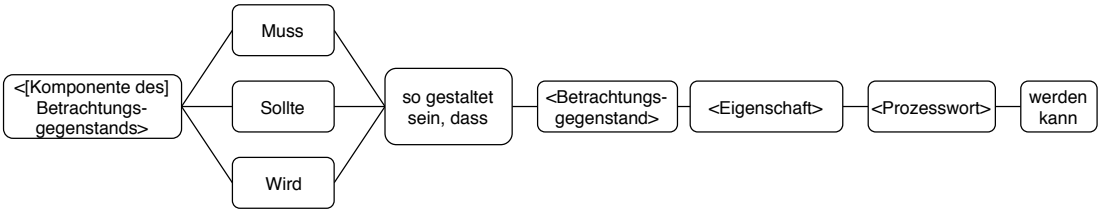

Abbildung 3.4: Schablone UmgebungsMASTER

3.2.1 Technologische Anforderungen

Der im Folgenden verwendete Begriff *Erweiterung* bezieht sich auf die Gesamtheit der Änderungen des Quellcodes, die notwendig sind, um die in Abschnitt 3.1 definierten Anforderungen zu erfüllen. *Software* meint die Anwendung der Änderungen der Erweiterung auf das ursprüngliche Produkt.

REQ-16: Die Erweiterung muss mittels eines Service Workers implementiert sein.

REQ-17: Der Service Worker muss so gestaltet sein, dass er einfach erweitert werden kann (Modifiability).

REQ-18: Der Service Worker muss so gestaltet sein, dass er entkoppelt vom bestehenden Produkt entwickelt werden kann (Modularity).

REQ-19: Die Software soll so gestaltet sein, dass gemeinsam nutzbarer Code von den entsprechenden Komponenten genutzt und nicht mehrfach definiert wird (Reusability).

REQ-20: Der Service Worker muss so gestaltet sein, dass er in das bestehende Build-System integriert werden kann (Installability).

REQ-21: Die Erweiterung muss so gestaltet sein, das sie mit dem bestehenden Test-System getestet werden kann (Testability).

REQ-22: Die Erweiterung wird so gestaltet sein, dass die Daten der Nutzer nicht von unberechtigten Nutzern einsehbar sind (Confidentiality).

REQ-23: Die Erweiterung muss so gestaltet sein, dass der Erzeuger einer Aktion dem tatsächlichen Erzeuger/Nutzer entspricht (Authenticity).

3.2.2 Anwenderbezogene Anforderungen

REQ-24: Der dynamisch beanspruchte Festplattenspeicher der durch die Software erzeugt wird, soll kleiner als 10MB sein (Resource Utilization).

REQ-25: Der Datenverkehr, der durch die Erweiterung zusätzlich erzeugt wird, soll kleiner also 100KB/Stunde sein (Resource Utilization).

REQ-26: Die Unterbrechung der Bedienung der Software durch den Benutzer, soll nicht größer als 2 Sekunden sein (Time behaviour).

Term	Semantische Bedeutung
Benutzer	Ein *Benutzer* ist eine physische Person, die mit der Anwendung interagiert
Aktion	Eine *Aktion* ist eine Interaktion des *Benutzers* mit der Anwendung. Diese kann nur lesend sein, z.B. das Laden der Editor-Seite, oder schreibend, z.B das Hinzufügen eines Codes initiiert durch einen Button-Klick
lokale Aktion	Eine *lokale Aktion* ist eine *Aktion*, die ohne aktive Internetverbindung ausgeführt wurde
Operation	Eine *Operation* ist ein HTTP Request oder eine Socket Message, die für eine *Aktion* notwendig sein können
SW	Der *SW* sei der Service Worker, der im Rahmen dieser Thesis implementiert wird und den Großteil zur Bereitstellung der Offline Benutzbarkeit der Anwendung beiträgt
Backend	Backend ist die Backend Komponente von QDAcity
Ressource	Eine *Ressource* ist ein Entität, die im *Backend* persistiert ist und deren Zustand sich ändern kann, beispielsweise ein Code
unterstützte Aktion	Eine *unterstützte Aktion* beschreibt eine *Aktion*, die auch ohne aktive Internetverbindung das erwartete Verhalten aufweist
nicht unterstützte Aktion	Eine *nicht unterstützte Aktion* beschreibt eine *Aktion*, die ohne aktive Internetverbindung nicht funktioniert
Synchronisation	Eine *Synchronisation* ist der Prozess, bei dem die durch *lokale Aktionen* des *Benutzers* entstandene *Operationen* und Zustandsänderungen von *Ressourcen* so verarbeitet werden müssen, dass diese auf dem Backend widergespiegelt werden können
Konflikt	Ein *Konflikt* entsteht genau dann, wenn während der *Synchronisation* festgestellt wird, dass der Zustand der *Ressource* im Backend, auf die die *Operation* angewendet werden soll, sich vom Zustand der *Ressource* vor der lokalen Ausführung der *Operation* unterscheidet

Tabelle 3.1: Definition der Terme, die in den Anforderungen benutzt werden

Charakteristik	Sub-Charakteristik	Definition
Maintainability	Modifiability	Grad, in dem die Software effizient angepasst oder erweitert werden kann.
	Reusability	Grad, in dem Komponenten der Software in mehreren Komponenten verwendet werden können.
	Modularity	Grad, in dem die Software modularisiert ist, sodass die Veränderung einzelner Module minimale Auswirkung auf die anderen Module hat.
Security	Authenticity	Grad, in dem ein Objekt tatsächlich dem entspricht, was es angibt zu sein.
	Confidentiality	Grad, in dem Daten nur Autorisierten Zugang gewährt wird.
Portability	Installability	Grad, in dem die Software effizient in einer Umgebung installiert werden kann.
Performance	Time Behaviour	Grad, in dem die Antwortzeiten der Software den Anforderungen entspricht.
	Resource Utilization	Grad, in dem die Menge der benötigten Ressourcen der Software denen der Anforderung entspricht.

Tabelle 3.3: Nicht-Funktionale Anforderungen, die in den hier benutzten Anforderungen Verwendung finden. Definition nach ISO/IEC 25010:2011

4 Architektur und Design

4.1 Allgemein

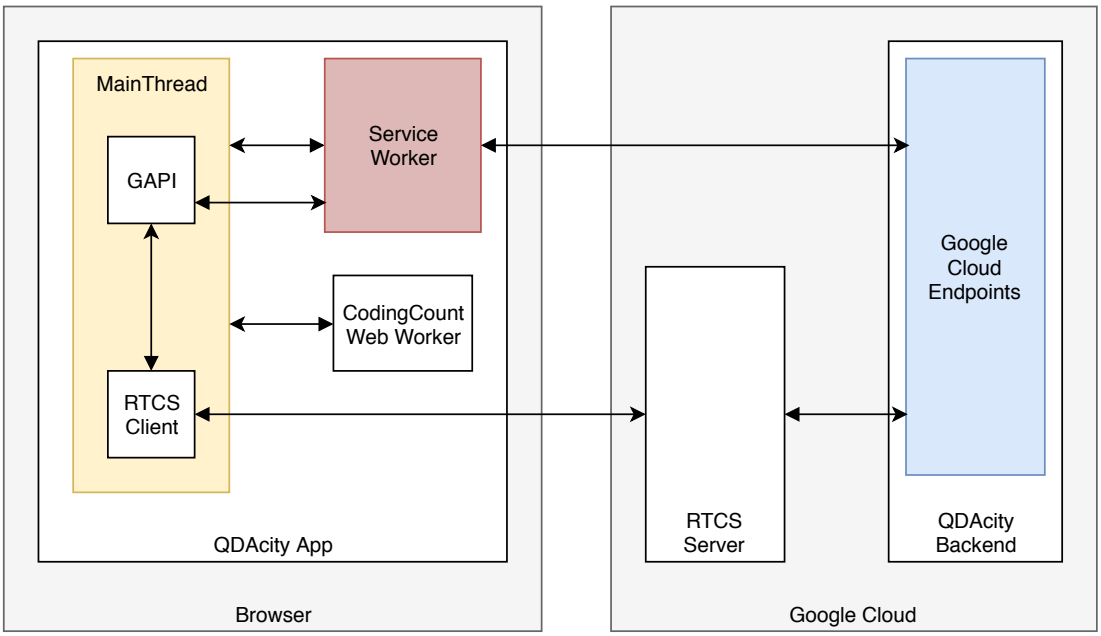

Abbildung 4.1: Kommunikation der beteiligten Komponenten in QDAcity

Abbildung 4.1 zeigt die an QDAcity beteiligten Komponenten und stellt die Kommunikation zwischen diesen dar. Vor der Implementierung des Service Workers kommunizierte die QDAcity Frontend Anwendung mit dem QDAcity Backend über die sich im main thread befindliche Google Client Library(GAPI) bzw. über den RTCS (Realtime Collaboration Service) Client. Da der Service Worker gegen die Ursprungsadresse registriert wird, werden alle Requests, die von der GAPI Library gesendet werden, vom Service Worker abgefangen. Requests, die durch nicht unterstützte Aktionen entstanden sind, werden trotzdem durch den Service Worker geleitet. Dieser ignoriert diese Requests aber, leitet sie also unmodifiziert und unbehandelt and das Netzwerk weiter und gibt die Antwort des Netzwerks

unbearbeitet zurück. Für Aktionen, die mittels des RTCS ausgeführt werden, geschieht die Kommunikation nicht direkt zwischen dem RTCS Client und dem Service Worker, sondern ebenfalls über die durch den GAPI Library entstandenen HTTPRequests. Zum Nachrichtenaustausch, der nicht über HTTPRequests realisiert wird, kann der Service Worker außerdem direkt mit dem main thread kommunizieren.

4.1.1 Caching und Persistenz

Um die geforderten Ressourcen offline bereitstellen zu können, müssen diese lokal persistiert werden. Der HTML5 Standard spezifiziert dafür verschiedene Möglichkeiten:

- **WebSQL** bietet relationale Datenbanken, ist jedoch veraltet bzw. wird nicht mehr aktiv entwickelt (Ian Hickson, 2010).
- **Web Storage** bietet mit *localStorage* und *SessionStorage* einfache Schlüssel-Wert Speicher, der jedoch, da er nicht indiziert ist, nur ineffizient durchsucht werden kann. Er arbeitet außerdem synchron und ist weder in Web Workern noch in Service Workern verfügbar.
- **IndexedDB API** bietet einen indizierten Schlüssel-Wert Speicher. Dieser lässt sich dank Indizierung effizient über die Einträge durchsuchen.
- **Cache Storage**, definiert in der Service Worker Spezifikation, bietet eine Schnittstelle zur Speicherung von Request/Response-Paaren. Dieser Cache ist unabhängig vom Browser-Cache (Alex Russell, Jungkee Song, Jake Archibald, Marijn Kruisselbrink, 2018a).
- **Application Cache** bietet, ähnlich wie der Cache Storage, die Möglichkeit, Ressourcen zu cachen. Dieses Interface ist jedoch veraltet (siehe Abschnitt 2.1).

Daraus ergibt sich, dass für dieses Projekt zwei verschieden Speicher genutzt werden. Zum einen wird der **Cache Storage** verwendet, um statische Ressourcen zu cachen, unter anderem verschiedene JavaScript, CSS oder HTML Dateien. Zum anderen wird daneben die **IndexedDB API** benutzt, um dynamische Ressourcen zu speichern. Sie stellt das lokale Gegenstück zur Datenbank dar, auf die das QDAcity Backend zurückgreift. Die IndexedDB ist in mehrere *Databases* unterteilt, die wiederum einer Ursprungsadresse (z.B qdacity.com oder localhost) zugehörig sind. Die Möglichkeit, mehrere *Databases* zu benutzen, wird hier nun dafür genutzt, die Daten verschiedener Nutzer zu speichern. Das bedeutet, für jeden authentifizierten Benutzer von QDAcity wird eine Database in der IndexedDB angelegt. Diese wird dann anhand der ID des Benutzers diesem zugeordnet. Je Ressourcetyp wird dann ein *Store* in einer dieser Databases angelegt.

Abbildung 4.2 veranschaulicht diesen Aufbau am Beispiel der Domäne qdacity.com und zwei Benutzern mit jeweils zwei Stores.

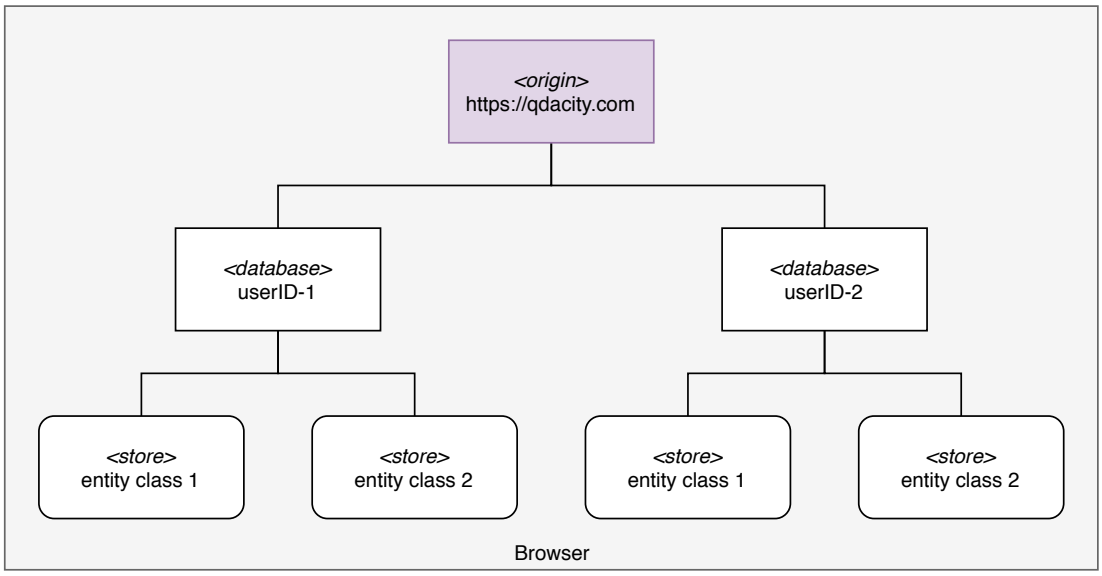

Abbildung 4.2: Aufbau der IndexedDB

Quota

Bei der Verwendung der jeweiligen Storage APIs muss jedoch auch der in Anspruch genommene Speicherplatz berücksichtigt werden. Die Regeln, wie viel Speicher welcher Storage einnehmen darf, ist allerdings browser-abhängig.

Bei Firefox werden zum Zweck der Speicherplatzberechnung die unterschiedlichen Storage APIs zu einem **browser storage** zusammengefasst. Darunter fallen dann auch unter anderem die Cache API und die IndexedDB API. Es wird ferner zwischen **global limit** und **group limit** unterschieden. Das global limit berechnet sich aus 50% des freien Festplattenspeicherplatzes. Das group limit bezeichnet den Speicherplatz, den eine bestimmte Domain zur Verfügung hat. Im Fall von QDAcity wäre das qdacity.com. Dieses berechnet sich aus 20% des global limits, jedoch maximal 2 GB (Mills et al, 2018).

Um die Arbeit mit der IndexedDB API zu erleichtern, wurde das Packet *idb*[1] von Jake Archibald, Mitentwickler der Service Worker Spezifikation, verwendet. Dieses wandelt *IDBRequests* in Promises um.

[1] https://github.com/jakearchibald/idb

4.1.2 Workbox

Wie in der Einführung zum Service Worker beschrieben, besteht dieser aus einem einzelnen Javascript File. Der Service Worker lauscht dabei vor allem auf das *fetch-Event*. Für die grundlegenden Funktionalitäten existiert die Javascript Library Workbox[2]. Sie ist der Nachfolger der Libraries sw-toolbox[3] und sw-precache[4]. Sie vereint und erweitert deren Funktionalitäten basierend auf einer moderneren Codebasis. Workbox wird aktiv, unter anderem von Mitarbeitern aus dem Google Web Developer Relations Team, entwickelt (GoogleChromeLabs, 2017).

Die Workbox Library ist ebenfalls ein einzelnes Javascript File, welches in das Service Worker File importiert werden kann. Sie gliedert sich neben dem Modul `workbox.core` in weitere Module, von denen von Folgenden Gebrauch gemacht wird:

Workbox.Routing

Anstatt im *fetch*-event Listener mit einer großen Verzweigung über die kommende URL den Quellcode zu strukturieren, kann mittels des workbox.routing Moduls einfach die entsprechende URL registriert werden. Der bei der Behandlung der URL auszuführende Code wird durch eine Funktion dargestellt. Damit URL Schablonen der tatsächlichen URL zugeordnet werden können, können die zu unterstützenden URLs als regulärer Ausdruck übergeben werden (Workbox, 2018b). Durch die Verwendung dieses Moduls kann der Quellcode wesentlich übersichtlicher strukturiert werden.

Workbox.Strategies

Das Modul `workbox.strategies` bietet vordefinierte Handler für übliche Caching-Strategien. Diese werden benutzt, falls keine komplexere Logik, wie sei bei den API-Routen (siehe Abschnitt 5.1.1 und 5.1.2) vorliegt, notwendig ist. Dabei stehen folgende Strategien zur Verfügung:

- Stale-While-Revalidate
- Cache First
- Network First

[2]https://developers.google.com/web/tools/workbox/
[3]https://googlechromelabs.github.io/sw-toolbox
[4]https://github.com/GoogleChromeLabs/sw-precache

- Network Only
- Cache Only

Stale-While-Revalidate versucht zuerst mit der Ressource aus dem Cache zu antworten. Wenn sie nicht verfügbar ist, wird auf das Netzwerk zurückgegriffen. In jedem Fall entsteht ein Netzwerkanfrage, damit die Ressource dann gecached werden kann.

Cache First unterscheidet sich zu Stale-While-Revalidate insofern, dass die Ressource nur dann gecached wird, wenn sie noch nicht gecached war. Das bedeutet, falls die Ressource aus dem Cache geliefert wird, wird kein zusätzlicher Request an das Netzwerk gesendet und die Ressource wird im Cache nicht aktualisiert.

Network First versucht zuerst die Ressource aus dem Netzwerk zu holen und cached diese dann. Falls das nicht gelingt, wird auf den Cache zurückgegriffen. (Workbox, 2018c)

Die letzten beiden Strategien sind eher ungebräuchlich und werden deshalb auch nicht weiter erläutert.

Während der Entwicklung wurden alle Requests von statischen Routen (siehe Abschnitt 5.1.1) mit der Strategie *Network First* bedient.

Workbox.Precaching

Das Modul `workbox.precaching` erlaubt es, Ressourcen mit dem Service Worker zu cachen, bevor diese tatsächlich angefragt werden. Der Service Worker lädt die benötigten Ressourcen herunter und speichert diese im Cache. Weiterhin wird eine Revisionsinformation für jede Ressource in der IndexedDB gespeichert. Dieser Vorgang findet im *install*-Event des Service Workers statt. Basierend auf den Revisionsinformationen werden dann nur die Ressourcen aktualisiert, deren Revision sich geändert hat. Zur automatisierten Anpassung der Revision sollte diese aus dem Hashwert der entsprechenden Dateien bestehen. Es existieren Tools, die diese Funktion bereits implementiert haben, unter anderem das Plugin *workbox-webpack-plugin*[5] (Workbox, 2018a)

[5]https://www.npmjs.com/package/workbox-webpack-plugin

5 Implementierung

Der Großteil der Implementierungsarbeit findet sich in dem service-worker Bündel wieder und wird zuerst in diesem Kapitel beschrieben. Danach werden Änderungen und Erweiterungen an teils bestehenden Modulen erläutert. Zuletzt werden die verwendeten Testszenarien vorgestellt.

5.1 Service Worker

Da der Service Worker auf keinen globalen Zustand zurückgreifen soll, sind alle Klassen statisch deklariert. Die notwendigen Daten, die über mehrere Instanzen des Service Workers hinweg, bereitgestellt werden müssen, d.h. wenn eine Transition des Service Workers aus dem Zustand *stopped* in den Zustand *activated* existiert, kommen dann jeweils aus der IndexedDB bzw. aus dem Cache.

5.1.1 sw.js

Die Datei *sw.js* ist der zentrale Einstiegspunkt des Service Workers. Zunächst wird das workbox Skript (siehe Abschnitt 4.1.2) importiert und workbox-spezifische Konfigurationen vorgenommen. Anschließend werden die Routen, die mit dem workbox.Routing Modul (siehe Abschnitt 4.1.2) registriert werden, definiert. Dabei wird unterschieden zwischen Routen, die geprecached werden können und solche, die erst zur "Laufzeit"geroutet werden können. Zur ersteren Kategorie gehören alle statischen Ressourcen, wie css-, js-, image- und html-Dateien. Diese werden, bevor ein tatsächlicher Request auf diese Ressource stattfindet, gecached und erst bei einer Veränderung ihrer Revision (die automatisch z.B. durch einen Hash erzeugt werden kann) versucht, neu zu laden. Zur zweiten Kategorie gehören die API-Routen. Eine API-Route strukturiert sich aus folgenden Eigenschaften:

- `id`
- `controller|handler`

- `sync`

Mittels der `id` und dem discovery document der API, kann die URL sowie die RequestMethod berechnet werden. Sie ist in `common/endpoints/constants.js` definiert und entspricht der ID, die bei der Deklaration der Endpoint-Methoden im Backend definiert wurde. Das discovery document wird bei jedem seiner Zugriffe nach dem Prinzip network first versucht zu laden, sofern es nicht bereits geladen ist. Das bedeutet, sollte sich das discovery document durch Änderungen im Backend ändern, muss auch der Service Worker aktualisiert werden, damit das discovery document neu geladen wird, da es sich ansonsten noch im Speicher befinden könnte. Der `controller` ist eine eigens definierte Controller Klasse(siehe Abschnitt 5.1.2), die dieser Route zugeordnet werden soll. Alternativ kann auch eine vordefinierte Strategie aus workbox.Strategies (siehe Abschnitt 4.1.2) als `handler` zugewiesen werden. Ist das `sync`-flag gesetzt, wird beim Ansprechen dieser Route eine entsprechende Operation zur Synchronisierung gespeichert (siehe Abschnitt 5.1.4)

Weiterhin wird in diesem Script auf das `message`-Event gelauscht, dessen `data`-Attribut an den MessageHandler weitergereicht wird (siehe Abschnitt 5.1.4)

5.1.2 Controller

Die Controller Klassen folgen alle dem gleichen Schema. Jeder Controller, der einem Endpoint zugeordnet wird, implementiert zwingend eine `good(resource, data)` und eine `bad(data)` Methode.

Die `good(resource, data)` Methode wird aufgerufen, wenn der entsprechende Request erfolgreich ausgeführt wurde, d.h. eine aktive Internetverbindung besteht. Deshalb kann dieser Methode auch das `resource`-Objekt übergeben werden, das den Body der HTTPResponse enthält. Das `data`-Objekt enthält alle Parameter des HTTPRequests als Schlüssel-Wert Paare, sowie den Body des HTTPRequests und die ID des Endpoints. Die meisten dieser `good()`-Methoden cachen die Ressource in der IndexedDB (bzw. entfernen diese), passen äquivalent zum Backend-Code ggf. abhängige Ressourcen an und geben die vom Backend übergebene Ressource in einem Promise zurück.

Die `bad(data)`-Methode wird aufgerufen, wenn der entsprechende Request fehlgeschlagen ist, also keine aktive Internetverbindung besteht. Folglich kann dieser Methode auch nicht die Ressource vom Backend übergeben werden. Diese Methode übernimmt dann zusätzlich auch die komplette Erstellung der Ressource (inklusive der Generierung von IDs) und gibt diese ebenfalls in einem Promise zurück. Da es sich hier nun um eine lokal erstellte Ressource handelt, wird dieser absteigend eine negative ID zugewiesen und zur Kennzeichnung mit einem *dirty*-flag versehen. Hierbei ist noch zu beachten, dass die Struktur der zurückgegeben

Ressource konform mit jener vom GAPI Endpoint ist. Beispielsweise wird ein Array noch in ein Objekt, das dieses als `items` definiert, gehüllt.

Auf diese Weise kann für jeden Endpoint, der behandelt werden soll, eine zugehörige `Controller`-Klasse erstellt werden.

5.1.3 DB

Wie in Abschnitt 4.1.1 beschrieben, wird zur Speicherung der Ressourcen die IndexedDB verwendet. Zur Abstrahierung der Zugriffe dient die Klasse `DB`. Sie kapselt alle nötigen Operationen auf der IndexedDB und stellt dafür eigene Methoden bereit. Wenn nicht anderes spezifiziert, erfolgen alle Zugriffe auf die Datenbank, die durch die ID des sich aktuell im Cache befindlichen Benutzers angesprochen wird. Um die einzelnen Stores zu erzeugen, wird bei jedem Öffnen der Datenbank überprüft, ob sich die Datenbankversion (spezifiziert in `db/constants.js`) erhöht hat. Falls dies der Fall ist, wird durch alle Stores (ebenfalls definiert in `db/constants.js`) iteriert und eventuell noch nicht vorhandene Stores werden erzeugt. Um einen weiteren Store hinzuzufügen, muss also lediglich der Store definiert und die `DB_VERSION` inkrementiert werden. Ein Store folgt dabei folgendem Schema:

- `name`: Name des Stores
- `options`: Options-Objekt aus der IndexedDB-Spezifikation
- `createCopy -> true|false`: Optional. Falls `true`, wird neben diesem Store ein weiterer Store erzeugt, der Kopien der Ressourcen enthält, bevor diese durch eine Aktion verändert werden.

Um das tatsächlich verwendete Persistenzverfahren zu entkoppeln, wird der Zugriff durch die `Controller`-Klassen noch einmal über die `Service`-Klassen Schicht gekapselt. Jede `Service`-Klasse erbt dabei von der `CrudService`-Klasse. Diese definiert wiederum alle nötigen Zugriffsmethoden unter Benutzung des Attributes `STORE_NAME`. Für jeden Store existiert eine `Service`-Klasse und im allgemeinen und einfachsten Fall unterscheiden sich diese lediglich durch den `STORE_NAME`. Abbildung 5.1 stellt diesen Zusammenhang in einem Klassendiagramm dar. Das Diagramm beinhaltet nicht alle Kindklassen von `CrudService`.

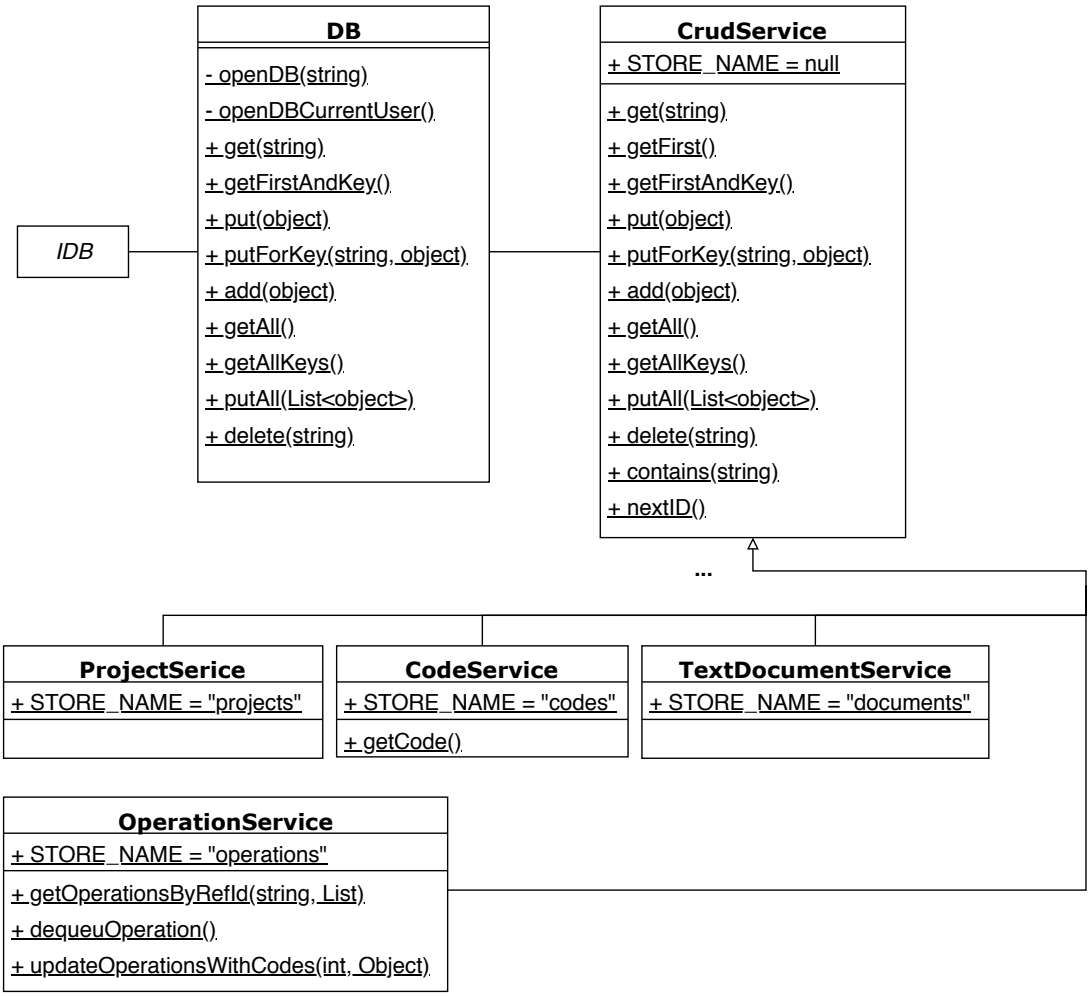

Abbildung 5.1: Klassendiagramm der Datenbank-Komponente

5.1.4 Handler

Dieser Abschnitt beschreibt folgende drei Klassen:
`ResponseHandler`, `SyncHandler` und `MessageHandler`. Diese sind in dem `handler` Paket enthalten.

ResponseHandler

Die `ResponseHandler`-Klasse definiert die drei Methoden `handleGoodAndBad(event, controller, shouldSync)`, `handleGoodResponse(response, event, handler, data)` und `handleBadResponse(handler, data, shouldSync)`.

Dabei bietet `handleGoodAndBad(event, controller, shouldSync)` die Schnittstelle nach außen und wird bei der Definition der *Routes* in `sw.js` als Funktion übergeben.

Die Methode `handleGoodAndBad(event, controller, shouldSync)` ist die oberste Schicht der Requestverarbeitung. Die Methode nimmt einen HTTPRequest entgegen und liefert eine HTTPResponse zurück. Zunächst werden Request Informationen wie Parameter und Body aus dem Request extrahiert (siehe Abschnitt 5.1.5) und in dem Objekt `data` zusammengefasst. Danach wird der eigentliche HTTPRequest (verfügbar über den Parameter `event`) versucht auszuführen. Bei erfolgreicher Ausführung wird die resultierende HTTPResponse zusammen mit der `good()`-Methode des `controller` und den extrahierten Request Daten an die Methode `handleGoodResponse(response, event, handler, data)` übergeben. Bei Fehlschlag der Ausführung des HTTPRequests wird die Methode `handleBadResponse(handler, data, shouldSync)`, populiert mit der `bad()`-Methode des `controller`, aufgerufen.

Die Methode `handleGoodResponse(response, event, handler, data)` ruft lediglich den übergebenen `handler` auf und gibt die durchgereichte, vom Backend erzeugte HTTPResponse zurück.

Die Methode `handleBadResponse(handler, data, shouldSync)` ruft ebenfalls den übergebenen `handler` auf, gibt aber dessen Rückgabewert als HTTPResponse zurück. Falls es sich bei dem Request um eine zu synchronisierende Aktion, spezifiziert durch den Parameter `shouldSync`, handelt, wird außerdem eine *Synchronisierungsoperation* in der IndexedDB gespeichert. Eine Synchronisierungsoperation hält alle Daten, die für eine spätere Synchronisation erforderlich sind. Das sind der Body des Requests, die Endpoint ID des Requests, die Parameter und eine Referenz zu einer Kopie der Ressource in der IndexedDB, bevor sie durch diesen Request verändert wird. Je lokal ausgeführter, zu synchronisierender Aktion, wird also mindestens eine Synchronisierungsoperation erstellt. Es existieren Aktionen, die mehrere solcher Operationen zur Folge haben. Dem main thread werden dann die entstanden Synchronisierungsoperationen durch eine Message mitgeteilt.

SyncHandler

Die Klasse `SyncHandler` bietet als äußere Schnittstelle die `sync()`-Methode. Sie wird durch den Erhalt einer Message vom main thread getriggert.

Der grobe Ablauf der `sync()`-Methode wird durch das Diagramm in Abbildung 5.2 dargestellt. Die Methode holt rekursiv jeweils das erste Element der Liste der gespeicherten Operationen aus der IndexedDB. Sie terminiert, sobald es kein erstes Element mehr gibt, also keine Operationen mehr synchronisiert werden müssen. Zunächst wird überprüft, ob ein Konflikt existiert. Dies übernimmt die Methode `checkForConflict(op, lastAnswer)`. Sie vergleicht den Zustand der Ressource, bevor die Operation `op` sie verändert hat, mit dem aktuellen Zustand der Res-

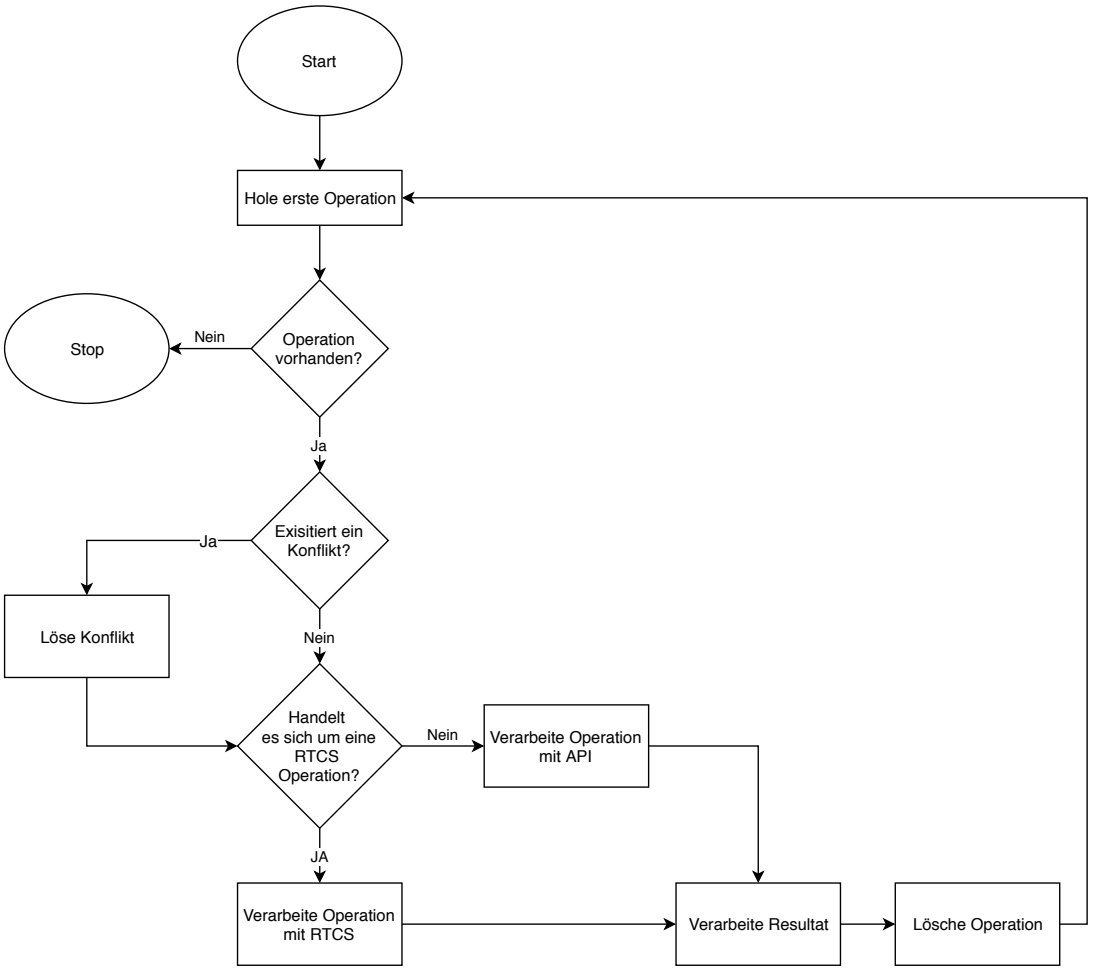

Abbildung 5.2: Ablauf der sync()-Methode

source im Backend. Unterscheiden sich diese beiden Objekte, wurde ein Konflikt erkannt und es wird eine Message an den main thread gesendet, der daraufhin einen Dialog zur Konfliktlösung anzeigt. Der Benutzer kann nun auswählen (siehe Abschnitt 5.2.2), ob er den Zustand durch seine Änderungen, oder den Zustand der Ressource auf dem Server akzeptieren will. Die Entscheidung wird als Nachricht wieder zurück an den Service Worker geschickt. Dieser entscheidet nun, ob die Operation direkt mittels der API, oder mit dem RTCS durchgeführt werden soll. Im ersten Fall kann die Synchronisation weiter im Service Worker statt finden. Handelt es sich aber um eine RTCS Operation, wird wieder eine Nachricht an den main thread gesendet, der die empfangene Operation nun mittels des RTCS Client ausführt. Diese Weiterleitung an den main thread ist notwendig, da WebSocket Verbindungen in Service Workern nicht unterstützt werden. Nach erfolgreicher Ausführung der Operation werden die neuen Informationen der Ressource aus dem Backend verarbeitet. Beispielsweise muss die referenzierte ID einer Ressource aller Operationen angepasst werden, die nun durch das Backend eine neue ID zugeteilt bekommen hat. Zuletzt wird die Operation aus dem Store in der IndexedDB entfernt und es wird mit der nächsten fortgefahren.

MessageHandler

Die Klasse `MessageHandler` ist dafür zuständig, eingehende Nachrichten vom main thread zu verarbeiten, bzw. Nachrichten an diesen zu versenden. Eine Nachricht ist ein Objekt, das aus einem Event-String `evt`, sowie einer Argument-Liste `args` besteht. Die Event-Strings sind in der Klasse `SWCommunicator` in `/common/SWCommunicator.js` definiert.

Das Bearbeiten der eingehenden Nachrichten übernimmt die Methode `handleMessage(message)`. Tabelle 5.1 beschreibt die Nachrichten, die vom Service Worker empfangen werden:

Event	**Argumente**	**Bedeutung**
`TOKEN`	*token*: Der Auth-token	Ein neuer QDAcity Authorisierungstoken wurde erstellt.
`DO_SYNC`	-	Initiiere die Synchronisierung.
`EMIT.SUCCESS`	*operationID*: ID der Operation die emittiert werden sollte. *response*: Die Antwort des RTCS Servers auf den Emit.	Das Emittieren des RTCS war erfolgreich. Dies ist eine Antwort auf eine vorhergehende emit Aufforderung.
`CONFLICT.MINE`	-	Antwort auf Aufforderung zu Lösung eines Konfliktes. Es soll der lokale Zustand synchronisiert werden.

Event	Argumente	Bedeutung
`CONFLICT.THEIRS`	-	Antwort auf Aufforderung zu Lösung eines Konfliktes. Es soll der auf dem Server gespeicherte Zustand beibehalten werden. Keine Synchronisation
`CONFLICT.NONE`	-	Es gibt keinen Konflikt. Wird nur innerhalb des ServiceWorkers verwendet
`CODING.ADD`	*documentId*: ID des veränderten Documents. *operations*: Liste der angewandten slate-Operationen.	Es wurde das Hinzufügen eines Codes ausgelöst. Reichere die erstellte Synchronisierungsoperation zusätzlich mit den notwendigen Informationen an.
`CODING.REMOVE`	*documentId*: ID des veränderten Documents. *pathRange*: Slate PathRange Objekt. *codeId*: ID des zu löschenden Codes.	Es wurde das Entfernen eines Codes ausgelöst. Reichere die erstellte Synchronisierungsoperation zusätzlich mit den notwendigen Informationen an.

Tabelle 5.1: Beschreibung der vom ServiceWorker empfangenen Nachrichten. Die Argumente stehen in der Reihenfolge, in der sie in der Argumentliste vorzufinden sind.

Um Nachrichten an den main thread zu senden, werden die zwei Methoden `send_message_to_all_clients(msg)` bzw. `send_message_to_first_client(msg)` verwendet. Dabei wird ein Nachrichtenkanal erstellt und die Nachricht auf Port 2 des Kanals an den main thread geschickt. Bei der zweiten Methode wartet der Service Worker zusätzlich auf Port 1 auf eine Antwort. Das bedeutet, das zurückgegebene Promise dieser Methode resolved erst, wenn eine Antwort auf Port 1 empfangen wird. Tabelle 5.2 beschreibt die Nachrichten, die vom Service Worker gesendet werden:

Event	Argumente	Bedeutung
`SYNC.STARTED`	-	Der Synchronisierungsprozess wurde gestartet.
`SYNC.FINISHED.SUCCESS`	-	Der Synchronisierungsprozess wurde erfolgreich beendet.

Event	Argumente	Bedeutung
`SYNC.FINISHED.FAIL`	-	Der Synchronisierungsprozess ist fehlgeschlagen.
`SYNC.CONFLICT`	*operation*: ID der Operation, die einen Konflikt ergab. *resource-local*: Die aktuelle lokale Ressource. *resource-backend*: Die aktuelle Ressource auf dem Server. *diff*: Der Diff zwischen der alten, nicht veränderten lokalen Ressource und der Ressource auf dem Server.	Es existiert ein Konflikt. Lasse diesen lösen.
`SYNC.RTCS`	*operationID*: ID der Operation, die ausgeführt werden soll. *operation*: Operation, die ausgeführt werden soll.	Führe diese Operation mit dem RTCS aus.

Tabelle 5.2: Beschreibung der vom ServiceWorker gesendeten Nachrichten. Die Argumente stehen in der Reihenfolge, in der sie in der Argumentliste vorzufinden sind.

5.1.5 Util

Gapi

Die `Gapi`-Klasse stellt Methoden zur Verfügung, um Requests anhand der Endpoint-ID auszuführen. Gegenüber der GoogleApi Client Library für Node.js [1] bietet diese Klasse noch zwei weitere Funktionalitäten. Zum einen kann mit der Methode `getRegex(id)` ein API-Endpoint anhand seiner `id` in einen regulären Ausdruck transformiert werden. Dies ist notwendig, da die `registerRoute()`-Methode des workbox.routing Moduls unter anderem einen regulären Ausdruck erwartet.

```
new RegExp(path.replace(/{\w+}/g, "\\w+") + "(\\?.*)?$");
```

Beispielsweise wird also eine `id`, die in einen vom discovery document definierten `path`→*"relocateCode/{codeId}/{newParentID}"* resultiert, in folgenden re-

[1] https://github.com/googleapis/google-api-nodejs-client

gulären Ausdruck umgewandelt: `/relocateCode\/\w+\/\w+(\?.*)?$/`. Dieser sorgt dann dafür, dass eine tatsächliche URL auf die entsprechende API-Route gematcht wird.

Die zweite Methode `getRequestData(request)` extrahiert aus einem HTTPRequest, mithilfe des geladenen discovery documents, ein Objekt der Form `{id, params, body}`, wobei das `params`-Objekt die Attribute mit den Werten aus dem HTTPRequest beinhaltet, die strukturell im discovery document definiert sind.

5.2 Hauptanwendung

Die im Folgenden beschriebenen Anpassungen wurden alle im Quellcode der Frontend Anwendung, die im main thread läuft, vorgenommen.

Zuerst muss der Service Worker registriert werden. Dies geschieht in der Funktion `initServiceWorker()` in `index.js`. Falls die Registrierung zwar erfolgreich war, der `serviceWorker.controller` aber nicht existiert, wird die Seite neu geladen. Das kann der Fall sein, wenn die Seite durch einen Force Refresh geladen wurde.

5.2.1 SyncService

Nicht alle Aktionen werden über die Google Client Library mittels HTTPRequest realisiert. Mit der Erweiterung der Anwendung durch den RTCS werden bestimmte Aktionen über Socket Messages von dem RTCS Client an den RTCS Server ausgeführt. Da der Service Worker diese Messages, im Gegensatz zu HTTPRequests, nicht abfängt, muss der bisherige Mechanismus angepasst werden.

Zuerst wird der Fall bei aktiver Verbindung behandelt. Der RTCS Client schickt über den WebSocket die Nachricht an den RTCS Server. Dieser schickt eine Antwort mit der resultierenden Ressource. Der Client registriert diese Antwort in einem Event und benachrichtigt alle nötigen Komponenten der Anwendung über die erfolgreiche Durchführung dieser Aktion und der zurückgegeben Ressource. Das geschieht in der Methode `fireEvent(evt, ...args)` in `index.js` im RTCS Client. Hier wird dann zusätzlich eine Nachricht an den Service Worker mit dem Event und den Argumenten geschickt, damit dieser dann die zurückgegeben Ressource beispielsweise cachen kann.

Ohne aktive Internetverbindung kann die Message vom RTCS Client über den WebSocket an der RTCS Server erst gar nicht gesendet werden. Der `emit(eventName , args, ack)`-Methode von Socket.js wird eine Callback Funktion `ack` übergeben. In dieser Callback Funktion wird nun ein Flag gesetzt, dass die Nachricht erfolgreich an den RTCS Server gesandt wurde. Falls also die Message nicht zu-

gestellt werden kann, da keine Verbindung existiert, wird dieses Callback nie ausgeführt. Die `emit()`-Methode wird also in jedem Fall aufgerufen. Direkt nach der Ausführung wird jedoch mittels `Socket.disconnected` geprüft, ob der Socket eine Verbindung hat. Ist das nicht der Fall, tritt die Fallback Behandlung ein. Es kann aber vorkommen, dass dieser direkt nach der Ausführung der `emit()`-Methode Socket noch den Zustand `Socket.connected=true` aufweist, obwohl die Nachricht nicht gesendet werden konnte. Deshalb wird noch drei Sekunden gewartet, ob die Callback Funktion aufgerufen und somit das entsprechende Flag auf `true` gesetzt wurde. Ist es `false`, tritt ebenfalls die Fallback Behandlung ein. Die Fallback Behandlung besteht aus der Klasse `ApiService` im RTCS Client. Diese implementiert, äquivalent zu der existierenden `emit()`-Methode, eine `emit (messageType, arg)`-Methode, welche dem `messageType` einen entsprechenden API Methodenaufruf über die Google Client Library zuordnet. Dieser wird dann wie gehabt vom Service Worker abgefangen. SocketIO speichert die Nachrichten, die mit `emit()` versucht wurden zu versenden, in einem Buffer. Da die Nachricht bereits durch den Fallback behandelt wurde, soll bei Wiederherstellung der Verbindung nicht nochmals versucht werden diese Nachricht zu verschicken. Deshalb wird bei Eintreten der Fallback-Behandlung dieser Buffer geleert.

Einerseits wird so das Problem bei unterbrochener Netzwerkverbindung gelöst. Andererseits kann damit eine durch den RTCS behandelte Aktion schneller ausgeführt werden, falls dieser für seine Antwort länger als drei Sekunden brauchen sollte. Dadurch wird die Benutzerfreundlichkeit der Anwendung erhöht.

5.2.2 App Component

In der React Komponente `App.jsx` werden alle weiteren nötigen Operationen ausgeführt bzw. an Unterkomponenten verteilt. Diese Änderungen betreffen unter anderem die Kommunikation zum Service Worker, die UI, sowie die Kommunikation zum RTCS Client.

UI

Damit dem Benutzer ersichtlich ist, dass er momentan keine Verbindung zum Server besitzt, wird ihm dies in der Navigationsleiste angezeigt (siehe Abbildung 5.3)

Abbildung 5.3: Navigationsleiste bei unterbrochener Verbindung.

Im HTML5 Standard existiert zwar das Event `'offline'`, ist aber je nach Browser unterschiedlich implementiert und bezieht sich meist auf den browser-eigenen ‘Offline‘-Modus. Um wirklich festzustellen, ob eine Verbindung zum Server existiert, wird dieser in einem Intervall von drei Sekunden angepingt. Dies erzeugt eine Request/Response mit einer Größe von ca. 50 Bytes. Schlägt dieser Request nun fehl, existiert keine Verbindung zum Backend.

Eine weitere graphische Komponente, die dem Nutzer bei der Verwendung der Anwendung ohne Verbindung hilfreich sein kann, ist die Operationsliste. Wenn eine unterstützte Aktion lokal ausgeführt wird, wird die entsprechende Operation zur Synchronisation dem Benutzer in einer Liste dargestellt (siehe Abbildung 5.4). Somit weiß der Benutzer, welche Änderungen er vorgenommen hat, die bei Verbindungswiederherstellung synchronisiert werden.

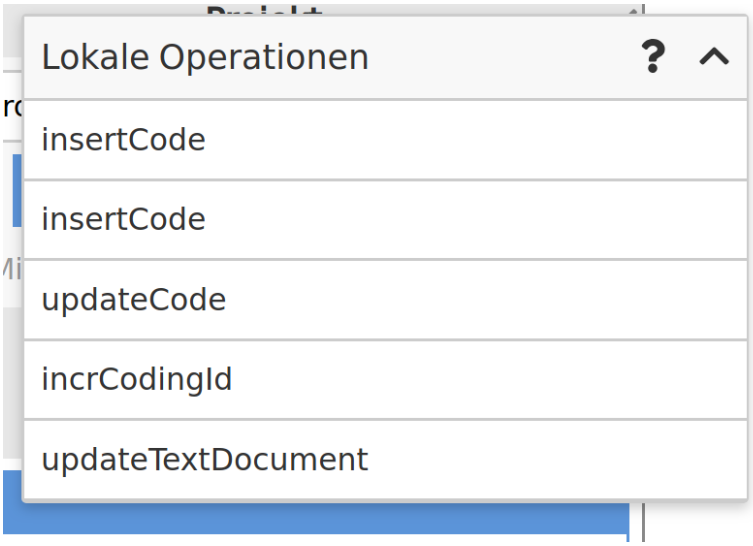

Abbildung 5.4: List der lokalen, nicht synchronisierten Operationen.

Falls bei der Synchronisation im Service Worker ein Konflikt auftritt, schickt dieser dem main thread eine Nachricht mit den jeweiligen Ressourcen und dem Diff, damit der Konflikt für den Benutzer dargestellt werden kann. Dieser kann mithilfe eines Dialogs entscheiden, wie der Konflikt gelöst werden soll.

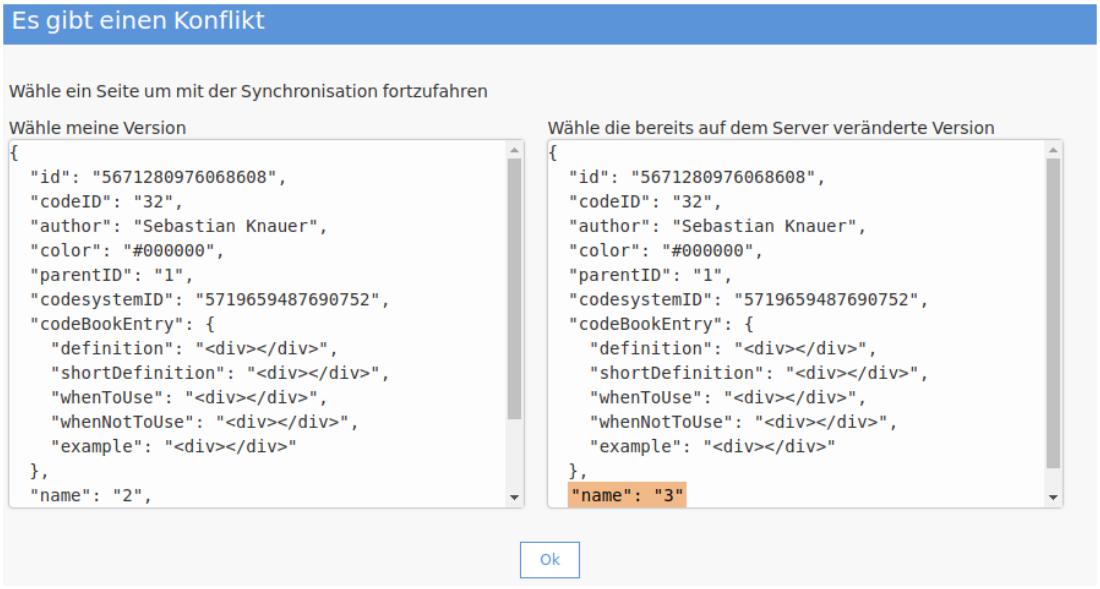

Abbildung 5.5: Dialog zur Lösung des Konflikts bei unterbrochener Verbindung.

Der Benutzer kann hier zwischen zwei Optionen wählen. Entweder kann er seine Änderungen, die er lokal bereits durchgeführt hat, auch auf dem Server übernehmen (links). Oder er kann die Version auf dem Server akzeptieren und sein lokalen Änderungen dadurch überschreiben (rechts). Der Unterschied der Versionen der Ressource auf dem Server und der Version vor der lokalen Änderung wird durch eine Markierung des entsprechenden Attributs dargestellt.

Kommunikation Service Worker und RTCS Client

Weiterhin wurde der bestehende Authentifikationsmechanismus erweitert, sodass bei Erneuerung des Authentifikationstokens dieser an den Service Worker gesendet werden kann.

Soll die zu synchronisierende Operation mittels des RTCS ausgeführt werden, schickt der Service Worker eine Nachricht mit der Operation an den main thread. Dieser übersetzt dann die Operation wieder in einen Aufruf an den RTCS Client.

5.3 Tests

Die Akzeptanztests für QDAcity werden aktuell mit dem Jest Framework[2] und Selenium[3] erstellt und mit dem Chromedriver[4] ausgeführt. Diese werden durch GitLabCI in einem Test-Job automatisch ausgeführt. Dieser Job startet einen Docker-Container, der wiederum die mit Jest geschriebene Test-Suite mittels *npm* ausführt. Dabei geht das Script in folgenden Schritten vor:

1. Starte den Selenium Server
2. Starte den RTCS und Backend-Entwicklungs Server(Dev Server)
3. Führe die Tests aus

Um das Verhalten der Anwendung ohne aktive Internetverbindung zu testen, muss also während der Tests bzw. während des Test-Jobs, ein Verbindungsverlust realisiert werden. Dies wird umgesetzt, in dem einfach die Prozesse des Dev Servers und des RTCS Servers terminiert werden. Die führt nun zu folgendem Ablauf:

1. Starte den Selenium Server
2. Starte den RTCS und Dev Server
3. Führe die Online Tests aus
4. Stoppe den RTCS und Dev Server
5. Führe die Offline Tests aus
6. Starte den RTCS und Dev Server
7. Führe weitere Online Tests aus

Damit der Test-Job den richtigen Exit-Code an GitLabCI zurückgibt, beendet das Script mit einer *OR*-Verknüpfung der Exit-Codes der einzelnen TestSuite Durchläufe. Es wurden folgende Akzeptanztests hinzugefügt:

- Laden eines Projekts ohne Verbindung, das zuvor mit Verbindung geladen wurde
- Einfügen und korrektes Darstellen eines Codes ohne Verbindung

[2]https://jestjs.io/
[3]https://www.seleniumhq.org/
[4]https://sites.google.com/a/chromium.org/chromedriver/

6 Evaluation

Im Folgenden wird geprüft, ob das tatsächliche Ergebnis der Arbeit die Anforderungen, die in Kapitel 3 beschrieben wurden, erfüllen. Damit kann bewertet werden, ob die implementierte Lösung den Anforderungen genügt.

6.1 Funktionale Anforderungen

Die Funktionalen Anforderungen, die in Abschnitt 3.1 gelistet sind, werden im Folgenden ausgewertet. Sie werden dabei, neben den allgemeinen Anforderungen, analog zur Definition gruppiert. Diese Gruppierung entstand aus der Erkenntnis, dass für jede unterstützte Aktion eine wiederkehrende Folge von Schritten sukzessiv abgearbeitet werden muss:

1. Cachen der Ressourcen
2. Lokale Behandlung der Aktion und eventueller Seiteneffekte
3. Synchronisation der Aktion
4. Eventuelle Konfliktlösung

Diese Abfolge von Schritten wird hier Zyklus genannt.

6.1.1 Allgemein

Anforderung REQ-01 wurde erfüllt. Sobald der Benutzer offline bzw. wieder online ist, wird ihm innerhalb von maximal drei Sekunden durch die Anpassung der Navigationsleiste mitgeteilt, in welchem Zustand sich die laufende Anwendung aktuell befindet. Das ist aus Sicht der Nutzererfahrung auch wichtig, denn somit kann er noch vor der lokalen Ausführung einer Aktion entscheiden, ober er diese tatsächlich ausführen will, im Bewusstsein dessen, dass seine Aktionen nicht sofortig mit Kollaboratoren synchronisiert werden. Weiterhin führt, sofern

er offline ist, ein Verweis auf eine Erklärung, was der offline-modus ist, wie er zustande kommt und was das für die Benutzung der Anwendung bedeutet.

6.1.2 Lokale Aktionen

Der erste Schritt des Zyklus beschreibt das lokale Behandeln unterstützter und nicht unterstützter Aktionen. Das wichtigste Feedback für den Benutzer bei der Ausführung einer Aktion ist, ob diese erfolgreich verarbeitet werden konnte. Anforderung REQ-02 ist erfüllt, denn es wurde eine generische Änderung implementiert, sodass dem Benutzer bei Ausführung jeder nicht unterstützten Aktion mitgeteilt wird, dass seine Aktion nicht ausgeführt werden konnte. Erst wenn eine Aktion explizit unterstützt wird, wird diese Meldung bei der Ausführung nicht mehr angezeigt.

Anforderung REQ-03 ist erfüllt, da die Anwendung um einen Service Worker erweitert wurde, der an das Netzwerk fehlgeschlagene Anfragen bearbeitet. Die den fehlgeschlagenen Anfragen resultierenden Ressourcen werden in der graphischen Oberfläche mit einem Asterisk gekennzeichnet dargestellt, sodass der Benutzer weiß, dass diese Ressource nur lokal und (noch) nicht im Backend in diesem Zustand vorliegt. Weiterhin wird die durch diese Anfrage erzeugte Operation ebenfalls in der Oberfläche in einer Liste dargestellt. REQ-04 und REQ-05 sind somit erfüllt.

6.1.3 Synchronisation

Der nächste Schritt adressiert die Synchronisation. Damit eine Synchronisation stattfinden kann, muss eine Folge von zu synchronisierenden Aktionen gespeichert werden. Der Service Worker fängt jede, durch Aktionen entstehende Anfragen ab und persistiert die resultierenden Operationen in der IndexedDB mittels der im Rahmen der Service Worker Komponente implementierten Klasse `OperationService`. Anforderung REQ-06 ist dadurch erfüllt.

Die Synchronisation wird mit einer Nachricht an den Service Worker durch die Klasse `SWCommunicator` gestartet, sobald die Anwendung wieder eine Verbindung mit dem Backend aufbauen kann und der angestoßene Authentifikationsmechanismus erfolgreich beendet wurde. Anforderung REQ-07 und REQ-08 sind erfüllt.

Um Anforderung REQ-09 zu erfüllen, die die Synchronisation um einer RTCS unterstützte Aktion behandelt, wurde der Service Worker, sowie die React Komponente `App.jsx`, angepasst. Der Service Worker schickt eine Nachricht an den main thread, der wiederum den RTCS Client zur Synchronisierung auffordert.

Das Vereinfachen von Operationen, also die Reduzierung sich überschreibender Operationen, wurde nicht implementiert. Beispielsweise könnten bei wiederholter Anwendung von *update*-Operationen auf die selbe Ressource alle Operationen bis auf die letzte ignoriert und damit entfernt werden. Das würde bei auftretenden Konflikten erheblich zur Verbesserung der Nutzererfahrung beitragen, da sonst bei jeder Synchronisation dieser Operationen ein Konflikt gelöst werden müsste. Anforderung REQ-10 wurde als nice-to-have definiert und ist nicht erfüllt.

Die React Komponenten, die für die Darstellung der Ressourcen verantwortlich sind, wurden so angepasst, dass sie nach der erfolgreichen Synchronisation die Ressourcen neu laden, damit diese die aktuelle Version des Backends bereitstellen. Das erfüllt Anforderung REQ-11.

6.1.4 Konfliktbehandlung

Um den Zyklus zu vollenden, müssen Konflikte erkannt und behandelt werden. Die Klasse `SyncHandler` des Service Workers besitzt Methoden, um einen solchen Konflikt bei der Synchronisation zu erkennen. Die nötigen Informationen zur Erkennung eines Konflikts werden bereits in den *Controller*-Klassen gesammelt und in der IndexedDB gespeichert. Anforderung REQ-12 ist also erfüllt.

Wie in Abbildung 5.5 dargestellt, wird bei einem Konflikt dem Benutzer ein Dialog angezeigt, um diesen zu lösen. Dabei stehen ihm zwei Auswahlmöglichkeiten zu Verfügung. Zum einen die eigene Änderung, zum Anderen die Änderung im Backend. Anforderung REQ-13 ist teilweise erfüllt. Die Visualisierung der Unterschiede bzw. des Konflikts ist aktuell nur eine primitive JSON Darstellung der Ressourcen, die Unterschiede sind farbliche hinterlegt. Hier muss die Gestaltung noch angepasst werden, um dem Benutzer bei der Konfliktlösung zu helfen.

Die dritte Möglichkeit, also dem Benutzer eine Zusammenführung der Ressourcen vorzuschlagen, bzw. diese automatisch durchzuführen, wurde nicht implementiert. Diese Anforderung ist jedoch für die korrekte Funktionsfähigkeit der Software nicht notwendig. Der Quellcode wurde so vorbereitet und kommentiert, dass diese Funktion leicht implementiert werden kann. Anforderung REQ-14 ist nicht erfüllt.

6.1.5 Unterstützte Aktionen

Anforderung REQ-15 ist teilweise erfüllt, da die Menge der tatsächlich unterstützten Aktionen nur eine Teilmenge der geforderten Menge der unterstützten Aktion ist. Folgende Aktionen wurden implementiert und sind deshalb Elemente der Menge der unterstützten Aktionen:

- Hinzufügen eines Codings

- Entfernen eines Codings
- Bearbeiten eines Codes
- Hinzufügen eines Codes
- Entfernen eines Codes
- Verschieben eines Codes
- Bearbeiten eines Dokumentes
- Hinzufügen eines Dokumentes
- Entfernen eines Dokumentes

6.2 Nicht Funktionale Anforderungen

Im Folgenden werden die nicht Funktionalen Anforderungen, die in Abschnitt 3.2 gelistet sind, ausgewertet. Zunächst werden die technologischen Anforderungen betrachtet, danach folgen die anwenderbezogenen Anforderungen.

6.2.1 Technologische Anforderungen

Anforderung REQ-16 ist erfüllt, da zur Implementierung ein Service Worker benutzt wurde (siehe Abschnitt 5.1.1)

Wie in Kapitel 5 beschrieben wurde der Service Worker modularisiert entworfen und so entwickelt dass neue Funktionen leicht hinzugefügt werden können. Es wurden Gemeinsamkeiten erkannt und diese entsprechend ausgelagert. Weiterhin wurde die Datenbankschnittstelle abstrahiert, sodass, falls ein anderes Persistenzverfahren verwendet wird, lediglich die Klasse `DB` angepasst werden muss. Schließlich kann die Funktionalität des Service Workers einfach abgeschaltet werden, indem er nicht mehr registriert wird. Anforderungen REQ-17 und REQ-18 sind damit erfüllt.

Da der Service Worker Funktionalität des Backends und des RTCS Servers implementieren muss, befindet sich die Logik dieser Funktionalität an mehreren Stellen des Projekts. Der RTCS Server, wie auch der Service Worker sind in Javascript geschrieben. Deshalb ist hier prinzipiell die Wiederverwendung von Code möglich und muss mit dem Build System konfiguriert werden. Diese Anpassung wurde nicht implementiert, sodass Anforderung REQ-19 nicht erfüllt ist. Das Backend ist in Java implementiert, hier kann also kein gemeinsamer Code zur Wiederverwendung im Backend und im Service Worker geschrieben werden.

Die Erstellung des Service Workers wurde in das Build-System mit entsprechenden Gulp-Tasks und Webpack-Entries integriert. Anforderung REQ-20 ist erfüllt.

Abschnitt 5.3 beschreibt, wie die bestehende Test-Suite angepasst wurde, sodass auch die Akzeptanztests im Falle einer Verbindungsunterbrechung getestet werden können. Dafür wurden mehrere Tests entworfen und diese im Test-job in der Pipeline integriert. Anforderung REQ-21 ist erfüllt. Um die Qualität der Anwendung sicherzustellen, müssen noch mehrere Testszenarien entworfen und implementiert werden. Weiterhin ist es sinnvoll alle Akzeptanztests auch mit dem geckodriver[1] zu testen.

In Abschnitt 4.1.1 wird der Aufbau und die Verwendung des Caches sowie der IndexedDB beschrieben. Sobald also ein Benutzer Zugang zu den Daten auf der Festplatte des Benutzers hat, hat er auch Zugang zu den bereits gespeicherten Daten des Benutzers durch den Service Worker. Eine Möglichkeit, dies zu verhindern, wäre, diese Daten zu verschlüsseln. Aufgrund des erhöhten Aufwands der Implementierung, sowie der verschlechterten Nutzererfahrung, wurde diese Funktion nicht implementiert. Anforderung REQ-22 ist deshalb nur teilweise erfüllt, da zur Nichterfüllung ein Zugang zu den Daten auf der Festplatte erforderlich ist.

In Abschnitt 5.2.2 wird der Nachrichtenaustausch vom main thread zum Service Worker beschrieben. Hier wird unter anderem der Authentifizierungstoken gesendet. Der Service Worker verwendet also den bestehenden Authentifizierungsmechanismus, der bereits Anforderung REQ-23 erfüllt. Anforderung REQ-23 ist also erfüllt.

6.2.2 Anwenderbezogene Anforderungen

Der beanspruchte dynamisch Festplattenspeicher setzt sich aus der Beanspruchung des Cache sowie der IndexedDB zusammen. Die folgenden Werte wurden gemessen mit Chromium 69.0.3497.100 (Official Build) Arch Linux (64-bit). Bei einer Sitzung mit fünf Dokumenten und sieben Codes, sowie fünf Synchronisationsoperationen, wurde ein benötigter Speicher von 336KB festgestellt. Den größten Anteil davon beansprucht die Log Datei mit 294KB, also ca 88%. Die eigentliche Datenbank benötigt hier nur 893B, also ca. 0,2%. Mit welcher Strategie der Browser die Log Datei rotiert, ist nicht bekannt. Die Größe des CacheStorage beläuft sich auf 108KB. Die Anforderung REQ-24 ist unter der Bedingung, dass die Log Datei der IndexedDB nicht über 9MB wächst, erfüllt.

Der zusätzliche verursachte Datenverkehr im Netzwerk besteht aus dem Ping zum Backend (siehe Abschnitt 5.2.2). Der Ping findet alle drei Sekunden statt und benötigt ca. 50B, als ca. 60KB/Stunde. Anforderung REQ-25 ist erfüllt.

[1] https://github.com/mozilla/geckodriver

Gänzlich ohne Beeinträchtigung des Nutzungserlebnisses kommt die Erweiterung nicht aus. Zum einen wird der Benutzer aufgefordert Konflikte zu lösen, wenn diese bei der Synchronisation entstanden sind. Bei einer Synchronisation ohne Konflikte ist die Dauer der Synchronisation abhängig von der Art und Anzahl der zu synchronisierenden Operationen. Die Werte in Tabelle 6.1 wurden auf einer Testumgebung mit einem Intel Core^{TM} i5-6500 Prozessor und 16 GB Arbeitsspeicher ermittelt.

Art der Operation	**Anzahl**	**Dauer in ms**
insertCode	10	850
insertCode	25	2223
insertCode	40	4150
relocateCode	10	950
relocateCode	25	2060

Tabelle 6.1: Gemessene Dauer der Synchronisation mit unterschiedlicher Operationsanzahl

Es ist zu erkennen, dass die Dauer der Synchronisation superlinear mit der Anzahl der Operationen steigt. Je nachdem, wie viele Operationen synchronisiert werden müssen, ist Anforderung REQ-26 erfüllt.

7 Fazit

Die in dieser Arbeit entwickelte Erweiterung für QDAcity behandelt einen kompletten Zyklus in einem Szenario, in dem eine aktive Internetverbindung unterbrochen wird. Nach einem Verbindungsverlust kann die Anwendung weiter benutzt und unterstützte Aktionen können offline verarbeitet werden. Sobald der Benutzer wieder verbunden ist, werden die Änderungen synchronisiert. Etwaige Konflikte, die bei der Synchronisation entstehen, können durch den Benutzer gelöst werden. Der im Zuge der Implementierung mittels eines Service Workers entstandene Quellcode ist gut dokumentiert und durch den strukturierten und gegliederten komponentenweisen Aufbau leicht wartbar. Aufgrund der Priorität, den Quellcode möglichst generisch zu entwickeln, können neue Funktionen mit wenig Aufwand durch Erweiterung des Quellcodes unterstützt und in den Zyklus integriert werden. Die Service Worker Komponente ist in das bestehende Build-System integriert, sodass die Abhängigkeit von projektspezifischen Parametern automatisiert gelöst wird. Komplexere Algorithmen wurden modul-getestet. Weiterhin wurden die Akzeptanztests erweitert, sodass Szenarien, in denen eine Verbindungsunterbrechung auftritt, getestet werden konnten. Diese können ebenfalls leicht ergänzt und in den bestehenden Ablauf eingefügt werden.

Die speziell für QDAcity relevante Funktion, Konflikte beheben zu können, kann noch erweitert werden. Falls ein Konflikt durch eine einfache Zusammenführung der Ressourcen behoben werden kann, könnte dies automatisch durch den Service Worker geschehen. Das verbessert das Nutzererlebnis, da so der Synchronisationsprozess häufiger ohne Nutzerinteraktion im Hintergrund durchlaufen kann.

Die genannten Vorteile sind sehr generisch. Deshalb können viele Webanwendungen davon profitieren, wenn sie mit dieser Funktionalität erweitert werden.

Anhang A Glossar

Begriff	Definition
Document	Ein Document ist eine Menge von Zeichen, die codiert sein können
Code	Ein Code ist eine Markierung, die einer Menge von Zeichen innerhalb des Dokumentes zugewiesen werden kann
Coding	Ein Coding ist ein Paar aus einem Code und einer Menge von Zeichen
Codieren	Codieren bezeichnet den Akt der Zuweisung eines Codes zu einer Menge von Zeichen

Tabelle 7.1: Beschreibung von Termen im Zusammenhang mit QDAcity

Literaturverzeichnis

Alex Russell, Jungkee Song, Jake Archibald, Marijn Kruisselbrink. (2018a). Cache. Zugriff 1. Oktober 2018 unter https://w3c.github.io/ServiceWorker/#cache

Alex Russell, Jungkee Song, Jake Archibald, Marijn Kruisselbrink. (2018b). Service Worker Specifikation. Zugriff 6. September 2018 unter https://w3c.github.io/ServiceWorker

Chris Rupp, Rainer Joppich. (2013). Anforderungsschablonen — der MASTER-Plan für gute Anforderungen, 216–245. Zugriff unter https://www.sophist.de/fileadmin/SOPHIST/Puplikationen/re6/Requirements-Engineering_und_-Management_6-Auflage_-_Kapitel_10_-Leseprobe.pdf

GoogleChromeLabs. (2017). sw-toolbox. Zugriff 1. Oktober 2018 unter https://github.com/GoogleChromeLabs/sw-toolbox

Ian Hickson. (2010). Fresher service workers. Zugriff 1. Oktober 2018 unter https://dev.w3.org/html5/webdatabase

ISO. (2011). Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.

Jeff Prosnick. (2018). Fresher service workers. Zugriff 1. Oktober 2018 unter https://developers.google.com/web/updates/2018/06/fresher-sw

Malavolta et al. (2015). Hybrid mobile apps in the google play store: An exploratory investigation, 56–59.

Matt Gaunt. (2018). Service Workers: an Introduction. Zugriff 1. Oktober 2018 unter https://developers.google.com/web/fundamentals/primers/service-workers/

MDN. (2018). Service Worker API. Zugriff 1. Oktober 2018 unter https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

Mills et al. (2018). Browser storage limits and eviction criteria. Zugriff 1. Oktober 2018 unter https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria

Osmani. (2015). Getting Started with Progressive Web Apps. Zugriff 1. Oktober 2018 unter https://developers.google.com/web/updates/2015/12/getting-started-pwa

Statistisches Bundesamt. (2018). 90 % der Bevölkerung in Deutschland sind online. Zugriff 1. Oktober 2018 unter https://www.destatis.de/DE/PresseService/Presse/Pressemitteilungen/2018/09/PD18_330_634.html;jsessionid=C0CED18E7BDFE69A4915241303397EB4.InternetLive2

Steering Group. (2018). HTML Specifikation. Zugriff 1. Oktober 2018 unter https://html.spec.whatwg.org/multipage/offline.html#offline

Workbox. (2018a). Workbox Precaching. Zugriff 1. Oktober 2018 unter https://developers.google.com/web/tools/workbox/modules/workbox-precaching

Workbox. (2018b). Workbox Routing. Zugriff 1. Oktober 2018 unter https://developers.google.com/web/tools/workbox/modules/workbox-routing

Workbox. (2018c). Workbox Strategies. Zugriff 1. Oktober 2018 unter https://developers.google.com/web/tools/workbox/modules/workbox-strategies