BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Masterarbeit

Medieninformatik
Fachbereich VI – Informatik und Medien

Untersuchung der Konfliktmanagementstrategien verschiedener offlinefähiger Systeme

---

Berlin, den 4. Mai 2018

Autorin:
Jacoba BRANDNER
Matrikelnummer:
833753

Betreuer:
Herr Prof. Dr. Hartmut SCHIRMACHER
Gutachterin:
Frau Prof. Dr. Petra SAUER

Abstract

In dieser Arbeit wird ...

Abstract

This work includes ...

INHALT

1 Einführung **5**

1.1 Motivation 5

1.2 Zielstellung 5

2 Grundlagen **7**

2.1 Offline First 7

2.1.1 Progressive Web Apps 7

2.2 Konflikte 8

2.2.1 Git 9

2.2.2 Consistency Availability Partition tolerance (CAP) Theorem 10

2.3 Sync in verteilten Systemen 10

2.3.1 LWW 10

2.3.2 OT 10

2.3.3 CRDT 10

2.4 Das CouchDB Replikationsmodell 10

2.4.1 schlussendliche Konsistenz 11

2.4.2 Replikation? 11

2.4.3 Konfliktmanagement 11

3 Bestehende offlinefähige Systeme / Konzepte **12**

3.1 Offline-First Frameworks/Bibliotheken 12

3.1.1 git? 12

3.1.2 Realm 12

3.1.3 Redux Offline 13

3.1.4 react-native-offline 15

3.1.5 offline-plugin für webpack 15

3.1.6 hoodie 16

4 Szenarien **18**

4.1 Szenarien bei der Datenübertragung 18

4.2 Szenarien zur Konfliktentstehung 20

5 Anforderungsdefinition **24**
5.1 Anwendungsfälle 24
5.2 Funktionalität 25
5.2.1 Konfliktmanagement 26
5.3 User experience? Bedienoberfläche? 27

Abkürzungen **28**

Glossar **30**

Abbildungsverzeichnis **32**

Literaturverzeichnis **32**

Anhang **35**

1 Einführung

> We live in a disconnected & battery powered world, but our technology and best practices are a leftover from the always connected & steadily powered past. [off]

Heutzutage besitzen mehr als fünf Milliarden Menschen ein Mobiltelefon und drei Milliarden haben Zugang zum Internet [Ban16].

langsame Verbindungen, Unterbrechungen. Auch bei 3G und 4G ist die Latenz schrecklich (Lie-Fi?) -> Offline-First Apps können eine bessere User experience bieten.

1.1 Motivation

Ich möchte eine offlinefähige (mobile?) Anwendung entwickeln und stelle mir folgende Fragen.
Welche Software/ Framework benutze ich dazu?
Auf was muss ich bei der Auswahl achten?
Was erwarte ich von einer offline fähigen App?
(funktioniert und kein Datenverlust) –> Synchronisation und Konfliktmanagement
Beispielhaft soll eine Anwendung betrachtet werden... Eine Kontaktliste / Adressbuch die mehrere Personen benutzen können.

1.2 Zielstellung

Wichtig: Offline nutzbar ohne Datenverlust.
Untersuchung des Verhaltens bei Konflikten (verursacht durch paralleles Arbeiten ohne Internetverbindung).
Wie leicht/schwer ist es zu implementieren? konkret werden. Definition offlinefähig
Dazu 3 Stufen: 1 Person (nativ)
1 Person Client—Server

Viele Personen Client–Server
Code soll nur illustrieren – keine ‘richtige‘ App

2 Grundlagen

Was bedeutet offlinefähig?
Native Applikations (Apps) existieren und funktionieren grundsätzlich solange offline, bis sie versuchen online Daten abzurufen.

2.1 Offline First

Offline-First heißt, die Bestandteile einer Anwendung so zu verwalten, dass nach der ersten Verwendung keine Internetverbindung mehr notwendig ist um deren grundlegenden Funktionen zu nutzen.Quelle
Eine Anwendung, die für den Offline-Gebrauch entwickelt wurde, ist sowohl mit, als auch ohne Internetverbindung vollständig einsatzbereit. Bei einer bestehenden Internetverbindung ist das Laden der Assets aus dem Cache schneller als aus dem Netz. Daten, die zuerst lokal gespeichert werden, gehen auch bei plötzlichen Verbindungsverlust nicht verloren.

2.1.1 Progressive Web Apps

Progressive Web App (PWA) ist eine Bezeichnung für eine mobil nutzbare Webseite, die eine Brücke zwischen der nativen Applikation und einer Webseite schlägt. Der Begriff PWA wurde im Jahr 2015 von Alex Russel und seiner Frau Frances Berriman geprägt. Dieser beschreibt Webseiten, die die positiven Funktionen von nativen Applikationen mitbringen, aber statt über App Stores installiert zu werden, im Webbrowser existieren. Die Webseiteninhalte sind ohne die Installation sofort und jederzeit für die NutzerInnen abrufbar. Schon beim zweiten Besuch der Webseite ist die Ladezeit der Daten verkürzt und sie ist offline, oder auch bei schlechter Internetverbindung nutzbar. Nach mehrmaligem Aufruf kann die PWA über den Browser installiert und zum Startbildschirm hinzugefügt werden. Russel und Berriman legen folgende Einenschaften einer PWA fest:

Responsive
: to fit any form factor

Connectivity independent
: Progressively-enhanced with Service Workers to let them work offline

App-like-interactions
Adopt a Shell + Content application model to create appy navigations & interactions

Fresh
Transparently always up-to-date thanks to the Service Worker update process

Safe
Served via TLS (a Service Worker requirement) to prevent snooping

Discoverable
Are identifiable as “applications” thanks to W3C Manifests and Service Worker registration scope a llowing search engines to find them

Re-engageable
Can access the re-engagement UIs of the OS; e.g. Push Notifications

Installable
to the home screen through browser-provided prompts, allowing users to “keep” apps they find most useful without the hassle of an app store

Linkable
meaning they’re zero-friction, zero-install, and easy to share. The social power of URLs matters.

Näher erläutern? [Rus15].

ServiceWorker

localForage und AsyncStorage

IndexedDB

2.2 Konflikte

Verteilte Systeme: Das ist ein mächtiger Begriff für viele Ideen und Konzepten, aber es läuft in der Regel darauf hinaus: Da sind zwei oder mehr Computer, die durch ein Netzwerk verbunden sind und es wird versucht, dass einige der Daten auf beiden Computern gleich aussehen. ==> Ein System das zuverlässig über ein Netzwerk funktioniert.
Zwei Geräte, ein Server, über Netzwerk verbunden.

Spezielle Eigenschaft von Netzwerken: Verbindung kann jederzeit abbrechen: Acht Irrtümer der verteilten Datenverarbeitung:

1. Das Netzwerk ist zuverlässig
Der Strom kann ausfallen oder Glasfaserkabel können kaputt sein — Das Netzwerk ist nicht zuverlässig.

2. Die Latenz ist gleich null

Glasfaserkabel werden durch Mikrowellen (oder andere Technologien) ersetzt um Millisekunden an Zeit zu sparen. Das würde nicht passieren, wäre die Latenz bei null. Es dauert nun mal eine gewisse Zeit(ms) wenn ein Signal eine (geografisch)weite Strecke zurücklegen muss — Die Latenz ist nicht gleich null.

3. Die Bandbreite ist unendlich

Daten können nicht schneller fließen als die Komponenten die sie verarbeiten (Middleware, Datenbank …) — Die Bandbreite ist nicht unendlich.

4. Das Netzwerk ist sicher

Der HEARTBEAT-BUG[1], der im Jahr 2014 behoben wurde und die Sicherheitslücke im ICE-Wireless Local Area Network (WLAN) im Jahr 2016[2] sind nur zwei Beispiele die zeigen, dass das Netzwerk nicht sicher ist.

5. Die Netzwerkstruktur wird sich nicht ändern

Eine Datenbank kann beispielsweise über mehrere Server verteilt sein, die (teilweise) voneinander abhängig sind. Ein Server mit Abhängigkeiten kann ausfallen, es kann eine Aktualisierung für einen anderen Server geben — die Struktur ändert sich.

5. Die Netzwerkstruktur wird sich nicht ändern

Eine Datenbank kann beispielsweise über mehrere Server verteilt sein, die (teilweise) voneinander abhängig sind. Ein Server mit Abhängigkeiten kann ausfallen, es kann eine Aktualisierung für einen anderen Server geben — die Struktur ändert sich.

6. Es gibt eineN AdministratorIn

Es kann beliebig viele AdministratorInnen geben.

7. Die Datentransportkosten sind gleich null

Netflix bezahlte anfang 2014 diversen InternetanbieterInnen dafür, dass Netflix KundInnen bevorzugten Internetzugang haben.

8. Das Netzwerk ist homogen

Es gibt verschiedene Arten von Netzwer: 3G, 4G, LTE, WiFi. Wird beeinflusst durch Hardware (Smartphone, Tablet, PC, Laptop, Router …) [fal]

2.2.1 GIT

Beschreiben wie Git Konflikte löst.

[1] `http://heartbleed.com/` – Zugriff: 07.04.2018

[2] `https://netzpolitik.org/2016/datenschutz-im-zug-deutsche-bahn-will-sicherheitsluecke-in-neuem-ice-wlan-schliessen/` – Zugriff: 07.04.2018

2.2.2 CAP Theorem

2.3 Sync in verteilten Systemen

Es stellt sich heraus, dass die Implementierung dieser Art von Echtzeit-Zusammenarbeit alles andere als trivial ist. Im Folgenden werden die drei Strategien Operational Transformation (OT), Conflict-free replicated data type (CRDT) und Last-Write-Wins (LWW) vorgestellt plus CouchDBs Peplikationsmodell.

2.3.1 LWW

2.3.2 OT

2.3.3 CRDT

2.4 Das CouchDB Replikationsmodell

Aufgabe der Replikation von CouchDB ist die Synchronisation 2+n Datenbanken. Lösungen: Zuverlässige **Synchronisation** von Datenbanken auf verschiedenen Geräten. **Verteilung** der Daten über ein Cluster von DB-Instanzen die jeweils einen Teil des requests beantworten (Lastverteilung) und **Spiegelung** der Daten über geografisch weit verteilte Standorte.
Durch die inkrementelle (schrittweise) Arbeitsweise kann CouchDB genau dort weitermachen wo es unterbrochen wurde wenn während der Replikation ein Fehler auftritt, beispielsweise durch eine ausfallende Netzwerkverbindung *Es werden auch nur die Daten übertragen, die notwendig sind, um die Datenbanken zu synchronisieren.*
Das Besondere an CouchDB ist, dass es darauf ausgerichtet ist, Fehler/Konflikte vernünftig zu behandeln statt anznehmen es träten keine auf (vgl. [ALS10] S. 7f). Wie oben beschrieben, gibt es in Verteilten Systemen einige Fehler die auftreten können.

Das CouchDB Replikationsmodell erlaubt eine nahtlose, peer-to-peer (direkte) Datensynchronisation zwischen beliebig vielen Geräten. Das CouchDB Replikationsprotokoll ist in CouchDB selbst implementiert, das die Serverkomponente abdeckt. Dann gibt es das PouchDB-Projekt, das dasselbe Protokoll in JavaScript implementiert, das auf Browser- und Node.js-Anwendungen abzielt. das deckt Ihre Kunden und dev-Server ab. Schließlich gibt es Couchbase Mobile und Cloudant Sync, die auf iOS und Android laufen und das CouchDB Synchronisationsprotokoll in Objective-C bzw. Java implementieren.

Vektoruhr [3]
content addressable versions: Idee: Nimm den Objektinhalt (content) und jag ihn durch eine Hashfunktion

2.4.1 SCHLUSSENDLICHE KONSISTENZ

LOKALE KONSISTENZ

VERTEILTE KONSISTENZ

2.4.2 REPLIKATION?

2.4.3 KONFLIKTMANAGEMENT

[3] https://en.wikipedia.org/wiki/Vector_clock

3 Bestehende offlinefähige Systeme / Konzepte

Harte, weiche, mittlere probleme (verschiedene Stufen von offlinefähig)
Native Apps sind offlinefähig
bei nativen Apps ist das Problem bei der Datenverteilung

3.1 Offline-First Frameworks/Bibliotheken

Ich möchte aber auch eigenständig Software entwickeln die man vielleicht nicht nur zum Arbeiten nehmen kann, sondern auch um Quatsch zu machen wie Katzengifs zu teilen.

3.1.1 git?

3.1.2 Realm

Backend für mobile Anwendungen (Java, Swift, C#, JS). Realm Datenbank oder Realm Platform(= DB+ Object Server). Schreiben groß 'Offline First Experience überall hin (webseite, whitepaper...)

- lokale DB, plattformübergreifend
- Object Server fungiert als Middleware-Komponente in der mobilen App-Architektur und managed die Datensynchronisation, eventhandling und Integration in Legacy-Systeme. Kann Daten effizient und simultan synchronisieren und **löst in Echtzeit, automatisch Konflikte**
- **Key-features:** Datensynchronisation in Echtzeit, Skalierbarkeit, Cross-Platform Datenmodell, Eventhandling, regelmäßige Backups, Datenintegrations API, Datensicherheit
- **key mobile use cases:** Reactive app architectures, Offline-first experiences, mobilizing legacy apis, realtime collaboration

[rea17b]

- Realtime Data Sync (sendet automatisch Änderungen in Echtzeit)
- Daten-sync-Protokoll komprimiert die marginalen Änderungen (statt das ganze Objekt) in Binärformat und übergibt sie zwischen Gerät und Server.
- synchronisiert die spezifischen Operationen zusammen mit den Daten
- Diese zusätzlichen Informationen erfassen genau das, was man beabsichtigt hat, **sodass das System Konflikte automatisch auflösen kann**. Dies führt zu einer vorhersagbaren Synchronisation ohne manuellen Eingriff, der die Leistung beeinträchtigt.
- (Objektorientierte) Datenbank auf dem Gerät
- Echtzeit Synchronisation
- Konfliktlösung benutzt OT (und vorgegebene Regeln): Man kann custom Konfliktlösungs-Regeln erstellen
- Unterstützung von Transaktionen? – Konfliktlösung passiert auf **Transaktionsebene**
- Wenn Änderungen aufgrund einer unterbrochenen Netzwerkverbindung offline gehen oder das Gerät leer ist, gehen keine Daten verloren.

[rea17a]

3.1.3 REDUX OFFLINE

"Persistenter Redux store für *reasonaboutable*™ Offline-First Anwendungen".
Ist ein eigenständiger Statuscontainer und kann mit jeder Webanwendung angewandt werden, die sich *deklarativ auf Basis einer einzigen Datenquelle rendern lässt*, wie beispielsweise React[4], Vue[5], oder Angular[6] [reda].
ist eine experimentelle Bibliothek die auf *battle-tested* patterns der Offline-First Architektur aufbaut
REDUX OFFLINE verspricht nicht, die Webanwendung offlinefähig zu machen. Um Assets zwischenzuspeichern, muss zusätzlich noch ein ServiceWorker implementiert sein.
Benutzt redux-persist und redux-optimist.
Bei jeder Änderung wird der Redux store auf dem Datenträger gespeichert, und bei jedem Start automatisch neugeladen. (Standardmäßig IndexedDB, localForage, AsyncStorage)
Eine mit REDUX OFFLINE erstellte Anwendung funktioniert ohne weitere Anpassung offline im Lesemodus. Also wenn die benutzende Person vom (Redux-)Status lesen möchte. Um auch im Schreibmodus offline zu funktionieren, werden alle Netzwerkgebunde-

[4] JavaScript Bibliothek: `https://reactjs.org/`
[5] JavaScript Framework: `https://vuejs.org/`
[6] JavaScript Framework: `https://www.angular.io`

nen Aktionen in einem Store-internem Queue gespeichert. Dann erstellt REDUX OFFLINE einen Unterbaum `offline`, wo unter anderem der internen Status und ein Array namens `outbox` verwaltet wird. Um diese Aktivitäten bei Internetverbindung ausführen zu können, müssen alle notwendigen Daten Plus Metadaten gespeichert werden. Die Metadaten sind für die Information zuständig, was davor oder danach passieren soll. Es gibt drei Metadaten die REDUX OFFLINE interpretieren kann:
`meta.offline.effect` - Die Daten die gesendet werden sollen?
`meta.offline.commit` - Aktion die ausgeführt wird sobald Daten erfolgreich gesendet wurden
`meta.offline.rollback` - Aktion die bei permanent fehlgeschlagener Internetverbindung [redb]

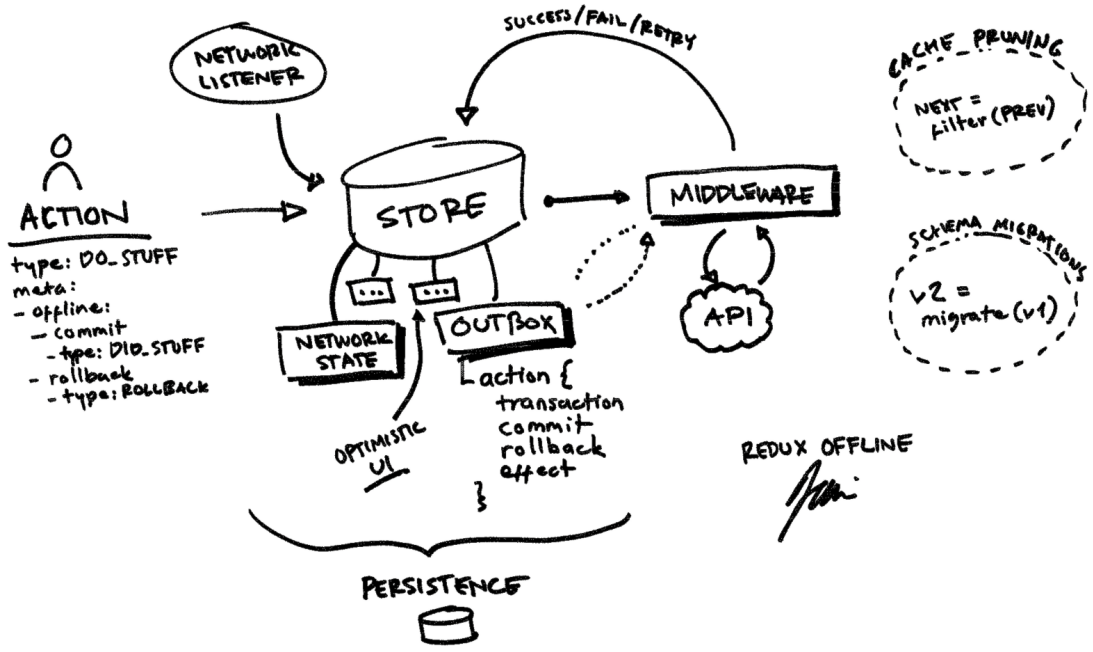

Abbildung 3.1: *Redux Offline Architektur Quelle: [Evä17]*

Die grundlegende Idee hinter Redux Offline ist, dass der REDUX STORE die Datenbank ersetzt/ist. Jede Aktion die benötigt wird um offline zu arbeiten, wird im STORE persisiert und durch die `meta.offline` -Daten weiß die Anwendung was online zu tun ist. Wie die Grafik 3.1 (links) zeigt, wird jede (offline-unterstützende) Aktion mit dem `offline.meta` Feld dekoriert. Darin wird beschrieben, wie der *Netzwerkeffekt* ausgeführt werden soll (`effect`) und welche Aktion ausgelöst werden soll, wenn sie erfolgreich (`commit`) ist oder fehlschlägt (`rollback`). Diese Offline-Aktionen werden im STORE-internen Queue gespeichert und werden, einmal online, an den Server gesendet.

User Interface (UI)
Es umfasst netzwerkfähige Application Programming Interface (API)-Aufrufe, das Persistieren des Zustands(Status), das Stapeln von Nachrichten und die Behandlung von Fehlern, Neuversuchen, optimistische Bedienoberfläche-Aktualisierungen, Migrationen, Cache-Bereinigung und mehr. Weil das kompliziert sein kann, zielt REDUX OFFLINE darauf ab, eine vernünftige Reihe von Standardverhaltensweisen zu bieten, die verwendet werden, und nach Bedarf einzeln überschrieben werden können.
Grundsätzlich ist das Problem, das REDUX OFFLINE löst, kein technisches, sondern ein Architekturproblem. Architekturen können im Code implementiert werden, aber um verstanden, angewendet und nachvollziehbar zu werden, müssen sie gut kommuniziert werden. –> Dokumentation
Konflikte?

REDUX-PERSIST

localStorage. github [redc] medium [San16]

REDUX-OPTIMIST

3.1.4 REACT-NATIVE-OFFLINE

React Native = JavaScript React Framework um native, mobile Apps zu bauen. blabla
Behandelt online/offline Verbindung. Kann die Internetverbindung auch regelmäßig prüfen
Speichert nur den Status online/offline im store. -> erlaubt so unterschiedliches Rendern von Componenten.

```
const YourComponent = ( isConnected ) => (
<Text>isConnected ? 'I am connected to the internet!' : 'Offline :('</Text>
);
```

Zusammen mir Redux hat es einen 'Mehrwert': Hat dann einen 'Offline-Queue um Aktionen zu wiederholen (im Intervall) `meta.retry? Array of actions which, once dispatched, will trigger a dismissal from the queue` oder nicht `meta.dismiss? []`. [Acu]

3.1.5 OFFLINE-PLUGIN FÜR WEBPACK

webpack ist ein JavaScript 'Bundler', packt JavaScript-Dateien und oder Assets für die Verwendung in Browsern.blabla

Bietet offline experience für webpack Projekte. Benutzt SERVICEWORKER und APPCACHE unter der Haube –> cached nur (gebündelten) von webpack generierten Assets. Für die anderen Dateien (z.B. index.html die nicht gebundled wird oder Modul von CDN) braucht mal ein `html-plugin` oder benutzt die `externals` :

```
const OfflinePlugin = require('offline-plugin')
const offline = new OfflinePlugin

const offline = new OfflinePlugin(
externals: ['index.html'],
)
```

Es gibt diverse config-options für SERVICEWORKER und AppCache... [Sto]

3.1.6 HOODIE

Benutzt CouchDB und PouchDB plus UI usw, Frameworkfunktionalität... [hoo]

COUCHDB

Apache CouchDB™ist ein Datenbank Management System (DBMS) das seit 2005 als freie Software entwickelt wird. Die dokumentenorientierte Datenbank (DB) funktioniert sowohl als einzelne Instanz, als auch im Cluster, in dem ein Datenbanksserver auf einer beliebig großen Anzahl an Servern oder Virtuelle Maschines (VMs) ausgeführt werden kann. So kann die Datenschicht beliebig skaliert werden, um die Anforderungen vieler BenutzerInnen zu erfüllen. CouchDB verwendet das HTTP–Protokoll und JavaScript Object Notation (JSON) als Datenformat, weswegen es mit jeder Webfähigen Anwendung kompatibel ist. CouchDB wird über ein REpresentational State Transfer (REST)ful Hypertext Transfer Protocol (HTTP) API angesprochen. Mit den für RESTful Services standardisierten Methoden z. B. GET, POST, PUT, DELETE können die Daten abgerufen und. manipuliert werden.
Das implementierte Replikationsmodell erlaubt die Synchronisation bzw. bidirektionale Replikation zu verschiedenen Geräten ist genau die Besonderheit, die CouchDB als eine Offline–Datenbank auszeichnet. Dessen Funktionsweise wird in Abschnitt 2.4 detailliert beschrieben. Dieses Protokoll ist die Grundlage für Offline First Anwendungen. Das Replikations-API von CouchDB bietet die Möglichkeit, eine Datenbank kontinuierlich oder selbstgesteuert mit einer anderen zu synchronisieren. So kann beispielsweise eine CouchDB-Instanz auf dem Mobiltelefon und eine auf dem Laptop bestehen und beide können sich bei bestehender Internetverbindung synchronisieren. Da so die gespeicherten Daten aus dem lokalen Speicher gelesen werden, sind ein schnelles Interface und

eine geringe Latenz die positive Folge. Wenn Konflikte auftreten, beispielsweise durch gleichzeitiges Bearbeiten eines Dokuments von zwei Personen ohne Netzwerkverbindung, werden diese als solche markiert, jedoch nicht von selbst aufgelöst. So gehen keine Daten verloren und es liegt an der benutzenden Person diese zu lösen. Referenz git? CouchDB ist für Server konzipiert. Für Browser gibt es PouchDB und für native iOS- und Android–Apps wurde Couchbase Lite entwickelt. Alle können Daten miteinander replizieren und verwenden das CouchDB Replikationsprotokoll [cou]. CouchDB Replikationsmodell hier?

PouchDB

Was benutzt Poch wann? – IndexedDB, Web SQL, LevelDB Als Ergänzung zu CouchDB kann PouchDB verwendet werden. PouchDB ist eine Open–Source–JavaScript–Datenbank, die so konzipiert wurde, dass sie im Browser läuft. PouchDB ermöglicht es Anwendungen zu erstellen, die sowohl offline als auch online funktionieren. Daten können lokal gespeichert werden, sodass alle Funktionen der Anwendung auch im Offline–Modus zur Verfügung stehen. Daten werden unabhängig von der nächsten Anmeldung (des nächsten Onlinezugangs) zwischen **Clients**, CouchDB oder kompatiblen Servern synchronisiert. PouchDB läuft auch in Node.js[7] und kann als direkte Schnittstelle zu CouchDB–kompatiblen Servern verwendet werden [pou].

[7] JavaScript Laufzeitumgebung, steht unter `https://nodejs.org/en/download/` zum Download bereit

4 Szenarien

Alle in Kapitel 3 angeführten Technologien haben die Unterstützung der Erstellung von offlinefähigen Anwendungen gemeinsam. Prinzipiell sollte eine Offline First Anwendung in der Lage sein, mit fehlender Internetverbindung zu funktionieren und mit auftretenden Konflikten so umgehen zu können, dass keine Daten verloren gehen. Sie muss die Fälle behandeln können, die sich aus den folgenden Szenarien ergeben. Dafür werden zunächst Szenarien in der Netzwerkübetragung als 'Voraussetzung für die der Konfliktentstehung' aufgezeigt.

4.1 Szenarien bei der Datenübertragung

Im einfachen Anwendungsbeispiel einer Kontaktliste gibt es zwei Parteien die miteinander interagieren: die Liste als Client und den Server. Folgende Situationen können bei der Übertragung von Daten über das Netzwerk eintreten.

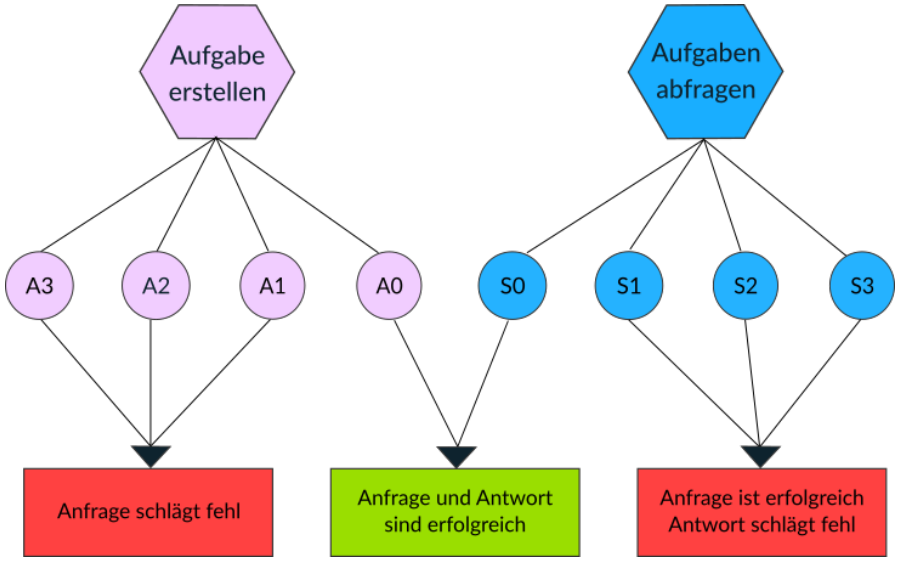

Abbildung 4.1: *Szenarien bei der Datenübertragung über das Netzwerk*

Szenario A0:

Der Client erstellt einen Adressbucheintrag, hat den Status ONLINE und der Server ist

erreichbar. Sowohl Anfrage als auch Antwort ist erfolgreich. Der Kontakt wird erfolgreich erstellt.

Szenario A1:
Der Client erstellt einen Adressbucheintrag, hat den Status OFFLINE und der Server ist nicht erreichbar. Die Anfrage schlägt fehl.

Szenario A2:
Der Client erstellt einen Adressbucheintrag und hat den Status ONLINE. Die Anfrage wird gestartet und währenddessen bricht die Internetverbindung ab. Die Anfrage 'wartet' bis ein Timeout getriggert wird und schlägt dann fehl. Wärend des Wartens ist der Client blockiert.

Szenario A3:
Der Client erstellt einen Adressbucheintrag und hat den Status ONLINE. Die Anfrage wird gestartet und währenddessen bricht die Internetverbindung ab. Die Anfrage ist teilweise erfolgreich. Nur ein Teil der Telefonnummer kommen beim Server an.

Szenario S0:
Der Client fordert eine Liste aller gespeicherten Kontakte vom Server an, hat den Status ONLINE und der Server ist erreichbar. Sowohl Anfrage als auch Antwort ist erfolgreich. Die Liste wird komplett ausgeliefert.

Szenario S1:
Der Client fordert eine Liste aller gespeicherten Kontakte vom Server an, hat den Status OFFLINE und der Server ist nicht erreichbar. Die Antwort schlägt fehl.

Szenario S2:
Der Client fordert eine Liste aller gespeicherten Kontakte vom Server an und hat den Status ONLINE. Während der Server antwortet bricht die Internetverbindung ab. Die Antwort 'wartet' bis ein Timeout getriggert wird schlägt dann fehl. Wärend des Wartens ist der Client blockiert.

Szenario S3:
Der Client fordert eine Liste aller gespeicherten Kontakte vom Server an und hat den Status ONLINE. Während der Server antwortet bricht die Internetverbindung ab. Die Antwort ist teilweise erfolgreich. Nur ein Teil der angefragten Daten kommen beim Client an.

In den obigen Szenarien wird nicht beschrieben warum die Internetverbindung abbricht. Dies kann verschiedene Gründe haben. Um nur einige Beispiele zu nennen: Eine langsame Internetverbindung, oder eine Fahrt durch einen Tunnel kann ein Timeout während einer Aktion hervorrufen. Ein auf einer Baustelle gekapptes Kabel oder ein Stromausfall kann zu zeitweise vollständigen Internetverlust (haha) führen.

Ergebnis

Da die Szenarien *A0* und *S0*, die Szenarien *A1*, *A2* und *A3* sowie die Szenazien *S1*, *S2* und *S3* zusammengefasst werden können, ergeben sich aus den acht Szenarien die drei nun aufgezählten Fälle.

- Fall a: Anfrage und Antwort sind erfolgreich.
- Fall b: Anfrage ist nicht erfolgreich
- Fall c: Anfrage ist erfolgreich, Antwort schlägt fehl

Von den erarbeiteten Fällen sind Fall *b* und *c* für die Szenarien zur Konfliktentstehung relevant: Die Anfrage oder die Antwort schlägt fehl.

4.2 Szenarien zur Konfliktentstehung

Im Anwendungsbeispiel einer Kontaktliste können mehrere Personen die Liste verwalten. Die Komplexität wird durch mehr Parteien – beliebig viele Clients – erhöht. Jede Person kann alle Einträge jederzeit laden und einzelne erstellen, bearbeiten oder löschen. Bei den Ausführungen der grundlegenden Create Read Update Delete (CRUD) Operarionen kann es bei der Synchronisation der beteiligten Parteien zu Konflikten kommen wenn einer der oben genannten Fälle (*b*,*c*) eintritt und ein Objekt von mehreren Parteien bearbeitet wird. Für die effiziente Nutzung der Anwendung sollen bei jedem Start der Anwendung nur die neuen und die gegebenenfalls aktualisierten Einträge geladen werden. Alle Adressbucheinträge müssen identifiziert und versioniert werden. Folgende Situationen können eintreten:

Szenario ID0:

Zur Identifizierung eines Adressbucheintrags wird eine Universally Unique Identifier (UUID) verwendet. Es wird sowohl auf dem Client als auch auf dem Server ein Kontakt mit dem Namen ‘Amilia Pond‘ erstellt. Währenddessen tritt Fall *b,c* ein und beide Parteien können nicht miteinander kommunizieren. Nach der Synchronisation existieren zwei Kontakteinträge mit gleichem Namen, aber unterschiedlicher ID. Sie sind voneinander zu unterscheiden und können einzeln behandelt werden.

Szenario ID1:

Zur Identifizierung eines Adressbucheintrags wird ein sprechender Schlüssel[8] verwendet. Es wird sowohl auf dem Client als auch auf dem Server ein Kontakt mit dem Namen ‘Amilia Pond‘ und dem sprechenden Schlüssel ‘amiliapond‘ erstellt. Währenddessen tritt Fall *b,c* ein. Es ist nicht zu ermitteln, ob derselbe Kontakt doppelt angelegt wurde, wenn beide Kontakteinträge sich unterscheiden, welcher der beiden korrekt ist oder ob es sich bei den Einträgen um zwei Personen mit demselben Namen handelt.

Szenario V0:

Zur Versionierung eines Adressbucheintrags werden Versionsnummern verwendet. Der Kontakt ‘Amilia‘ hat die Version ‘1.0.0‘. Sowohl auf dem Client, als auch auf dem Server wird der Kontakt aktualisiert und geben ihm beide die Versonsnummer ‘2.0.0‘. Währenddessen tritt Fall *b,c* ein und beide Parteien können nicht miteinander Kommunizieren. Bei der Synchronisation entsteht ein Konflikt weil es zwei (unterschiedliche) Einträge mit derselben Verion gibt.

Szenario V1:

Zur Versionierung eines Adressbucheintrags wird ein Zeitstempel verwendet. Der Kontakt ‘Amilia‘ hat die Version ‘2018-04-03 10:00:00Z‘. Amilia ist umgezogen und ihre Adresse ändert sich. Der Eintrag wird bearbeitet und hat nun die Version ‘2018-04-13 11:44:22Z‘. Während der Editierung tritt Fall *b,c* ein. Es stellt sich heraus, dass die Hausnummer einen Zahlendreher hat und es wird sofort berichtigt. ‘Amilia‘ hat nun die Version ‘2018-04-13 11:45:33‘. Nach der Synchronisation gibt es nun zwei Objekte mit derselben ID: ‘Amilia‘ aber unterschiedlichen und zeitlich zuordenbaren Versionen. Durch den Zeitstempel ist sichergestellt, welcher Eintrag der neueste und wahrscheinlich korrekte ist.

Szenario V2:

Zur Versionierung eines Adressbucheintrags wird ein Zeitstempel verwendet. Das Szenario ist dasselbe wie V1 mit dem Unterschied, dass der Server eine spätere Uhrzeit als der Client hat. So hat nach der Synchronisation der später korrigierte Eintrag einen früheren Zeitstempel. Es wird die falsche, alte Adresse gespeichert, die korrekte hat einen älteren Zeitstempel und wird verworfen.

[8] Ein Schlüssel, der sich aus einem Attribut des Objekts ergibt oder sich aus mehreren Attributen zusammensetzt. So könnte ein sprechender Schlüssel von Jean-Luc Picard mit der E-Mail-Adresse picard@enterpise.com beispielsweise ‘picard@enterprise.com‘ (E-Mail) oder ‘Jean-LucPicard‘ (Zusammensetzung aus Vor- und Nachnamen) sein.

Szenario V3:

Zur Versionierung eines Adressbucheintrags wird eine Logische Uhr[9] verwendet. Der Kontakt ‘Amilia‘ hat die Version Beispiel Logische Uhr?. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst (Version:). Währenddessen tritt Fall *b*, *c* ein. Amilia sieht ihre falsche Telefonnummer und berichtigt diese ebenfalls. weil die Versionen identisch sind? Bei der Synchronisation kommt es zum Konflikt. wirklich? auch wenn das Ergebnis dasselbe ist?

Szenario V4:

Zur Versionierung eines Adressbucheintrags wird eine inhaltsbasierte Version verwendet. Um eine Zuordnung zwischen Inhalt und Version machen zu können kommen Hashfunktionen zum Einsatz. Hierbei wird als Version der Hashwert des Adressbucheintrags gespeichert.

Der Kontakt ‘Amilia‘ hat die Version ‘5560348cec1b08c3d53e1508b4a46868‘. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst. Amilia sieht ihre falsche Telefonnummer und berichtigt diese zur gleichen Zeit. Da die Telefonnummer von beiden Parteien korrigiert wird und somit der Inhalt identisch ist, wird für beide Aktionen beide Version ‘88da3f8d82ab58551d2a48d74d9a4986‘ generiert. Es kommt zu keinem Konflikt, da die Versionen und der Inhalt identisch sind.

Szenario V5:

Zur Versionierung eines Adressbucheintrags wird eine inhaltsbasierte Version verwendet. Dem Kontakt ‘Amilia‘ ist die Version ‘5560348cec1b08c3d53e1508b4a46868‘ zugeordnet. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst, während dieser OFFLINE ist. Im selben Status berichtigt der Client die Telefonnummer. Bei der Synchronisation kommt es zum Konflikt, da es nun zwei Einträge mit unterschiedlichem Inhalt, aber identischer Version gibt und nicht festzustellen ist welche Version die neuere ist.

Szenario V6:

Zur Versionierung eines Adressbucheintrags wird eine geordnete Liste von inhaltsbasierten Versionen verwendet. Dem Kontakt ‘Amilia‘ ist eine Liste von Verionen mit einem Eintrag ‘5560348cec1b08c3d53e1508b4a46868‘ zugeordnet. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst, während dieser OFFLINE ist.

[9] Eine Logische Uhr ist eine Komponente die dazu dient, dem Datenobjekt einen eindeutigen Zeitstempel zuzuweisen. Die bekanntesten Verfahren für Logische Uhren in verteilten Systemen sind die Lamport-Uhr und die Vektoruhr. Beide verwenden Zähler die sich bei jedem Ereignis erhöhen. Einfach gesagt besteht die Lamport-Uhr aus einem Zeitstempel und einem Zähler, die Vektoruhr aus einem Zeitstempel und einem Vektor – einer Liste aus Zählern.

Im selben Status berichtigt der Client die Telefonnummer. Jede Aktion fügt der Versionsliste einen neuen Hashwert hinzu. Auch wenn der Content des Adresbucheintrags in den zwei letzten Versionen identisch ist, kann festgestellt werden welcher der neueste Eintrag ist. Kommt es zum Konflikt, werden die beiden *riskanten* Verionen verschachtelt in der Liste gespeichert. In diesem Fall sieht die Liste nun so aus: '[[88da3f8d82ab58551d2a48d74d9a4986, 88da3f8d82ab58551d2a48d74d9a4986], 5560348cec1b08c3d53e1508b4a46868]' – eine Liste der beiden konfliktbehafteten Versionen am Anfang der Liste.

Ergebnis

Die Szenarien *ID0* und *ID1* beschreiben die Identifizierung einzelner Kontakte. Eine eindeutige Identifizierung des Kontakts ist im Szenario *ID0* gewährleistet.
Die Szenarien *V1*, *V2* und *V4*, *V5* beschreiben Situationen mit demselben Ausgangspunkt. In einem Fall kommt zu keinem Konflikt, in dem nächsten schon. Deswegen können *V1* und *V2*, sowie *V4* und *V5* zusammengefasst werden. Es wird deutlich, dass es in jedem Fall zu einem Konflikt kommen kann. Es gilt zu unterscheiden in welchen Fällen mit Konflikten umgegangen werden kann und in welchen Daten verloren gehen.
Grafik?

Im weiteren Verlauf dieser Arbeit werden aus diesen Fällen die Anforderungen an eine offlinefähige Anwendung erarbeitet.

5 Anforderungsdefinition

Dieses Kapitel beschreibt die Anforderungen an eine Offline First Anwendung unter Berücksichtigung von Konfliktmanagement und Funktionalität. UI auch mit rein? Aus den oben genannten Szenarien werden im Folgenden die Anwendungsfälle und Anforderungen hergeleitet, die eine offlinefähige Anwendung erfüllen soll.

5.1 Anwendungsfälle

Aus den in Kapitel 4 erarbeiteten Szenarien ergeben sich die folgenden Use-Cases, die von einer offlinefähigen Anwendung erfüllt werden sollen. Die folgende Tabelle zeigt die Anwendungsfälle aus der Entwicklungsperspektive.

ID	Anwendungsfall	Beschreibung
UC1	Um die Anwendung auch ohne Internetzugang zu nutzen, sollen die Daten auch offline erreichbar sein.	Die Daten werden auf dem Server und lokal gespeichert. Lokal bedeutet in einer lokalen Datenbank oder im Browser (localStorage, IndexedDB usw.).
UC2	Die Anwendung soll die Kontaktliste schnell und effizient laden.	Es werden nur Einträge oder deren Aktualisierungen geladen, die sich noch nicht auf dem Endgerät befinden.
UC3	Ich möchte Einträge immer und überall editieren können.	Jeder Eintrag muss identifizierbar und versionierbar sein.
UC4	Um jedem Adressbucheintrag Operationen zuzuweisen und einzelne Kontakte zu finden, möchte ich die Einträge identifizieren.	Jeder Eintrag bekommt zur eindeutigen Identifikation eine UUID zugewiesen.
UC5	Um zu wissen ob, wie oft und wann ein Eintrag bearbeitet wurde, möchte ich die Einträge versionieren.	Jeder Eintrag bekommt ein Versionsattribut.

UC6	Ich möchte dass alle Änderungen ankommen und keine Daten verloren gehen.	Wenn ein Konflikt auftritt, wird er effizient gespeichert (statt eine Version zu verwerfen).
UC7	Ich weiß, Konflikte können immer auftreten, deswegen möchte ich mit ihnen umgehen können.	Konflikte werden effizient gespeichert, sodass sie nach und nach von NutzerInnen aufgelöst werden können.

Tabelle 5.1: *Anwendungsfälle*

Das in Abbildung 5.1 gezeigte Use-Case-Diagramm veranschaulicht die in der obigen Tabelle 5.1 aufgeführten Anwendungsfälle.

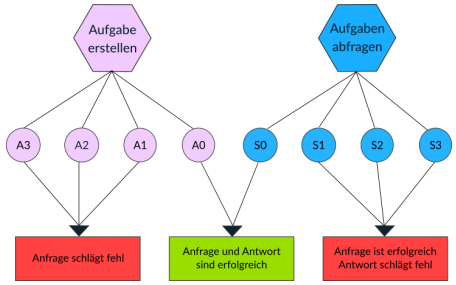

Abbildung 5.1: *Platzhalter für UC-Diagramm*

5.2 Funktionalität

Es soll ein System entwickelt werden, welches an dem Beispiel eines kollaborativen Adressbuchs die Offlinekompatibilität mit dem Schwerpunkt auf das Konfliktmanagement der verwendeten Technologien illustriert. Bezug zu Szenarien und Anwendungsfälle

Ein offlinefähiges, kollaboratives Adressbuch zeigt eine Liste von Kontakten, welche jederzeit – unabhängig von der Internetverbindung – von den verwendenden Personen gelesen, bearbeitet, erstellt und gelöscht werden können. Geschieht eine dieser Operationen offline, werden die Daten bei wieder bestehender Internetverbindung synchronisiert. Im einfachen Fall erfolgt die Synchronisation zwischen der Server und Client. Da die Beispielanwendung kollaborativ ist, erfolgt die sie zwischen allen Beteiligten. Synchronisation erfordert in jedem Fall den Umgang mit Konflikten.

Bein ersten Start der Anwendung müssen, wenn vorhanden, alle Kontakte geladen werden. Sobald sie einmal geladen sind, sollen sie auch offline verfügbar sein. Damit ein Datensatz, wie zum Beispiel ein Adressbucheintrag, offline erreichbar ist, sollte er wenigstens so lange auf dem Client gespeichert werden, bis er vollständig beim Server angekommen sind. Im aktuellen Anwendungsfall bedeutet das, es gibt zwei Kopien des Adressbucheintrags. Eine auf dem Anwendungsgerät, eine auf dem Server.
Danach sollen nur die Einträge geladen werden, die nicht auf dem Gerät existieren. Die Daten würden sonst doppelt geladen werden, der Server hätte mehr zu arbeiten was wiederum die Antwortzeit verlängern würde. Der Server muss also in der Lage sein die Einträge zu sortieren und nur bestimmte Einträge zu versenden und die Anwendung muss wissen, welche Daten sie bereits hat. Dazu muss jeder Kontakt mittels einer ID identifiziert werden.
Wenn es zwei Kontakteinträge mit derselben ID gibt, muss feststellbar sein, welcher Eintrag der aktuellere ist. Gibt es mehr als zwei Einträge müssen diese sortiert werden, sodass ersichtlich wird welcher der aktuellste oder älteste ist, welcher Eintrag vor oder nach welchem kommt. Dazu muss jeder Kontakt versioniert werden.

5.2.1 Konfliktmanagement

Die in Kapitel 4 erarbeiteten Szenarien zeigen, Konflikte können immer auftreten. Werden Konflikte falsch oder gar nicht behandelt, kann es zu Datenverlust führen. Aus diesem Grund müssen sie als Teil der Anwendung betrachtet statt ignoriert zu werden. Im einfachen Konfliktfall kann das System entscheiden welches die konfliktfreie Version ist. So kann zum Beispiel der Kontakt ‘Amilia Pond‘ von einer Person eine neue Telefonnummer, von einer anderen eine neue Adresse bekommen. Die Aktualisierungen finden in unterschiedlichen Bereichen statt und stellen kein Problem dar. in Szenarien aufnehmen?

Die oben erarbeiteten Konfliktszenarien beschreiben Konflikte die nicht vom System gelöst werden können. Diese sollen effizient gespeichert werden. Wichtig hierbei ist die Möglichkeit immer zu einem konfliktfreien Status zu gelangen – unabhängig davon wie viele Konflikte es gibt.
Jeder Fehlerfall muss kommuniziert werden. Wenn es konliktbehaftete Daten gibt muss dies mitgeteilt, und angeboten werden die Konflikte zu lösen. Nur so kann sichergestellt werden, dass keine Daten verloren gehen. siehe Anforderungen PWA?

5.3 User experience? Bedienoberfläche?

Einfache Liste mit Einträgen, editierbar, bei Konflikt: Dialog: ‘welche Version möchtest du behalten¿
UI soll sagen wenn es einen Konflikt gab / gibt und mich entscheiden lassen. Bzw ihn lösen lassen. Auf keinen Fall selber lösen und mich nichts davon wissen lassen. Im besten Fall soll die UI mir sagen **warum** es zum Konflikt gekommen ist.

Abkürzungen

API	Application Programming Interface
App	Applikation
CAP	Consistency Availability Partition tolerance
CRDT	Conflict-free replicated data type
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DB	Datenbank
DBMS	Datenbank Management System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LWW	Last-Write-Wins
OT	Operational Transformation
PWA	Progressive Web App
REST	REpresentational State Transfer
UI	User Interface
UUID	Universally Unique Identifier
VM	Virtuelle Maschine

WLAN Wireless Local Area Network

Glossar

Assets

alle Bestandteile einer Webanwendung die für die für die erfolgreiche Ansicht im Browser benötigt werden. Es sind Hypertext Markup Language (HTML)– Cascading Style Sheets (CSS)– und JavaScriptdateien zu nennen, aber auch Mediendateien wie Bilder.

Bandbreite

gibt an, wie viele Daten pro festgelegter Zeitspanne über ein Netzwerk übertragen werden können.

Hashfunktion

TODO: ist eine Abbildung, die eine große Eingabemenge (die Schlüssel) auf eine kleinere Zielmenge (die Hashwerte) abbildet

Latenz

Die Wartezeit, die im Netzwerk verbraucht wird bevor eine Kommunikation beginnen kann, wird als Latenz oder als Netzwerklatenz bezeichnet.

Middleware

Schicht zwischen Anwendung und Betriebssystem.

optimistische Bedienoberfläche

auch: optimistic UI, wartet nicht mit der Aktualisierung der Oberfläche auf das Ende einer Operation. Die zeigt also den gewünschten Zustand der App an, befor die Anwendung fertig ist indem sie z.B. Fakedaten zeigt.

Progressive Web App

Oder “fortschrittliche Web App“ eine mobil nutzbare Webseite, erstellt mit den Webstandards, die als Symbiose aus einer nativen, mobilen Anwendung und einer responsiven Webseite beschrieben werden kann. Die Idee dahinter ist, dass Apps zukünftig nicht mehr über einen App Store, sondern über den Browser installiert werden kann.

Queue

auch Warteschlange, eine Datenstruktur die zur Zwischenspeicherung von Objekten dient. Hierbei wird das zuerst eingegebene Objekt auch zuerst verarbeitet (wie

bei einer Warteschlange).

ABBILDUNGSVERZEICHNIS

3.1 Redux Offline 14

4.1 Szenarien bei der Datenübertragung über das Netzwerk 18

5.1 Use-Case Diagramm 25

LITERATURVERZEICHNIS

[Acu] ACUÑA, Raúl G.: *react-native-offline*. https://github.com/rauliyohmc/react-native-offline, . – Zugriff: 12.04.2018 3.1.4

[ALS10] ANDERSON, J. C. ; LENHARDT, Jan ; SLATER, Noah: *CouchDB: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, 2010. – ISBN 978–0–596–15589–6. – oreilly.com 2.4

[Ban16] BANK, World: World Development Report 2016: Digital Dividends / International Bank for Reconstruction and Development / The World Bank. Washington DC, 2016. – Research Report. – doi: doi:10.1596/978-1-4648-0671-1 1

[cou] *CouchDB – relax*. https://couchdb.apache.org/, . – Zugriff: 12.04.2018 3.1.6

[Evä17] EVÄKALLADO, Jani: Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native. In: *HACKERnoon* (2017), 3. – Zugriff: 12.04.2018 3.1

[fal] *Eight Fallacies of Distributed Computing*. https://blog.fogcreek.com/eight-fallacies-of-distributed-computing-tech-talk/, . – Zugriff: 12.04.2018 2.2

[hoo] *Hoodie – The Offline First Backend*. http://hood.ie, . – Zugriff: 12.04.2018 3.1.6

[off] *Offline First*. http://offlinefirst.org/, . – Zugriff: 12.04.2018 1

[pou] *pouchdb – Tha Database that Syncs!* https://pouchdb.com/learn, . – Zugriff: 12.04.2018 3.1.6

[rea17a] *The Offline-First Approach to Mobile App Development - Beyond Caching to a Full Data Sync Platform*. https://www2.realm.io/whitepaper/offline-first-approach-registration, october 2017. – Zugriff: 12.04.2018 3.1.2

[rea17b] *BUILD BETTER APPS, FASTER WITH REALM - An Overview of the Realm Platform*. https://www2.realm.io/whitepaper/realm-overview-registration, october 2017. – Zugriff: 12.04.2018 3.1.2

[reda] *Redux Offline Release BREAKING: Migrate to Store Enhancer API*. `https://github.com/redux-offline/redux-offline/releases/tag/v2.0.0`, . – Zugriff: 12.04.2018 3.1.3

[redb] *Redux Offline*. `https://github.com/redux-offline/redux-offline`, . – Zugriff: 12.04.2018 3.1.3

[redc] *react-native-offline*. `https://github.com/rt2zz/redux-persist`, . – Zugriff: 12.04.2018 3.1.3

[Rus15] RUSSELL, Alex: Progressive Web Apps: Escaping Tabs Without Losing Our Soul. In: *Medium* (2015), 08. – Zugriff: 15.04.2018 2.1.1

[San16] SANFORD, Clark: Persistence is Key: Using Redux-Persist to Store Your State in LocalStorage. In: *Medium* (2016), 12. – Zugriff: 12.04.2018 3.1.3

[Sto] STOLYAR, Arthur: *offline-plugin*. `https://github.com/NekR/offline-plugin`, . – Zugriff: 12.04.2018 3.1.5

Anhang

Eidesstattliche Erklärung

CD-Inhalt

Auf der beigefügten CD befinden sich

- Die schriftliche Ausarbeitung dieser Masterrarbeit im PDF-Format
- Das erstellte Projekt