BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Masterarbeit

Medieninformatik
Fachbereich VI – Informatik und Medien

Untersuchung der Konfliktmanagementstrategien verschiedener offlinefähiger Systeme

---

Berlin, den 1. August 2018

Autorin:
Jacoba BRANDNER
Matrikelnummer:
833753

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

Inhalt

1 WIP Einführung **5**

1.1 WIP Motivation 5

1.2 WIP Zielstellung 5

2 Grundlagen **7**

2.1 Offline First 7

2.1.1 Speichern der Assets 8

2.1.2 Speichern der generierten Daten 10

2.1.3 Datenbanksynchronisation 12

2.2 Konfliktmanagementstrategien 12

2.2.1 Konflikte 12

2.2.2 Last-Write-Wins 14

2.2.3 Operational Transformation 14

2.2.4 Conflict-Free Replicated Data Type 17

2.2.5 WIP Logische Uhr? 19

2.2.6 WIP? Differential Synchronization? 19

2.2.7 Das CouchDB Replikationsmodell 19

3 Bestehende offlinefähige Technologien **22**

3.1 Offline plugin für webpack 22

3.2 Redux Offline 22

3.2.1 Redux 23

3.2.2 Redux Offline 23

3.2.3 Redux Persist 25

3.3 React Native Offline 25

3.4 hoodie 26

3.4.1 CouchDB 27

3.4.2 PouchDB 28

3.5 Realm 28

3.6 Übersicht 30

4 Szenarien **31**
4.1 Szenarien bei der Datenübertragung 31
4.2 Szenarien zur Konfliktentstehung 33

5 Anforderungsdefinition **37**
5.1 Umfang 37
5.2 Funktionale Anforderungen 39
5.3 User-Stories 40
5.3.1 NutzerInnen Perspektive 40
5.3.2 EntwicklerInnen Perspektive 40
5.3.3 TesterInnen Perspektive 41
5.4 Bedienoberfläche 42

6 Konzeption **44**
6.1 Anwendungsaufbau 44
6.1.1 Aufbau der React Komponenten 45
6.1.2 Verwendung von Redux Offline 47
6.2 Architektur 49
6.2.1 Das Speichern der Daten 52
6.2.2 Verbindungsstatus feststellen und ändern 56
6.3 Die graphische Oberfläche 57
6.4 Testdurchläufe 59

7 Implementierung der Prototypen **61**
7.1 Die Contacts Komponente 61
7.2 Offlinefunktionalität 63
7.2.1 Datenspeicherung 63
7.2.2 Datenbanksynchronisation 64
7.3 Konfliktmanagement 66
7.4 Installationsanleitung 68
7.4.1 amilia-qouch 69
7.4.2 amilia-rdx 69
7.5 Testfälle 70

8 WIP Evaluation **71**
8.1 Manuelle Tests 71
8.1.1 Erste Testreihe 71
8.1.2 Zweite Testreihe 73
8.1.3 Dritte Trestreihe 74
8.1.4 Übersicht der Testergebnisse 76

8.2 Auswertung 77
8.2.1 Auswertung der manuellen Tests 77
8.2.2 WIP Erfüllung der Anforderungen 78
8.2.3 Implementierungsaufwand 78

9 WIP Zusammenfassung und Ausblick **80**

Abkürzungen **81**

Glossar **82**

Abbildungsverzeichnis **83**

Literaturverzeichnis **84**

Anhang **91**

1 WIP 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 WIP 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 WIP 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

Eine offlinefähige Anwendung ist sowohl mit, als auch ohne Internetverbindung vollständig einsatzbereit. Dieses Kapitel beschreibt die grundlegenden Optionen eine Anwendung offlinefähig zu machen, geht im Speziellen auf Konflikte und deren Lösungsstrategien ein.

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. Native Applikationen (Apps) existieren und funktionieren grundsätzlich solange offline, bis sie versuchen online Daten abzurufen. Um Webanwendungen offlinefähig zu machen, erfordert es einige Grundvoraussetzungen.
Eine Webanwendung besteht aus zwei Bestandteilen. Den Daten die von den BenutzerInnen generiert werden und den Assets. Assets sind alle Bestandteile der App die für die erfolgreiche Ansicht im Browser benötigt werden. Das sind Hypertext Markup Language (HTML)– und Cascading Style Sheets (CSS)–Dateien sowie Skripte und Mediendateien wie zum Beispiel Bilder.
Diese Daten müssen zuerst auf dem Endgerät gespeichert werden, um offline erreichbar zu sein. Bei einer bestehenden Internetverbindung ist das Laden der Ressourcen aus dem Cache schneller als aus dem Netz. Daten, die zuerst auf dem Endgerät gespeichert werden, gehen auch bei plötzlichen Verbindungsverlust nicht verloren.
Gibt es eine serverseitige Datenbank, müssen die Daten zwischen Server und Client synchronisiert werden. Ist die Anwendung kollaborativ, muss die Synchronisation zwischen allen Beteiligten stattfinden. Synchronisation erfordert den Umgang mit Konflikten, denn es dürfen auf keinen Fall Daten verloren gehen.

Dieser Abschnitt zeigt die verschiedenen Möglichkeiten der lokalen Dateinspeicherung und deren Synchronisation mit einer serverseitigen Datenbank auf.

2.1.1 Speichern der Assets

Das Cachen der Assets ist der erste Schritt um Daten offline verfügbar zu machen. Browser haben die Möglichkeit, diese Dateien in ihrem Cache zu speichern. Dieser ist nicht persistent, denn sobald der Speicherplatz voll ist werden enthaltene Daten gelöscht [Net].

AppCache

Um mehr Kontrolle darüber zu bekommen, was wann und für wie lange gespeichert werden soll, wurde der Application Cache (AppCache) zur HTML-Spezifikation hinzugefügt. Im Juni 2016[1] wurde der AppCache wieder aus den Web-Standards entfernt, und wird nicht mehr empfohlen. In der Theorie stellte sich der Application Cache als einfach anzuwenden und unproblematisch dar. Um eine webbasierte Anwendung offline auszuliefern wurde benötigte es eine Textdatei – der `cache manifest`-Datei – mit der Endung `.appcache`. Dort wurden alle Ressourcen aufgelistet, welche der Browser cachen sollte. Die Datei wurde dann über das `manifest`-Attribut in die HTML-Dateien der Webanwendung eingebunden werden.

```
<!DOCTYPE html>
<html manifest="example.appcache">
  <head>
    <title>Example Application Cache</title>
    <link rel="stylesheet" href="style.css">
    <script src="index.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

Listing 2.1: *Beispiel einer HTML-Datei mit einer Manifest-Attribut Einbindung*

Die über das `manifest`-Attribut eingebundene Cache-Datei kann folgendermaßen aussehen:

```
CACHE MANIFEST
# version comment for triggering updates
# v1
style.css
index.js
assets/cat.png
```

Listing 2.2: *Beispiel einer* `.appcache`*-Datei*

[1] siehe `https://github.com/w3c/html/pull/444/commits`

Alle Seiten mit dem manifest-Attribut und die, die explizit in der Textdatei beschrieben wurden, wurden vom Browser gespeichert [Net18].
In der Praxis jedoch zeigten sich zahlreiche Probleme mit dem AppCache. So wurde der Application Cache nur aktualisiert wenn sich der Inhalt der des Manifests geändert hat. Dann mussten alle Dateien neu heruntergeladen werden. Wurden das Manifest und eine andere Datei geändert, wurden die geänderten Dateien nicht unbedingt erneut gespeichert. Denn wenn der Server zusammen mit den Dateien keine Cache-Header sendete, so speicherte der Browser die Datei nach einem Cache-Header-Wert den er 'errät'. So konnte es passieren, dass der Browser annahm, eine Datei brauche keine Aktualisierung und weiterhin die alte, gecachte Version auslieferte [Arc12].
Als Reaktion auf diese Probleme wurde der Service Worker entworfen.

Service Worker

Ein Service Worker ist ein Skript, das zwischen Netzwerk und Browser sitzt und von Letzterem im Hintergrund ausgeführt wird. Die Kernfunktion des Service Workers ist es, Netzwerkanfragen abzufangen um sie zu verarbeiten und im Cache zu verwalten [Gau18].
Gegenwärtig besitzen – bis auf den Internet Explorer – sämtliche Desktop-Browser, und alle gängigen mobilen Browser eine Unterstützung für Service Worker.

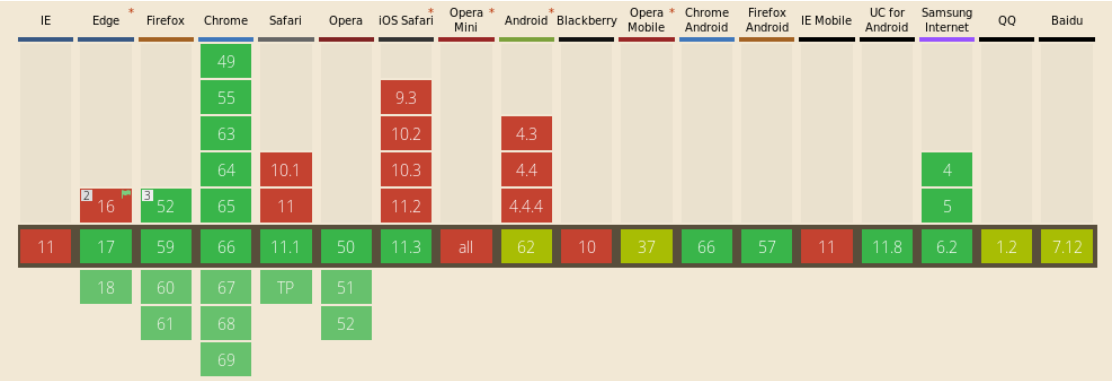


Abbildung 2.1: *Browserkompatibilität für Service Worker, Quelle: [canb]*

Mit dem Service Worker können wie mit dem App Cache statische Ressourcen sofort beim ersten Besuch der Seite im Cache gespeichert werden. Es lässt sich hierbei unterscheiden, ob die Daten vor der ersten Verwendung, oder später im Cache gespeichert werden sollen. Für den ersten Fall eignen sich statische Inhalte wie Schriften oder JavaScript–Dateien, für den zweiten größere Ressourcen die nicht sofort benötigt werden.
Zusätzlich bietet der Service Worker die Möglichkeit auf Interaktionen zu reagieren. Den NutzerInnen kann angeboten werden bestimmte Inhalte der Seite, wie zum Beispiel ein Video, später bzw. offline anzuschauen. Diese werden dann im Cache gespeichert und

sind somit offline verfügbar. Service Worker erlauben außerdem den Zugriff auf Push-Benachrichtigungen und das Background Sync Application Programming Interface (API). Die Hintergrundsynchronisation kann einmalig oder in festgelegten Intervallen stattfinden und ist besonders für nicht dringende Aktualisierungen wertvoll [Arc14].

2.1.2 Speichern der generierten Daten

Um NutzerInnenspezifische Daten offline verfügbar zu machen können sie im Browser gespeichert werden. Einige Methoden hierfür werden im Folgenden erläutert.

Web Storage

Das Web Storage API ist ein Web-Standard mit dessen Hilfe Daten als Schlüssel / Wert Paare im Browser gespeichert werden können. Es wird, wie Abbildung 2.2 zeigt, von allen, bis auf den Opera Mini, Browsern unterstützt. Opera Mini speichert grundsätzlich keine Daten, weswegen dieser Browser auch in den nachfolgenden Bildern mit Rot markiert ist. Web Storage umfasst zwei Mechanismen. Den Session Storage und den Local Storage. Der Session Storage existiert nur so lange der Browser geöffnet ist. Das heißt alle Daten die im Session Storage gespeichert werden, existieren nicht mehr sobald der Browser geschlossen wird. Daten die im Local Storage gespeichert sind, existieren dort solange bis der Browser Cache geleert wird [webd].

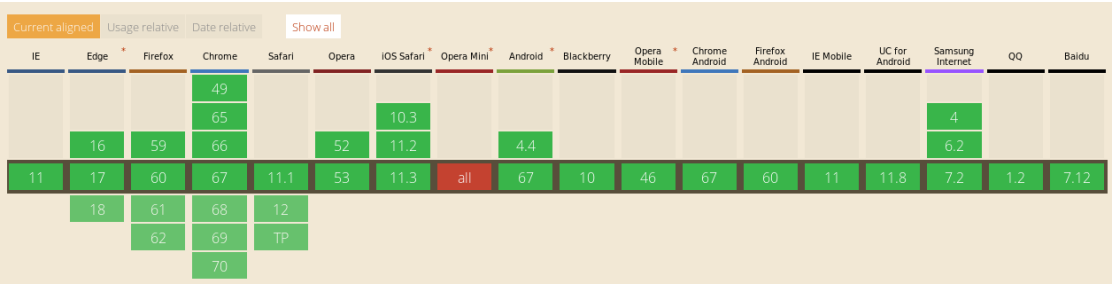


Abbildung 2.2: *Browserkompatibilität für Web Storage, Quelle: [cand]*

Der von den Browsern freigegebene Cache-Space für Web Storage variiert, ist aber meist auf 10 MB begrenzt. Der größte Nachteil ist wohl, dass Web Storage synchron arbeitet und so andere Operationen, wie zum Beispiel das Rendern der Seite, blockieren kann [Hei12].

Web SQL

Eine andere der lokalen Speicherung im Browser ist die Web SQL Datenbank. Sie hat ein asynchrones API und unterstützt die grundlegenden SQL Abfragen. Web SQL sollte in den W3C Standards aufgenommen werden. Aus Mangel unabhängigen Implementierungen wie eine andere Datenbank (DB) als SQLite im Backend wurde es abgelehnt [webc]. Das Web SQL API wird nur von den Webkit–Browsern unterstützt. Also nicht von Firefox, dem Internet Explorer oder dessen neueren Variante Edge [canc].

IndexedDB

IndexedDB ist eine weitere Variante der clientseitigen Datenspeicherung. Die auf JavaScript basierende, objektorientierte Datenbank erlaubt neben dem Speichern von größeren Datenmengen in Form von Objekten auch das Speichern von Dateien. Durch die Verwendung von Indizes lassen sich Objekte schnell speichern und finden. Das asynchrone API erlauben Datenbankabfragen die keinen anderen Prozess blockieren [idba].

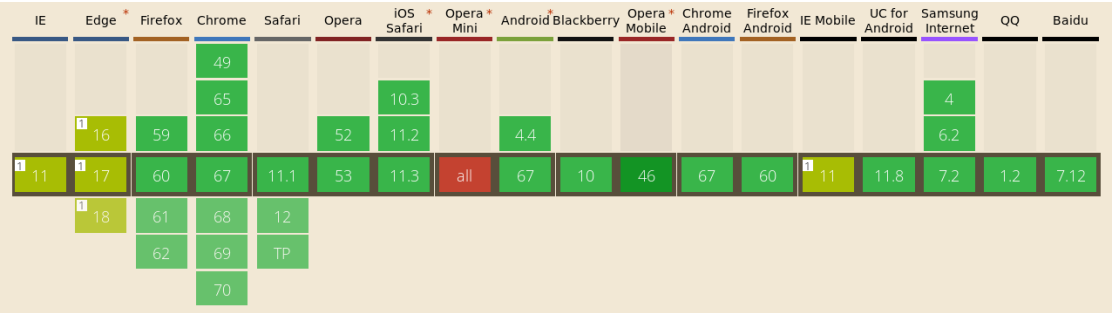


Abbildung 2.3: *Browserkompatibilität für IndexedDB, Quelle: [cana]*

Wie in Abbildung 2.3 zu sehen ist, wird IndexedDB von allen gängigen Browsern unterstützt.

IndexedDB 2.0

Im Januar 2018 wurde die zweite Version des IndexedDB APIs zur W3C Spezifikation hinzugefügt. Es erweitert die erste Verion um Funktionalität und verbessert die Performance [idbb]. Die Aktualität dieser Spezifikation spiegelt sich in der Browserunterstützung wider. Die Abbilung 2.4 zeigt, dass die zweite Version nur von den beliebten (haha) Desktopbrowsern und einigen mobilen Browsern unterstützt wird.

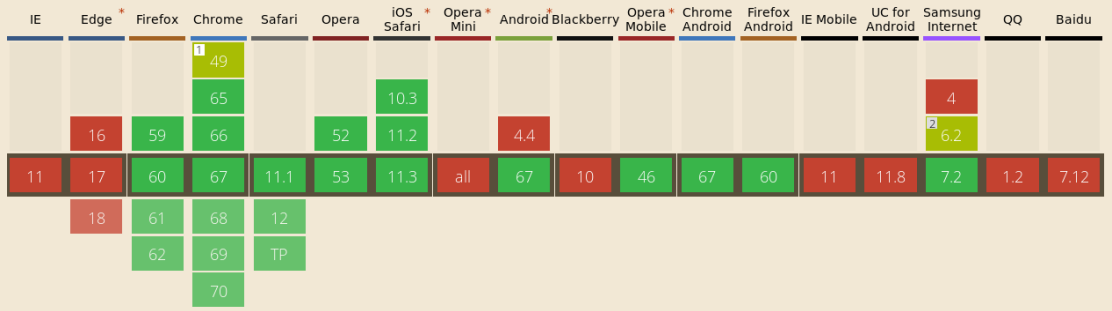


Abbildung 2.4: *Browserkompatibilität für IndexedDB 2.0, Quelle: [cana]*

IndexedDB wird stetig weiterentwickelt und es gibt bereits einen Spezifikationsentwurf für die dritte Version [idbc].

2.1.3 Datenbanksynchronisation

Eine Datenbanksynchronisation ist dann vonnöten, wenn die Anwendung auf mehr als einem Gerät laufen soll. Idealerweise wird hierfür eine Datenbank gewählt, die einen Synchronisationsalgorithmus bereitstellt. Im Allgemeinen ist ein Synchronisationsprotokoll die Möglichkeit für zwei Partenen, beispielsweise Client und Server, den Zustand voneinander zu kennen. Der Client schickt seine Daten an den Server und umgekehrt, solange bis beide Parteien denselben Zustand haben. Leider sind Synchronisationsprotokolle schwer zu implementieren und führen häufig zu einem frustrierenden Ergebnis: Dokumente oder Fotos werden nicht repliziert, es gibt doppelte oder verlorene Daten und alle möglichen Arten von Fehlverhalten bei kollaborativen Anwendungen.
In Kapitel 3 werden einige Technologien vorgestellt, die einen integrierten Synchronisationsmechanismus besitzen.

2.2 Konfliktmanagementstrategien

Diese Arbeit untersucht die Konfliktmanagementstrategien offlinefähiger Systeme. Nachdem in Abschnitt 2.1 beschrieben wurde was eine Anwendung offlinefähig macht, wird in diesem Abschnitt definiert was ein Konflikt ist und wie er entsteht.

2.2.1 Konflikte

Als Konflikt wird die Situation beschrieben in der es ein Dokument mit unterschiedlichen Informationen in mehreren Geräten oder Datenbanken gespeichert ist (vgl. [J. 10] S.

153). Konflikte gehören in verteilten Systemen zur Realität und lassen sich nicht vermeiden. Ein verteiltes System ist per Definition ein Zusammenschluss unabhängiger Computer, die sich für die NutzerInnen als ein einziges System präsentieren (vgl. [Att04] S. 2). Einfacher gesagt besteht ein verteiltes System, sobald zwei oder mehrere Computer über das Netzwerk miteinander verbunden sind. Die spezielle Eigenschaft von Netzwerken ist jedoch, dass die Verbindung jederzeit abbrechen kann. Gareth Wilson beschreibt in seinem Artikel die acht Irrtümer der verteilten Datenverarbeitung [fal]:

1. Das Netzwerk ist zuverlässig
2. Die Latenzzeit ist gleich null
3. Die Bandbreite ist unendlich
4. Das Netzwerk ist (informations)sicher
5. Die Netzwerkstruktur wird sich nicht ändern
6. Es gibt eineN AdministratorIn
7. Die Datentransportkosten sind gleich null
8. Das Netzwerk ist homogen

Aus diesen irrtümlichen Annahmen über das Netzwerk ergeben sich Fehlerszenarien. So kann es passieren, dass der zweite Computer entweder sehr weit entfernt, sehr beschäftigt oder ausgeschaltet ist. Diese Fehlerszenarien können dazu führen, dass ein Konflikt entsteht.
Anhand des folgenden Beispiels wird ein mögliches Fehlerszenario für den Fall des unzuverlässigen Netzwerks aufgegriffen. Zwei Personen treffen sich im Zug und verstehen sich auf Anhieb sehr gut. Person A, nennen wir sie Amilia, gibt Person B, nennen wir sie Rory, ihre Telefonnummer. Der Zug fährt durch einen Tunnel und das Netzwerk bricht ab als Rory sie in ihr Adressbuch, das in Form einer App auf ihrem Laptop gespeichert ist, schreibt. Amilia diktiert ihre Telefonnummer falsch, mit einem Zahlendreher, weil sie die Nummer noch nicht lange hat. Zur Sicherheit schickt Amilia Rory ihre Nummer zusätzlich per E-Mail. Rory schaltet ihren Laptop aus, weil er sich mit Amilia unterhalten möchte. Am Abend ist Rory zu Hause angekommen und sie speichert Amilias Nummer aus der E-Mail in ihrem Adressbuch auf seinem stationären Desktop PC. Jetzt gibt es Amilias Telefonnummer mit unterschiedlichen Informationen in Rorys Adressbuch auf verschiednene Geräten. Wenn Rory am nächsten Tag seinen Laptop startet und das Adressbuch sich mit dem auf dem PC synchronisiert entsteht ein Konflikt. Die falsche Telefonnummer wird gespeichert und die richtige ist verloren.

Konflikte sind in zwei Kategorien einzuteilen. Es gibt solche, die vom System gelöst werden können und welche die eine spezielle Behandlung brauchen. Gibt es eine gleichzeitige Änderung an unterschiedlichen Stellen eine Dokuments muss das kein Problem darstellen. Das Dokument kann beide Aktualisierungen erhalten indem die Änderungen zu-

sammengefügt werden. Diese Prozedur wird *merge* genannt und wird unter anderem von Git[2], einer Software zur verteilten Versionsverwaltung, angewandt. Diese Art Konflikt kann selbstständig vom System gelöst werden.

Die Konflikte die durch die gleiche Änderung an ein und derselben Stelle des Dokuments entstehen, benötigen eine aufwändigere Behandlung. Es muss festgestellt werden welche Version die korrekte ist und gespeichert werden soll. Die Wiederherstellung der Datenkonsistenz bei Konflikten kann dazu führen, dass einige oder alle Aktualisierungen ganz oder teilweise gelöscht werden. Zur Lösung dieses Problems wurden verschiedene Managementstrategien entworfen die im Folgenden vorgestellt werden.

2.2.2 Last-Write-Wins

Der Last-Write-Wins (LWW) Ansatz geht davon aus, dass Schreibvorgänge immer in der richtigen Reihenfolge ausgeführt werden. Der letzte an die Datenbank gesendete Schreibvorgang wird als korrekt angenommen. Diese Strategie ist leicht zu implementieren da lediglich festgestellt werden muss welche Manipulation die neuere ist, was durch die Vergabe eines Zeitstempels problemlos zu errechnen ist [Yar17].

2.2.3 Operational Transformation

Operational Transformation (OT) ist eine weit verbreitete Technik zur Unterstützung von Funktionalitäten in Kollaborativer Software. Sie stammt aus einer im Jahre 1989 veröffentlichten Forschungsarbeit und wurde ursprünglich nur für die gemeinsame Bearbeitung von Klartext-Dokumenten entwickelt [Ell89]. Etwas später wurden einige Unvollständigkeiten im Algorithmus entdeckt und es wurden unabhängig voneinander verschiedene Lösungsvorschläge gebracht [Sun98]. Heute unterstützt OT zusätzlich das Kollaborative Bearbeiten von HTML, RTF und XML Dokumenten, Adobe Flash[3] Grafiken und Dokumenten in CAD[4] Tools wie Autodesk Maya[5]. Des Weiteren können mit OT Dateien die in Microsoft Office[6] enthalten sind wie Word Dokumente, PowerPoint Folien und Excel Tabellen kollaborativ bearbeitet werden, aber auch Dokumente in webbasierte An-

[2] git steht unter `https://git-scm.com/downloads` zum Download bereit

[3] Browserplugin, steht unter `https://get.adobe.com/de/flashplayer/` zum Download bereit

[4] CAD, engl.: computer-aided design, deutsch: rechnerunterstütztes Konstruieren

[5] 3D Design Software. steht unter `https://www.autodesk.com/products/maya/free-trial` zum Download bereit

[6] Bürosoftware, steht unter `https://products.office.com/de-de/compare-all-microsoft-office-products?tab=1` zum Download bereit

wendungen wie Google Docs[7] und Google Wave[8] [Sun11b].

Operational Transform ist ein Algorithmus für die Transformationen von Operationen, die auf Dokumente mit unterschiedlichen Zustand angewendet werden, um diese Dokumente in den identischen Zustand zu versetzen. Jede Änderung an einem freigegebenen Dokument wird als Operation dargestellt. Eine Operation ist die Repräsentation einer Änderung an einem Dokument und zeichnet im Wesentlichen den Unterschied zwischen einer und der nachfolgenden Version eines Dokuments auf. Die Anwendung einer Operation auf das aktuelle Dokument führt zu einem neuen Dokumentstatus. Es gibt beispielsweise die Operation *Einfügen*. Das Einfügen besteht aus dem eingefügten Text und dessen Position im Dokument. Die Operation die beschreibt 'Füge den Buchstaben x an dritter Stelle im Dokument ein' sieht so aus: (`insert('x', 3)`). Für die Position kann ein Koordinatensystem ermittelt werden, Zeilennummer und die Position in einer Zeile, oder das Dokument wie eine Folge von Zeichen behandeln werden.
Kollaborative Systeme, die OT verwenden, benutzen normalerweise den replizierten Dokumentenspeicher. Das heißt auf jedem Gerät befindet sich eine eigene Kopie des Dokuments. Die Operationen erfolgen auf der lokalen Kopie und die Änderungen werden an alle anderen Geräte weitergegeben.

Um konkurrierende Operationen zu behandeln, gibt es die `transform` Funktion. Wenn ein Client die Änderungen von einem anderen empfängt, werden die Änderungen vor ihrer Ausführung transformiert. Sie nimmt zwei Operationen die auf demselben Dokument auf unterschiedlichen Geräten angewendet wurden und berechnet daraus eine neue. Die neue Operation wird dann nach der zweiten Operation angewendet, die erste, beabsichtigte Operation wird beibehalten [Ell89].
Die Grundidee des OT Algorithmus wird anhand eines Beispiels, das die Abbildung 2.5 veranschaulicht, illustriert.
Es gibt den initialen Text "abc", der auf den Geräten der kollaborativ arbeitenden Personen, Amilia und Rory, besteht. Amilia fügt lokal vor dem "abc" ein "x" ein, Rory löscht auf ihrem Gerät den Buchstaben "c"'. Die konkurrierenden Operationen sind `insert(0, "x")`, 'Füge das Zeichen "x" an Stelle Null ein', und `delete(2, 1)`, 'lösche ein Zeichen an der zweiten Stelle'. In OT werden lokale Änderungen angewandt, so wie sie passieren. Operationen über das Netzwerk werden, bevor sie auf zuvor ausgeführte Operationen angewandt werden, transformiert.
Wenn die Operation von Rory zuerst ausgeführt wird, gibt es kein Problem. Der Buchsta-

[7] Webanwendung, mit der Dokumente erstellt und kollaborativ bearbeitet werden können, `https://docs.google.com`

[8] Google Wave war eine von Google vorgestelle Webanwendung zur Kommunikation und kollaborativer Zusammenarbeit

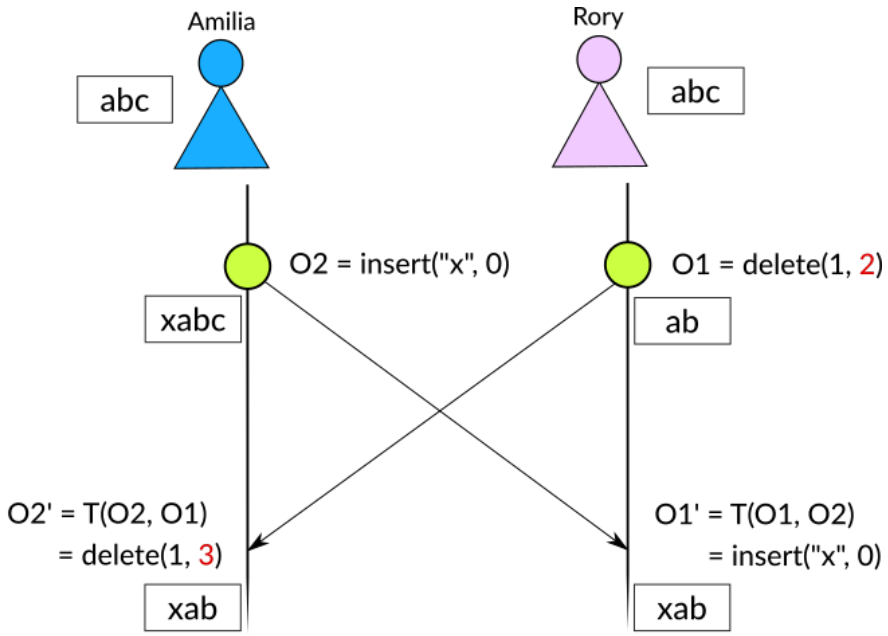


Abbildung 2.5: *Die Grundidee von OT*

be an zweiter Stelle wird gelöscht und danach wird an Stelle Null ein ”x” eingefügt. Das Ergebnis ist das gewünschte ”xab”. Im Bild ist auf der rechten Seite der Ablauf dieses Vorgangs abzulesen, wie er in OT stattfindet. Rory löscht den Buchstaben ”c” und im Dokument steht ein ”ab”. Alimias Editierung, *O2* kommt an und wird gegen *O1* transformiert. In diesem Fall ist das Ergebnis der Transformation dasselbe wie vorher. Die Ausführung von Rories Änderung hat keinen Einfluss auf die von Amilia. Wird Amilias Änderung auf ”ab” angewendet, wird das ”x” an erster Stelle hinzugefügt und das Ergebnis, ”xab” ist identisch mit dem auf Amilias Seite.
Würde nämlich die Operation von Amilia zuerst ausgeführt werden, gäbe es Problem. Dann würde der Buchstabe ”b” gelöscht, da er sich, nachdem das ”x” eingefügt wurde, an zweiter Stelle steht. Das Ergebnis nach der ersten Operation wäre ”xac”. Die Dokumentenstatus wären nicht identisch.
Auf der linken Seite wird die Operation *O2* zuerst ausgeführt im Dokument steht ”xabc”. Dann kommt *O1* auf Amilias Gerät an und wird zu *O2'* transformiert. Die neue Operation ist nun `delete(1, 3)`, ’lösche ein Zeichen an dritter Stelle’. Der Positionsparameter wurde um eins hochgesetzt, weil das Einfügen des Buchstabens ”x”, durch die konkurrierenden Operation, beachtet wurde. Wird nun *O2'* auf der Operation *O1* angewendet, wird der korrekte Buchstabe gelöscht und das Ergebnis ist, wie auf Rories Seite, ein ”xab”. Wird die Operation `delete(1, 2)` gegen die Operation von Amilia transformiert, wird berücksichtigt, dass Amilia ein Zeichen vor der Position zwei eingefügt hat.

Zuammenfassend besteht die Grundidee von OT darin, eine Operation in eine neue Form umzuwandeln. Die neue Form ergibt sich aus den Auswirkungen zuvor durchgeführter Operationen. So können die transformierten Operationen die korrekte Wirkung erzielen und eine Konsistenz der replizierten Dokumente sicherstellen [Sun98].

2.2.4 Conflict-Free Replicated Data Type

Ein Conflict-free replicated data type (CRDT) ist eine spezielle Datenstruktur die in verteilten Systemen auf mehreren Geräten repliziert werden kann. Jede Operation wird an alle Replikate gesendet. Jedes Replikat wendet alle ankommenden Aktualisierungen an und ein Algorithmus löst alle konfliktbehafteten Aktualisierungen auf, wodurch sichergestellt ist, dass Konflikte gar nicht erst auftreten [Sha11a]. Eine Synchronisation ist nicht notwendig, da die Aktualisierung sofort ausgeführt wird [Sha11b].

Es gibt zwei Konzepte bei Replikationsmodellen mit CRDTs: den zustandsbasierten und den operationsbasierten Ansatz.

Zustandbasierter Ansatz

Wenn ein Replikat eine Aktualisierung von einem Client empfängt, wird es zuerst auf den lokalen Status angewandt. Dann sendet es etappenweise eine Kopie des eigenen, aktualisierten Status an andere Replikate im System. Wenn ein Replikat den Status eines anderen Replikats empfängt, führt es mit einer `merge` Funktion den empfangenen Status mit dem lokalen zusammen. Entsprechend sendet dieses Replikat gelegentlich auch seinen Status an ein anderes Replikat, sodass jede Aktualisierung schließlich alle Replikate im System erreicht [Sha11b]. Die folgende Abbildung stellt eine zustandsbasierte Replikation mit drei Geräten dar.

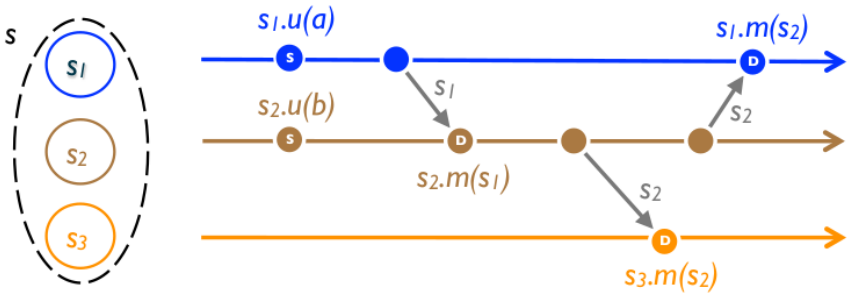


Abbildung 2.6: *Replikation von zustandsbasierten CRDTs, Quelle: [Sha11b]*

Die drei Kreise auf der linken Seite repräsentieren drei identische Datensätze auf drei

Geräten. Die Datensätze *S1* und *S2* werden zeitgleich mit unterschiedlichem Kontent aktualisiert. Kurz darauf, weiter rechts im Bild, sendet *S1* seinen neuen Status an *S2*. *S2* führt die beiden Status, den von *S1* empfangenen und den eigenen, aktualisierten, in der `merge` Funktion m zusammen. Dann schickt es den eigenen, neuen Status an *S3* und *S1*, welche ebenfalls den empfangenen mit dem eigenen Status zusammenführen. Beide Aktualisierungen haben nun alle Geräte erreicht und alle Daten sind identisch.

Damit die Replikation konfliktfrei funktioniert müssen einige Voraussetzungen erfüllt sein. Der Status, den ein CRDT hat muss ein Semi–Gitter, also eine geordnete Menge abbilden. Die Aktualisierungen müssen zunehmend sein, ein Status kann beispielsweise eine Zahl sein und die Aktualisierung ist die Operation die sie inkrementiert. Die `merge` Funktion muss die kleinste obere Grenze der letzten Aktualisierung berechnen. Nur wenn ein Objekt diese Eigenschaften erfüllt, ist es dem CRDT zugehörig [Sha11b]

Operationsbasierter Ansatz

Wenn ein Replikat eine Aktualisierung von einem Client empfängt, wird es ebenfalls zuerst auf den lokalen Status angewandt. Im Gegensatz zum zustandsbasiertem Ansatz wird nicht der gesamte Status des Replikats gesendet, sondern nur der Aktualisierungsvorgang. Ein weiterer Unterschied ist das Fehlen der `merge` Funktion. Statt ihrer gibt es im operationsbasierten Ansatz zwei `update` Methoden: eine vorbereitende Aktualisierungsfunktionen und eine ausführende. Erstere wird auf dem Replikat angewandt, umgehend gefolgt von der zweiten. Über ein Kommunikationsprotokoll wird die Aktualisierung an alle anderen Replikas asynchron versendet. Über das Protokoll wird sichergestellt, dass die Nachricht nur einmal überliefert wird. Die restlichen Replikas wenden die Operation mit der ausführenden Aktualisierungsmethode auf sich an [Sha11b]. Die folgende Abbildung stellt eine operationsbasierte Replikation mit drei Geräten dar.

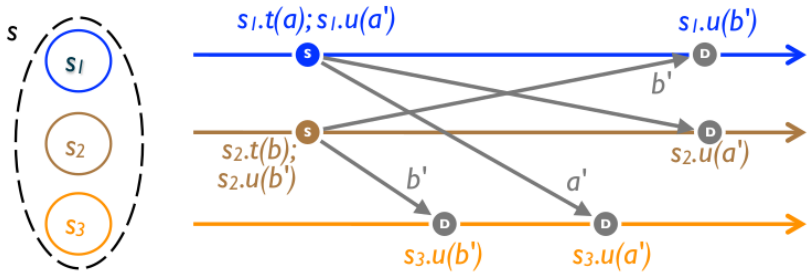


Abbildung 2.7: *Replikation von Operationsbasierten CRDTs, Quelle: [Sha11b]*

Die drei Kreise auf der linken Seite repräsentieren drei identische Datensätze auf drei

Geräten. Die Datensätze *S1* und *S2* werden zeitgleich mit unterschiedlichem Kontent aktualisiert. Sowohl *S1* also auch *S2* wenden die vorbereitende Aktualisierungsmethode t auf sich an. Dann, mit dessen Ergebnis, die ausführende u. Sofort senden beide Replikas die Aktualisierung an alle anderen, welche ebenfalls die ausführende Aktualisierungsmethode u auf sich anwenden.

Die Voraussetzung für eine konfliktfreie Replikation sind hier kommuntative Aktualisierungen. Nur so spielt die Reihenfolge in der die Aktualisierungen bei den Replikaten ankommen keine Rolle – das Ergebnis ist dasselbe [Sha11b].

Umgesetzte CRDTs sind zum Beispiel Counter, ein Datenobjekt das zum Zählen verwendet wird. Es gibt einen, der nur hochzählen kann und eine andere Variante die auch die Subtraktion unterstützt. Weitere CRDTs sind Sets, eine Listenrepräsentation ohne Duplikate. Auch hier gibt es eine Variante mit der nur Daten hinzugefügt werden können und eine die auch das Entfernen der Daten erlaubt [Sha11a].

2.2.5 WIP Logische Uhr?

2.2.6 WIP? Differential Synchronization?

Wenn noch Zeit ist

2.2.7 Das CouchDB Replikationsmodell

CouchDB ist ein quelloffenes, dokumentenorientiertes Datenbank Management System (DBMS) mit einem integrierten Replikationsprotokoll.
Die Aufgabe der Replikation von CouchDB ist die nahtlose, direkte Datensynchronisation zweier oder mehrerer Datenbanken. CouchDB verwendet Replikation um Änderungen an Dokumenten zwischen einzelnen Knoten zu synchronisieren. Hierbei werden nur die Dokumente übertragen, die neu sind oder sich geändert haben. Die Replikation in CouchDB erfolgt schrittweise. Alle Änderungen an Dokumenten werden periodisch zwischen den Servern kopiert. Wenn der Replikationsprozess unterbrochen wurde weil eine Datenbank keinen Internetzugang hat, haben zwei sich replizierende Datenbanken unterschiedliche Daten gespeichert – einen inkosistenten Status. Bei wieder bestehender Internetverbindung wird die Replikation erneut ausgelöst und CouchDB setzt an dem Punkt an dem es ausgefört hat, die Arbeit fort.

Das Besondere an CouchDB ist, dass es darauf ausgerichtet ist, Konflikte vernünftig zu

behandeln statt anznehmen es träten keine auf. Das interne Replikationssystem besitzt eine automatische Konflikterkennung und –lösung.
Wichtig für diesen Mechanismus sind die Revisionsnummern. Dokumente werden mit Revisionsnummern versioniert. Mit jeder Aktualisierung bekommt es eine neue Revision, die neben der alten gespeichert wird (vgl. [J. 10] S. 15ff & S. 150ff). Eine Revisionsnummer in CouchDB kann wie folgt aussehen `2-5560348cec1b08c3d53e1508b4a46868` und ist in zwei Bereiche zu teilen. Die Zahl vor dem – erhöht sich mit jeder neuen Revision des Dokuments, also mit jeder Aktualisierung. Alles hinter dem Strich ist ein md5–Hash aus dem Dokumenteninhalt, den Dateianhängen und dem `_deleted` Attribut[9]. Jede Revision hat außerdem eine Liste von vorherigen Revisionen.

Hat ein Dokument durch gleichzeitiges Bearbeiten in zwei unterschiedlichen Datenbanken dieselbe Revisionsnumer, erkennt CouchDB den Konflikt und markiert indem das Dokument das Attribut `_conficts` mit dem Wert `true` bekommt. Dann entscheidet CouchDB, welche Version gewinnt und welche verliert. CouchDB wird nie zwei Versionen zusammenführen. Das muss in der Anwendung implementiert sein. Die Entscheidung darüber, welche Version gewinnt erfolgt über den Längenvergleich der Revisionslisten. Die Version mit der längsten Liste aus vorherigen Revisionen gewinnt. Sind beide Listen gleich lang, gewinnt die Revision die laut alphabetischer Sortierung am größten ist.
Auch wenn CouchDB die gewinnende und die verlierende Revision festlegt, werden beide Versionen gespeichert. Die gewinnende Revision wird als letztes gespeichert, die verlierende davor. Diese Konfliktlösungsstrategie wird auf allen CouchDB Instanzen angewandt, weswegen dazu keine Internetverbindung notwendig ist. Dadurch, dass alle Instanzen denselben Algorithmus verwenden werden die Revisionen immer in identischer Reihenfolge gespeichert. Dadurch bleiben die Daten konsistent.
Jetzt kann im Entwicklungsprozess der Anwendung entschieden werden, wie mit den Konflikten umgegangen wird. Es kann aus beiden Version eine festgelegt werden, die behalten wird oder es können beide zusammengeführt werden (vgl. [J. 10] S. 153ff).

Eventual Consistency

Das muss noch woanders hin. oder raus

Das Consistency Availability Partition tolerance (CAP) Theorem, veranschaulicht in Abbildung 2.8, besagt, dass jedes System mit dem Daten über das Netzwerk gesendet werden, nur zwei von den drei möglichen Eigenschaften, Konsistenz, Verfügbarkeit und Partitionstoleranz, garantieren kann. Konsistenz der gespeicherten Daten bedeutet, es muss sichergestellt werden dass nach Abschluss der Transaktion auch alle Replikate des mani-

[9] Dokumente werden in CouchDB nicht ohne Weiteres gelöscht. Stattdessen werden sie als solches markiert.

pulierten Datensatzes aktualisiert werden. Der Datensatz ist in jeder Datenbank identisch.

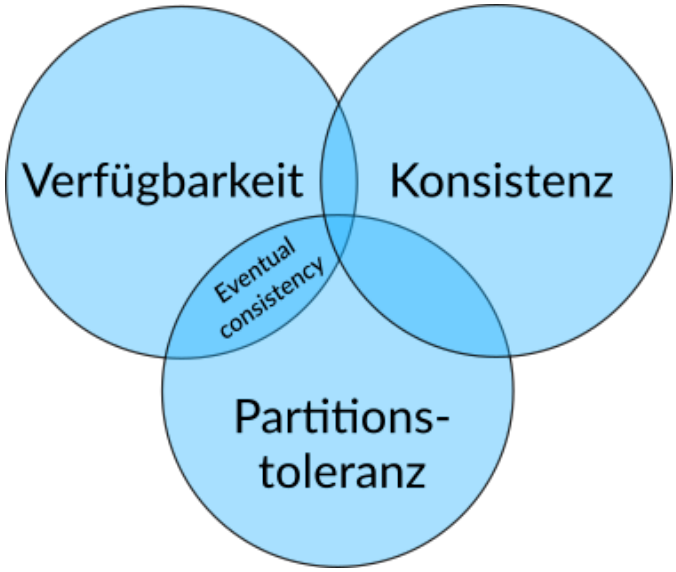


Abbildung 2.8: *Das CAP Theorem*

Das System ist besitzt eine hohe Verfügbarkeit wenn alle Anfragen an das System stets beantwortet werden. Die Verfügbarkeit ist gering, wenn die Antwortzeiten des Systems lang sind. Partitionstoleranz ist gleichzusetzen mit Ausfalltoleranz. Die Datenbank kann auf mehreren Servern verteilt sein. Selbst wenn ein Server oder eine Partition ausfällt, kann das System weiterhin funktionieren.
Eventual Consistency kommt häufig bei verteilten Datenbanken zur Anwendung und stellt die Konsistenz der Daten nach einem gewissen Zeitfenster sicher (vgl. [J. 10] S. 11ff.).

3 Bestehende offlinefähige Technologien

Um eine Webapplikation offlinefähig zu machen, müssen alle Daten auf dem Client gespeichert werden und von diesem zu jeder Zeit abrufbar sein. Für Anwendungen mit einer serverseitigen Datenbank ist die Synchronisation der Daten zwischen Server und Client notwendig.
Es gibt verschiedene Technologien, die sich diesen Problematiken widmen. Diese umfassen Bibliotheken und Frameworks, die die Entwicklung offlinefähiger Anwendungen unterstützen, sowie Datenbanklösungen. In den nächsten Punkten werden einige dieser Technologien näher beschreiben.

3.1 Offline plugin für webpack

Webpack ist ein JavaScript 'Bundler' und bündelt alle Skripte, Bilder und Assets für die Verwendung in Browsern.
Das Offline Plugin bietet Offlinefunktionalität für Webpackprojekte indem es die gebündelten, also von Webpack generierten Assets cached. Dazu benutzt es intern den ServiceWorker und AppCache als Reserve, für den Fall dass der Browser ServiceWorker nicht unterstützt [Sto].
Auch ungebundelte Assets können über das Plugin gecached werden. Diese Dateien müssen dann in den Optionen explizit angegeben werden. Auch der ServiceWorker und der AppCache lassen sich über die Optionen konfigurieren oder auch ausschalten [webb].
Es werden allerdings nur die Assets und nicht die von BenutzerInnen generierten Daten gecached. Diese müssen manuell im gespeichert werden.

3.2 Redux Offline

Redux Offline kann nur zusammen mit Redux verwendet werden [red17]. Deswegen ist für die Verwendung von Redux Offline die Implementierung von Redux vorausgesetzt.

3.2.1 Redux

Redux ist eine JavaScript Bibliothek die Probleme im Zusammenhang mit dem *Zustand* einer Anwendung löst. Redux ist eine Bibliothek zur Zustandsverwaltung in JavaScript-anwendungen. Es gibt einen zentralen Ort, in dem der *Zustand* der App gespeichert ist, auf den von jeder Komponente aus zugegriffen werden kann. Dieser Ort wird Store genannt und jede Applikation hat genau einen davon. Als einzige Informationsquelle für den Store als zentralen Speicher dienen Aktionen. Sie senden Daten von der Anwendung mittels `store.dispatch()` an den Store und beschreiben dabei nicht wie etwas passiert, sondern was passiert. Der dritte wichtige Bestandteil von Redux sind die Reducer. Sie spezifizieren wie der Status sich als Reaktion auf die Aktionen ändert [reda].
Der Datenfluss in der Reduxarchitektur ist unidirektional. Zur Veranschaulichung wird anhand der folgenden Abbildung der Redux Datenfluss beschrieben.

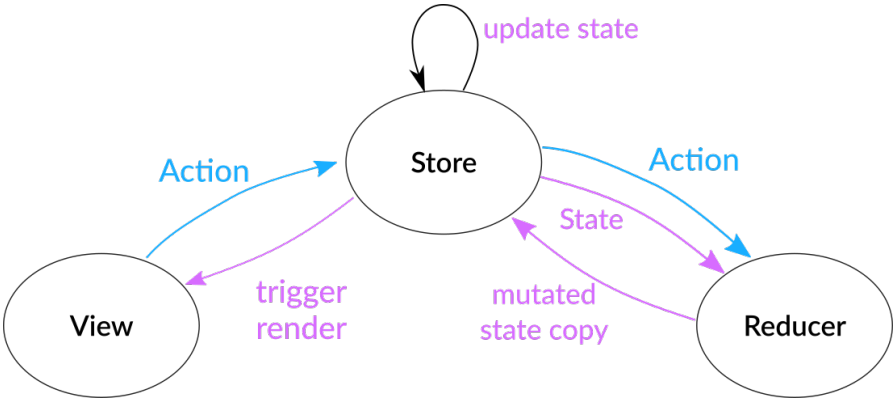


Abbildung 3.1: *Redux Datenfluss*

Zuerst sendet die View eine Aktion an den Store. Dieser empfängt die Aktion und schickt sie zusammen mit dem Applikationsstatus an den Reducer. Der Reducer erstellt eine Kopie des Status, verändert diese und schickt sie wieder zurück an den Store. Der Store ersetzt nun den alten mit dem neuen Status und löst ein erneutes Rendern der View aus.

3.2.2 Redux Offline

Redux Offline erweitert Redux um einen persistenten Store mit Offline-First Technologie und ist kompatibel mit allen *View Frameworks wie React[10], Vue[11], oder Angular[12] [redb]. Es umfasst unter anderem netzwerkfähige API-Aufrufe, das Persistieren des Zustands der Anwendung, das Speichern von Aktionen, die Behandlung von Fehlern und erneute

[10] JavaScript Bibliothek: `https://reactjs.org/`
[11] JavaScript Framework: `https://vuejs.org/`
[12] JavaScript Framework: `https://www.angular.io`

Versuche die Verbindung wieder herzustellen. Redux Offline verspricht nicht, die Webanwendung komplett offlinefähig zu machen. Um Assets zwischenzuspeichern, muss zusätzlich noch ein ServiceWorker implementiert sein [redd].
Die Idee hinter Redux Offline ist, dass der Redux STORE die Datenbank ersetzt [Evä17]. Bei jeder Änderung wird der Redux STORE auf dem Datenträger gespeichert, und bei jedem Start automatisch neugeladen. Für das Speichern der Daten in einer lokalen Datenbank wird intern Redux Persist verwendet.

Eine mit Redux Offline erstellte Anwendung funktioniert ohne weitere Codeimplementierung offline im Lesemodus, da das Lesen und Schreiben aus der lokalen Datenbank bereits eigebunden ist. Damit die Anwendung auch im Schreibmodus offline funktioniert, müssen einige Anpassungen vorgenommen werden. Sämtliche Daten der Anwendung können nur über Aktionen manipuliert werden. Alle netzwerkgebundenen Aktionen werden in einem STOREinternem Queue gespeichert und müssen mit einem Metaattribut dekoriert werden um offline arbeiten zu können. Durch die Metaattribute weiß die Anwendung was vor der eigentlichen Ausführung der Aktion und was danach zu tun ist. Es gibt drei Metadaten die Redux Offline interpretieren kann:
`meta.offline.effect` - Die initiale Aktion wird ausgeführt. Hier kann eine URL angegeben werden die Redux Offline anfragen soll.
`meta.offline.commit` - Hier wird die Aktion definiert die ausgeführt wird sobald die Netzwerkanfrage erfolgreich ist.
`meta.offline.rollback` - Hier kann die Aktion angegeben werden, die bei permanent fehlgeschlagener Internetverbindung oder wenn der Server einen Serverfehler zurückgibt gefeuert wird. Dann fügt Redux Offline dem APPSTATE automatisch ein `offline` Objekt hinzu. Dort wird unter anderem ein Array namens `outbox` verwaltet wird. Dieses Array repräsentiert den Queue. Hier werden die Aktionen inklusive Metadaten gespeichert, um bei bestehender Internetverbindung abgearbeitet zu werden [redc]. Die von Jani Eväkallio erstellte Grafik 3.2 veranschaulicht die oben erklärte Architektur. Links ist eine Aktion zu sehen die Zeug machen möchte. Sie hat ein Mateattribut das weitere Aktionen definiert. Eine Aktion für den Erfolg und eine für den Fehlschlag von 'DO_STUFF'. In der Mitte ist der STORE zu sehen. Der STORE kennt den Netzwerkstatus und hat den Queue namens `outbox` in dem Aktionen mitsamt ihrer Metafelder gespeichert werden. Rechts befindet sich das API, das über die Middleware mit dem STORE redet.
Wird die Aktion 'DO_STUFF' gefeuert gelangt sie in den STORE, damit dieser den APPSTATE aktualisieren kann, und wird ersteinmal im Queue gespeichert. Ist die Anwendung online, wird sie sofort abgearbeitet. Wenn nicht wird sie dort gespeichert bis die Anwendung wieder eine Verbindung zum Internet hat.

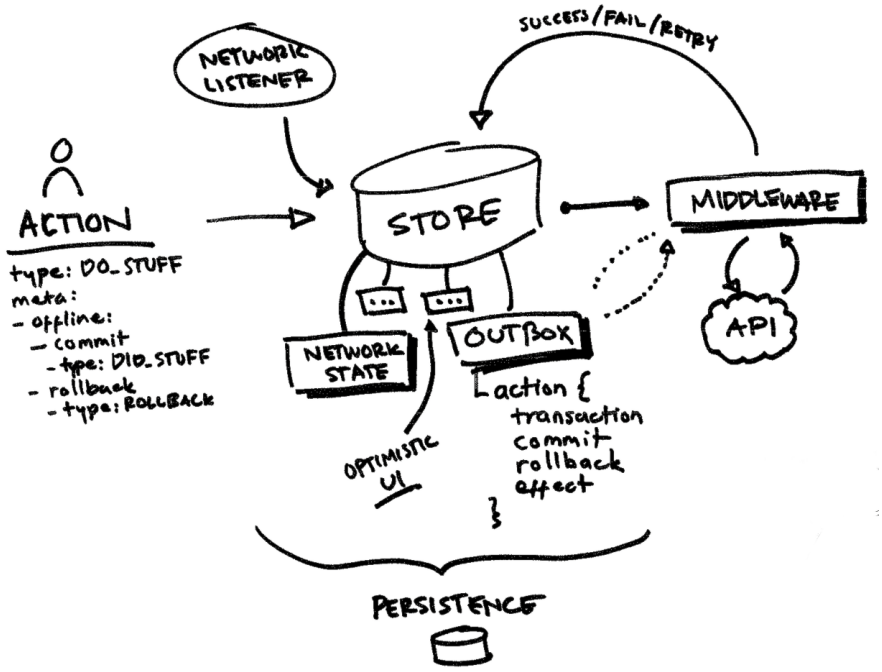


Abbildung 3.2: *Redux Offline Architektur Quelle: [Evä17]*

3.2.3 Redux Persist

Redux Persist ist eine Bibliothek, die als Wrapper für den Redux Store funktioniert. Mit Redux Persist wird der `state` automatisch lokal, per default im LocalStorage, gespeichert [Iwa16]. Es kann konfiguriert werden wo die Daten gespeichert werden. Hier gibt es diverse Möglichkeiten wie zum Beispiel im SessionStorage, per localForage oder in Dateisystemen [redf]. LocalForage ist eine Bibliothek mit der Daten in IndexedDB, WebSQL gespeichert werden können. Wenn der Browser die Speichermöglichkeiten nicht unterstützt, wird der LocalStorage genommen [loc].
Es ist auch möglich einen eigenen Speicher zu konfigurieren. Die einzige Voraussetzung hierfür ist, das API muss die Standardmethoden `setItem`, `getitem` und `removeItem` implementieren und Promises unterstützen [redf].

3.3 React Native Offline

React Native ist ein Framework mit dem native, mobile Apps mit JavaScript und React gebaut werden können [Souc].
Die Bibliothek React Native Offline erweitert das Framework um Offlinefunktionalität. React Native Offline unterstützt die Behandlung des Netzwerkstatus. Dieser kann einmalig oder regelmäßig abgefragt werden um je nach Status z.B. einen anderen Inhalt zu

rendern.
Zusammen mit Redux implementiert können weitere Fähigkeiten genutzt werden. Genau wie Redux Offline hat React Native Offline einen Offline Queue in dem Aktionen gespeichert werden können. Allerdings nur solche Aktionen die fehlgeschlagen sind weil die Anwendung nicht mit dem Internet verbunden ist. Auch hier wird der Aktion ein Metaattribut gegeben. Dieses hat die Felder `retry` und `dismiss`. Das erste erwartet ein Boolean, es kann angegeben werden ob die Aktion noch einmal bei bestehender Internetverbindung ausgeführt werden soll oder nicht. Das Feld `dismiss` erwartet ein Array. Hier können Aktionen angegeben werden die das Wiederholen der Aktion abbrechen [Acu].

Ein Beispiel im folgenden Listing soll die Funktionsweise der Metaattribute besser beschreiben.

```
const action = {
  type: 'FETCH_CONTACT',
  contact,
  meta: {
    retry: true,
    dismiss: ['CANCEL']
  }
}
```

Listing 3.1: *Beispiel React Native Offline Aktion*

Bei Aufruf dieser Aktion soll ein Kontakt geladen werden. Im Metafeld ist `retry` auf `true` gesetzt. Wurde die Aktion im Offlinemodus versucht auszuführen, wird sie erneut aufgerufen sobald die Anwendung wieder einen Internetzugang hat. Die Aktionen die in `dismiss` angegeben werden unterbrechen diesen Vorgang. Wurde also der Kontakt im Offlinemodus angefragt und dann die Aktion 'CANCEL' gefeuert, wird 'FETCH_CONTACT' aus dem Queue gelöscht und nicht erneut ausgeführt.

3.4 HOODIE

HOODIE ist eine JavaScript Bibliothek für offlinefähige Webapplikationen, die ein komplettes Backend zur Verfügung stellt. Wird HOODIE für die Entwicklung einer Webanwendung verwendet, muss also lediglich das Frontend implementiert werden. Den Rest erledigt die Bibliothek. Über eine integrierte Programmierschnittstelle kommuniziert die Anwendung mit dem von HOODIE zur Verfügung gestelltem Backend. Über das API können unter anderem BenutzerInnen authentifiziert, Daten gespeichert und synchronisiert werden [hooa].

Anhand der Abbildung 3.3 wird erklärt wie HOODIE funktioniert.

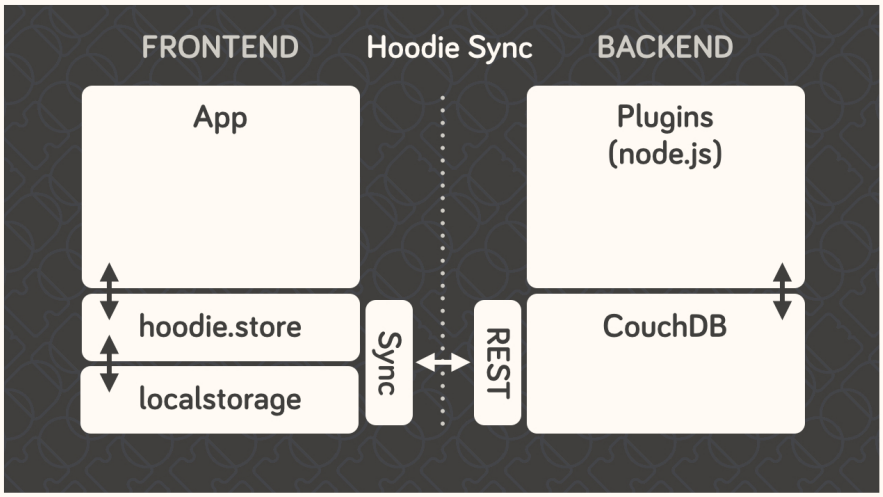


Abbildung 3.3: *HOODIE Architektur Quelle: [hoob]*

Im Frontend–Bereicht ist die App zu sehen die über das HOODIE API mit dem lokalen Speicher kommuniziert. Die Anwendung spricht niemals direkt mit dem Server oder der Datenbank. Für die lokale Speicherung der Daten benutzt HOODIE intern PouchDB, was wiederum IndexedDB verwendet. Durch das lokale Speichern sind die Daten auch offline verfügbar. Dann werden über eine REpresentational State Transfer (REST) Schnittstelle mit einer CouchDB synchronisiert. CouchDB ist eine Datenbank mit der Superkraft des Synchronisierens und in HOODIE haben alle AnwenderInnen ihre eigene private CouchDB. Hinter der Datenbank befindet sich ein kleiner Server der auf die Daten in der CouchDB reagiert, die wiederum die Änderungen an den Client schickt [hoob]. So können NutzerInnen nur auf ihre eigenen Daten zugreifen. Wenn es mehrere Geräte gibt, die mit einem Account assoziiert werden, werden die Änderungen von einem Gerät zuerst auf die serverseitige CouchDB synchronisiert, um dann von dort in die lokalen Datenbanken der anderen Geräte zu gelangen.
Dadurch, dass das Frontend und das Backend nicht direkt miteinander sprechen, ist die Funktionalität beider Komponenten auch dann gewährleistet, wenn die Verbindung unterbrochen wird.

3.4.1 CouchDB

Apache CouchDB™ist ein DBMS das seit 2005 als freie Software entwickelt wird. Die dokumentenorientierte DB funktioniert sowohl als einzelne Instanz, als auch im Cluster, in dem ein Datenbanksserver auf einer beliebig großen Anzahl an Servern oder Virtuellen Masschinen ausgeführt werden kann.

CouchDB verwendet das Hypertext Transfer Protocol (HTTP)–Protokoll und JavaScript Object Notation (JSON) als Datenformat, weswegen es mit jeder Webfähigen Anwendung kompatibel ist. CouchDB wird über ein RESTful HTTP API angesprochen. So können Daten über die für den RESTful Services standardisierten Methoden wie zum Beispiel GET, POST, PUT, DELETE abgerufen oder manipuliert werden.

Das implementierte Replikationsmodell erlaubt die Synchronisation bzw. bidirektionale Replikation zu verschiedenen Geräten, was das besondere Merkmal von CouchDB ist. Die genaue Funktionsweise des Protokolls ist in Unterabschnitt 2.2.7 detailliert beschrieben.
Dank des Replikations–API kann kann sich eine CouchDB kontinuierlich und eigenständig mit einer anderen Datenbank die dasselbe Protokoll implementiert, synchronisieren. Wenn Konflikte auftreten, beispielsweise durch gleichzeitiges Bearbeiten eines Dokuments von zwei Personen ohne Netzwerkverbindung, werden diese als solche markiert, jedoch nicht von selbst aufgelöst [coua]. Die Lösung der Konflikte muss in der Anwendung implementiert sein. So kann gewährleistet werden, dass keinerlei Daten verloren gehen.

CouchDB ist für Server konzipiert. Für Browser gibt es PouchDB und für native iOS- und Android–Apps wurde Couchbase Lite entwickelt. Des Weiteren gibt es noch die Datenbanken Couchbase und Cloudant. Alle verwenden das CouchDB Replikationsprotokoll und können Daten miteinander replizieren und [coua].

3.4.2 PouchDB

PouchDB ist die JavaScript Implementierung von CouchDB. Sie ist quelloffen und wurde so konzipiert, dass sie in allen modernen Browsers läuft. Dort ermöglicht PouchDB es Daten zu persistieren, sodass sowohl offline als auch online verfügbar sind. PouchDB speichert die Daten in IndexedDB und stellt für das Abrufen und Manipulieren derer ein einheitliches API zur Verfügung.
Sind die Daten einmal offline gespeichert können sie, dank des CouchDB Replikationsprotokolls, sobald die Awendung wieder online ist mit CouchDB kompatiblen Servern synchronisiert werden [poua].

3.5 Realm

Realm ist eine Backendtechnologie für mobile Anwendungen und umfasst die Realm Datenbank und den Realm Object Server. Die Datenbank ist quelloffen, der Object Server jedoch nicht – dieser ist außerdem nicht kostenfrei [reaa].

Die Realm Datenbank ist eine objektorientierte, plattformübergreifende lokale Datenbank die eine Echtzeitsynchronisation mit dem Realm Object Server bereitstellt. Der Object Server fungiert als Middleware-Komponente in der mobilen App und handhabt unter anderem die Ereignisbehandlung und Datensynchronisation. Im Zusammenspiel ermöglichen die beiden Technologien die Erstellung von offlinefähigen, kollaborativen, mobilen Anwendungen [rea17b].

Zur Offline First Funktionalität stellt Realm eine umfassende Lösung bereit. Die lokale Realm Datenbank unterstützt die Echtzeitsynchronisation von Daten sodass alle Änderungen sofort automatisch gesendet werden. Das Synchronisationsprotokoll komprimiert statt dem gesamten Objekt nur die marginalen Änderungen und synchronisiert sie auf dem Endgerät und dem Server. Zusätzlich zu den Daten werden die spezifischen Operationen erfasst. Wird beispielsweise ein Kontakt bearbeitet, wird neben den geänderten Daten die Information *update* mitgesendet. Dank dieser zusätzlichen Information kann der Aktionswunsch genau erfasst werden sodass das System eventuelle Konflikte automatisch auflösen kann. Das hat zur Folge, dass die Synchronisation keinen manuellen Eingriff bedarf, der die Leistung des Systems beeinträchtigen könnte [rea17a].
Zusätzlich zu dem OT Algorithmus benutzt Realm vorgegebene Regeln zur automatischen Konfliktlösung. Es gibt drei Grundregeln die die hauptsächlichen Aktivitäten abdecken. Neue Einträge in Listen werden zeitlich sortiert. Für den Fall dass zwei Objekte gleichzeitig zur selben Liste hinzugefügt werden, wird das mit dem neueren Zeitstempel hinter dem älteren Objekt gespeichert. Löschungen haben immer Vorrang; auch dann wenn das auf dem einen Gerät gelöschte Objekt auf einem anderen zu einem späteren Zeitpunkt bearbeitet wurde. Für die Aktualisierungen wird LWW angewandt. Wird ein Objekt auf zwei Geräten bearbeitet, wird das mit dem neueren Zeitstempel behalten.
Es besteht auch die Möglichkeit eigene Regeln zu definieren oder die bestehenden zu überschreiben [reab]. Darüber hinaus passiert die interne Konfliktlösung auf Transaktionsebene. Das heißt der Vorgang ist nur erfolgreich wenn er auch vollständig und fehlerfrei ist. Andererseits wird er zurückgesetzt. Das gewährleistet die Konsistenz der Daten und verhindert deren Verlust wenn Änderungen aufgrund einer unterbrochenen Netzwerkverbindung nicht stattfinden können [rea17a].

3.6 Übersicht

Alle oben genannten Systeme unterstützen, nach eigener Aussage, die Erstellung offlinefähiger Anwendungen. Die folgende Tabelle fasst die oben genannten Systeme zusammen und zeigt inwiefern die Technologien die in Abschnitt 2.1 genannten Voraussetzungen an eine Offline First App erfüllen.

Produkt	Cachen der Assets	Lokale Datenspeicherung	Datenbank-synchronisation
Offline Plugin für webpack	Ja	–	–
Redux Offline	–	Ja	–
React Native Offline	–	Ja	–
HOODIE	–	Ja	Ja
Realm	–	Ja	Ja

Tabelle 3.1: *Übersicht der offlinefähigen Technologien*

Das Offline Plugin für webpack cacht lediglich die Assets einer Anwendung, was sämtliche andere Technologien in dieser Tabelle jedoch nicht tun. Hierbei sollte beachtet werden, dass React Native Offline und Realm für die Entwicklung von mobilen Anwendungen gemacht sind. In mobilen Apps ist das Cachen der Assets nicht notwendig, denn alle Dateien die zur Ausführung notwendig sind, werden bei der Installation auf dem Gerät gespeichert.
Die lokale Speicherung der von den NutzerInnen generierten Daten wird von allen Technologien, bis auf das Plugin für webpack, unterstützt. Hierbei unterscheiden sich die Speicherorte der Daten. Eine Synchronisation zu einer Serverdatenbank stellen allein HOODIE und Realm bereit. Die restlichen Technologien stellen die Verwendung einer Serverdatenbank frei.

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 Anwendung als Client und der Server. Immer wenn beide Parteien miteinander kommunizieren möchten, kann eine der beiden offline sein. Der Client könnte zum Beispiel, ohne dass eine Technologie implementiert ist die Offlinestatus behandelt, einen Kontakt erstellen wollen. Der Kontakt kann aber nicht erstellt werden, da kein Netzwerk verfügbar ist.
‘Client push‘ steht dafür, dass der Client etwas an den Server schickt. ‘Server push/Client pull‘ beschreibt den Fall in dem der Client Daten vom Server anfragt, oder der Server Push-Nachrichten an den Client sendet.
Peer-to-Peer? Als Abstraktion über dem Client-Server-Modell. Ein Client = mal Server, mal Client-Rolle

Szenario C0 – Client push:

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 C1 – Client push:

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

Szenario C2 – Client push:
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 S0 – Server push/Client pull:
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 – Server push/Client pull:
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 – Server push/Client pull:
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 – Server push/Client pull:
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.
Die Abbildung 4.1 veranschaulicht die beschriebenen Situationen, die bei der Übertragung von Daten über das Netzwerk eintreten können. Die in Felder in lila beschreiben die Szenarien bei denen der Client etwas an den Server schickt. Die blauen Felder auf der rechten Seite zeigen solche Szenarien, die eintreten können wenn der Server etwas an den Client sendet. Die Sechsecke stehen für die Ausgangssituation, die Kreise repräsenzieren die Szenarien und die Rechtecke die daraus resultierenden Fälle.

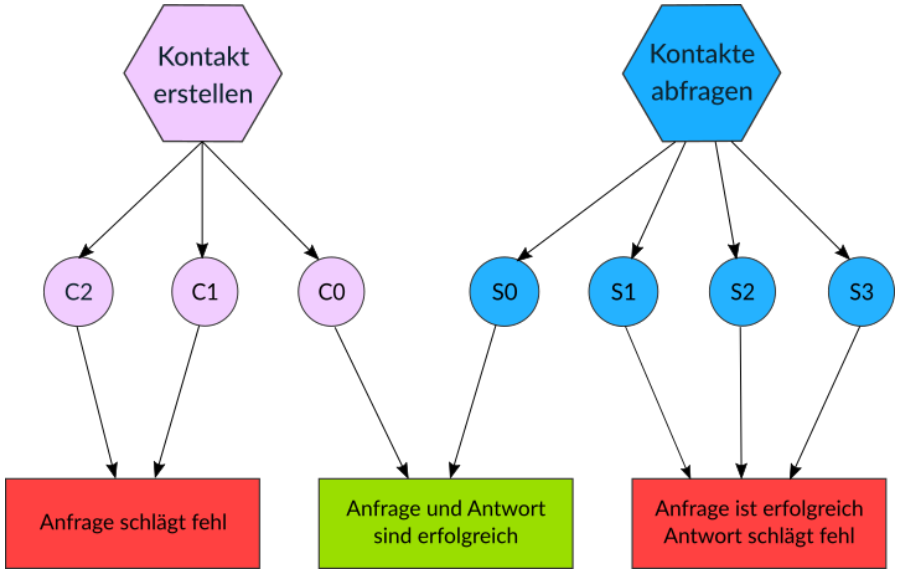


Abbildung 4.1: *Szenarien bei der Datenübertragung über das Netzwerk*

Ergebnis

Da die Szenarien *C0* und *S0*, die Szenarien *C1* und *C2* sowie die Szenazien *S1*, *S2* und *S3* zusammengefasst werden können, ergeben sich aus den sieben 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 schlägt fehl und die Anfrage ist erfolgreich aber die Antwort schlägt fehl.

4.2 Szenarien zur Konfliktentstehung

Im Anwendungsbeispiel einer kollaborativen Kontaktliste können mehrere Personen die Liste verwalten. Oder eine Person benutzt die Anwendung auf mehreren Geräten, was zum selben Ergebnis führt: 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* oder *c* eintritt und ein Objekt

von mehreren Parteien bearbeitet wird.

Aus einem Interview mit Jan Lenhardt[13] ergeben sich unterschiedliche, angewandte Vorgehensweisen Objekte für die Verwendung in verteilten Systemen zu identifizieren und versionieren. Nachfolgend werden diese Vorgehensweisen berücksichtigend, Konfliktszenarien erarbeitet. Es ist zu erwähnen, dass es bei diesen Methoden nicht immer zu Konflikten kommen muss. Da es Thema dieser Arbeit ist, wird für jede Variante nur der ungünstigste Fall, der Konfliktfall aufgeschrieben.

Szenario ID0 – UUID:

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* oder *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 – sprechender Schlüssel:

Zur Identifizierung eines Adressbucheintrags wird ein sprechender Schlüssel[14] 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* oder *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 – Versionsnummer:

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 bearbeitet und aktualisiert und geben ihm beide die Versonsnummer ‘2.0.0‘. Währenddessen tritt Fall *b* oder *c* ein und beide Parteien können nicht miteinander Kommunizieren. Bei der Synchronisation entsteht ein Konflikt weil es zwei unterschiedliche Einträge mit derselben Verion gibt.

[13] Eine berühmte Person

[14] 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 V1 – Zeitstempel:

Zur Versionierung eines Adressbucheintrags wird ein Zeitstempel verwendet. Der Kontakt ‘Amilia‘ hat die initiale 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* oder *c* ein. Es stellt sich heraus, dass die Hausnummer einen Zahlendreher hat und es wird sofort, immernoch offline, berichtigt. ‘Amilia‘ hat nun die Version ‘2018-04-13 11:45:33Z‘.

Der Server hat eine eigene Uhr mit spätere Uhrzeit als der Client. So hat nach der Synchronisation der später korrigierte Eintrag einen früheren Zeitstempel. Es wird die falsche, alte Adresse gespeichert, die korrekte hat den älteren Zeitstempel und wird verworfen.

Diese Variante funktioniert in vielen Fällen gut. Trotzdem kommt es selbst in großen Firmen wie Google zu Problemen[15] wenn verschiedene Geräte eine eigene Uhr besitzen.

Des Weiteren gibt es die Möglichkeit, dass auf dem einen Gerät die Hausnummer berichtigt wurde und auf einem anderen, zur ungefähr selben Zeit, ein Zusatz zur Adresse gespeichert wird. Das könnte die Etage sein in der Amilia wohnt. Dann gibt es zwei Versionen mit unterschiedlichen Zeitstempeln und eine davon wird in jedem Fall verworfen. Auch wenn beide Versionen richtig sind.

Szenario V2 – Logische Uhr:

Weil der Zeitstempel so fehleranfällig ist, wurde die Logische Uhr zur Versionierung von Objekten in Verteilten Systemen die Logische Uhr entwickelt.

Zur Versionierung eines Adressbucheintrags wird eine Logische Uhr[16] verwendet. Der Kontakt ‘Amilia‘ hat die Version Beispiel Logische Uhr?. Amilias Adresse ändert sich und wird auf dem Client angepasst. Währenddessen tritt Fall *b* oder *c* ein. Amilia sieht ihre falsche Hausnummer und berichtigt diese ebenfalls. Bei der Synchronisation kommt es zum Konflikt. wirklich? auch wenn das Ergebnis dasselbe ist?

Szenario V3 – Inhaltsbasierte Version:

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 Adress-

[15] `https://support.google.com/accounts/answer/185834?hl=en#sync`

[16] 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.

bucheintrags gespeichert.
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 V4 – Liste von inhaltsbasierten Versionen:
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. Im selben Status berichtigt der Client die Telefonnummer. Jede Aktion fügt der Versionsliste einen neuen Hashwert hinzu. Auch wenn der Content des Adressbucheintrags in den zwei letzten Versionen identisch ist, kann festgestellt werden welcher der neueste Eintrag ist. Bei der Synchronisation kommt es zum Konflikt weil der Kontakt Amilia mit unteschiedlichen Informationen existiert.
Beide konfliktbehafteten Versionen werden verschachtelt in der Liste gespeichert. In diesem Fall sieht die Liste so aus:
’[[88da3f8d82ab58551d2a48d74d9a4986, 5560348cec1b08c3d53e1508b4a46868], 88da3f8d82ab58551d2a48d74d9a4986]’ – 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*, mittels der Verwendung einer UUID, gewährleistet. Es wird deutlich, dass es in jedem Fall zu einem Konflikt kommen kann. Es gilt zu unterscheiden in welchen Fällen mit Konflikten umgegangen werden muss weil sonst Daten verloren gehen.

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

5 Anforderungsdefinition

Dieses Kapitel beschreibt die Anforderungen an eine Offline First Anwendung unter Berücksichtigung von Funktionalität, Konfliktmanagement und der Bedienoberfläche. Zuerst wird der Umfang dieser Arbeit konkret beschrieben. Dann werdem aus den oben genannten Szenarien die Anforderungen an eine offlinefähige Anwendung hergeleitet.

5.1 Umfang

Das Ziel dieser Arbeit ist die Untersuchung des Konfliktmanagements offlinefähiger Technologien. Dazu soll die beispielhafte Anwendung entwickelt werden, welche an dem Beispiel eines kollaborativen Adressbuchs die Offlinekompatibilität mit dem Schwerpunkt auf das Konfliktmanagement der verwendeten Technologien illustriert. Ein offlinefähiges, kollaboratives Adressbuch ist eine Anwendung mit der mehrere Personen, Kontakte verwalten können. Dass diese Anwendung auch ohne Internetzugang funktioniert, ist obligatorisch. Das Adressbuch beinhaltet eine Liste von Kontakteinträgen, welche jederzeit – unabhängig von der Internetverbindung – von den verwendenden Personen gelesen, bearbeitet, erstellt und gelöscht werden können. Die Offlinefunktionalität soll durch die verwendeten Technologien gegeben sein und wird nicht ausgiebig getestet. 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 Parteien. Der Schwerpunkt liegt auf den Konflikten die entstehen können, wenn die Internetverbindung abbricht. Dabei sind Aspekte aus unterschiedlichen Rollen zu betrachten. So werden die Rollen der AnwenderInnen, der EntwicklerInnen und die der TesterInnen berücksichtigt.

Bein ersten Start der Anwendung müssen, sofern 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. Dazu muss ermittelt werden können, welche Daten neu angelegt oder aktualisiert wurden. 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. Wird ein Eintrag angelegt, bearbeitet oder gelöscht, müssen alle Parteien wissen um welchen Kontakt es sich handelt. Dazu muss jeder Kontakt mittels einer eindeutigen 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.

Um eine Aussage darüber zu treffen, welches der untersuchten Technologien besser für die Entwicklung einer offlinefähigen Anwendung geeignet ist und warum, ist eine Testumgebung mitzuentwickeln. Hier soll die Möglichkeit geschaffen werden, Konflikte durch gleichzeitiges Bearbeiten eines Kontakts im Offlinestatus herbeizuführen und die Testfälle auswerten zu können.
Es wird ermittelt welche Strategie zur Konfliktlösung von der Technologie verwendet wird, ob die Funktionalität der Anwendung auch bei auftretenden Konflikten gewährleistet ist und ob dabei Daten verloren gehen. Dabei wird auch der Aufwand betrachtet, der aufgebracht werden muss die Technologie zu verwenden und wie leicht der geschriebene Quellcode zu verstehen ist.
Dinge, die von vornherein ausgeschlossen werden:
Nur auf 2 Geräten testen
perfektes UI
Kein Merge von Konfliktversionen. Nur entweder, oder.
Implementierungen von Push notifications bei Redux

Die in Abschnitt 4.2 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 sind deshalb unproblematisch zuzuordnen.

Die oben erarbeiteten Konfliktszenarien beschreiben die Konflikte die nicht vom System gelöst werden können. Diese sollen effizient gespeichert werden. 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.

5.2 Funktionale Anforderungen

Die folgende Tabelle zählt die funktionalen Anforderungen an einen Prototypen auf, der im Rahmen dieser Arbeit zur Untersuchung der Konfliktmanagementstrategien ausgewählter offlinefähiger Systeme entwickelt wird.

ID	Funktionale Anforderungen
F1	Die Anwendung muss unabhängig vom Netzwerkstatus und auf mindestens zwei Geräten funktionieren.
F2	Die Anwendung soll fähig sein den Netzwerkstatus zu ändern.
F3	Die Anwendung muss den Netzwerkstatus erkenntlich machen.
F4	Die Anwendung muss fähig sein die Kontakte unabhängig vom Netzwerkstatus zu laden, sofern diese einmal aus dem Netzwerk geladen wurden.
F5	Die Anwendung muss die Möglichkeit bieten, unabhängig vom Netzwerkstatus einen Kontakt anzulegen, zu bearbeiten oder zu löschen.
F6	Die Anwendung muss alle Kontakte sowohl lokal als auch serverseitig persisitieren, identifizieren können und versionieren.
F7	Die Anwendung muss die lokal gespeicherten Kontakte mit denen auf der Serverdatenbank persistierten synchronisieren, sobald die Anwendung mit dem Internet verbunden ist.
F8	Die Anwendung muss die Möglichkeit bieten die Konfliktmanagementstrategien der zu untersuchenden Technologien zu evaluieren.
F9	Die Anwendung soll Konflikte speichern, sofern diese auftreten.
F10	Die Anwendung muss die Möglichkeit bieten die Konflikte zu lösen, sofern diese auftreten.
F11	Die Anwendung muss sicherstellen, dass auf keinen Fall Daten verloren gehen.

Tabelle 5.1: *Funktionale Anforderungen*

5.3 User-Stories

Aus den in Kapitel 4 erarbeiteten Szenarien ergeben sich die folgenden User-Stories, die von der offlinefähigen Adressbuchanwendung erfüllt werden sollen. Ein Ziel der Arbeit ist es, EntwicklerInnen bei der Wahl einer Technologie mit der eine offlinefähige Anwendung entwickelt werden kann, zu unterstützen. Dafür wird untersucht, inwieweit die ausgewählten Technologien die Erwartungen an diese erfüllen. Deswegen werden zunächst die Anforderungen aus Sicht der NutzerInnen definiert, danach die aus Entwicklingsperspektive und dann die aus der Perspektive der TesterInnen.

5.3.1 NutzerInnen Perspektive

Die folgende Tabelle zeigt die Software-Anforderungen an eine offlinefähige Kontaktliste aus der NutzerInnenperspektive.

ID	Anforderung aus NutzerInnenperspektive
U1	Ich als NutzerIn möchte die Anwendung immer und überall, auch ohne Internetzugang zu nutzen.
U2	Ich als NutzerIn möchte, dass die Kontaktliste schnell und effizient geladen wird, um Zeit zu sparen.
U3	Ich als NutzerIn möchte Einträge immer und überall, auch ohne Internetzugang, erstellen, editieren oder löschen können, um meine Liste zu verwalten.
U4	Ich als NutzerIn möchte keine in der Anwendung gespeicherten Daten verlieren.

Tabelle 5.2: *Anforderungen aus NutzerInnenperspektive*

5.3.2 EntwicklerInnen Perspektive

Die folgende Tabelle zeigt die Software-Anforderungen an eine offlinefähige Kontaktliste aus der Perspektive der EntwicklerInnen.

ID	Anforderung aus Entwicklungsperspektive
D1	Ich als EntwicklerIn möchte die Daten lokal und auf dem Server speichern, um deren Erreichbarkeit unabhängig vom Internetstatus zu gewährleisten.

D2	Ich als EntwicklerIn möchte nur die Adressbucheinträge oder deren Aktualisierungen laden, die sich nicht schon auf dem Endgerät befinden, um Datentraffic und Ladezeiten zu sparen.
D3	Ich als EntwicklerIn möchte jeden Eintrag identifizieren, um jedem Adressbucheintrag Operationen zuzuweisen und einzelne Kontakte zu finden.
D4	Ich als EntwicklerIn möchte, dass jeden Eintrag versioniert ist, um zu wissen ob wann ein Eintrag bearbeitet wurde.
D5	Ich als EntwicklerIn möchte, dass alle von NutzerInnen vorgenommenen Änderungen beim System ankommen und keine Daten verloren gehen.
D6	Ich als EntwicklerIn möchte auftretende Konflikte effizient speichern, um mit ihnen umgehen können. Mit ihnen umgehen heißt: selbstständig oder von User lösen, zum konfliktfreien Zustand gelangen
D7	Ich als EntwicklerIn möchte eine Technologie verwenden die leicht zu verstehen und implemenieren ist, um den Arbeitsaufwand gering zu halten.
D8	Ich als EntwicklerIn möchte sauberen und verständlichen Code schreiben, um die Les– und Wartbarkeit zu erhöhen.

Tabelle 5.3: *Anforderungen aus Entwicklungsperspektive*

5.3.3 TesterInnen Perspektive

Die folgende Tabelle zeigt die Software-Anforderungen an eine offlinefähige Kontaktliste aus Perspektive der TesterInnen.

ID	Anforderung aus TesterInnenperspektive
T1	Ich als TesterIn möchte sicherstellen, dass der Netzwerkstatus der Anwendung änderbar ist, um zwischen offline und online zu wechseln.
T2	Ich als TesterIn möchte wissen, ob die Anwendung mit dem Internet verbunden ist oder nicht.
T3	Ich als TesterIn möchte die Anwendung auf mindestens zwei Geräten verwenden, um Kontakte gleichzeitig zu bearbeiten zu können.
T4	Ich als TesterIn möchte Konflikte forcieren, um das Verhalten der Anwendung zu evaluieren.

T5	Ich als TesterIn möchte einen Eintrag editieren *können* wenn 1. beide Client und Server online sind, 2. entweder Client oder Server offline ist oder 3. beide Parteien offine sind. daraus 3 Stories machen? oder nur testen online–offline?
T6	Ich als TesterIn möchte die Testfälle detailliert dokumentieren, um sie auswerten zu kpönnen.

Tabelle 5.4: *Anforderungen aus TesterInnenperspektive*

5.4 Bedienoberfläche

Da der Schwerpunkt dieser Arbeit auf dem Umgang mit Konflikten der zu testenden offlineunterstützenden Technologien liegt, soll die Bedienoberfläche der Anwendung möglichst einfach gehalten werden.
Alle Adressbucheinträge sollen in einer Liste angezeigt werden. Zum Anlegen, Editieren und Löschen eines einzelnen Eintrags soll es eine zweite Ansicht geben, auf die man per Klick auf den entsprechenden Eintrag in der Liste gelangt. Wenn es zum Konflikt kommt, kann dieser über ein Dialogfenster von den NutzerInnen aufgelöst werden. Im Dialog muss erkennbar sein bei welchem Kontakteintrag der Konflikt auftrat. Außerdem müssen sich die entsprechenden Bereiche beider Versionen unterscheiden lassen und auswählbar sein. Abbildung 5.1 zeigt wie so ein Dialogfenster aussehen könnte.

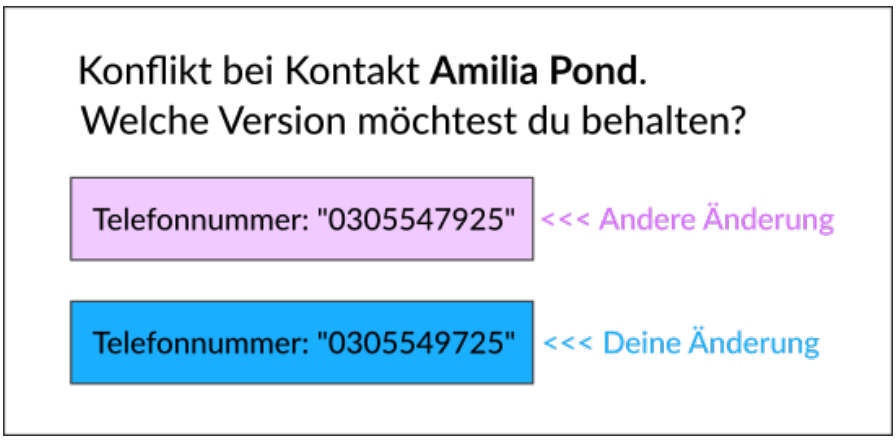


Abbildung 5.1: *Dialogfenster im Konfliktfall*

Wurde beispielsweise Amilias Telefonnummer von zwei Personen gleichzeitig bearbeitet, bildet der Dialog zwei Bereiche mit der Nummer in den unterschiedlichen Versionen

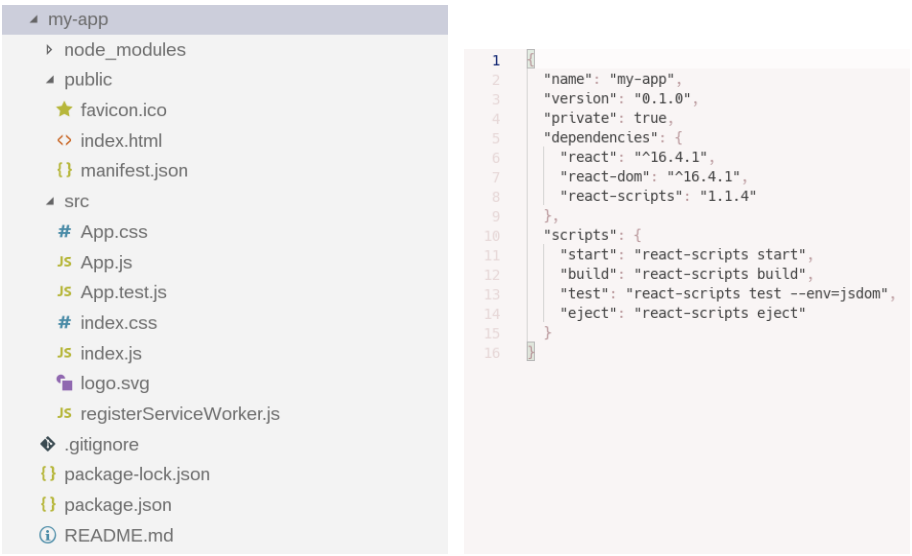
ab. Daneben ist abzulesen woher die Änderung kommt. Die lokal gespeicherte Version kann mit 'Deine Änderung' betitelt werden, die Version die vom Server kommt mit 'Andere Änderung'. Durch einen Klick auf die Knopf mit der Telefonummer kann entschieden werden welche Version die richtige ist und behalten wird.

6 Konzeption

Die erarbeiteten Anforderungen zur Untersuchung der Konfliktmanagementstrategien offlinefähiger Systeme werden in diesem Kapitel für die Konzeption angewendet.
Es soll für jede zu untersuchende Technologie ein Prototyp entwickelt werden. Im Rahmen dieser Arbeit entsteht ein Prototyp der Redux Offline verwendet und ein zweiter in dem PouchDB und CouchDB eingesetzt wird. Für letzteren könnte genauso gut HOODIE benutzt werden, da HOODIE sowohl PouchDB als auch CouchDB benutzt [hoob]. Doch da für den zu entwickelnden Prototyp lediglich diese beiden Komponenten benötigt werden, wurde sich dagegen entschieden. Bis zu einem gewissen Status, nämlich dem der Verwendung der Technologien, sind beide Prototypen – bis auf den Namen– identisch.

6.1 Anwendungsaufbau

Die Prototypen bestehen im Frontend aus React und wurden mit Create React App erstellt. Create React App erstellt ein Projekt mit dem gewünschten Namen, generiert eine initiale Projektstruktur (vgl. Abbildung 6.1a) und installiert die dafür benötigten Abhängigkeiten [Soua].



(a) *Die initiale Projektstruktur* (b) *Die initiale package.json Datei*

Abbildung 6.1: *einer mit Create React App erstellten Testapplikation*

Diese sind im Verzeichnis node_modules installiert. Außerdem ist ein ServiceWorker enthalten der die Assets cacht. Als Template gibt es nun die `public/index.html`- Datei. In der `index.js`-Datei werden die React-Komponenten und der ServiceWorker initialisiert. Alle `App.*`-Dateien umfassen eine minimale Beispielanwendung. In der generierten `package.json`-Datei (vgl. Abbildung 6.1b), befinden sich Informationen über die Anwendung und ihre Abhängigkeiten. Im Unterpunkt `scripts` werden Kommandozeilen-Aufrufe definiert und können mit dem Befehl `npm run` aufgerufen werden.

6.1.1 Aufbau der React Komponenten

React ist eine Open-Source Bibliothek, die dazu dient, die View-Komponente des Model-View-Controller-Ansatzes abzudecken, also die Seite der Anwendung die für die Anzeige und Interaktion zuständig ist. Ein Vorteil von React sind die Komponentenbasierte Philosopie. Eine Komponente ermöglicht die Aufteilung der User Interface (UI) in kleine Teile und ist eine abstrakte Basisklasse. Einmal implementiert, lässt sich eine Komponente immer wieder verwenden [Soub].
Eine Komponente kann einen internen `state` besitzen, oder die Daten aus den `props` nehmen. `Props` sind Daten die von der Elternkomponente übergeben werden und können auch nur von dieser manipuliert werden. Eine React Komponente hat immer eine `render()`-Funktion die die Daten aus dem `state` oder den `props` liest und zurückgibt was dargestellt werden soll. Hier wird das zur Komponente gehörende HTML erzeugt. Jede Änderung des `states` führt einen erneuten Aufruf der `render()`-Funktion mit sich.

React Komponenten können in zwei Kategorien aufgeteilt werden. Die eine Kategorie ist die Containerkomponente. Sie dienen als Datenquelle und in ihnen steckt die Logik wie etwas funktioniert. Sie stellen außerdem Callbackfunktionen für die Viewkomponenten bereit. Viewkomponenten haben keine andere Zuständigkeit als die Daten, die sie über ihre `props` erhalten, anzuzeigen und ggf. die ebenfalls empfangenden Callbackfunktionen aufzurufen [Abr].
Zur Veranschaulichung wird anhand des Listings 6.1 eine beispielhafte React Containerkomponente, und im Listing 6.2 die dazugehörige Viewkomponente beschrieben. Zusammen repräsentieren sie ein Formular in dem der Name des Kontakts angezeigt und geändert werden kann.

```
1 class FormContainer extends Component {
2   constructor (props) {
3     super(props)
4     this.state = {
5       contact: props.contact
6     }
```

```
 7   }
 8
 9   handleChange (event) {
10     let contact = this.state.contact
11     contact = {...contact, name: event.target.value}
12     this.setState(() => ({contact: contact}))
13   }
14
15   render () {
16     return (
17       <ContactForm
18         handleChange={this.handleChange}
19         contact={this.state.contact} />
20     )
21   }
22 }
23
24 const ContactForm = ({contact, handleChange}) => (
```

Listing 6.1: *Eine React Containerkomponente*

Die Formular–Containerkomponente hat ein Kontaktobjekt im internen `state` gespeichert. Auf dieses Objekt haben andere Komponenten keinen Zugriff und es ist nur via `setState()` änderbar. Initial wird das Kontaktobjekt über die `props` in Zeile fünf geladen. So kann das Vorausfüllen der Eingabefelder realisiert werden.
In neun sieben steht die `handleChange` Funktion, die den internen `state` in Zeile zwölf mit den reingegebenen Werten aktualisiert. Die Viewkomponente wird wie in in den Zeilen 17 bis 19 aufgerufen. Ihr wird der interne State übergeben, also das Kontaktobjekt und die `handleChange` Funktion.

```
1   <form className='contact-form'>
2     <label>Name</label>
3     <input
4       value={contact.name}
5       onChange={this.handleChange.bind(this)} />
6   </form>
7 )
```

Listing 6.2: *Eine React Viewkomponente*

Die beiden Parameter sind in der Zeile eins des folgenden Listings wiederzufinden. Die Viewkomponente macht nichts anderes als die ihr übergebenen Daten anzuzeigen. Im

Ereignisbehandler des Eingabefeldes wird die übergebene `handleChange` Funktion, in der Zeile fünf, aufgerufen. Diese Komponente besitzt keinerlei Logik.

6.1.2 Verwendung von Redux Offline

Redux Offline kann nur zusammen mit Redux verwendet werden. Deswegen ist für diesen Prototypen die Implementierung von Redux vorausgesetzt. Redux ist eine JavaScript Bibliothek die Probleme im Zusammenhang mit dem *Zustand* einer Anwendung löst. Es gibt einen zentralen Ort, in dem der *Zustand* der App gespeichert ist, auf den von jeder Komponente aus zugegriffen werden kann. Dieser Ort wird Store genannt. Alle Änderungen der Daten im zentralen Speicher erfolgen ausschließlich über Aktionen.

Redux Offline ist eine erweiternde Bibliothek für Redux dessen Funktionsweise in Abschnitt 3.2 detailliert beschrieben wird.
Nach der Installation muss der Redux `store` zusammen mit dem `offline store enhancer` erzeugt werden. Listing 6.3 visualisiert diesen Vorgang. Ein Redux `store` wird mit dem `storeCreator` in Zeile fünf erzeugt. Ein `store enhancer` ist eine Funktion die den `storeCreator` neu zusammenfügt und einen neuen, erweiterten `storeCreator` zurückgibt. Redux Offline kommt mit einer Grundkonfiguration (siehe Zeile drei). Diese wird dem `offline store enhancer` in Zeile acht übergeben.

```
1
2 const store = createStore(
3   reducer,
4   compose(
5     offline(offlineConfig)
6   )
7 )
```

Listing 6.3: *Erstellen eines Stores mit Redux Offline*

Der gesamte Kontext, der zum Synchronisieren einer Aktion erforderlich ist in einem zusätzlichen Metaattribut gespeichert. Damit die Anwendung weiß wie die Aktionen verarbeitet werden sollen wird sie mit dem Metafeld dekoriert. Die Aktion zum Lesen der Kontakte könnte dann wie im folgenden Listing aussehen.

```
1 export function readContacts () {
2   return {
3     type: FETCH_CONTACTS,
4     meta: {
```

```
5         offline: {
6           effect: { url: `${API}/contacts` },
7           commit: { type: 'FETCH_CONTACTS_COMMIT' },
8           rollback: { type: 'FETCH_CONTACTS_ROLLBACK' }
9         }
10      }
11    }
12  }
```

Listing 6.4: *Aktion `fetchContacts` mit Metaattribut*

Das erste `meta.offline` Feld beschreibt die Netzwerkaktion die ausgeführt werden soll, also den Aufruf an die angegebene URL in Zeile sechs. Bei `commit` in Zeile sieben wird festgelegt welche Aktion bei erfolgreichem Netzwerkaufruf augeführt werden soll. Für den Fall, dass von dem angefragtem API ein 4xx oder 5xx HTTP Statuscode zurückkommt wird die im `rollback` definierte Aktion gefeuert [rede].
Die Aktionen beschreiben nur was passiert. Wie der Status sich ändert, wird im `Reducer` beschrieben. Das Listing 6.5 illustriert wie das im entsprechendem Prototypen umgesetzt werden könnte.

```
1  function contacts (state=[], action) {
2    switch (action.type) {
3      case FETCH_CONTACTS:
4        return state
5
6      case FETCH_CONTACTS_COMMIT:
7        if (state === action.payload) return state
8        return action.payload
9
10     case FETCH_CONTACTS_ROLLBACK:
11       return state
12   }
13 }
14
15 import { createStore, compose } from 'redux'
```

Listing 6.5: *Reducer mit allen Aktionen die im Meta Feld beschrieben werden*

In diesem Beispiel wird der Appstatus nur bei erfolgreichem Netzwerkaufruf aktualisiert. Das ist in den Zeilen sechs bis acht nachzulesen. Und auch nur dann, wenn sich die Antwort vom Server von diesem unterscheidet.

6.2 Architektur

Die zu erstellenden Prototypen erhalten die Namen *amilia-qouch* und *amilia-rdx*, wobei Amilia der Name ist, der sich in den Beispielkontakten in den Szenarien wiederfindet. Die Abkürzung *rdx* steht für Redux und zeigt, dass dieser Prototyp Redux Offline verwendet. Die Endung `qouch` soll die Symbiose von CouchDB und PouchDB darstellen. Der Buchstabe Q klingt wie das hart ausgesprochene C in Couch und wenn man das kleine Q horizontal spiegelt, sieht man das P für Pouch.

Beide Prototypen setzen sich aus den nachfolgend beschriebenen Komponenten zusammen, welche in Abbildung 6.2 veranschaulicht werden. Die Abbildung stellt ein Komponentendiagramm dar. Es handelt sich hierbei nicht um das UML Komponentendiagramm, sondern um ein eigens entworfenes. Die Bezeichnung ist durch die Darstellung von React Komponenten begründet.

Jeder Kasten repräsentiert eine Komponente, deren Bezeichnung im Kopf steht. Alle Komponenten auf der linken Seite sind View–Komponenten und können den Appstatus nicht manipulieren. Sie können nur die von der Elternkomponente durchgereichten Funktionen aufrufen. Anhand der Linien ist abzulesen auf welche Funktionen und Eigenschaften die View–Komponenten Zugriff haben. Es werden für jede Komponente nur die eigens implementierten Funktionen aufgelistet.
Auf der rechten Seite stehen die Model–Komponenten. Sie manipulieren den Appstatus und entscheiden welche View–Komponente gerendert wird. Die Komponente `Contacts` fungiert als Container und ist das Herzstück der Anwendung. Er definiert die graphische Oberfläche und stellt alle notwendigen Funktionen bereit. Sie hat einen internen `state` in dem sowohl die Kontaktliste, als auch die Daten für das Formular gespeichert sind. Im Diagramm ist der `state` an der blauen Schrift zu erkennen. Das Objekt `editView` zeigt, welche Ansicht – Liste oder Formular – gerade aktuell ist und speichert den im Formular zu ladenden Kontakt. Wie das Kontaktobjekt aufgebaut ist zeigt der blaue Kasten im Diagramm. Wird eine Aktion zum Ändern der Ansicht aufgerufen, beispielsweise durch das Betätigen eines Knopfes, wird über die Funktion `toggleEdit()` der interne `state` aktualisiert und ein erneutes Rendern der Komponente eingeleitet. Dann wird entsprechend die Liste oder das Formular gerendert.
Der `Header` implementiert das externe Modul `react-detect-offline` und kann so den Netzwerkstatus anzeigen. Die Komponente hat Zugriff auf die `toggleEdit`–Funktion und einen Knopf, der an diese gebunden ist. Damit kann das Rendern des Kontaktformulars eingeleitet werden. Dieser Knopf soll nur angezeigt werden, wenn die Liste aktiv ist. Diese Information ist in dem Attribut `isOpen` abzulesen.
Die `ContactList` repräsentiert die Kontaktliste und wird initial gerendert. Hier werden

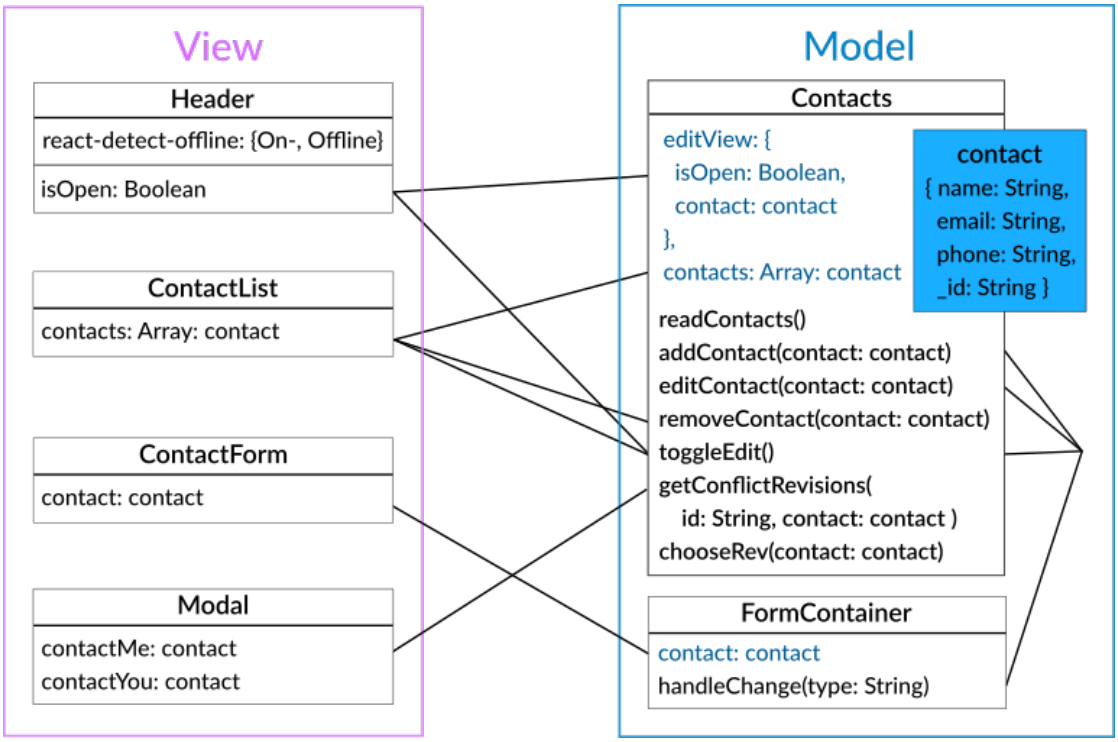


Abbildung 6.2: *Komponentendiagramm der Prototypen*

alle Kontakte als Liste dargestellt. Es kann das Bearbeiten, durch den Aufruf der durchgereichten Funktion `toggleEdit()` eingeleitet werden. Durch den Aufruf von `removeContact()` wird der Kontakt gelöscht.
Die Komponente `ContactForm` zeigt, sofern vorhanden, alle im Kontakt gespeicherten Daten an. Gibt es keine Kontaktdaten die geladen werden können, kann ein neuer Kontakt angelegt werden. In der View, dem `ContactForm` gibt es zum Hinzufügen und Bearbeiten des Kontakts Eingabefelder für den Namen, die E-Mail und die Telefonnummer des Kontakts. Zusätzlich zu den Eingabefeldern für jedes Kontaktattribut hat sie zwei Knöpfe mit denen die Aktion bestätigt oder abgebrochen werden kann.
Der dazugehörige Ereignisbehandler befindet sich im `FormContainer`. Der "lauscht' auf die Veränderung der einzelnen Eingabefelder und speichert alle Änderungen in einer Kopie des Kontakts im internen `state`'. Wird die Aktion durch den entsprechenden Knopf bestätigt, wird die Kopie des Kontakts an `Contacts` gegeben und dort entweder via `addContact()` hinzugefügt, oder via `editContact()` bearbeitet. Wenn die Aktion abgebrochen wird, ruft der `FormContainer` die `toggleEdit()` Funktion in der Hauptkomponente auf.

Kommt es beim Hinzufügen, Bearbeiten oder Löschen eines Kontakts zum Konflikt, werden mittels `getConflictRevisions()` die beiden Versionen des konfliktbehafteten

Kontakts ermittelt. Der Kontakt wird dann in beiden Varianten, der alten und der neueren Version, an einen Konfiktdialog übergeben, der sich prompt öffnet. Dieser hat Zugriff auf die Funktion `chooseRev` in der `Contacts` Komponente. Der Dialog hat pro Kontakt einen Knopf. Jeder dieser Knöpfe ruft auf Klick die `ChooseRev` Funktion mit dem ausgewähltem Kontakt als Parameter auf. Innerhalb von `chooseRev` wird der übergebene Kontakt gespeichert und die konkurrierende Version verworfen.

Eine Backendimplementierung ist für den Prototypen *amilia-qouch* nicht notwendig, da dies bereits durch die Verwendung von CouchDB gegeben ist. Die Abbildung 6.3 skizziert die Architektur von *amilia-qouch*. Die Contacts Komponente fungiert als Model. Sie handhabt die Daten die in den View Komponenten angezeigt werden und speichert sie in der PouchDB.Die PouchDB synchronisiert sich mit der CouchDB und schreibt die Daten zurück in das Model.

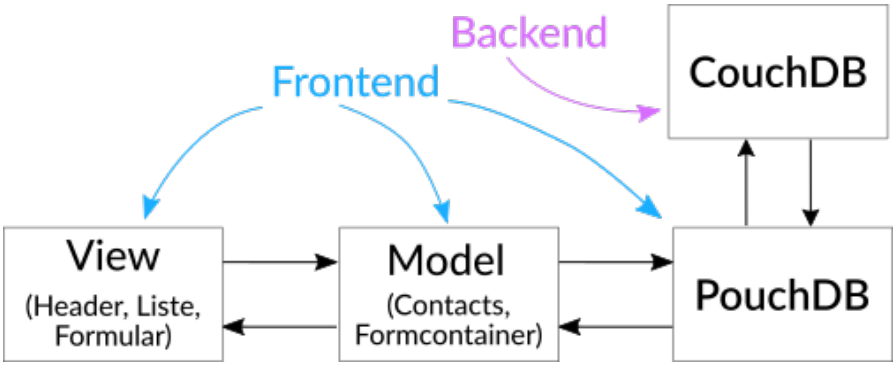


Abbildung 6.3: *Client-Server-Modell des Prototypen amilia-qouch*

Obwohl Redux Offline nach eigener Aussage die Datenbank ersetzt [Evä17], stellt es keine Serverdatenbank zur Verfügung. Deswegen wird ein Node Server erstellt der alle CRUD Operationen unterstützt. Die Kontakte werden in einer JSON Datei persistiert. Die Abbildung 6.4 zeigt sehr gut, dass der Redux Store in *amilia-rdx* die PouchDB *amilia-pouch* ersetzt. Dabei handelt es sich um den Redux Store, der dank Redux Offline alle Daten in einer lokalen Datenbank speichert. Die im *amilia-qouch* verwendete CouchDB wird von dem Server und einer JSON Datei ersetzt. Der Datenfluss ist derselbe wie im Model des Prototyps *amilia-qouch* mit dem Unterschied, dass der Store und der Server sich nicht automatisch synchronisieren. Der Store schickt zwar automatisch die Daten an den Server sobald er mit dem Internet verbunden ist, aber die andere Richtung ist ist das nicht implementiert. Deswegen werden nach jeder Aktion, die Kontakte vom Server geladen.

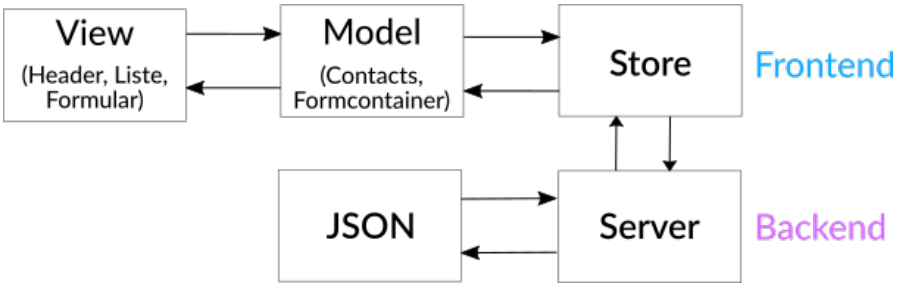


Abbildung 6.4: *Client-Server-Modell des Prototypen amilia-rdx*

6.2.1 Das Speichern der Daten

Das Seichern von Kontakten wird in den Prototypen unterschiedlich implementiert.

Daten mit PochDB und CouchDB speichern

Für den Protoyp *amilia-qouch* muss zunächst CouchDB installiert werden. Sobald dieser Schritt erledigt ist läuft CouchDB auf `localhost:5984` und ist einsatzbereit. Das asynchrone API von PouchDB stellt alle notwendigen Funktionen bereit die sowohl Callbacks, Promises als auch asynchrone Funktionen unterstützen. Das Listing 6.6 führt alle benötigeten Funktionen auf und zeigt die notwendigen Schritte zur Synchronisation der lokalen PouchDB und CouchDB.
Die lokale Datenbank wird in Zeile zwei erstellt. Wenn es die Datenbank mit dem Namen ‘contacts‘ bereits gibt, wird sie gestartet.
Um eine CouchDB Instanz zu erzeugen ist der Aufruf in Zeile drei mit der URL zur CouchDB–Datenbank notwendig. Auch hier erstellt PouchDB die Datenbank, sofern sie noch nicht existiert. PouchDB funktioniert nun als Client zu einer online CouchDB Instanz. Zur kontinuierlichen Synchronisation beider Instanzen, der lokalen PouchDB und der CouchDB, ist lediglich der Aufruf in Zeile fünf erforderlich. Im optionalen Parameter können zum Beispiel FIlter oder Einstellungen zum wiederholten Synchronisationsversuch im Falle eines Fehlschlags gesetzt werden [poud].
Der Aufruf `localDB.allDocs()` in Zeile acht fragt alle in der lokalen Datenbank gespeicherten Kontakte an. Ohne den Parameter `include_docs: true` werden nur die ID, `_id` und die Revisionsnummer, `_rev` eines jeden gespeicherten Kontakts zurückgegeben. Ist die Option `conflicts` auf `true` gesetzt, werden Konfliktinformationen zu jedem Kontakt gespeichert. Alle konfliktbehafteten Kontakte haben nun ein `_conflicts` Attribut. Dort bedindet sich eine Liste mit allen konkurrierenden Revisionen.

Für die nachfolgenden Aufrufe gibt es mehrere Varianten. Eine Anleitung zu dem Umgang mit Konflikten in der PouchDB Dokumentation empfiehlt für das Erstellen, Aktualisieren

und Löschen eines Dokuments das Modul PouchDB Upsert zu verwenden. Das wird in Zeile eins zu, PouchDB Objekt hinzugefügt. Jedes Mal wenn eine der CRUD Operationen auf einem Dokument ausgeführt wird und das API eine Revision verlangt, kann ein Konfliktfehler geworfen werden. Zum praktischen Umgang empflieht PouchDB die Aktion zu wiederholen, bis sie erfolgreich war. Dazu kann PouchDB Upsert verwendet werden. Es fügt ein neues Dokument hinzu wenn noch keins existiert und aktualisiert es, wenn es vorhanden ist [pouc]. Konflikte treten so trotzdem auf und CouchDB wählt eine gewinnende Version aus. Alle Konflikte werden immenrnoch gespeichert und die beteiligten Dokumente können wie oben beschrieben aus der CouchDB geladen werden.

```
1  PouchDB.plugin(require('pouchdb-upsert'))
2  const localDB = new PouchDB('contacts')
3  const remoteDB = new PouchDB('http://localhost:5984/contacts')
4
5  localDB.sync(remoteDB, [options])
6
7  // get all contacts
8  localDB.allDocs({
9    include_docs: true,
10   conflicts: true
11 })
12
13 // create a contact
14 const id = new Date().toISOString()
15 await localDB.upsert(id, function () {
16   return contact
17 })
18
19 // update a contact
20 await localDB.upsert(contact._id, function (doc) {
21   doc = {...contact}
22   return doc
23 })
24
25 // remove a contact
26 await localDB.upsert(contact._id, function (doc) {
27   doc._deleted = true
28   return doc
29 })
```

Listing 6.6: *Persistierung der Daten mit PouchDB und CouchDB*

Die Zeilen 13 bis 16 zeigen wie ein Kontakt erzeugt werden kann. Bevor das geschieht wird eine ID erstellt die aus dem aktuellen Zeitstempel besteht. Diese wird dann per

PouchDB Upsert in dem neuen Dokument gespeichert.
Das Aktualisieren eines Kontakts sieht sehr ähnlich aus, da die `upsert` Funktion für beide Operaktionen verwendet wird. Mit jedem Update bekommt ein Kontakt von PouchDB eine neue Revision.
Man kann einen Kontakt mit PouchDB Upsert wie in Zeile 35 löschen. Der Kontakt nicht wirklich gelöscht sondern wird durch ein `_deleted` Attribut als solches markiert.

Daten mit Redux Offline speichern

Die Idee hinter Redux Offline ist, dass der Redux Store die lokale Datenbank ersetzt. Sobald der Appstatus sich ändert, also irgendwo im Code `setState()` ausgeführt wird, wird er automatisch lokal gespeichert. Dazu wird intern `redux-persist` benutzt, dessen Funktionsweise in Abschnitt 3.2.3 erläutert wird. Der Redux Store wird bei jeder Änderung persistiert und beim Start der Anwendung neu geladen. Wie die Daten mit Redux Offline gespeichert synchronisiert werden ist am folgenden Listing erklärt. Das Beispiel erklärt das Hinzugügen eines Kontakts. Die restlichen Operationen, das Bearbeiten, Löschen oder Lesen eines Kontakts sind analog dazu.

```
1  // action creator
2  export function addContact () {
3    return {
4      type: ADD_CONTACT,
5      contact,
6      meta: {
7        offline: {
8          effect: {
9            url: `${API}/contacts`, method: 'POST',
10           body: JSON.stringify({contact})
11         },
12         commit: { type: 'ADD_CONTACT_COMMIT', meta: { contact } },
13         rollback: { type: 'ADD_CONTACT_ROLLBACK', meta: { contact } }
14       }
15     }
16   }
17 }
18
19 // reducer
20 function contacts (state=[], action) {
21   switch (action.type) {
22     case ADD_CONTACT:
23       console.log('started to add contact')
24       return [...state, action.contact]
25
26     case ADD_CONTACT_COMMIT:
```

```
27        console.log('successfully added contact')
28        return [...state, action.payload]
29
30        case ADD_CONTACT_ROLLBACK:
31        console.log('failed to add contact')
32        return state
33    }
34 }
```

Listing 6.7: *Speicherung der Daten mit Redux Offline*

Mit dem Aufruf der Aktion ADD_CONTACT, die in Zeile drei bis 15 erzeugt wird, wird der Vorgang einen Kontakt hinzuzufügen gestartet. Die anzufragende URL ist im `effect` Feld des `meta.offline` Objekts in Zeile neun festgelegt. Die Anfrage geht an den Server, welcher die Daten in der JSON Datei persisitiert. Der Reducer hat Zugriff auf das Aktionsobjekt. Dort ist der gerade hinzugefügte Kontakt gespeichert. Mit diesem wird in Zeile 23 der Appstate aktualisiert und so lokal gespeichert.
Ist die Netzwerkanfrage erfolgreich, wird die Aktion `commit` in Zeile zwölf gefeuert. Die wird im Reducer in Zeile 26 behandelt. Der `state` wird mit der Antwort vom Server aktualisiert und die Synchronisation ist vollzogen. Für den Fall dass die Netzwerkanfrage nicht erfolgreich war wird das `meta.offline.rollback` Feld definiert. Hier könnte man die BenutzerInnen darüber informieren, dass etwas nicht geklappt hat. Für den Rahmen dieser Arbeit genügt es jedoch, wie es in den Zeilen zehn und elf steht, den vorherigen Status zurückzuliefern.

Wie die Serverimplementierung für den gerade beschriebenen Fall aussehen könnte, beschreibt das Listing 6.8.

```
1 const fs = require('fs')
2 const contacts = require('./contacts.json')
3
4 app.post('/contacts', function (req, res) {
5   let contact = req.body.contact
6   if (!contact || typeof contact === 'undefined') {
7     res.status(400).send({ msg: 'contact malformed.' })
8   } else {
9       // persist contact
10      contacts.push(contact)
11      fs.writeFile('utils/contacts.json', contacts, 'utf8', function (err) {
12        if (err) {
13          res.status(500).send({msg: 'failed to write file'})
14          throw err
15        }
```

```
16          res.json(contact).send()
17        })
18    }
19 })
```

Listing 6.8: *Mögliche Serverimplementierung für das Hinzufügen eines Kontakts*

In Zeile zwei werden die Kontakte aus einer JSON Datei geladen. Diese sind als Objekt in einem Array gespeichert. Bekommt der Server eine post–Anfrage, wird ein Kontaktobjekt mitgesendet. Dieser wird in Zeile fünf in einer Variable zwischengespeichert. Wird der Kontakt korrekt gesendet wird er in Zeile zehn dem Array hinzugefügt. Andererseits sendet der Server den HTTP–Statuscode 400 an den Client. Außerdem wird mithilfe des in Node integriertem Dateisystem[17] Moduls die JSON Datei neu geschrieben. Nun ist der neue Kontakt persistiert. Kommt es beim Schreiben der Datei zu keinem Fehler, sendet der Server den frisch gespeicherten Kontakt zurück an den Client.

6.2.2 Verbindungsstatus feststellen und ändern

Für die Überprüfung der Verbindung zum Server wird das Modus React Detect Offline verwendet. Es beobachtet den Online– und Offlinestatus und bietet zwei Komponenten entsprechend des Status den Inhalt rendern. Der folgende Codeausschnitt zeigt eine Verwendung dieser beiden Komponenten. Ist die Anwendung online, wird ‘you are online‘ gerendert. Im anderen Fall ‘you are offline‘.

```
<Online>
  <span className='green'>you are online</span>
</Online>
<Offline>
  <span className='red'>you are offline</span>
</Offline>
```

Listing 6.9: *Beispiel einer React Detect Offline Implementierung*

Das Modul verfolgt kontinuierlich den Onlinestatus des Browsers indem es auf die Online- und Offlineereignisse der Webspezifikation reagiert. Zusätzlich fragt es alle fünf Sekunden die URL `https://ipv4.icanhazip.com` ab und rendert je nach Verbindungsstatus die entsprechende Komponente. Verschiedene Parameter wie die URL oder das Intervall in dem die URL aufgerufen wird können konfiguriert werden [Bol].

[17] siehe hierzu: `https://nodejs.org/api/fs.html`

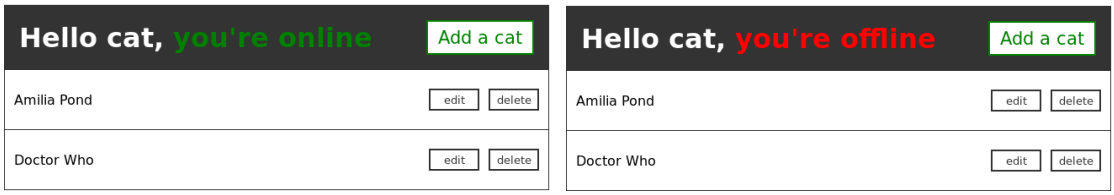
Der Verbindungsstatus eines Geräts kann im Browser geändert werden. Die Prototypen, die im Rahmen dieser Arbeit entwickelt werden, sollen in den Browsern Firefox und Chrome laufen.
In Firefox lässt sich der Netzwerkstatus über das Einstellungsmenü ändern. Dort kann man entweder unter dem Punkt ‘Sonstiges‘ oder dem Punkt ‘Web-Entwickler‘ ‘Offline arbeiten‘ auswählen und ist vom Internet getrennt. Dieser Status lässt sich über den selben Weg rückgängig machen.
In Chrome öffnet man dazu die Entwicklertools, geht auf ‘Netzwerk‘ und klickt auf die Checkbox ‘Offline‘ am oberen Rand. Dieselbe Checkbox ist auch im ‘Application‘–Tab unter ‘Service Workers‘ zu finden.
Sollen der Offlinestatus für zwei oder mehr Geräte hergestellt werden, kann der Server oder die CouchDB gestoppt werden. Damit der Status in der Oberfläche abgelesen werden kann, muss dem React Detect Offline Modul die entsprechende URL übergeben werden.

6.3 Die graphische Oberfläche

Aus den minimalen Anforderungen an die graphische Oberfläche ergibt sich das Design. Anhand der folgenden Abbildungen werden die gefertigten Entwürfe der BenutzerInnenoberfläche dargestellt.

Diese Listenansicht in Abbildung 6.5 besteht aus dem Header / Kopf und den Listeneinträgen. Sie zeigt die Kontakteinträge in beiden Netzwerkstatus: online (Abbildung 6.5a) und offline (6.5b).

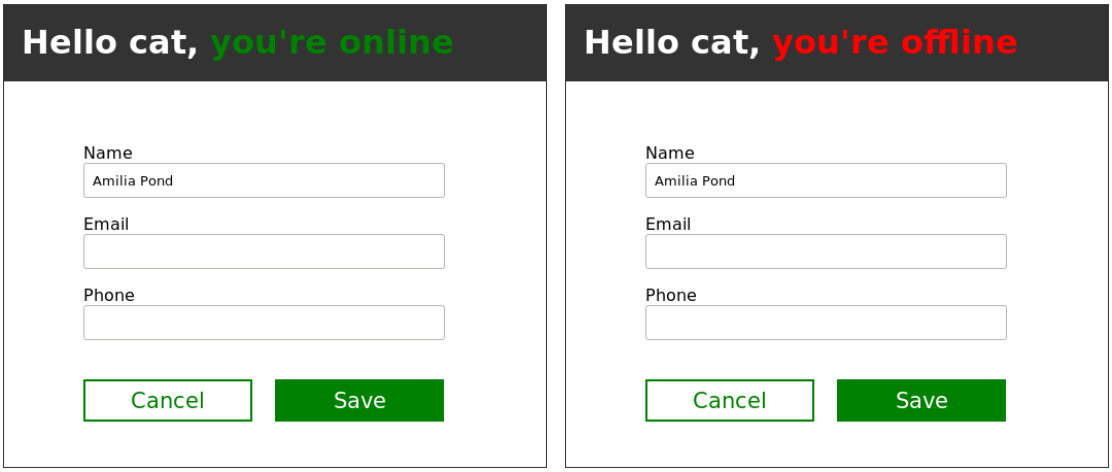
Im Header ist abzulesen ob die Anwendung gerade eine Netzwerkverbindung hat oder nicht. Für eine bessere Prägnanz wurden hierzu unterstützend die Farben Rot für keine Verbindung und Grün für eine bestehende Netzwerkverbindung gewählt. Rechts im Header gibt es einen Knopf mit dem man in die Ansicht gelangt in der ein Kontakt hinzugefügt werden kann.
In der Liste sieht man die Namen der Person und jeweils einen Knopf zum Bearbeiten oder Löschen. Mit der Betätigung des ‘Delete‘–Knopfs wird der entsprechende Eintrag in der Liste gelöscht



(a) *Kontaktliste im Onlinestatus* (b) *Kontaktliste im Offlinestatus*

Abbildung 6.5: *Die Kontaktliste in beiden Netzwerkstatus*

Klickt man auf den Knopf zum Bearbeiten oder auf den zum Hinzufügen eines Kontakts gelangt man in die Bearbeitungsansicht (vgl. Abbildung 6.6). Der Header ist bis auf den Knopf zum Hinzufügen eines Kontakts identisch zu dem der Liste. Auch hier ist abzulesen ob die Anwendung on– oder offline ist. Da man sich bereits in der Ansicht zum Anlegen oder Editieren eines Kontaks befindet, ist der Knopf im Header überflüssig.
Ein Kontakt hat einen Namen, eine E-Mailadresse und eine Telefonnummer. In dieser Ansicht gibt es für jedes Attribut ein Eingabefeld. Die Felder sind beim Bearbeiten des Kontakts vorausgefüllt. Mittels Betätigung des ‘Speichern‘ Knopfs werden die Änderungen übernommen, klickt man auf ‘Cancel‘ werden sie verworfen. In beiden Fällen gelangt man wieder zur Listenansicht.



(a) *Editieransicht im Onlinestatus* (b) *Editieransicht im Offlinestatus*

Abbildung 6.6: *Die Editieransicht in beiden Netzwerkstatus*

Sobald ein Konflikt entstanden ist soll sich ein Dialog öffnen, der den nutzenden Personen darüber informiert welcher Kontakteintrag konfliktbehaftet ist und mit welcher Version er konkurriert. Anhand des Dialoginhalts kann unterschieden werden welche Version die lokal gespeicherte ist und welche vom Server kommt. Der Dialog beinhaltet außerdem zwei unterschiedlich farbige Knöpfe die jeweils den Kontakteintrag in einer anderen Version anzeigen. Durch Klick auf einen Knopf wird die entsprechende Version des Kontakts

gespeichert und die andere verworfen. So kann ein Mensch entscheiden welche Version behalten werden soll und es wird sichergestellt dass keine Daten verloren gehen.
Für die Implementierung des Konfliktdialogs ist es notwendig, dass die zu untersuchenden Technologien die Möglichkeit bieten Konflikte zu speichern oder wenigstens als solche zu identifizieren, sodass sie manuell gespeichert werden können.

6.4 Testdurchläufe

Um das Konfliktmanagement der zu testenden Technologien untersuchen zu können werden manuelle Tests durchgeführt.
Es müssen zunächst Konflikte erstellt werden, um zu untersuchen wie die verwendeten Technologien damit umgehen. Die zu entwickelnden Prototypen müssen auf mindestens zwei Geräten funktionieren und es muss die Möglichkeit bestehen den die Verbindung zum Internet und zum Server zu unterbrechen. Eine Variante Konflikte entstehen zu lassen ist, einen Kontakt an derselben Stelle zu bearbeiten während ein oder beide Geräte offline sind. Daher ist es wichtig zu wissen, in welchem Status sich die Anwendung befindet. Die Testdurchläufe sind immer gleich und unabhängig vom Prototypen.

Zuerst wird die App in zwei Browsern gestartet. So kann die Verwendung von zwei Geräten simuliert werden. Da die Anwendung für die Browser Firefox und Chrome entwickelt wird, findet der Testdurchlauf auch in diesen beiden Browsern statt.
Es werden die Aktionen 'Kontakt anlegen', 'Kontakt bearbeiten' und 'Kontakt löschen' in unterschiedlichen Kombinationen und beiden Browsern durchgeführt. In der ersten Testreihe sind beide Browser mit dem Internet verbunden und der Server ist an. Diese Testreihe soll der Untersuchung auf Kollaborationsfähigkeit, also ob Die Anwendung auf mehreren Geräten funktioniert, dienen In der zweiten Testreihe wird ein Browser für eine Aktion vom Internet getrennt indem er in den Offlinemodus geschalten wird. Ist die Aktion in beiden Browsern abgeschlossen, wird der Offlinemodus des einen Browsers deaktiviert, sodass sich beide synchronisieren können. In der dritten Testreihe wird der Server gestoppt, sodass beide Prototypen offline sind. Nachdem eine Aktion auf beiden Geräten vollständig durchgeführt wurde, wird der Server wieder gestartet und beide Anwendungen synchronisieren sich mit dem Server, oder der CouchDB. In den letzten beiden Testreihen werden durch das Bearbeiten derselben Einträge aktiv Konflikte erzeugt. Jede Testreihe wird einmal, in interschiedlichen Netzwerkstatus der Anwendung durchgeführt, pro Reihe gibt es drei bis vier Testdurchläufe. In den Testreien eins und drei wird jeder genannte Testdurchlauf genau einmal durchgeführt. Für die zweite Testreihe kann es, je nachdem in welchem Broswer die Aktion zuerst durchgeführt wird, unterschiedliche Ergebnisse geben. Deswegen wird dort jeder Testdurchlauf zwei mal durchgeführt.

Einmal wird die Aktion in der Anwendung die mit dem Internet verbunden ist zuerst gespeichert, dann die im Offlinemodus.
Die folgende Tabelle veranschaulicht die durchzuführenden Testdurchläufe.

Nr.	Firefox online	Firefox offline	Chrome online	Chrome offline
1a	anlegen		anlegen	
1b	bearbeiten		bearbeiten	
1c	löschen		löschen	
2a	anlegen			anlegen
2b	bearbeiten			bearbeiten
2c	bearbeiten			löschen
2d	löschen			löschen
3a		anlegen		anlegen
3b		bearbeiten		bearbeiten
3c		bearbeiten		löschen
3d		löschen		löschen

Tabelle 6.1: *Testdurchläufe*

Zur Auswertung der Tests werden alle Ausgangspositionen, Vorgänge und Ergebnisse dokumentiert werden. Für einen Testdurchlauf wird hierfür der Kontakt und um welchen Testdurchlauf es sich handelt aufgeschrieben. Außerdem wird festgehalten auf welchem der beiden Geräte die Aktion zuerst durchgeführt wurde, das erwartete und das tatsächliche Ergebnis.

7 Implementierung der Prototypen

In diesem Kaptitel wird nach dem in Kapitel 6 präsentiertem Lösungsweg die detaillierte Beschreibung der technischen Realisierung der Prototypen vorgestellt.
Nach der Beschreibung der Realisierung der grundlegenden Funktionen der Anwendung wird auf die Umsetzung der Offlinefunktionalität und des Konfliktmanagements eingegangen. Im Zuge dessen wird der implementierte Algorithmus zur Lösung auftretender Konflikte vorgestellt.

7.1 Die Contacts Komponente

Die Herzstück jeder zu entwickelnden Anwendung ist die Komponente `Contacts`. Sie beinhaltet die zentralen Funktionalitäten und entscheidet welche Anzeigeelemente gerendert werden und welche nicht. Im Prototypen *amilia-qouch* besteht sie aus der Datei `Contacts.js` in der ein interne State festgelegt ist. Hier sind außerdem Funktionen implementiert über die der State manipuliert werden kann.
Durch den in Unterabschnitt 3.2.1 beschriebenen, Redux–spezifischen Datenfluss ist sie auf mehrere Dateien aufgeteilt. `Contacts.js` ist für das Rendern der anderen Komponenten zuständig. In `actions.js` werden die Aktionen definiert, die über die Containerkomponente `ContactsContainer.js` von jeder Komponente aufgerufen werden können um den Appstate zu manipulieren. Die Manipulation wird in der `reducer.js` behandelt, wo die geänderte Kopie wieder zurück an den `Store` gegeben wird und sie wieder bei der `Contacts.js` ankommt. Der Einfachheit halber wird im Folgenden von der `Contacts` Komponente geschrieben, wobei für den Redux Prototypen der Zusammenschluss dieser soeben beschriebenen Dateien gemeint ist.

Die Methode `componentDidMount` ist die dritte im React Komponenten–Lebenszyklus[18] und wird aufgerufen sobald die Anwendung gemountet ist. Hier werden werden die Kontakte geladen und im Appstate gespeichert. Sobald sicht der Appstate ändert wird die `render` Funktion der Komponente aufgerufen. Listing 7.1 zeigt die Renderfunktion der `Contacts` Komponente.

[18] `https://reactjs.org/docs/react-component.html#the-component-lifecycle` - Zugriff: 28.07.1018

```
1 render () {
2   const { contacts, editView, modalView } = this.state
3   return (
4     <div>
5       <Header
6         isOpen={editView.isOpen}
7         handleGoToEdit={this.toggleEdit.bind(this, null)} />
8
9       {modalView.hasConflict &&
10        <Modal
11          contactMe={modalView.contactMe}
12          contactYou={modalView.contactYou}
13          removeRev={this.removeRev.bind(this)} />
14      }
15
16      {editView.isOpen
17        ? <FormContainer
18          addContact={this.addContact}
19          editContact={this.editContact}
20          handleCancel={this.toggleEdit.bind(this, null)}
21          contact={editView.contact} />
22
23        : <ContactList
24          contacts={contacts}
25          handleOnEditClick={this.toggleEdit}
26          handleOnDeleteClick={this.deleteContact} />
27      }
28    </div>
29  )
30 }
```

Listing 7.1: *Die Renderfunktion der* `Contacts` *Komponente des Prototypen amilia-qouch*

Unabhängig von einem Wert im State wird der Header, in den Zeilen fünf bis sieben, gerendert. Im wird die Information `isOpen` übergeben die aussagt ob der Editiermodus an ist, also ob das Formular gerade geöffnet ist oder nicht. Ihm wird auch die Funktion `toggleEdit` gegeben, welche genau diese Statuseigenschaft wechselt.
Die Zeilen neun bis 14 gibt es nur im *amilia-qouch* Prototypen. Näheres dazu wird weiter unten im Abschnitt 7.3 erklärt.
In den Zeilen 16 bis 27 wird anhand der `isOpen` Information entschieden ob das Formular oder die Liste gerendert wird. Die Liste bekommt alle im Appstate gespeicherten Kontakte, ebenfalls die `toggleEdit` Funktion und eine zum Löschen eines Kontakt. Es ist zu beachten, dass hier bei der `toggleEdit()` das `bind(this, null)` fehlt. Das liegt daran, dass wenn man aus der Liste das Formular öffnet, man den 'Bearbeiten' Knopf

betätigt hat und der zu bearbeitende Kontakt im Formular geladen wird. Dieser Kontakt wird dann für die Dauer seiner Bearbeitung im `state.editView.contact` gespeichert.
Das KontaktFormular ist aufgeteilt in eine Container– und eine Viewkomponente, weswegen hier in `Contacts` die Containerkomponente gerendert wird. Wo in der Viewkomponende alle beinhaltenden Elemente nur dargestellt werden, besitzt die Containerkomponente Logik. Der `FormContainer` bekommt in Zeile 21 den zu bearbeitenden Kontakt. Ist dieser leer, wird über das Formular ein neuer angelegt. Des Weiteren bekommt sie die ebenfalls die `toggleEdit` Funktionen und eine zum Anlegen, eine zum Bearbeiten des Kontakts (Zeilen 18 bis 20).

In Unterabschnitt 6.2.1 wurde beschrieben wie die Daten in den Prototypen angelegt, bearbeitet und gelöscht werden. Deswegen wird hier nur gezeigt, wie diese Funktionen zum Einsatz kommen.

7.2 Offlinefunktionalität

Die Prototypen *amilia-qouch* und *amilia-rdx* sind vollständig offline verwendbar. Alle in Abschnitt 2.1 beschriebenen Grundvoraussetzungen werden von beiden Prototypen erfüllt und sämtliche umgesetzten Funktionen sind sowohl mit als auch ohne Internetverbindung durchführbar.

7.2.1 Datenspeicherung

Cachen der Assets geht erst nach dem Build -> ServiceWorkerAbschnitt 6.1
Redux Offline speichert, wie in Unterabschnitt 3.2.3 bereits beschrieben, alle im Redux Store verwalteten Daten im LocalStorage. Abbildung 7.1 zeigt alle gespeicherten, nutzerInnengenerierten Daten im LocalStorage.

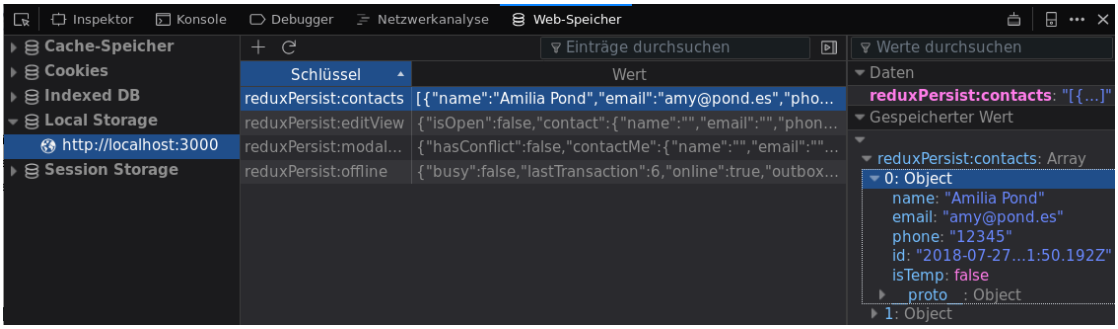


Abbildung 7.1: *Gespeicherte Daten des Prototypen amilia-rdx im LocalStorage, Screenhot: Developer Tools im Firefox Browser*

Der Prototyp *amilia-qouch* nutzt zur lokalen Datenspeicherung PouchDB. PouchDB speichert die von NutzerInnen generierten Daten in IndexedDB, vgl. Unterabschnitt 3.4.2. In Abbildung 7.2 sind die gespeicherten Daten in der IndexedDB zu sehen.

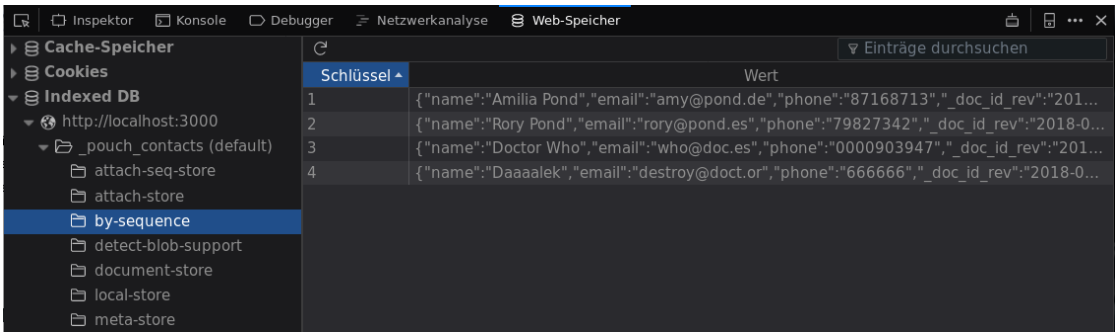


Abbildung 7.2: *Gespeicherte Daten des Prototypen aus amilia-qouch in IndexedDB, Screenhot: Developer Tools im Firefox Browser*

7.2.2 Datenbanksynchronisation

Zwischen PouchDB und CouchDB können Daten in Echtzeit synchronisiert werden. Um die Live–Replikation zu aktivieren, muss im Synchronisationsaufruf der Parameter `live: true` gesetzt sein. Bricht die Internetverbindung ab, stoppt auch die Synchronisation. Dank der angegebenen Parameter `retry: true` versucht PouchDB die Synchronisation solange neuzustarten bis die Anwendung wieder mit dem Internet verbunden ist. Listing 7.2 zeigt die Implementation der Datenbankensynchronisation im Prototypen *amilia-qouch*.

```
1 localDB.sync(remoteDB, {
2   live: true,
```

```
3   retry: true
4 })
```

Listing 7.2: *Synchronisation zwischen PouchDB und CouchDB im Prototyp amilia-qouch*

Redux Offline nimmt einem auch Arbeit bei der Datensynchronisation ab. Alle Daten die sich im Queue befinden werden automatisch an den Server gesendet, sobald eine Internetverbindung besteht. Das funktioniert jedoch nicht so einfach wenn der Server nicht an ist. In der Dokumentation von Redux Offline steht, dass die Aktion solange versucht wird auszuführen, bis die Anwendung wieder mit dem Internet verbunden ist [giv]. Allerdings wird die ROLLBACK Aktion gefeuert wenn der Server nicht verfügbar ist und die Aktion wird abgebrochen.

```
1 const discard = (error, _action, _retries) => {
2   const { response } = error
3   return response && response.status >= 400 && response.status <= 500
4 }
5
6 // apply dircard to config
7 const store = createStore(
8   reducer,
9   compose(
10     offline({
11       ...offlineConfig,
12       discard
13     })
14   )
15 )
```

Listing 7.3: *Discard Konfiguration für amilia-rdx*

Die Discard Konfiguration bestimmt wann eine Aktion abgebrochen, und wann sie immer wieder neugestartet wird. Im Listing 7.3 ist in den Zeilen eins bis vier abzulesen wie diese Konfiguration überschrieben wird. Nun wird die Aktion nur abgebrochen wenn der Server verfügbar ist und einen HTTP Status zwischen 400 und 500 zurückgibt. In den darauffolgenden Zeilen wird die eigens implementierte Konfiguration mit der Standardkonfiguration von Redux Offline zusammengeführt wird. Alle Daten werden nun auch nach einem Ausfall an den Server gesendet.
Im Gegensatz zum *amilia-qouch* Prototypen werden die Daten nur in eine Richtung automatisch gesendet. Es gibt keine Implementierung einer automatischen Replikation von den Daten auf dem Server zum lokalen Speicher.

7.3 Konfliktmanagement

Konflikte werden in beiden Prototypen manuell erzeugt wobei die Vorgehensweise identisch ist. Wie ein Konflikt herbeigeführt werden kann, wird in Abschnitt 6.4 beschrieben. Durch das regelmäßige Anfragen an den Server wird dargestellt, ob sich die Anwendung im Onlinestatus befindet oder nicht. Konflikte können erzeugt werden wenn ein oder beide Geräte auf denen die Anwendung läuft nicht mit dem Internet verbunden ist. Die Anzeige im Header dient der Kontrolle über diesen Status. Die folgenden Codeausschnitte illustrieren das Konfliktmanagement im Prototypen *amilia-qouch*.
Konflikte werden in CouchDB gespeichert, damit in der Anwendung entschieden werden kann wie damit umgegangen wird. Im Listing 7.4 werden alle Kontakteinträge geladen. Weil der Parameter in Zeile drei als Option mitgegeben wird, sind Konfliktinformationen für jeden Kontakt verfügbar. Gibt es unterschiedliche Versionen eines Kontaktes kommt er mit dem Attribut `_conflicts` beim Client an, eine Liste aus alles korrelierenden Revisionsnummern. In Zeile sieben werden wird der erste konfliktbehaftete Kontakt der vom Server kommt ermittelt. Wenn es einen Konflikt gibt, wird die Funktion `getConflictRevisions` in Zeile zehn aufgerufen.

```
1 async function getPouchDocs () {
2   const completeContacts = await localDB.allDocs({
3     include_docs: true,
4     conflicts: true
5   })
6   const contacts = completeContacts.rows.map(c => c.doc)
7   const conflictedContact = contacts.find(contact => contact._conflicts)
8
9   if (conflictedContact) {
10    this.getConflictRevisions(conflictedContact)
11  }
12 }
```

Listing 7.4: *Das Laden von konfliktbehafteten Kontakten*

Listing 7.5 zeigt die Umsetzung der `getConflictRevisions()`. Hier wird die konkurrierende Version des Kontakts ermittelt. Außerdem wird die Herkunft jeder Version festgestellt und das Öffnen eines Konfliktdialogs eingeleitet.
In Zeile zwei werden die Varieblen `contactMe` und `contactYou` initialisiert. Die erste repräsentiert die lokale Version, `contactYou` steht für die Version die aus der CouchDB kommt. In Zeile vier wird überprüft ob die übergebene Version des Kontakts mit dem zuletzt bearbeiteten übereinstimmt. Entsprechend werden die Variablen `contactMe` und `contactYou` befüllt. Die übergebene Version ist die von CouchDB festgelegte gewin-

nende Revision. Die andere, konkurrierende Version wird in Zeile acht, bzw. in Zeile 13 durch die Übergabe der Revisionsnummer im Parameter geladen. Die ID ist bei beiden Versionen identisch.
Dann wird das Öffnen des Konfliktdialogs durch das Aktualisieren den Appstatus initialisiert. Die beiden Kontaktversionen werden ebenfalls in den State geladen, um im Dialog korrekt angezeigt zu werden.

```
1  async function getConflictRevisions (conflictedContact) {
2    let contactMe, contactYou
3
4    if (equals(conflictedContact, this.state.lastEditedContact)) {
5      contactMe = conflictedContact
6
7      // To fetch the losing revision, simply get() it using the rev option
8      contactYou = await localDB.get(conflictedContact._id, {
9        rev: conflictedContact._conflicts[0]
10     })
11   } else {
12     contactYou = conflictedContact
13     contactMe = await localDB.get(conflictedContact._id, {
14       rev: conflictedContact._conflicts[0]
15     })
16   }
17
18   this.setState({
19     modalView: {
20       hasConflict: true, // open the modal
21       contactMe,
22       contactYou
23     }
24   })
25 }
```

Listing 7.5: *Das Ermitteln von konkurrierenden Kontaktversionen*

Der Konfliktdialog ist in Abbildung 7.3 dargestellt. Im Titel steht der Name der lokalen Version, rot hervorgehoben. Darunter befinden sich zwei große Knöpfe in unterschiedlichen Farben. Der erste zeigt die Version des Kontakts, die vom Server kommt, die auf einem anderen Gerät bearbeitet wurde. Der zweite, untere Knopf beinhaltet alle relevanten Informationen über die lokale Version des konfliktbehafteten Kontakts. So ist zu erkennen welche der beiden Versionen die eigene ist und worin sie sich von der anderen unterscheidet.

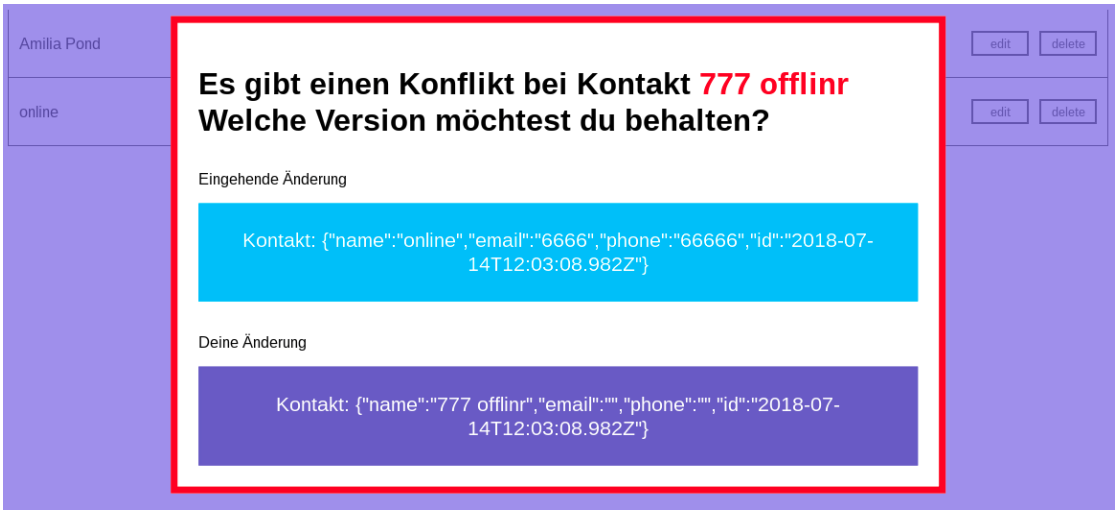


Abbildung 7.3: *Konfliktdialog des Prototypen amilia-qouch*

Durch das Klicken einer dieser Knöpfe wird entschieden welche der beiden Versionen behalten und welche eliminiert wird. Wird der Knopf mit der lokalen Version betätigt, wird die in Listing 7.6 gelistete Funktion `removeRev` mit der anderen Version im Parameter aufgerufen.

```
1 // remove the losing revision
2 async function removeRev (contact) {
3   await localDB.remove(contact._id, contact._rev)
4 }
```

Listing 7.6: *Das Eliminieren der verlierenden Version*

Dort, in Zeile drei, wird die verlierende Revision gelöscht.

Der Konfliktdialog konnte für den Protoypen *amilia-rdx* nicht umgesetzt werden da Redux offline nicht die Möglichkeit bietet Konflikte zu erkennen, gescheige denn zu speichern.

7.4 Installationsanleitung

build und so Beide entwickelten Prototypen sind als öffentliche Repositories auf GitHub[19] zu finden. Um sie zu installieren müssen folgende Schritte ausgeführt werden.

[19] Software-Entwicklungs-Plattform `https://github.com/`

7.4.1 AMILIA-QOUCH

Für die Ausführung dieser Anwendung muss CouchDB installiert sein auf der ein Admin mit dem Passwort ”admin” besteht. Zur Installation von CouchDB kann die Anleitung auf `http://docs.couchdb.org/en/2.1.2/install/index.html` verwendet werden.
Dann müssen die Cross-Origin Resource Sharing (CORS) aktiviert werden. Das ist eine Webtechnologie die es Webanwendungen erlaubt Ressourcen von einer anderen Domain zu benutzen. Es gibt bereit ein Skript, das genau das macht. Dazu müssen die folgenden Befehle ausgeführt werden.

```
npm install -g add-cors-to-couchdb
add-cors-to-couchdb
```

Waren diese Schritte erfolgreich, kann die Anwendung installiert werden.

1. Zuerst muss das Repository kopiert werden:

```
git clone git@github.com:hulkoba/amilia-qouch.git
# oder
git clone https://github.com/hulkoba/amilia-qouch.git
```

2. Dann wird in das Verzeichnis navigiert um dort alle Abhängigkeiten zu installieren.

```
cd amilia-qouch && npm install
```

3. Durch den Aufruf

```
npm start
```

wird die Anwendung gestartet und läuft auf `http://localhost:3000/`.

7.4.2 AMILIA-RDX

1. Auch hier muss das Repository zuerst kopiert werden:

```
git clone git@github.com:hulkoba/amilia-rdx.git
# oder
git clone https://github.com/hulkoba/amilia-rdx.git
```

2. Schritt zwei ist identisch mit dem in der Anleitung auf *amilia-qouch*
3. Durch die Aufrufe

```
npm run server
npm start
```

in separaten Terminalfenstern wird zuerst der Server und dann die Anwendung gestartet. Die Anwendung läuft nun auf `http://localhost:3000/`.

7.5 Testfälle

Folgende Testfälle zur Offlinefunktionalität an einem Gerät werden während der Entwicklung stetig durchgeführt. Das erfolgreiche Bestehen dieser Tests ist eine notwendige Qualitätseigenschaft der zu entwickelnden Prototypen.

Netzwerkstatus:
Die Anwendung muss zu jeder Zeit den korrekten Netzwerkstatus anzeigen.

Kontakte lesen:
Die Anwendung muss bei jedem Start die Kontakte aus dem lokalen Speicher oder aus der Datenbank, bzw der JSON Datei laden.

Kontakt anlegen:
Die Anwendung muss zu jedem Zeitpunkt in der Lage sein einen Kontakt mit jedem seiner Attribute anzulegen. Dazu muss er immer lokal gespeichert werden und sobald eine Internetverbindung besteht, persistiert werden. Das Anlegen eines Kontakts im Offlinestatus ist für die Konfliktherbeiführung erforderlich.

Kontakt bearbeiten:
Die Anwendung muss zu jedem Zeitpunkt in der Lage sein einen Kontakt mit jedem seiner Attribute zu bearbeiten. Ist keine Internetverbindung vorhanden, müssen die Änderungen lokal übernommen und später, sobald sich der Netzwerkstatus ändert, synchronisiert werden. Das Bearbeiten eines Kontakts im Offlinestatus ist für die Konfliktforcierung erforderlich.

Kontakt löschen:
Die Anwendung muss zu jedem Zeitpunkt in der Lage sein einen Kontakt zu löschen. Das Löschen eines Kontakts im Offlinestatus ist für die Konfliktherbeiführung erforderlich.

8 WIP Evaluation

Zur Evaluation der Prototypen wurden zuerst manuelle Tests durchgeführt, die im Anschluss ausgewertet werden. Hierbei lag das Augenmerk auf der Konfliktbehandlung der verwendeten Technologien, ob Daten im Falle eines Konflikts verloren gehen oder nicht. Tests: Anforderungen? Allgemeine Auswertung, welche Anfordrungen erfüllt wurden und welche nicht. Schließlich wird der Implementierungsaufwand von beiden Technologien verglichen.

8.1 Manuelle Tests

Um das Konfliktmanagement der zu testenden Technologien untersuchen zu können wurden manuelle Tests durchgeführt. Die Art und Weise wie die Tests durchgeführt wurden, ist in Abschnitt 6.4 detailliert beschrieben. Der entsprechende Prototyp läuft in den beiden Browsern Chrome und Firefox. In jedem Testdurchlauf werden unterschiedliche Aktionen in anderer Reihenfolge durchgeführt.

Als Ergebnis wird der Zustand der App bezeichnet wie er nach der Synchronisation beider Geräte ist. Das erwartete Ergebnis ist das gewünschte Ergebnis, welches eintritt wenn der Schwerpunkt bei der Verwendung der Anwendung darauf liegt, keine BenutzerInnendaten zu verlieren.
Das tatsächliche Ergebnis ist, wie der Name schon sagt, das was eingetreten ist. Weicht das tatsächliche Ergebnis von dem erwarteten ab, sind die Ansprüche in diesem Testdurchlauf nicht erfüllt.

8.1.1 Erste Testreihe

Dies ist Durchführung der ersten Testreihe. Bei diesen Tests werden Aktionen ausgeführt während beide Geräte mit dem Internet verbunden sind. Diese Tests zeigen, dass die Anwendung grundsätzlich auf mehreren Geräten einsatzbereit ist.

Kontakt anlegen

Ein Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsern angelegt.

Erwartetes Ergebnis
Der Kontakteintrag mit dem Namen ”Amilia Pond” existiert zwei mal auf jedem Gerät.
Tatsächliches Ergebnis *amilia-qouch*
Der Kontakteintrag mit dem Namen ”Amilia Pond” existiert zwei mal auf jedem Gerät.
Tatsächliches Ergebnis *amilia-rdx*
Der Kontakteintrag mit dem Namen ”Amilia Pond” existiert zwei mal auf jedem Gerät.

Kontakt bearbeiten

Ein Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsern bearbeitet. Zuerst wird im Firefox die E-Mail Adresse ”amilia@pond.de” hinzugefügt und gespeichert. Dann wird im Chrome die E–Mail Adresse des Kontakts auf ”amilia.pond@rory.de” geändert und gespeichert.

Erwartetes Ergebnis
Ein Kontakteintrag mit dem Namen ”Amilia Pond” hat die E–Mail Adresse ”amilia.pond@rory.de”.
Tatsächliches Ergebnis *amilia-qouch*
Ein Kontakteintrag mit dem Namen ”Amilia Pond” hat die E–Mail Adresse ”amilia.pond@rory.de”.
Tatsächliches Ergebnis *amilia-rdx*
Ein Kontakteintrag mit dem Namen ”Amilia Pond” hat die E–Mail Adresse ”amilia.pond@rory.de”.

Kontakt löschen

Der Kontakt mit dem Namen ”Amilia Pond” der keine E–Mail Adresse besitzt wird im Firefox gelöscht. Der ander Adressbucheintrag wird im Chrome gelöscht.

Erwartetes Ergebnis
Die Adressliste ist leer. Es ist kein gespeicherter Kontakt vorhanden.
Tatsächliches Ergebnis *amilia-qouch*
Die Adressliste ist leer. Es ist kein gespeicherter Kontakt vorhanden.
Tatsächliches Ergebnis *amilia-rdx*
Die Adressliste ist leer. Es ist kein gespeicherter Kontakt vorhanden.

8.1.2 Zweite Testreihe

Dies ist Durchführung der zweiten Testreihe. Bei diesen Tests werden Aktionen ausgeführt während ein Browser mit dem Internet verbunden ist, der andere jedoch nicht. Der Browser Chrome wird in den Offlinestatus versetzt und Firefox bleibt online. Nachdem in beiden Browsern die beschriebene Aktion vollendet wurde, wird im Chrome der Internetzugriff wieder aktiviert sodass die Daten synchronisiert werden können.

Kontakt anlegen

Ein Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsern angelegt.

Erwartetes Ergebnis
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.
Tatsächliches Ergebnis *amilia-qouch*
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.
Tatsächliches Ergebnis *amilia-rdx*
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.

Kontakt bearbeiten

Der Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsern bearbeitet. Zuerst wird im Firefox die E-Mail Adresse ”amilia@pond.de” hinzugefügt und gespeichert. Im Chrome wird dem Kontakt die E–Mail Adresse ”amilia.pond@rory.de” gegeben und gespeichert.

Erwartetes Ergebnis
Es entsteht ein Konflikt der von einem Menschen gelöst werden kann.
Tatsächliches Ergebnis *amilia-qouch*
Ein Konflikt ist entstanden und wurde gespeichert. Ein Dialog öffnet sich mit allen notwendigen Informationen. Es kann entschieden werden welche E–Mail Adresse übernommen werden soll.
Tatsächliches Ergebnis *amilia-rdx*
Der Kontakt hat die E-Mail Adresse ”amilia.pond@rory.de”.

Kontakt bearbeiten und denselben Kontakt löschen

Defr Kontakt mit dem Namen ”Amilia Pond” wird in einem Browser bearbeitet, in dem anderen gelöscht. Zuerst im Chrome wird der Name des Kontakts zu ”Rory Pond” geändert und gespeichert. Danach wird der Kontak im Firefox gelöscht.

Erwartetes Ergebnis
Auf beiden Geräten ist ein Kontakt mit dem Namen ”Rory Pond” gespeichert.
Tatsächliches Ergebnis *amilia-qouch*
Auf beiden Geräten ist ein Kontakt mit dem Namen ”Rory Pond” gespeichert.
Tatsächliches Ergebnis *amilia-rdx*
Die Kontaktliste ist leer.

Kontakt löschen und denselben Kontakt bearbeiten

Der Kontakt mit dem Namen ”Amilia Pond” wird in einem Browser bearbeitet, in dem anderen gelöscht. Zuerst wird der Kontakt im Chrome gelöscht. Danach wird im Firefefox der Name des Kontakts zu ”Rory Pond” geändert und gespeichert.

Erwartetes Ergebnis
Auf beiden Geräten ist ein Kontakt mit dem Namen ”Rory Pond” gespeichert.
Tatsächliches Ergebnis *amilia-qouch*
Auf beiden Geräten ist ein Kontakt mit dem Namen ”Rory Pond” gespeichert.
Tatsächliches Ergebnis *amilia-rdx*
Die Kontaktliste ist leer.

Kontakt löschen

Der Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsers gelöscht.

Erwartetes Ergebnis
Die Kontaktliste ist leer.
Tatsächliches Ergebnis *amilia-qouch*
Die Kontaktliste ist leer.
Tatsächliches Ergebnis *amilia-rdx*
Die Kontaktliste ist leer.

8.1.3 Dritte Trestreihe

Dies ist Durchführung der dritten Testreihe. Bei diesen Tests werden Aktionen ausgeführt während beide Browser mit dem Internet, aber nicht mit dem Server verbunden ist. Nachdem in beiden Browsern die beschriebene Aktion vollendet wurde, wird der Server wieder gestartet, sodass die Daten synchronisiert werden können.

Kontakt anlegen

Ein Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsern angelegt.

Erwartetes Ergebnis
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.
Tatsächliches Ergebnis *amilia-qouch*
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.
Tatsächliches Ergebnis *amilia-rdx*
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.

Kontakt bearbeiten

Ein Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsern bearbeitet. Im Firefox dem Kontakt die E-Mail Adresse ”amilia@pond.de” hinzugefügt und gespeichert. Im Chrome wird dem Kontakt die E–Mail Adresse ”amilia.pond@rory.de” gegeben und gespeichert.

Erwartetes Ergebnis
Es entsteht ein Konflikt der von einem Menschen gelöst werden kann.
Tatsächliches Ergebnis *amilia-qouch*
Ein Konflikt ist entstanden und wurde gespeichert. Ein Dialog öffnet sich mit allen notwendigen Informationen. Es kann entschieden werden welche E–Mail Adresse übernommen werden soll.
Tatsächliches Ergebnis *amilia-rdx*
Der Kontakt hat die E-Mail Adresse ”amilia@pond.de”.

Kontakt bearbeiten und denselben Kontakt löschen

Der Kontakt mit dem Namen ”Amilia Pond” wird in einem Browser bearbeitet, in dem anderen gelöscht. Zuerst im Firefox wird der Name des Kontakts zu ”Rory Pond” geändert und gespeichert. Danach wird der Kontak im Chrome gelöscht.

Erwartetes Ergebnis
Auf beiden Geräten ist ein Kontakt mit dem Namen ”Rory Pond” gespeichert.
Tatsächliches Ergebnis *amilia-qouch*
Auf beiden Geräten ist ein Kontakt mit dem Namen ”Rory Pond” gespeichert.
Tatsächliches Ergebnis *amilia-rdx*
Die Kontaktliste ist leer.

KONTAKT LÖSCHEN

Der Kontakt mit dem Namen ”Amilia Pond” wird in beiden Browsers gelöscht.

Erwartetes Ergebnis
Die Kontaktliste ist leer.
Tatsächliches Ergebnis *amilia-qouch*
Die Kontaktliste ist leer.
Tatsächliches Ergebnis *amilia-rdx*
Die Kontaktliste ist leer.

8.1.4 ÜBERSICHT DER TESTERGEBNISSE

Die folgende Tabelle gibt eine Übersicht der Testergebnisse. Sie zeigt an in welchem Testdurchlauf das erwartete Ergebnis eingetreten ist, und bei welchen Prototypen dies der Fall war.

Testreihe	Aktion	Erwartung erfüllt: *amilia-qouch*	Erwartung erfüllt: *amilia-rdx*
1. Online Online	Kontakt anlegen	ja	ja
	Kontakt bearbeiten	ja	ja
	Kontakt löschen	ja	ja
2. Online Offline	Kontakt anlegen	ja	ja
	Kontakt bearbeiten	ja	nein
	Kontakt bearbeiten und löschen	ja	nein
	Kontakt löschen und bearbeiten	ja	nein
	Kontakt löschen	ja	ja
3. Offline Offline	Kontakt anlegen	ja	ja
	Kontakt bearbeiten	ja	nein
	Kontakt bearbeiten und löschen	ja	nein
	Kontakt löschen	ja	ja

Tabelle 8.1: *Testergebnisse*

Es ist deutlich zu erkennen, dass überall dort wo ein Kontakt bearbeitet wird, die Redux Offline App die Erwartungen nicht erfüllt. Überall dort wo die Erwartungen nicht erfüllt worden sind, sind Daten verloren gegangen. Im folgenden Abschnitt werden die Testergebnisse ausgewertet.

8.2 Auswertung

In diesem Kapitel werden die in Abschnitt 8.1 durchgeführten Tests ausgewertet. Anschließend wird kontrolliert welche der in Kapitel 5 erarbeiteten Anforderungen von den Prototypen erfüllt werden. Da die Arbeit insbesondere aus Entwicklungsperspektive relevant ist wird auch der Implementierungsaufwand für beide Prototypen verglichen.

8.2.1 Auswertung der manuellen Tests

Zur Auswertung der Testergebnisse werden die Testreihen gruppert. Die zweite und dritte Testreihe befassen sich mit der Offlinefähigkeit der Prototypen. Es spielt hierbei keine Rolle ob ein oder zwei der Geräte während der Testdurchläufe offline sind. Deswegen werden diese Testreihen in der Auswertung zusammengefasst.

Kollaborationsfähigkeit

In der ersten Testreihe sind beide Geräte die ganze Zeit mit dem Internet verbunden. Das Hauptaugenmerk lag auf der allgemeinen Funktionalität des jeweiligen Prototypen auf zwei Geräten. In der Tabelle 8.1 ist abzulesen dass beide Prototypen alle Tests dieser Reihe eindeutig bestanden haben. Damit kann bestätigt werden, dass beide Prototypen auf zwei Geräten funktionieren, solange diese online sind.

Offlinefunktionalität und Konfliktmanagement

Während der Testreihen zwei und drei waren entweder ein oder beide Geräte für die Ausführung bestimmter Aktionen offline. Der Fokus dieser Tests lag auf dem eventuell eintretenden Datenverlust im Konfliktfall. Tabelle 8.1 zeigt, dass das Anlegen und Löschen weder bei *amilia-qouch*, noch bei *amilia-rdx* ein Problem darstellte. Beim Anlegen eines Kontakts war dieser auf beiden Geräten doppelt in der Liste. Auf diesem Wege ging kein Kontakt verloren. Auch das Löschen eines erfolgte reibungslos. Der gelöschte Kontakt war nach der Synchronisation auf keinem der Geräte vorhanden.
Sowohl in der zweiten, als auch in der dritten Testreihe hat der Redux offline Prototyp

beim Bearbeiten, bzw. beim Bearbeiten und Löschen eines Kontakts versagt. Wo im Prototypen *amilia-qouch* die aufgetretenen Konflikte gespeichert wurden, um sie auf Benutzungsebene zu lösen, gingen im anderen Protoypen eine der konfliktbehafteten Datensätze verloren. Bei gleichzeitigen Bearbeiten eines Kontakts gewann die Version, die zuletzt synchronisiert wurde. Die andere wurde überschrieben. Wurde ein Kontakt auf einem Gerät bearbeitet und auf dem anderen gelöscht, war es in jedem Fall nicht mehr in der Liste. Löschungen haben immer Vorrang, wodurch ebenfalls Daten verloren gehen.

Beobachtungen

Zusammenfassend lässt sich sagen, dass Redux Offline per se nicht auf mehreren Geräten offlinefähig ist. Bei den Testfällen in Abschnitt 7.5 die während der Entwicklung stetig – auf einem Gerät – durchgeführt wurden, bestand *amilia-rdx* jeden Test. Auf nur einem Gerät ist es jedoch schwierig Konflikte zu erzeugen. Deswegen ist die LWW Strategie von Redux Offline für die Verwendung auf einem Gerät akzeptabel.
Der Prototyp *amilia-qouch* hingegen hat alle Tests erwartungsgemäß bestanden. Immer wenn es zu einem Konflikt gekommen ist, konnte dieser über den Konfliktdialog einfach und vor allem korrekt gelöst werden. In keinem Testdurchlauf gingen Daten verloren.
Diesen Absatz woanders hin? Sobald die Anwendung auf mehreren Geräten benutzt soll, ist eine Backendimplementierung notwendig. Und wenn die Anwendung zusätzlich offlinefähig sein soll, bedarf es einen Synchronisationsalgorithmus. Soll Redux Offline verwendet werden, muss beides zusätzlich implementiert werden, damit die Anwendung auf mehreren Geräten offlinefähig ist und keine Daten verloren gehen.

8.2.2 WIP Erfüllung der Anforderungen

8.2.3 Implementierungsaufwand

Beim Vergleich des Implementierungsaufwands werden verschiedene Punkte Berücksichtigt. Es wird der Arbeitsaufwand betrachtet, der benötigt wird die verwendete Technologie zu verstehen und zu benutzen. Ebenfalls von hoher Wichtigkeit ist die Lesbarkeit des geschriebenen Quellcodes. Sauberer und verständlicher Code ist einfacher les– und somit auch wartbarer. Interessant ist auch die Menge des geschriebenen Codes.

Einbinding der Technologien

amilia-qouch
Der Aufwand eine Anwendung mit PouchDB und CouchDB zu schreiben, ist gering. Man muss nur beide Technologien installieren und im Code instanziieren. PouchDB dient als

Schnittstelle zu CouchDB, weswegen nur das simpel gehaltene API von PouchDB benutzt wird. Die Dokumentationen beider Technologien sind sehr ausfühlich und tragen ausgeprochen zum Verständnis der Funktionsweise bei.

amilia-rdx

Der Aufwand eine Anwendung mit Redux Offline zu schreiben, ist deutlich komplexer, da Redux eingesetzt und verstanden werden muss. Die Einbindung von Redux Offline war allerdings minimal aufwändig, sobald Redux einmal implementiert war. Redux ist sehr gut Dokumentiert und bietet viele Beispiele zur Anwendung. Die Dokumentation von Redux Offline besteht aus einer verschachtelten Readme Datei, was die Lesbarkeit und somit das Verständnis der Funktionsweise der Technologie erschwert.

Lesbarkeit des Codes

amilia-qouch

Auch wenn man mit den Technologien nicht vertraut ist, lässt sich der Code problemlos lesen. Für die Synchronisation beider Technologien reicht eine Zeile Code und umgesetzten CRUD Operationen sind dank unmissverständlicher API eindeutig nachvollziehbar.

amilia-rdx

Um den Code ohne Probleme lesen zu können sollte man mit Redux und dessen Datenfluss vertraut sein. Ist das der Fall, ist ein Blick in die Dokumentation von Redux Offline hilfreich, da die verwendeten Metaattribute der Aktionen nicht unbedingt eindeutig sind. Sind diese Dinge klar, ist auch der Code von *amilia-rdx* gut lesbar und begreiflich.

Menge des geschriebenen Quellcodes

Beide Prototypen wurden mit React Create App erstellt. Die Menge des Codes die dadurch erzeugt wurde, wurde nicht mitgezählt. Ebensowenig wurden die Kommentare und das geschriebene CSS gezählt.

amilia-qouch

Es wurden 371 Zeilen Code hinzugefügt.

amilia-rdx

Es wurden 586 Zeilen Code hinzugefügt.

9 WIP Zusammenfassung und Ausblick

Interessant wäre es andere, 'richtige' Datenbanklösungen zu untersuchen. Realm ist leider kostenpflichtig, wäre aber äußerst spannend weil Realm so viel verspricht.
Argumente gegen Realm:

- benutzt für Update LWW
- wenn Löschen immer gewinnt, gehen auch Daten erloren
- Object Server ist nict Open Source. Dort befindet sich aber die Logik für die Synchronisation. Es ist also nicht nazulesen und nur spärlich dokumentiert

Pouch hat eine äüßerst gute und ausführliche Dokumentation.

Interessant: mongoDB stitch

Abkürzungen

API	Application Programming Interface
App	Applikation
CAP	Consistency Availability Partition tolerance
CORS	Cross-Origin Resource Sharing
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

Glossar

Assets
: alle Bestandteile einer Webanwendung die für die für die erfolgreiche Ansicht im Browser benötigt werden. Es sind HTML– CSS– und JavaScriptdateien zu nennen, aber auch Mediendateien wie Bilder.

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

Kollaborativ
: Als kollaborative Software oder kollaboratives System wird eine Software zur Unterstützung der computergestützten Zusammenarbeit in einer Gruppe über zeitliche und/oder räumliche Distanz hinweg bezeichnet

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.

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

2.1 Browserkompatibilität für Service Worker 9
2.2 Browserkompatibilität für Web Storage 10
2.3 Browserkompatibilität für IndexedDB 11
2.4 Browserkompatibilität für IndexedDB 2.0 12
2.5 Die Grundidee von OT 16
2.6 Replikation von zustandsbasierten CRDTs 17
2.7 Replikation von Operationsbasierten CRDT 18
2.8 Das CAP Theorem 21

3.1 Redux Datenfluss 23
3.2 Redux Offline 25
3.3 HOODIE Architektur 27

4.1 Szenarien bei der Datenübertragung über das Netzwerk 33

5.1 Dialogfenster im Konfliktfall 42

6.1 Create React App: initiale Testapplikation 44
6.2 Komponentendiagramm 50
6.3 Client-Server-Modell des Prototypen *amilia-qouch* 51
6.4 Client-Server-Modell des Prototypen *amilia-rdx* 52
6.5 Die Kontaktliste in beiden Netzwerkstatus 58
6.6 Die Editieransicht in beiden Netzwerkstatus 58

7.1 Gespeicherte Daten im LocalStorage 64
7.2 Gespeicherte Daten in IndexedDB 64
7.3 Konfliktdialog des Prototypen amilia-qouch 68

QUELLCODEVERZEICHNIS

2.1 Beispiel einer HTML-Datei mit einer Manifest-Attribut Einbindung 8
2.2 Beispiel einer `.appcache`-*Datei* 8

3.1 Beispiel React Native Offline Aktion 26

6.1 Eine React Containerkomponente 45
6.2 Eine React Viewkomponente 46
6.3 Erstellen eines Stores mit Redux Offline 47
6.4 Aktion `fetchContacts` mit Metaattribut 47
6.5 Reducer mit allen Aktionen die im Meta Feld beschrieben werden 48
6.6 Persistierung der Daten mit PouchDB und CouchDB 53
6.7 Speicherung der Daten mit Redux Offline 54
6.8 Mögliche Serverimplementierung für das Hinzufügen eines Kontakts . . . 55
6.9 Beispiel einer React Detect Offline Implementierung 56

7.1 Die Renderfunktion der `Contacts` Komponente des Prototypen *amilia-qouch* 62
7.2 Synchronisation zwischen PouchDB und CouchDB im Prototyp *amilia-qouch* 64
7.3 Discard Konfiguration für amilia-rdx 65
7.4 Das Laden von konfliktbehafteten Kontakten 66
7.5 Das Ermitteln von konkurrierenden Kontaktversionen 67
7.6 Das Eliminieren der verlierenden Version 68

LITERATURVERZEICHNIS

[Abr] ABRAMOV, Dan: *Presentational and Container Components*. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0,. – Zugriff: 22.07.2018 6.1.1

[Acu] ACUÑA, Raúl G.: *React Native Offline*. https://github.com/rauliyohmc/react-native-offline,. – Zugriff: 12.04.2018 3.3

[Acu17] ACUÑA, Raúl G.: Your React Native Offline Tool Belt. In: *Medium* (2017), 6. – Zugriff: 12.04.2018

[Arc12] ARCHIBALD, Jake: *Application Cache is a Douchebag*. http://alistapart.com/article/application-cache-is-a-douchebag, 2012. – Zugriff: 06.05.2018 2.1.1

[Arc14] ARCHIBALD, Jake: *The offline cookbook*. https://jakearchibald.com/2014/offline-cookbook/, December 2014. – Zugriff: 06.05.2018 2.1.1

[Att04] ATTIYA, HAGIT AND WELCH, JENNIFER: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. USA : John Wiley & Sons, Inc., 2004. – ISBN 0471453242 2.2.1

[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

[Bol] BOLIN, Chris: *React Detect Offline – Offline and Online components for React*. https://github.com/chrisbolin/react-detect-offline,. – Zugriff: 17.06.2018 6.2.2

[cana] *Can I use IndexedDB?* https://caniuse.com/#search=indexedDB,. – Zugriff: 06.06.2018 2.3, 2.4

[canb] *Can I use Service Workers?* https://caniuse.com/#feat=serviceworkers,. – Zugriff: 06.05.2018 2.1

[canc] *Can I use Web SQL Database?* https://caniuse.com/#feat=sql-storage,. – Zugriff: 26.06.2018 2.1.2

[cand] *Can I use Web Storage?* https://caniuse.com/#feat=feat=namevalue-storage,. – Zugriff: 06.06.2018 2.2

[coua] *CouchDB – relax.* https://couchdb.apache.org/,. – Zugriff: 12.04.2018 3.4.1

[coub] *CouchDB Replication Protocol.* http://docs.couchdb.org/en/master/replication/protocol.html,. – Zugriff: 12.04.2018

[Ell89] ELLIS, C. A. AND GIBBS, S. J.: Concurrency Control in Groupware Systems. In: *SIGMOD Rec.* 18 (1989), jun, Nr. 2, 399–407. http://dx.doi.org/10.1145/66926.66963. – DOI 10.1145/66926.66963. – ISSN 0163-5808 2.2.3

[Evä17] EVÄKALLADO, Jani: Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native. In: *HACKERnoon* (2017), 3. – Zugriff: 12.04.2018 3.2.2, 3.2, 6.2

[fal] *Eight Fallacies of Distributed Computing.* https://blog.fogcreek.com/eight-fallacies-of-distributed-computing-tech-talk/, . – Zugriff: 12.04.2018 2.2.1

[Fra09] FRASER, Neil: Differential Synchronization. Version: Jan 2009. https://neil.fraser.name/writing/sync/eng047-fraser.pdf. 2009. – Research Report. – 8 S.

[Gau18] GAUNT, Matt: *Service Workers: an Introduction.* https://developers.google.com/web/fundamentals/primers/service-workers/, 2018. – Zugriff: 06.05.2018 2.1.1

[giv] *Redux Offline – Giving up is hard to do.* https://github.com/redux-offline/redux-offline#giving-up-is-hard-to-do, . – Zugriff: 09.04.2018 7.2.2

[Hei12] HEILMAN, Chris: There is no simple solution for local storage. In: *Mozilla HACKS* (2012), 3. – Zugriff: 12.06.2018 2.1.2

[hooa] *Hoodie – The Offline First Backend.* http://hood.ie,. – Zugriff: 12.04.2018 3.4

[hoob] *How Hoodie Works.* http://docs.hood.ie/en/latest/about/how-hoodie-works.html,. – Zugriff: 12.04.2018 3.3, 3.4, 6

[idba] *IndexedDB*. https://developer.mozilla.org/de/docs/IndexedDB,. – Zugriff: 01.07.2018 2.1.2

[idbb] *Indexed Database API 2.0*. https://www.w3.org/TR/IndexedDB-2/,. – Zugriff: 01.07.2018 2.1.2

[idbc] *Indexed Database API 3.0*. https://w3c.github.io/IndexedDB/,. – Zugriff: 01.07.2018 2.1.2

[Iwa16] IWANIUK, Daniel: *How to persist Redux State to the Local Storage*. https://www.hawatel.com/blog/how-to-persist-redux-state-to-the-local-storage/, 10 2016. – Zugriff: 29.06.2018 3.2.3

[J. 10] J. CHRIS ANDERSON AND JAN LENHARDT AND NOAH SLATER: *CouchDB: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, 2010. – ISBN 978-0-596-15589-6 2.2.1, 2.2.7, 2.2.7

[Li,10] LI, DU AND LI, RUI: An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems. In: *Comput. Supported Coop. Work* 19 (2010), feb, Nr. 1, 1–43. http://dx.doi.org/10.1007/s10606-009-9103-1. – DOI 10.1007/s10606-009-9103-1. – ISSN 0925-9724

[loc] *localForage - Offline storage, improved*. https://localforage.github.io/localForage/,. – Zugriff: 26.06.2018 3.2.3

[Net] NETWORK, Mozilla D.: *Browser storage limits and eviction criteria*. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria,. – Zugriff: 05.07.2018 2.1.1

[Net18] NETWORK, Mozilla D.: *Using the application cache*. https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache, 2018. – Zugriff: 05.05.2018 2.1.1

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

[ot-] *Operational Transformation*. https://operational-transformation.github.io,. – Zugriff: 12.04.2018

[poua] *pouchdb – Tha Database that Syncs!* https://pouchdb.com/learn.html,. – Zugriff: 12.04.2018 3.4.2

[poub] *API Reference - Create/update a document.* https://pouchdb.com/guides/updating-deleting.html#why-must-we-dance-this-dance,. – Zugriff: 24.06.2018

[pouc] *pouchdb – Conflicts.* https://pouchdb.com/guides/conflicts.html,. – Zugriff: 12.04.2018 6.2.1

[poud] *Replicate a database.* https://pouchdb.com/api.html#replication,. – Zugriff: 28.06.2018 6.2.1

[poue] *Couchbase, CouchDB, Couch-what?* https://pouchdb.com/guides/#couchbase-couchdb-couch-what,. – Zugriff: 02.07.2018

[reaa] *realm – The new standard in data synchronization.* https://realm.io/,. – Zugriff: 12.04.2018 3.5

[reab] *Conflict Resolution.* https://docs.realm.io/platform/self-hosting/customize/conflict-resolution,. – Zugriff: 06.07.2018 3.5

[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 3.5

[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 3.5

[reda] *Redux.* https://redux.js.org/basics,. – Zugriff: 07.06.2018 3.2.1

[redb] *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 3.2.2

[redc] *Redux Offline Documentation.* https://github.com/redux-offline/redux-offline/tree/develop/docs,. – Zugriff: 27.06.2018 3.2.2

[redd] *Redux Offline.* https://github.com/redux-offline/redux-offline,. – Zugriff: 12.04.2018 3.2.2

[rede] *Redux Offline Documentation.* https://www.npmjs.com/package/redux-offline#giving-up-is-hard-to-do,. – Zugriff: 12.07.2018 6.1.2

[redf] *Redux Persist.* https://github.com/rt2zz/redux-persist#storage-engines,. – Zugriff: 12.04.2018 3.2.3

[red17] *Redux Offline without Redux*. https://github.com/redux-offline/redux-offline/issues/48, 2017. – Zugriff: 05.07.2018 3.2

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

[Sha11a] SHAPIRO, MARC AND PREGUIÇA, NUNO AND BAQUERO, CARLOS AND ZAWIRSKI, MAREK: A comprehensive study of Convergent and Commutative Replicated Data Types / Inria – Centre Paris-Rocquencourt ; INRIA. Version: Jan 2011. https://hal.inria.fr/inria-00555588. 2011 (RR-7506). – Research Report. – 50 S. 2.2.4, 2.2.4

[Sha11b] SHAPIRO, MARC AND PREGUIÇA, NUNO AND BAQUERO, CARLOS AND ZAWIRSKI, MAREK: Conflict-free Replicated Data Types. Version: Jul 2011. https://hal.inria.fr/inria-00609399. 2011 (RR-7687). – Research Report. – 18 S. 2.2.4, 2.2.4, 2.6, 2.2.4, 2.2.4, 2.7, 2.2.4

[Soua] SOURCE, Facebook O.: *Create React App – Create React apps with no build configuration*. https://github.com/facebook/create-react-app,. – Zugriff: 11.06.2018 6.1

[Soub] SOURCE, Facebook O.: *React – A JavaScript library for building user interfaces*. https://reactjs.org/,. – Zugriff: 22.06.2018 6.1.1

[Souc] SOURCE, Facebook O.: *React Native – Build native mobile apps using JavaScript and React*. https://facebook.github.io/react-native/,. – Zugriff: 27.06.2018 3.3

[Sto] STOLYAR, Arthur: *offline-plugin*. https://github.com/NekR/offline-plugin,. – Zugriff: 12.04.2018 3.1

[Sun98] SUN, CHENGZHENG AND ELLIS, CLARENCE: Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. New York, NY, USA : ACM, 1998 (CSCW '98). – ISBN 1–58113–009–0, 59–68 2.2.3, 2.2.3

[Sun11a] SUN, Chengzheng: *A scenario for illustrating the basic idea of OT for consistency maintenance*. http://www3.ntu.edu.sg/home/czsun/projects/otfaq/#fig_basic_idea_OT, 2010 – 2011. – Zugriff: 22.07.2018

[Sun11b] SUN, Chengzheng: *What collaboration capabilities can OT support?* http://www3.ntu.edu.sg/home/czsun/projects/otfaq/#_Toc321146127, 2010 – 2011. – Zugriff: 22.07.2018 2.2.3

[weba] *webpack – bundle your [scripts, images, styles, assets].* https://webpack.js.org/,. – Zugriff: 12.06.2018

[webb] *Options.* https://github.com/NekR/offline-plugin/blob/master/docs/options.md,. – Zugriff: 25.06.2018 3.1

[webc] *Web SQL Database.* https://www.w3.org/TR/webdatabase/,. – Zugriff: 26.06.2018 2.1.2

[webd] *Web Storage API.* https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API,. – Zugriff: 06.06.2018 2.1.2

[Yar17] YARABARLA, Sandeep: *Learning Apache Cassandra – Second Edition.* 35 Livery Street, Birmingham B3 2PB, UK. : Pack Publishing, 2017. – ISBN 9781787127296. – https://www.safaribooksonline.com/library/view/learning-apache-cassandra/9781787127296/1bf11eaf-f276-4762-b192-fd7b20e7f531.xhtml – Zugriff: 19.07.18 2.2.2

Anhang

Eidesstattliche Erklärung

CD-Inhalt

Auf der beigefügten CD befinden sich

- Die schriftliche Ausarbeitung dieser Masterarbeit im PDF-Format
- Der Quellcode beider Prototypen