



BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Masterarbeit

Medieninformatik

Fachbereich VI – Informatik und Medien

Untersuchung der Konfliktmanagementstrategien verschiedener offlinefähiger Systeme

Berlin, den 20. Juli 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	WIP Konfliktmanagementstrategien	13
2.2.1	Konflikte	13
2.2.2	Eventual Consistency	15
2.2.3	Last-Write-Wins	16
2.2.4	WIP Operational Transformation	16
2.2.5	WIP Conflict-Free Replicated Data Type	18
2.2.6	Differential Synchronization?	19
2.2.7	WIP Das CouchDB Replikationsmodell	19
3	Bestehende offlinefähige Technologien	21
3.1	Offline plugin für webpack	21
3.2	Redux Offline	21
3.2.1	Redux	22
3.2.2	Redux Offline	22
3.2.3	Redux Persist	24
3.3	React Native Offline	24
3.4	hoodie	25
3.5	Realm	26
3.6	Übersicht	28
4	Szenarien	29
4.1	Szenarien bei der Datenübertragung	29
4.2	Szenarien zur Konfliktentstehung	31

5	Anforderungsdefinition	35
5.1	Umfang	35
5.2	Funktionale Anforderungen	37
5.3	User-Stories	38
5.3.1	NutzerInnen Perspektive	38
5.3.2	EntwicklerInnen Perspektive	38
5.3.3	TesterInnen Perspektive	39
5.4	Bedienoberfläche	40
6	Konzeption	42
6.1	Anwendungsaufbau	42
6.1.1	Aufbau der React Komponenten	43
6.1.2	Verwendung von Redux Offline	44
6.2	Architektur	46
6.2.1	Das Speichern der Daten	49
6.2.2	Verbindungsstatus feststellen und ändern	53
6.3	Die graphische Oberfläche	54
6.4	Testdurchläufe	56
7	WIP Implementierung der Prototypen	58
7.1	WIP package.json?	58
7.2	WIP Hauptmodul Contacts	58
7.3	WIP Konfliktmanagement	58
7.4	WIP Umsetzung der Szenarien? Anforderungen?	58
7.5	WIP Installationsanleitung	59
7.5.1	amilia-qouch	59
7.5.2	amilia-rdx	59
7.6	WIP Testfälle	59
8	WIP Evaluation	61
8.1	WIP Test Konfliktmanagement	61
8.2	WIP Auswertung	61
8.2.1	WIP Implementierungsaufwand	61
8.2.2	WIP Offlinefunktionalität	62
8.2.3	WIP Konfliktmanagement	62
9	WIP Zusammenfassung und Ausblick	63
	Abkürzungen	64

Glossar	65
Abbildungsverzeichnis	66
Literaturverzeichnis	67
Anhang	73

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 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 Dateispeicherung 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 2016¹ 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

¹siehe <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

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].

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android	Blackberry	Opera Mobile	Chrome Android	Firefox Android	IE Mobile	UC for Android	Samsung Internet	QQ	Baidu
			49														
			65														
	16	59	66		52	10.3		4.4							4		
11	17	60	67	11.1	53	11.2	all	67	10	46	67	60	11	11.8	6.2	1.2	7.12
	18	61	68	12													
		62	69	TP													
			70														

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 Version um Funktionalität und verbessert die Performance [idbb]. Die Aktualität dieser Spezifikation spiegelt sich in der Browserunterstützung wider. Die Abbildung 2.4 zeigt, dass die zweite Version nur von den beliebten (haha) Desktopbrowsern und einigen mobilen Browsern unterstützt wird.

2 Grundlagen

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android *	Blackberry	Opera Mobile *	Chrome Android	Firefox Android	IE Mobile	UC for Android	Samsung Internet	QQ	Baidu
			1 49														
			65			10.3									4		
	16	59	66		52	11.2		4.4							2 6.2		
11	17	60	67	11.1	53	11.3	all	67	10	46	67	60	11	11.8	7.2	1.2	7.12
	18	61	68	12													
		62	69	TP													
			70														

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

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, Konflikte werden gelöst, 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.

CouchDB hat ein integriertes Replikationsprotokoll, das diesen Teil behandelt.

COUCHDB

Apache CouchDB™ ist ein Datenbank Management System (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 REpresentational State Transfer (REST)ful 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 wird in Unterabschnitt 2.2.7 detailliert beschrieben.

Dieses Protokoll ist die Grundlage für Offline First Anwendungen. Dank des Replikations-API 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].

POUCHDB

Als Ergänzung zu CouchDB kann PouchDB verwendet werden. PouchDB ist die JavaScript Implementierung von CouchDB. Sie ist quelloffen und wurde so konzipiert, dass sie in allen modernen Browsern 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 Anwendung wieder online ist mit CouchDB kompatiblen Servern synchronisiert werden [poua].

2.2 WIP 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. [ALS10] S. 153). Konflikte gehören in verteilten Systemen zur Realität und lassen sich nicht vermei-

den. Ein verteiltes System ist per Definition ein Zusammenschluss unabhängiger Computer, die sich für die NutzerInnen als ein einziges System präsentieren (vgl. [AW04] 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 Berta, ihre Telefonnummer. Der Zug fährt durch einen Tunnel und das Netzwerk bricht ab als Berta 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 Berta ihre Nummer zusätzlich per E-Mail. Berta schaltet ihren Laptop aus, weil sie sich mit Amilia unterhalten möchte. Am Abend ist Berta zu Hause angekommen und sie speichert Amilias Nummer aus der E-Mail in ihrem Adressbuch auf ihrem stationären Desktop PC. Jetzt gibt es Amilias Telefonnummer mit unterschiedlichen Informationen in Bertas Adressbuch auf verschiedene Geräten. Wenn Berta am nächsten Tag ihren Laptop startet und das Adressbuch sich mit dem auf dem PC synchronisiert besteht 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 eines Dokuments muss das kein Problem darstellen. Das Dokument kann beide Aktualisierungen erhalten indem die Änderungen zusammengefügt werden. Diese Prozedur wird *merge* genannt und wird unter anderem von

Git², 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. Zur Lösung dieses Problems wurden verschiedene Managementstrategien entworfen die im Folgenden vorgestellt werden.

2.2.2 EVENTUAL CONSISTENCY

Das muss noch woanders hin. oder raus

Das Consistency Availability Partition tolerance (CAP) Theorem, veranschaulicht in Abbildung 2.5, 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 Replike des manipulierten Datensatzes aktualisiert werden. Der Datensatz ist in jeder Datenbank identisch.

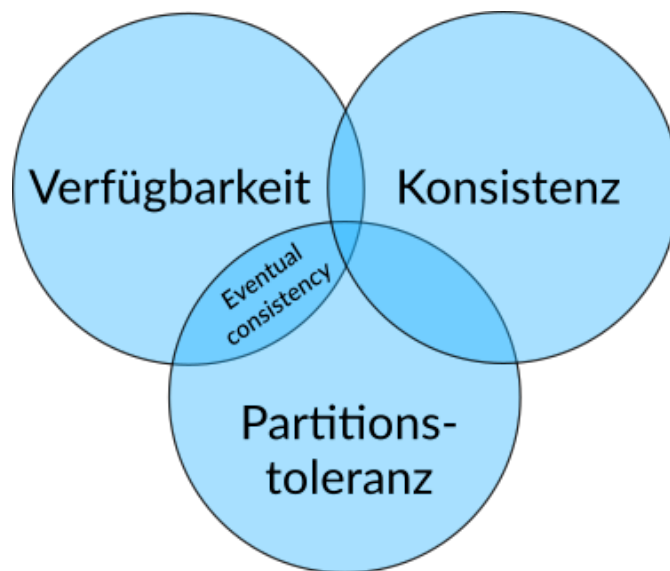


Abbildung 2.5: Das CAP Theorem

Das System besitzt eine hohe Verfügbarkeit wenn alle Anfragen an das System stets beantwortet werden. Die Verfügbarkeit ist gering, wenn die Antwortzeiten des Systems

²git steht unter <https://git-scm.com/downloads> zum Download bereit

lang sind. Partitionstoleranz ist gleichzusetzen mit Ausfalltoleranz. Die Datenbank kann auf mehreren Servern verteilt sein. Trotzdem ein Server oder eine Partition ausfällt, kann das System weiterhin funktionieren (vgl. [ALS10] S. 11 f.).

Eventual Consistency kommt häufig bei verteilten Datenbanken zur Anwendung und stellt die Konsistenz der Daten nach einem gewissen Zeitfenster sicher.

2.2.3 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].

2.2.4 WIP OPERATIONAL TRANSFORMATION

siehe woot

Operational Transformation (OT) ist eine weit verbreitete Technologie 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 [EG89]. Später ermöglichte weitere Forschung OT durch Unterstützung von *Sperrungen*, *Konfliktlösungen*, *Benachrichtigungen*, *Bearbeitung von Baumstrukturierten Dokumenten*, ... zu verbessern und erweitern. Im Jahr 2009: Google Wave, Google Docs

Kollaborative Systeme, die OT verwenden, benutzen normalerweise den replizierten Dokumentenspeicher. Das heißt jeder **Client** verfügt über eine eigene Kopie des Dokuments. Jede Änderung an einem freigegebenen Dokument wird als Operation dargestellt. Operationen sind Repräsentationen von Änderungen an einem Dokument. (Beispielsweise: Füge 'Hello world!' an Position 0 in das Textdokument ein). Eine Operation 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. Die Operationen erfolgen auf lokalen Kopie und die Änderungen werden an alle anderen **Clients** weitergegeben. Wenn ein **Client** die Änderungen von einem anderen **Client** empfängt, werden die Änderungen normalerweise **vor** ihrer Ausführung transformiert. Die Transformation stellt sicher, dass anwendungsabhängige Konsistenzkriterien (Invarianten) von allen Standorten gepflegt werden.

Es gibt die Operationen **Einfügen**

Das Einfügen besteht aus dem eingefügten Text und dessen Position im Dokument (`insert('h',`

0)). Für die Position kann ein Koordinatensystem ermittelt werden (Zeilennummer: Position in Zeile oder einfacher: Dokument wie eine Folge von Zeichen behandeln, also einfach einen nullbasierten Index vergeben.)

und Löschen

Löschen(5,6) = löscht 5 Zeichen, beginnend bei Position 6. Mehr benötigt man nicht, denn update = delete & insert

Um gleichzeitige Operationen zu behandeln, gibt es eine Funktion (normalerweise `Transform` genannt), die zwei Operationen übernimmt, die auf denselben Dokumentstatus angewendet wurden (aber auf verschiedenen Clients). Daraus wird eine neue Operation berechnet, die nach der zweiten Operation angewendet werden kann. Diese behält die erste beabsichtigte Änderung der Operation. Des Weiteren unterstützt OT Operationen wie `update`, `point`, `lock`.

Beispiel : Benutzer A fügt an Position 12 das Zeichen 'A' ein Benutzer B fügt am Anfang des Dokuments ein 'B' ein. Die konkurrierenden Operationen sind daher Einfügen (12, 'A') und Einfügen (0, 'B'). Wenn wir die Operation von B einfach an Client A senden und dort anwenden würden, gäbe es ein Problem. Aber wenn die Operation von A an B gesendet, und angewandt wird nachdem Operation B angewandt wurde ist, würde das Zeichen 'A' eine Position zu weit links von der korrekten Position eingefügt werden. Dokumentstatus A und Dokumentstatus B sind nicht identisch.

Daher muss A's `insert(12, 'A')` gegen die Operation von B transformiert werden. So wird berücksichtigt, dass B ein Zeichen vor der Position 12 eingefügt hat (die die Operation `insert(13, 'A')` erzeugt.)

Diese neue Operation kann auf Dokument B nach B's Operation angewandt werden. Die Grundidee von OT besteht darin, die Parameter einer Editieroperation gemäß den Auswirkungen **zuvor ausgeführter** konkurrierender Operationen anzupassen, so dass die transformierte Operation die korrekte Wirkung erzielen und die Dokumentenkonsistenz aufrechterhalten kann. False-Tie puzzle? *A Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems* und *Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems*

Während der klassische OT-Ansatz, Operationen durch ihre Versätze im Text zu definieren, einfach und natürlich zu sein scheint, werfen real verteilte Systeme ernsthafte Probleme auf. Nämlich, dass sich die Operationen mit endlicher Geschwindigkeit fortpflanzen, die Zustände der TeilnehmerInnen sind oft verschieden, so dass die resultierenden Kombinationen von Zuständen und Operationen extrem schwer vorherzusehen und zu verstehen sind. Wie Li und Li es ausdrückten: "Aufgrund der Notwendigkeit, eine komplizierte Fallabdeckung in Betracht zu ziehen, sind formale Beweise sehr kompliziert und

fehleranfällig, selbst für OT-Algorithmen, die nur zwei charakteristische Primitive behandeln (Einfügen und Löschen)“ [LL10].

Damit OT funktioniert, muss jede einzelne Änderung an den Daten erfasst werden: “Einen Schnappschuss des Zustands zu erhalten, ist normalerweise trivial, aber das Erfassen von Bearbeitungen ist eine ganz andere Sache. [...] Der Reichtum moderner Benutzerschnittstellen kann dies problematisch machen, besonders in einer browserbasierten Umgebung“ [Fra09].

2.2.5 WIP CONFLICT-FREE REPLICATED DATA TYPE

Die Idee von Conflict-free replicated data types (CRDTs) ist, dass jeder "Typ"(wie ein Einkaufswagen) mit Intelligenz handelt, um Konflikte automatisch zu lösen.

CRDTs sind Objekte, die ohne teure Synchronisation aktualisiert werden können. Sie konvergieren schließlich, wenn alle gleichzeitigen Aktualisierungen kommutativ³ sind und wenn alle Aktualisierungen schließlich von jeder Replik ausgeführt werden [SPBZ11a]. Um diese Garantien zu geben, müssen diese Objekte bestimmte Kriterien erfüllen, welche im Folgenden beschrieben werden [SPBZ11b].

ZUSTANDBASIERTER ANSATZ

Wenn ein Replikat ein Update von einem Client empfängt, aktualisiert es zuerst seinen lokalen Status und dann, einige Zeit später, seinen **vollständigen Status**. So sendet jedes Replikat gelegentlich seinen vollständigen Status an ein anderes Replikat im System. Um ein Replikat, das den Status eines anderen Replikats empfängt, wendet eine **Zusammenführungsfunktion** (merge) an, um den empfangenen Status mit dem lokalen Status zusammenzuführen. Entsprechend sendet dieses Replikat gelegentlich auch seinen Status an ein anderes Replikat, sodass jedes Update schließlich alle Replikate im System erreicht.

OPERATIONSBASIERTER ANSATZ (OT?)

Bei diesem Ansatz sendet ein Replikat seinen vollständigen Status (kann groß sein) nicht an ein anderes Replikat. Stattdessen sendet es nur den **Aktualisierungsvorgang** an **alle** anderen Replikate im System und erwartet von ihnen, dass sie das Update auf sich anwenden.

Da es sich um einen Sendevorgang handelt, wenn zwei Updates u_1 und u_2 , bei einem Replikat i angewendet werden und diese Updates an zwei Replikate r_1 und r_2 gesendet

³Unverändert bei Vertauschen der Operanden

werden, können diese Updates in unterschiedlicher Reihenfolge bei diesen replikaten ankommen. r_1 kann sie in der Reihenfolge u_1, u_2 empfangen, während bei r_2 die Updates in umgekehrter Reihenfolge (u_2, u_1) ankommen können. Sind die Aktualisierungen **kommutativ**, können die Repliken zusammengeführt werden, egal in welcher Reihenfolge die Updates bei ihnen ankommen - der resultierende Zustand ist derselbe. In diesem Modell wird ein Objekt, für das alle gleichzeitigen Aktualisierungen kommutativ sind, CmRDT (commutative replicated data type - kommutativ replizierter Datentyp) genannt.

Beispiel:...

CRDTs befassen sich mit einem interessanten und grundlegendem Problem in verteilten Systemen, haben jedoch eine wichtige Einschränkung: "Da ein CRDT konstruktionsbedingt keinen Konsens verwendet, hat der Ansatz starke Einschränkungen; Dennoch sind einige interessante und nicht-triviale CRDTs bekannt- [SPBZ11b]. Die Einschränkung ist, dass die CRDT-Adresse nur einen Teil des Problemraums betrifft, da nicht alle möglichen Aktualisierungsoperationen kommutativ sind und daher nicht alle Probleme in CRDTs umgewandelt werden können. Auf der anderen Seite können CRDTs für einige Arten von Anwendungen durchaus nützlich sein, da sie eine nette Abstraktion zur Implementierung replizierter verteilter Systeme bieten und gleichzeitig theoretische Konsistenzgarantien bieten.

2.2.6 DIFFERENTIAL SYNCHRONIZATION?

2.2.7 WIP DAS COUCHDB REPLIKATIONSMODELL

Aufgabe der Replikation von CouchDB ist die Synchronisation 2+n Datenbanken. Lösungen: Zuverlässige **Synchronisation** von Datenbanken auf verschiedenen Geräten. **Verteilung** der Daten über ein Cluster von DB-Instanzen die jeweils einen Teil des requests beantworten (Lastverteilung) und **Spiegelung** der Daten über geografisch weit verteilte Standorte.

Durch die inkrementelle (schrittweise) Arbeitsweise kann CouchDB genau dort weitermachen wo es unterbrochen wurde wenn während der Replikation ein Fehler auftritt, beispielsweise durch eine ausfallende Netzwerkverbindung *Es werden auch nur die Daten übertragen, die notwendig sind, um die Datenbanken zu synchronisieren.*

Das Besondere an CouchDB ist, dass es darauf ausgerichtet ist, Fehler/Konflikte vernünftig zu behandeln statt anzunehmen es träten keine auf (vgl. [ALS10] S. 7f). Wie oben beschrieben, gibt es in Verteilten Systemen einige Fehler die auftreten können.

Das CouchDB Replikationsmodell erlaubt eine nahtlose, peer-to-peer (direkte) Datensynchro-

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

Vektoruhr⁴

content addressable versions: Idee: Nimm den Objekthalt (content) und jag ihn durch eine Hashfunktion

Diese Art von Konflikten sollten von Menschen gelöst werden. Nur so kann sichergestellt werden, dass die korrekte Änderung gespeichert wird und keine Daten verloren gehen.

SCHLUSSENDLICHE KONSISTENZ

LOKALE KONSISTENZ

VERTEILTE KONSISTENZ

REPLIKATION?

KONFLIKTMANAGEMENT

⁴https://en.wikipedia.org/wiki/Vector_clock

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 ungebündelte 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.

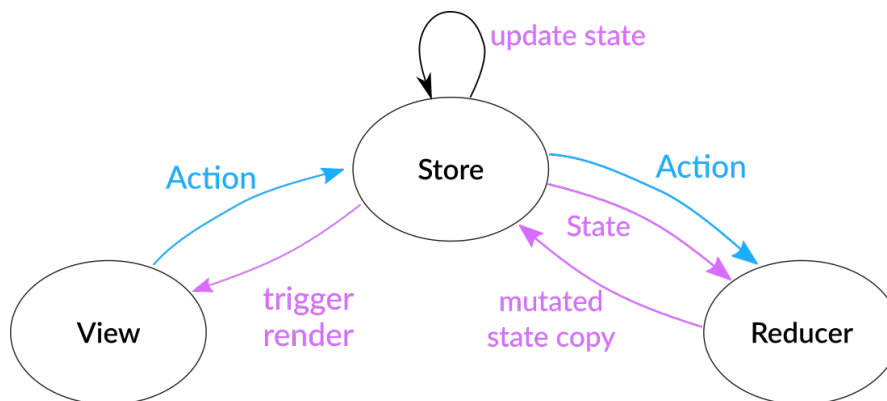


Abbildung 3.1: Redux Datenfluss

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 React⁵, Vue⁶, oder Angular⁷ [redb]. Es umfasst unter Anderem netzwerkfähige API-Aufrufe, das Persistieren des Zustands der Anwendung, das speichern von Aktionen, die Behandlung von Fehlern und erneute

⁵JavaScript Bibliothek: <https://reactjs.org/>

⁶JavaScript Framework: <https://vuejs.org/>

⁷JavaScript Framework: <https://www.angular.io>

Versuche die Verbindung wieder herzustellen. Redux Offline verspricht nicht, die Webanwendung komplett offlinefähig zu machen. Um Assets zwischenspeichern, 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 neu geladen. 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 eingebunden 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 STOREinternen 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. 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 Metaattribut 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 APP-STATE 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 `localStorage` 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 Anderen BenutzerInnen authentifiziert, Daten gespeichert und synchronisiert werden [hooa].

Anhand der Abbildung 3.3 wird erklärt wie HOODIE funktioniert.

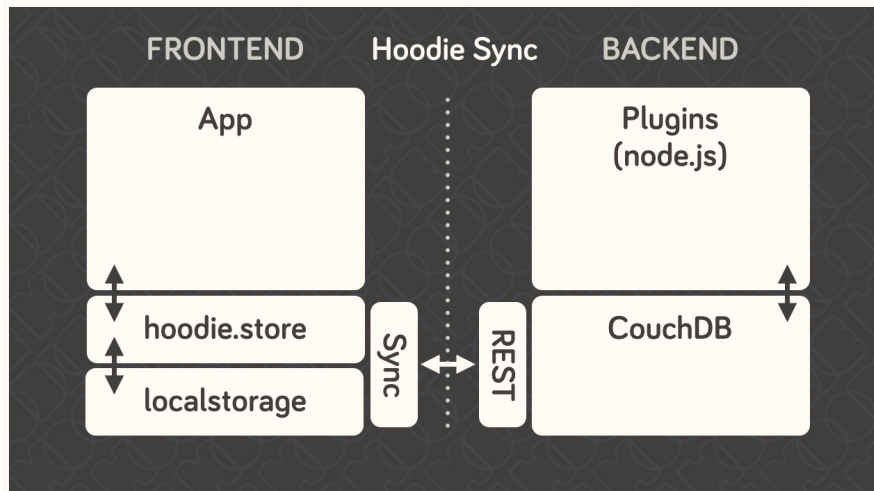


Abbildung 3.3: HOODIE Architektur Quelle: [hoob]

Im Frontend-Bereich 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 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.

3.5 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 Daten-

bank 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. Dort ist das Cachen der Assets nicht notwendig, da das Ergebnis eine auf dem Mobilgerät installierte App, mitsamt der Dateien die zur Ausführung notwendig sind, ist.

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 Netzwerkübertragung 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. Während 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. Während 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 repräsentieren die Szenarien und die Rechtecke die daraus resultierenden Fälle.

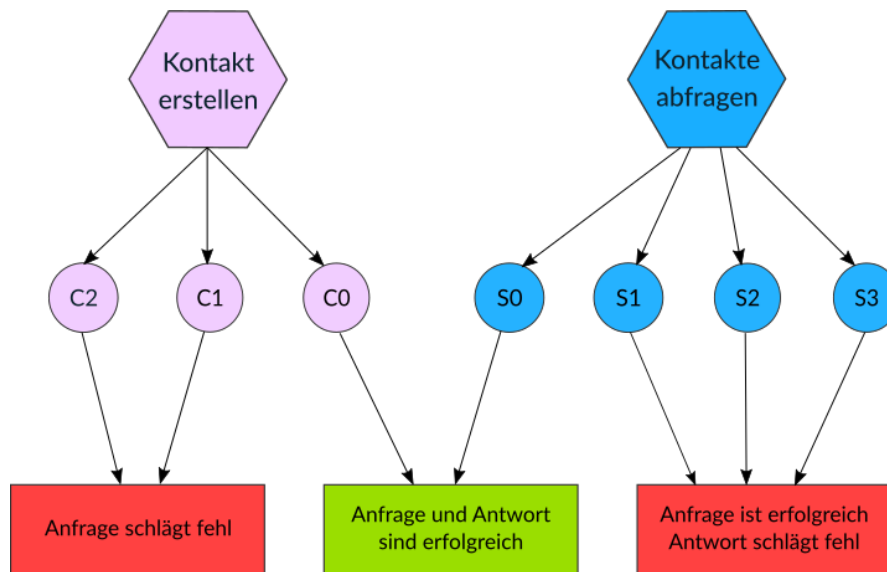


Abbildung 4.1: Szenarien bei der Datenübertragung über das Netzwerk

ERGEBNIS

Da die Szenarien C0 und S0, die Szenarien C1 und C2 sowie die Szenarien 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) Operationen 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**⁸ 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 Schlüssel⁹ 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 Versionsnummer '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 Version gibt.

⁸Eine berühmte Person

⁹Ein 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@enterprise.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 Problemen¹⁰ 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 entwickelt.

Zur Versionierung eines Adressbucheintrags wird eine Logische Uhr¹¹ 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-

¹⁰<https://support.google.com/accounts/answer/185834?hl=en#sync>

¹¹Eine 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 berichtet 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 Versionen mit einem Eintrag '5560348cec1b08c3d53e1508b4a46868' zugeordnet. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst, während dieser offline ist. Im selben Status berichtet 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 unterschiedlichen 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. Nach der konkreten Beschreibung dessen, was diese Arbeit umfassen soll, folgen die aus den oben genannten Szenarien hergeleiteten Anforderungen, die eine offlinefähige Anwendung erfüllen soll.

5.1 UMFANG

Das Ziel dieser Arbeit ist die Untersuchung des Konfliktmanagements offlinefähiger Technologien. Dazu soll die beispielhafte Anwendung eines kollaborativen Adressbuchs 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.

Beim 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 Adress-

bucheintrags. 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 mit zu entwickeln. 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 2 Geräte

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. *Wichtig hierbei ist die Möglichkeit immer zu einem konfliktfreien Status zu gelangen – unabhängig davon wie viele Konflikte es gibt.*

Jeder Fehlerfall muss kommuniziert werden. Wenn es konfliktbehaftete 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 funktionieren.
F2	Die Anwendung soll fähig sein den Netzwerkstatus zu ändern.
F3	Die Anwendung muss fähig sein die Kontakte unabhängig vom Netzwerkstatus zu laden, sofern diese einmal aus dem Netzwerk geladen wurden.
F4	Die Anwendung muss die Möglichkeit bieten, unabhängig vom Netzwerkstatus einen Kontakt anzulegen, zu bearbeiten oder zu löschen.
F5	Die Anwendung muss alle Kontakte sowohl lokal als auch serverseitig persistieren, identifizieren können und versionieren.
F6	Die Anwendung muss die lokal gespeicherten Kontakte mit denen auf der Serverdatenbank persistierten synchronisieren, sobald die Anwendung mit dem Internet verbunden ist.
F7	Die Anwendung muss die Möglichkeit bieten die Konfliktmanagementstrategien der zu untersuchenden Technologien zu evaluieren.
F8	Die Anwendung soll Konflikte speichern, sofern diese auftreten.
F9	Die Anwendung muss die Möglichkeit bieten die Konflikte zu lösen, sofern diese auftreten.
F10	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 Entwicklungsperspektive 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 ich nur die Adressbucheinträge oder deren Aktualisierungen laden, die sich nicht schon auf dem Endgerät befinden, um Datentráfico und Ladezeiten zu sparen.
D3	Ich als EntwicklerIn möchte ich 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 implementieren 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 <i>können</i> wenn 1. beide Client und Server online sind, 2. entweder Client oder Server offline ist oder 3. beide Parteien offline sind. daraus 3 Stories machen? oder nur testen online-offline?
T6	Ich als TesterIn möchte die Testfälle detailliert dokumentieren <i>können</i> , um sie auswerten zu können.

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.

Konflikt bei Kontakt **Amilia Pond.**
Welche Version möchtest du behalten?

Telefonnummer: "0305547925" <<< **Andere Änderung**

Telefonnummer: "0305549725" <<< **Deine Änderung**

Abbildung 5.1: Dialogfenster im Konfliktfall

Wurde beispielsweise Amalias 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 Telefonnummer kann entschieden werden welche Version die richtige ist und behalten wird.

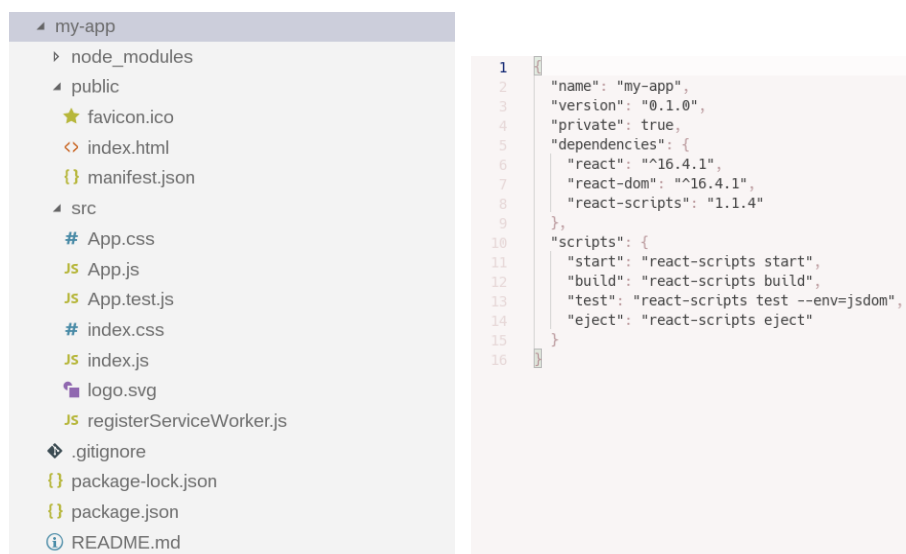
6 KONZEPTION

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 Philosophie. 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. Zur Veranschaulichung wird anhand des Listings 6.1 eine beispielhafte `React` Komponente beschrieben. Sie repräsentiert ein Formular in dem der Name des Kontakts angezeigt und geändert werden kann.

Die Formularkomponente 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` geladen (Zeile vier). So kann das Vorfüllen der Eingabefelder realisiert werden.

In Zeile sieben ist die `handleChange()` Funktion, die am Eingabefeld (Zeile 22) auf die Änderungen reagiert (Zeile zehn) und den interne `state` aktualisiert (Zeile 12).

```
1 class ContactForm extends Component {
2   constructor (props) {
3     super(props)
4     this.state = { contact: props.contact }
5   }
6
7   handleChange (type, event) {
```

```
8   let contact = this.state.contact
9   if (type === 'name') {
10     contact = {...contact, name: event.target.value}
11   }
12   this.setState(() => ({contact: contact}))
13 }
14
15 render () {
16   return (
17     <form onSubmit={this.props.addOrEditContact(this.state.contact)}>
18       <label>Name</label>
19       <input
20         value={this.state.contact.name}
21         onChange={this.handleChange.bind(this, 'name')} />
22
23       <button onClick={this.props.handleCancel}>Cancel</button>
24       <button type='submit'>Save</button>
25     </form>
26   )
27 }
28 }
29 export default ContactForm
```

Listing 6.1: Limitierte Version der React Komponente *ContactForm* beider Prototypen

In der Elternkomponente `Contacts` wird die Formularkomponente so wie es im folgenden Listing zu lesen ist, aufgerufen. Alle im Formular verfügbaren Daten werden hier übergeben.

```
<ContactForm contact={editView.contact}
  addOrEditContact={this.addContact}
  handleCancel={this.toggleEdit} />
```

Listing 6.2: Aufruf der React *ContactForm* Komponente

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 createStore in Zeile fünf erzeugt. Ein store enhancer ist eine Funktion die den createStore neu zusammenfügt und einen neuen, erweiterten createStore zurückgibt. Redux Offline kommt mit einer Grundkonfiguration (siehe Zeile drei). Diese wird dem offline store enhancer in Zeile acht übergeben.

```
1 import { createStore, compose } from 'redux'
2 import { offline } from '@redux-offline/redux-offline'
3 import offlineConfig from '@redux-offline/redux-offline/lib/defaults'
4
5 const store = createStore(
6   reducer,
7   compose(
8     offline(offlineConfig)
9   )
10 )
```

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 ausgeführt 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       console.log('started to fetch contacts ')
5       return state
6
7     case FETCH_CONTACTS_COMMIT:
8       console.log('successfully fetched contacts ')
9       if (state === action.payload) return state
10      return action.payload
11
12    case FETCH_CONTACTS_ROLLBACK:
13      console.log('failed to fetch contacts')
14      return state
15  }
```

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 sieben bis zehn 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 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.

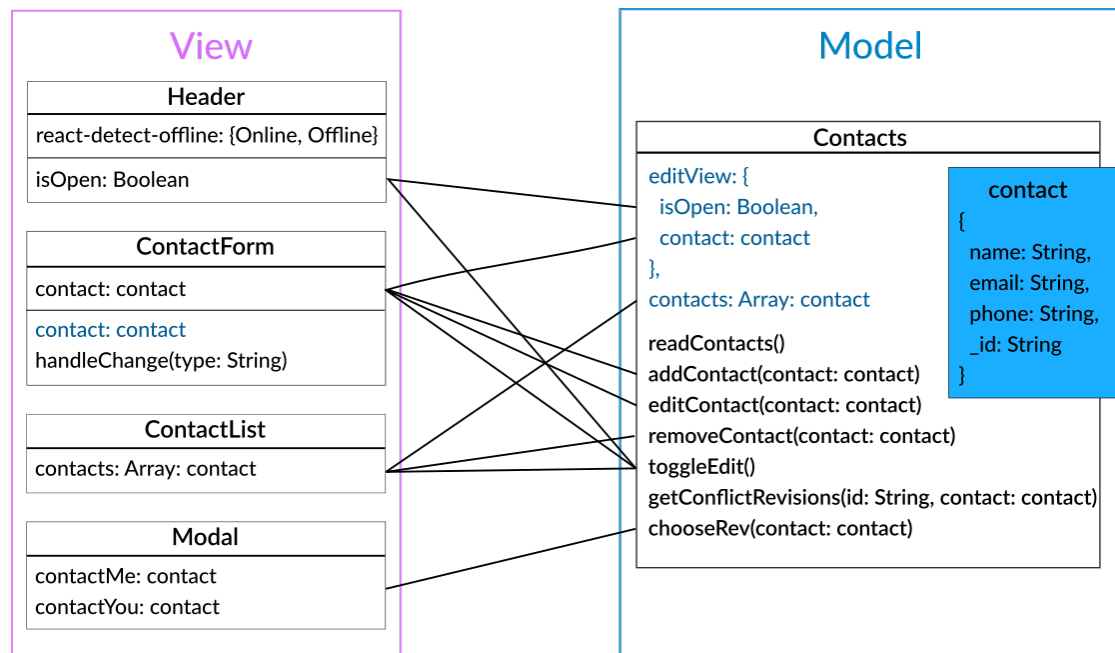


Abbildung 6.2: Komponentendiagramm der Prototypen

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 Kom-

ponente 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 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.

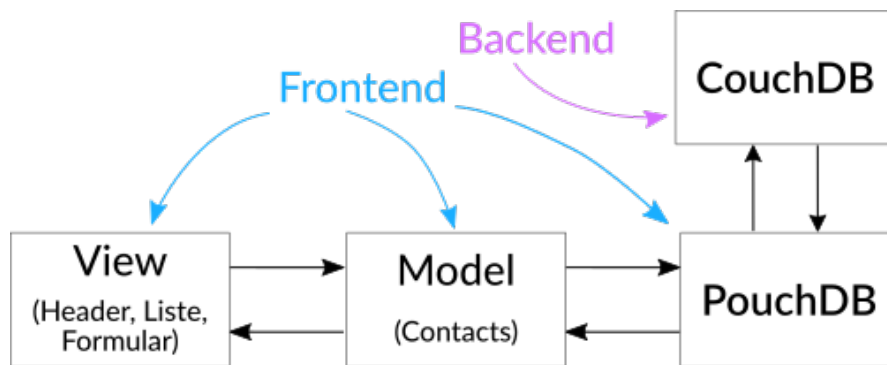
Die Komponente `ContactForm` zeigt, sofern vorhanden, alle im Kontakt gespeicherten Daten an. Diese können hier bearbeitet werden. Gibt es keine Kontaktdaten die geladen werden kann, kann hier ein neuer Kontakt angelegt werden. Zusätzlich zu den Eingabefeldern für jedes Kontaktattribut hat sie zwei Knöpfe mit denen die Aktion bestätigt oder abgebrochen werden kann. Für Ersteres hat sie Zugriff auf die Funktionen `addContact` und `editContact`, für Letzteres kann sie `toggleEdit()` aufrufen.

Sie ist neben `Contacts` die einzige Komponente mit einem internen `state`. Dieser wird für die Ereignisbehandlung benötigt, die auf die Veränderung der einzelnen Eingabefelder zu "lauschen". Der zu bearbeitende Kontakt wird hier zwischengespeichert und dann als Ganzes an `Contacts` gegeben.

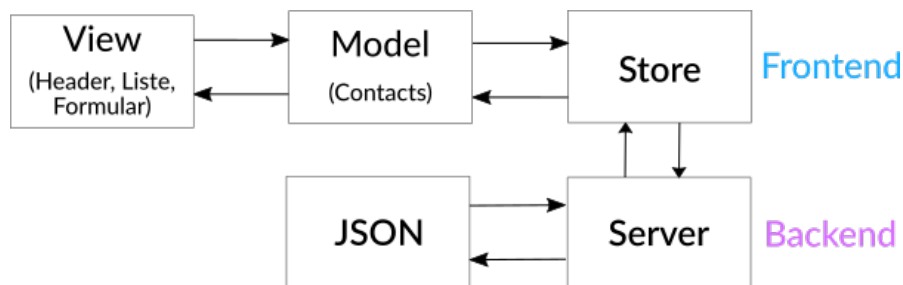
Redux Offline???

Kommt es beim Lesen aller Kontakte oder 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 Konfliktdialog 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 ausgewählten 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.

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

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 nicht implementiert. Deswegen werden nach jeder Aktion, die Kontakte vom Server geladen.

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 benötigten Funktionen auf und zeigt die notwendigen Schritte zur Synchronisation der lokalen PouchDB und CouchDB.

Die lokale Datenbank wird in Zeile eins 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 zwei 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 vier erforderlich. Im optionalen Parameter können zum Beispiel Filter oder Einstellungen zum wiederholten Synchronisationsversuch im Falle eines Fehlschlags gesetzt werden [poud].

```
1 const localDB = new PouchDB('contacts')
2 const remoteDB = new PouchDB('http://localhost:5984/contacts')
3
4 localDB.sync(remoteDB, [options])
5
6 // create a contact
7 localDB.put({
8   ...contact,
9   _id: new Date().toISOString()
10 })
11 // update a contact
12 localDB.get(contact._id).then(function (doc) {
13   doc = {...contact}
14   return localDB.put(doc)
15 })
16 // get all contacts
17 localDB.allDocs({
18   include_docs: true,
19   conflicts: true
20 })
21 // remove a contact
22 localDB.remove(contact)
```

Listing 6.6: Persistierung der Daten mit PouchDB und CouchDB

Die Zeilen sieben bis zehn zeigen wie ein Kontakt erzeugt werden kann. Bevor das geschieht wird die ID gesetzt. PouchDB bietet zur Erstellung von Objekten auch `localDB.post()` an. Bei dessen Verwendung wird `_id` von PouchDB automatisch generiert. Diese Variante wird jedoch nicht empfohlen, weil dann die IDs zufällig sind, die Objekte nicht danach sortiert werden können [poub].

Das Aktualisieren eines Kontakts sieht ähnlich aus. Zuerst wird der entsprechende Kontakt wie in Zeile zwölf aus der Datenbank angefragt um dann in der Datenbank aktualisiert zu werden. Mit jedem Update bekommt ein Kontakt von PouchDB eine neue Revision.

Der Aufruf `localDB.allDocs()` in Zeile 17 fragt alle in der lokalen Datenbank gespeicherten Kontakte an. Ohne den Parameter `include_docs: true` werden nur die `_id` und die `_rev` Eigenschaften eines jeden gespeicherten Kontakts zurückgegeben. Ist die Option `conflicts` auf `true` gesetzt, werden unter dem Attribut `_conflicts` Konfliktinformationen zu jedem Kontakt gespeichert.

Man kann einen Kontakt in PouchDB wie in Zeile 22 mittels `localDB.remove(contact)` löschen. Der Kontakt ist dann 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.

```
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      }
14    }
15  }
```

```
16 }
17
18 // reducer
19 function contacts (state=[], action) {
20   switch (action.type) {
21     case ADD_CONTACT:
22       console.log('started to add contact')
23       return [...state, action.contact]
24
25     case ADD_CONTACT_COMMIT:
26       console.log('successfully added contact')
27       return [...state, action.payload]
28   }
29 }
```

Listing 6.7: Speicherung der Daten mit Redux Offline

Mit dem Aufruf der Aktion `ADD_CONTACT` wird der Vorgang gestartet einen Kontakt hinzuzufügen. Die anzufragende URL ist im `meta.offline.effect` Feld in Zeile neun festgelegt. Die Anfrage geht an den Server, welcher die Daten in der JSON Datei persitiert. 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 25 behandelt. Der `state` wird mit der Antwort vom Server aktualisiert und die Synchronisation ist vollzogen.

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    })
17  }
18 })
```

```
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 integrierten Dateisystems¹² 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 Modul 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].

¹²siehe 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

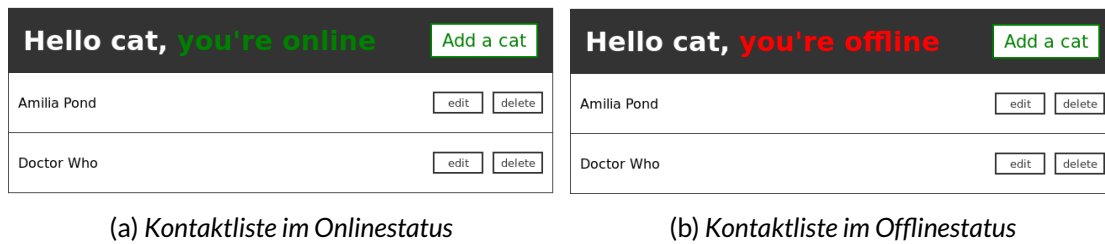


Abbildung 6.5: Die Kontaktliste in beiden Netzwerkstatus

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 Kontakts 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.

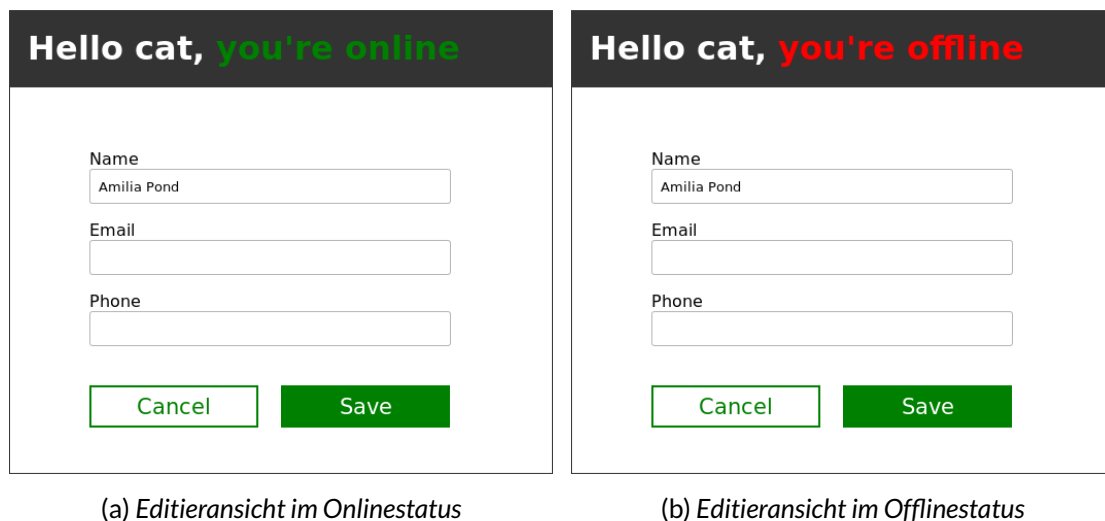


Abbildung 6.6: Die Editieransicht in beiden Netzwerkstatus

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 oder 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. In der zweiten Testreihe wird ein Browser für eine Aktion vom Internet getrennt indem er in den Offlinemodus geschaltet 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.

Jede Testreihe wird einmal, in unterschiedlichen Netzwerkstatus der Anwendung durchgeführt, pro Reihe gibt es drei bis vier Testdurchläufe. In den Testreihen eins und drei wird jeder genannte Testdurchlauf genau einmal durchgeführt. Für die zweite Testreihe kann es, je nachdem in welchem Browser 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	löschen			löschen
2d	bearbeiten			löschen
3a		anlegen		anlegen
3b		bearbeiten		bearbeiten
3c		löschen		löschen
3d		bearbeiten		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. **Das mit dem erwartetem Ergebnis ist komisch, denn ich weiß ja was rauskommt weil ich es implementiert habe...**

7 WIP IMPLEMENTIERUNG DER PROTOTYPEN

In diesem Kapitel wird nach dem in Kapitel 6 präsentierten Lösungsweg die detaillierte Beschreibung der technischen Realisierung der Prototypen vorgestellt.

Nach der Beschreibung der Konfigurationsdateien wird auf die Umsetzung der Szenarien eingegangen. Im Zuge dessen werden die implementierten Algorithmen vorgestellt, wobei sich der erste mit dem Auffinden der jeweilig nächsten relevanten Ampel befasst und der zweite die jeweils empfohlene Geschwindigkeit berechnet.

7.1 WIP PACKAGE.JSON?

Vergleichen? build Vorgang? Ich glaube das würde ich rauslassen (habe mich hier an anderen Arbeiten orientiert)

7.2 WIP HAUPTMODUL CONTACTS

wird beim Start ausgeführt

Stellt zentrale Funktionen bereit... Erklärung

7.3 WIP KONFLIKTMANAGEMENT

7.4 WIP UMSETZUNG DER SZENARIEN? ANFORDERUNGEN?

Umsetzung der Funktionalität?

Wie wurde CRUD implementiert?

Wie kann ein Konflikt erzeugt werden?

(manuell – siehe 6.2.2)

7.5 WIP INSTALLATIONSANLEITUNG

build und so Beide entwickelten Prototypen sind als öffentliche Repositories auf GitHub¹³ zu finden. Um sie zu installieren müssen folgende Schritte ausgeführt werden.

7.5.1 AMILIA-QOUCH

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 muss man in das Verzeichnis navigieren und alle Abhängigkeiten installieren.

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

3. Mittels

```
npm start
```

wird die Anwendung gestartet.

7.5.2 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 `amilia-qouch` Anleitung

3. Mittels

```
npm run server
npm start
```

wird zuerst der Server, dann die Anwendung gestartet.

7.6 WIP TESTFÄLLE

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

¹³Software-Entwicklungs-Plattform <https://github.com/>

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* 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 Konfliktforcierung erforderlich.

8 WIP EVALUATION

Welche Anforderungen wurden erfüllt? alle funktionalen. Die nichtfunktionalen werden getestet

8.1 WIP TEST KONFLIKTMANAGEMENT

Diese Tests wurden x oft durchgeführt. Kontakt anlegen:

- online – online
- offline – online
- offline – offline
- trennen zwischen selben Feldern oder immer nur selbes Attribut bearbeiten?

Dasselbe für bearbeiten, löschen, anlegen und bearbeiten, bearbeiten und löschen, anlegen und löschen

8.2 WIP AUSWERTUNG

8.2.1 WIP IMPLEMENTIERUNGSAUFWAND

Setup der App:

Irrelevant und identisch wegen Create React App.

Einbindung der Technologien:

PouchDB und CouchDB: gering, nur Installation und Anwendung der Pouch API
Neue Instanz, Synchronisation (1 Zeile) und CRUD

Redux Offline: komplex, da Redux eingesetzt und verstanden werden muss. Für komplexere Apps mag das generell besser sein, aber für den einfachen Anwendungsfall zu viel Overhead. Einbindung von Redux Offline war allerdings wenig aufwändig sobald Redux implementiert war.

Lesbarkeit des Codes:

PouchDB und CouchDB: nicht mehr als erwartet und auch verständlich wenn man mit den Technologien nicht vertraut ist. Es liest sich so wie man es verwendet dank unmissverständlicher API.

Redux Offline: Auch um den Code ohne Probleme lesen zu können muss man mit Redux vertraut sein. Ist das der Fall, ist trotzdem ein Blick in die Dokumentation von Redux Offline hilfreich, weil die Metaattribute nicht für sich selbst sprechen (effect, commit, rollback).

8.2.2 WIP OFFLINEFUNKTIONALITÄT

OFFLINEFÄHIGKEIT AUF EINEM GERÄT

OFFLINEFÄHIGKEIT AUF ZWEI GERÄTEN

8.2.3 WIP KONFLIKTMANAGEMENT

9 WIP ZUSAMMENFASSUNG UND AUSBLICK

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

ABKÜRZUNGEN

API	Application Programming Interface
App	Applikation
CAP	Consistency Availability Partition tolerance
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.

Hashfunktion

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, Quelle: [canb]	9
2.2	Browserkompatibilität für Web Storage, Quelle: [cand]	10
2.3	Browserkompatibilität für IndexedDB, Quelle: [cana]	11
2.4	Browserkompatibilität für IndexedDB 2.0, Quelle: [cana]	12
2.5	Das CAP Theorem	15
3.1	Redux Datenfluss	22
3.2	Redux Offline	24
3.3	HOODIE Architektur	26
4.1	Szenarien bei der Datenübertragung über das Netzwerk	31
5.1	Dialogfenster im Konfliktfall	40
6.1	Create React App: initiale Testapplikation	42
6.2	Komponentendiagramm	47
6.3	Client-Server-Modell des Prototypen <i>amilia-qouch</i>	49
6.4	Client-Server-Modell des Prototypen <i>amilia-rdx</i>	49
6.5	Die Kontaktliste in beiden Netzwerkstatus	55
6.6	Die Editieransicht in beiden Netzwerkstatus	55

QUELLCODEVERZEICHNIS

2.1	Beispiel einer HTML-Datei mit einer Manifest-Attribut Einbindung	8
2.2	Beispiel einer <code>.appcache-Datei</code>	8
3.1	Beispiel React Native Offline Aktion	25
6.1	Limitierte Version der React Komponente <code>ContactForm</code> beider Prototypen	43
6.2	Aufruf der React <code>ContactForm</code> Komponente	44
6.3	Erstellen eines Stores mit Redux Offline	45
6.4	Aktion <code>fetchContacts</code> mit Metaattribut	45
6.5	Reducer mit allen Aktionen die im Meta Feld beschrieben werden	46
6.6	Persistierung der Daten mit PouchDB und CouchDB	50
6.7	Speicherung der Daten mit Redux Offline	51
6.8	Mögliche Serverimplementierung für das Hinzufügen eines Kontakts	52
6.9	Beispiel einer React Detect Offline Implementierung	53

LITERATURVERZEICHNIS

- [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
- [ALS10] ANDERSON, J. C.; LENHARDT, Jan; SLATER, Noah: *CouchDB: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, 2010. – ISBN 978-0-596-15589-6 2.2.1, 2.2.2, 2.2.7
- [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
- [AW04] ATTIYA, Hagit; 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, 9
- [canb] *Can I use Service Workers?* <https://caniuse.com/#feat=serviceworkers>,.- Zugriff: 06.05.2018 2.1, 9

- [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, 9
- [coua] *CouchDB - relax.* <https://couchdb.apache.org/>, . – Zugriff: 12.04.2018 2.1.3
- [coub] *CouchDB Replication Protocol.* <http://docs.couchdb.org/en/master/replication/protocol.html>,. – Zugriff: 12.04.2018
- [EG89] ELLIS, C. A. ; 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 2.2.4
- [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. 2.2.4
- [Gau18] GAUNT, Matt: *Service Workers: an Introduction.* <https://developers.google.com/web/fundamentals/primers/service-workers/>, 2018. – Zugriff: 06.05.2018 2.1.1
- [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
- [lwa16] 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
- [LL10] LI, Du ; 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 2.2.4
- [loc] *localStorage – 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 2.1.3
- [poub] *API Reference – Create/update a document.* <https://pouchdb.com/guides/updating-deleting.html#why-must-we-dance-this-dance>,. – Zugriff: 24.06.2018 6.2.1
- [pouc] *pouchdb – Conflicts.* <https://pouchdb.com/guides/conflicts.html>,. – Zugriff: 12.04.2018
- [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

- [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
- [SPBZ11a] SHAPIRO, Marc ; PREGUIÇA, Nuno ; BAQUERO, Carlos ; 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. 2.2.5
- [SPBZ11b] SHAPIRO, Marc ; PREGUIÇA, Nuno ; BAQUERO, Carlos ; ZAWIRSKI, Marek: Conflict-free Replicated Data Types. Version: Jul 2011. <https://hal.inria.fr/inria-00609399>. 2011 (RR-7687). – Research Report. – 18 S. 2.2.5, 2.2.5
- [Sto] STOLYAR, Arthur: *offline-plugin*. <https://github.com/NekR/offline-plugin>, . – Zugriff: 12.04.2018 3.1
- [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 2.2.3

ANHANG

EIDESSTATTLICHE ERKLÄRUNG

CD-INHALT

Auf der beigefügten CD befinden sich

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