BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Masterarbeit

Medieninformatik
Fachbereich VI – Informatik und Medien

Untersuchung der Konfliktmanagementstrategien verschiedener offlinefähiger Systeme

---

Berlin, den 24. November 2018

Autorin:
Jacoba Brandner
Matrikelnummer:
833753

Betreuer:
Herr Prof. Dr. Hartmut Schirmacher
Gutachterin:
Frau Prof. Dr. Petra Sauer

Inhalt

1 Einführung **5**

1.1 Motivation 5

1.2 Zielstellung 5

2 Bestehende offlinefähige Systeme / Konzepte **6**

2.1 Kollaborative Software 6

2.1.1 Google Docs 6

2.1.2 Google Wave 6

2.1.3 Dropbox / 'Clouds' 6

2.1.4 Kollaborative Editoren 6

2.2 Offline-First Frameworks/Bibliotheken 7

2.2.1 redux? react-native? webpack? 7

2.2.2 Realm 7

2.2.3 Datomic? (Closure) 8

2.2.4 Redux Offline 8

2.2.5 react-native-offline 10

2.2.6 offline-plugin für webpack 10

2.2.7 hoodie 11

2.2.8 CouchDB 11

2.2.9 PouchDB 12

2.2.10 Couchbase 12

3 Grundlagen **13**

3.1 Konflikte 13

3.1.1 Consistency Availability Partition tolerance (CAP) Theorem? 14

3.2 Offline First 14

3.2.1 Anforderungen an Offline-First / PWA? 15

3.3 Progressive Web Apps? 15

3.3.1 ServiceWorker? 15

3.3.2 localForage || AsyncStorage? 15

3.3.3 IndexedDB? 15

3.4 Replikation in verteilten Systemen 15

3.4.1 Last-Write-Wins (LWW) || Blockieren? 15

3.4.2 Operational Transformation 16
3.4.3 Conflict-free replicated data type 18
3.5 Das CouchDB Replikationsmodell 19
3.5.1 schlussendliche Konsistenz 20
3.5.2 Replikation? 20
3.5.3 Konfliktmanagement 20

4 Vorgehen **21**

5 Fazit **22**

Abkürzungen **23**

Glossar **24**

Abbildungsverzeichnis **25**

Literaturverzeichnis **25**

Anhang **28**

1 Einführung

1.1 Motivation

Heutzutage haben mehr als drei Milliarden Menschen Zugang zum Internet [Ban16].
Es herrscht die weit verbreitete Annahme, dass die Erreichbarkeit dieser Menschen sowohl standortgebunden als auch mobil über das Internet stets gewährleistet ist.
Doch instabile Datenverbindungen sind ein allgegenwärtiges Problem. Während der Fahrt durch Tunnel, bei Stromausfällen, auf Großveranstaltungen oder sogar bei einem Aufenthalt in einer beliebten Urlaubsregion kann es zu Verbindungsproblemen kommen [**?**, **?**].

> 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]

Aus diesem Grund werden unter dem Begriff Offline First immer mehr Anwendungen entwickelt, die offline funktionieren [**?**]. Gleichzeitig gibt es mehr Technologien, die die Entwicklung dieser Anwendungen erleichtern.
Die Anforderungen an eine alleinstehende, offlinefähige Anwendung sind weniger hoch, als die an eine kollaborative, offlinefähige Anwendung. Sobald eine Anwendung auf mehreren Geräten läuft, können durch paralleles Arbeiten Konflikte entstehen. In dieser Arbeit sollen Konfliktmanagementstrategien der Technologien, die die Entwicklung offlinefähiger Anwendungen unterstützen, untersucht werden, um EntwicklerInnen die Entscheidung zu erleichtern, welche dieser Systeme sich für die Erstellung einer Offline First–Applikation (App) am besten eignet.

1.2 Zielstellung

Ziel dieser Masterarbeit ist es, offlinefähige Prototypen unter Verwendung der zu untersuchenden Systeme zu erstellen, um das Verhalten bei Konflikten zu evaluieren. Die Arbeit soll den Blick auf die Voraussetzungen für offlinefähige Anwendungen schärfen und so die Wahl der zu verwendenden Technologien für die Entwicklung dieser Apps erleichtern. Der Fokus liegt neben der Offlinefunktionalität der Anwendung auf dem Konfliktmanagement der benutzten Technologie. Beim Umgang mit Konflikten ist es wichtig, dass keinerlei Daten verloren gehen. Ein weiteres Kriterium ist der Implementierungsaufwand, der für die Einbindung und Verwendung der zu untersuchenden Technologie aufgebracht werden muss.

Zur Untersuchung der Konfliktmanagementstrategien offlinefähiger Systeme soll eine beispielhafte Anwendung betrachtet werden, ein offlinefähiges Adressbuch, welches mehrere Personen benutzen können. Des Weiteren werden bestehende Konfliktmanagementstrategien sowie verschiedene Technologien vorgestellt, die die Entwicklung offlinefähiger Anwendungen unterstützen.
Für die Evaluation der zu entwickelnden Anwendung werden manuelle Tests durchgeführt, auf denen die Auswertung der oben genannten Aspekte beruht.

2 Grundlagen

Eine offlinefähige Anwendung ist sowohl mit, als auch ohne Internetverbindung einsatzbereit. Dieses Kapitel beschreibt die grundlegenden Optionen, eine Anwendung offlinefähig zu machen und 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 Apps existieren und funktionieren grundsätzlich solange offline, bis sie versuchen online Daten abzurufen. Um Webanwendungen offlinefähig zu machen, sind einige Grundvoraussetzungen erforderlich.
Anders als native Apps werden Webanwendungen nicht auf dem Gerät installiert, sondern in der Regel über einen Webbrowser genutzt. 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. Einmal gespeichert, ist das Laden der Ressourcen aus dem Cache, auch bei bestehender Internetverbindung, schneller, als aus dem Netz [**?**]. Daten, die zuerst auf dem Endgerät gespeichert werden, gehen auch bei plötzlichen Verbindungsverlust nicht verloren.
Gibt es eine serverseitige Datenhaltung, müssen die Daten zwischen Server und Client synchronisiert werden. Ist die Anwendung kollaborativ, muss die Synchronisation zwischen allen Beteiligten stattfinden.

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

2.1.1 Speichern der Assets

Das Cachen der Assets ist der erste Schritt, 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 [?].

App Cache

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

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

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

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

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

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

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

Alle Seiten mit dem manifest-Attribut und die, die explizit in der Textdatei beschrieben wurden, wurden vom Browser gespeichert [?].
In der Praxis jedoch zeigten sich zahlreiche Probleme mit dem App Cache. So wurde der Application Cache nur aktualisiert, wenn sich der Inhalt des Manifests geändert hat. Dann mussten alle Dateien neu heruntergeladen werden. Änderungen des Manifests oder anderer Dateien bedeuteten nicht zwingend eine erneute Speicherung dieser Dateien. Denn wenn der Server zusammen mit den Dateien keine Cache-Header mit einem Gültigkeitsdatum sendete, speicherte der Browser die Datei nach dem bereits gespeicherten Gültigkeitsdatum. So konnte es passieren, dass der Browser annahm, eine Datei brauche keine Aktualisierung und weiterhin die alte, gecachte Version auslieferte [?].
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 [?].
Gegenwärtig besitzen – bis auf den Internet Explorer – sämtliche Desktop-Browser, und alle gängigen mobilen Browser eine Unterstützung für Service Worker.

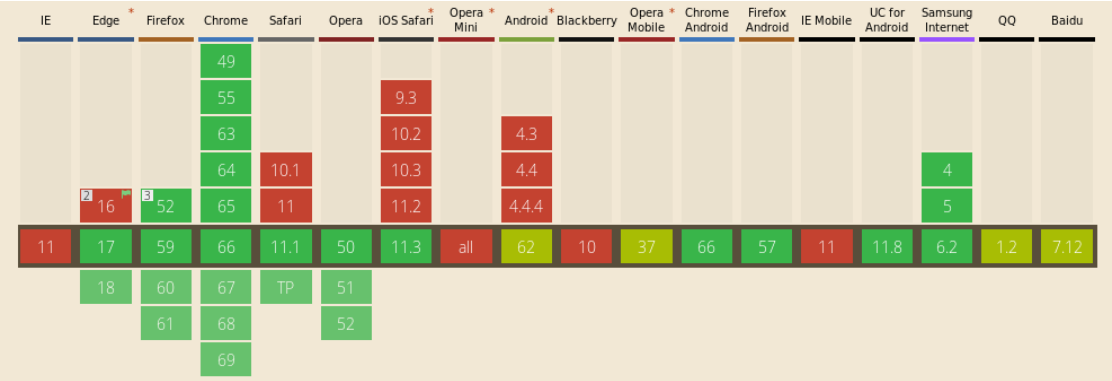


Abbildung 2.1: *Browserkompatibilität für Service Worker, Quelle: [?]*

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

Video, später bzw. offline anzuschauen. Diese werden dann im Cache gespeichert und sind somit offline verfügbar. Service Worker erlauben außerdem den Zugriff auf Push-Benachrichtigungen und das Background Sync Application Programming Interface (API). Die Hintergrundsynchronisation kann einmalig oder in einem festgelegten Intervall stattfinden und ist besonders für nicht dringende Aktualisierungen wertvoll [**?**].

2.1.2 Speichern der generierten Daten

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

Web Storage

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

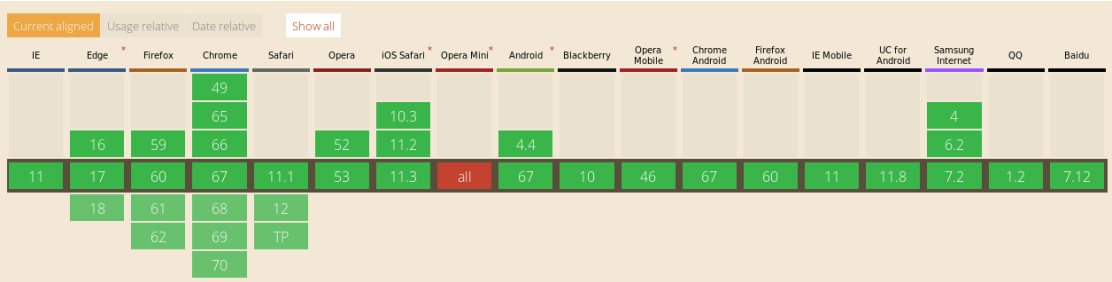


Abbildung 2.2: *Browserkompatibilität für Web Storage, Quelle: [?]*

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

Web SQL

Eine andere Form 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 an unabhängigen Implementierungen wie z.B. eine andere Datenbank (DB) als SQLite im Backend, wurde es abgelehnt [**?**].
Das Web SQL API wird nur von den webkitbasierten Browsern unterstützt, also nicht von Firefox, dem Internet Explorer oder dessen neueren Variante Edge [**?**].

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 erlaubt Datenbankabfragen, die keinen anderen Prozess blockieren [**?**].

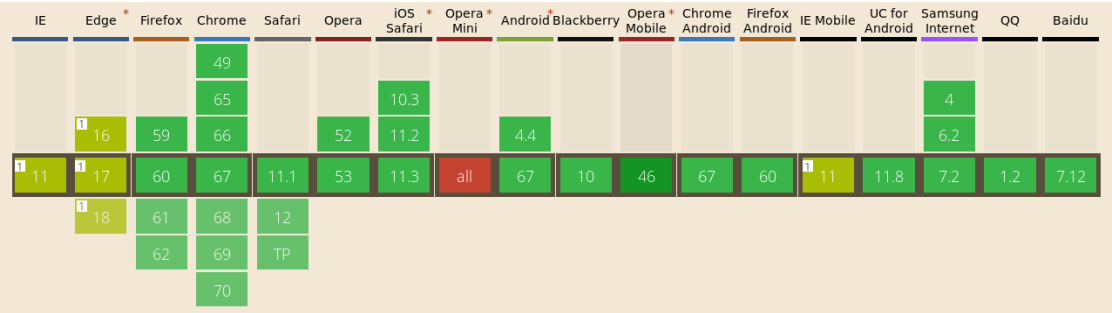


Abbildung 2.3: *Browserkompatibilität für IndexedDB, Quelle: [?]*

Wie in Abbildung **??** 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 [**?**]. Die Aktualität dieser Spezifikation spiegelt sich in der Browserunterstützung wider. Die **??** zeigt, dass die zweite Version nur von den gängigen Desktopbrowsern und einigen mobilen Browsern unterstützt wird. Die Unterstüzung von IndexedDB 2.0 in Edge wird von Microsoft geplant [**?**]. IndexedDB wird stetig weiterentwickelt und es gibt bereits einen Spezifikationsentwurf für die dritte Version [**?**].

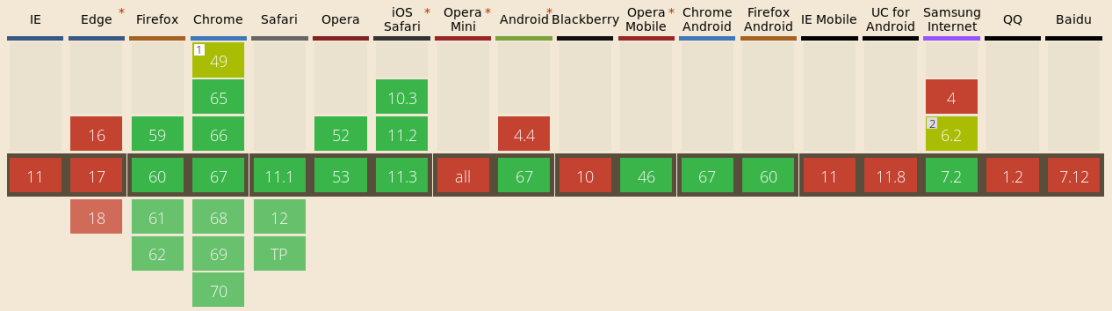


Abbildung 2.4: *Browserkompatibilität für IndexedDB 2.0, Quelle: [?]*

2.1.3 Datenbanksynchronisation

Eine Datenbanksynchronisation ist dann vonnöten, wenn die Anwendung auf mehr als einem Gerät laufen soll. Idealerweise wird hierfür eine Datenbank gewählt, die einen Synchronisationsalgorithmus bereitstellt. Im Allgemeinen ist ein Synchronisationsprotokoll die Möglichkeit für zwei Parteien, beispielsweise Client und Server, den Zustand voneinander zu kennen. Der Client schickt seine Daten an den Server und umgekehrt, solange bis beide Parteien denselben Zustand haben. Leider sind Synchronisationsprotokolle schwer zu implementieren und führen häufig zu einem frustrierenden Ergebnis: Dokumente oder Fotos werden nicht repliziert, es gibt doppelte oder verlorene Daten und alle möglichen Arten von Fehlverhalten bei kollaborativen Anwendungen [**?**].
In Kapitel 2 werden einige Technologien vorgestellt, die einen integrierten Synchronisationsmechanismus besitzen.

2.2 Konfliktmanagementstrategien

Diese Arbeit untersucht die Konfliktmanagementstrategien offlinefähiger Systeme. Nachdem in **??** 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 bezeichnet, in der verschiedene Versionen des gleichen Dokuments, also ein ein Datenobjekt oder eine Datei, auf mehreren Geräten oder Datenbanken gespeichert sind (vgl. [ALS10] S. 153). Konflikte gehören in verteilten Systemen zur Realität und lassen sich nicht vermeiden. Ein verteiltes System ist per Definition ein Zusammenschluss unabhängiger Computer, die sich für die NutzerInnen als ein einziges

System präsentieren (vgl. [**?**] 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 zur 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, wenn Daten über das Netzwerk übertragen werden und ein solches Szenario eintritt. Anhand des folgenden Beispiels wird ein mögliches Fehlerszenario für den Fall des unzuverlässigen Netzwerks aufgegriffen.
Zwei Personen treffen sich im Zug und verstehen sich auf Anhieb sehr gut. Person A, nennen wir sie Amilia, gibt Person B, nennen wir sie Rory, ihre Telefonnummer. Der Zug fährt durch einen Tunnel und das Netzwerk bricht ab, als Rory Amilias Nummer in sein Adressbuch, das in Form einer App auf seinem Laptop gespeichert ist, schreibt. Amilia diktiert ihre Telefonnummer falsch, mit einem Zahlendreher, weil sie die Nummer noch nicht lange hat. Zur Sicherheit schickt Amilia Rory ihre Nummer zusätzlich per E-Mail. Rory schaltet seinen Laptop aus, weil er sich mit Amilia unterhalten möchte. Am Abend ist Rory zu Hause angekommen und er speichert Amilias Nummer aus der E-Mail in seinem Adressbuch auf seinem stationären Desktop PC. Jetzt gibt es Amilias Telefonnummer mit unterschiedlichen Informationen in Rorys Adressbuch auf verschiedenen Geräten. Wenn Rory am nächsten Tag seinen Laptop startet und das Adressbuch sich mit dem auf dem PC synchronisiert, entsteht ein Konflikt. Es gibt zwei unterschiedliche Versionen von Amilias Telefonnummer auf Rorys Geräten.

Konflikte sind in zwei Kategorien einzuteilen. Es gibt solche, die vom System selbst gelöst werden können und solche, 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 ist durch Systeme wie

Git[2], einer Software zur verteilten Versionsverwaltung, den meisten EntwicklerInnnen bekannt. Diese Art Konflikt kann selbstständig vom System gelöst werden.

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

2.2.2 Last-Write-Wins

Der Last-Write-Wins (LWW) Ansatz geht davon aus, dass Schreibvorgänge immer in der richtigen Reihenfolge ausgeführt werden. Der letzte an die Datenbank gesendete Schreibvorgang wird als korrekt angenommen. Diese Strategie ist leicht zu implementieren, da lediglich festgestellt werden muss, welche Manipulation die neuere ist. Das kann beispielsweise durch die Vergabe eines Zeitstempels, sofern alle Uhren synchron sind, problemlos errechnet werden [**?**].

2.2.3 Operational Transformation

Operational Transformation (OT) ist eine weit verbreitete Technik zur Unterstützung von Funktionalitäten in kollaborativer Software. Sie stammt aus einer 1989 veröffentlichten Forschungsarbeit und wurde ursprünglich nur für die gemeinsame Bearbeitung von Klartext-Dokumenten entwickelt [EG89]. Etwas später wurden einige Unvollständigkeiten im Algorithmus entdeckt und es wurden unabhängig voneinander verschiedene Lösungsvorschläge erarbeitet [**?**]. Heute unterstützt OT zusätzlich das kollaborative Bearbeiten von HTML, RTF und XML Dokumenten, Adobe Flash[3] Grafiken und Dokumenten in CAD[4] Tools wie Autodesk Maya[5]. Des Weiteren können mit OT Dateien die in Microsoft Office[6] enthalten sind, kollaborativ bearbeitet werden. Hierzu gehören Word–Dokumente, PowerPoint–Folien und Excel–Tabellen. Aber auch Dokumente in webba-

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

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

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

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

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

sierten Anwendungen werden unterstützt, wie Google Docs[7] und Google Wave[8] [**?**].

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

Um konkurrierende Operationen zu behandeln, gibt es die `transform` Funktion. Wenn ein Client die Änderungen von einem anderen empfängt, werden die Änderungen vor ihrer Ausführung transformiert. Sie nimmt zwei Operationen, die auf demselben Dokument, aber auf unterschiedlichen Geräten angewendet wurden und berechnet daraus eine neue. Die neue Operation wird dann nach der zweiten Operation angewendet. Die erste, beabsichtigte Operation wird beibehalten [EG89].
Die Grundidee des OT–Algorithmus wird anhand eines Beispiels, das die **??** veranschaulicht, illustriert.
Es gibt den initialen Text "abc", der auf den Geräten der kollaborativ arbeitenden Personen, Amilia und Rory, besteht. Amilia fügt lokal vor dem "abc" ein "x" ein, Rory löscht auf seinem Gerät den Buchstaben "c"'. Die konkurrierenden Operationen sind `insert(0, "x")`, "Füge das Zeichen "x" an Stelle Null ein", und `delete(1, 2)`, "lösche ein Zeichen an der zweiten Stelle". In OT werden lokale Änderungen angewandt, so wie sie passieren. Operationen über das Netzwerk werden, bevor sie auf zuvor ausgeführte Operationen angewandt werden, transformiert.
Wenn die Operation von Rory zuerst ausgeführt wird, gibt es kein Problem. Der Buch-

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

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

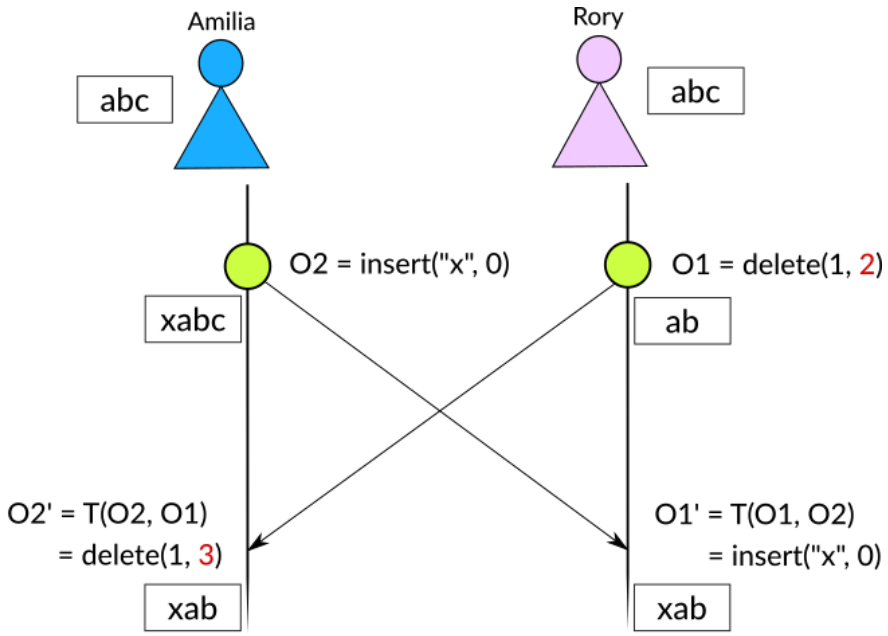


Abbildung 2.5: *Die Grundidee von OT*

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

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

2.2.4 Conflict-Free Replicated Data Type

Ein Conflict-free replicated data type (CRDT) ist eine spezielle Datenstruktur, die in verteilten Systemen auf mehreren Geräten repliziert werden kann. Jede Operation wird an alle Replikate gesendet. Jedes Replikat wendet alle ankommenden Aktualisierungen an und ein Algorithmus löst alle konfliktbehafteten Versionen auf, wodurch sichergestellt ist, dass Konflikte gar nicht erst auftreten [SPBZ11a]. Eine Synchronisation ist nicht notwendig, da jede Aktualisierung sofort auf dem Datenobjekt ausgeführt wird [SPBZ11b].

Umgesetzte CRDTs sind zum Beispiel Counter, Datenobjekte die zum Zählen verwendet werden. Es gibt einen, der nur hochzählen kann und eine andere Variante die auch die Subtraktion unterstützt. Weitere CRDTs sind Sets, eine Listenrepräsentation ohne Duplikate. Auch hier gibt es eine Variante, mit der nur Daten hinzugefügt werden können und eine, die auch das Entfernen der Daten erlaubt [SPBZ11a].

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

Zustandbasierter Ansatz

Wenn ein Replikat eine Aktualisierung von einem Client empfängt, wird sie zuerst auf den lokalen Status angewandt. Dann sendet es etappenweise eine Kopie des eigenen, aktualisierten Status an andere Replikate im System. Wenn ein Replikat den Status eines anderen Replikats empfängt, führt es mit einer `merge` Funktion den empfangenen Status mit dem lokalen zusammen. Entsprechend sendet dieses Replikat regelmäßig auch seinen Status an ein anderes Replikat, sodass jede Aktualisierung schließlich alle Replikate im System erreicht [SPBZ11b]. Die folgende Abbildung stellt eine zustandsbasierte Replikation mit drei Geräten dar.

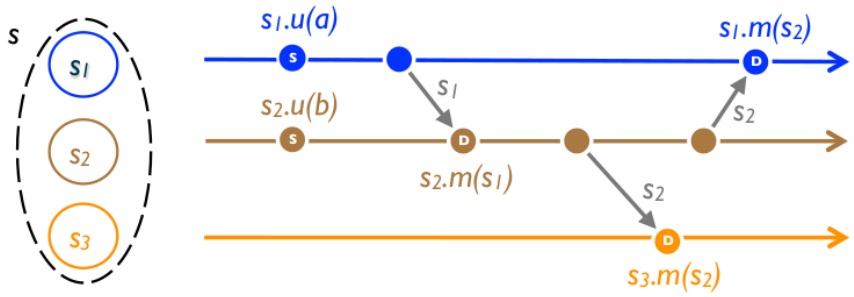


Abbildung 2.6: *Replikation von zustandsbasierten CRDTs, Quelle: [SPBZ11b]*

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

Damit die Replikation konfliktfrei funktioniert, müssen einige Voraussetzungen erfüllt sein. Der Status, den ein CRDT hat, muss ein "Semi–Gitter", also eine geordnete Menge abbilden. Die Aktualisierungen müssen zunehmend sein, ein Status kann beispielsweise eine Zahl sein und die Aktualisierung ist die Operation, die sie inkrementiert. Die `merge` Funktion muss die kleinste obere Grenze der letzten Aktualisierung berechnen. Nur wenn ein Objekt diese Eigenschaften erfüllt, ist es dem CRDT zugehörig [SPBZ11b]

Operationsbasierter Ansatz

Wenn ein Replikat eine Aktualisierung von einem Client empfängt, wird es ebenfalls zuerst auf den lokalen Status angewandt. Im Gegensatz zum zustandsbasiertem Ansatz wird nicht der gesamte Status des Replikats gesendet, sondern nur der Aktualisierungsvorgang. Ein weiterer Unterschied ist das Fehlen der `merge`–Funktion. Statt ihrer gibt es im operationsbasierten Ansatz zwei `update`–Methoden: eine vorbereitende Aktualisierungsfunktion und eine ausführende. Erstere wird auf dem Replikat angewandt, umgehend gefolgt von der zweiten. Über ein Kommunikationsprotokoll wird die Aktualisierung an alle anderen Replikas asynchron versendet. Über das Protokoll wird sichergestellt, dass die Nachricht nur einmal überliefert wird. Die restlichen Replikas wenden die Operation mit der ausführenden Aktualisierungsmethode auf sich an [SPBZ11b]. Die folgende Abbildung stellt eine operationsbasierte Replikation mit drei Geräten dar.

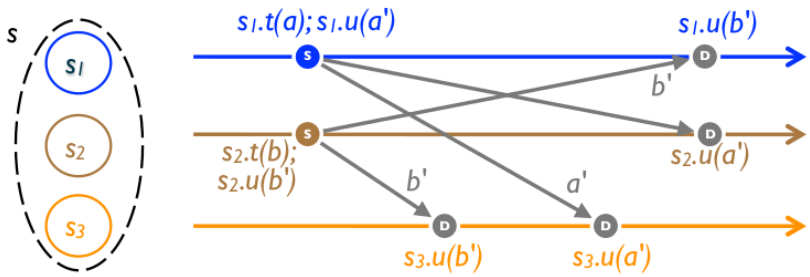


Abbildung 2.7: *Replikation von operationsbasierten CRDTs, Quelle: [SPBZ11b]*

Die drei Kreise auf der linken Seite repräsentieren drei identische Datensätze auf drei Geräten. Die Datensätze *S1* und *S2* werden zeitgleich mit unterschiedlichem Kontent aktualisiert. Sowohl *S1* als auch *S2* wenden die vorbereitende Aktualisierungsmethode t auf sich an. Dann mit deren Ergebnis die ausführende u. Sofort senden beide Replikas die Aktualisierung an alle anderen, welche ebenfalls die ausführende Aktualisierungsmethode u auf sich anwenden.

Die Voraussetzung für eine konfliktfreie Replikation sind hier kommunitative Aktualisierungen. Nur so spielt die Reihenfolge in der die Aktualisierungen bei den Replikaten ankommen, keine Rolle – das Ergebnis ist dasselbe [SPBZ11b].

2.2.5 Das CouchDB Replikationsmodell

CouchDB ist ein quelloffenes, dokumentenorientiertes Datenbankmanagementsystem (DBMS) mit einem integrierten Replikationsprotokoll.
CouchDB verwendet Replikation um Änderungen an Dokumenten zwischen zwei oder mehreren Datenbanken zu synchronisieren. Hierbei werden nur die Dokumente übertragen, die neu sind oder sich geändert haben. Die Replikation in CouchDB erfolgt schrittweise. Alle Änderungen an Dokumenten werden periodisch zwischen den Servern kopiert. Wenn der Replikationsprozess unterbrochen wurde, weil eine Datenbank keinen Internetzugang hat, haben zwei sich replizierende Datenbanken unterschiedliche Daten gespeichert. Bei wieder bestehender Internetverbindung wird die Replikation erneut ausgelöst und CouchDB setzt an dem Punkt, an dem es aufgehört hat, die Arbeit fort.

CouchDB ist darauf ausgerichtet, Konflikte vernünftig zu behandeln, statt anzunehmen, es träten keine auf. Das interne Replikationssystem besitzt eine automatische Konflikterkennung und –lösung.
Wichtig für diesen Mechanismus sind die Revisionsnummern. Dokumente werden mit

Revisionsnummern versioniert. Mit jeder Aktualisierung bekommt es eine neue Revision, die neben der alten gespeichert wird (vgl. [ALS10] S. 15ff & S. 150ff). Eine Revisionsnummer in CouchDB kann wie folgt aussehen `2-5560348cec1b08c3d53e1508b4a46868` und ist in zwei Bereiche zu teilen. Die Zahl vor dem Strich erhöht sich mit jeder neuen Revision des Dokuments, also mit jeder Aktualisierung. Alles hinter dem Strich ist ein md5-Hash aus dem Dokumenteninhalt, den Dateianhängen und dem `_deleted` Attribut[9]. Jede Revision hat außerdem eine Liste von vorherigen Revisionen.

Hat ein Dokument durch gleichzeitiges Bearbeiten in zwei unterschiedlichen Datenbanken dieselbe Revisionsnummer, erkennt CouchDB den Konflikt und markiert das Dokument, indem das Attribut `_conficts` den Wert `true` bekommt. Dann entscheidet CouchDB, welche Version gewinnt und welche verliert. CouchDB wird nie zwei Versionen zusammenführen. Das muss in der Anwendung implementiert sein. Die Entscheidung darüber, welche Version gewinnt, erfolgt über den Längenvergleich der Revisionslisten. Die Version mit der längsten Liste aus vorherigen Revisionen gewinnt. Sind beide Listen gleich lang, gewinnt die Revision die laut alphabetischer Sortierung am größten ist.
Auch wenn CouchDB die gewinnende und die verlierende Revision festlegt, werden beide Versionen gespeichert. Die gewinnende Revision wird als letztes gespeichert, die verlierende davor. Diese Konfliktlösungsstrategie wird auf allen CouchDB-Instanzen angewandt, weswegen dazu keine Internetverbindung notwendig ist. Dadurch, dass alle Instanzen denselben Algorithmus verwenden, werden die Revisionen immer in identischer Reihenfolge gespeichert und die Daten bleiben konsistent.
Jetzt kann im Entwicklungsprozess der Anwendung entschieden werden, wie mit den Konflikten umgegangen wird. Es kann aus beiden Versionen eine festgelegt werden, die behalten wird oder es können beide zusammengeführt werden (vgl. [ALS10] S. 153ff).

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

3 Bestehende offlinefähige Technologien

Um eine Webapplikation offlinefähig zu machen, müssen alle Ressourcen und die nutzerInnengenerierten 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 beschrieben.

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 cacht. Dazu benutzt es intern den Service Worker und App Cache als Fallback, für den Fall dass der Browser Service Worker nicht unterstützt [Sto].
Auch ungebündelte Assets können über das Plugin gecached werden. Diese Dateien müssen dann in den Optionen explizit angegeben werden. Auch der Service Worker und der App Cache lassen sich über die Optionen konfigurieren oder auch ausschalten [**?**].
Es werden allerdings nur die Assets und nicht die von BenutzerInnen generierten Daten gecached. Diese müssen manuell gespeichert werden.

3.2 Redux Offline

Redux Offline kann nur zusammen mit Redux verwendet werden [**?**]. Deswegen ist für die Verwendung von Redux Offline die Implementierung von Redux eine Voraussetzung.

3.2.1 Redux

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. Aktionen beschreiben was passiert und senden Daten von der Anwendung an den Store. Eine Aktion könnte beispielsweise die Information beinhalten, dass es eine Aktualisierung gab und auf welchem Objekt diese Aktualisierung stattgefunden hat. Der dritte wichtige Bestandteil von Redux sind die Reducer. Sie spezifizieren, wie der Appstate, der Status der Applikation, sich als Reaktion auf die Aktionen ändert [**?**].
Der Datenfluss in der Reduxarchitektur ist unidirektional. Zur Veranschaulichung wird anhand der folgenden Abbildung der Redux Datenfluss beschrieben.

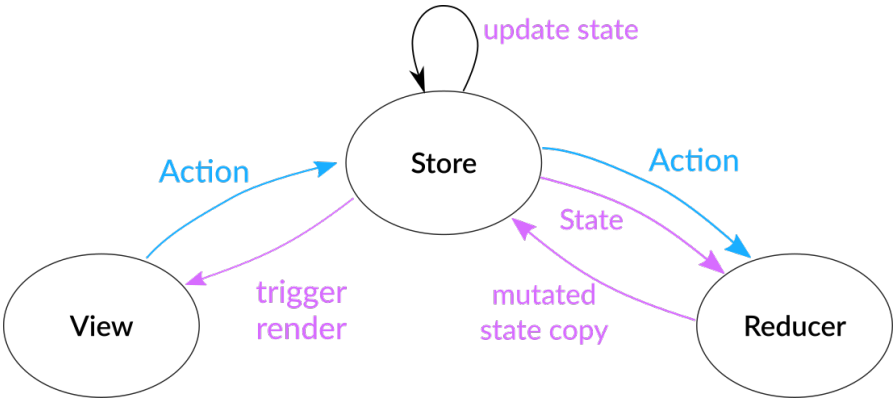


Abbildung 3.1: *Redux Datenfluss*

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

3.2.2 Redux Offline

Redux Offline erweitert Redux um einen persistenten Store mit Offline First–Technologie und ist kompatibel mit allen *View Frameworks wie React[10], Vue[11], oder Angular[12] [reda]. Es umfasst unter anderem netzwerkfähige API-Aufrufe, das Persistieren des Zustands

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

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

Eine mit Redux Offline erstellte Anwendung funktioniert ohne weitere Codeimplementierung lokal offline, da das Lesen und Schreiben aus der lokalen Datenbank bereits eingebunden ist. Damit die Anwendung auch offline funktioniert, wenn sie über das Netzwerk kommuniziert, müssen einige Anpassungen vorgenommen werden. Sämtliche Daten der Anwendung können nur über Aktionen manipuliert werden. Alle netzwerkgebundenen Aktionen werden in einer 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 die Queue. Hier werden die Aktionen inklusive Metadaten gespeichert, um bei bestehender Internetverbindung abgearbeitet zu werden [**?**]. Die von Jani Eväkallio erstellte Grafik 2.1 veranschaulicht die oben erklärte Architektur. Links ist eine Aktion zu sehen, die ”Zeug” machen soll. 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 umfasst die Queue namens `outbox` in dem Aktionen mitsamt ihrer Metafelder gespeichert werden. Rechts befindet sich das API, das über die Middleware mit dem Store kommuniziert.
Wird die Aktion ’DO_STUFF’ gefeuert gelangt sie in den Store, damit dieser den Appstate aktualisieren kann und wird ersteinmal in der Queue gespeichert. Ist die Anwendung online, wird sie sofort abgearbeitet. Wenn nicht, wird sie erst einmal in der Queue gespeichert bis die Anwendung wieder eine Verbindung zum Internet hat.

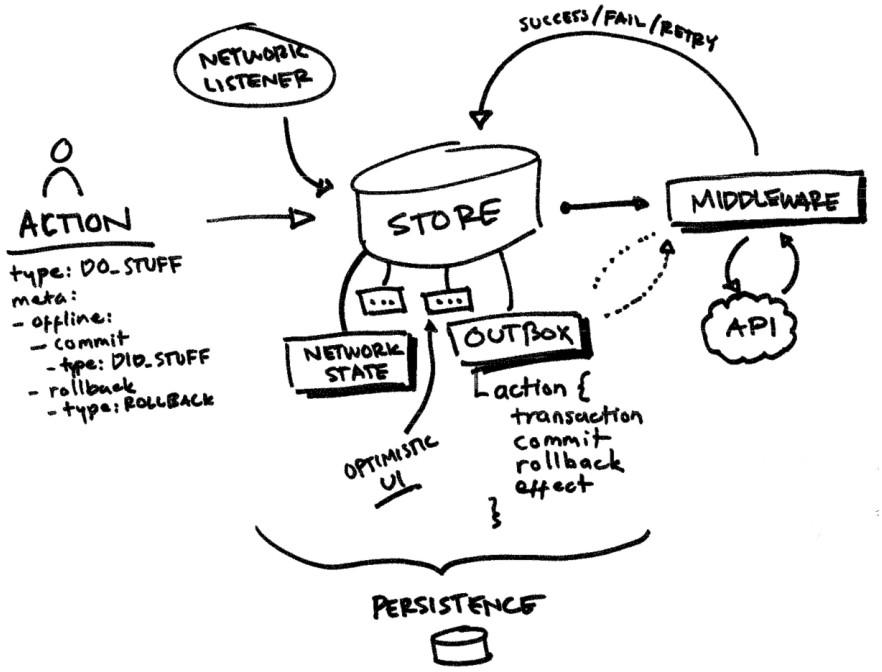


Abbildung 3.2: *Redux Offline–Architektur – Quelle: [Evä17]*

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 Local Storage, gespeichert [San16]. Es kann konfiguriert werden, wo die Daten gespeichert werden. Hierfür gibt es diverse Möglichkeiten, wie zum Beispiel im Session Storage, per localForage oder in Dateisystemen [redc]. LocalForage ist eine Bibliothek, mit der Daten in der IndexedDB oder in WebSQL gespeichert werden können. Wenn der Browser die Speichermöglichkeiten nicht unterstützt, wird der Local Storage genommen [**?**].
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 [redc].

3.3 React Native Offline

React Native ist ein Framework mit dem native, mobile Apps mit JavaScript und React gebaut werden können [**?**].
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 eine Offline Queue, in der 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 Feld erwartet ein Boolean. Hier 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 einer 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 der 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 bereitgestelltem Backend. Über das API können unter anderem BenutzerInnen authentifiziert, Daten gespeichert und synchronisiert wer-

den [hoo].
Anhand der Abbildung **??** wird erklärt wie HOODIE funktioniert.

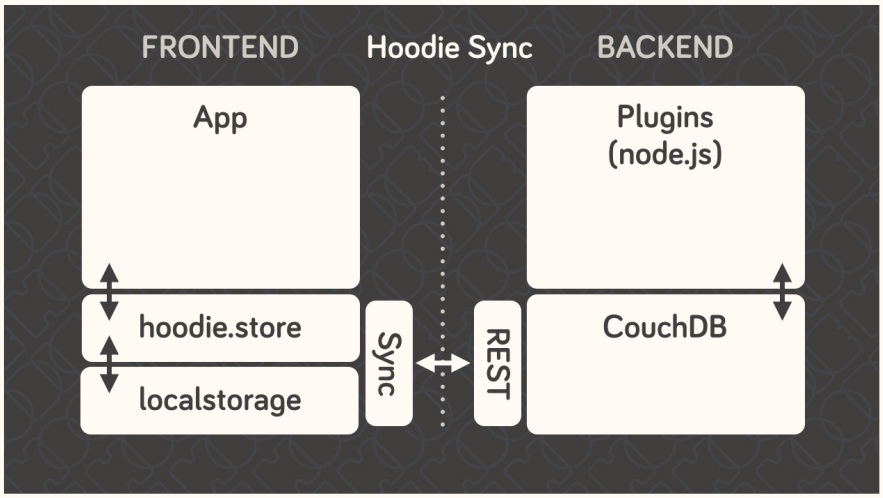


Abbildung 3.3: *HOODIE Architektur Quelle: [?]*

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 sie über eine REpresentational State Transfer (REST)–Schnittstelle mit einer CouchDB synchronisiert. In HOODIE haben alle AnwenderInnen ihre eigene private CouchDB. Hinter der Datenbank befindet sich ein Server der auf die Daten in der CouchDB reagiert, die wiederum die Änderungen an den Client schickt [**?**]. 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.
Das Frontend spricht nie direkt mit dem Backend, sondern zunächst mit der lokalen Datenbank. Dadurch ist die Funktionalität beider Komponenten auch dann gewährleistet, wenn die Verbindung zwischen ihnen unterbrochen wird [**?**].

3.4.1 CouchDB

Apache CouchDB™ ist ein DBMS, das seit 2005 als freie Software entwickelt wird. Die dokumentenorientierte Datenbank funktioniert sowohl als einzelne Instanz, als auch im Cluster. In einem Cluster kann ein Datenbankserver auf einer beliebig großen Anzahl an Servern oder virtuellen Maschinen ausgeführt werden.

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

Das implementierte Replikationsmodell erlaubt die Synchronisation bzw. bidirektionale Replikation zu verschiedenen Geräten, was das besondere Merkmal von CouchDB ist. Die genaue Funktionsweise des Protokolls ist in **??** detailliert beschrieben.
Dank des Replikations–API kann kann sich eine CouchDB kontinuierlich und eigenständig mit jeder 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 und gespeichert, jedoch nicht von selbst aufgelöst [cou]. 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 [cou].

3.4.2 PouchDB

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 diese sowohl offline als auch online verfügbar sind. PouchDB speichert die Daten in IndexedDB und stellt für das Abrufen und Manipulieren derer ein einheitliches API zur Verfügung.
Sind die Daten einmal offline gespeichert, können sie dank des CouchDB–Replikationsprotokolls, sobald die Awendung wieder online ist, mit CouchDB–kompatiblen Servern synchronisiert werden [pou].

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 [**?**].

Die Realm–Datenbank ist eine objektorientierte, plattformübergreifende lokale Datenbank die eine Echtzeitsynchronisation mit dem Realm Object–Server bereitstellt. Der Object–Server fungiert als Middleware-Komponente in der mobilen App und handhabt unter anderem die Ereignisbehandlung und Datensynchronisation. Im Zusammenspiel ermöglichen beide 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 auftretende Konflikte automatisch auflösen kann. Das hat zur Folge, dass die Synchronisation keinen manuellen Eingriff bedarf [rea17a].
Zusätzlich zu dem OT–Algorithmus benutzt Realm vorgegebene Regeln zur automatischen Konfliktlösung. In Realm gibt es 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 die Konfliktmanagementstrategie LWW angewandt, die letzte Aktualisierung gewinnt. 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 [**?**]. Darüber hinaus läuft die interne Konfliktlösung auf Transaktionsebene ab. Das heißt, der Vorgang ist nur erfolgreich, wenn er auch vollständig und fehlerfrei ist. Andernfalls 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 **??** genannten Voraussetzungen an eine

Offline First–App erfüllen.

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

Tabelle 3.1: *Übersicht der offlinefähigen Technologien*

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

4 Szenarien

Alle in Kapitel 2 angeführten Technologien haben die Unterstützung der Erstellung von offlinefähigen Anwendungen gemeinsam. Prinzipiell sollte eine Offline First–Anwendung in der Lage sein, trotz fehlender Internetverbindung zu funktionieren und mit auftretenden Konflikten so umzugehen, dass keine Daten verloren gehen. Sie muss die Fälle behandeln können, die sich aus den folgenden Szenarien ergeben. Dafür werden zunächst Szenarien in der Netzwerkübetragung als Voraussetzung für die der Konfliktentstehung aufgezeigt.

4.1 Szenarien bei der Datenübertragung

Im einfachen Anwendungsbeispiel einer Kontaktliste gibt es zwei Parteien, die miteinander interagieren: die Anwendung als Client und der Server. Immer wenn beide Parteien miteinander kommunizieren möchten, kann eine der beiden offline sein. Der Client könnte zum Beispiel, ohne dass eine Technologie implementiert ist, die Offlinestatus behandelt, einen Kontakt erstellen wollen. Der Kontakt kann aber nicht erstellt werden, da kein Netzwerk verfügbar ist.
”Client push” steht dafür, dass der Client etwas an den Server schickt. ”Server push/Client pull” beschreibt den Fall, in dem der Client Daten vom Server anfragt, oder der Server Push-Nachrichten an den Client sendet.

Szenario C0 – Client push:
Der Client erstellt einen Adressbucheintrag, hat den Status online und der Server ist erreichbar. Sowohl Anfrage als auch Antwort sind erfolgreich. Der Kontakt wird erfolgreich erstellt.

Szenario C1 – Client push:
Der Client erstellt einen Adressbucheintrag, hat den Status offline und es können keine Daten an den Server gesendet werden. Die Anfrage schlägt fehl.

Szenario C2 – Client push:
Der Client erstellt einen Adressbucheintrag und hat den Status online. Die Anfrage wird gestartet und währenddessen bricht die Internetverbindung ab. Die Anfrage ”wartet” bis ein Timeout getriggert wird und schlägt dann fehl. Wärend des Wartens ist der Client blockiert.

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

Szenario S1 – Server push/Client pull:
Der Client fordert eine Liste aller gespeicherten Kontakte vom Server an. Als dieser die angefragten Kontakte bearbeitet, um sie in der gewünschten Reihenfolge zu schicken, fällt der Strom aus. Die Daten können nicht gesendet werden, 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 und schlägt dann fehl. Wärend des Wartens ist der Client blockiert.

Szenario S3 – Server push/Client pull:
Der Client fordert initial eine Liste aller gespeicherten Kontakte vom Server an und hat den Status online. Während der Server antwortet, bricht die Internetverbindung ab. Nur ein Teil der angefragten Daten, die HTML–Datei der Anwendung und Metadaten der Kontaktliste, kommen beim Client an. Die Antwort ist teilweise erfolgreich.

In den obigen Szenarien wird nicht beschrieben warum die Verbindung zwischen den beiden Parteien 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 vollständigem Internetverlust führen. Genausogut kann es jedoch sein, dass der Server kaputt oder nicht erreichbar ist. Es gibt meherere Gründe dafür, dass die beiden Parteien nicht mehr miteinander kommunizieren können. Die Abbildung **??** veranschaulicht die beschriebenen Situationen, die bei der Übertragung von Daten über das Netzwerk eintreten können. Die 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 Ausgangssituationen, die Kreise repräsentieren die Szenarien und die Rechtecke die daraus resultierenden Fälle.

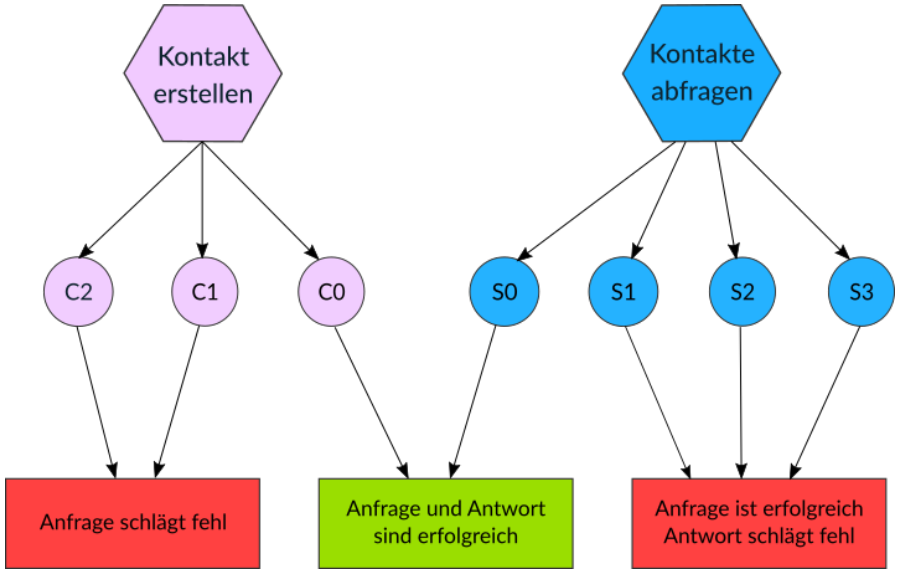


Abbildung 4.1: *Szenarien bei der Datenübertragung über das Netzwerk*

Ergebnis

Da die Szenarien *C0* und *S0*, *C1* und *C2* sowie *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.

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.

Laut den Aussagen von Jan Lehnardt[13], die er in einem Interview getroffen hat, ergeben sich unterschiedliche, angewandte Vorgehensweisen, Objekte für die Verwendung in verteilten Systemen zu identifizieren und versionieren. Nachfolgend werden diese Vorgehensweisen berücksichtigend, Konfliktszenarien dargestellt. 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 beschrieben.

Methode 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.

Methode ID1 – sprechender Schlüssel:
Zur Identifizierung eines Adressbucheintrags wird ein sprechender Schlüssel[14] verwendet. Sowohl auf dem Client als auch auf dem Server wird 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.

Methode 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 beide geben ihm die Versonsnummer ”2.0.0”. Währenddessen tritt Fall *b* oder *c* ein und beide Parteien können nicht miteinander kommunizieren. Bei der Synchronisation entsteht ein Konflikt weil es zwei unterschiedliche Einträge mit derselben Version gibt.

[13] Mitgründer und Geschäftsführer der Neighbourhoodie Software GmbH, Committer und Vice President von Apache CouchDB

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

Methode 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, immer noch offline, berichtigt. ”Amilia” hat nun die Version ”2018-04-13 11:45:33Z”.

Der Server hat eine eigene Uhr mit späterer 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 z.B. Google zu Problemen[15], wenn verschiedene Geräte eine eigene Uhr besitzen.

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

Methode 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 Uhr[16] verwendet. Amilias Adresse ändert sich und wird auf dem Client, mit einem Fehler im Straßennamen, angepasst. Währenddessen tritt Fall *b* oder *c* ein. Amilia sieht ihre falsche Hausnummer und berichtigt diese, indem sie ebenfalls das Adressfeld bearbeitet. Bei der Synchronisation kommt es zum Konflikt, denn es gibt zwei unterschiedliche Versionen von Amilias Adresse.

Methode V3 – Inhaltsbasierte Version:

Zur Versionierung eines Adressbucheintrags wird eine inhaltsbasierte Version verwendet. Um eine Zuordnung zwischen Inhalt und Version machen zu können kommen

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

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

Hashfunktionen zum Einsatz. Hierbei wird als Version der Hashwert des Adressbucheintrags gespeichert.
Dem Kontakt ”Amilia” ist die Version ”5560348cec1b08c3d53e1508b4a46868” zugeordnet. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst, während dieser offline ist. Im selben Status berichtigt der Client die Telefonnummer. Bei der Synchronisation kommt es zum Konflikt, da es nun zwei Einträge mit unterschiedlichem Inhalt, aber identischer Version gibt und nicht festzustellen ist, welche Version die neuere ist.

Methode 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 berichtigt der Client die Telefonnummer. Jede Aktion fügt der Versionsliste einen neuen Hashwert hinzu. Auch wenn der Content des Adressbucheintrags in den zwei letzten Versionen identisch ist, kann festgestellt werden welcher der neueste Eintrag ist. Bei der Synchronisation kommt es zum Konflikt weil der Kontakt Amilia mit unteschiedlichen Informationen existiert.
Beide konfliktbehafteten Versionen werden verschachtelt in der Liste gespeichert. In diesem Fall sieht die Liste so aus:
’[[88da3f8d82ab58551d2a48d74d9a4986, 5560348cec1b08c3d53e1508b4a46868], 88da3f8d82ab58551d2a48d74d9a4986]’ – eine Liste der beiden konfliktbehafteten Versionen am Anfang der Liste.

Ergebnis

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. Die Methoden *ID0* und *ID1* beschreiben die Identifizierung einzelner Kontakte. Eine eindeutige Identifizierung des Kontakts ist in Methode *ID0*, mittels der Verwendung einer UUID, gewährleistet.

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

5 Anforderungsdefinition

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

5.1 Umfang

Das Ziel dieser Arbeit ist die Untersuchung des Konfliktmanagements offlinefähiger Technologien. Dazu soll die beispielhafte Anwendung entwickelt werden, welche an dem Beispiel eines kollaborativen Adressbuchs die Offlinekompatibilität mit dem Schwerpunkt auf das Konfliktmanagement der verwendeten Technologien illustriert. Ein offlinefähiges, kollaboratives Adressbuch ist eine Anwendung, mit der mehrere Personen Kontakte verwalten können. Dass diese Anwendung auch ohne Internetzugang funktioniert, ist obligatorisch. Das Adressbuch beinhaltet eine Liste von Kontakteinträgen, welche jederzeit – unabhängig von der Internetverbindung – von den verwendenden Personen gelesen, bearbeitet, erstellt und gelöscht werden können. Die Offlinefunktionalität soll durch die verwendeten Technologien gegeben sein und wird nicht ausgiebig getestet. Geschieht eine dieser Operationen offline, werden die Daten bei wieder bestehender Internetverbindung synchronisiert. Im einfachen Fall erfolgt die Synchronisation zwischen dem Server und Client. Da die Beispielanwendung kollaborativ ist, erfolgt sie zwischen allen beteiligten Parteien. Der Schwerpunkt liegt auf den Konflikten, die entstehen können wenn keine Internetverbindung besteht. 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 ist. Im aktuellen Anwendungsfall bedeutet das, es gibt zwei Kopien des Adressbucheintrags. Eine auf dem Anwendungsgerät, eine auf dem Server.

Danach sollen nur die Einträge geladen werden, die nicht auf dem Gerät existieren. Andernfalls werden unnötige Mengen von Daten über das Netzwerk gesendet. Das soll vermieden werden. 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 identifizierbar sein. Wenn es mehrere 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, welche der untersuchten Technologien besser für die Entwicklung einer offlinefähigen Anwendung geeignet ist und warum, ist eine Testumgebung mitzuentwickeln. Hier soll die Möglichkeit geschaffen werden, Konflikte durch gleichzeitiges Bearbeiten eines Kontakts im Offlinestatus herbeizuführen und die Testfälle auswerten zu können.
Es wird ermittelt welche Strategie zur Konfliktlösung von der Technologie verwendet wird, ob die Funktionalität der Anwendung auch bei auftretenden Konflikten gewährleistet ist und ob dabei Daten verloren gehen. Dabei wird auch der Implementierungsaufwand betrachtet, der aufgebracht werden muss, um die Technologie zu verwenden und wie leicht der geschriebene Quellcode zu verstehen ist.

Um den Rahmen dieser Arbeit nicht zu sprengen, werden die Testdurchläufe auf nur zwei Geräten durchgeführt. Dort werden durch das Bearbeiten eines Kontakts auf beiden Geräten einfache Konflikte erzeugt. Auf das Zusammenführen von zwei konfliktbehafteten Versionen wird verzichtet. Gibt es zwei unterschiedliche Versionen eines Kontakts auf beiden Geräten, wird eine davon behalten.
Die verwendeten Technologien werden nur an die Anwendung angepasst, wenn es unbedingt notwendig ist. Um ein unverfälschtes Ergebnis erzielen zu können, werden die Technologien nur benutzt und erst an die Prototypen angepasst, wenn anderfalls die Grundfunktionalität der App eingeschränkt wäre. Es wird weiterhin auf ein komplexes User Interface (UI) und ein perfektes Nutzungserlebnis verzichtet.

Die in **??** erarbeiteten Szenarien zeigen, dass Konflikte immer auftreten können. Werden Konflikte falsch oder gar nicht behandelt, kann das zu Datenverlust führen. Aus diesem Grund müssen sie als Teil der Anwendung betrachtet, statt ignoriert zu werden. Im einfachen Konfliktfall kann das System entscheiden welches die konfliktfreie Version ist. So

kann zum Beispiel der Kontakt ”Amilia Pond” von einer Person eine neue Telefonnummer, von einer anderen eine neue Adresse bekommen. Die Aktualisierungen finden in unterschiedlichen Bereichen statt und sind deshalb unproblematisch zuzuordnen.

Die oben erarbeiteten Konfliktszenarien beschreiben die Konflikte die nicht vom System gelöst werden können. Diese sollen effizient gespeichert werden. Jeder Fehlerfall muss kommuniziert werden. Wenn es konfliktbehaftete Daten gibt, muss dies mitgeteilt und es muss angeboten werden, die Konflikte zu lösen.

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 auf mindestens zwei Geräten funktionieren.
F2	Die Anwendung soll fähig sein, den Netzwerkstatus zu ändern.
F3	Die Anwendung muss den Netzwerkstatus erkenntlich machen.
F4	Die Anwendung muss fähig sein, die Kontakte unabhängig vom Netzwerkstatus zu laden, sofern diese einmal aus dem Netzwerk geladen wurden.
F5	Die Anwendung muss die Möglichkeit bieten, unabhängig vom Netzwerkstatus einen Kontakt anzulegen, zu bearbeiten oder zu löschen.
F6	F6 Die Anwendung muss alle Kontakte sowohl lokal als auch serverseitig persistieren. Die Konntakte müssen identifiziert und versioniert werden.
F7	Die Anwendung muss die lokal gespeicherten Kontakte mit denen in der Serverdatenbank persistierten synchronisieren, sobald die Anwendung mit dem Internet verbunden ist.
F8	Die Anwendung muss die Möglichkeit bieten, die Konfliktmanagementstrategien der zu untersuchenden Technologien zu evaluieren.
F9	Die Anwendung soll Konflikte speichern, sofern diese auftreten. Das heißt, es muss solange nachvollziehbar sein, welche Versionen eines Kontakts konfliktbehaftet sind, bis der Konflikt gelöst ist.
F10	Die Anwendung muss die Möglichkeit bieten, Konflikte zu lösen, sofern diese auftreten.

F11	Die Anwendung muss sicherstellen, dass auf keinen Fall Daten verloren gehen.

Tabelle 5.1: *Funktionale Anforderungen*

5.3 User-Stories

Aus den in **??** 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 über die Anwendung immer und überall, auch ohne Internetzugang verfügen.
U2	Ich als NutzerIn möchte, dass die Kontaktliste schnell und effizient geladen wird, um Zeit zu sparen.
U3	Ich als NutzerIn möchte Einträge immer und überall, auch ohne Internetzugang, erstellen, editieren oder löschen können, um meine Liste zu verwalten.
U4	Ich als NutzerIn möchte keine in der Anwendung gespeicherten Daten verlieren.

Tabelle 5.2: *Anforderungen aus NutzerInnenperspektive*

5.3.2 EntwicklerInnen–Perspektive

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

ID	Anforderung aus Entwicklungsperspektive
D1	Ich als EntwicklerIn möchte die Daten lokal und auf dem Server speichern, um deren Erreichbarkeit unabhängig vom Internetstatus zu gewährleisten.
D2	Ich als EntwicklerIn möchte nur die Adressbucheinträge oder deren Aktualisierungen laden, die sich nicht schon auf dem Endgerät befinden, um Datenübertragungen und Ladezeiten zu sparen.
D3	Ich als EntwicklerIn möchte jeden Eintrag eindeutig identifizieren, um jedem Adressbucheintrag Operationen zuzuweisen und einzelne Kontakte zu finden.
D4	Ich als EntwicklerIn möchte, dass jeder Eintrag versioniert ist, um zu wissen ob und 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 speichern, um mit ihnen umgehen zu können.
D7	Ich als EntwicklerIn möchte eine Technologie verwenden, die leicht zu verstehen und implemenieren ist, um den Arbeitsaufwand gering zu halten.
D8	Ich als EntwicklerIn möchte sauberen und verständlichen Code schreiben, um die Les– und Wartbarkeit zu gewährleisten.

Tabelle 5.3: *Anforderungen aus Entwicklungsperspektive*

5.3.3 TesterInnen–Perspektive

Die folgende Tabelle zeigt die Software-Anforderungen an eine offlinefähige Kontaktliste aus der 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 bearbeiten zu können.

T4	Ich als TesterIn möchte Konflikte forcieren, um das Verhalten der Anwendung zu evaluieren.
T5	Ich als TesterIn möchte einen Eintrag editieren können wenn 1. beide, Client und Server online sind, 2. entweder Client oder Server offline ist oder 3. beide Parteien offine sind.
T6	Ich als TesterIn möchte die Testfälle detailliert dokumentieren, 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 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. **??** zeigt, wie so ein Dialogfenster aussehen könnte.

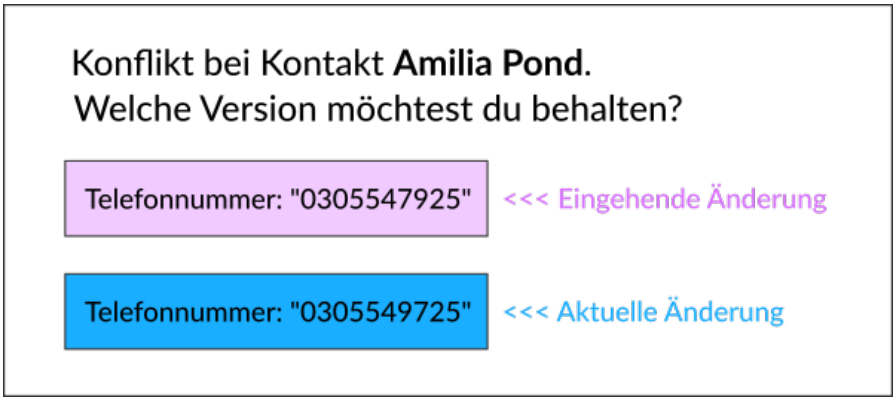


Abbildung 5.1: *Dialogfenster im Konfliktfall*

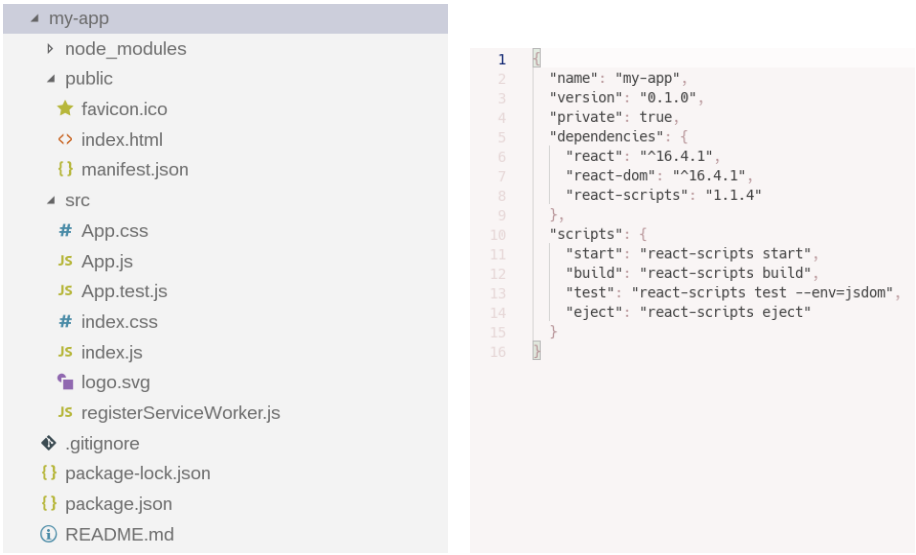
Wurde beispielsweise Amilias Telefonnummer von zwei Personen gleichzeitig bearbeitet, bildet der Dialog zwei Bereiche mit der Nummer in den unterschiedlichen Versionen ab. Daneben ist abzulesen woher die Änderung stammt. Die lokal gespeicherte Version kann mit ”Deine Änderung” betitelt werden, die Version, die vom Server kommt mit ”Andere Änderung”. Durch einen Klick auf den Knopf mit der Telefonummer kann entschieden werden welche als korrekte Version behalten wird.

6 Konzeption

Die erarbeiteten Anforderungen zur Untersuchung der Konfliktmanagementstrategien offlinefähiger Systeme werden in diesem Kapitel für die Konzeption angewandt.
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 [**?**]. Doch da für den zu entwickelnden Prototyp lediglich diese beiden Komponenten benötigt werden, wurde gegen den Einsatz von HOODIE entschieden.
Die Wahl fiel auf CouchDB und PouchDB, weil diese Technologien häufig in unterschiedlichen Artikeln, die im Zusammenhang mit Offline First–Anwendungen stehen, genannt und empfohlen werden.
Als einzige Technologie die neben HOODIE bzw. CouchDB eine Datenbanksynchronisation bereitstellt, stand auch Realm zur Wahl. Realm unterstützt jedoch nur die Entwicklung von mobilen Anwendungen und ist außerdem zahlungspflichtig. Dazu sind der Object–Server, der Teil, der für die Datensynchronisation zuständig ist, und dessen Dokumentation nicht quelloffen. Das heißt, es ist nicht nachvollziehbar welche der im Realm Whitepaper genannten Konfliktmanagementstrategien, OT oder LWW, angewandt wird. Bei Betrachtung der von Realm voreingestellten Regeln, dass Löschungen und jede letzte Aktualisierung immer gewinnen, ist kein herausstechender Synchronisationsalgorithmus zu erkennen.
Aus diesen Gründen wurde als Alternative zu Realm Redux Offline gewählt. Redux ist ein beliebtes und besonders im Zusammenhang mit React ein oft verwendetes Tool, das sowohl in mobilen, als auch Desktopapplikationen integriert werden kann [**?**]. Die Versprechungen von Redux Offline sind hoch und es soll untersucht werden, inwiefern diese eingehalten werden.
Bis zu einem gewissen Status, nämlich dem der Verwendung der Technologien, sind beide Prototypen – bis auf den Namen– identisch. Beginnend mit dem Aufbau der exemplarischen Anwendung werden in den folgenden Abschnitten der Anwendungsaufbau, Architektur und schließlich die graphische Oberfläche aufgeführt.

6.1 Anwendungsaufbau

Die Prototypen bestehen im Frontend aus React und wurden mit Create React App erstellt. Create React App ist ein Command Line Tool (CLI), das ein Projekt mit dem gewünschten Namen erstellt, eine initiale Projektstruktur generiert (vgl. **??**) und die dafür benötigten Abhängigkeiten installiert [**?**].



(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 Service Worker 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 Service Worker initialisiert. Alle `App.*`–Dateien umfassen eine minimale Beispielanwendung. In der generierten `package.json`–Datei (vgl. **??**), 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 UIs zu erstellen, also die Seite der Anwendung, die für die Anzeige und Interaktion zuständig ist. Ein Vorteil von React ist die komponentenbasierte Philosopie. Eine Komponente ermöglicht die Aufteilung der UI in kleine Teile und ist eine abstrakte Basisklasse. Einmal implementiert, lässt sich eine Komponente immer wieder verwenden [**?**].
Eine Komponente kann einen internen State besitzen, oder die Daten aus den `props`

nehmen. `Props` sind Daten, die von der Elternkomponente übergeben werden und können auch nur von dieser manipuliert werden. Eine React–Komponente hat immer eine `render()`–Funktion, die die Daten aus dem State oder den `props` liest und zurückgibt, was dargestellt werden soll. Hier wird das zur Komponente gehörende HTML erzeugt. Jede Änderung des States führt einen erneuten Aufruf der `render()`–Funktion mit sich.

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

```
1  class FormContainer extends Component {
2    constructor (props) {
3      super(props)
4      this.state = {
5        contact: props.contact
6      }
7    }
8
9    handleChange (event) {
10     let contact = this.state.contact
11     contact = {...contact, name: event.target.value}
12     this.setState(() => ({contact: contact}))
13   }
14
15   render () {
16     return (
17       <ContactForm
18         handleChange={this.handleChange}
19         contact={this.state.contact} />
20     )
21   }
22 }
```

Listing 6.1: *Eine React Containerkomponente*

Die Formular–Containerkomponente hat ein Kontaktobjekt im internen State gespeichert.

Auf dieses Objekt haben andere Komponenten keinen Zugriff und es ist nur via `setState()` änderbar. Initial wird das Kontaktobjekt über die `props` in Zeile 5 geladen. So kann das Vorausfüllen der Eingabefelder realisiert werden.
In Zeile 9 steht die `handleChange()`–Funktion, die den internen State in Zeile 12 mit den eingegebenen Werten aktualisiert. Eine Viewkomponente wird wie in in den Zeilen 17 bis 19 eingebunden. Ihr wird der interne State übergeben, also das Kontaktobjekt und die `handleChange()`–Funktion. Die beiden Parameter sind in Zeile 1 des folgenden Listings wiederzufinden.

```
1 const ContactForm = ({contact, handleChange}) => (
2   <form className='contact-form'>
3     <label>Name</label>
4     <input
5       value={contact.name}
6       onChange={this.handleChange.bind(this)} />
7   </form>
8 )
```

Listing 6.2: *Eine React Viewkomponente*

Die Viewkomponente macht nichts anderes, als die ihr übergebenen Daten anzuzeigen. Im Ereignisbehandler des Eingabefeldes wird die übergebene `handleChange()`–Funktion in Zeile 5 aufgerufen. Diese Komponente besitzt keinerlei Logik.

6.1.2 Verwendung von Redux Offline

Redux Offline kann nur zusammen mit Redux verwendet werden, eine Bibliothek die in **??** vorgestellt wird. Deswegen ist für diesen Prototypen die Implementierung von Redux vorausgesetzt.

Redux Offline ist eine erweiternde Bibliothek für Redux, dessen Funktionsweise in Abschnitt **??** detailliert beschrieben wird.
Nach der Installation muss der Redux `store` zusammen mit dem `offline store enhancer` erzeugt werden. Listing **??** visualisiert diesen Vorgang. Ein Redux `store` wird mit dem `storeCreator` in Zeile 5 erzeugt. Ein `store enhancer` ist eine Funktion, die den `storeCreator` neu zusammenfügt und einen neuen, erweiterten `storeCreator` zurückgibt. Redux Offline kommt mit einer Grundkonfiguration (siehe Zeile 3). Diese wird dem `offline store enhancer` in Zeile 8 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, 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 6. Bei `commit` in Zeile 7 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 [**?**].
Die Aktionen beschreiben nur was passiert. Wie der Status sich ändert, wird im Reducer beschrieben. Das Listing **??** illustriert, wie das prototypisch umgesetzt werden könnte.

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

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

In den Zeilen 6 bis 8 ist nachzulesen, wie sich der Appstatus bei erfolgreichem Netzwerkaufruf aktualisiert. Der Status aktualisiert sich nur, 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 das Zusammenspiel 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 **??** 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. In dem Teilbereich darunter sind der State oder die Props der Komponente abzulesen. Bei Containerkomponenten handelt es sich um den Status. Die Viewkomponenten haben keinen

eigenen State, bei Ihnen stehen an dieser Stelle die Props.
Alle Komponenten, bei denen im Namen das Wort ”Container” vorkommt, sind Containerkomponenten, beinhalten Logik und entscheiden welche Viewkomponente gerendert wird. Die anderen sind Viewkomponenten und können im Gegensatz zu den Containerkomponenten 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 Viewkomponenten Zugriff haben. Kursiv geschriebene Werte kennzeichnen externe Module, die von der Komponente verwendet werden. Es werden für jede Komponente, bis auf ”Contacts”, nur die eigens implementierten Funktionen aufgelistet. Um besser nachvollziehen zu können, welche Funktionen und Eigenschaften von der Contacts–Komponente an die Kindkomponenten durchgereicht werden, werden diese in grauer Schrift aufgezählt.

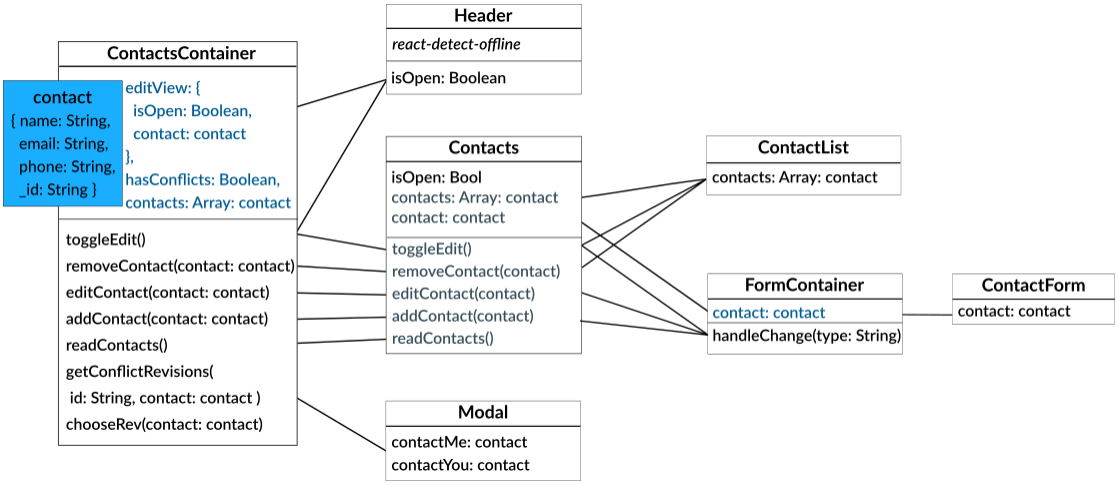


Abbildung 6.2: *Komponentendiagramm der Prototypen*

Die Komponente `ContactsContainer` ist das Herzstück der Anwendung. Hier wird die graphische Oberfläche definiert und es werden alle notwendigen Funktionen bereitgestellt. Die Komponente hat einen internen State in dem sowohl die Kontaktliste, als auch die Daten für das Formular gespeichert sind. Im Diagramm ist der Status an der blauen Schrift zu erkennen. Das Objekt `editView` speichert den im Formular zu ladenden Kontakt und zeigt welche Ansicht gerade aktuell ist. Ist das Attribut `isOpen` auf `true` gesetzt, wird das Formular gerendert, andernfalls die Kontaktliste. 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, wodurch ein erneutes Rendern der Komponente eingeleitet wird. 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. Das Bearbeiten kann durch den Aufruf der durchgereichten Funktion `toggleEdit()` eingeleitet werden. Durch den Aufruf von `removeContact()` wird der Kontakt gelöscht.
Die Komponente `Contacts` gehört als Viewkomponente zu `ContactsContainers`. Je nach Wert des Attributs `isOpen`, wird hier entweder die Kontaktliste oder das Formular gerendert. Die restlichen, grau geschriebenen Eigenschaften und Methoden werden an die Kondkomponenten durchgereicht.
Die Komponente `ContactForm` zeigt, sofern vorhanden, alle im Kontakt gespeicherten Daten an. Gibt es keine Kontaktdaten, die geladen werden können, kann ein neuer Kontakt angelegt werden. In der View, dem `ContactForm`, gibt es zum Hinzufügen und Bearbeiten des Kontakts Eingabefelder für den Namen, die E-Mail und die Telefonnummer des Kontakts. Zusätzlich zu den Eingabefeldern für jedes Kontaktattribut hat sie zwei Knöpfe, mit denen die Aktion bestätigt oder abgebrochen werden kann.
Der dazugehörige Ereignisbehandler befindet sich im `FormContainer`. Der ”lauscht” auf die Veränderung der einzelnen Eingabefelder und speichert alle Änderungen in einer Kopie des Kontakts im internen State. Wird die Aktion durch den entsprechenden Knopf bestätigt, wird die Kopie des Kontakts an `Contacts` gegeben und dort entweder via `addContact()` hinzugefügt oder via `editContact()` bearbeitet. Wenn die Aktion abgebrochen wird, ruft der `FormContainer` die `toggleEdit()` Funktion in der Hauptkomponente auf.

Kommt es beim Hinzufügen, Bearbeiten oder Löschen eines Kontakts zum Konflikt, werden mittels `getConflictRevisions()` die beiden Versionen des konfliktbehafteten Kontakts ermittelt. Der Kontakt wird dann in beiden Varianten, der alten und der neueren Version, an einen 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 ausgewähltem Kontakt als Parameter auf. Innerhalb von `chooseRev()` wird der übergebene Kontakt gespeichert und die konkurrierende Version verworfen.

Eine Backendimplementierung ist für den Prototypen *amilia-qouch* nicht notwendig, da dies bereits durch die Verwendung von CouchDB gegeben ist. Die **??** skizziert die Architektur von *amilia-qouch*. Die Containerkomponenten handhaben die Daten, die in den

Viewkomponenten angezeigt werden und speichert sie in PouchDB. PouchDB synchronisiert sich mit der CouchDB und schreibt die Daten zurück in den State, der in den Containerkomponenten verwaltet wird.

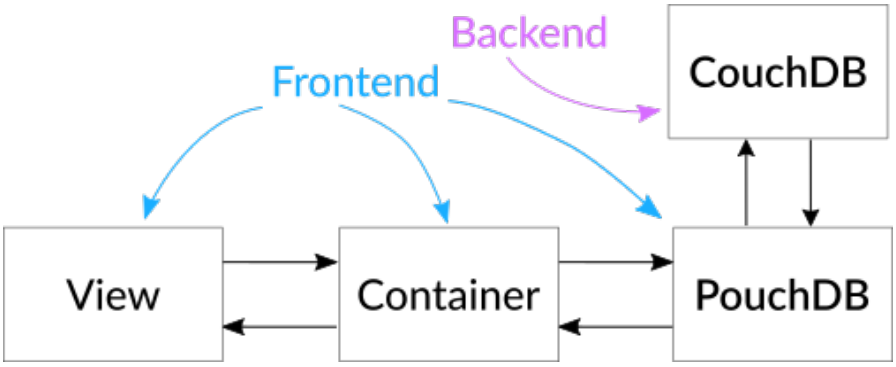


Abbildung 6.3: *Client-Server-Modell des Prototypen amilia-qouch*

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 **??** zeigt, dass der Redux Store in *amilia-rdx* PouchDB in *amilia-qouch* 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 durch den Server und einer JSON–Datei ersetzt.

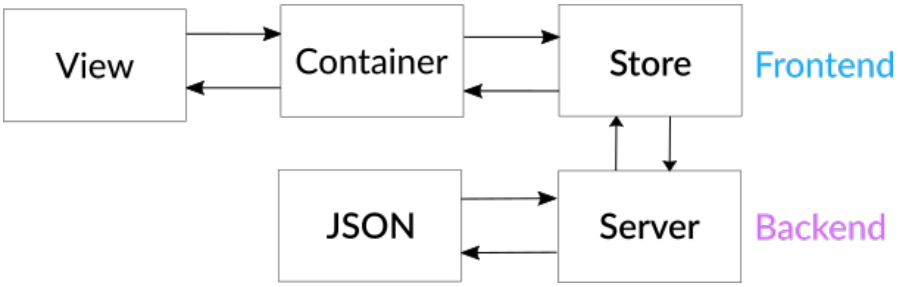


Abbildung 6.4: *Client-Server-Modell des Prototypen amilia-rdx*

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 in die andere Richtung ist das nicht implementiert. Deswegen werden nach jeder Aktion die Kontakte vom Server geladen.

6.2.1 Das Speichern der Daten

Das Speichern von Kontakten wird in den Prototypen unterschiedlich implementiert.

Daten mit PochDB und CouchDB speichern

Für den Protoyp *amilia-qouch* muss zunächst CouchDB installiert und gestartet 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 **??** führt alle benötigeten Funktionen auf und zeigt die notwendigen Schritte zur Synchronisation der lokalen PouchDB und CouchDB.
Die lokale Datenbank wird in Zeile 2 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 3 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 5 erforderlich. Im optionalen Parameter können zum Beispiel Filter oder Einstellungen zum wiederholten Synchronisationsversuch im Falle eines Fehlschlags gesetzt werden [**?**].
Der Aufruf `localDB.allDocs()` in Zeile 8 fragt alle in der lokalen Datenbank gespeicherten Kontakte an. Ohne den Parameter `include_docs: true` werden nur die ID `_id` und die Revisionsnummer `_rev` eines jeden gespeicherten Kontakts zurückgegeben. Ist die Option `conflicts` auf `true` gesetzt, werden Konfliktinformationen zu jedem Kontakt gespeichert. Alle konfliktbehafteten Kontakte haben nun ein `_conflicts`-Attribut. Dort befindet sich eine Liste mit allen konkurrierenden Revisionen.

Für die nachfolgenden Aufrufe gibt es mehrere Varianten. Eine Anleitung zum Umgang mit Konflikten in der PouchDB–Dokumentation empfiehlt für das Erstellen, Aktualisieren und Löschen eines Dokuments, das Modul PouchDB Upsert zu verwenden. Das wird in Zeile 1 zum PouchDB–Objekt hinzugefügt. Jedes Mal, wenn eine der CRUD–Operationen auf einem Dokument ausgeführt wird und das API eine Revision verlangt, kann ein Konfliktfehler geworfen werden. Das Modul PouchDB Upsert wiederholt die Aktion solange, bis sie erfolgreich ist. Es fügt ein neues Dokument hinzu, wenn noch keins existiert und aktualisiert es, wenn es vorhanden ist [**?**]. Konflikte treten so trotzdem auf und CouchDB wählt eine gewinnende Version aus. Alle Konflikte werden immer noch gespeichert und die beteiligten Dokumente können wie oben beschrieben aus der CouchDB geladen werden.

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

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

Die Zeilen 13 bis 16 zeigen, wie ein Kontakt erzeugt werden kann. Bevor das geschieht, wird eine ID erstellt die aus dem aktuellen Zeitstempel besteht. Diese wird dann per PouchDB Upsert in dem neuen Dokument gespeichert.
Das Aktualisieren eines Kontakts sieht sehr ähnlich aus, da die `upsert`-Funktion für beide Operationen verwendet wird. Mit jedem Update bekommt ein Kontakt von PouchDB eine neue Revisionsnummer.
Man kann einen Kontakt mit PouchDB Upsert wie in den Zeilen 26 bis 29 löschen. Der Kontakt wird hierbei 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 **??** erläutert wird. Der Redux Store wird bei jeder Änderung persistiert und beim Start der Anwendung aus dem Local Storage geladen.
Wie die Daten mit Redux Offline gespeichert und synchronisiert werden ist im folgenden Listing erklärt. Das Beispiel erklärt das Hinzufügen eines Kontakts. Die restlichen Operationen, das Bearbeiten, Löschen oder Lesen eines Kontakts sind analog dazu.

Mit dem Aufruf der Aktion ADD_CONTACT, die in Zeile 3 bis 16 erzeugt wird, wird der Vorgang ”einen Kontakt hinzuzufügen” gestartet. Die anzufragende URL ist im `effect` Feld des `meta.offline`–Objekts in Zeile 9 festgelegt. Die Anfrage geht an den Server, welcher die Daten in der JSON–Datei persisitiert. Der Reducer hat Zugriff auf das Aktionsobjekt. Dort ist der gerade hinzugefügte Kontakt gespeichert. Mit diesem wird in Zeile 25 der Appstate aktualisiert und so lokal gespeichert.

```
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`,
10           method: 'POST',
11           body: JSON.stringify({contact})
12         },
13         commit: { type: 'ADD_CONTACT_COMMIT', meta: { contact } },
14         rollback: { type: 'ADD_CONTACT_ROLLBACK', meta: { contact } }
15       }
16     }
17   }
18 }
19
20 // reducer
21 function contacts (state=[], action) {
22   switch (action.type) {
23     case ADD_CONTACT:
24       console.log('started to add contact')
25       return [...state, action.contact]
26
27     case ADD_CONTACT_COMMIT:
```

```
28        console.log('successfully added contact')
29        return [...state, action.payload]
30
31        case ADD_CONTACT_ROLLBACK:
32        console.log('failed to add contact')
33        return state
34    }
35 }
```

Listing 6.7: *Speicherung der Daten mit Redux Offline*

Ist die Netzwerkanfrage erfolgreich, wird die Aktion `commit` in Zeile 13 gefeuert. Die wird im Reducer in Zeile 27 behandelt. Der State wird mit der Antwort vom Server aktualisiert und die Synchronisation ist vollzogen. Für den Fall, dass die Netzwerkanfrage nicht erfolgreich war, wird das `meta.offline.rollback`–Feld in Zeile 14 definiert. Hier könnte man die BenutzerInnen darüber informieren, dass etwas nicht geklappt hat. Um den Rahmen dieser Arbeit nicht zu sprengen, genügt es jedoch, den vorherigen Status zurückzuliefern, wie es in Zeile 33 passiert.

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

```
 1 const fs = require('fs')
 2 const contacts = require('./contacts.json')
 3
 4 app.post('/contacts', function (req, res) {
 5   let contact = req.body.contact
 6   if (!contact || typeof contact === 'undefined') {
 7     res.status(400).send({ msg: 'contact malformed.' })
 8   } else {
 9       // persist contact
10       contacts.push(contact)
11       fs.writeFile('utils/contacts.json', contacts, 'utf8', function (err) {
12         if (err) {
13           res.status(500).send({msg: 'failed to write file'})
14           throw err
15         }
16         res.json(contact).send()
17       })
18   }
19 })
```

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

In Zeile 2 werden die Kontakte aus einer JSON–Datei geladen. Diese sind als Objekt in einem Array gespeichert. Bekommt der Server eine Anfrage zum Erstellen eines Kontakts, wird ein Kontaktobjekt mitgesendet. Dieses wird in Zeile 5 in einer Variable zwischengespeichert. Wird der Kontakt korrekt gesendet, wird er in Zeile 10 dem Array hinzugefügt. Andererseits sendet der Server den HTTP–Statuscode 400 an den Client. Außerdem wird mithilfe des in Node integrierten Dateisystem[17]–Moduls die JSON–Datei neu geschrieben. Nun ist der neue Kontakt persistiert. Kommt es beim Schreiben der Datei zu keinem Fehler, sendet der Server den frisch gespeicherten Kontakt zurück an den Client.

6.2.2 Verbindungsstatus feststellen und ändern

Für die Überprüfung der Verbindung zum Server wird das Modul React Detect Offline verwendet. Es bietet zwei Komponenten, die entsprechend des Netzwerkstatus den ihnen gegebenen 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 [**?**].

Der Verbindungsstatus eines Gerätes kann im Browser geändert werden. Die Prototypen, die im Rahmen dieser Arbeit entwickelt werden, sollen in den Browsern Firefox und Chrome lauffähig sein.
In Firefox lässt sich der Netzwerkstatus über das Einstellungsmenü ändern. Dort kann

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

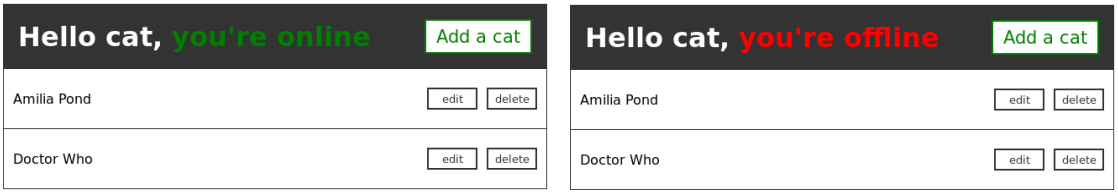
man entweder unter dem Punkt ”Sonstiges” oder dem Punkt ”Web-Entwickler” ”Offline arbeiten” auswählen und ist vom Internet getrennt. Dieser Status lässt sich über den selben Weg rückgängig machen.
In Chrome öffnet man dazu die Entwicklertools, geht auf ”Netzwerk” und klickt auf die Checkbox ”Offline” am oberen Rand. Dieselbe Checkbox ist auch im ”Application”–Tab unter ”Service Workers” zu finden.
Soll 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 URL vom Server 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 **??** besteht aus dem Header / Kopf und den Listeneinträgen. Sie zeigt die Kontakteinträge in beiden Netzwerkstatus: online (Abbildung **??**) und offline (**??**).

Im Header ist abzulesen, ob die Anwendung gerade eine Verbindung zum Server 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”–Knopfes wird der entsprechende Eintrag in der Liste gelöscht

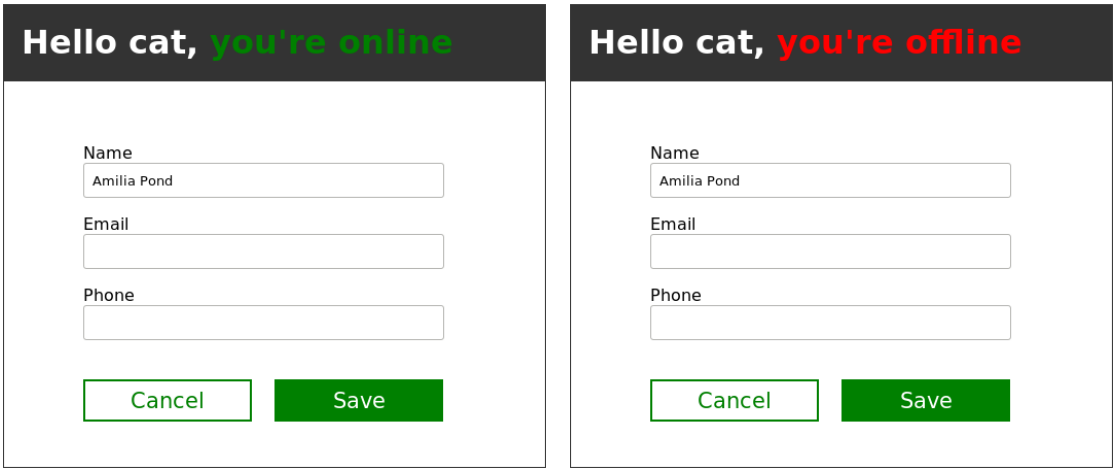


(a) *Kontaktliste im Onlinestatus* (b) *Kontaktliste im Offlinestatus*

Abbildung 6.5: *Die Kontaktliste in beiden Netzwerkstatus*

Klickt man auf den Knopf zum Bearbeiten oder auf den zum Hinzufügen eines Kontakts, gelangt man in die Bearbeitungsansicht (vgl. Abbildung **??**). Der Header ist bis auf den

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



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

Abbildung 6.6: *Die Editieransicht in beiden Netzwerkstatus*

Sobald ein Konflikt aufgetreten ist, soll sich ein Dialogfenster öffnen, der die 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 bevorzugte 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 gelöst 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 in **??** erarbeiteten Szenarien zeigen auf, dass es immer zu Konflikten kommen kann, wenn die Internetverbindung, oder der Kontakt zum Server abbricht. Deswegen werden Konflikte erstellt, indem die Verbindung zum Netzwerk unterbrochen wird.
Die zu entwickelnden Prototypen müssen auf mindestens zwei Geräten funktionieren und es muss die Möglichkeit bestehen, die Verbindung zum Internet und zum Server zu unterbrechen. Eine Variante Konflikte entstehen zu lassen, ist einen Kontakt an derselben Stelle zu bearbeiten während ein oder beide Geräte offline sind. Daher ist es wichtig zu wissen, in welchem Status sich die Anwendung befindet. Die Testdurchläufe sind immer gleich und unabhängig vom Prototypen.

Zuerst wird die App in zwei Browsern gestartet. So kann die Verwendung von zwei Geräten simuliert werden. Da die Anwendung für die Browser Firefox und Chrome entwickelt wird, findet der Testdurchlauf auch in diesen 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 erreichbar. Diese Testreihe soll der Untersuchung auf Kollaborationsfähigkeit dienen. Es wird also untersucht, ob die Anwendung auf mehreren Geräten funktioniert. 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. In den letzten beiden Testreihen werden durch das Bearbeiten derselben Einträge aktiv Konflikte erzeugt.
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. Zuerst wird die Aktion in der Anwendung, die mit dem Internet verbunden ist ausgeführt und gespeichert, danach in der Anwendung, die sich im Offlinemodus befindet.

Die folgende Tabelle veranschaulicht die durchzuführenden Testdurchläufe.

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

Tabelle 6.1: *Die Testdurchläufe*

Zur Auswertung der Tests werden alle Ausgangspositionen, Vorgänge und Ergebnisse dokumentiert. 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, was das erwartete und was das tatsächliche Ergebnis war.

7 Implementierung der Prototypen

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

7.1 Die Contacts–Komponenten

Das Herzstück der hier zu entwickelnden Anwendungen sind die Contacts–Komponenten. Sie beinhalten die zentralen Funktionalitäten und entscheiden welche Anzeigeelemente gerendert werden und welche nicht. `Contacts.js` ist die Viewkomponente und somit für das Rendern der anderen Komponenten zuständig. Die Containerkomponente ist die Datei `ContactsContainer.js`. Hier sind Funktionen implementiert über die der State manipuliert werden kann.
Im Prototypen *amilia-rdx* werden in `actions.js` die Aktionen definiert, die über die Containerkomponente `ContactsContainer.js` von jeder Komponente aufgerufen werden können um den Appstate zu manipulieren. Die Manipulation wird in `reducer.js` behandelt, wo die geänderte Kopie des Appstatus wieder an den Store zurückgegeben wird. Die Methode `componentDidMount` ist die dritte im React Komponenten–Lebenszyklus[18] und wird aufgerufen sobald die Anwendung an die DOM Elemente gemountet ist. Hier werden die Kontakte geladen und im Appstate gespeichert. Sobald sicht der Appstate ändert, wird die `render` Funktion der Komponente aufgerufen. **??** zeigt die Renderfunktion in `ContactsContainer.js` des Prototypen *amilia-qouch*.

```
1 render () {
2   const { contacts, editView, modalView } = this.state
3   return (
4     <div>
5       <Header
```

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

```
6           isOpen={editView.isOpen}
7           handleGoToEdit={this.toggleEdit.bind(this, null)} />
8
9         {modalView.hasConflict &&
10          <Modal
11            contactMe={modalView.contactMe}
12            contactYou={modalView.contactYou}
13            removeRev={this.removeRev.bind(this)} />
14        }
15
16        <Contacts
17          isOpen={editView.isOpen}
18          toggleEdit={this.toggleEdit}
19          addContact={this.addContact}
20          editContact={this.editContact}
21          deleteContact={this.deleteContact}
22          contact={editView.contact}
23          contacts={contacts} />
24      </div>
25    )}
```

Listing 7.1: *Die Renderfunktion in* `ContactsContainer.js` *des Prototypen amilia-qouch*

Der Header wird in den Zeilen 5 bis 7 gerendert. In Zeile 6 wird ihm die Information `isOpen` übergeben, die aussagt, ob der Editiermodus aktiv ist, also ob das Formular gerade geöffnet ist oder nicht. Ihm wird auch die Funktion `toggleEdit` übergeben, welche genau diese Statuseigenschaft wechselt.
Die Zeilen 9 bis 14 wird, sofern der Wert `hasConflict` im State auf `true` gesetzt ist, der Konfliktdialog gerendert. Der Dialog existiert nur im *amilia-qouch* Prototypen. Ihm werden die beiden konfliktbehafteten Versionen des Kontakts und eine Funktion zum Löschen der verlierenden Version übergeben. Näheres dazu wird im **??** erklärt.
In den Zeilen 16 bis 23 wird die Contacts–Viewkomponente gerendert. Ihr werden alle Eigenschaften und Funktionen übergeben, die für ihre Kindkomponenten notwendig sind. **??** zeigt die Renderfunktion der Viewkomponente.
In den Zeilen 2 bis 4 werden der kürzeren Schreibweise halber die übergebenen Eigenschaften genommen und den entsprechenden Variablen zugewiesen.

```
1 render () {
2 const {
3   isOpen, toggleEdit, addContact, editContact, deleteContact, contact, contacts
4 } = this.props
5
6 return(
```

```
 7   isOpen
 8     ? <FormContainer
 9       addContact={addContact}
10       editContact={editContact}
11       handleCancel={toggleEdit.bind(this, null)}
12       contact={contact}
13     />
14
15     : <ContactList
16       contacts={contacts}
17       handleOnEditClick={toggleEdit}
18       handleOnDeleteClick={deleteContact}
19     />
20 )}
```

Listing 7.2: *Die Renderfunktion in* `Contacts.js` *beider Prototypen*

In den Zeilen 7 bis 19 wird anhand der `isOpen`–Information entschieden ob das Formular oder die Liste gerendert wird. Das Kontaktformular ist aufgeteilt in eine Container– und eine Viewkomponente, weswegen hier in `Contacts.js` die Containerkomponente gerendert wird. Der `FormContainer` bekommt in Zeile 12 den zu bearbeitenden Kontakt. Ist dieser leer, wird über das Formular ein neuer angelegt. Des Weiteren bekommt er ebenfalls die `toggleEdit()`–Funktion, eine zum Anlegen und eine zum Bearbeiten des Kontakts (Zeilen 9 bis 11).
Der Liste werden alle im Appstate gespeicherten Kontakte, die durch den Container durchgereicht werden, gegeben. Außerdem bekommt sie die `toggleEdit()`–Funktion und eine zum Löschen eines Kontakts. Es ist zu beachten, dass hier bei der `toggleEdit()` das `bind(this, null)` fehlt. Das liegt daran, dass wenn man aus der Liste das Formular öffnet, man den ”Bearbeiten”–Knopf betätigt hat und der zu bearbeitende Kontakt im Formular geladen wird. Dieser Kontakt wird dann für die Dauer seiner Bearbeitung im `state.editView.contact` gespeichert.

In **??** wurde beschrieben, wie die Daten in den Prototypen angelegt, bearbeitet und gelöscht werden. Darauf verweisend wird hier nun gezeigt, wie diese Funktionen zum Einsatz kommen.

7.2 Offlinefunktionalität

Die Prototypen *amilia-qouch* und *amilia-rdx* sind offline verwendbar. Daten, die von NutzerInnen generiert werden, werden sowohl lokal als auch in der Serverdatenbank ge-

speichert. Der Service Worker ist für die Live–Nutzung ausgelegt und funktioniert im Entwicklungsmodus nicht. Die Assets werden erst nach dem Deploy gecacht. Alle in **??** beschriebenen Grundvoraussetzungen werden im Live–Modus von beiden Prototypen erfüllt. Sämtliche umgesetzten Funktionen sind bereits im Entwicklungsmodus sowohl mit, als auch ohne Internetverbindung durchführbar.

7.2.1 Datenspeicherung

Redux Offline speichert, wie in **??** beschrieben, alle im Redux Store verwalteten Daten im Local Storage. **??** zeigt alle gespeicherten Daten im Local Storage.

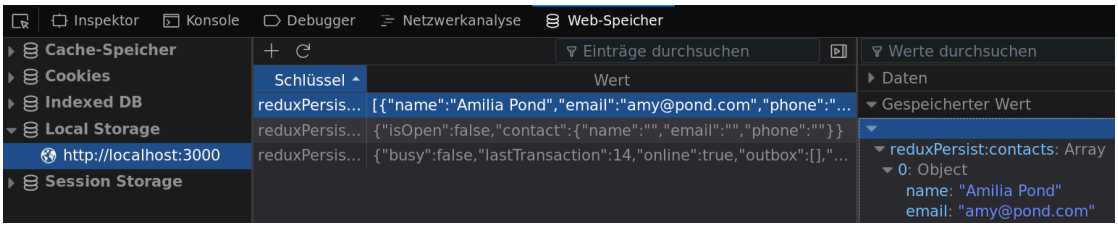


Abbildung 7.1: *Gespeicherte Daten des Prototypen amilia-rdx im Local Storage, Screenshot: Developer Tools im Firefox Browser*

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

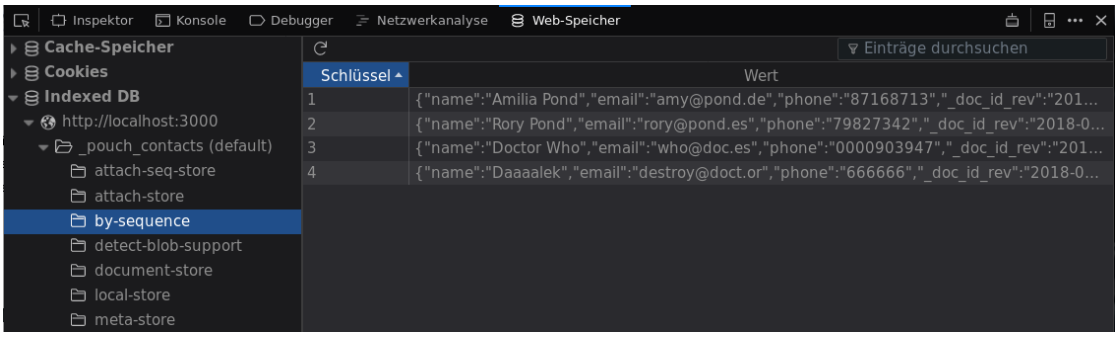


Abbildung 7.2: *Gespeicherte Daten des Prototypen aus amilia-qouch in IndexedDB, Screenshot: Developer Tools im Firefox Browser*

Es ist zu beachten, dass der Prototyp *amilia-rdx* den gesamten Appstate der Anwendung persistiert und nicht nur die erstellten Kontakte, so wie es beim Prototypen *amilia-qouch* der Fall ist.

7.2.2 Datenbanksynchronisation

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

```
1 localDB.sync(remoteDB, {
2   live: true,
3   retry: true
4 })
```

Listing 7.3: *Synchronisation zwischen PouchDB und CouchDB im Prototyp amilia-qouch*

Redux Offline nimmt den EntwicklerInnen auch Arbeit bei der Datensynchronisation ab. Alle Daten, die sich in der Queue befinden, werden automatisch an den Server gesendet, sobald eine Internetverbindung besteht. Das funktioniert jedoch nicht, wenn der Server nicht erreichbar ist. In der Dokumentation von Redux Offline steht, dass die Aktion so oft wiederholt wird, bis die Anwendung wieder mit dem Internet verbunden ist [**?**]. Allerdings wird die Aktion abgebrochen und die rollback Aktion gefeuert, wenn der Server nicht verfügbar ist.

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

Listing 7.4: *Discard–Konfiguration für amilia-rdx*

Die Discard–Konfiguration bestimmt wann eine Aktion abgebrochen und wann sie immer wieder neu gestartet wird. Im **??** ist in den Zeilen 1 bis 4 abzulesen wie diese Konfiguration überschrieben werden kann. Nun wird die Aktion nur abgebrochen, wenn der Server verfügbar ist und einen HTTP–Status von 400 bis 500 zurückgibt. Die 4xx Statuscodes werden geliefert, wenn die Ursache des Scheiterns der Anfrage clientseitig verantwortet wird. Der HTTP–Statuscode 500 wird bei einem internen Serverfehler als Antwort geliefert. In den darauffolgenden Zeilen wird die eigens implementierte Konfiguration mit der Standardkonfiguration von Redux Offline zusammengeführt. Alle Daten werden nun auch nach einem Serverausfall an den Server gesendet.
Im Gegensatz zum *amilia-qouch*–Prototypen werden die Daten nur in eine Richtung automatisch gesendet. Es gibt keine Implementierung einer automatischen Replikation der Daten vom Server zum lokalen Speicher.

7.3 Konfliktmanagement

Konflikte werden in beiden Prototypen manuell erzeugt, wobei die Vorgehensweise identisch ist. Wie ein Konflikt herbeigeführt werden kann, wird in **??** beschrieben. Durch das regelmäßige Anfragen an den Server wird ermittelt, ob sich die Anwendung im Onlinestatus befindet oder nicht. Konflikte können erzeugt werden, wenn mindestens ein Gerät auf dem die Anwendung läuft, nicht mit dem Internet verbunden ist. Die Anzeige des Netzwerkstatus im Header dient der Kontrolle über diesen Status. Die folgenden Codeausschnitte illustrieren das Konfliktmanagement im Prototypen *amilia-qouch*.
Konflikte werden in CouchDB gespeichert, damit in der Anwendung entschieden werden kann wie damit umgegangen wird. Im **??** werden alle Kontakteinträge geladen. Weil der Parameter in Zeile 3 als Option mitgegeben wird, sind Konfliktinformationen für jeden Kontakt verfügbar. Gibt es unterschiedliche Versionen eines Kontaktes kommt er mit dem Attribut `_conflicts` beim Client an, und zwar in Form einer Liste aus allen korrelierenden Revisionsnummern. In Zeile 8 wird der erste konfliktbehaftete Kontakt, der vom Server kommt, ermittelt. Wenn es einen Konflikt gibt, wird die Funktion `getConflictRevisions()` in Zeile 11 aufgerufen.

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

```
10    if (conflictedContact) {
11      this.getConflictRevisions(conflictedContact)
12    }
13  }
```

Listing 7.5: *Das Laden von konfliktbehafteten Kontakten*

?? zeigt die Umsetzung der `getConflictRevisions()`. Hier wird die konkurrierende Version des Kontakts ermittelt. Außerdem wird die Herkunft jeder Version festgestellt und das Öffnen eines Konfliktdialogs eingeleitet.
In Zeile 2 werden die Variablen `contactMe` und `contactYou` initialisiert. Die erste repräsentiert die lokale Version, `contactYou` steht für die Version die aus der CouchDB kommt. In Zeile 4 wird überprüft, ob die übergebene Version des Kontakts mit der zuletzt bearbeiteten übereinstimmt. Entsprechend werden die Variablen `contactMe` und `contactYou` befüllt. Die übergebene Version ist die von CouchDB festgelegte gewinnende Revision. Die andere, konkurrierende Version wird in Zeile 8, bzw. in Zeile 13 durch die Übergabe der Revisionsnummer im Parameter geladen. Die ID ist bei beiden Versionen identisch.
Dann wird das Öffnen des Konfliktdialogs durch das Aktualisieren des Appstatus initialisiert. Die beiden Kontaktversionen werden ebenfalls in den State geladen, um im Dialog korrekt angezeigt zu werden.

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

```
22         contactYou
23       }
24     })
25 }
```

Listing 7.6: *Das Ermitteln von konkurrierenden Kontaktversionen*

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

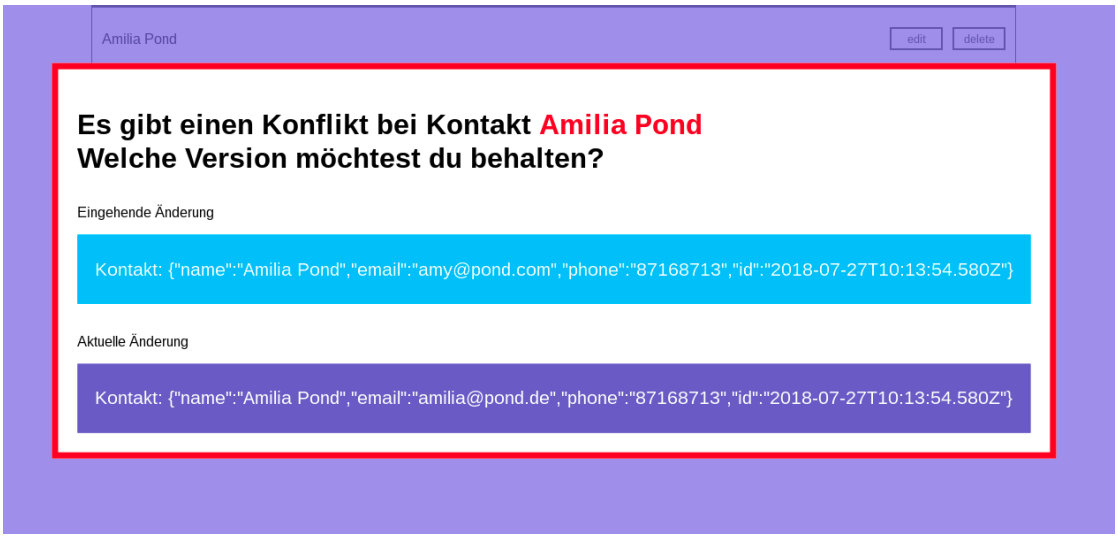


Abbildung 7.3: *Konfliktdialog des Prototypen amilia-qouch*

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

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

Listing 7.7: *Das Eliminieren der verlierenden Version*

In Zeile 3 wird die verlierende Revision gelöscht.

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

7.4 Installationsanleitung

Beide entwickelten Prototypen sind auf der beigelegten CD zu finden. Für die Ausführung beider Prototypen ist die Installation von Node.js eine Voraussetzung. Ein Node.js–Installationsprogramm steht unter `https://nodejs.org/en/download/` zum Download bereit. Um die Prototypen zu starten müssen folgende Schritte ausgeführt werden.

7.4.1 amilia-qouch

Für die Ausführung dieser Anwendung muss CouchDB auf dem Anwendungsgerät installiert sein. Zur Installation von CouchDB kann die Anleitung auf `http://docs.couchdb.org/en/2.1.2/install/index.html` verwendet werden. Es muss außerdem ein User ”admin” mit dem Passwort ”admin” existieren, um auf die Datenbank zugreifen zu können.
Dann müssen die Cross-Origin Resource Sharing (CORS) Optionen aktiviert werden. Das ist eine Webtechnologie, die es Webanwendungen erlaubt, Ressourcen von einer anderen, entfernten Domain zu benutzen. Es gibt bereits ein Skript, welches dies durchführt. Dazu müssen folgende Befehle ausgeführt werden.

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

Waren diese Schritte erfolgreich, kann die Anwendung gestartet werden.

1. Zuerst muss der Ordner *amilia-qouch* auf den Computer kopiert werden.

2. Dann werden dort mit dem folgenden Befehl alle Abhängigkeiten installiert.

```
npm install
```

3. Durch den Aufruf

```
npm start
```

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

7.4.2 AMILIA-RDX

1. Auch hier muss der Ordner *amilia-qouch* auf den Computer kopiert werden.

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

3. Durch die Aufrufe

```
npm run server
npm start
```

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

7.5 TESTFÄLLE

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

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

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

Kontakt anlegen:
Die Anwendung muss zu jedem Zeitpunkt in der Lage sein, einen Kontakt mit jedem seiner Attribute anzulegen. Dazu muss der Kontakt immer lokal gespeichert und sobald eine Internetverbindung besteht, auf dem Server 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 löschen zu können. Das Löschen eines Kontakts im Offlinestatus ist für die Konfliktherbeiführung erforderlich.

8 Evaluation

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

8.1 Manuelle Tests

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

Das Ergebnis wird als der Zustand der App bezeichnet, wie er nach der Synchronisation beider Geräte ist. Das erwartete Ergebnis ist der Zustand, der eintreten soll. Ein Ergebnis, in dem keine BenutzerInnendaten verloren gehen.
Das tatsächliche Ergebnis ist, wie der Name schon sagt, das welches eingetreten ist. Weicht das tatsächliche Ergebnis von dem erwarteten ab, sind die Ansprüche in diesem Testdurchlauf nicht erfüllt.

8.1.1 Erste Testreihe

In der ersten Testreihe werden Aktionen ausgeführt, während beide Geräte mit dem Internet verbunden sind. Diese Tests zeigen, dass die Anwendung grundsätzlich auf mehreren Geräten benutzbar ist.

Kontakt anlegen

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

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

Kontakt bearbeiten

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

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

Kontakt löschen

Der Kontakt mit dem Namen ”Amilia Pond” ohne E–Mail–Adresse, wird im Firefox gelöscht. Der andere Adressbucheintrag wird im Chrome gelöscht.

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

8.1.2 Zweite Testreihe

In der zweiten Testreihe werden Aktionen ausgeführt, während ein Browser mit dem Internet verbunden ist, der andere jedoch nicht. Der Browser Chrome wird in den Offlinestatus versetzt und Firefox bleibt online. Nachdem in beiden Browsern die beschriebene

Aktion vollendet wurde, wird im Chrome der Internetzugriff wieder aktiviert, sodass die Daten synchronisiert werden können.

Kontakt anlegen

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

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

Kontakt bearbeiten

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

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

Kontakt bearbeiten und denselben Kontakt löschen

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

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

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

Kontakt löschen und denselben Kontakt bearbeiten

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

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

Kontakt löschen

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

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

8.1.3 Dritte Trestreihe

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

Kontakt anlegen

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

Erwartetes Ergebnis
Es existieren zwei Kontakteinträge mit dem Namen ”Amilia Pond” auf jedem Gerät.

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

Kontakt bearbeiten

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

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

Kontakt bearbeiten und denselben Kontakt löschen

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

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

Kontakt löschen

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

Erwartetes Ergebnis
Die Kontaktliste ist leer.

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

8.1.4 Übersicht der Testergebnisse

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

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

Tabelle 8.1: *Testergebnisse*

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

8.2 Auswertung

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

8.2.1 Auswertung der manuellen Tests

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

Kollaborationsfähigkeit

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

Offlinefunktionalität und Konfliktmanagement

Während der zweiten und dritten Testreihe waren entweder ein oder beide Geräte für die Ausführung bestimmter Aktionen offline. Der Fokus dieser Tests lag auf dem eventuell eintretenden Datenverlust im Konfliktfall. **??** zeigt, dass das Anlegen und Löschen weder bei *amilia-qouch*, noch bei *amilia-rdx* ein Problem darstellte. Beim Anlegen eines Kontakts war dieser auf beiden Geräten doppelt gelistet. Auf diesem Wege ging kein Kontakt verloren. Auch das Löschen eines Kontakts erfolgte reibungslos. Der gelöschte Kontakt war nach der Synchronisation auf keinem der Geräte mehr vorhanden.
Sowohl in der zweiten, als auch in der dritten Testreihe sind im Redux Offline–Prototyp beim Bearbeiten, bzw. beim Bearbeiten und Löschen eines Kontakts, Daten verloren gegangen. Wo im Prototypen *amilia-qouch* die aufgetretenen Konflikte gespeichert wurden, um sie auf Benutzungsebene zu lösen, ging im anderen Protoypen *amilia-rdx* einer der konfliktbehafteten Datensätze verloren. Bei gleichzeitigem Bearbeiten eines Kontakts gewann die Version, die zuletzt synchronisiert wurde, die andere wurde überschrieben. Wurde ein Kontakt auf einem Gerät bearbeitet und auf dem anderen gelöscht, war es in jedem Fall nicht mehr in der Liste. Löschungen haben immer Vorrang, wodurch ebenfalls Daten verloren gehen.

Beobachtungen

Zusammenfassend lässt sich sagen, dass Redux Offline per se nicht auf mehreren Geräten offlinefähig ist. Bei den Testfällen in **??**, die während der Entwicklung stetig – auf einem Gerät – durchgeführt wurden, bestand *amilia-rdx* jeden Test. Auf nur einem Gerät ist es jedoch schwierig, Konflikte zu erzeugen. Deswegen ist die LWW Strategie von Redux Offline für die Verwendung auf einem Gerät akzeptabel.
Der Prototyp *amilia-qouch* hingegen hat alle Tests erwartungsgemäß bestanden. Immer wenn es zu einem Konflikt gekommen war, konnte dieser über den Konfliktdialog einfach und vor allem korrekt gelöst werden. In keinem Testdurchlauf gingen Daten verloren.

8.2.2 Erfüllung der Anforderungen

In **??** wurden die funktionalen Anforderungen an eine offlinefähige, kollaborative Anwendung erarbeitet. Es wird überprüft, welche dieser Anforderungen prototypisch erfüllt wurden und welche nicht.

F1 Die Anwendung muss auf mindestens zwei Geräten funktionieren.
Die Auswertung der ersten Testreihe in **??** zeigt, dass beide Prototypen auf zwei Geräten funktionieren, sofern beide Geräte mit dem Internet verbunden sind. Da die Testdurchläufe nur auf zwei Geräten durchgeführt wurden, ist keine Aussage darüber zu treffen, ob die Anwendung auch auf mehr als zwei Geräten uneingeschränkt funktioniert.

F2 Die Anwendung soll fähig sein, den Netzwerkstatus zu ändern.
Diese Anforderung ist indirekt erfüllt. Die Prototypen wurden für die Browser Firefox und Chrome entwickelt. Beide ermöglichen es in den Einstellungen, den Offlinemodus zu aktivieren und so die Anwendung vom Internet zu trennen. Auch durch das Ausschalten des Servers kann die Verbindung zwischen den beiden Geräten, auf denen die Anwendung läuft, unterbrochen werden.

F3 Die Anwendung muss den Netzwerkstatus erkenntlich machen.
Im Header eines jeden Prototypen wird der aktuelle Netzwerkstatus der Anwendung korrekt angezeigt. Dafür wird das externe Modul React Detect Offline verwendet, dem die Adresse des Servers übergeben wird. Über diese Adresse wird alle fünf Sekunden geprüft, ob der Server erreichbar ist. Aus letzterem folgt, dass mit einer leichten zeitlichen Abweichung in der Anzeige zu rechnen ist.

F4 Die Anwendung muss fähig sein, die Kontakte unabhängig vom Netzwerkstatus zu laden, sofern diese einmal aus dem Netzwerk geladen wurden.
Einmal geladen, werden alle Kontakte in beiden Prototypen aus dem lokalen Speicher

gelesen, wenn die Anwendung offline ist oder der Server nicht läuft. Der *amilia-qouch* Prototyp liest die Kontakte dann aus der IndexedDB und der *amilia-rdx* Prototyp aus dem Local Storage.

F5 Die Anwendung muss die Möglichkeit bieten, unabhängig vom Netzwerkstatus einen Kontakt anzulegen, zu bearbeiten oder zu löschen.
Während der Entwicklung wurden stetig Tests zur Offlinefunktionalität auf einem Gerät durchgeführt (siehe **??**). Diese bestätigen, dass in jedem Prototypen unabhängig vom Netzwerkstatus jederzeit ein Kontakt angelegt, bearbeitet oder gelöscht werden kann.

F6 Die Anwendung muss alle Kontakte sowohl lokal als auch serverseitig persistieren. Die Konntakte müssen identifiziert und versioniert werden.
Auch diese Anforderung erfüllen beide Prototypen. Die lokale Speicherung erfolgt, wie bereits beschrieben, im Local Storage oder in IndexedDB. Serverseitig werden die Kontakte im Prototypen *amilia-qouch* in einer CouchDB persistiert. Da Redux Offline keine serverseitige Datenbank zur Verfügung stellt, werden die Daten in diesem Prototypen in einer serverseitigen JSON–Datei gespeichert. Die Identifizierung der Kontakte erfolgt in beiden Prototypen über einen Zeitstempel. In *amilia-rdx* erfolgt die Versionierung ebenfalls über den Zeitstempel. Der Prototyp *amilia-qouch* verwendet hierzu eine Revisionsnummer. Diese Revisionsnummer ist der Dreh– und Angelpunkt für das Konfliktmanagement in CouchDB.

F7 Die Anwendung muss die lokal gespeicherten Kontakte mit denen in der Serverdatenbank persistierten synchronisieren, sobald die Anwendung mit dem Internet verbunden ist.
Ist die Anwendung offline oder der Server ausgeschaltet, werden alle Aktionen die während dieses Zeitraums durchgeführt werden, mit der Serverdatenbank, bzw. der JSON–Datei synchronisiert. Diese Funktionalität wird von allen in den Prototypen verwendeten Technologien bereitgestellt.

F8 Die Anwendung muss die Möglichkeit bieten, die Konfliktmanagementstrategien der zu untersuchenden Technologien zu evaluieren.
Um die Konfliktmanagementstrategien der verwendeten Technologien untersuchen zu können, müssen Konflikte herbeigeführt werden. Die Möglichkeit, die Anwendung auf zwei Geräten nutzen zu können und diese vom Internet zu trennen, macht den Vorgang durchführbar. Die Visualisierung des Netzwerkstatus im Header der Kontaktliste vereinfacht diesen Vorgang. Die in **??** durchgeführten manuellen Tests waren nur aufgrund dieser Funktionalitäten ausführbar. Die Auswertung erfolgte in **??**, wodurch diese Anforderung erfüllt ist.

F9 Die Anwendung soll Konflikte speichern, sofern diese auftreten. Das heißt, es muss solange nachvollziehbar sein, welche Versionen eines Kontakts konfliktbehaftet sind, bis der Konflikt gelöst ist.
Redux Offline bietet keine Möglichkeit, Konflikte zu speichern, weswegen diese Funktionalität für *amilia-rdx* nicht umgesetzt werden konnte. Im anderen Prototypen werden die Revisionsnummern der konfliktbehafteten Kontakte im Kontakt gespeichert, wodurch der Prototyp *amilia-qouch* diese Anforderung erfüllt.

F10 Die Anwendung muss die Möglichkeit bieten, Konflikte zu lösen, sofern diese auftreten.
Wenn Konflikte auftreten, werden sie in beiden Prototypen gelöst. Im Redux Offline–Prototypen wird dazu im groben die LWW–Strategie verwendet. Bei Löschungen ist es egal, welche Änderung zuletzt durchgeführt wurde. Löschungen gewinnen immer. Im CouchDB–Prototypen werden Konflikte zuerst automatisch durch den in **??** beschriebenen Algorithmus gelöst. Da dieser Algorithmus die gewinnende Version mehr oder weniger zufällig auswählt, werden alle konfliktbehafteten Versionen gespeichert. So können sie auf Anwendungsebene gelöst werden, was über den Modaldialog passiert.

F11 Die Anwendung muss sicherstellen, dass auf keinen Fall Daten verloren gehen.
Durch das praktische Konfliktmanagement in CouchDB gehen im Prototypen keine Daten verloren. In *amilia-rdx* gehen in jeder getesteten Konfliktsituation Daten verloren. Durch die automatische Festlegung der gewinnenden Version wird die andere überschrieben, unabhängig davon, welche die richtige Version ist.

8.2.3 Implementierungsaufwand

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

Einbindung der Technologien

amilia-qouch
Der Aufwand, eine Anwendung mit PouchDB und CouchDB zu schreiben, ist gering. Man muss nur beide Datenbanken installieren und im Code instanziieren. PouchDB dient als Schnittstelle zu CouchDB, weswegen nur das simpel gehaltene API von PouchDB be-

nutzt wird. Die Dokumentationen beider Technologien sind sehr ausführlich und tragen ausgesprochen gut zum Verständnis der Funktionsweise bei.

amilia-rdx

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

Lesbarkeit des Codes

amilia-qouch

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

amilia-rdx

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

Menge des geschriebenen Quellcodes

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

amilia-qouch

Es wurden 388 Zeilen Code hinzugefügt.

amilia-rdx

Es wurden 544 Zeilen Code hinzugefügt.

9 Zusammenfassung und Ausblick

Das Thema Offline First ist sehr interessant und offlinefähige Anwendungen erfreuen sich immer größerer Beliebtheit. Die Entwicklung dieser Anwendungen wird auch in Zukunft weiterhin von verschiedenen, unterstützenden Technologien vorangetrieben werden. Das Thema des Konfliktmanagements geht mit Offline First–Apps einher, wird jedoch nicht von allen offlinefähigen Systemen hinreichend beachtet. Das mag unter anderem daran liegen, dass die meisten bestehenden Konzepte zur Konfliktlösung nicht auf alle Datenobjekte anwendbar sind.

Der Operational Transformation–Algorithmus ist eine Lösung für kollaborative Textverarbeitung. Er ist jedoch nur für Text in verschiedener Form konzipiert und nicht für allgemeine Datenobjekte. Der CRDT–Datentyp ist auf spezialisierte Datenstrukturen wie Listen und Zähler beschränkt und kann ebenfalls nicht auf ein generisches JSON–Datenobjekt angewandt werden, was die Verwendung dessen stark einschränkt.
Überraschend viele Systeme verwenden den Last-Write-Wins–Ansatz in der einen oder anderen Form. Da gibt es beispielsweise die Datenbank Cassandra, deren Konfliktmanagementstrategie LWW ist [**?**]. Aber auch die Datenbanklösung Realm verwendet LWW für Datenaktualisierungen und eine abgewandelte Version davon für das Löschen von Objekten, denn Löschungen gewinnen immer. Das Speichern aller Aktionen und Abarbeitung dieser in einer Queue, so wie Redux Offline es handhabt, ist ebenfalls eine Variante des LWW–Ansatzes. So gewinnen die Aktionen, die zuletzt ausgeführt werden. Der Nachteil dieser Strategie ist, dass NutzerInnendaten willkürlich verloren gehen können.

Ein weiterer Punkt, der von vielen offlinefähigen Systemen ausgelassen wird, ist die Datenbanksynchronisation. Die meisten in Kapitel 2 aufgeführten Frameworks und Bibliotheken stellen keine Lösung für die serverseitige Datenspeicherung bereit. Aber sobald die Anwendung auf mehreren Geräten benutzt wird, ist eine Backendimplementierung notwendig. Wenn die Anwendung zusätzlich offlinefähig sein soll, bedarf es eines Synchronisationsalgorithmus'. Bei der Datensynchronisation zwischen dem lokalen Speicher und einer Serverdatenbank kann es immer zu Konflikten kommen. Diese müssen gespeichert werden, um von den NutzerInnen gelöst werden zu können. Nur so kann sichergestellt werden, dass keine Daten verloren gehen.
Sobald in der Anwendung Daten generiert werden, muss also bei der Wahl einer Tech-

nologie zur Erstellung einer offlinefähigen App darauf geachtet werden, dass diese Technologie eine Datenbanklösung bereitstellt, die sich dieser Problematik widmet und sie erfolgreich löst.

Die im Rahmen dieser Arbeit entwickelten Prototypen können eine Kontaktliste kollaborativ verwalten. Da beide Prototypen mit dem Werkzeug Create React App erstellt wurden, welches einen Service Worker mitliefert, werden die Assets erst nach dem Deploy gecacht.

Der Prototyp *amilia-qouch* speichert die erstellten Kontakte zunächst mit der Hilfe von PouchDB in der Browserdatenbank IndexedDB und repliziert sie dann zu der CouchDB–Serverdatenbank. Wenn Konflikte durch gleichzeitiges Bearbeiten desselben Kontakts entstehen, werden diese in der CouchDB mittels des internen Replikationsprotokolls gespeichert und können clientseitig aufgelöst werden. Nur durch die Konfliktlösung durch Menschen kann sichergestellt werden, dass die korrekte Änderung gespeichert wird und keine Daten verloren gehen. Somit erfüllt *amilia-qouch* sämtliche in **??** genannten Voraussetzungen an eine offlinefähige Anwendung.

Der Prototyp *amilia-rdx* speichert die erstellten Kontakte mit Hilfe von Redux Offline im Local Storage. Da Redux Offline keine Datenbanklösung bereitstellt, wurde ein einfacher Server entwickelt, über den die Daten in einer JSON–Datei persistiert werden. Wenn Konflikte durch gleichzeitiges Bearbeiten desselben Kontakts entstehen, werden diese mittels des LWW–Ansatzes gelöst. Bei gleichzeitiger Löschung und Bearbeitung eines Kontakts hat die Löschung Vorrang. Das hat zur Folge, dass willkürlich Daten verloren gehen, wodurch nicht alle Voraussetzungen an eine offlinefähige Anwendung erfüllt sind.

Die Evaluierung wurde in mehrere Testphasen unterteilt. Zuerst wurden manuelle Tests zur Offlinefähigkeit und zum Konfliktmanagement durchgeführt, die anschließend ausgewertet wurden. Die Ergebnisse dieser Tests bestätigen die Resultate aus den Überprüfungen, welche gestellten Anforderungen an eine offlinefähige App von den entwickelten Prototypen erfüllt wurden. Aufgrund der Tatsache, dass unter Verwendung des mit Redux Offline erstellten Prototypen Daten verloren gehen, hat diese Anwendung die Tests nicht bestanden.
Auch in der Auswertung aus Entwicklungsperspektive, in der der Implementierungsaufwand, die Codelesbarkeit und die Menge des geschriebenen Quellcodes begutachtet wurden, schneidet *amilia-rdx* schlechter als *amilia-qouch* ab. Es ist zu erwähnen, dass die Auswertung unter der Annahme stattfand, mit keiner der verwendeten Technologien vertraut zu sein. Die Ergebnisse wären unter Umständen besser für den Redux Offline–Prototypen ausgefallen, wenn von der Kenntnis der Verwendung von Redux augegangen

worden wäre.

Die Untersuchungen lassen sich auf weitere offlinefähige Systeme ausweiten. Da die Konfliktmanagementstrategien untersucht werden, sollte der Fokus auf Datenbanklösungen liegen. Denn bei der Verwendung der App auf nur einem Gerät und ohne eine Serveranwendung müssen keine Konflikte behandelt werden, da sie nicht auftreten.
Die Datenbanklösung Realm ist sehr interessant, da sie verspricht Konflikte in Offline First–Anwendungen ohne Datenverlust zu lösen. Die Untersuchung von Realm ist unter Anderem deswegen spannend, weil sie diese Versprechungen trotz der Verwendung des Last-Write-Wins–Ansatzes und des Vorrangs von Löschungen machen.

Abkürzungen

API	Application Programming Interface
App	Applikation
CAP	Consistency Availability Partition tolerance
CRDT	Conflict-free replicated data type
DB	Datenbank
DBMS	Datenbankmanagementsystem
JSON	JavaScript Object Notation
LWW	Last-Write-Wins
OT	Operational Transformation
UI	User Interface
WLAN	Wireless Local Area Network

Glossar

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

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
: Eine Schicht zwischen Anwendung und Betriebssystem.

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

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 Redux Offline 9

Quellcodeverzeichnis

Literaturverzeichnis

[Acu] Acuña, Raúl G.: *react-native-offline*. https://github.com/rauliyohmc/react-native-offline, . – Zugriff: 12.04.2018 2.2.5

[ALS10] Anderson, J. C. ; Lenhardt, Jan ; Slater, Noah: *CouchDB: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, 2010. – ISBN 978-0-596-15589-6. – oreilly.com 3.5

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

[cou] *CouchDB - relax*. https://couchdb.apache.org/, . – Zugriff: 12.04.2018 2.2.8

[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 3.4.2

[Evä17] Eväkallado, Jani: Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native. In: *HACKERnoon* (2017), 3. – Zugriff: 12.04.2018 2.1

[fal] *Eight Fallacies of Distributed Computing*. https://blog.fogcreek.com/eight-fallacies-of-distributed-computing-tech-talk/, . – Zugriff: 12.04.2018 3.1

[Fra09] Fraser, Neil: Differential Synchronization. Version: Jan 2009. https://neil.fraser.name/writing/sync/eng047-fraser.pdf. 2009. – Research Report. – 8 S. 3.4.2

[hoo] *Hoodie – The Offline First Backend*. http://hood.ie, . – Zugriff: 12.04.2018 2.2.7

[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 3.4.2

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

[pou] *pouchdb - Tha Database that Syncs!* https://pouchdb.com/learn, . – Zugriff: 12.04.2018 2.2.9

[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 2.2.2

[rea17b] *BUILD BETTER APPS, FASTER WITH REALM - An Overview of the Realm Platform*. https://www2.realm.io/whitepaper/realm-overview-registration, october 2017. – Zugriff: 12.04.2018 2.2.2

[reda] *Redux Offline Release BREAKING: Migrate to Store Enhancer API*. https://github.com/redux-offline/redux-offline/releases/tag/v2.0.0, . – Zugriff: 12.04.2018 2.2.4

[redb] *Redux Offline*. https://github.com/redux-offline/redux-offline, . – Zugriff: 12.04.2018 2.2.4

[redc] *react-native-offline*. https://github.com/rt2zz/redux-persist, . – Zugriff: 12.04.2018 2.2.4

[San16] Sanford, Clark: Persistence is Key: Using Redux-Persist to Store Your State in LocalStorage. In: *Medium* (2016), 12. – Zugriff: 12.04.2018 2.2.4

[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. 3.4.3

[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. 3.4.3, 3.4.3

[Sto] Stolyar, Arthur: *offline-plugin*. https://github.com/NekR/offline-plugin, . – Zugriff: 12.04.2018 2.2.6

Anhang

Eidesstattliche Erklärung

CD-Inhalt

Auf der beigefügten CD befinden sich

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