BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Masterarbeit

Medieninformatik
Fachbereich VI – Informatik und Medien

Untersuchung der Konfliktmanagementstrategien verschiedener offlinefähiger Systeme

---

Berlin, den 30. Juni 2018

Autorin:
Jacoba BRANDNER
Matrikelnummer:
833753

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

Abstract

In dieser Arbeit wird ...
Muss das in deutsch und englisch geschrieben werden?

Abstract

This work includes ...

Inhalt

1 Einführung

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

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

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

1.1 Motivation

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

1.2 Zielstellung

Hier schon reinschreiben welche Technologien ich nutze? Wenn nicht, ist es bei den Szenarien leichter zu verstehen, dass die noch ‘nackig‘ sind.
Wichtig: Offline nutzbar ohne Datenverlust.
Untersuchung des Verhaltens bei Konflikten (verursacht durch paralleles Arbeiten ohne Internetverbindung).
Wie leicht/schwer ist es zu implementieren? konkret werden. Definition offlinefähig
Dazu 3 Stufen: 1 Person (nativ)

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

2 Grundlagen

Was bedeutet offlinefähig?
Harte, weiche, mittlere probleme (verschiedene Stufen von offlinefähig)
Native Applikations (Apps) existieren und funktionieren grundsätzlich solange offline, bis sie versuchen online Daten abzurufen.
bei nativen Apps ist das Problem bei der Datenverteilung

2.1 Offline First

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

2.1.1 Lokale Speicherung im Cache

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

AppCache

Um mehr Kontrolle darüber zu bekommen, was wann und für wie lange gespeichert werden soll, wurde der Application Cache (AppCache) zur Hypertext Markup Language (HTML)-Spezifikation hinzugefügt. Im Juni 2016[1] wurde der AppCache wieder aus den Web-Standards entfernt, und wird nicht mehr empfohlen. In der Theorie stellte sich der Application Cache als einfach anzuwenden und unproblematisch dar. Um eine webbasierte Anwendung

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

offline auszuliefern wurde benötigte es eine Textdatei – der `cache manifest`-Datei – mit der Endung `.appcache`. Dort wurden alle Ressourcen aufgelistet, welche der Browser cachen sollte. Die Datei wurde dann über das `manifest`-Attribut in die HTML-Dateien der Webanwendung eingebunden werden.

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

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

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

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

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

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

Service Worker

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

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

Mit dem Service Worker können wie mit dem App Cache statische Ressourcen sofort beim ersten Besuch der Seite im Cache gespeichert werden. Es lässt sich hierbei unterscheiden, ob die Daten vor der ersten Verwendung, oder später im Cache gespeichert werden sollen. Für den ersten Fall eignen sich statische Inhalte wie Schriften oder JavaScript–Dateien, für den zweiten größere Ressourcen die nicht sofort benötigt werden.
Zusätzlich bietet der Service Worker die Möglichkeit auf Interaktionen zu reagieren. Den NutzerInnen kann angeboten werden bestimmt Inhalte der Seite, wie zum Beispiel ein Video, später, bzw. offline anzuschauen. Diese werden dann im Cache gespeichert und sind somit offline verfügbar. Service Worker erlauben außerdem den Zugriff auf Push-Benachrichtigungen und das Background Sync Application Programming Interface (API). Die Hintergrundsynchronisation kann einmalig oder in festgelegten Intervallen stattfinden und ist besonders für nicht dringende Aktualisierungen wertvoll [Arc14].

2.1.2 Lokale Speicherung im Browser

Der zweite Schritt Daten offline verfügbar zu machen ist sie im Browser zu speichern. Einige Konzepte hierfür werden im Folgenden erläutert.

Web Storage

Das Web Storage API ist ein Web-Standard mit dessen Hilfe Daten als Schlüssel / Wert Paare im Browser gespeichert werden können. Es wird, wie Abbildung 2.2 zeigt, von allen relevanten Browsern unterstützt. Web Storage umfasst zwei Mechanismen. Den Session Storage und den Local Storage.
Der Session Storage existiert nur so lange der Browser geöffnet ist. Das heißt alle Daten die im Session Storage gespeichert werden, existieren nicht mehr sobald der Browser geschlossen wird. Daten die im Local Storage gespeichert sind, existieren dort solange bis der Browser Cache geleert wird [webb].

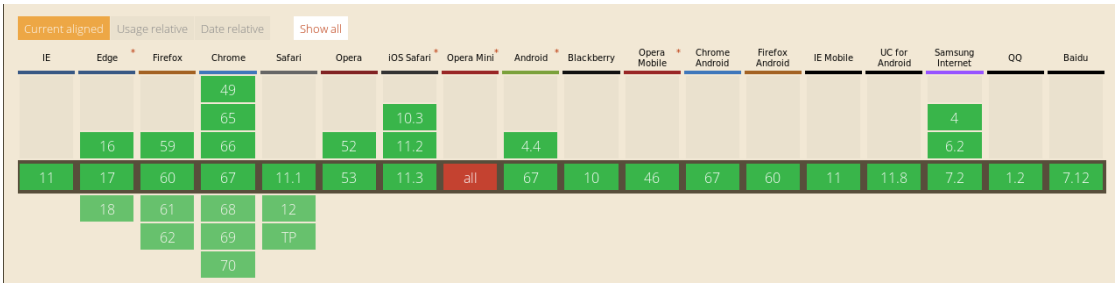


Abbildung 2.2: *Browserkompatibilität für Web Storage, Quelle: [canc]*

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

WebSQL

IndexedDB

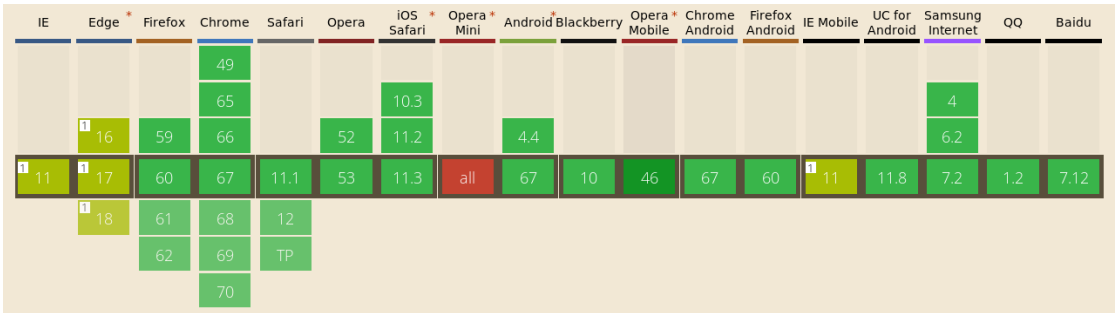


Abbildung 2.3: *Browserkompatibilität für IndexedDB, Quelle: [cana]*

IndexedDB 2

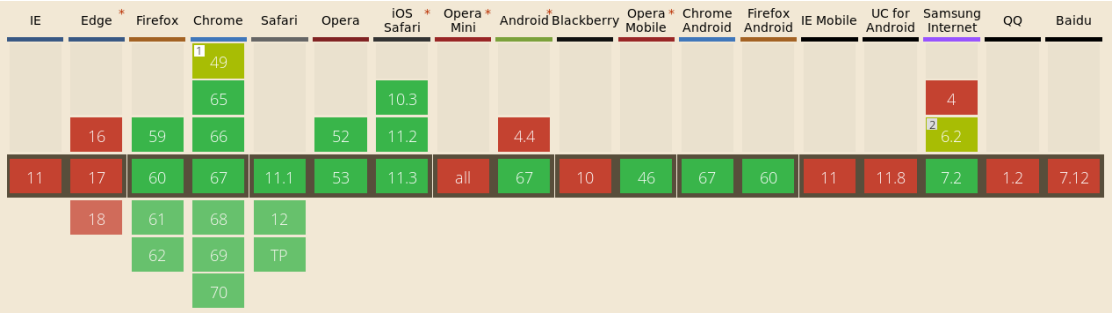


Abbildung 2.4: *Browserkompatibilität für IndexedDB 2.0, Quelle: [cana]*

https://www.w3.org/TR/IndexedDB-2/

2.1.3 Datenbanksynchronisation

CouchDB

Apache CouchDB™ist ein Datenbank Management System (DBMS) das seit 2005 als freie Software entwickelt wird. Die dokumentenorientierte Datenbank (DB) funktioniert sowohl als einzelne Instanz, als auch im Cluster, in dem ein Datenbanksserver auf einer beliebig großen Anzahl an Servern oder Virtuelle Maschines (VMs) ausgeführt werden kann. So kann die Datenschicht beliebig skaliert werden, um die Anforderungen vieler BenutzerInnen zu erfüllen. CouchDB verwendet das Hypertext Transfer Protocol (HTTP)–Protokoll und JavaScript Object Notation (JSON) als Datenformat, weswegen es mit jeder Webfähigen Anwendung kompatibel ist. CouchDB wird über ein REpresentational State Transfer (REST)ful HTTP API angesprochen. Mit den für RESTful Services standardisierten Methoden z. B. GET, POST, PUT, DELETE können die Daten abgerufen und. manipuliert werden.
Das implementierte Replikationsmodell erlaubt die Synchronisation bzw. bidirektionale Replikation zu verschiedenen Geräten ist genau die Besonderheit, die CouchDB als eine Offline–Datenbank auszeichnet. Dessen Funktionsweise wird in Unterabschnitt 2.3.3 detailliert beschrieben. Dieses Protokoll ist die Grundlage für Offline First Anwendungen. Das Replikations-API von CouchDB bietet die Möglichkeit, eine Datenbank kontinuierlich oder selbstgesteuert mit einer anderen zu synchronisieren. So kann beispielsweise eine CouchDB-Instanz auf dem Mobiltelefon und eine auf dem Laptop bestehen und beide können sich bei bestehender Internetverbindung synchronisieren. Da so die gespeicherten Daten aus dem lokalen Speicher gelesen werden, sind ein schnelles Interface und eine geringe Latenz die positive Folge. Wenn Konflikte auftreten, beispielsweise

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

PouchDB

Was benutzt Poch wann? – IndexedDB, Web SQL, LevelDB Als Ergänzung zu CouchDB kann PouchDB verwendet werden. PouchDB ist eine Open–Source–JavaScript–Datenbank, die so konzipiert wurde, dass sie im Browser läuft. PouchDB ermöglicht es Anwendungen zu erstellen, die sowohl offline als auch online funktionieren. Daten können lokal gespeichert werden, sodass alle Funktionen der Anwendung auch im Offline–Modus zur Verfügung stehen. Daten werden unabhängig von der nächsten Anmeldung (des nächsten Onlinezugangs) zwischen **Clients**, CouchDB oder kompatiblen Servern synchronisiert. PouchDB läuft auch in Node.js[2] und kann als direkte Schnittstelle zu CouchDB–kompatiblen Servern verwendet werden [poua].

2.1.4 Progressive Web Apps raus?

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

Responsive
: to fit any form factor

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

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

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

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

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

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

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

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

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

Näher erläutern? [Rus15].

2.2 Konflikte

was ist ein Konflikt? Ab wann kann ein Konflikt vom System gelöst werden, ab wann nicht? (siehe git: gleichzeitige Änderung an unterschiedlichen Stellen: kein Problem. Gleichzeitige Änderung an derselben Stelle: Konflikt!)
Verteilte Systeme: Das ist ein mächtiger Begriff für viele Ideen und Konzepten, aber es läuft in der Regel darauf hinaus: Da sind zwei oder mehr Computer, die durch ein Netzwerk verbunden sind und es wird versucht, dass einige der Daten auf beiden Computern gleich aussehen. ==> Ein System das zuverlässig über ein Netzwerk funktioniert.
Zwei Geräte, ein Server, über Netzwerk verbunden.

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

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

2. Die Latenz ist gleich null
Glasfaserkabel werden durch Mikrowellen (oder andere Technologien) ersetzt um Mil-

lisekunden an Zeit zu sparen. Das würde nicht passieren, wäre die Latenz bei null. Es dauert nun mal eine gewisse Zeit(ms) wenn ein Signal eine (geografisch)weite Strecke zurücklegen muss — Die Latenz ist nicht gleich null.

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

4. Das Netzwerk ist sicher
Der HEARTBEAT-BUG[3], der im Jahr 2014 behoben wurde und die Sicherheitslücke im ICE-Wireless Local Area Network (WLAN) im Jahr 2016[4] sind nur zwei Beispiele die zeigen, dass das Netzwerk nicht sicher ist.

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

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

6. Es gibt eineN AdministratorIn
Es kann beliebig viele AdministratorInnen geben.

7. Die Datentransportkosten sind gleich null
Netflix bezahlte anfang 2014 diversen InternetanbieterInnen dafür, dass Netflix KundInnen bevorzugten Internetzugang haben.

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

2.3 KONFLIKTMANAGEMENTSTRATEGIEN

GIT

Beschreiben wie Git Konflikte löst.

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

2.3.1 CAP Theorem?

2.3.2 Operational Transformation

siehe woot

Operational Transformation (OT) ist eine weit verbreitete Technologie zur Unterstützung von Funktionalitäten in Kollaborativer Software. Sie stammt aus einer im Jahre 1989 veröffentlichten Forschungsarbeit und wurde ursprünglich nur für die gemeinsame Bearbeitung von Klartext-Dokumenten entwickelt [EG89]. Später ermöglichte weitere Forschung OT durch Unterstützung von *Sperrungen*, *Konfliktlösungen*, *Benachrichtigungen*, *Bearbeitung von Baumstrukturierten Dokumenten*, ... zu verbessern und erweitern. Im Jahr 2009: Google Wave, Google Docs

Es wird das Problem untersucht, dass OT in einer idealen Umgebung löst und dadurch zu einem funktionierendem Algorithmus gelangt.

Ziel: Mehrere BenutzerInnen können gleichzeitig an einem Dokument arbeiten, sehen Änderungen der anderen in Echtzeit (live), ohne dass einer Verzögerung durch die Latenz verursacht wird. Gleichzeitig auftretende Mehrfachänderungen sollen nicht zu unterschiedlichen Dokumentenzuständen führen.

Funktionsweise

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

Es gibt die Operationen **Einfügen**

Das Einfügen besteht aus dem eingefügten Text und dessen Position im Dokument (`insert('h', 0)`). Für die Position kann ein Koordinatensystem ermittelt werden (Zeilennummer: Position in Zeile oder einfacher: Dokument wie eine Folge von Zeichen behandeln, also einfach einen nullbasierten Index vergeben.)

und **Löschen**

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

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

Beispiel : Benutzer A fügt an Position 12 das Zeichen 'A' ein Benutzer B fügt am Anfang des Dokuments ein 'B' ein. Die konkurrierenden Operationen sind daher Einfügen (12, 'A') und Einfügen (0, 'B'). Wenn wir die Operation von B einfach an Client A senden und dort anwenden würden, gäbe es ein Problem. Aber wenn die Operation von A an B gesendet, und angewandt wird nachdem Operation B angewandt wurde ist, würde das Zeichen 'A' eine Position zu weit links von der korrekten Position eingefügt werden. Dokumentstatus A und Dokumentstatus B sind nicht identisch.
Daher muss A's `insert(12, 'A')` gegen die Operation von B transformiert werden. So wird berücksichtigt, dass B ein Zeichen vor der Position 12 eingefügt hat (die die Operation `insert(13, 'A')` erzeugt.)
Diese neue Operation kann auf Dokument B nach B's Operation angewandt werden. Die Grundidee von OT besteht darin, die Parameter einer Editieroperation gemäß den Auswirkungen **zuvor ausgeführter** konkurrierender Operationen anzupassen, so dass die transformierte Operation die korrekte Wirkung erzielen und die Dokumentenkonsistenz aufrechterhalten kann.

KRITIK

False-Tie puzzle? *A Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems* und *Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems*

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

fehleranfällig, selbst für OT-Algorithmen, die nur zwei charakteristische Primitive behandeln (Einfügen und Löschen)“ [LL10].
Damit OT funktioniert, muss jede einzelne Änderung an den Daten erfasst werden: “Einen Schnappschuss des Zustands zu erhalten, ist normalerweise trivial, aber das Erfassen von Bearbeitungen ist eine ganz andere Sache. [...] Der Reichtum moderner Benutzerschnittstellen kann dies problematisch machen, besonders in einer browserbasierten Umgebung“ [Fra09].

2.3.3 Das CouchDB Replikationsmodell

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

Das CouchDB Replikationsmodell erlaubt eine nahtlose, peer-to-peer (direkte) Datensynchronisation zwischen beliebig vielen Geräten. Das CouchDB Replikationsprotokoll ist in CouchDB selbst implementiert, das die Serverkomponente abdeckt. Dann gibt es das PouchDB-Projekt, das dasselbe Protokoll in JavaScript implementiert, das auf Browser- und Node.js-Anwendungen abzielt. das deckt Ihre Kunden und dev-Server ab. Schließlich gibt es Couchbase Mobile und Cloudant Sync, die auf iOS und Android laufen und das CouchDB Synchronisationsprotokoll in Objective-C bzw. Java implementieren.
Vektoruhr [5]
content addressable versions: Idee: Nimm den Objektinhalt (content) und jag ihn durch eine Hashfunktion

[5] `https://en.wikipedia.org/wiki/Vector_clock`

schlussendliche Konsistenz

Lokale Konsistenz

Verteilte Konsistenz

Replikation?

Konfliktmanagement

3 Bestehende offlinefähige Systeme / Konzepte

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

3.1 Offline plugin für webpack

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

3.2 Redux Offline

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

Redux Offline erweitert Redux um einen persistenten STORE mit Offline-First Technologie und ist kompatibel mit allen *View Frameworks wie React[6], Vue[7], oder Angular[8] [redb]. Es umfasst unter Anderem netzwerkfähige API-Aufrufe, das Persistieren des Zustands 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 noch ein ServiceWorker implementiert sein [redd].
Die Idee hinter Redux Offline ist, dass der Redux STORE die Datenbank ersetzt [Evä17]. Bei jeder Änderung wird der Redux STORE auf dem Datenträger gespeichert, und bei jedem Start automatisch neugeladen. Für das Speichern der Daten in einer lokalen Datenbank wird intern Redux Persist verwendet.

Eine mit Redux Offline erstellte Anwendung funktioniert ohne weitere Codeimplementierung offline im Lesemodus, da das Lesen und Schreiben aus der lokalen Datenbank bereits eigebunden ist. Damit die Anwendung auch im Schreibmodus offline funktioniert, müssen einige Anpassungen vorgenommen werden. Sämtliche Daten der Anwendung können nur über Aktionen manipuliert werden. Alle netzwerkgebundenen Aktionen werden in einem STOREinternem Queue gespeichert und müssen mit einem Metaattribut dekoriert werden um offline arbeiten zu können. Durch die Metaattribute weiß die Anwendung was vor der eigentlichen Ausführung der Aktion und was danach zu tun ist. Es gibt drei Metadaten die Redux Offline interpretieren kann:
`meta.offline.effect` - Die initiale Aktion wird ausgeführt. Hier kann eine URL angegeben werden die Redux Offline anfragen soll.
`meta.offline.commit` - Hier wird die Aktion definiert die ausgeführt wird sobald die Netzwerkanfrage erfolgreich ist.
`meta.offline.rollback` - Hier kann die Aktion angegeben werden, die bei permanent fehlgeschlagener Internetverbindung oder wenn der Server einen Serverfehler zurückgibt gefeuert wird. Dann fügt Redux Offline dem APPSTATE automatisch ein `offline` Objekt hinzu. Dort wird unter anderem ein Array namens `outbox` verwaltet wird. Dieses Array repräsentiert den Queue. Hier werden die Aktionen inklusive Metadaten gespeichert, um bei bestehender Internetverbindung abgearbeitet zu werden [redc]. Die von Jani Eväkallio erstellte Grafik 3.1 veranschaulicht die oben erklärte Architektur.
Links ist eine Aktion zu sehen die Zeug machen möchte. Sie hat ein Mateattribut das weitere Aktionen definiert. Eine Aktion für den Erfolg und eine für den Fehlschlag von 'DO_STUFF'. In der Mitte ist der STORE zu sehen. Der STORE kennt den Netzwerkstatus und hat den Queue namens `outbox` in dem Aktionen mitsamt ihrer Metafelder gespei-

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

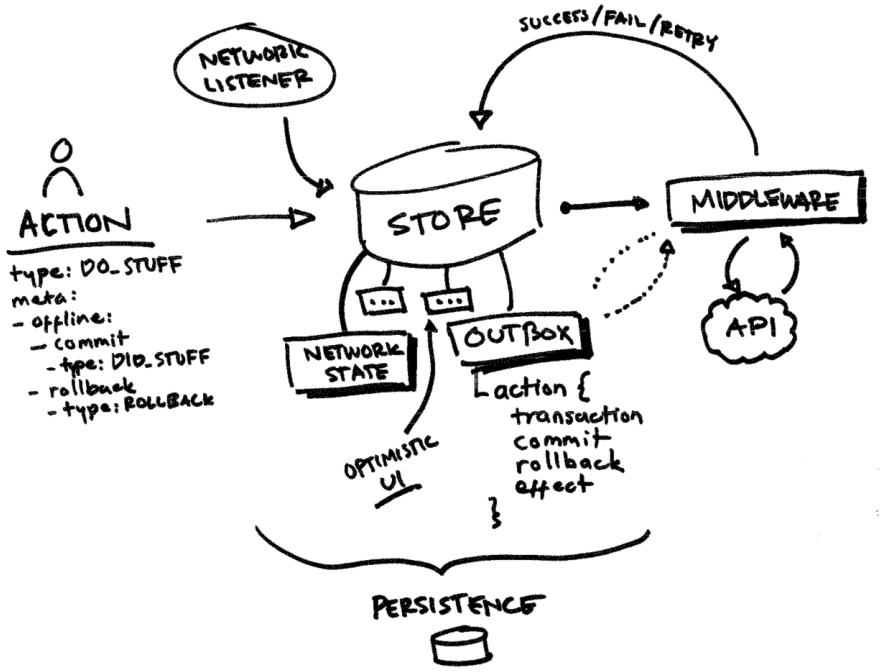


Abbildung 3.1: *Redux Offline Architektur Quelle: [Evä17]*

chert werden. Rechts befindet sich das API, das über die Middleware mit dem STORE redet.
Wird die Aktion 'DO_STUFF' gefeuert gelangt sie in den STORE, damit dieser den APP-STATE aktualisieren kann, und wird ersteinmal im Queue gespeichert. Ist die Anwendung online, wird sie sofort abgearbeitet. Wenn nicht wird sie dort gespeichert bis die Anwendung wieder eine Verbindung zum Internet hat.

3.2.1 REDUX PERSIST

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

3.3 React Native Offline

React Native ist ein Framework mit dem native, mobile Apps mit JavaScript und React gebaut werden können [Soub].
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 nun auch einen Offline Queue in dem Aktionen gespeichert werden können. Allerdings nur Aktionen die fehlgeschlagen sind weil die Anwendung nicht mit dem Internet verbunden ist. Auch hier wird der Aktion ein Metaattribut gegeben. Dieses hat die Felder `retry` und `dismiss`. Das erste erwartet ein Boolean, es kann angegeben werden ob die Aktion noch einmal bei bestehender Internetverbindung ausgeführt werden soll oder nicht. Das Feld `dismiss` erwartet ein Array. Hier können Aktionen angegeben werden die das Wiederholen der Aktion abbrechen [Acu].

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

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

Listing 3.1: *Beispiel React Native Offline Aktion*

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

3.4 HOODIE

Hoodie ist eine JavaScript Bibliothek für offlinefähige Webapplikationen, die ein komplettes Backend zur Verfügung stellt. Wird Hoodie für die Entwicklung einer Webanwendung verwendet, muss also lediglich das Frontend implementiert werden. Den Rest erledigt die Bibliothek. Über eine integrierte Programmierschnittstelle kommuniziert die Anwendung mit dem von Hoodie zur Verfügung gestelltem Backend. Über das API können unter Anderen BenutzerInnen authentifiziert, Daten gespeichert und synchronisiert werden [hooa].
Anhand der Abbildung 3.2 wird erklärt wie Hoodie funktioniert.

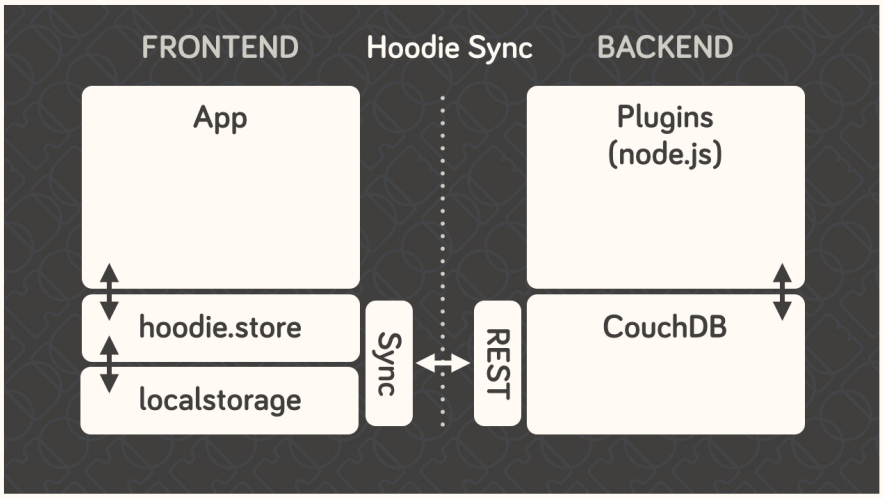


Abbildung 3.2: *Hoodie Architektur Quelle: [hoob]*

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

terbrochen wird.

3.5 Realm

Realm ist eine Backendtechnologie für mobile Anwendungen und umfasst die Realm Datenbank und den Realm Object Server. Beide Technologien sind quelloffen, jedoch nicht kostenfrei [rea].
Die Realm Datenbank ist eine objektorientierte, plattformübergreifende lokale Datenbank die eine Echtzeitsynchronisation mit dem Realm Object Server. Der Object Server fungiert als Middleware-Komponente in der mobilen App und handhabt unter anderem die Ereignisbehandlung und Datensynchronisation. Im Zusammenspiel ermöglichen die beiden Technologien die Erstellung von offlinefähigen, kollaborativen, mobilen Anwendungen [rea17b].

Zur Offline First Funktionalität stellt Realm eine umfassende Lösung bereit. Die lokale Realm Datenbank unterstützt die Echtzeitsynchronisation von Daten sodass alle Änderungen sofort automatisch gesendet werden. Das Synchronisationsprotokoll komprimiert statt dem gesamten Objekt nur die marginalen Änderungen und synchronisiert sie auf dem Endgerät und dem Server. Zusätzlich zu den Daten werden die spezifischen Operationen erfasst. Wird beispielsweise ein Kontakt bearbeitet, wird neben den geänderten Daten die Information *update* mitgesendet. Dank dieser zusätzlichen Information kann der Aktionswunsch genau erfasst werden sodass das System eventuelle Konflikte automatisch auflösen kann. Das hat zur Folge, dass die Synchronisation keinen manuellen Eingriff bedarf, der die Leistung des Systems beeinträchtigen könnte. Zusätzlich zu dem OT Algorithmus benutzt Realm vorgegebene Regeln zur automatischen Konfliktlösung. Es besteht die Möglichkeit eigene Regeln zu definieren. Eine Regel könnte zum Beispiel sein, dass alle Änderungen die von der Nutzerin, nennen wir sie Amilia Pond, gemacht werden den Vorrang haben. Das ist keine gute Regel, denn dabei würden definitiv Daten verloren gehen wenn eine andere Person eine andere Änderung an derselben Stelle wie Amilia Pond macht, aber sie soll ja nur als Beispiel dienen.
Darüber hinaus passiert die interne Konfliktlösung auf Transaktionsebene. Das heißt der Vorgang ist nur erfolgreich wenn er auch vollständig und fehlerfrei ist. Andererseits wird er zurückgesetzt. Das gewährleistet die Konsistenz der Daten und verhindert deren Verlust wenn Änderungen aufgrund einer unterbrochenen Netzwerkverbindung nicht stattfinden können [rea17a].

3.6 DerbyJS – wenn noch Zeit bleibt

4 Szenarien

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

4.1 Szenarien bei der Datenübertragung

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

Szenario C0 – Client push:

Der Client erstellt einen Adressbucheintrag, hat den Status ONLINE und der Server ist erreichbar. Sowohl Anfrage als auch Antwort ist erfolgreich. Der Kontakt wird erfolgreich erstellt.

Szenario C1 – Client push:

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

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

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

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

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

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

In den obigen Szenarien wird nicht beschrieben warum die Internetverbindung abbricht. Dies kann verschiedene Gründe haben. Um nur einige Beispiele zu nennen: Eine langsame Internetverbindung, oder eine Fahrt durch einen Tunnel kann ein Timeout während einer Aktion hervorrufen. Ein auf einer Baustelle gekapptes Kabel oder ein Stromausfall kann zu zeitweise vollständigen Internetverlust (haha) führen.
Die Abbildung 4.1 veranschaulicht die beschriebenen Situationen, die bei der Übertragung von Daten über das Netzwerk eintreten können. Die in Felder in Lila beschreiben die Szenarien bei denen der Client etwas an den Server schickt. Die blauen Felder auf der rechten Seite zeigen solche Szenarien, die eintreten können wenn der Server etwas an den Client sendet. Die Sechsecke stehen für die Ausgangssituation, die Kreise repräsenzieren die Szenarien und die Rechtecke die daraus resultierenden Fälle.

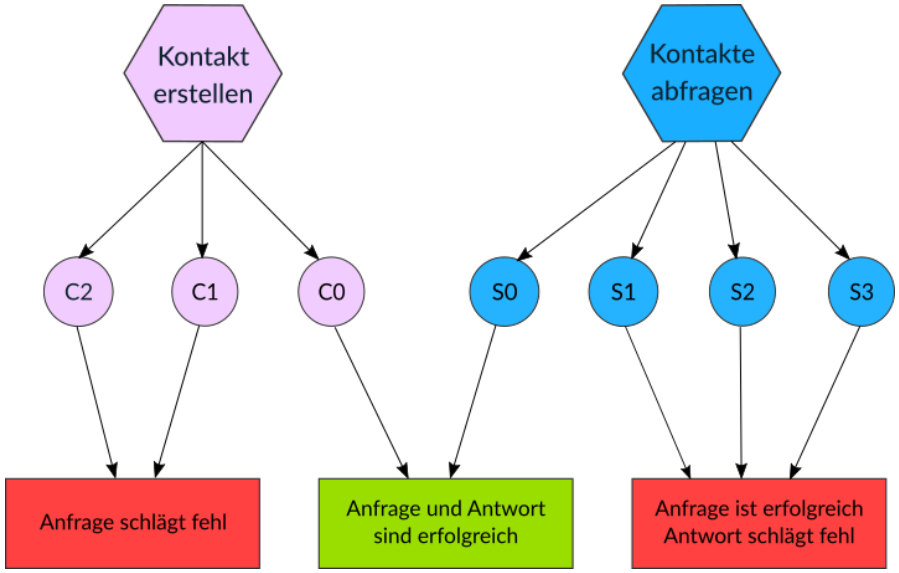


Abbildung 4.1: *Szenarien bei der Datenübertragung über das Netzwerk*

Ergebnis

Da die Szenarien *C0* und *S0*, die Szenarien *C1* und *C2* sowie die Szenazien *S1*, *S2* und *S3* zusammengefasst werden können, ergeben sich aus den sieben Szenarien die drei nun aufgezählten Fälle.

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

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

4.2 Szenarien zur Konfliktentstehung

Im Anwendungsbeispiel einer Kontaktliste können mehrere Personen die Liste verwalten. Oder eine Person benutzt die Anwendung auf mehreren Geräten, was zum selben Ergebnis führt: Die Komplexität wird durch mehr Parteien – beliebig viele Clients – erhöht. Jede Person kann alle Einträge jederzeit laden und einzelne erstellen, bearbeiten oder löschen. Bei den Ausführungen der grundlegenden Create Read Update Delete (CRUD) Operarionen kann es bei der Synchronisation der beteiligten Parteien zu Konflikten kommen wenn einer der oben genannten Fälle (*b*,*c*) eintritt und ein Objekt von mehreren Parteien bearbeitet wird. Für die effiziente Nutzung der Anwendung sollen bei jedem Start

der Anwendung nur die neuen und die gegebenenfalls aktualisierten Einträge geladen werden. Alle Adressbucheinträge müssen identifiziert und versioniert werden. Folgende Situationen können eintreten: Muss bei Implementation nicht unbedingt zum Konflikt kommen. Szenarien beschreiben den worst case! Es geht schließlich um Konflikte. Quellenangaben zu den Szenarien (Wikipedia, Interview...)

Szenario ID0 – UUID:

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

Szenario ID1 – sprechender Schlüssel:

Zur Identifizierung eines Adressbucheintrags wird ein sprechender Schlüssel[9] verwendet. Es wird sowohl auf dem Client als auch auf dem Server ein Kontakt mit dem Namen ‘Amilia Pond‘ und dem sprechenden Schlüssel ‘amiliapond‘ erstellt. Währenddessen tritt Fall *b,c* ein. Es ist nicht zu ermitteln, ob derselbe Kontakt doppelt angelegt wurde, wenn beide Kontakteinträge sich unterscheiden, welcher der beiden korrekt ist oder ob es sich bei den Einträgen um zwei Personen mit demselben Namen handelt.

Szenario V0 – 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 aktualisiert und geben ihm beide die Versonsnummer ‘2.0.0‘. Währenddessen tritt Fall *b,c* ein und beide Parteien können nicht miteinander Kommunizieren. Bei der Synchronisation entsteht ein Konflikt weil es zwei (unterschiedliche) Einträge mit derselben Verion gibt.

Szenario V1 – Zeitstempel:

Zur Versionierung eines Adressbucheintrags wird ein Zeitstempel verwendet. Der Kontakt ‘Amilia‘ hat die Version ‘2018-04-03 10:00:00Z‘. Amilia ist umgezogen und ihre Adresse ändert sich. Der Eintrag wird bearbeitet und hat nun die Version ‘2018-04-13 11:44:22Z‘. Während der Editierung tritt Fall *b,c* ein. Es stellt sich heraus, dass die Hausnummer einen Zahlendreher hat und es wird sofort berichtigt. ‘Amilia‘ hat nun

[9] 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.

die Version ‘2018-04-13 11:45:33Z‘. ... dass der Server eine spätere Uhrzeit als der Client hat. So hat nach der Synchronisation der später korrigierte Eintrag einen früheren Zeitstempel. Es wird die falsche, alte Adresse gespeichert, die korrekte hat einen älteren Zeitstempel und wird verworfen.

Funktioniert meistens trotzdem gut. Spezifische Gegenbeispiele nennen.

Szenario V2 – Logische Uhr:

Wurde für die Versionierung in verteilten Systemen entwickelt weil timestamp so fehleranfällig ist. Zur Versionierung eines Adressbucheintrags wird eine Logische Uhr[10] verwendet. Der Kontakt ‘Amilia‘ hat die Version Beispiel Logische Uhr?. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst (Version:). Währenddessen tritt Fall *b*, *c* ein. Amilia sieht ihre falsche Telefonnummer und berichtigt diese ebenfalls. weil die Versionen identisch sind? Bei der Synchronisation kommt es zum Konflikt. wirklich? auch wenn das Ergebnis dasselbe ist?

Szenario V3 – Inhaltsbasierte Version:

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

hash kann gleich sein wenn sich nur 1 Zeichen ändert? Dem Kontakt ‘Amilia‘ ist die Version ‘5560348cec1b08c3d53e1508b4a46868‘ zugeordnet. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst, während dieser OFFLINE ist. Im selben Status berichtigt der Client die Telefonnummer. Bei der Synchronisation kommt es zum Konflikt, da es nun zwei Einträge mit unterschiedlichem Inhalt, aber identischer Version gibt und nicht festzustellen ist welche Version die neuere ist.

Szenario V4 – Liste von inhaltsbasierten Versionen:

Zur Versionierung eines Adressbucheintrags wird eine geordnete Liste von inhaltsbasierten Versionen verwendet. Dem Kontakt ‘Amilia‘ ist eine Liste von Verionen mit einem Eintrag ‘5560348cec1b08c3d53e1508b4a46868‘ zugeordnet. Amilias Telefonnummer ändert sich und wird auf dem Client angepasst, während dieser OFFLINE ist. Im selben Status berichtigt der Client die Telefonnummer. Jede Aktion fügt der Versionsliste einen neuen Hashwert hinzu. Auch wenn der Content des Adresbucheintrags in den zwei letzten Versionen identisch ist, kann festgestellt werden welcher

[10] 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.

der neueste Eintrag ist. Kommt es zum Konflikt, werden die beiden *riskanten* Versionen verschachtelt in der Liste gespeichert. In diesem Fall sieht die Liste nun so aus: ‘[[88da3f8d82ab58551d2a48d74d9a4986, 88da3f8d82ab58551d2a48d74d9a4986], 5560348cec1b08c3d53e1508b4a46868]‘ – eine Liste der beiden konfliktbehafteten Versionen am Anfang der Liste.

Ergebnis

Die Szenarien *ID0* und *ID1* beschreiben die Identifizierung einzelner Kontakte. Eine eindeutige Identifizierung des Kontakts ist im Szenario *ID0*, mittels der Verwendung einer UUID, gewährleistet.
Es wird deutlich, dass es in jedem Fall zu einem Konflikt kommen kann. Es gilt zu unterscheiden in welchen Fällen mit Konflikten umgegangen werden kann und in welchen Daten verloren gehen.
Grafik?

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

5 Anforderungsdefinition

Dieses Kapitel beschreibt die Anforderungen an eine Offline First Anwendung unter Berücksichtigung von Funktionalität, Konfliktmanagement, Tests und der Bedienoberfläche. Nach der konkreten Beschreibung dessen, was das *Projekt* umfassen soll, folgen die aus den oben genannten Szenarien hergeleiteten Anforderungen, die eine offlinefähige Anwendung erfüllen soll.

5.1 Umfang

Das Ziel dieser Arbeit ist die Untersuchung des Konfliktmanagements offlinefähiger Technologien. Dazu soll die beispielhafte Anwendung eines kollaborativen Adressbuchs betrachtet werden. Dass diese Anwendung auch ohne Internetzugang funktioniert, ist obligatorisch. Der Schwerpunkt liegt auf den Konflikten die entstehen können, wenn die Internetverbindung abbricht. Dabei sind Aspekte aus unterschiedlichen Rollen zu betrachten. So werden die Rollen der AnwenderInnen, der EntwicklerInnen und die der TesterInnen berücksichtigt.

Es soll eine Applikation entwickelt werden, welche an dem Beispiel eines kollaborativen Adressbuchs die Offlinekompatibilität mit dem Schwerpunkt auf das Konfliktmanagement der verwendeten Technologien illustriert. Die Anwendung umfasst eine Liste von Kontakten, welche einzeln bearbeitet werden können. Die Offlinefunktionalität soll durch die verwendeten Technologien gegeben sein.
Um eine Aussage darüber zu treffen, welches der untersuchten Technologien besser für die Entwicklung einer offlinefähigen Anwendung geeignet ist und warum, ist eine Testumgebung mit zu entwickeln. Hier soll die Möglichkeit geschaffen werden, Konflikte durch gleichzeitiges Bearbeiten eines Kontakts im Offlinestatus zu forcieren und die Testfälle auswerten zu können. Wie oft muss ich was testen?
Es wird ermittelt welche Strategie zur Konfliktlösung von der Technologie verwendet wird, ob die Funktionalität der Anwendung auch bei auftretenden Konflikten gewährleistet ist und ob dabei Daten verloren gehen. Dabei wird auch der Aufwand betrachtet, der aufgebracht werden muss die Technologie zu verwenden und wie leicht der geschriebene Quellcode zu verstehen ist.

Dinge, die von vornherein ausgeschlossen werden und Bezug zu Szenarien und Anwendungsfällen?

5.2 User-Stories

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

5.2.1 User Perspektive

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

ID	Anforderung aus Userperspektive
User-Story 1	Ich als NutzerIn möchte die Anwendung immer und überall, auch ohne Internetzugang zu nutzen.
User-Story 2	Ich als Nutzerin möchte, dass die Kontaktliste schnell und effizient geladen wird, um Zeit zu sparen.
User-Story 3	Ich als NutzerIn möchte jeden Kontakteintrag finden, um zu wissen ob ich ihn schon gespeichert habe.
User-Story 4	Ich als NutzerIn möchte Einträge immer und überall, auch ohne Internetzugang, erstellen, editieren oder löschen können, um meine Liste zu verwalten.
User-Story 5	Ich als NutzerIn möchte keine in der Anwendung gespeicherten Daten verlieren.

Tabelle 5.1: *Anforderungen aus Userperspektive*

5.2.2 Developer Perspektive

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

ID	Anforderung aus Entwicklungsperspektive
User-Story 6	Ich als EntwicklerIn möchte die Daten lokal und auf dem Server speichern, um deren Erreichbarkeit unabhängig vom Internetstatus zu gewährleisten.
User-Story 7	Ich als EntwicklerIn möchte ich nur die Adressbucheinträge oder deren Aktualisierungen laden, die sich nicht schon auf dem Endgerät befinden, um Datentraffic und Ladezeiten zu sparen.
User-Story 8	Ich als EntwicklerIn möchte ich jeden Eintrag identifizieren, um jedem Adressbucheintrag Operationen zuzuweisen und einzelne Kontakte zu finden.
User-Story 9	Ich als EntwicklerIn möchte, dass jeden Eintrag versioniert ist, um zu wissen ob wann ein Eintrag bearbeitet wurde.
User-Story 10	Ich als EntwicklerIn möchte, dass alle von NutzerInnen vorgenommenen Änderungen beim System ankommen und keine Daten verloren gehen.
User-Story 11	Ich als EntwicklerIn möchte auftretende Konflikte effizient speichern, um mit ihnen umgehen können. Mit ihnen umgehen heißt: selbstständig oder von User lösen, zum konfliktfreien Zustand gelangen
User-Story 12	Ich als EntwicklerIn möchte eine Technologie verwenden die leicht zu verstehen und implemenieren ist, um den Arbeitsaufwand gering zu halten.
User-Story 13	Ich als EntwicklerIn möchte sauberen und verständlichen Code schreiben, um die Les– und Wartbarkeit zu erhöhen.

Tabelle 5.2: *Anforderungen aus Entwicklungsperspektive*

5.2.3 TesterInnen Perspektive

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

ID	Anforderung aus TesterInnenperspektive
User-Story 14	Ich als TesterIn möchte sicherstellen, dass der Netzwerkstatus der Anwendung änderbar ist, um zwischen offline und online zu wechseln.
User-Story 15	Ich als TesterIn möchte wissen ob die Anwendung mit dem Internet verbunden ist oder nicht.
User-Story 16	Ich als TesterIn möchte die Anwendung auf mindestens zwei Geräten verwenden, um Kontakte gleichzeitig zu bearbeiten zu können.
User-Story 17	Ich als TesterIn möchte Konflikte forcieren, um das Verhalten der Anwendung zu evaluieren.
User-Story 18	Ich als TesterIn möchte einen Eintrag editieren *können* wenn 1. beide Client und Server online sind, 2. entweder Client oder Server offline ist oder 3. beide Parteien offine sind. daraus 3 Stories machen? oder nur testen online–offline?
User-Story 19	Ich als TesterIn möchte die Testfälle detailliert dokumentieren *können*, um sie auswerten zu können.

Tabelle 5.3: *Anforderungen aus TesterInnenperspektive*

5.3 Funktionalität

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

Bein ersten Start der Anwendung müssen, sofern vorhanden, alle Kontakte geladen werden. Sobald sie einmal geladen sind, sollen sie auch offline verfügbar sein. Damit ein Datensatz, wie zum Beispiel ein Adressbucheintrag, offline erreichbar ist, sollte er wenigstens so lange auf dem Client gespeichert werden, bis er vollständig beim Server angekom-

men sind. Im aktuellen Anwendungsfall bedeutet das, es gibt zwei Kopien des Adressbucheintrags. Eine auf dem Anwendungsgerät – beispielsweise in einer lokalen Datenbank ref: grundlagen/lokale Speicherung?, eine auf dem Server.
Danach sollen nur die Einträge geladen werden, die nicht auf dem Gerät existieren. Die Daten würden sonst doppelt geladen werden, der Server hätte mehr zu arbeiten was wiederum die Antwortzeit verlängern würde. Dazu muss ermittelt werden können, welche Daten neu angelegt oder aktualisiert wurden. Der Server muss also in der Lage sein die Einträge zu sortieren und nur bestimmte Einträge zu versenden und die Anwendung muss wissen, welche Daten sie bereits hat. Wird ein Eintrag angelegt, bearbeitet oder gelöscht, müssen alle Parteien wissen um welchen Kontakt es sich handelt. Dazu muss jeder Kontakt mittels einer eindeutigen ID identifiziert werden. Wenn es zwei Kontakteinträge mit derselben ID gibt, muss feststellbar sein, welcher Eintrag der aktuellere ist.
Gibt es mehr als zwei Einträge müssen diese sortiert werden, sodass ersichtlich wird welcher der aktuellste oder älteste ist, welcher Eintrag vor oder nach welchem kommt. Dazu muss jeder Kontakt versioniert werden.

5.3.1 Konfliktmanagement

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

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

5.4 Bedienoberfläche

Da der Schwerpunkt dieser Arbeit auf dem Umgang mit Konflikten der zu testenden offlineunterstützenden Technologien liegt, soll die Bedienoberfläche der Anwendung mög-

lichst 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 aufgelöst werden. Im Dialog muss erkennbar sein wo, bei welchem Kontakteintrag, der Konflikt auftrat. Außerdem müssen sich die entsprechenden Bereiche beider Versionen unterscheiden lassen und auswählbar sein. Abbildung 5.1 zeigt wie so ein Dialogfenster aussehen könnte.

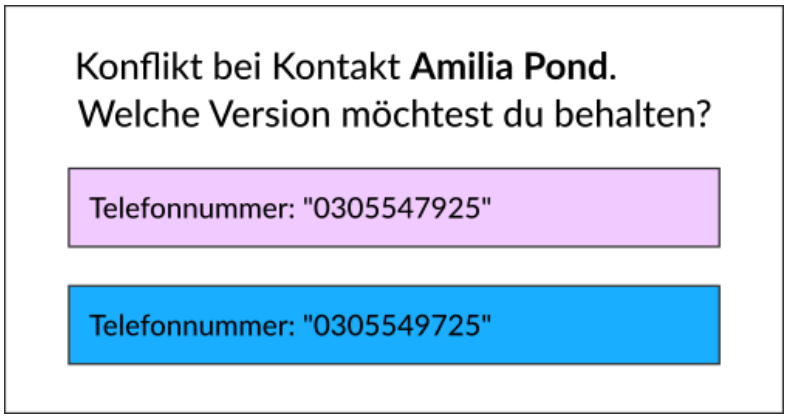


Abbildung 5.1: *Dialogfenster im Konfliktfall*

Wurde beispielsweise Amilias Telefonnummer von zwei Personen gleichzeitig bearbeitet, bildet der Dialog zwei Bereiche mit der Nummer in den unterschiedlichen Versionen ab. Durch Klick auf die korrekte Nummer kann entschieden werden welche Version die richtige ist und behalten wird.

5.5 Tests

Auch testen ob Offlinefunktionalität gewährleistet ist? – Scope: Zeitlichen Rahmen sprengen usw.
Um das Konfliktmanagement der zu testenden Technologien untersuchen zu können, müssen zunächst Konflikte erstellt werden. Dazu muss in erster Linie die Anwendunng auf mindestens zwei Geräten laufen und der Netzwerkstatus muss änderbar sein. Es ist außerdem hilfreich, wenn die Anwendung zeigt, in welchem Status sie sich befindet. Sie sollte wissen, ob sie on– oder offline ist.
Kontakte editieren (auf 2 Geräten gleichzeitig) on-on, off-off, on-off? Dasselbe für Anlegen und löschen?
Zur Auswertung der Daten sollten alle Ausgangspositionen, Vorgänge und Ergebnisse

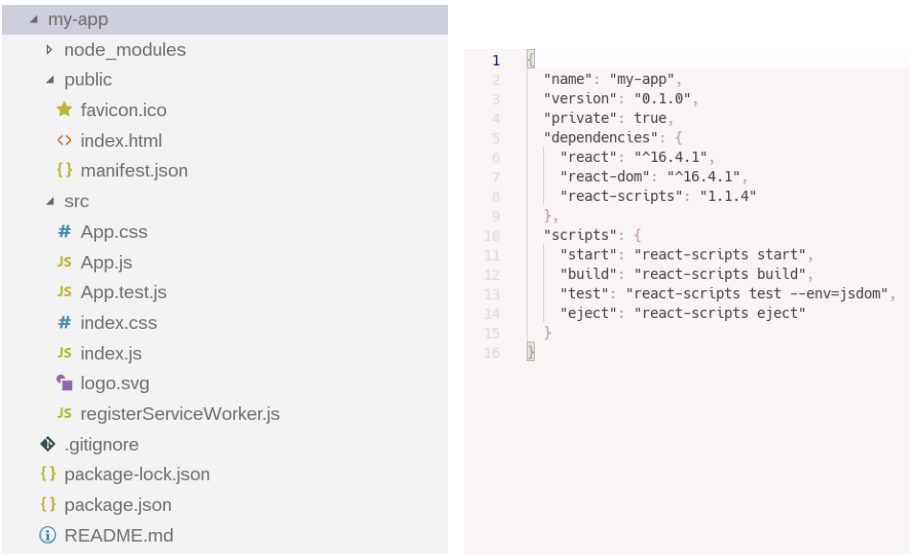
dokumentiert werden. Für einen Testfall kann hierfür zuerst der Kontakt aufgeschrieben werden. Dann die Operation, z.B. ‘bearbeiten des Kontakts Amilia‘ zusammen mit dem Verbindungsstatus (von Client und Server). Ebenso muss der Kontakt aufgeschrieben werden, wie er nach der Synchronisation aussieht. Lösung anbieten: .log–Datei? Es stellt sich die Frage wie oft ein Testfall durchlaufen werden muss um eine sinnvolle Aussage treffen zu können.

6 Konzeption

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

6.1 Anwendungsaufbau

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



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

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

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

6.1.1 Aufbau der React Komponenten ?

React ist eine open-source Bibliothek, die dazu dient, die View-Komponente des Model-View-Controller-Ansatzes abzudecken, also die Seite der Anwendung die für die Anzeige und Interaktion zuständig ist. Ein Vorteil von React sind die wiederverwendbaren Komponenten. Eine Komponente ermöglicht die Aufteilung der User Interface (UI) in kleine Teile und ist eine abstrakte Basisklasse. Einmal implementiert, lässt sich eine Komponente immer wieder verwenden [Soua].
Eine Komponente kann einen internen `state` besitzen, oder die Daten nur aus den `props` nehmen. `Props` sind Eigenschaften die übergeben werden und nur von der Elternkomponente änderbar. Zur Veranschaulichung wird anhand des Listings 6.1 die reduzierte Version der Formularkomponente beider zu entwickelnden Prototypen beschrieben. In dieser Version wird nur der Name des Kontakts gezeigt und verändert.
Die Formularkomponente hat Kontaktobjekt im internen `state` gespeichert. Auf dieses Objekt haben andere Komponenten keinen Zugriff und es ist nur via `setState()` änderbar. Initial wird das Kontaktobjekt über die `props` geladen (Zeile vier). So kann das Vorausfüllen der Eingabefelder realisiert werden.
In Zeile sieben ist die `handleChange()` Funktion, die am Eingabefeld (Zeile 22) auf die Änderungen reagiert (Zeile zehn) und der interne `state` aktualisiert (Zeile 12).
Eine React Komponente hat immer eine `render()`-Funktion (Zeile 15) 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.

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

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

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

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

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

Listing 6.2: *Aufruf der React* `ContactForm` *Komponente*

6.1.2 Erweiterungen für AMILIA-RDX

Redux Offline kann nur zusammen mit Redux verwendet werden. Deswegen ist für diesen Prototypen die Implementierung von Redux vorausgesetzt.

Redux

Redux ist eine Bibliothek zur Zustandsverwaltung in JavaScriptanwendungen. Mit Redux hat jede Applikation genau einen `store`. Dieser hat den Applikationsstatus, der via `dispatch()` aktualisiert, und via `getState()` gelesen werden kann. Als einzige Informationsquelle für den `store` dienen Aktionen. Sie senden Daten von der Anwendung mittels `store.dispatch()` an den `store` und beschreiben dabei nicht wie etwas passiert, sondern was passiert.
Der dritte wichtige Bestandteil von Redux sind die `Reducer`. Sie spezifizieren wie der Status sich als Reaktion auf die Aktionen ändert [reda].

Redux Offline

Redux Offline ist eine erweiternde Bibliothek für Redux dessen Funktionsweise in Abschnitt 3.2 detailliert beschrieben wird.
Nach der Installation muss der Redux `store` zusammen mit dem `offline store enhancer` erzeugt werden. Listing 6.3 visualisiert diesen Vorgang. Ein Redux `store` wird mit dem `storeCreator` in Zeile fünf erzeugt. Ein `store enhancer` ist eine Funktion die den `storeCreator` neu zusammenfügt und einen neuen, erweiterten `storeCreator` zurückgibt. Redux Offline kommt mit einer vorgegebenen Konfiguration (siehe Zeile 3). Diese wird dem `offline store enhancer` in Zeile acht übergeben.

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

Listing 6.3: *Erstellen eines Stores mit Redux Offline*

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

```
1 export function readContacts () {
2   return {
```

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

Listing 6.4: *Aktion `fetchContacts` mit Metaattribut*

Das erste `meta.offline` Feld beschreibt die Netzwerkaktion die ausgeführt werden soll, also den Aufruf an die angegebene URL in Zeile sechs. Bei `commit` in Zeile sieben wird festgelegt welche Aktion bei erfolgreichem Netzwerkaufruf augeführt werden soll. Für den Fall dass von dem angefragtem API der Statuscode 500 zurückkommt oder der Server nicht läuft, wird die im `rollback` definierte Aktion gefeuert.
Die Aktionen beschreiben nur was passiert. Wie der Status sich ändert, wird im `Reducer` beschrieben. Das Listing 6.5 illustriert wie das im entsprechendem Prototypen umgesetzt werden könnte.

```
1 function contacts (state=[], action) {
2   switch (action.type) {
3     case FETCH_CONTACTS:
4       console.log('started to fetch contacts ')
5       return state
6
7     case FETCH_CONTACTS_COMMIT:
8       console.log('successfully fetched contacts ')
9       if (state === action.payload) return state
10      return action.payload
11
12    case FETCH_CONTACTS_ROLLBACK:
13      console.log('failed to fetch contacts')
14      return state
15  }
```

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

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

6.2 Architektur

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

Beide Prototypen setzen sich aus den nachfolgend beschriebenen Komponenten zusammen, welche in Abbildung 6.2 veranschaulicht werden.

Die Komponente `Contacts` fungiert als Container und ist das Herzstück der Anwendung. Er definiert die graphische Oberfläche und stellt alle notwendigen Funktionen bereit. Sie hat einen internen `state` in dem sowohl die Kontaktliste, als auch die Daten für das Formular gespeichert sind. Im Diagramm ist der `state` an der blauen Schrift zu erkennen. Das Objekt `editView` zeigt, welche Ansicht – Liste oder Formular – gerade aktuell ist und speichert den im Formular zu ladenden Kontakt. Wie das Kontaktobjekt aufgebaut ist zeigt der blaue Kasten im Diagramm. Wird eine Aktion zum Ändern der Ansicht aufgerufen, beispielsweise durch das Betätigen eines Knopfes, wird über die Funktion `toggleEdit()` der interne `state` aktualisiert und ein erneutes Rendern der Komponente eingeleitet. Dann wird entsprechend die Liste oder das Formular gerendert.
Die `Header`-Komponente zeigt den Netzwerkstatus der Anwendung an und hat einen Knopf, der zum `ContactForm` führt.
Die `ContactList` wird initial gerendert. Es werden alle Kontakte als Liste dargestellt. Hier kann das Bearbeiten (`handleOnEditClick()`) oder das Löschen (`handleOnDeleteClick()`) des Kontakts eingeleitet werden.

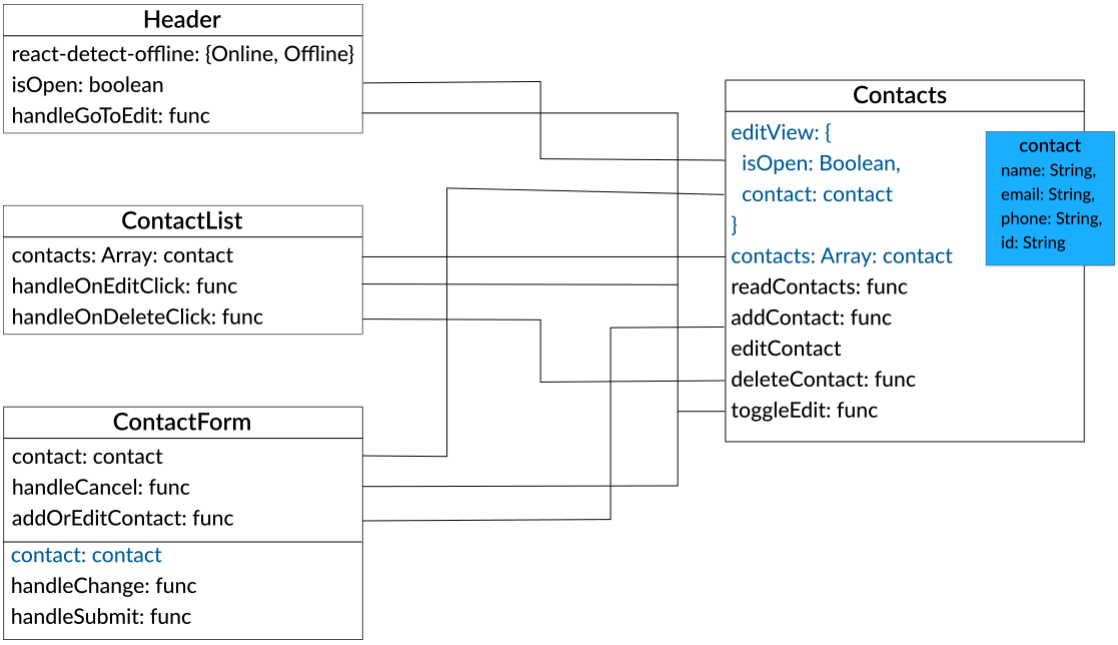


Abbildung 6.2: *Komponentenarchitektur*

Die Komponente `ContactForm` zeigt, sofern vorhanden, alle im Kontakt gespeicherten Daten an. Diese können hier bearbeitet werden. Gibt es keine Kontaktdaten die geladen werden kann, kann hier ein neuer Kontakt angelegt werden. Zusätzlich zu den Eingabefeldern für jedes Kontaktattribut hat sie zwei Knöpfe mit denen die Aktion bestätigt (`handleSubmit()`) oder abgebrochen (`handleCancel()`) werden kann. Sie ist neben `Contacs` die einzige Komponente mit einem internen `state`. Dieser wird für die Ereignishandler benötigt, die auf die Veränderung der einzelnen Eingabefelder zu "lauschen". Der zu bearbeitende Kontakt wird hier zwischengespeichert und dann als Ganzes an `Contacts` gegeben.
Konflikt–Dialog

Backend: Implementierung bei qouch nicht notwendig. Nur Installation von Couch.
Für Redux Offline: Server der alle CRUD Operationen unterstützt und JSON Datei zur Persisierung (DB Ersatz?) um Ergebnisse nicht zu verfälschen UND Redux- store, actions, Reducer

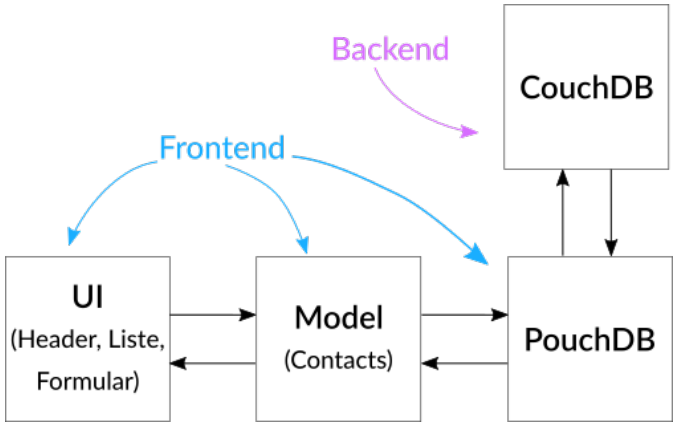


Abbildung 6.3: *Client-Server-Modell*

6.2.1 Das Speichern der Daten

Das Seichern von Kontakten wird in den Prototypen unterschiedlich implementiert.

Daten mit PochDB und CouchDB speichern

Für den Protoyp *amilia-qouch* muss zunächst CouchDB installiert werden. Sobald dieser Schritt erledigt ist läuft CouchDB auf `localhost:5984` und ist einsatzbereit. Das asynchrone API von PouchDB stellt alle notwendigen Funktionen bereit die sowohl Callbacks, Promises als auch asynchrone Funktionen unterstützen. Das Listing 6.6 führt alle benötigeten Funktionen auf und zeigt die notwendigen Schritte zur Synchronisation der lokalen PouchDB und CouchDB.
Die lokale Datenbank wird in Zeile eins erstellt. Wenn es die Datenbank mit dem Namen ‘contacts‘ bereits gibt, wird sie gestartet.
Um eine CouchDB Instanz zu erzeugen ist der Aufruf in Zeile zwei mit der URL zur CouchDB–Datenbank notwendig. Auch hier erstellt PouchDB die Datenbank, sofern sie noch nicht existiert. PouchDB funktioniert nun als Client zu einer online CouchDB Instanz. Zur kontinuierlichen Synchronisation beider Instanzen, der lokalen PouchDB und der CouchDB, ist lediglich der Aufruf in Zeile vier erforderlich. Im optionalen Parameter können zum Beispiel FIlter oder Einstellungen zum wiederholten Synchronisationsversuch im Falle eines Fehlschlags gesetzt werden [pouc].

```
1 const localDB = new PouchDB('contacts')
2 const remoteDB = new PouchDB('http://localhost:5984/contacts')
3
4 localDB.sync(remoteDB, [options])
5
```

```
 6 // create a contact
 7 localDB.put({
 8   ...contact,
 9   _id: new Date().toISOString()
10 })
11 // update a contact
12 localDB.get(contact._id).then(function (doc) {
13   doc = {...contact}
14   return localDB.put(doc)
15 })
16 // get all contacts
17 localDB.allDocs({
18   include_docs: true,
19   conflicts: true
20 })
21 // remove a contact
22 localDB.remove(contact)
```

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

Die Zeilen sieben bis zehn zeigen wie ein Kontakt erzeugt werden kann. Bevor das geschieht wird die ID gesetzt. PouchDB bietet zur Erstellung von Objekten auch `localDB.post()` an. Bei dessen Verwendung wird `_id` von PouchDB automatisch generiert. Diese Variante wird jedoch nicht empfohlen, weil dann die IDs zufällig sind, die Objekte nicht danach sortiert werden können [poub].
Das Aktualisieren eines Kontakts sieht ähnlich aus. Zuerst wird der entsprechende Kontakt wie in Zeile zwölf aus der Datenbank angefragt um dann in der Datenbank aktualisiert zu werden. Mit jedem Update bekommt ein Kontakt von PouchDB eine neue Revision.
Der Aufruf `localDB.allDocs()` in Zeile 17 fragt alle in der lokalen Datenbank gespeicherten Kontakte an. Ohne den Parameter `include_docs: true` werden nur die `_id` und die `_rev` Eigenschaften eines jeden gespeicherten Kontakts zurückgegeben. Ist die Option `conflicts` auf `true` gesetzt, werden unter dem Attribut `_conflicts` Konfliktinformationen zu jedem Kontakt gespeichert.
Man kann einen Kontakt in PouchDB wie in Zeile 22 mittels `localDB.remove(contact)` löschen. Der Kontakt ist dann nicht wirklich gelöscht sondern wird durch ein `_deleted` Attribut als solches markiert.

Daten mit Redux Offline speichern

Die Idee hinter Redux Offline ist, dass der Redux Store die lokale Datenbank ersetzt. Sobald der Appstatus sich ändert, also irgendwo im Code `setState()` ausgeführt wird,

wird er automatisch lokal gespeichert. Dazu wird intern `redux-persist` benutzt, dessen Funktionsweise in Abschnitt 3.2.1 erläutert wird. Der Redux Store wird bei jeder Änderung persistiert und beim Start der Anwendung neu geladen. Wie die Daten mit Redux Offline gespeichert synchronisiert werden ist am folgenden Listing erklärt.

```
1  // action creator
2  export function addContact () {
3    return {
4      type: ADD_CONTACT,
5      contact,
6      meta: {
7        offline: {
8          effect: {
9            url: `${API}/contacts`, method: 'POST',
10           body: JSON.stringify({contact})
11         },
12         commit: { type: 'ADD_CONTACT_COMMIT', meta: { contact } },
13       }
14     }
15   }
16 }
17
18 // reducer
19 function contacts (state=[], action) {
20   switch (action.type) {
21     case ADD_CONTACT:
22       console.log('started to add contact')
23       return [...state, action.contact]
24
25     case ADD_CONTACT_COMMIT:
26       console.log('successfully added contact')
27       return [...state, action.payload]
28   }
29 }
```

Listing 6.7: *Speicherung der Daten mit Redux Offline*

Mit dem Aufruf der Aktion ADD_CONTACT wird der Vorgang gestartet einen Kontakt hinzuzufügen. Die anzufragende URL ist im `meta.offline.effect` Feld in Zeile neun festgelegt. Die Anfrage geht an den Server, welcher die Daten in der JSON Datei persisitiert. Der Reducer hat Zugriff auf das Aktionsobjekt. Dort ist der gerade hinzugefügte Kontakt gespeichert. Mit diesem wird in Zeile 23 der Appstate aktualisiert und so lokal gespeichert.
Ist die Netzwerkanfrage erfolgreich, wird die Aktion `commit` in Zeile zwölf gefeuert. Die

wird im Reducer in Zeile 25 behandelt. Der `state` wird mit der Antwort vom Server aktualisiert und die Synchronisation ist vollzogen.

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

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

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

In Zeile zwei werden die Kontakte aus einer JSON Datei geladen. Diese sind als Objekt in einem Array gespeichert. Bekommt der Server eine post–Anfrage, wird ein Kontaktobjekt mitgesendet. Dieser wird in Zeile fünf in einer Variable zwischengespeichert. Wird der Kontakt korrekt gesendet wird er in Zeile zehn dem Array hinzugefügt. Andererseits sendet der Server den HTTP–Statuscode 400 an den Client. Außerdem wird mithilfe des in Node integriertem Dateisystem[11] Moduls die JSON Datei neu geschrieben. Nun ist der neue Kontakt persistiert. Kommt es beim Schreiben der Datei zu keinem Fehler, sendet der Server den frisch gespeicherten Kontakt zurück an den Client.

6.2.2 Verbindungsstatus feststellen und ändern

Für die Überprüfung der Verbindung zum Server wird das Modus React Detect Offline verwendet. Es beobachtet den Online– und Offlinestatus und bietet zwei Komponen-

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

ten entsprechend des Status den Inhalt rendern. Der folgende Codeausschnitt zeigt eine Verwendung dieser beiden Komponenten. Ist die Anwendung online, wird ‘you are online‘ gerendert. Im anderen Fall ‘you are offline‘.

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

Listing 6.9: *Beispiel einer React Detect Offline Implementierung*

Das Modul fragt 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 Poll–Interval können konfiguriert werden [Bol].

Der Verbindungsstatus kann im Browser geändert werden. Die Prototypen, die im Rahmen dieser Arbeit entwickelt werden, sollen in den Browsern Firefox und Chromium laufen.
In Firefox lässt sich der Netzwerkstatus über das Einstellungsmenü ändern. Dort kann man entweder unter dem Punkt ‘Sonstiges‘ oder dem Punkt ‘Web-Entwickler‘ ‘Offline arbeiten‘ auswählen und ist vom Internet getrennt. Dieser Status lässt sich über den selben Weg rückgängig machen.
In Chrome öffnet man dazu die Entwicklertools, geht auf ‘Netzwerk‘ und klickt auf die Checkbox ‘Offline‘ am oberen Rand. Dieselbe Checkbox ist auch im ‘Application‘–Tab unter ‘Service Workers‘ zu finden.

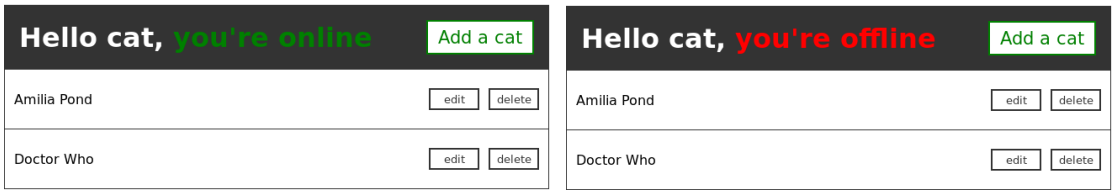
6.3 Die graphische Oberfläche

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

Diese Listenansicht in Abbildung 6.4 besteht aus dem Header / Kopf und den Listeneinträgen. Sie zeigt die Kontakteinträge in beiden Netzwerkstatus: online (Abbildung 6.4a) und offline (6.4b).

Im Header ist abzulesen ob die Anwendung gerade eine Netzwerkverbindung hat oder

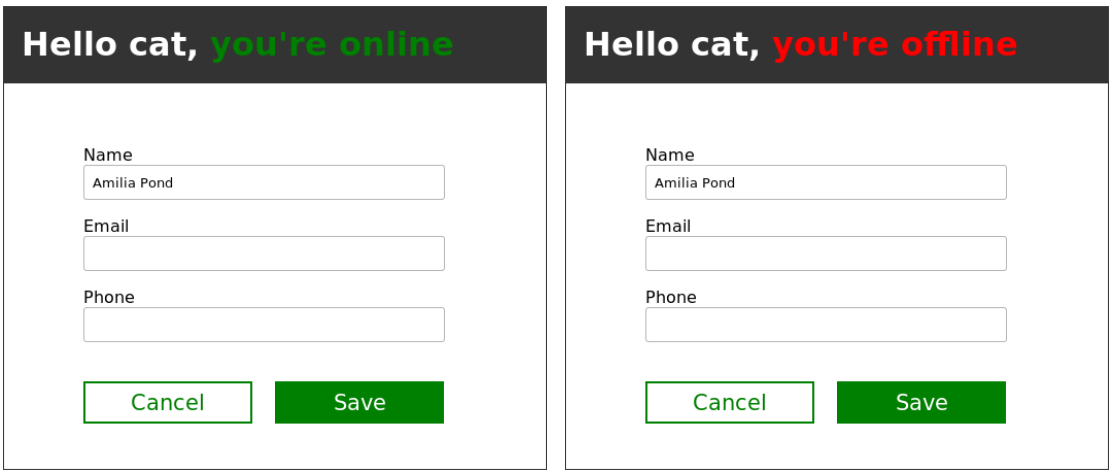
nicht. Für eine bessere Prägnanz wurden hierzu unterstützend die Farben Rot für keine Verbindung und Grün für eine bestehende Netzwerkverbindung gewählt. Rechts im Header gibt es einen Knopf mit dem man in die Ansicht gelangt in der ein Kontakt hinzugefügt werden kann.
In der Liste sieht man die Namen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



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

Abbildung 6.4: *Die Kontaktliste in beiden Netzwerkstatus*

Klickt man auf den Knopf zum Bearbeiten oder auf den zum Hinzufügen eines Kontakts gelangt man in die Bearbeitungsansicht (vgl. Abbildung 6.5). Der Header ist bis auf den Knopf zum Hinzufügen eines Kontakts identisch zu dem der Liste. Auch hier ist abzulesen ob die Anwendung on– oder offline ist. Da man sich bereits in der Ansicht zum Anlegen oder Editieren eines Kontaks befindet, ist der Knopf im Header überflüssig.
Ein Kontakt hat einen Namen, eine E-Mailadresse und eine Telefonnummer. In dieser Ansicht gibt es für jedes Attribut ein Eingabefeld. Die Felder sind beim Bearbeiten des Kontakts vorausgefüllt. Mittels Betätigung des ‘Speichern‘ Knopfs werden die Änderungen übernommen, klickt man auf ‘Cancel‘ werden sie verworfen. In beiden Fällen gelangt man wieder zur Listenansicht.



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

Abbildung 6.5: *Die Editieransicht in beiden Netzwerkstatus*

Konfliktdialog

6.4 Testfälle

Folgende Testfälle zur Offlinefunktionalität 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 bei jedem Start die Kontakte aus dem lokalen Speicher oder aus der *Datenbank* laden.

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

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

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

7 Implementierung der Prototypen

In diesem Kaptitel wird nach dem in Kapitel **??** präsentiertem Lösungsweg die detaillierte Beschreibung der technischen Realisierung der Prototypen vorgestellt.
Nach der Beschreibung der Konfigurationsdateien wird auf die Umsetzung der Szenarien eingegangen. Im Zuge dessen werden die implementierten Algorithmen vorgestellt, wobei sich der erste mit dem Auffinden der jeweilig nächsten relevanten Ampel befasst und der zweite die jeweils empfohlene Geschwindigkeit berechnet.

7.1 package.json?

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

7.2 Hauptmodul Contacts

wird beim Start ausgeführt
Stellt zenrale Funktionen bereit... Erklärung

7.3 Umsetzung der Szenarien? Anforderungen?

Umsetzung der Funktionalität?
Wie wurde CRUD implementiert?
Wie kann ein Konflikt erzeugt werden?
(manuell – siehe 6.2.2)

7.4 Installationsanleitung

build und so Beide entwickelten Prototypen sind als öffentliche Repositories auf GitHub[12] zu finden. Um sie zu installieren müssen folgende Schritte ausgeführt werden.

7.4.1 amilia-qouch

1. Zuerst muss das Repository kopiert werden:

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

2. Dann muss man in das Verzeichnis navigieren und alle Abhängigkeiten installieren.

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

3. Mittels

```
npm start
```

wird die Anwendung gestartet.

7.4.2 amilia-rdx

1. Auch hier muss das Repository zuerst kopiert werden:

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

2. Schritt zwei ist identisch mit dem in der `amilia-qouch` Anleitung
3. Mittels

```
npm run server
npm start
```

wird zuerst der Server, dann die Anwendung gestartet.

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

8 Evaluation

8.1 Test

8.1.1 Implementierungsaufwand

Kann ja nicht wirklich getestet werden, würde ich nur in die Auswertung schreiben.

8.1.2 Offlinefunktionalität

Offlinefunktionalität wurde stetig getestet, siehe 6.4

8.1.3 Konfliktmanagement

Tests mit 1 Gerät / Browser

(um Redux Offline eine Chance zu geben...)

Tests mit 2 Geräten / Browser

Kontakt anlegen:

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

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

8.2 Auswertung

8.2.1 Implementierungsaufwand

Setup der App:

Irrelevant und identisch wegen Create React App.

Einbinding der Technologien:

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

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

Lesbarkeit des Codes:

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

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

8.2.2 Offlinefunktionalität

8.2.3 Konfliktmanagement

Vergleich

	PouchDB	Redux Offline
Storage	IndexedDB WebSQL	(redux-persist) Brwoser: IndexedDB oder WebSQL / LocalStorage, Fallbacks via localForage. ReactNative: AsyncStorage
Requirements		Redux
Lines of Code	?	?

Tabelle 8.1: *Anforderungen aus Entwicklungsperspektive*

9 Zusammenfassung und Ausblick

Abkürzungen

API	Application Programming Interface
App	Applikation
CAP	Consistency Availability Partition tolerance
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DB	Datenbank
DBMS	Datenbank Management System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
OT	Operational Transformation
PWA	Progressive Web App
REST	REpresentational State Transfer
UI	User Interface
UUID	Universally Unique Identifier
VM	Virtuelle Maschine
WLAN	Wireless Local Area Network

GLOSSAR

Assets

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

Bandbreite

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

Hashfunktion

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

Kollaborativ

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

Latenz

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

Middleware

Schicht zwischen Anwendung und Betriebssystem.

Progressive Web App

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

Queue

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

Abbildungsverzeichnis

QUELLCODEVERZEICHNIS

LITERATURVERZEICHNIS

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

[ALS10] ANDERSON, J. C. ; LENHARDT, Jan ; SLATER, Noah: *CouchDB: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, 2010. – ISBN 978-0-596-15589-6. – oreilly.com 2.3.3

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

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

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

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

[cana] *Can I use IndexedDB?* https://caniuse.com/#search=indexedDB,. – Zugriff: 06.06.2018 2.3, 2.4, 9

[canb] *Can I use Service Workers?* https://caniuse.com/#feat=serviceworkers,. – Zugriff: 06.05.2018 2.1, 9

[canc] *Can I use Web Storage?* https://caniuse.com/#feat=feat=namevalue-storage,. – Zugriff: 06.06.2018 2.2, 9

[cou] *CouchDB – relax*. https://couchdb.apache.org/,. – Zugriff: 12.04.2018 2.1.3

[EG89] ELLIS, C. A. ; GIBBS, S. J.: Concurrency Control in Groupware Systems. In: *SIGMOD Rec.* 18 (1989), jun, Nr. 2, 399–407. http://dx.doi.org/10.1145/66926.66963. – DOI 10.1145/66926.66963. – ISSN 0163-5808 2.3.2

[Evä17] EVÄKALLADO, Jani: Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native. In: *HACKERnoon* (2017), 3. – Zugriff: 12.04.2018 3.2, 3.1

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

[Fra09] FRASER, Neil: Differential Synchronization. Version: Jan 2009. https://neil.fraser.name/writing/sync/eng047-fraser.pdf. 2009. – Research Report. – 8 S. 2.3.2

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

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

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

[hoob] *How Hoodie Works*. http://docs.hood.ie/en/latest/about/how-hoodie-works.html,. – Zugriff: 12.04.2018 3.2, 3.4, 6

[Iwa16] IWANIUK, Daniel: *How to persist Redux State to the Local Storage*. https://www.hawatel.com/blog/how-to-persist-redux-state-to-the-local-storage/, 10 2016. – Zugriff: 29.06.2018 3.2.1

[LL10] LI, Du ; LI, Rui: An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems. In: *Comput. Supported Coop. Work* 19 (2010), feb, Nr. 1, 1–43. http://dx.doi.org/10.1007/s10606-009-9103-1. – DOI 10.1007/s10606-009-9103-1. – ISSN 0925-9724 2.3.2

[loc] *localForage – Offline storage, improved*. https://localforage.github.io/localForage/,. – Zugriff: 26.06.2018 3.2.1

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

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

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

[poua] *pouchdb – Tha Database that Syncs!* https://pouchdb.com/learn,. – Zugriff: 12.04.2018 2.1.3

[poub] *API Reference – Create/update a document*. https://pouchdb.com/guides/updating-deleting.html#why-must-we-dance-this-dance,. – Zugriff: 24.06.2018 6.2.1

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

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

[rea17a] *The Offline-First Approach to Mobile App Development - Beyond Caching to a Full Data Sync Platform*. https://www2.realm.io/whitepaper/offline-first-approach-registration, october 2017. – Zugriff: 12.04.2018 3.5

[rea17b] *BUILD BETTER APPS, FASTER WITH REALM - An Overview of the Realm Platform*. https://www2.realm.io/whitepaper/realm-overview-registration, october 2017. – Zugriff: 12.04.2018 3.5

[reda] *Redux*. https://redux.js.org/basics,. – Zugriff: 07.06.2018 6.1.2

[redb] *Redux Offline Release BREAKING: Migrate to Store Enhancer API*. https://github.com/redux-offline/redux-offline/releases/tag/v2.0.0,. – Zugriff: 12.04.2018 3.2

[redc] *Redux Offline Documentation*. https://github.com/redux-offline/redux-offline/tree/develop/docs,. – Zugriff: 27.06.2018 3.2

[redd] *Redux Offline*. https://github.com/redux-offline/redux-offline, . – Zugriff: 12.04.2018 3.2

[rede] *Redux Persist*. https://github.com/rt2zz/redux-persist#storage-engines,. – Zugriff: 12.04.2018 3.2.1

[Rus15] RUSSELL, Alex: Progressive Web Apps: Escaping Tabs Without Losing Our Soul. In: *Medium* (2015), 08. – Zugriff: 15.04.2018 2.1.4

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

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

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

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

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

Anhang

Eidesstattliche Erklärung

CD-Inhalt

Auf der beigefügten CD befinden sich

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