



HOCHSCHULE BREMEN

FAKULTÄT 4

INTERNATIONALER STUDIENGANG MEDIENINFORMATIK B.Sc.

BACHELORTHESIS

# Synchronisierung von Daten in einer Offline-First Webanwendung mithilfe von CRDTs

**Autor:** Lennart Ploog

**Matrikelnummer:** 354556

**Erstgutachter:** Prof. Dr. Volker Paelke

**Zweitgutachter:** Prof. Dr. Martin Hering-Bertram

**Abgabedatum:** 28.08.2020

# **Zusammenfassung**

Die Arbeit “Synchronisierung von Daten in einer Offline-First Webanwendung mithilfe von CRDTs” behandelt die Konzeption und Umsetzung einer Offline-First Webanwendung mit einer Datenstruktur auf Basis von konfliktfreien replizierten Datentypen (CRDTs). Zuerst werden die Grundlagen vom Offline-First Prinzip und CRDTs erkundet. Dabei wird untersucht, wie und warum der Einsatz von CRDTs in Offline-First Applikationen hilfreich sein kann, denn die Themen stehen nicht zwangsläufig in Verbindung. Anschließend wird eine Webanwendung, welche auch offline vollständig nutzbar ist, als Prototyp konzipiert und umgesetzt. Die Datenstruktur wird mithilfe von CRDTs so implementiert, dass ein konfliktfreies Synchronisieren von Daten mit einem Server möglich ist, auch wenn Änderungen offline erfolgt sind. Nach der Fertigstellung des Prototypen folgt eine Evaluation der entwickelten Anwendung und ihrer Funktionalitäten. Abgeschlossen wird die Arbeit mit dem Fazit, in welchem zusammengefasst wird, welche Erkenntnisse über den Einsatz von CRDTs in Offline-First Anwendungen gewonnen werden konnten.

## **Abstract**

The bachelor's thesis “Data Synchronization In an Offline-First Web Application Using CRDTs” discusses the design and implementation of an offline-first web application with a data structure based on conflict-free replicated data types(CRDTs). First, the basics of the Offline-First principle and CRDTs are explored. Because the topics are not necessarily related, it is investigated how and why the use of CRDTs in offline-first applications can be helpful. Subsequently, a web application, which is fully usable offline, is designed and implemented as a prototype. With the help of CRDTs, the data structure is implemented to allow conflict-free synchronization of it's data with a server, even if changes have been made offline. After completion of the prototype an evaluation of the developed application and its functionalities is conducted. The thesis is concluded with a summary of the knowledge gained on the use of CRDTs in Offline-First applications.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Problemfeld . . . . .	1
1.2	Ziel der Arbeit . . . . .	2
1.3	Vorgehen . . . . .	2
<b>2</b>	<b>Grundlagen</b>	<b>4</b>
2.1	Verwandte Arbeiten . . . . .	4
2.2	Definition: Offline-First . . . . .	5
2.3	Service Worker . . . . .	5
2.4	Caching . . . . .	6
2.5	Offline-First Applikationen als verteilte Systeme . . . . .	8
2.6	Replikation . . . . .	10
2.6.1	Pessimistische Replikation: Strong Consistency . . . . .	11
2.6.2	Optimistische Replikation: Eventual Consistency . . . . .	11
2.6.3	Strong Eventual Consistency . . . . .	12
2.7	Conflict-free Replicated Data Types (CRDTs) . . . . .	12
2.7.1	Last-Write-Wins Register . . . . .	14
2.7.2	Last-Write-Wins Map . . . . .	15
2.7.3	Grow-Only Set . . . . .	15
<b>3</b>	<b>Konzeption</b>	<b>16</b>
3.1	Idee des Prototypen . . . . .	16
3.2	Anforderungsanalyse . . . . .	16
3.2.1	Funktionale Anforderungen . . . . .	16
3.2.2	Nicht-funktionale Anforderungen . . . . .	18
3.3	Architektur . . . . .	18
3.4	Entwurf der Benutzeroberfläche . . . . .	21

3.5	Datenstruktur . . . . .	22
3.5.1	Grundlagen der Datenstruktur . . . . .	23
3.5.2	Erweiterung zum CRDT . . . . .	25
3.5.3	Parallele Änderungen . . . . .	27
3.5.4	Anwenden von Operationen . . . . .	28
3.6	Server . . . . .	31
3.6.1	Aufgabe des Servers . . . . .	31
3.6.2	API . . . . .	32
3.6.3	Datenbank . . . . .	32
3.6.4	Synchronisation . . . . .	32
<b>4</b>	<b>Prototypische Realisierung</b>	<b>35</b>
4.1	Wahl der verwendeten Technologien . . . . .	35
4.1.1	JavaScript, HTML und CSS . . . . .	35
4.1.2	VueJS . . . . .	35
4.1.3	IndexedDB . . . . .	36
4.1.4	idb . . . . .	36
4.1.5	Node.js, Express.js, MongoDB . . . . .	37
4.1.6	Sonstige Bibliotheken und Ressourcen . . . . .	38
4.1.7	Ordnerstruktur . . . . .	38
4.2	Nutzeroberfläche . . . . .	39
4.3	Service Worker . . . . .	41
4.4	Lokales Speichern über die IndexedDB-API . . . . .	44
4.5	Generieren der Operationen . . . . .	45
4.6	Verarbeiten von Operationen . . . . .	47
4.6.1	Lokale Operationen laden . . . . .	47
4.6.2	Operationen filtern . . . . .	49
4.6.3	Operationen anwenden . . . . .	52
4.7	Synchronisieren . . . . .	55
4.8	Löschen überflüssiger Operationen . . . . .	56
<b>5</b>	<b>Evaluation</b>	<b>58</b>
5.1	Anlegen, Bearbeiten und Löschen von Rezepten und Zutaten . . . . .	58
5.2	Zugriff mit mehreren Endgeräten zugleich . . . . .	58
5.3	Offline Erreichbarkeit . . . . .	59
5.4	Lokales Speichern . . . . .	59

5.5	Synchronisierung von Daten und SEC . . . . .	59
5.6	Konfliktfreies Synchronisieren . . . . .	60
5.7	Unabhängigkeit von Datenbank-Technologien . . . . .	60
<b>6</b>	<b>Fazit und Ausblick</b>	<b>62</b>
6.1	Fazit . . . . .	62
6.2	Ausblick . . . . .	64

# Abkürzungsverzeichnis

**CRDT** Conflict-free replicated data type

**PWA** Progressive Web-App

**SC** Strong Consistency

**SEC** Strong Eventual Consistency

**EC** Eventual Consistency

**P2P** Peer To Peer

**HLC** Hybrid Logical Clock

**JSON** JavaScript Object Notation

**HTTP** Hypertext Transfer Protocol

**HTTPS** Hypertext Transfer Protocol Secure

**LWW** Last-Write-Wins

**LWW-Register** Last-Write-Wins Register

**G-Set** Grow-Only Set

**CORS** Cross-Origin Resource Sharing

**BaaS** Backend as a Service

# 1 Einleitung

## 1.1 Problemfeld

Ob auf Reisen, im Supermarkt oder im Fahrstuhl – Situationen, in denen mobile Endgeräte keine stabile Internetverbindung haben, kommen im Alltag häufiger vor als gewünscht. In vielen Entwicklungsländern und auch in ländlichen Gegenden entwickelter Industriestaaten fehlt dafür gar die komplette Infrastruktur. Im Laufe der letzten Jahre eröffneten Innovationen im Bereich der Browser-Technologien, allen voran der Service-Worker, neue Möglichkeiten für die Webentwicklung, insbesondere für sogenannte Offline-First Anwendungen. Als Offline-First Applikationen werden Webanwendungen bezeichnet, die ihre Funktionalität so weit es geht behalten, wenn die Verbindung zum Internet getrennt ist.

Eine der Kernherausforderungen der Entwicklung von Offline-First Applikationen ist die Synchronisation von Daten. Werden offline Änderungen vorgenommen, sollen diese nicht verloren gehen. Hat sich der Zustand der Applikation, beispielsweise durch Modifikationen anderer Nutzer/innen, in der Zwischenzeit jedoch geändert, müssen die Änderungen zusammengebracht, also synchronisiert werden. Der Prozess der Synchronisation ist oft aufwendig, denn zum Einen muss ermittelt werden, wo sich die Replikationen unterscheiden und zum Anderen muss vermieden werden, dass die Änderungen sich in die Quere kommen.

Gängige Lösungen zu einer solchen Zusammenführung von Daten umfassen die Nutzung bestimmter Datenbank-Technologien. Dazu gehören Datenbanken mit implementierter Synchronisation wie CouchDB<sup>1</sup> oder auch Backend as a Service (BaaS) Produkte wie Google Firebase<sup>2</sup> oder IBM Cloudant<sup>3</sup>, welche ebenfalls eine solche Funktionalität anbieten. Um zu vermeiden, die Applikation mit suboptimalen Datenbank-Technologien

---

<sup>1</sup><https://docs.couchdb.org/en/stable/>

<sup>2</sup><https://firebase.google.com/docs>

<sup>3</sup><https://cloud.ibm.com/apidocs/cloudant>

umsetzen zu müssen, verzichten viele Applikationen bei Konflikten auf eine Synchronisation und eine/r der beiden Nutzer/innen verliert die vollbrachte Arbeit.

## 1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine Lösung zur Synchronisation von Daten in Offline-First Anwendungen mit Hilfe von Conflict-free replicated data types (CRDTs) umzusetzen. Der Einsatz von CRDTs ermöglicht, dass Daten in einem verteilten System in beliebiger Reihenfolge ausgetauscht werden können und dennoch zum gleichen Zustand aller Replikationen führen. Durch die Implementierung einer Datenstruktur, welche auf CRDTs aufbaut, kann der Prozess der Synchronisierung somit vermieden werden. Diese Lösung soll unabhängig von der gewählten Datenbank sein.

So ergeben sich folgende Forschungsfragen:

- Wie können CRDTs in Offline-First Applikationen verwendet werden?
- Welche CRDTs bieten sich zur Umsetzung von Offline-First Applikationen an, und wie werden diese in die Datenbanken (Client und Server) implementiert?
- Welche Vor- und Nachteile bietet die Nutzung von CRDTs im Vergleich zu anderen Optionen zur Synchronisierung von Daten in Offline-First Applikationen?

## 1.3 Vorgehen

Es gibt verschiedene Möglichkeiten, wie CRDTs in Webapplikationen eingesetzt werden können. Bevor die Implementation der Datenstruktur im hier entwickelten Prototypen beginnen kann, muss ermittelt werden, welche CRDTs sich am besten für die Daten des Prototypen eignen. Um dies herauszufinden, eignet sich die Recherche in den in Abschnitt 2.1 aufgeführten Publikationen. Darüber hinaus lohnt es sich an dieser Stelle auch, in Erfahrung zu bringen, welche CRDTs bis heute in fertigen Applikationen verwendet wurden. Auch Literatur zum Austausch von Daten in Applikationen zu kollaborativem Editieren in Peer To Peer (P2P) Netzwerken bietet sich zur Recherche an, denn in diesen Bereichen sind CRDTs schon weiter verbreitet als in anderen Anwendungsgebieten.



Damit der Prototyp als praxisnahes Beispiel dienen kann, sollte ebenso der Stand der Technik im Themenbereich der Offline-First Anwendungen ermittelt werden. Um einzuordnen, auf welcher Ebene der Architektur der Applikation sich die umzusetzende Funktionalität zur Zusammenführung der Daten am besten einbauen lässt, lohnt sich auch ein Blick auf bestehende Lösungen, welche die Synchronisation nicht direkt auf der Datenbankebene durchführen, sondern zwischen Applikation und Datenbank.

## 2 Grundlagen

Dieses Kapitel erläutert die Grundlagen, Ideen und Konzepte, auf welchen Offline-First Applikationen und CRDTs aufbauen.

### 2.1 Verwandte Arbeiten

CRDTs sind aus der Forschung an Datenstrukturen für kollaboratives Editieren entstanden. Shapiro, Preguiça, Baquero und Zawirski (2011) formulieren die theoretischen Grundlagen von CRDTs, um Strong Eventual Consistency (SEC) in großen verteilten Systemen zu garantieren. SEC erweitert den bis dahin verbreiteten Ansatz der Eventual Consistency (EC). Während EC nur garantiert, dass sämtliche Updates schlussendlich alle Replizierungen der Datenbank erreichen, garantiert SEC zusätzlich, dass Updates unabhängig von Reihenfolge und Zeitpunkt immer zum gleichen Zustand der Replizierungen führen.

Seitdem hat sich die Verwendung von CRDTs in verschiedenen Bereichen der Webentwicklung verbreitet. Kleppmann und Beresford (2017) entwerfen eine Bibliothek CRDT konformer JavaScript Object Notation (JSON) Datenstrukturen, genannt “automerge”, die beliebig verschachtelte Listen und Maps unterstützt. Mit “Hypermerge” entstand auch eine spezielle Version für Peer-to-Peer Netzwerke. Eine weitere Bibliothek, die vorgefertigte CRDTs bereitstellt, ist yjs (Nicolaescu, Jahns, Derntl & Klamma, 2015). Einen wichtigen Anhaltspunkt für die Nutzung in Offline-First Applikationen bietet der Vortrag “CRDTs for Mortals” (Long, 2019), in welchem über den Einsatz von CRDTs im Client-Server Modell gesprochen wird.

Mit Woot (Oster, Urso, Molli & Imine, 2006), Logoot (Weiss, Urso & Molli, 2009) und LSEQ (Nédelec, Molli, Mostefaoui & Desmontils, 2013) sind bereits CRDTs speziell für den Bereich des kollaborativen Editierens entwickelt worden.

Der Anzahl an Quellen und Ressourcen rund um CRDTs mangelt es weder an theoretischen noch an praktischen Beispielen. Während einige Arbeiten die Nutzung von CRDTs für Offlinefunktionalität empfehlen und die Umgebung von Offline-First Applikationen sehr den verteilten Netzwerken ähnelt, für die CRDTs konzipiert sind, sind dem Autoren keine wissenschaftlichen Arbeiten über den konkreten Einsatz von CRDTs in Offline-First Webanwendungen bekannt. Ziel dieser Arbeit ist es deshalb, die umfangreich erforschten Grundlagen zum Einsatz von CRDTs in einer Offline-First Webanwendung umzusetzen und zu ermitteln, wie dies bei der Bewältigung der Herausforderungen dieser Umgebung helfen kann.

## 2.2 Definition: Offline-First

Als Offline-First wird ein Vorgehen bezeichnet, bei welchem eine Applikation den Fall der unterbrochenen Internetverbindung nicht als Ausnahme, sondern als Standard ansieht (Google Developers, 2019). Teilweise wird der Begriff auch anders interpretiert, im Rahmen dieser Arbeit sei Offline-First jedoch unter folgenden Kriterien zu verstehen: Die Applikation geht davon aus, dass die Verbindung mit dem Internet nach dem ersten Laden der Seite stets unterbrochen werden kann. Dies gilt auch im Falle von Verbindungsproblemen, welche nicht vom Endgerät des Nutzers als solche erkannt werden, z.B. wenn das Endgerät mit dem Internet verbunden ist, aber die Route zur Website an anderer Stelle unterbrochen ist. Sämtliche Use-Cases werden so geplant, dass Nutzer/innen auch offline so viele Funktionalitäten wie möglich zur Verfügung stehen (Feyerke, 2013).

## 2.3 Service Worker

Ein Service Worker ist ein sogenannter Web Worker. Web Worker sind Skripte, die unabhängig von anderen Skripten, welche auf Interaktionen mit der Benutzeroberfläche reagieren, im Hintergrund der Webanwendung laufen (WHATWG, 2019).

In traditionellen Webanwendungen werden alle benötigten Dateien, Markups, Skripte und Assets über Netzwerk-Requests an den Server angefordert. Der Service Worker ist ein event-basiertes Skript, welches als Proxy zwischen Client und Server agiert. Damit

dies kein Sicherheitsrisiko darstellt, funktionieren Service Worker nur, wenn die Applikation Hypertext Transfer Protocol Secure (HTTPS) nutzt. Requests, welche üblicherweise direkt an den Server gehen würden, werden erst vom Service Worker verarbeitet. Entwickler/innen können gezielt entscheiden, welche Netzwerk-Requests auf welche Art und Weise verarbeitet werden sollen. Mithilfe dieser Funktionalität können Entwickler/innen sogenannte Caching-Strategien für den Service Worker implementieren, womit das Verbindungsverhalten der Applikation festgelegt werden kann (Jaiswal, 2019). Dies ist eine für Offline-First Applikationen essenzielle Funktionalität, denn so kann garantiert werden, dass die Applikation auch ohne Internetverbindung funktionsfähig ist.

## 2.4 Caching

In dieser Sektion werden einige grundlegende Caching-Strategien beschrieben, mit Beispielen, für welche Art von Netzwerk-Requests sie sich eignen könnten.

### “Erst Netzwerk, dann Cache”

Abbildung 1 beschreibt einen Request, den der Service Worker zuerst über das Netzwerk delegiert. Falls die Kommunikation mit dem Internet unterbrochen ist, beispielsweise wenn der Client offline ist, leitet der Service Worker den Request an den Cache weiter. Diese Methode eignet sich für Requests, bei denen aktuelle Daten bevorzugt sind, dem/-der Benutzer/in aber eine ältere Version zur Verfügung gestellt werden soll, wenn die Internetverbindung unterbrochen ist.

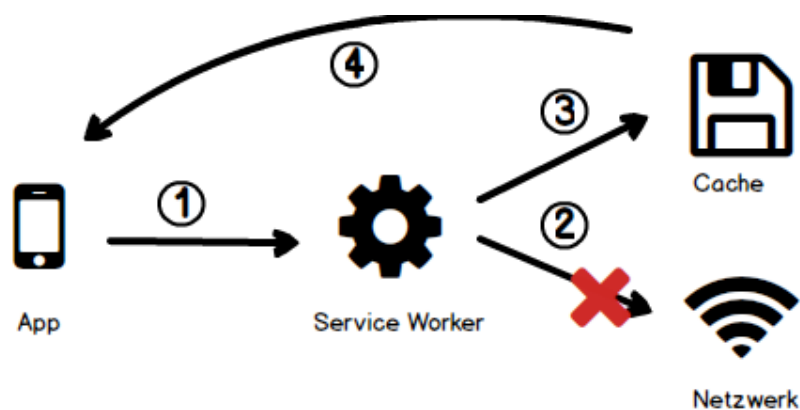


Abbildung 1: Caching-Strategie: “Erst Netzwerk, dann Cache” (eigene Darstellung)

**“Erst Cache, dann Netzwerk”**

Wie Abbildung 2 zeigt, wird der Request hier zuerst an den Cache weitergeleitet. Befindet sich die angefragte Datei nicht im Cache, wird die Anfrage an den Server weitergeleitet. Dies ist die bevorzugte Strategie für die meisten Netzwerk-Requests in Offline-First Anwendungen (Ater, 2017, Kapitel 05).

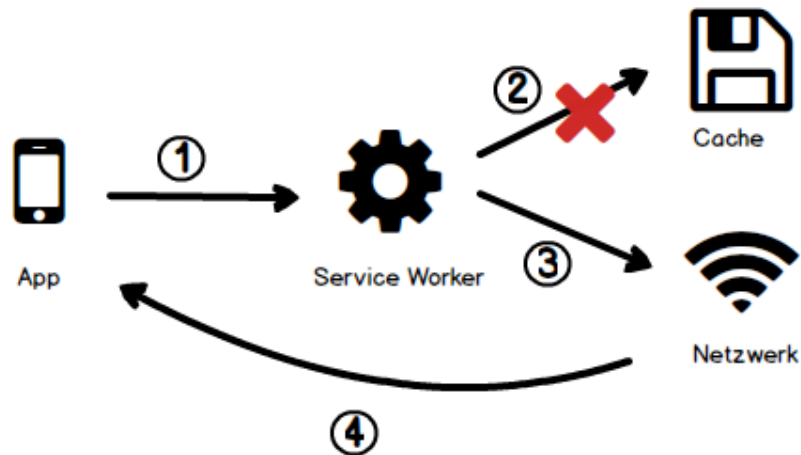


Abbildung 2: Caching-Strategie: “Erst Cache, dann Netzwerk” (eigene Darstellung)

**“Nur Netzwerk”**

Für Aufgaben, die nur online zu erfüllen sind, eignet sich die in Abbildung 3 bezeichnete Strategie. Hier leitet der Service Worker den Request nur an das Netzwerk und nie an den Cache weiter. Offline-First Applikationen sollten so konzipiert sein, dass diese Art von Requests im Falle einer unterbrochenen Internetverbindung nachgeholt werden können, sobald der Client wieder online ist.

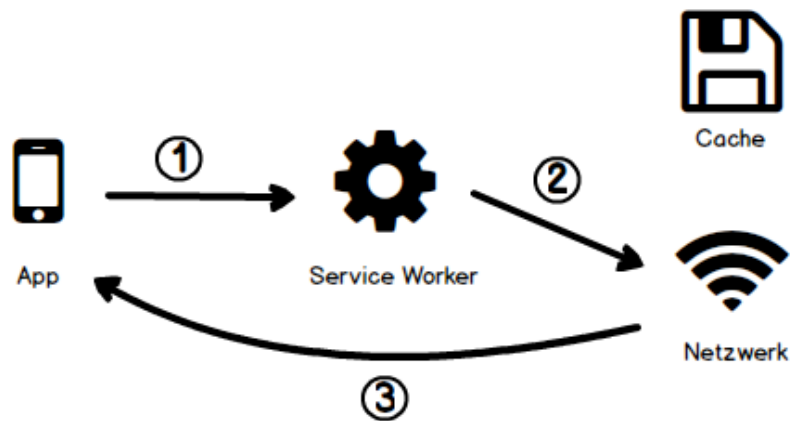


Abbildung 3: Caching-Strategie: “Nur Netzwerk” (eigene Darstellung)

**“Nur Cache”**

Abbildung 4 zeigt, wie die angefragte Ressource nur im Cache abgefragt wird. Diese Strategie ist nur dann sinnvoll, wenn die betroffenen Daten in einem vorherigen Schritt, beispielsweise beim Installieren des Service Workers, mit gecached wurden.

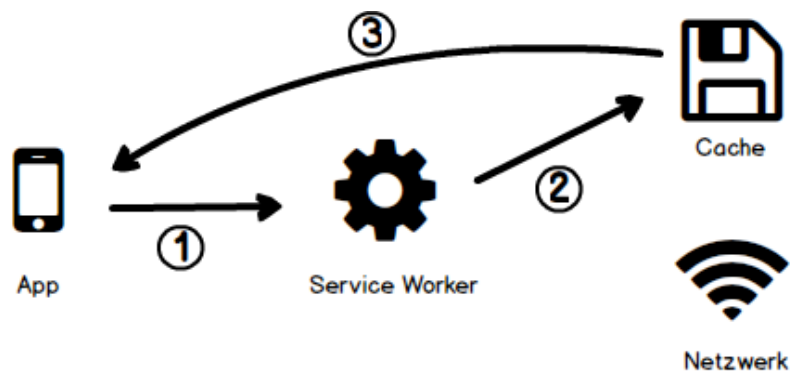


Abbildung 4: Caching-Strategie: “Nur Cache” (eigene Darstellung)

## 2.5 Offline-First Applikationen als verteilte Systeme

Van Steen und Tanenbaum (2016) beschreiben verteilte Systeme wie folgt: “Ein verteiltes System ist eine Sammlung von autonomen Rechenelementen, die dem Nutzer als ein einziges kohärentes System erscheint.”

Die folgende Auflistung zeigt die Charakteristika von verteilten Systemen, zusammengefasst nach Van Steen und Tanenbaum (2007, S. 2-3).

**Eigenständige Computer** In einem verteilten System sind mehrere eigenständige Computer zu einem System verbunden. Diese können sich sowohl in der Hardware als auch in der Funktionsweise unterscheiden.

**Singuläres Erscheinungsbild** Für Nutzer/innen sind die Unterschiede zwischen den einzelnen Computern im System unersichtlich. Sie nehmen die verteilten Computer als ein einzelnes System wahr.

**Konsistente und einheitliche Interaktion** Eine Konsequenz aus dem singulären Erscheinen ist, dass die Interaktion der Nutzer/innen mit dem System immer gleich sein sollte, unabhängig davon, mit welchem Computer des Systems sie tatsächlich interagieren.

**Kontinuierliche Verfügbarkeit** Das System soll den Nutzer/innen kontinuierlich zur Verfügung stehen, auch wenn einzelne Teile des Systems vorübergehend ausgefallen oder nicht erreichbar sind.

Die folgende Liste zeigt, dass funktionsfähige Offline-First Applikationen die gleichen Charakteristika aufweisen und fasst zusammen, welche Rolle diese Merkmale in der Applikation spielen.

**Eigenständige Computer** Der Server und verschiedene Endgeräte bilden ein System. Sobald ein Endgerät die Applikation zwischenspeichert, ist sie als eigenständiger Computer im System aktiv. Als Endgerät qualifiziert sich jedes Gerät, welches einen kompatiblen Browser betreibt, weshalb die Endgeräte auch untereinander über unterschiedlichste Hardware verfügen können.

**Singuläres Erscheinungsbild** Die Grundidee hinter Offline-First Applikationen ist das Gewährleisten von möglichst umfangreicher Offlinefunktionalität (Google Developers, 2020). Offline interagieren Nutzer/innen nur mit ihren Endgeräten, online werden die Daten auch an den Server geschickt. Diese Unterschiede sind für die Nutzer/innen jedoch nicht von Belang, da die Applikation auch offline funktioniert.

**Konsistente und einheitliche Interaktion** Unabhängig davon, welche Endgeräte Nutzer/innen verwenden, sollen ihnen früher oder später die Änderungen aller im System aktiven Geräte angezeigt werden. Nutzerinnen und Nutzer müssen sich zu

keinem Zeitpunkt Gedanken darüber machen, aus welchen Computern das System besteht.

**Kontinuierliche Verfügbarkeit** Ein weiterer Aspekt, welcher sich aus der verbindlichen Offlinefunktionalität ergibt, ist die kontinuierliche Verfügbarkeit. Nutzer/innen können ihre Arbeit auch fortführen, wenn der Server nicht erreichbar ist. Durch SEC landen diese Änderungen früher oder später im System, wodurch die getrennte Verbindung zum Server keine Auswirkungen auf dessen Funktionsumfang hat.

Daraus ergibt sich, dass es sich bei Offline-First Webanwendungen um verteilte Systeme handelt. Diese Erkenntnis kann dabei helfen, Probleme von Offline-First Applikationen zu lösen. Bei Offline-First handelt es sich um ein relativ junges Konzept. Obwohl Progressive Web Apps mittlerweile häufig im Netz anzutreffen sind, erfüllt deren Offlinefunktionalität selten Offline-First Kriterien. Für Probleme wie die in Abschnitt 1.1 beschriebene Synchronisation von Daten gibt es deshalb wenige beschriebene Lösungsansätze oder konkrete wissenschaftliche Arbeiten (vgl. Sektion 2.1). Mit verteilten Systemen hingegen beschäftigt sich die Informatik bereits seit den 70er Jahren (Andrews, 1999). Lösungen, welche für die Herausforderungen von verteilten Systemen entwickelt wurden, kommen also auch für Offline-First Webapplikationen in Frage. Eine dieser Lösungen ist die Nutzung von optimistischen Replikationsverfahren.

## 2.6 Replikation

Eine der wichtigsten Grundlagen verteilter Systeme ist die Replikation von Daten. Datenreplikation beschreibt das Verwalten mehrerer Datenspeicher, genannt Replikationen. Diese Replikationen halten die gleichen Daten, befinden sich jedoch auf unterschiedlichen Computern (Saito & Shapiro, 2005, S. 42). Coulouris, Dollimore und Kindberg (2005) nennen drei Aspekte, zu denen Replikation in verteilten Systemen entscheidend beiträgt: Performancesteigerung, erhöhte Verfügbarkeit und Fehlertoleranz. Somit trägt diese Technik entscheidend dazu bei, sowohl kontinuierliche Verfügbarkeit als auch konsistente Interaktion, beschrieben in 2.5, zu garantieren. Vom aus dem Netz nicht mehr wegzudenkenden Caching bis hin zu aufwendigeren Aufgaben wie Load-Balancing oder der Verarbeitung von DNS-Requests bietet das Internet zahlreiche Anwendungsfelder für Replikationen.



### 2.6.1 Pessimistische Replikation: Strong Consistency

Traditionelle Strategien, um die Replikationen auf dem selben Stand zu halten, folgen dem Modell der Strong Consistency (SC). SC setzt voraus, dass alle Replikationen stets identisch sind, als gäbe es konstant nur eine singuläre Kopie der Daten. Wenn ein Update auf einer Replikation erfolgt, muss es direkt auf allen weiteren Replikationen übernommen werden.

Es gibt ein weites Spektrum an Lösungen, um SC zu gewährleisten. Diese reichen von Update-Everywhere Systemen, die einzelne Änderungen sofort auf allen Replikationen speichern (Kemme, 2000) bis zu “primary copy” Lösungen (Bernstein, Hadzilacos & Goodman, 1987, S. 14), welche Änderungen von einem primären Datenspeicher auf alle weiteren Replikationen verteilen. Gemeinsam haben diese Algorithmen die Tatsache, dass sie keinen Zugriff auf Replikationen gewähren, welche nicht auf dem aktuellsten Stand sind (Saito & Shapiro, 2005, S. 43). Für Offline-First Anwendungen kommt diese Art der Replikation nicht in Frage. Die Anforderung, dass die Verbindung zum Netzwerk stets unterbrochen sein kann (vgl. 2.2), ist mit diesem Prinzip nicht vereinbar. Sobald eine Replikation vom Netzwerk getrennt ist, ist es unmöglich zu garantieren, dass sie auf dem aktuellsten Stand ist.

### 2.6.2 Optimistische Replikation: Eventual Consistency

Die optimistische Replikation, auch genannt Eventual Consistency (EC), ist ein alternatives Modell der Datenreplikation, welches den Replikationen erlaubt, voneinander abzuweichen.

Die Implementierung von optimistischer Replikation bietet sich somit als Lösung für Systeme an, welche besonderen Wert auf kontinuierliche Verfügbarkeit legen, wie Offline-First Applikationen (vgl. Sektion 2.5).

Bei der Verwendung von optimistischer Replikation sind Änderungen an Replikationen jederzeit gestattet, auch wenn diese nicht auf dem aktuellsten Stand sind oder keine Verbindung zum Netzwerk haben. Nehmen Nutzer/innen Änderungen vor, so werden diese auf der jeweiligen Replikation sofort umgesetzt. Im Hintergrund wartet die Applikation nun darauf, diese Änderung an die restlichen Computer des verteilten Systems weiterzugeben sowie selbst Änderungen entgegenzunehmen und zu verarbeiten (Saito &

Shapiro, 2005, S. 46). Ziel ist es, wie beim Modell der Strong Consistency, Einheitlichkeit unter den Replikationen herzustellen. Das Modell der EC setzt jedoch nicht voraus, dass diese Einheitlichkeit sofort erfolgen muss, sondern nur zu einem beliebigen späteren Zeitpunkt.

Da die Computer im System parallel Änderungen vornehmen können, kann es vorkommen, dass mehrere Replikationen das gleiche Datenobjekt modifizieren. Im Allgemeinen werden die Modifikationen zu unterschiedlichen Ergebnissen führen. Ist dies der Fall, spricht man von einem Konflikt. Das Ziel, Konvergenz zwischen den Replikationen zu erlangen, kann nur erreicht werden, wenn aus allen im Konflikt stehenden Änderungen eine einheitliche Lösung entsteht.

Deshalb muss ein System, welches EC implementiert, die Funktionalität aufweisen, Konflikte zu beheben. Problematisch dabei ist, dass die Replikationen nicht auf dem gleichen Stand sind, bis der Konflikt vollständig behoben ist, selbst nachdem sie ihre Änderungen untereinander ausgetauscht haben. Der Prozess der Konfliktbehandlung kann voraussetzen, auf die manuelle Konfliktlösung von Nutzer/innen oder die Daten anderer Replikationen zu warten (Terry et al., 1995, S. 174-175).

Gerade in Offline-First Anwendungen ist die Konfliktbehandlung eine große Herausforderung, wie in Abschnitt 1.1 ausgeführt wird. Um diese Herausforderung zu bewältigen, bietet das Modell der Strong Eventual Consistency (SEC) einen Ansatz, die Flexibilität von EC um die Sicherheit von SC zu erweitern.

### **2.6.3 Strong Eventual Consistency**

SEC beschreibt eine spezielle Form der Eventual Consistency (EC), welche das System von der Last der Konfliktbehandlung befreit. SEC garantiert, dass zwei Replikationen nach dem Austauschen ihrer Änderungen immer konvergent sind. Im Gegensatz zur EC wird die Konfliktbehandlung nicht vom System übernommen, stattdessen wird dieser Prozess durch die Nutzung spezieller Datentypen überflüssig.

## **2.7 Conflict-free Replicated Data Types (CRDTs)**

CRDTs sind abstrakte Datentypen, die in verteilten Systemen eingesetzt werden, um SEC zu ermöglichen. Sie basieren auf klassischen Datentypen wie Registern, Sets und

Maps. CRDTs erweitern diese Datentypen um eine Schnittstelle, welche das Daten-Objekt neben den klassischen Operationen wie dem Auslesen des gespeicherten Wertes um zusätzliche Funktionalitäten erweitert, um SEC zu gewährleisten (Preguiça, 2018, S. 1).

Der Satz an CRDT-Funktionalitäten enthält immer eine Funktion zum Aktualisieren des Objekts. Weitere Daten, um welche CRDTs klassische Datentypen erweitern, lassen sich als Metadaten beschreiben. Ihr Zweck ist es, die Aktualisierungs-Funktionalität möglich zu machen. Soll ein CRDT-Objekt beispielsweise so aktualisiert werden, dass sich die neueste Änderung des Wertes immer gegen ältere durchsetzt, muss zusätzlich zum Wert noch ein Zeitstempel der letzten Änderung verwaltet werden (vgl. Abschnitt 2.7.1). Die Datentypen werden speziell so modelliert, dass das Aktualisieren von CRDTs kommutativ, assoziativ und idempotent erfolgen kann (Martyanov, 2018).

**Kommutativ** Wenn zwei CRDTs Aktualisierungen austauschen, ist das Ergebnis identisch, unabhängig davon, in welcher Reihenfolge dies geschieht.

**Assoziativ** Wenn drei CRDTs nacheinander zusammengeführt werden, ist das Ergebnis immer gleich, unabhängig davon, welche zwei der Objekte zuerst Aktualisierungen austauschen.

**Idempotent** Das einmalige Zusammenführen zweier CRDTs hat das gleiche Ergebnis wie ein beliebig häufiges Wiederholen des Vorgangs.

Diese Eigenschaften führen dazu, dass sich zwei Replikationen deterministisch im gleichen Zustand befinden müssen, sobald sie den Stand ihrer Daten synchronisiert haben. Im Gegensatz zur EC ist das System nicht mehr von einer im Konsens zu geschehenden Konfliktlösung abhängig.

Dieser Vorteil von SEC ist gleichzeitig der größte Nachteil bei der Nutzung von CRDTs in der Praxis. Nicht alle Datenstrukturen lassen sich so modellieren, dass sie die benötigten oben genannten Anforderungen erfüllen. Zwar gibt es bereits viele gut dokumentierte, kompatible Datentypen (vgl. Sektion 2.1), dennoch bedarf die Verwendung von CRDTs und SEC ausgiebiger Planung und ist in manchen Fällen schlichtweg nicht möglich.

In der Literatur werden CRDTs in state-based und operation-based unterteilt (Saito & Shapiro, 2005, S. 10). Die Unterscheidung erfolgt danach, wie das Zusammenführen der Objekte funktioniert.

**State-based** Beim Synchronisieren von state-based CRDTs wird der gesamte Zustand der Objekte ausgetauscht. Die Implementierung eines state-based CRDTs muss über eine *Merge* Methode verfügen, damit der Zustand zweier CRDTs zu einem kombiniert werden kann, um so Konvergenz zu erreichen. Der Zustand eines state-based CRDTs wird durch einen Halbverband<sup>4</sup> gebildet, dessen obere Schranke den aktuellen Wert des CRDTs darstellt. (Almeida, Shoker & Baquero, 2015)

**Operation-based** Bei der Variante der operation-based CRDTs wird nicht der gesamte Zustand der Objekte ausgetauscht. Stattdessen erfolgt der Austausch über einzelne Updates. Wird der Zustand eines Objektes geändert, so wird die Operation, welche die Änderung hervorgerufen hat, gespeichert. Ändert sich beispielsweise der Wert eines Counters von 5 zu 10, so würde die Operation nicht den neuen Wert enthalten, sondern die durchgeführte Addition, also “+ 5”. Jede so erzeugte Operation muss an alle anderen Replikationen des Systems verteilt werden. Wenn eine Replikation eine solche Operation empfängt, wird die gleiche Änderung, welche schon im Ursprung angewendet wurde, auch hier angewandt. Da die Operationen kommutativ sein müssen, ist die Reihenfolge der Updates nicht relevant. Die Implementierung eines operation-based CRDTs muss garantieren, dass alle Replikationen diese Updates zuverlässig erhalten und anwenden (Preguiça, 2018, S.18-19). Im Gegensatz zu der *Merge* Methode des State-based CRDTs muss ein operation-based CRDT also über zwei Methoden verfügen: Eine zum Generieren von Operationen und eine weitere zum Anwenden von Operationen.

Beide Kategorien sind äquivalent (Shapiro et al., 2011, S. 9), was bedeutet, dass ein state-based CRDT ein operation-based CRDT emulieren kann und umgekehrt.

### 2.7.1 Last-Write-Wins Register

Ein Register ist ein Objekt, welches einen einzelnen Wert verwaltet. Dieser Wert kann jede vom System unterstützte Datenstruktur sein. Ein Last-Write-Wins Register (LWW-Register) verfügt neben dem Wert noch über einen Zeitstempel. Wird der Wert des Registers geändert, wird auch der Zeitstempel auf den Zeitpunkt dieser Änderung gesetzt. Beim Zusammenführen zweier Register kann so immer die neueste Änderung übernommen werden. Da sowohl Wert als auch Zeitstempel ausgetauscht werden,

---

<sup>4</sup>“Ein Halbverband ist eine kommutative Halbgruppe (d.h. eine Menge  $H$  mit einer assoziativen und kommutativen binären Verknüpfung  $\sqcap$ ), in der jedes Element idempotent ist  $[..]$ ” (Erné, 2006)

handelt es sich hier um ein state-based CRDT.

### **2.7.2 Last-Write-Wins Map**

Während ein LWW-Register ein einzelnes Schlüssel-Wert-Paar darstellt, besteht eine LWW-Map aus einem bis beliebig vielen LWW-Registern. (da Silva Tavares, 2018, S. 43).

### **2.7.3 Grow-Only Set**

Ein Set ist ein abstrakter Datentyp, welcher eine Sammlung von Objekten verwaltet. Traditionelle Operationen eines Sets umfassen das Hinzufügen und Entfernen von Objekten. Diese Operationen sind jedoch nicht kommutativ: Wird dem Set ein Objekt erst hinzugefügt und anschließend entfernt, ist es im Endeffekt nicht mehr im Set vorhanden. Wird es jedoch erst entfernt und anschließend hinzugefügt, so existiert das Objekt weiterhin im Set.

Ein Grow-Only Set (G-Set) löst dieses Problem, indem keine Objekte aus dem Set entfernt werden können (Baquero, Almeida & Shoker, 2017, S.17). Das Entfernen des Objektes kann durch eine “tombstone” Markierung ersetzt werden. Ist diese Markierung gesetzt, wird der Eintrag vom System so behandelt, als wäre er nicht vorhanden. Somit ist das Set wieder kommutativ, und die Funktionalität des Löschens bleibt bestehen (Aslan, Molli, Skaf-Molli & Weiss, 2011, S.7).

## 3 Konzeption

In diesem Kapitel geht es um die Idee und Planung des im Rahmen der Arbeit entwickelten Prototypen.

### 3.1 Idee des Prototypen

Ziel der Arbeit ist es, einen Prototypen zu entwickeln, welcher die Nutzung von CRDTs in Offline-First Webanwendungen demonstriert. Da die Datenstruktur hier im Mittelpunkt steht, liegt es nahe, als Prototyp eine Applikation zu entwickeln, in der Nutzer/innen auf verschiedenen Wegen mit Daten interagieren.

Deshalb wird eine Applikation zum Verwalten von Kochrezepten entwickelt, welche es Nutzer/innen ermöglicht, ein gemeinsames Rezeptebuch zu pflegen. Da die Nutzer/innen in der Lage sein sollen, online, offline und zur gleichen Zeit Rezepte anlegen, speichern und bearbeiten zu können, ist dies ein geeignetes Szenario, um die Nutzung von CRDTs zu untersuchen.

### 3.2 Anforderungsanalyse

In diesem Abschnitt werden die funktionalen und nicht-funktionalen Anforderungen an den im Rahmen dieser Arbeit entwickelten Prototypen beschrieben.

#### 3.2.1 Funktionale Anforderungen

Zweck der funktionalen Anforderungen ist es, Verhalten und Funktionalität des zu entwickelnden Systems zu beschreiben. Die formulierten Anforderungen sollen somit einen

Überblick darüber geben, was das implementierte System erfüllen muss. Da das Ziel dieser Arbeit die Entwicklung einer Datenstruktur für eine Offline-First Applikation ist, liegt der Fokus der gesammelten Anforderungen eindeutig auf dem Zusammenspiel zwischen Offline-First und CRDTs.

Üblich in einer Anforderungsanalyse ist, Anforderungen in die Kategorien notwendig (“muss”), wünschenswert (“soll”) und möglich (“kann”) einzuteilen. Da es sich hier um die Entwicklung eines Prototypen handelt, wird sich in diesem Teil exklusiv auf notwendige Anforderungen beschränkt. Hierbei handelt es sich um Anforderungen, welche essenziell für das System sind.

**[A 01] Anlegen von Rezepten** Nutzer/innen sollen Rezepte anlegen können. Ein Rezept hat einen Namen und eine Liste an Zutaten.

**[A 02] Anlegen von Zutaten** Die Nutzer/innen sollen in der Lage sein, Zutaten zu einem Rezept hinzuzufügen. Zutaten haben einen Namen und eine Mengenangabe, welche die Nutzer/innen setzen können.

**[A 03] Bearbeiten und Löschen** Den Nutzer/innen soll es möglich sein, Rezepte und Zutaten zu bearbeiten und zu löschen.

**[A 04] Zugriff mit mehreren Endgeräten gleichzeitig** Nutzer/innen sollen in der Lage sein, mit mehreren Geräten gleichzeitig auf die gleiche Applikation zuzugreifen.

**[A 05] Offline-Erreichbarkeit** Nutzer/innen der Anwendung müssen in der Lage sein, die Anwendung offline zu nutzen. Der Funktionsumfang der Applikation soll dadurch nicht beeinflusst werden.

**[A 06] Lokales Speichern** Die Anwendung muss über die Funktionalität verfügen, Änderungen von Nutzer/innen lokal zu speichern, damit diese auch zur Verfügung stehen, wenn keine Verbindung zum Netzwerk besteht.

**[A 07] Online Speichern** Die Anwendungsdaten sollen nicht exklusiv lokal, sondern auch im Internet gespeichert werden können. Das Speichern der Daten erfolgt sinnvollerweise auf einem Server.

**[A 08] Synchronisierung von Daten** Änderungen der Nutzer/innen müssen mit dem Server synchronisiert werden können, wenn eine Verbindung zum Internet besteht. Einzelne Nutzer/innen sollten niemals ihre offline umgesetzten Änderungen verlieren.

**[A 09] SEC** Nachdem sich zwei Nutzer/innen mit dem Stand des jeweils anderen Nutzens synchronisiert haben, müssen beide Applikationen den gleichen Stand haben.

**[A 10] Konfliktfreies Synchronisieren** Das Synchronisieren der Daten soll ohne Konflikte erfolgen.

### 3.2.2 Nicht-funktionale Anforderungen

Während die funktionalen Anforderungen direkt Bezug auf die Funktionalität des Prototypen nehmen, werden zusätzlich sogenannte nicht-funktionale Anforderungen definiert, um die Rahmenbedingungen des Projekts einzuschränken. Diese beziehen sich nicht auf die konkreten Funktionen, sondern beschreiben stattdessen Ziele, welche die Umsetzung des Prototypen über die Funktionalität hinaus hat.

**[A 11] Anschauliche Umsetzung: Code** Ein Ziel der Arbeit ist, zu ermitteln, wie CRDTs in modernen Webanwendungen umgesetzt werden können (vgl. 1.1). Da sich moderne Offline-First Anwendungen leider nicht auf ein einziges Beispiel reduzieren lassen, ist bei der Umsetzung darauf zu achten, die Datenstruktur so anschaulich und verständlich wie möglich zu implementieren. Dies bedeutet beispielsweise, Lesbarkeit über Effizienz beim Schreiben von Programmcode zu priorisieren.

**[A 12] Unabhängigkeit von Datenbank-Technologien** Die implementierte Datenstruktur muss unabhängig von der benutzten Datenbank sein. Keine der umgesetzten CRDT-Funktionalitäten darf von der gewählten Datenbank abhängig sein.

## 3.3 Architektur

Da Daten sowohl online als auch offline gespeichert werden sollen, folgt die Umsetzung der Applikation dem für Webseiten üblichen Client-Server Modell.

Dass der Prototyp als Offline-First Anwendung umgesetzt wird und umfangreiche Offlinefunktionalität zentraler Bestandteil der funktionalen Anforderungen ist, wirkt sich entscheidend auf die Architektur aus.



Die Nutzung eines Service-Workers ermöglicht das Zwischenspeichern der gesamten Clientseitigen Dateien, welche zum Ausführen der Applikation benötigt werden. Da die Speicherung der Anwendungsdaten auch lokal erfolgen soll, ist die Anwendung nicht darauf angewiesen, bei jeder Aktion der Nutzer/innen Daten an den Server zu schicken. Die Applikation ist bewusst so gestaltet, dass ein Synchronisieren nur stattfindet, wenn der ein entsprechender Knopf den Prozess startet. In einer Applikation welche veröffentlicht werden soll, wäre es Sinnvoll, diesen Prozess öfter im Hintergrund auszuführen. Im Prototypen wäre es dadurch jedoch schwieriger die Zustände “synchronisiert” und “nicht synchronisiert” darzustellen.

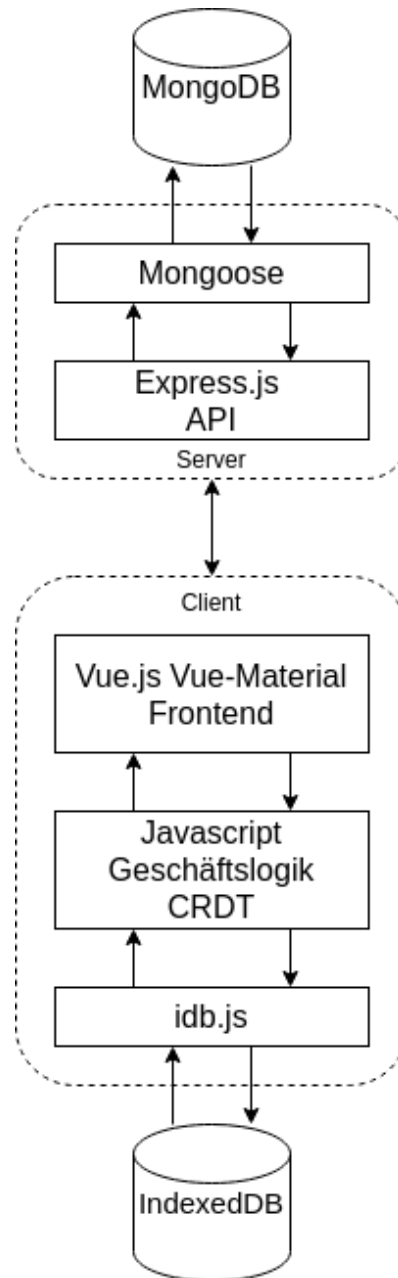


Abbildung 5: Architektur des Prototypen (eigene Darstellung)

Abbildung 5 zeigt die Architektur des Prototypen. Die Clientseite setzt sich zusammen aus Frontend, Geschäftslogik und der Browser-API IndexedDB als lokalen Speicher. Zusätzlich wird die JavaScript-Bibliothek idb.js für den Zugriff auf IndexedDB genutzt. Die CRDT Funktionalitäten befinden sich in dieser Darstellung in der Geschäftslogik.

Die Serverseite beginnt mit einer API, welche nur einen einzelnen Endpunkt (`/sync`) zur Synchronisierung von lokalen und Online-Daten bereitstellt. Über diese Schnittstelle

kann der Client seine Operationen an den Server schicken und erhält als Antwort die gesammelten Operationen des Servers. Das serverseitige Speichern der Applikationen erfolgt auf einer MongoDB Datenbank. Ergänzend zu MongoDB wird noch die Bibliothek Mongoose genutzt. Als Laufzeitumgebung zum Betreiben des Webservers dient Node.js. Eine ausführliche Erläuterung, aus welchen Gründen die bestimmten Technologien genutzt werden, findet sich in Abschnitt 4.1.

## 3.4 Entwurf der Nutzeroberfläche

Aus den funktionalen Anforderungen (siehe Abschnitt 3.2.1) lässt sich ein Entwurf für die Nutzeroberfläche des Prototypen erstellen. Abbildung 6 zeigt den Hauptbildschirm der Website. Hier können Nutzer/innen Rezepte hinzufügen, löschen und die Namen der angelegten Rezepte bearbeiten. Der “Sync” Button initialisiert das Synchronisieren mit dem Server. Über den Knopf “Zutaten” gelangen Nutzer/innen zur Übersicht der Zutaten des jeweiligen Rezepts. Wie auf Abbildung 7 zu erkennen ist, können Nutzer/innen hier neue Zutaten für das Rezept anlegen und bestehende Zutaten bearbeiten oder löschen. Darüber hinaus können sie jeder angelegten Zutat eine Menge zuweisen und auch diese bearbeiten.



Abbildung 6: Entwurf Nutzeroberfläche  
Hauptbildschirm (eigene  
Darstellung)



Abbildung 7: Entwurf Nutzeroberfläche  
Zutaten (eigene Darstel-  
lung)

Wenn Nutzer/innen Änderungen über die Nutzeroberfläche einleiteten, bekommen sie diese schon zu sehen, bevor die Änderung in der Datenbank gespeichert wurde. Diese Praxis nennt sich Optimistic UI und ist ein gängiges Mittel um die Usability in modernen Applikationen zu steigern (Mishunov, 2016).

### 3.5 Datenstruktur

Dieser Abschnitt beschreibt die Planung der Datenstruktur für den Prototypen. Erster Schritt dieser Planung ist es, zu ermitteln, welche Daten von der Datenstruktur verwaltet werden müssen und wie diese von der Applikation genutzt werden. Das Vorgehen ist, die Grundlagen der Datenstruktur erst in konventionellen Datentypen zu entwerfen und diese anschließend um CRDT Funktionalitäten zu erweitern. Dies bietet sich an, da CRDTs üblicherweise auf konventionellen Datentypen aufbauen. Die Datenstruktur soll

schließlich auf die Applikation zugeschnitten sein und nicht umgekehrt. Würde mit dem Entwurf des CRDT begonnen werden, bevor die Grundlagen der Applikation festgelegt sind, müsste sich die Applikation der Datenstruktur anpassen. Dies gilt es zu vermeiden, es sei denn, die Rahmenbedingungen fordern aus anderen Gründen eine bestimmte Datenstruktur, was in dieser Arbeit jedoch explizit nicht der Fall ist. Ziel des Prototypen ist es, die Datenstruktur an die Applikation anzupassen.

### 3.5.1 Grundlagen der Datenstruktur

Die in Sektion 3.2 formulierten Anforderungen zeigen, dass es in der Applikation um das Speichern, Bearbeiten und Löschen von Rezepten und ihren Zutaten geht. Nun gilt es, zu modellieren, wie ein solches Rezept als Objekt in der Applikation aussieht.

Listing 1 zeigt die Struktur eines Rezeptes als einzelnes JavaScript Objekt. Neben einer ID verfügt es noch über die Eigenschaften “name”, welche den Namen des Rezepts speichert und die Eigenschaft “ingredients”. Bei “ingredients” handelt es sich um ein Array an Objekten, welche jeweils eine Zutat mit Namen (“name”) und Mengenangabe (“measure”) abbilden. Darüber hinaus erhält das Rezept zur Identifikation eine generierte ID. Da, wie in Sektion 5 erläutert, das lokale Speichern über die IndexedDB-API erfolgt, können die Rezepte auch direkt als Objekte in der Datenbank des Browsers gespeichert werden. Auch die Verschachtelung der Objekte stellt kein Problem dar, da IndexedDB auch Objekte mit Objekten als Eigenschaft speichert. Zwischen Applikation und lokalem Speichern muss folglich keine Umwandlung der Daten erfolgen.

---

**Listing 1** Beispiel Rezept verschachtelt

---

```
1 var exampleRecipe = {
2   _id: "re_ckdiwlbxw002411w66m7mol85s",
3   name: "Tomatensalat",
4   ingredients: [
5     {
6       name: "Tomaten",
7       measure: "500g",
8     },
9     {
10      name: "Zwiebeln",
11      measure: "1",
12    },
13    {
14      name: "Olivenöl",
15      measure: "3 EL",
16    },
17    {
18      name: "Essig",
19      measure: "1 EL",
20    },
21  ],
22 };
```

---

Ein solches Modell ist nicht unüblich und gerade in JavaScript Anwendungen häufig praktisch, weil Objekte in diesen oft verschachtelt sind. Beim Einsatz von CRDTs jedoch lohnt es sich, die Datenstruktur so flach wie möglich zu gestalten. Wie in Abschnitt 2.7 beschrieben, bauen CRDTs auf herkömmlichen Datenstrukturen auf. Wenn es die Möglichkeit gibt, die Komplexität der ausgehenden Datenstruktur zu reduzieren, bedeutet dies auch eine Reduzierung an Komplexität für das zu implementierende CRDT.

Eine Option, die Datenstruktur abzuflachen, besteht darin, die in Listing 1 gezeigten Daten zu normalisieren und somit relational abzubilden. In diesem Fall werden die Zutaten nicht als Array in dem Rezept verschachtelt, sondern werden einzeln gespeichert. Listing 2 zeigt ein Rezept in der relationalen Modellierung, Listing 3 eine dazugehörige Zutat. Die Verbindung zwischen Rezepten und Zutaten wird nun über die Eigenschaft “recipe” der Zutaten definiert. Hier wird die ID des Rezepts hinterlegt, welchem die Zutat zugehörig ist.

---

**Listing 2** Beispiel Rezept relational

---

```
1 var exampleRecipe = {
2   _id: "re_ckdiw1bxw002411w66m7mol85s",
3   name: "Tomatensalat"
4 }
```

---

---

**Listing 3** Beispiel Zutat relational

---

```
1 var exampleIngredient = {
2   _id: "in_ckdpnyi0c00081w6j7q4nwb6",
3   name: "Tomaten",
4   measure: "500g",
5   recipe: "re_ckdiw1bxw002411w66m7mol85s"
6 }
```

---

### 3.5.2 Erweiterung zum CRDT

Während sich bei konventionellen Datentypen nicht viel dadurch ändert, ob eine Zutat über “recipe.ingredient”, wie es in Listing 1 der Fall wäre, oder “ingredient”, vgl. Listing 2 angesprochen wird, wäre die Umsetzung als CRDT in der verschachtelten Variante deutlich komplexer. Die Ursache dafür ist, dass die Änderungen an Zutaten schließlich auch von CRDT-Funktionalitäten profitieren sollten. Das Modell für ein Rezept müsste deshalb als CRDT entworfen werden, welches ein weiteres CRDT als Eigenschaft hält. Dies ist in der Umsetzung durchaus möglich, in diesem Fall aber leicht durch die relationale Herangehensweise zu vermeiden.

Da in der Applikation kein universell einsetzbares CRDT entworfen werden soll, sondern eines, welches den Anforderungen der Anwendung gerecht wird, gilt es nun zu ermitteln, welche Schnittstellen es der Applikation zur Verfügung stellen muss. Aus der Anforderungsanalyse, welche in Abschnitt 3.2 durchgeführt wurde, lassen sich die Anforderungen an das CRDT ableiten. Tabelle 1 fasst diese als Schnittstellen zusammen, welche die Datenstruktur der Applikation bereitstellen muss.

Tabelle 1: CRDT Schnittstellen

Funktionale Anforderung	Anforderung an die Datenstruktur
Anlegen von Rezepten	Verwaltung einer Sammlung von beliebig vielen Rezepten Hinzufügen eines Rezepts in die Rezeptesammlung
Anlegen von Zutaten	Verwaltung einer Sammlung von beliebig vielen Zutaten Hinzufügen einer Zutat in die Zutatenammlung
Bearbeiten und Löschen	Bearbeitung der Eigenschaften Rezeptname, Zutatenname, Zutatenmenge Entfernen eines Rezepts aus der Rezeptesammlung Entfernen einer Zutat aus der Zutatenammlung

Die Umsetzung im Prototypen erfolgt durch ein “Operation-based” CRDT. Da beide möglichen Formen gleichwertig sind (vgl. Abschnitt 2.7) wurde diese Entscheidung getroffen weil “Operation-based” CRDTs unter Umständen eine geringere Nutzlast für Hypertext Transfer Protocol (HTTP)-Requests und Responses bei der Synchronisierung verursachen. Dies liegt daran, dass “Operation-based” CRDTs nicht darauf angewiesen sind, ihren gesamten Zustand auszutauschen (vgl. ebenfalls Abschnitt 2.7).

Wie in Abschnitt 2.7 erläutert, werden alle Änderungen, welche an den Applikationsdaten erfolgen, einzeln als Operationen abgespeichert. Wie Rezepte und Zutaten werden auch die Operationen über die IndexedDB-API lokal im Browser gespeichert. Neben der Funktion zum Erzeugen von Operationen muss das CRDT auch eine Funktion zum Anwenden der Operationen implementieren.

Listing 4 zeigt eine Operation, welche beim Umbenennen des in Listing 2 entworfenen Rezepts erzeugt und gespeichert wird. Die Felder “store”, “object” und “key” dienen der Identifikation des Wertes, der geändert werden soll, “value” beschreibt den neuen Wert. Die dargestellte Operation dokumentiert also, dass im Object-Store “recipes” der Wert des Schlüssels “name” vom Objekt “re\_ckdiw1bxw002411w66m7mol85s” zu “Tomaten-Paprika-Salat” geändert werden soll. Darüber hinaus erhält die Operation noch eine eigene ID und einen Zeitstempel. Der Zeitstempel dient dazu, eine konfliktfreie Synchronisation zu gewährleisten, mehr dazu im nächsten Abschnitt.



---

**Listing 4** Beispiel Operation

---

```
1 var operation = {
2   _id: "op_ckdpnyar800061w6jtko9ew8k",
3   store: "recipes",
4   object: "re_ckdiw1bxw002411w66m7mol85s",
5   key: "name",
6   value: "Tomaten-Paprika-Salat",
7   timestamp: 1597133338100,
8 };
```

---

### 3.5.3 Parallele Änderungen

Zwei Replikationen können sich synchronisieren, indem sie ihre gespeicherten Operationen austauschen. Nachdem eine Replikation über diesen Weg eingehende Operationen erhalten hat, müssen die Operationen auf die Applikationsdaten angewendet werden, damit die Änderungen einen Effekt haben. Nachdem eine eingehende Operation angewandt wurde, wird sie zu den lokalen Operationen abgespeichert.

Beim Anwenden der Operationen kann es vorkommen, dass zwei Operationen Änderungen am gleichen Wert vorgenommen haben. Für solche Fälle, genannt parallele Änderungen, müssen Regeln implementiert werden, um zu entscheiden, welche der parallelen Änderungen den Vorzug bekommt. Nur so ist gewährleistet, dass zwei Replikationen nach dem Synchronisieren auf dem gleichen Stand sind. Wäre die Auswahl zufällig oder würde sich stets die lokale Änderung durchsetzen, könnten die Applikationsdaten voneinander abweichen. So wäre Konvergenz zwischen den Replikationen und somit SEC nicht gewährleistet und der Einsatz von CRDTs überflüssig.

Im Prototypen werden parallele Änderungen mit Last-Write-Wins (LWW) Regeln behandelt, wodurch sich einzelne Rezepte und Zutaten als LWW-Map Bezeichnen lassen (vgl. Abschnitt 2.7.2). Abbildung 8 zeigt, wie entschieden wird, ob eine eingehende Operation angewendet wird. Zuerst wird überprüft, ob es für eine eingehende Operation, welche angewendet werden soll, bereits eine lokale Operation gibt, welche denselben Wert betrifft. Ist dies der Fall, werden die Zeitstempel der aktuellsten lokalen Operation und der eingehenden Operationen verglichen. Ist die lokale Operation neuer als die eingehende Operation, wird die eingehende Operation ignoriert. Ist hingegen die eingehende Operation neuer, wird sie angewendet und anschließend, wie vorgesehen, zu den

lokalen Operationen hinzugefügt. Dies verhindert auch das mehrfache Anwenden derselben Operation, wodurch das Verarbeiten eingehender Operationen idempotent ist. Dies ist eine wichtige Eigenschaft für das CRDT (vgl. Abschnitt 2.7).

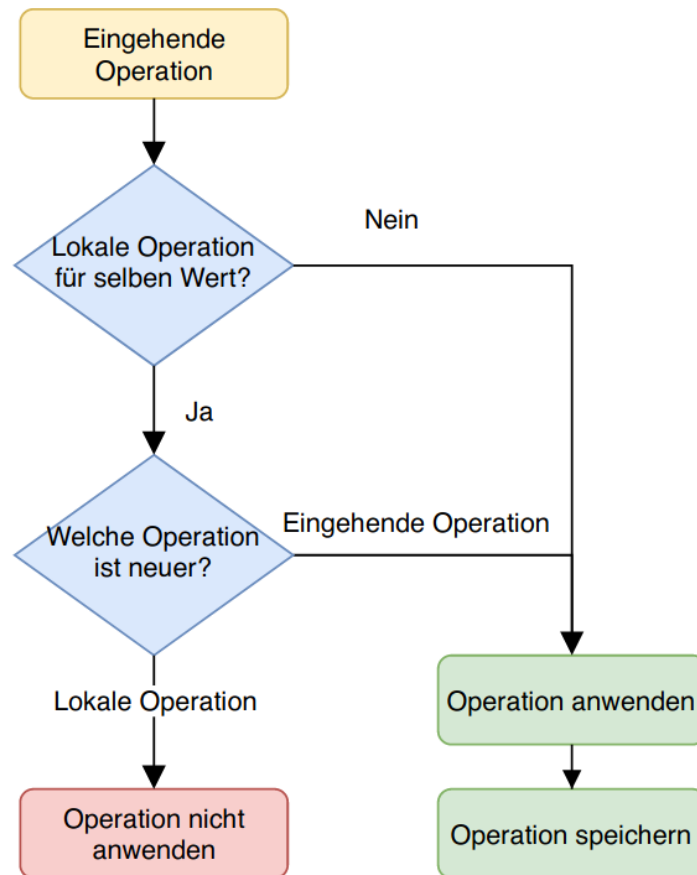


Abbildung 8: Bearbeiten einer eingehenden Operation (eigene Darstellung)

### 3.5.4 Anwenden von Operationen

Auch beim Anwenden von Operationen gilt es, einigen potentiellen Fehlern aus dem Weg zu gehen. Da die Operationen kommutativ anwendbar sein sollen, kann es vorkommen, dass ein Objekt bearbeitet werden soll, welches lokal noch nicht existiert. Damit eine solche Änderung nicht verloren geht, wird das Objekt beim Anwenden der Änderung erstellt, sollte es noch nicht existieren. Wichtig hierbei ist, dass das neue Objekt keine zufällige ID zugewiesen bekommt, sondern die in der Operation hinterlegte ID des fehlenden Objektes. Anschließend kann die Operation wie vorgesehen auf das neue Objekt angewandt werden. Durch diese Semantik ist es möglich, die “Erstellen” und

“Bearbeiten” Schnittstellen (vgl. Tabelle 1) mit der gleichen Methode umzusetzen. Eine Operation zum Erstellen eines Objektes wird einfach genau wie eine Operation zum Bearbeiten angelegt, statt einer bestehenden ID beim Bearbeiten wird jedoch eine neue ID erzeugt. Somit wird das Objekt beim Anwenden der Operation erzeugt, da es sich um eine unbekannte ID handelt.

Ein weiterer Fehlerfall droht beim Entfernen von Rezepten und Zutaten, welches neben dem Anlegen und Bearbeiten auch eine weitere Schnittstelle zur Datenstruktur ist (vgl. Tabelle 1). Abbildung 9 zeigt die Problematik, welche auftritt, wenn das Objekt tatsächlich aus der Datenbank gelöscht wird. Wird ein Objekt gelöscht und anschließend bearbeitet, kommt die Applikation nicht auf den gleichen Stand wie wenn die Operationen in umgekehrter Reihenfolge angewendet werden. Das liegt daran, dass das gelöschte Objekt beim anschließenden Bearbeiten wieder erzeugt wird.

Somit wäre erneut das Kommutativgesetz verletzt und die Applikationsdaten könnten sich nach dem Synchronisieren unterscheiden.

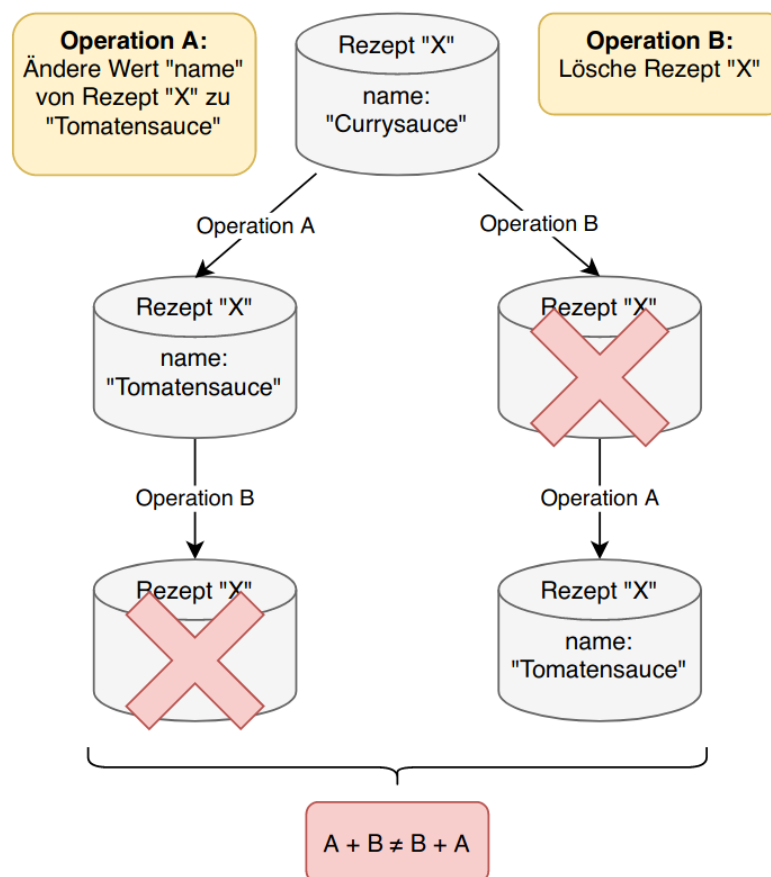


Abbildung 9: Fehlerfall beim Entfernen eines Rezeptes (eigene Darstellung)

Im Prototyp werden die Rezepte- und Zutatenansammlungen deshalb als G-Set (vgl. Abschnitt 2.7.3) implementiert. Dies bedeutet, dass die Elemente des Sets, also die Rezepte und Zutaten, nicht mehr gelöscht werden können, nachdem sie einmal angelegt wurden. Stattdessen wird das Entfernen der Objekte über eine weitere Eigenschaft gelöst, den sogenannten “tombstone”. Beim “tombstone”-Wert handelt es sich um einen Statusindikator, welcher bestimmt, ob ein Objekt von der Applikation als aktiv oder inaktiv behandelt werden soll. Somit kann auch das Entfernen eines Objektes auf die gleiche Herangehensweise erfolgen wie das Anlegen und Bearbeiten.

---

**Listing 5** Beispiel Entfernen-Operation

---

```
1 var operation = {
2   _id: "op_ckdpnyar800051w6jocpy5sjm",
3   store: "recipes",
4   object: "re_ckdiw1bxw002411w66m7mol85s",
5   key: "tombstone",
6   value: "1",
7   timestamp: 159713335680,
8 };
```

---

Listing 5 zeigt, wie eine Entfernen-Operation ebenso Informationen über das Zielobjekt enthält. Der angesprochene Schlüssel ist dabei immer *tombstone*, der Wert “1” indiziert, dass das Objekt als entfernt zu behandeln ist. Wäre eine Schnittstelle zum Reaktivieren des Objekts gewünscht, so könnte dies durch das Ändern des *tombstone* Werts auf “0” umgesetzt werden. Abbildung 10 zeigt, wie die Anwendung der vorher fehlerverursachenden Operationen durch die Nutzung des *tombstone* Indikators wieder kommutativ ist.

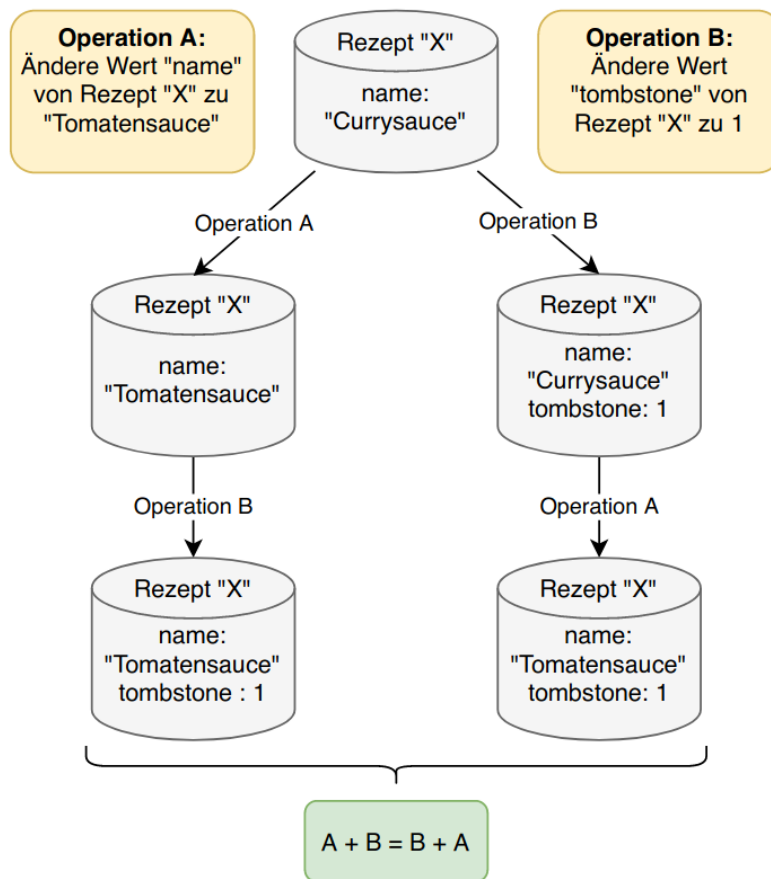


Abbildung 10: Korrektes Entfernen eines Rezeptes (eigene Darstellung)

## 3.6 Server

Da die Replikationen im Client-Server Modell nicht direkt miteinander kommunizieren können, ist es Aufgabe des Servers, dafür zu sorgen, dass dennoch Operationen ausgetauscht werden können. Dieser Abschnitt erläutert, wie der Server dazu genutzt wird, dass Clients indirekt miteinander kommunizieren können.

### 3.6.1 Aufgabe des Servers

Voraussetzung eines “operation-based” CRDTs ist es, die erzeugten Operationen verlässlich an die anderen Replikationen des Systems, im Falle des Prototypen also die anderen Clients, zu verteilen (vgl. Abschnitt 2.7). Deshalb dient der Server in der Umsetzung des

Prototypen als eine Art Cloud-Speicher für Operationen, auf dessen Schnittstelle alle Clients Zugriff haben.

### **3.6.2 API**

Die Schnittstelle, welche der Server den Clients bereitstellt, ist eine API mit genau einem Endpunkt. Bei diesem handelt es sich um eine POST-Route, welche ein Array an eingehenden Operationen als HTTP-Request akzeptiert. Nachdem die eingehenden Operationen vom Server verarbeitet wurden, werden die lokalen Operationen des Servers als HTTP-Response zurückgeschickt.

### **3.6.3 Datenbank**

Eingehende Operationen werden unter den gleichen Konditionen in der Datenbank des Servers gespeichert, wie sie in Abbildung 8 beschrieben sind. Im Gegensatz zu dem Client wendet der Server jedoch keine Operationen an. Die Datenbank des Servers dient exklusiv der Speicherung von Operationen, Rezepte und Zutaten verwaltet er nicht. Dementsprechend muss die Datenbank des Servers keine weiteren Aufgaben erfüllen als das Lesen und Schreiben einer einzelnen Tabelle.

### **3.6.4 Synchronisation**

Um sich mit dem Server zu synchronisieren, schickt ein Client erst alle seine lokalen Operationen an den Server, um anschließend alle auf dem Server gespeicherten Operationen als Antwort zu bekommen.

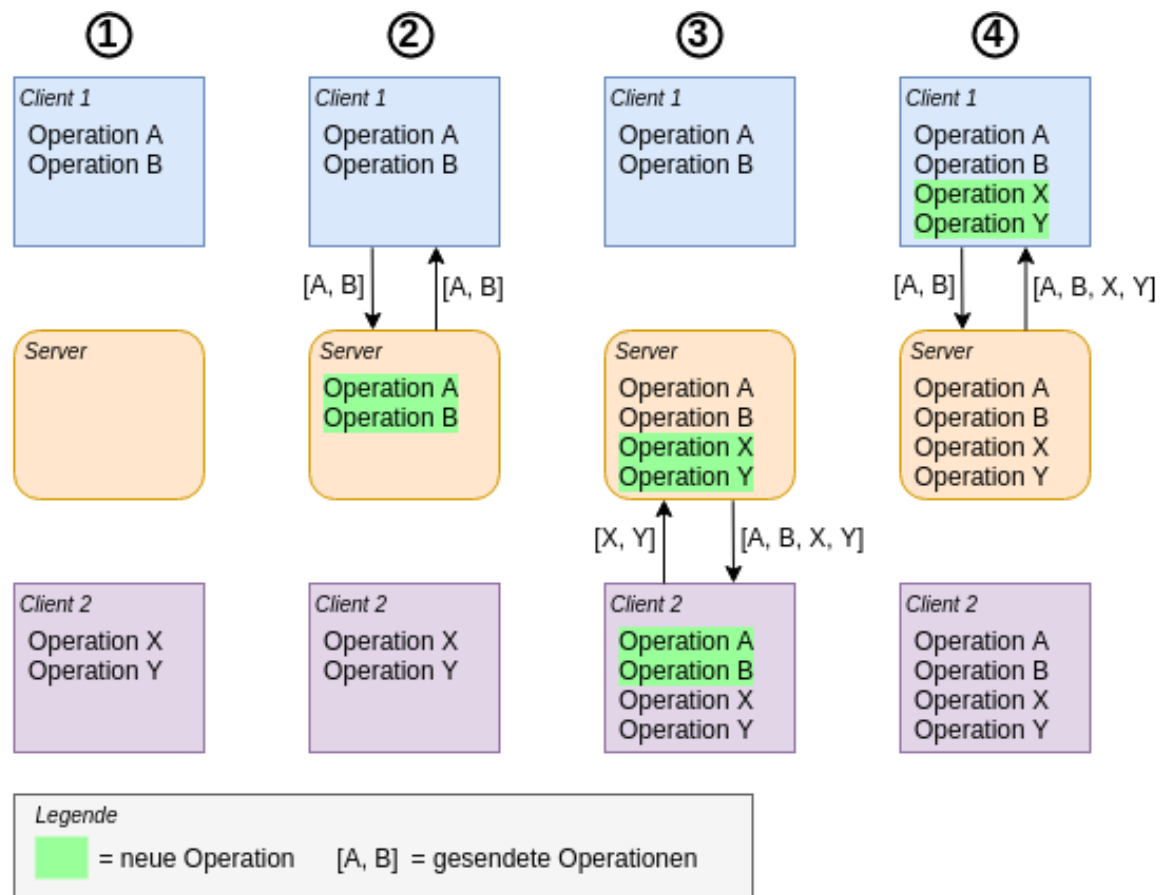


Abbildung 11: Synchronisieren über den Server (eigene Darstellung)

Abbildung 11 demonstriert das Synchronisieren zweier Clients in vier Schritten. Diese lassen sich wie folgt beschreiben:

**Schritt 1** dokumentiert die Ausgangssituation. Beide Clients verfügen über unterschiedliche Operationen, während auf dem Server noch keine Operationen gespeichert sind. Für dieses Beispiel ist anzunehmen, dass Operationen A und B nicht die gleichen Werte bearbeiten wie Operationen X und Y, die Regeln für den Fall das gleiche Werte geändert werden, sind in Abschnitt 3.5.3 erklärt.

**Schritt 2** zeigt die Synchronisation von Client 1 und dem Server. Zuerst schickt der Client seine Operationen an den Server, welcher diese speichert, da die Operationen zuvor nicht in seiner Datenbank existierten. Als Antwort schickt der Server seine Operationen zurück. Dies hat jedoch keinen Effekt auf die Daten des Clients, da die gleichen Operationen zurückkommen, welche sich schon in der lokalen Datenbank befinden. Beim Bearbeiten eingehender Operationen (vgl. Abbildung 8) werden

diese also ignoriert.

**Schritt 3** beschreibt die anschließende Synchronisation von Client 2 und dem Server. Client 2 initiiert diesen, indem er seine Operationen X und Y an den Server sendet. Der Server speichert diese, da sie noch nicht in seiner Datenbank vorhanden sind und die Operationen andere Werte betreffen als Operationen A und B. Daraufhin antwortet der Server und schickt die Sammlung seiner gespeicherten Operationen zurück. Beim Bearbeiten der Operationen wendet Client 2 die neuen Operationen A und B an und speichert sie in seiner Datenbank. Die Operationen X und Y hingegen werden ignoriert, da diese schon in der Datenbank des Clients vorhanden sind.

**Schritt 4** stellt dar, wie Client 1 sich erneut mit dem Server synchronisiert. Seit Schritt 1 hat sich auf dem Client nichts geändert, weshalb er die Synchronisation wieder mit dem Senden der Operationen A und B beginnt. Der Server kennt diese schon, fügt seiner Datenbank also keine Operationen hinzu und fährt fort, indem er seine Operationen an den Client sendet. In der Antwort des Servers sind nun auch die Operationen von Client 2 enthalten, welche den Server in Schritt 3 erreichten. Anschließend wendet Client 1 die eingehenden neuen Operationen X und Y an und speichert sie in seiner Datenbank. Somit sind Client 1 und Client 2 auf dem gleichen Stand.



# 4 Prototypische Realisierung

Dieses Kapitel beschreibt die Realisierung des Prototypen auf Basis des zuvor angefertigten Konzeptes. Zuerst wird auf die Wahl der bei der Umsetzung verwendeten Technologien eingegangen, anschließend folgen Erläuterungen zu den implementierten Funktionalitäten.

## 4.1 Wahl der verwendeten Technologien

Es folgt eine kompakte Vorstellung der zur Umsetzung des Prototypen genutzten Technologien. Auch wird erläutert, warum die einzelnen Technologien verwendet werden und welchen Einfluss sie auf die Implementierung des Prototypen haben.

### 4.1.1 JavaScript, HTML und CSS

JavaScript ist eine Skriptsprache welche zur clientseitigen Programmierung von Websites genutzt wird. Mit einer Nutzung in über 95% (W3Techs, 2020) aller Websites ist sie aus dem Internet nicht wegzudenken. In den letzten Jahren gewannen auch Sprachen an Popularität, die zu JavaScript kompilieren, wie zum Beispiel TypeScript oder ClojureScript. Da JavaScript die Basis dieser Sprachen bildet und sich gut in diese übersetzen lässt, ist eine Implementierung in der “Grundsprache” am besten dazu geeignet, die Anforderung einer anschaulichen und verständlichen Umsetzung zu erfüllen.

### 4.1.2 VueJS

Da es sich beim Prototypen nicht um eine statische Website handelt, sondern oft Änderungen an der Nutzeroberfläche oder an den Applikationsdaten vorgenommen werden, bietet sich die Nutzung eines Frameworks an, welches diese Interaktionen vereinfacht.

VueJS ist ein JavaScript Framework zum Erstellen von Benutzeroberflächen. Die Nutzung von VueJS erlaubt die Verknüpfung von HTML-Elementen mit Anwendungsdaten. So können HTML-Elemente automatisch aktualisiert werden, wenn sich ein Wert in einem Vue-Objekt ändert. Diese Verknüpfung macht auch Änderungen in die umgekehrte Richtung möglich, also die Aktualisierung von Applikationsdaten bei Änderungen, welche auf Seite der Nutzeroberfläche gemacht werden.

Eine Funktionalität von VueJS ist die Erzeugung von “components”. Hierbei handelt es sich um VueJS Instanzen, welche in HTML wiederverwendet werden können. Bei der Umsetzung des Prototypen wird die Komponenten-Bibliothek vue-material genutzt, um Zugriff auf einige vorgefertigte Komponenten wie Cards, Buttons und Eingabefelder zu haben.

### **4.1.3 IndexedDB**

IndexedDB ist eine Browser-API, welche das clientseitige Speichern von Daten im Browser ermöglicht. Das bedeutet, dass über IndexedDB gespeicherte Daten der Applikation auch offline zur Verfügung stehen. Die Nutzung der IndexedDB Schnittstelle bietet dem Prototypen die Funktionalität der Speicherung von Offline-Daten, welche für die Erfüllung der funktionalen Anforderungen rund um Offline-First-Eigenschaften essenziell ist.

Über sogenannte Object-Stores können JavaScript Objekte gespeichert und ausgelesen werden. Die Abfrage der Objekte erfolgt über einen der Werte des Objekts, welcher als Schlüssel konfiguriert wird. Alle Änderungen, welche an der Datenbank vorgenommen werden, erfolgen in Transaktionen. Änderungen an der Datenbank bestehen nur, wenn eine Transaktion erfolgreich war. Bei Komplikationen wird die Transaktion abgebrochen, wodurch die an der Datenbank während der Transaktion vorgenommenen Änderungen rückgängig gemacht werden. Somit bieten Transaktionen einen gewissen Schutz von Anwendungs- und Systemfehlern. Dies ist ein wichtiger Aspekt für die fehlerfreie Gewährleistung der CRDT-Funktionalitäten.

### **4.1.4 idb**

Die Event- und Callback-basierte Schnittstelle, welche IndexedDB bietet, führt dazu, dass sehr komplexer Code geschrieben werden muss, um simple Operationen an der Da-

tenbank durchzuführen (Kimak & Ellman, 2015). Um dieses Problem zu vermeiden und schlüssigen, lesbaren Code zu schreiben, wird bei der Umsetzung des Prototypen auf die JavaScript-Bibliothek idb zurückgegriffen. Bei idb handelt es sich um einen API-Wrapper, welche die API vereinfacht und Zugriff auf die Datenbank über JavaScript Promises anstelle von Events und Callbacks bietet. Listing 6 zeigt das Erzeugen eines Object-Stores über die gewöhnliche IndexedDB-API, in Listing 7 ist die gleiche Aktion mit der idb API umgesetzt. Während in der gewöhnlichen Variante mit Callbacks weitergearbeitet werden muss, gibt die idb API mit “idb.open” ein Promise zurück.

---

**Listing 6** IndexedDB: öffnen eines Object-Stores

---

```
1 function initDB() {
2   var request = indexedDB.open("example-db", 1);
3   request.onsuccess = function (evt) {
4     db = request.result;
5   };
6
7   request.onupgradeneeded = function (evt) {
8     var objectStore = evt.currentTarget.result.createObjectStore(
9       "example-store", { keyPath: "id", autoIncrement: true });
10    };
11  };
```

---

---

**Listing 7** idb: öffnen eines Object-Stores

---

```
1 function initDB(){
2   return idb.open('example-db', 1, function(upgradeDb) {
3     upgradeDb.createObjectStore('example-store', { keyPath: 'id'});
4   });
5 }
```

---

### 4.1.5 Node.js, Express.js, MongoDB

Obwohl die entwickelte CRDT Implementierung mit jeder Datenbank kompatibel ist, die Daten schreiben, lesen und suchen kann, wird zu demonstrativen Zwecken doch ein Node.js Server aufgesetzt, der mit einer MongoDB Datenbank verbunden ist. Node.js ist eine JavaScript Laufzeitumgebung, welche gewöhnlich zum Betreiben von Webservern genutzt wird. Für das Bereitstellen der einzelnen Route wird, wie üblich, Express.js

genutzt. Die Datenbank muss, wie in Sektion 3.3 erläutert, nur eine Tabelle für das Speichern von Operationen verwalten. Als Beispiel wird hier eine MongoDB Datenbank mit Mongoose als Object Data Modeling Bibliothek verwendet. Die Wahl fiel auf MongoDB, weil es eine populäre Datenbank ist. Funktionale Kriterien spielten bei der Wahl der Datenbank keine Rolle.

#### 4.1.6 Sonstige Bibliotheken und Ressourcen

**CUID** Die Bibliothek CUID wird zum Generieren der IDs von Rezepten, Zutaten und Operationen genutzt. Die IDs, welche als Primärschlüssel in den Datenbanken dienen, müssen einzigartig sein. Für diesen Zweck ist CUID in der Lage, kurze Strings zu erzeugen, welche mit Client-Fingerabdruck, Timestamp, Counter und einer zufälligen Komponente aus mehreren kollisionsresistenten Elementen bestehen.

**Roboto Font und Material Icons** Als Ergänzung zu den verwendeten vue-material Komponenten werden die Schrift Roboto und die SVG-Icons Material Icons von Google genutzt. Diese Assets sind komplett kosmetisch, sie passen schlicht gut zu den verwendeten Komponenten.

**cors** Das node.js Paket cors wird genutzt, um Cross-Origin Resource Sharing (CORS) zwischen Client und Server zu ermöglichen.

#### 4.1.7 Ordnerstruktur

Abbildung 12 zeigt die Ordnerstruktur des Clients. Neben den für Webanwendungen üblichen Bestandteilen Template (index.html) und Stylesheet (app.css) ist die JavaScript Komponente des Prototypen in drei Dateien aufgeteilt. Sinn dieser Aufteilung ist es, die implementierten Funktionalitäten thematisch zu ordnen, einen praktischen Nutzen hat sie nicht. In app.js befinden sich Daten und Funktionalitäten, welche das Frontend betreffen. Die Datei crdt.js umfasst die Schnittstellen und Logik des implementierten CRDTs. In database.js befinden sich die Methoden, welche zum Zugriff auf die IndexedDB Datenbank umgesetzt wurden.

Der implementierte Service-Worker befindet sich in sw.js. Darüber hinaus enthält der Client noch die Bibliothek idb.js. Die weiteren genutzten externen Ressourcen Vue.js,

vue-material, CUID, Material Icons und die Roboto-Schrift werden über für prototypische Entwicklung übliche Content Delivery Networks in der index.html geladen.

```
crdt-app
├── index.html
├── app.js
├── crdt.js
├── database.js
├── app.css
├── sw.js
├── ext
└── idb.js
```

Abbildung 12: Ordnerstruktur des Clients

## 4.2 Nutzeroberfläche

Die Nutzung von Vue.js ermöglicht die Erzeugung sogenannter Vue Instanzen. Diese Instanzen dienen dem Template als eine Art interaktives Viewmodel (vgl. Sektion 4.1.2). Im Prototypen wird eine Instanz zur Darstellung der gesamten Applikation mit dem Namen “app” erzeugt. Weitere Komponenten werden aus der Bibliothek vue-material genutzt.

Listing 8 zeigt, wie die Verwendung von Vue.js eine Verbindung von Anzeige und Applikationsdaten ermöglicht. Mit der Direktive “v-for” wird für jedes Rezept, welches sich in der visibleRecipes Eigenschaft der “app” Instanz befindet, ein `<md-card>` Block erzeugt. Bei `<md-card>` handelt es sich um eine vue-material Komponente. Innerhalb dieser Komponente kann nun auf die Eigenschaften des verbundenen Rezeptes zugegriffen werden. In diesem Fall wird beispielsweise der Name des jeweiligen Rezeptes als Titel der Komponente gesetzt.

Eine weitere Funktionalität, die Vue.js zur Verfügung stellt, ist “v-on:click”. Hierbei handelt es sich um einen Klick-Event-Listener, welcher direkt in ein HTML-Element geschrieben werden kann. Der Wert der Direktive definiert, welche Funktion ausgeführt wird, wenn auf das Element geklickt wurde. In Listing 8 ist zu sehen, wie so das Bearbeiten des Rezeptnamens (vgl. Zeile 7), das Löschen des Rezeptes (vgl. Zeile 11) und

die Verlinkung zu den Zutaten des Rezeptes (vgl. Zeile 16) gesteuert werden. An jede der hier über “v-on:click” aufgerufenen Methoden wird das Rezept, mit welchem die Komponente verbunden ist, als Parameter übergeben.

---

**Listing 8** HTML-Code der Rezepteliste

---

```
1 <md-card v-for="recipe in visibleRecipes" :key="recipe._id" :  
  data-id="recipe._id">  
2   <md-card-header>  
3     <div class="nowrap">  
4       <div class="md-title">  
5         {{ recipe.name }}  
6       </div>  
7       <md-button class="md-icon-button md-dense md-primary  
          md-raised" v-on:click="onClickEditRecipeTitle(recipe)">  
8         <md-icon>edit</md-icon>  
9       </md-button>  
10    </div>  
11    <md-button class="md-icon-button md-dense md-warn md-raised"  
      v-on:click="onClickDeleteRecipe(recipe)">  
12      <md-icon>delete</md-icon>  
13    </md-button>  
14  </md-card-header>  
15  <md-card-actions>  
16    <md-button class="md-primary md-raised" v-on:click="  
      onClickRecipe(recipe)">  
17      Ingredients  
18      <md-icon>chevron_right</md-icon>  
19    </md-button>  
20  </md-card-actions>  
21 </md-card>
```

---

Neben der Auflistung an Rezepten verfügt die Hauptansicht der Applikation noch über ein Dialogfeld zum Anlegen der Rezepte und einen Button zum Synchronisieren der Applikationsdaten mit dem Server (vgl. Abbildung 13). In der Zutatenübersicht eines Rezeptes (vgl. Abbildung 14) lassen sich Zutaten anlegen, Namen und Mengen bearbeiten und Zutaten löschen.

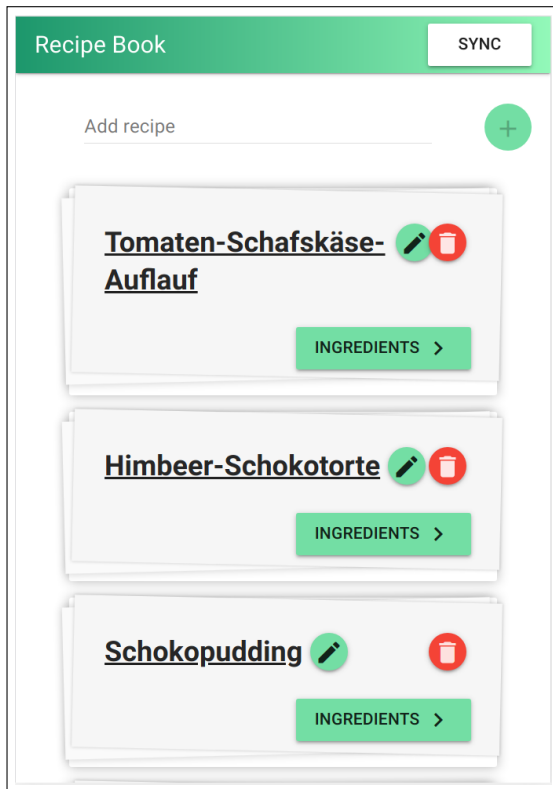


Abbildung 13: Hauptbildschirm (eigene Darstellung)

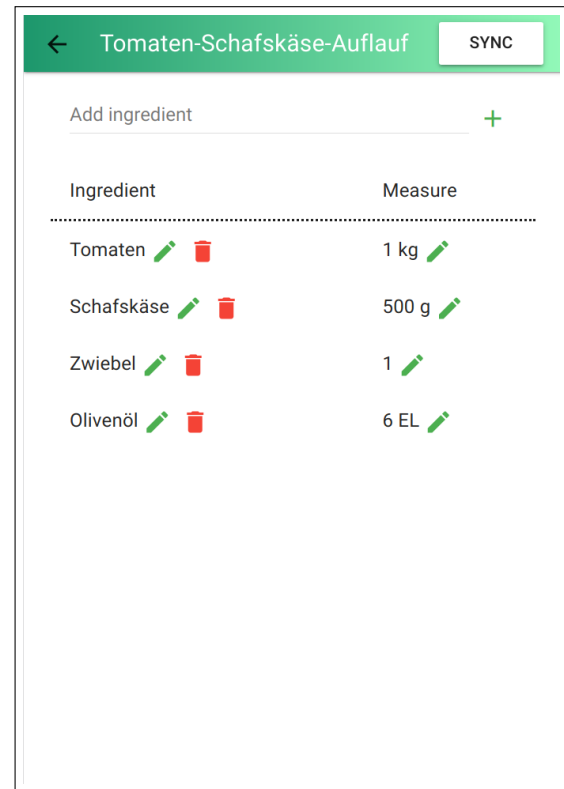


Abbildung 14: Übersicht Zutaten (eigene Darstellung)

## 4.3 Service Worker

Der Service Worker ermöglicht, dass der Prototyp nach dem ersten Laden der Applikation auch offline nutzbar ist. Hierzu werden alle Dateien, aus welchen der Client besteht, bei der Installation des Service Workers gecached (vgl. Listing 9). Die zu speichernden Ressourcen werden dabei als Parameter angegeben (vgl. Listing 10). Die Installation des Service Workers findet immer dann statt, wenn die Website geöffnet wird und noch kein Service Worker vorhanden ist.

---

**Listing 9** Installation des Service Workers

---

```
1 self.addEventListener("install", function (event) {
2   event.waitUntil(
3     caches.open(cachename).then(function (cache) {
4       console.log("cache opened");
5       return cache.addAll(urlstocache);
6     }),
7   );
8 });
```

---

---

**Listing 10** Parameter zur Installation des Service Workers

---

```
1 var cachename = 'recipebook-v0.1';
2 var urlstocache = [
3   'index.html',
4   'app.js',
5   'crdt.js',
6   'database.js',
7   [...]
8 ];
```

---

Das Bereitstellen der Dateien aus dem Cache erfolgt über das Abfangen von Netzwerk-Requests. Das *fetch* Event, für welches der in Listing 11 dargestellte Event Listener registriert wird, feuert jedes Mal, wenn eine Ressource im Anwendungsbereich<sup>5</sup> des Service Workers aus dem Netzwerk abgerufen werden soll. Dies bezieht sich sowohl auf Dokumente innerhalb des Anwendungsbereiches, wie die bei der Initialisierung gecachten Dateien, als auch auf *fetch* Requests an externe Ressourcen.

Wenn der Service Worker einen Request zum Synchronisieren abfängt, wird der Request nie vom Cache bedient, sondern immer an das Netzwerk weitergeleitet, wie im if-Block von Listing 11 zu erkennen ist. Eine Synchronisierung muss schließlich stets mit neuen Daten erfolgen, ein erneutes Synchronisieren mit alten Daten hätte keine Auswirkung. Wenn keine Netzwerkverbindung besteht, funktioniert das Synchronisieren also auch nicht.

Um das Laden der Applikation so schnell wie möglich zu gestalten, ist der Service Worker für alle anderen Requests so konfiguriert, dass er eine “Cache dann Netzwerk” Strategie

---

<sup>5</sup>Im Fall des Prototypen umfasst der Anwendungsbereich die gesamte Applikation



(vgl. Sektion 2.4) verfolgt. Im `else`-Block wird überprüft, ob sich die zum Request passende Response bereits im Cache befindet. Nur wenn dies nicht der Fall ist, wird der Request über das Netzwerk bearbeitet.

---

**Listing 11** Service Worker: Abfangen von Netzwerk-Requests

---

```
1 self.addEventListener("fetch", function (event) {
2   console.log(event.request.url);
3   if (event.request.url == "https://crdt-app-server.herokuapp.com
    /api/sync") {
4     event.respondWith(fetch(event.request));
5   } else {
6     event.respondWith(
7       caches.match(event.request).then(function (response) {
8         return response || fetch(event.request);
9       }),
10    );
11  }
12 });
```

---

Im Prototypen ist dieses Verhalten wie folgt umgesetzt: Beim Aufrufen der Webseite werden alle lokal gespeicherten Applikationen aus der Datenbank geladen (vgl. Listing 12) und in zwei lokalen Variablen, *recipes* und *ingredients*, der *app* Instanz gespeichert (vgl. Listing 13).

---

**Listing 12** loadUIData Methode

---

```
1 loadUIData: function (dataName) {
2   return new Promise ((resolve, reject) => {
3     return getAllFromStore(dataName).then(loadedObjects => {
4       this[dataName] = loadedObjects.reverse();
5       resolve('Updated ' + dataName + ' UI Components');
6     });
7   });
8 }
```

---

**Listing 13** Laden aller Rezepte und Zutaten

---

```
1 loadAllUIData: async function () {
2   return Promise.all([
3     this.loadUIData('recipes'),
4     this.loadUIData('ingredients')
5   ]).catch( e => {
6     console.log(e)
7   }).then((messages) => {
8     messages.forEach(message => {console.log(message)});
9   })
10 }
```

---

## 4.4 Lokales Speichern über die IndexedDB-API

Über die IndexedDB Schnittstelle müssen drei verschiedene Sätze an Daten lokal gespeichert werden: Rezepte, Zutaten und Operationen (vgl. Sektion 3.5). Die Datensätze werden jeweils in einem eigenen ObjectStore in der Datenbank gespeichert. Listing 14 zeigt die Variable “idbPromise”, welche bei allen Zugriffen auf die Datenbank genutzt wird. Wie in Sektion 4.1.4 erklärt, wird statt der üblichen IndexedDB-API die Promise-basierte Schnittstelle der idb.js Bibliothek genutzt. Beim jedem Zugriff auf die Datenbank wird geprüft, ob die ObjectStores bereits erstellt sind. Wenn dies nicht der Fall ist, werden sie angelegt. Nach erfolgreichem Öffnen der Datenbank gibt das Promise ein Objekt zurück, mit welchem Transaktionen an der Datenbank ausgeführt werden können.

**Listing 14** Öffnen der IndexedDB Datenbank

---

```
1 var idbPromise = idb.open("crdt-app", 2, function (upgradeDb) {
2   switch (upgradeDb.oldVersion) {
3     case 0:
4     case 1:
5       upgradeDb.createObjectStore("operations", { keyPath: "_id"
6         });
7       upgradeDb.createObjectStore("recipes", { keyPath: "_id" });
8       upgradeDb.createObjectStore("ingredients", { keyPath: "_id"
9         });
10  }
11 });
```

---

Somit können anschließend Zugriffe wie das Speichern von Objekten (vgl. Listing 15) oder das Abrufen aller Objekte eines ObjectStores (vgl. Listing 16) erfolgen. Der Ablauf folgt dabei stets dem gleichen Muster. Zu Beginn wird die Datenbank mit dem “idb-Promise” geöffnet, woraufhin eine Transaktion für den gewünschten ObjectStore erzeugt wird. Über die Transaktion ist schließlich der ObjectStore verfügbar, an welchem dann Daten ausgelesen oder geschrieben werden können.

---

**Listing 15** Speichern eines Objektes im ObjectStore

---

```
1 function saveObject(storeName, object) {  
2   idbPromise.then((db) => {  
3     let transaction = db.transaction(storeName, "readwrite");  
4     let store = transaction.objectStore(storeName);  
5     return store.put(object);  
6   });  
7 }
```

---

---

**Listing 16** Abrufen aller Objekte eines ObjectStores

---

```
1 function getAllFromStore(storeName) {  
2   return idbPromise.then((db) => {  
3     let transaction = db.transaction(storeName, "readonly");  
4     let store = transaction.objectStore(storeName);  
5     return store.getAll();  
6   });  
7 }
```

---

## 4.5 Generieren der Operationen

Wie in Sektion 3.5.2 beschrieben, muss das CRDT über Methoden zum Generieren und Anwenden von Operationen verfügen. Welche Operationen generiert werden dürfen, ergibt sich aus den Anforderungen der Applikation. In der Konzeption (vgl. Tabelle 1) wurden diese Anforderungen für den Prototypen definiert. Sie umfassen das Anlegen, Löschen und Bearbeiten von Rezepten und Zutaten. Da sich die Operationen zum Anlegen und Bearbeiten nicht unterscheiden müssen (vgl. Sektion 3.5.4) und das Löschen eines Objekts über das Bearbeiten des *tombstone* Werts erfolgt, können alle drei Anforderungen über eine Methode erfüllt werden, die in Listing 17 aufgeführt ist.

**Listing 17** Funktion zum Generieren einer Operation

---

```
1 function generateUpdateOperation(store, Id, key, value) {
2   let targetStore = store;
3   let targetObject = Id;
4   let targetKey = key;
5   let targetValue = value;
6   let timestamp = Date.now();
7   console.log("Changed " + targetObject + "property " + targetKey
8     + "to " + targetValue + "in " + targetStore,);
9   return {
10     store: targetStore,
11     object: targetObject,
12     key: targetKey,
13     value: targetValue,
14     timestamp: timestamp,
15     _id: "op_" + cuid(),
16   };
17 }
```

---

Folgende Parameter müssen zum Generieren einer Operation übergeben werden:

**store:** Der ObjectStore, in dem sich das zu bearbeitende Objekt befindet. Im implementierten Prototypen gibt es die ObjectStores “recipes” und “ingredients”.

**Id:** Die ID des Objektes, auf welches die Operation angewendet werden soll. Beim Anlegen eines Objektes muss zuvor eine neue CUID generiert werden.

**key:** Der Schlüssel des Wertes, der bearbeitet werden soll, z.B. “name”.

**value:** Der neue Wert.

Neben dem Mapping der übergebenen Parameter erzeugt die Methode noch einen aktuellen Zeitstempel und eine eigene CUID zum Identifizieren der Operation. Der Zeitstempel wird über die JavaScript Methode `Date.now()` erstellt, welche die Anzahl der vergangenen Millisekunden seit dem 01.01.1970 zurückgibt. Die `Date.now()` Methode orientiert sich an der Systemuhr des Clients, zu welcher es keine Garantie gibt, dass sie korrekt eingestellt ist. Im Rahmen der Entwicklung einer Anwendung, welche über einen Prototypen hinaus geht, sollten andere Methoden zur Erzeugung eines Zeitstempels genutzt werden, wie zum Beispiel die Implementierung einer Logischen Uhr, oder einer Hybrid Logical Clock (HLC). Da das Vergleichen der Zeitstempel sich dabei jedoch nicht verändert, wurde aufgrund des begrenzten Umfangs und Zeitfensters bei der Implementierung

des Prototypen auf die Umsetzung einer solchen Uhr verzichtet.

## 4.6 Verarbeiten von Operationen

Unabhängig davon, ob Operationen selbst erzeugt werden oder die Applikation über das Synchronisieren erreichen, werden sie konstant mit der gleichen Methode, *processOperations*, verarbeitet. Es wäre zwar angemessen, davon auszugehen, dass eine gerade eben erstellte Operation immer neuer ist als andere lokale Operationen, und somit immer angewendet werden muss. Ein sofortiges Anwenden solcher Operationen würde jedoch ein Risiko darstellen, denn somit gäbe es zwei Methoden zum Anwenden von Operationen, welche laut CRDT Anforderung beide immer das gleiche Ergebnis liefern müssten. Wenn es nur eine universelle Funktion zum Anwenden von Operationen gibt, ist hingegen sichergestellt, dass die Operation auf allen Clients gleichermaßen angewendet wird. Dies ist eine wichtige Garantie, um SEC zwischen den Replikationen herzustellen.

Operationen, die angewendet werden sollen, werden als *eingehende Operationen* bezeichnet, die bestehenden gespeicherten Operationen als *lokale Operationen*.

Aufgrund der Länge der *processOperations* Methode wird diese in der folgenden Beschreibung in drei Schritte unterteilt. Der Vorgang beginnt mit dem Laden der lokalen Operationen. Mithilfe dieser Operationen wird anschließend gefiltert, welche der eingehenden Operationen anzuwenden sind. Schlussendlich werden die gefilterten Operationen angewendet.

### 4.6.1 Lokale Operationen laden

Der Erste Schritt der *processOperations* Funktion ist das Laden der lokalen Operationen. Dies ist notwendig, weil die eingehenden Operationen im weiteren Verlauf der Methode mit den lokalen Operationen verglichen werden müssen. Wie Listing 18 zeigt, erfolgt das Laden der lokalen Operationen mit der im vorherigen Abschnitt (vgl. 4.4) beschriebenen Funktion *getAllFromStore*.

**Listing 18** Verarbeiten der Operationen, Teil 1

---

```
1 function processOperations(newOperations) {  
2   if (newOperations.length > 0) {  
3     return getAllFromStore("operations")  
4       .then((localOperations) => {  
5         //console.log(localOperations)  
6         return mapOperations(localOperations, newOperations);  
7       })  
8     [...]
```

---

Sobald die lokalen Operationen geladen sind, wird mit der Methode *mapOperations* (vgl. Listing 19) noch eine Vorkehrung für das Filtern im nächsten Schritt getroffen. *mapOperations* gibt ein Promise zurück, welches beim auflösen ein *Map* Objekt zurückgibt. Eine *Map* wird verwendet, da sie das auch Objekte als Keys erlaubt, in diesem Fall eignet sich dies perfekt um zwei Operationen gegenüberzustellen. Ziel ist es hier, zu jeder eingehenden Operation eine lokale Operation zu finden, deren Zielwert übereinstimmt. Zielwert bedeutet hier die Kombination aus ObjectStore, Objekt und Schlüssel.

Die Methode beginnt mit der Erzeugung eines leeren *Map* Objektes und der Sortierung der lokalen Operationen. Die Sortierung erfolgt nach den Zeitstempeln, wobei die aktuellsten Operationen, also die mit dem höchsten Zeitstempel, am Anfang des Arrays platziert werden.

Anschließend wird für jede der eingehenden Operationen eine passende lokale Operation gesucht. Eine Operation ist “passend”, wenn sie den exakt gleichen Zielwert wie die eingehende Operation bearbeitet. Durch das vorherige Sortieren der lokalen Operationen ist sichergestellt, dass die erste gefundene passende Operation stets die neueste ist. Nachdem die erste passende Operation gefunden wurde, ist also kein weiteres Suchen notwendig.

Daraufhin werden die eingehende Operation und die passende, lokale Operation zu der zuvor initialisierten *Map* hinzugefügt. Die eingehende Operation wird als Schlüssel gesetzt, die lokale Operation als Wert. Wenn keine passende lokale Operation gefunden wird, ist der Wert *undefined*. Nachdem jede eingehende Operation entweder ein lokales Gegenstück oder *undefined* zugeordnet bekommen hat, wird die nun gefüllte Map zurückgegeben.

**Listing 19** mapOperations Methode

---

```
1 function mapOperations(localOperations, newOperations) {
2   return new Promise((resolve, reject) => {
3     let mappedOperations = new Map();
4     let sortedLocalOperations = localOperations.sort((a, b) => {
5       return b.timestamp - a.timestamp;
6     });
7     newOperations.forEach((newOperation) => {
8       let mostRecentMatchingLocalOperation =
9         sortedLocalOperations.find(
10          (localOperation) =>
11            localOperation.store === newOperation.store &&
12            localOperation.object === newOperation.object &&
13            localOperation.key === newOperation.key,
14          );
15       mappedOperations.set(newOperation,
16         mostRecentMatchingLocalOperation);
17     });
18     if (mappedOperations.size > 0) {
19       console.log("Mapped new operations to local operations");
20       resolve(mappedOperations);
21     }
22   });
23 }
```

---

#### 4.6.2 Operationen filtern

Mit der zurückgegebenen Map von eingehenden zu lokalen Operationen kann das Filtern beginnen. In *processOperations* werden die *mappedOperations* nun an die Funktion *filterOperationsToApply* weitergegeben (vgl. Listing 20).

**Listing 20** Verarbeiten der Operationen, Teil 2

---

```
1 [...]
```

```
2 .then((mappedOperations) => {
3   return filterOperationsToApply(mappedOperations);
4 })
5 [...]
```

---

Ziel des Filterns ist es, zu ermitteln, ob die eingehenden Operationen angewendet werden dürfen oder nicht. Die Regeln, nach welchen entschieden wird, welche Operationen angewendet werden und welche nicht, werden aus der Konzeption übernommen (vgl.

Abbildung 8). Das vorherige Zuordnen der lokalen zu den eingehenden Operationen erleichtert diesen Vorgang, da somit nicht während des Filterns nach den passenden lokalen Operationen gesucht werden muss. Technisch wäre es zwar durchaus möglich, das Zuordnen von lokalen zu eingehenden Operationen während des Filterns durchzuführen, die Konsequenz wäre jedoch verschachtelter, schwer lesbarer Code.

Listing 21 zeigt den Code für das Filtern der Operationen, auf Abbildung 15 ist der Vorgang noch einmal bildlich dargestellt. Das Filtern beginnt mit dem Initialisieren zweier Arrays, eines zum Halten der Operationen für Rezepte, ein weiteres für Operationen, die sich auf Zutaten beziehen. Anschließend läuft eine for-Schleife über jeden Schlüssel, also jede eingehende Operation, der *mappedOperations* Map. In jeder Iteration der for-Schleife wird über die Zeitstempel geprüft, ob die eingehende Operation aktueller ist als die passende lokale Operation. Ist dies der Fall, wird die eingehende Operation zu dem jeweils passenden Array hinzugefügt. Dies passiert ebenso, wenn keine passende lokale Operation vorhanden ist und der Wert stattdessen *undefined* ist. Wenn ein Wert in der Map *undefined* ist, bedeutet es schließlich, dass das Objekt noch nicht existiert und angelegt werden muss. Diese Operationen müssen also immer angewendet werden.

---

**Listing 21** Filtern der Operationen, Teil 1

---

```
1 function filterOperationsToApply(mappedOperations) {
2   return new Promise((resolve, reject) => {
3     let recipeOperationsToApply = [];
4     let ingredientOperationsToApply = [];
5     Array.from(mappedOperations.keys()).forEach((newOperation) =>
6       {
7         let localOperation = mappedOperations.get(newOperation);
8         if (
9           !localOperation ||
10          newOperation.timestamp > localOperation.timestamp
11        ) {
12          if (newOperation.store == "recipes") {
13            recipeOperationsToApply.push(newOperation);
14          } else ingredientOperationsToApply.push(newOperation);
15        }
16      }
17    )
18  }
19 }
```

---



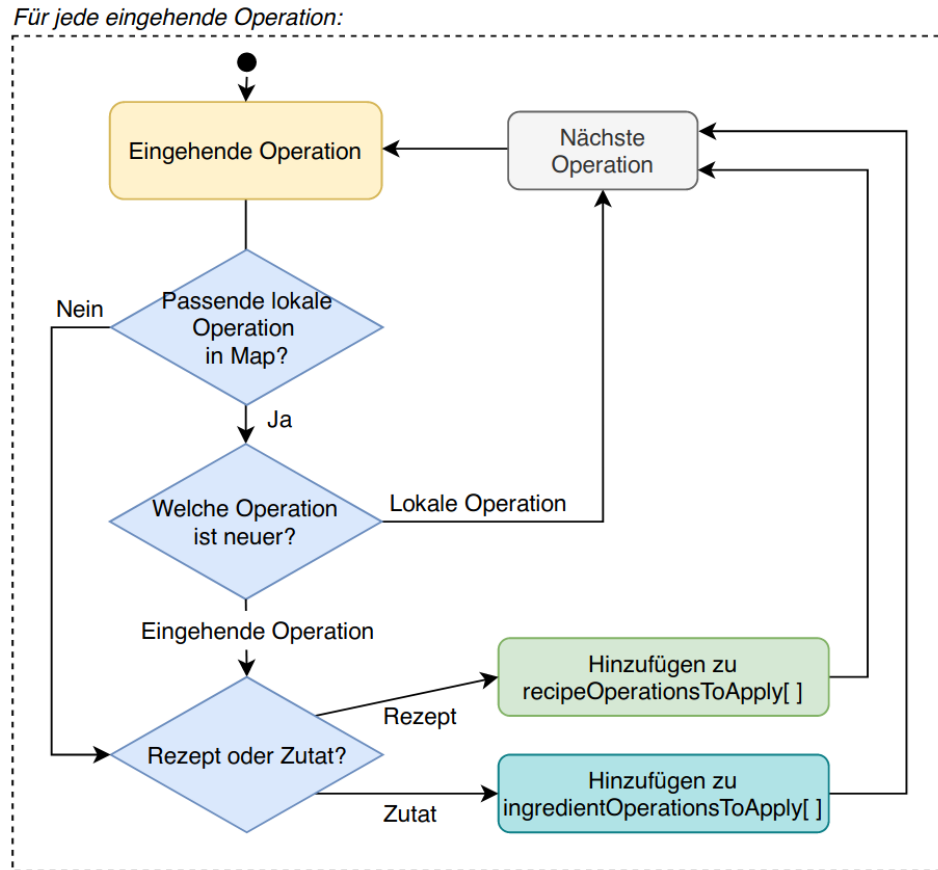


Abbildung 15: Filtern der Operationen (eigene Darstellung)

Nachdem alle eingehenden Operationen die for-Schleife durchlaufen haben, werden die beiden Arrays *recipeOperationsToApply* und *ingredientOperationsToApply* in der Map *operationsToApply* gespeichert (vgl. Listing 22). Schlüssel der in der Map gespeicherten Arrays ist jeweils der Name des ObjectStores. Dies geschieht, damit alle Operationen sich weiterhin in einem Objekt befinden und somit das Promise, welches *filterOperationsToApply* zurückgibt, einen einzelnen Rückgabewert hat. Anschließend kann so die Promise-chain<sup>6</sup> zum Verarbeiten der Operationen nahtlos fortgeführt werden, ohne Kompromisse zum Nutzen mehrerer Rückgabewerte treffen zu müssen.

<sup>6</sup>Aneinanderreihung mehrerer Promises über das wiederholte Aufrufen der *.then* Methode des jeweils vorherigen Promises

**Listing 22** Filtern der Operationen, Teil 2

```

1 [...]
```

---

```

2 let operationsToApply = new Map([
3   ["recipes", recipeOperationsToApply],
4   ["ingredients", ingredientOperationsToApply],
5 ]);
6 if (
7   operationsToApply.get("recipes").length > 0 ||
8   operationsToApply.get("ingredients").length > 0
9 ) {
10  resolve(operationsToApply);
11 [...]
```

---

**4.6.3 Operationen anwenden**

Sobald das Filtern der Operationen abgeschlossen ist, kann mit dem Anwenden der Operationen begonnen werden. Die Map, welche von *filterOperationsToApply* zurückgegeben wurde, enthält nun Operationen, die definitiv angewendet werden müssen, zugeordnet zum jeweiligen ObjectStore. Diese Operationen werden im Folgenden als *valide Operationen* bezeichnet. Als Beispiel, wie diese Struktur in der Anwendung aussieht, zeigt Abbildung 16 eine *validOperationsMap* mit vier validen Operationen für Rezepte und sechs validen Operationen für Zutaten.

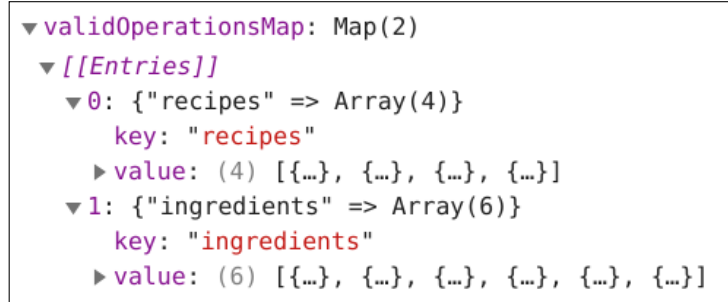


Abbildung 16: *validOperationsMap* im Chrome-Debugger (eigene Darstellung)

Listing 23 zeigt die Fortsetzung der *processOperations* Methode durch Verschachtelung

zweier *Promise.all*<sup>7</sup> Methoden. Was auf den ersten Blick komplex aussieht, ist in der Praxis relativ einfach.

---

**Listing 23** Verarbeiten der Operationen, Teil 3

---

```
1 [...]
2 .then((validOperationsMap) => {
3   return Promise.all(
4     Array.from(validOperationsMap.keys()).map((storeName) => {
5       return Promise.all(
6         validOperationsMap.get(storeName).map((validOperation) =>
7           {
8             return updateObject(validOperation, storeName);
9           }
10    )
11  )
12 })
```

---

Die erste *Promise.all* Methode bekommt für jeden Schlüssel in der *validOperationsMap* eine weitere *Promise.all* Methode als Input. Diese weiteren *Promise.all* Methoden bekommen für jede Operation in dem entsprechenden *ObjectStore* ein *updateObject* Promise als Input. Das erste *Promise.all* sorgt also dafür, dass jeder *ObjectStore*, der aktualisiert werden muss, aktualisiert wird. Die weiteren *Promise.all* Methoden, jeweils eine pro *ObjectStore*, stellen sicher, dass für jede Operation die *updateObject* Methode aufgerufen wird.

---

**Listing 24** Aktualisieren eines Objekts

---

```
1 function updateObject(operation, storeName) {
2   return getSingleObjectFromStore(storeName, operation.object)
3     .then((objectToUpdate) => {
4       return modifyObject(objectToUpdate, operation);
5     })
6     .then((modifiedObject) => {
7       return saveObject(storeName, modifiedObject);
8     });
9 }
```

---

In *updateObject* (vgl. 24) wird das Objekt, auf das die Operation angewendet wird, zuerst aus der Datenbank geladen. Dies ist nötig, weil es nicht möglich ist, einzelne

---

<sup>7</sup>Die *Promise.all* Methode nimmt mehrere Promises als Input und gibt ein einzelnes Promise zurück, welches zu einem Array der Ergebnisse aller Input-Promises auflöst. Das zurückgegebene Promise wird nur aufgelöst, nachdem alle Input-Promises erfolgreich aufgelöst wurden. Wird eines der Input-Promises abgelehnt, ist auch sofort das Rückgabe-Promise abgelehnt.

Werte eines Objektes über die IndexedDB-Schnittstelle zu ändern, ohne die anderen Werte des Objektes dabei zu verlieren. Deshalb ist es üblich, erst das komplette Objekt zu laden, dieses anschließend zu modifizieren, um schlussendlich die modifizierte Version in der Datenbank zu speichern und die alte Version zu überschreiben. So geschieht es auch in der *updateObject* Methode, welche nach dem Laden des Objektes die Funktion *modifyObject* (vgl. Listing 25) aufruft.

---

**Listing 25** *modifyObject* Methode

---

```
1 function modifyObject(object, operation) {  
2   return new Promise(function (resolve, reject) {  
3     if (object == undefined) {  
4       object = {  
5         _id: operation.object,  
6       };  
7     }  
8     var modifiedObject = object;  
9     modifiedObject[operation.key] = operation.value;  
10    saveObject("operations", operation);  
11    resolve(modifiedObject);  
12  });  
13 }
```

---

Es kann vorkommen, dass das Objekt, welches modifiziert werden soll, noch nicht existiert (vgl. Abschnitt 3.5.2). Ist dies der Fall, gibt das Laden des Objektes im vorherigen Schritt *undefined* zurück. In dieser Situation muss das Objekt in der *modifyObject* Methode erst erstellt werden, bevor die Operation angewendet werden kann. Diese Anforderung ist auch in der Konzeption festgehalten (vgl. Abbildung 8).

Anschließend wird die Operation angewendet, indem der Wert des in der Operation definierten Schlüssels mit dem neuen Wert überschrieben wird. Die Operation wird nun über die *saveObject* Funktion (vgl. Sektion 4.4) in der Datenbank der lokalen Operationen gespeichert.

Zurück in der *updateObject* Methode steht nur noch das Speichern des modifizierten Objektes an. Dies erfolgt über die gleiche Methode, mit welcher zuvor schon die Operation gespeichert wurde.

## 4.7 Synchronisieren

Zum Synchronisieren werden Operationen zwischen Client und Server über eine API (siehe Sektion 3.3) ausgetauscht. In diesem Abschnitt wird beschrieben, wie diese Funktionalität auf Client und Server implementiert ist.

Das Synchronisieren beginnt auf Seite des Clients mit dem Abrufen aller lokalen Operationen (vgl. Listing 26). Sind die Operationen erfolgreich abgerufen, werden sie über den `/sync` Endpunkt der API an den Server geschickt.

---

**Listing 26** Synchronisation Client

---

```
1 function sync() {
2   return getAllFromStore("operations")
3     .then((operations) => {
4     return fetch("https://crdt-app-server.herokuapp.com/api/
5       sync", {
6       method: "POST",
7       body: JSON.stringify(operations),
8       headers: {
9         "Content-Type": "application/json",
10      },
11    });
12  })
13  .then((res) => {
14    return res.json();
15  })
16  .then((data) => {
17    return processOperations(data);
18  })
19  .catch((e) => console.log(e));
20 }
```

---

Auf dem Server werden die Operationen nach den gleichen Kriterien verarbeitet wie die Operationen des Clients (vgl. Abbildung 8) mit dem Unterschied, dass der Server die Operationen nur speichern und nicht anwenden muss (vgl. Abschnitt 3.6.1). Über die MongoDB Funktion *insertMany* können alle Operationen auf einmal in die Datenbank geschrieben werden (vgl. Listing 27). Wenn eine Operation bereits in der Datenbank existiert, wird sie mit *insertMany* nicht erneut gespeichert. Zwar meldet MongoDB bei solchen Fällen einen Fehler, doch mit der Option *ordered: false* können diese ignoriert werden.

Nach dem Speichern der eingehenden Operationen werden die lokalen Operationen des Servers mit der MongoDB Funktion *find* geladen und anschließend als HTTP-Response an den Client geschickt. Auf diesem werden die eingehenden Operationen nun mit der *processOperations* Methode (siehe Abschnitt 4.6) verarbeitet.

---

**Listing 27** Synchronisation Server

---

```
1 exports.processOperations = (req, res) => {
2   const operations = req.body;
3   [...]
4   .then(() => {
5     return Operation.insertMany(operations, { ordered: false });
6   })
7   .catch((e) => {
8     console.log(e);
9   })
10  .then(() => {
11    return Operation.find({});
12  })
13  .then((operations) => {
14    return res.send(operations);
15  })
16  [...]
```

---

## 4.8 Löschen überflüssiger Operationen

Durch das ständige Hinzufügen von Operationen droht die Kommunikation zwischen Client und Server auf Dauer immer langsamer zu werden. Um dies zu verhindern, sind sowohl im Client als auch im Server Maßnahmen zur Reduzierung der Menge an Operationen implementiert.

Kommt es beim Verarbeiten der Operationen vor, dass zwei Operationen dieselbe Kombination aus ObjectStore, Objekt, Schlüssel und Wert haben, setzt sich immer die aktuellere der beiden Operationen durch. In der Implementierung des Prototypen wird sich die ältere Operation nie gegen die aktuelle durchsetzen, was bedeutet, dass die ältere Operation irrelevant ist. Deshalb wird die alte Operation diesem Fall gelöscht.

Auf dem Client findet das Löschen alter Operationen während des Filterns der Operationen statt (vgl. Abschnitt 4.6.2). In diesem Schritt werden die lokalen Operationen schließlich ohnehin mit den eingehenden Operationen verglichen.

**Listing 28** Löschen überflüssiger Operationen auf dem Client

---

```
1 [...]  
2   if (  
3     !!recentMatchingOperation &&  
4     mappedOperation.timestamp > recentMatchingOperation.timestamp  
5   ) {  
6     deleteOperation(recentMatchingOperation._id);  
7   }  
8 [...]
```

---

Um zu verhindern, dass überflüssige Operationen auf dem Server gespeichert bleiben, wird für jede eingehende Operation nach einer älteren lokalen Operation gesucht, die den gleichen Wert bearbeitet (vgl. Listing 29). Wird eine solche Operation gefunden, wird sie aus der Datenbank gelöscht. Anschließend wird noch überprüft, ob sich die gerade gelöschte Operation auch in den eingehenden Operationen befindet. Ist dies der Fall, wird sie auch dort gelöscht, damit sie nicht direkt wieder in die Datenbank geschrieben wird.

---

**Listing 29** Löschen überflüssiger Operationen auf dem Server

---

```
1 const query = {  
2   _id: { $ne: op._id },  
3   store: op.store,  
4   object: op.object,  
5   key: op.key,  
6   timestamp: { $lt: op.timestamp },  
7 };  
8 return await Operation.findOneAndDelete(query, function (err, doc  
9   ) {  
10    if (err) {  
11      console.log(err);  
12    }  
13    if (doc) {  
14      console.log("Deleted operation ", doc, "from database");  
15      const index = operations.findIndex((o) => o._id === doc._id);  
16      if (operations[index] !== undefined) {  
17        console.log("Deleting ", operations[index], "from incoming  
18          operations");  
19        operations.splice(index, 1);  
20      }  
21    }  
22  });
```

---

# 5 Evaluation

In diesem Abschnitt wird die Umsetzung der funktionalen und nicht-funktionalen Anforderungen reflektiert und auf Funktionsfähigkeit und Vollständigkeit überprüft.

## 5.1 Anlegen, Bearbeiten und Löschen von Rezepten und Zutaten

Das Anlegen, Bearbeiten und Löschen von Rezepten und Zutaten funktioniert über das implementierte CRDT. Somit können diese Aktionen von der Applikation vorgenommen werden, ohne dabei direkt die Anwendungsdaten zu verändern. Stattdessen nutzt die Applikation die Schnittstelle des CRDTs zum Generieren von Operationen, welche anschließend mit der *processOperations* Methode angewendet werden können.

Die Anforderungen zum Bearbeiten und Anlegen der Objekte sind im Prototypen vollständig erfüllt, sollte dieser jedoch um komplexere Datenstrukturen erweitert werden, müsste auch das CRDT nachziehen. Da ein G-Set umgesetzt wurde, findet das Löschen eines Objekts nicht wirklich statt, stattdessen wird die Sichtbarkeit des Objektes über einen Indikator gesteuert. Somit wächst die Menge der gespeicherten Daten konstant. Auf die Funktionalität des Prototypen hat dies zwar keine Auswirkung, da die gespeicherten Datenmengen winzig sind. Für große, skalierbare Projekte sollte jedoch auf eine andere Methodik zum Löschen gesetzt werden.

## 5.2 Zugriff mit mehreren Endgeräten zugleich

Mehrere Nutzer/innen können zur gleichen Zeit auf die selbe Applikation zugreifen. Solange sie nicht den exakt gleichen Wert bearbeiten, sorgt die CRDT-basierte Synchronisierung dafür, dass niemand seine Arbeit verliert. Der gewählte Weg der LWW



Konfliktlösung ist ein Kompromiss, der für die Anwendung am meisten Sinn ergibt. Es sollte immer individuell entschieden werden, welche Art der Lösung für gleichzeitige Änderungen desselben Wertes am besten zu den Anforderungen der Applikation passt.

## 5.3 Offline Erreichbarkeit

Die Offline Erreichbarkeit wird durch die Implementierung eines Service-Workers ermöglicht. Die Applikation ist offline vollständig funktionsfähig, nur das Synchronisieren über den Server funktioniert natürlich nicht, wenn der Server nicht erreichbar ist. Der Service Worker konnte gezielt so konfiguriert werden, dass die HTTP-Requests an den Server ignoriert werden.

## 5.4 Lokales Speichern

Für das lokale Speichern wird IndexedDB genutzt. Da der direkte Weg über die IndexedDB-API unnötig komplex erscheint, nutzt der Prototyp die Bibliothek idb.js. Diese ermöglicht durch das Bereitstellen von Promises, einen sehr angenehmen Umgang mit der Schnittstelle. Somit können die Anwendungsdaten offline und online problemlos gespeichert und ausgelesen werden.

## 5.5 Synchronisierung von Daten und SEC

Die Synchronisierung wird durch den Austausch von Operationen und das anschließende Anwenden eben dieser umgesetzt. Da die Clients nicht direkt miteinander kommunizieren, erfolgt dieses Austauschen über einen Server, welcher für alle Clients erreichbar ist.

Während die Nutzung des API-Endpunktes zum Synchronisieren nur das Hin- und Herschicken von Operationen umfasst, sind die Methoden, welche die Synchronisierung ermöglichen, weitaus komplexer. Sie ermöglichen das Erzeugen und Verarbeiten der Operationen. Somit stellen die Schnittstelle zum CRDT dar, welches wiederum so konzipiert wurde, dass zwei Replikationen nach dem Synchronisieren tatsächlich konvergent sind.

Die Implementierung der Synchronisierung funktioniert für alle Änderungen an Daten, die Nutzer/innen vornehmen kann und ist somit in vollem Umfang erfüllt. Durch die eingebundenen Regeln bei der Anwendung von Operationen ist garantiert, dass sich zwei Clients, welche die gleichen Operationen angewendet haben, im gleichen Zustand befinden. Somit ist auch die Anforderung nach SEC unter den Clients erfüllt.

Ein interessanter Ansatz wäre, Synchronisation auch über andere Wege als den Server zu erlauben. Das Synchronisieren könnte schließlich auch von Client zu Client erfolgen, da hierfür lediglich das gegenseitige Austauschen der lokalen Operationen nötig ist.

## 5.6 Konfliktfreies Synchronisieren

Die Anforderung des konfliktfreien Synchronisierens ist nur teilweise erfüllt. Zwar werden zwei synchronisierte Clients immer den gleichen Zustand haben, dieser kann jedoch für beide Akteure fehlerhaft sein. Dies liegt an der Nutzung der *Date.now* Methode. Da sie die Systemzeit des jeweiligen Endgerätes zum Erstellen von Zeitstempeln nutzt, ist nicht garantiert, dass die neueste Operation sich durchsetzt. Somit ist die umgesetzte “LWW” Regel fehleranfällig, wenn auf den Endgeräten der Clients inkorrekte Systemzeiten eingestellt sind.

Diese Anfälligkeit sollte sich jedoch mit der Nutzung einer HLC beheben lassen. HLC orientiert sich neben der Zeit auch an der Reihenfolge der Änderungen in der Applikation und kann somit als verlässlicher Zeitstempel in verteilten Systemen dienen (Kulkarni, Demirbas, Madappa, Avva & Leone, 2014, S. 1).

## 5.7 Unabhängigkeit von Datenbank-Technologien

Im Entwickelten Prototypen muss keine der beiden Datenbanken, sowohl auf der Clientseite noch auf der Serverseite, Funktionalitäten für das Synchronisieren der Daten bereitstellen. Somit ist die Anforderung der Unabhängigkeit von Datenbank-Technologien erfüllt.

Die Serverseitige Datenbank lediglich als zentraler Speicher von Operationen. Jede Datenbank, die in der Lage ist, Tabellen oder Objekte auszulesen und zu schreiben, sollte

also qualifiziert sein, diese Aufgabe zu übernehmen. Auf der Clientseite gibt es für Webanwendungen derzeit keine Alternativen zur IndexedDB-API.

# 6 Fazit und Ausblick

## 6.1 Fazit

Mit der Umsetzung des Prototypen wurde gezeigt, dass der Einsatz von CRDTs eine valide Lösung für die Synchronisierung von Applikationsdaten in Offline-First Anwendungen ist.

Bei der Planung der Datenstruktur ist es wichtig, die Anforderungen, welche die Applikation erfüllen muss und somit auch die geplante Interaktion mit den Applikationsdaten zu kennen. Im Fall des Prototypen wird das CRDT aus einer Sammlung von Grow-Only Sets, welche wiederum LWW-Maps beinhalten, gebildet. Diese CRDTs sind in Komplexität und Funktionsumfang relativ simpel, eignen sich jedoch sehr gut für die flache Datenstruktur des Prototypen. Mit 204 Zeilen JavaScript Code, was zu einer Dateigröße von 6,9 kB in ausgeschriebener Fassung und 2,1 kB in minimierter Version führt, ist die Implementierung des CRDT Prototypen sehr schlank.

Herausforderungen zeigen sich bei der implementierten Datenstruktur noch in der Skalierbarkeit. In dieser Hinsicht bedenklich ist, dass die Elemente des G-Sets nicht gelöscht werden können und auch die Menge der Operationen, selbst mit regelmäßigem Löschen, kontinuierlich ansteigen dürfte. Auch bedarf der Prototyp einer Erweiterung um eine sichere Methode zum Erzeugen von Timestamps, da die umgesetzte Nutzung von JavaScripts *Date.now* nicht zuverlässig ist. Eine bekannte Lösung für dieses Problem wäre die Implementierung einer Hybrid-Logical-Clock (HLC).

Eine generelle Empfehlung zur Nutzung bestimmter CRDTs in Offline-First Webanwendungen lässt sich jedoch nicht aussprechen. Dies ist aber eher eine positive Erkenntnis als eine negative. Wenn beispielsweise im Prototypen beim Synchronisieren das Bearbeiten stets das Löschen von Rezepten und Zutaten überschreiben soll, hat man als Entwickler/in die Möglichkeit, das Verhalten des CRDTs anzupassen. Statt LWW-Maps

würde sich dann die Umsetzung von Update-Wins-Maps anbieten. Die große an Anzahl bereits erforschten, entwickelten und umgesetzten CRDTs bietet Webentwickler/innen eine gute Grundlage, passende Bausteine für die eigene maßgeschneiderte Datenstruktur auszuwählen. Dabei sollte beachtet werden, dass die Umsetzung von CRDTs nicht für jeden Anwendungsfall geeignet ist. Für komplexe Funktionalitäten kann es schwierig sein, eine CRDT-basierte Datenstruktur umzusetzen, welche den Anforderungen entspricht, ohne die Regeln der SEC zu verletzen. In solchen Fällen lohnt sich auch ein Blick auf Bibliotheken, welche eine Reihe vorgefertigter CRDTs beinhalten wie zum Beispiel `automerger`<sup>8</sup>, `yjs`<sup>9</sup> oder `js-delta-crdts`<sup>10</sup>.

Ein weiterer Vorteil, den Entwickler/innen bei der Umsetzung einer eigenen CRDT-basierten Datenstruktur haben, ist die Unabhängigkeit von Datenbank-Technologien. Eine Implementierung, wie sie im Prototypen erfolgt ist, kann problemlos mit dem lokalen Speichern der IndexedDB-API und einer beliebigen Server-Datenbank erfolgen. Im Gegensatz zur Verwendung spezieller Datenbanken oder BaaS Anbietern, welche Funktionen zum Synchronisieren von Daten bereitstellen, ist ein selbst implementiertes CRDT unabhängig von Funktionalitäten, die an bestimmte Datenbanken geknüpft sind. Die Lösung durch die Nutzung von CRDTs erfordert jedoch einen nicht unerheblichen Planungsaufwand und könnte bei komplexen Datenstrukturen schwierig sein. Entwickler müssen hier also abwägen, ob sie eine “out of the box” Lösung bevorzugen, welche sie an bestimmte Datenbanken bindet, oder mit CRDTs eine feinabstimbare Variante bevorzugen, die jedoch nicht ohne Aufwand umzusetzen ist.

Die Wahl, das CRDT operation-based statt state-based umzusetzen, ist jedoch in Frage zu stellen. Die initiale Motivation, so ließe sich die Nutzlast der HTTP-Requests und Responses verringern, hat sich nicht bewahrheitet. Zwar ist es möglich, immer nur einzelne Operationen an den Server zu schicken, dies wäre jedoch kontraproduktiv. Im Client-Server-Modell gibt es keine Garantie, dass der Server sämtliche Operationen erhält, wenn diese jeweils nur einmal geschickt werden. Dadurch wäre das Verpassen einzelner Operationen möglich und SEC wäre nicht mehr gewährleistet. Um dies zu vermeiden, müssen im Prototypen stets alle Operationen ausgetauscht werden. Somit emuliert das operation-based CRDT im Gegensatz zur initialen Motivation ein state-based CRDT. Aus allen lokalen Operationen eines Clients lässt sich schließlich der ganze Zustand der Anwendung zusammensetzen. Dies ist äquivalent zu dem Halbverband, aus welchem

---

<sup>8</sup><https://github.com/automerger/automerger>

<sup>9</sup><https://github.com/yjs/yjs>

<sup>10</sup><https://github.com/peer-base/js-delta-crdts>

ein state-based CRDT besteht. Im Prototypen bilden die Operationen die Elemente des Halbverbandes, wobei die obere Schranke jeweils aus den Operationen mit dem neuesten Datum für jeden Wert des Zustands besteht.

## 6.2 Ausblick

Während die Hauptanwendungsfelder von CRDTs mit Sicherheit weiterhin in den Themengebieten großer verteilter Systeme und kollaborativem Editieren bleiben werden, sind sie doch auch für den Gebrauch in Webanwendungen von Nutzen. Die Garantie, dass Offline- und Online-Daten automatisch synchronisieren, ist gerade im Bereich der Offline-First Anwendungen viel Wert, da diese sonst auf bestimmte Technologien zurückgreifen müssen.

Es bleibt abzuwarten wie sich Offline-First Applikationen in der Zukunft entwickeln. PWAs mitsamt Offlinefunktionalität sind zwar im Web allgegenwärtig und werden auch von Web-Giganten wie Twitter, Facebook und Instagram angeboten. Wahre Offline-First Apps jedoch, welche offline komplett nutzbar sind, haben den großen kommerziellen Durchbruch noch nicht erlebt. In Ländern und Gebieten hingegen, in denen eine schlechte Internetverbindung keine Ausnahme, sondern der Standard ist, wird der Bedarf nach Lösungen, welche auf die lokale Nutzung ausgelegt sind, stets vorhanden sein. In diesen Ländern sind Offline-First Anwendungen eine Chance für systemrelevante Bereiche, wie Gesundheitswesen und Landwirtschaft, die Probleme der schlechten Netzabdeckung zu überbrücken. Somit sollte auch der Einsatz von CRDTs in Offline-First Anwendungen weiter evaluiert und optimiert werden, selbst wenn diese sich nicht als nächster großer Trend entpuppen.

Webentwicklern und Webentwicklerinnen, die sich mit dem Thema auseinandersetzen wollen, mangelt es dazu nicht an Ressourcen. Durch die jahrzehntelange Forschung an verteilten Systemen und den umfangreichen Forschungsstand, der zu Entwicklung und Validierung von CRDTs besteht, steht genug Material zur Verfügung. Für die Zukunft gilt es, diese Erkenntnisse auf die Welt des modernen Webs zu übertragen. Beispiele wie automerge und yjs, zeigen, dass auch die Entwicklung spezieller CRDTs für das Web möglich und sinnvoll ist.

Für den Prototypen wäre es interessant, das entwickelte CRDT erneut nach dem Muster eines state-based CRDT umzusetzen. So könnte verglichen werden, wie beide Varianten

sich in der Umsetzung und Anwendung unterscheiden. Für den Einsatz von operation-based CRDTs wäre die Nutzung einer Herangehensweise spannend, welche dafür sorgt, dass nicht alle Operationen an den Server gesendet werden müssen. Eine Technologie, die dies verspricht, ist zum Beispiel das  $\delta$ -CRDT Framework (Almeida, Shoker & Baquero, 2016). Auch eine Kompromisslösung wäre vorstellbar, in welcher Operationen regelmäßig einzeln ausgetauscht werden, das Synchronisieren mit allen Operationen aber dennoch in größerem Abstand durchgeführt werden würde. Somit könnten verpasste Operationen korrigiert werden, und die Nutzlast des regelmäßigem Austausches wäre minimiert.

# Abbildungsverzeichnis

1	Caching-Strategie: “Erst Netzwerk, dann Cache” . . . . .	6
2	Caching-Strategie: “Erst Cache, dann Netzwerk” . . . . .	7
3	Caching-Strategie: “Nur Netzwerk” . . . . .	8
4	Caching-Strategie: “Nur Cache” . . . . .	8
5	Architektur des Prototypen . . . . .	20
6	Entwurf Nutzeroberfläche Hauptbildschirm . . . . .	22
7	Entwurf Nutzeroberfläche Zutaten . . . . .	22
8	Bearbeiten einer eingehenden Operation . . . . .	28
9	Fehlerfall beim Entfernen eines Rezeptes . . . . .	29
10	Korrektes Entfernen eines Rezeptes . . . . .	31
11	Synchronisieren über den Server . . . . .	33
12	Ordnerstruktur des Clients . . . . .	39
13	Hauptbildschirm . . . . .	41
14	Übersicht Zutaten . . . . .	41
15	Filtern der Operationen . . . . .	51
16	<i>validOperationsMap</i> im Chrome-Debugger . . . . .	52



# Listings

1	Beispiel Rezept verschachtelt . . . . .	24
2	Beispiel Rezept relational . . . . .	25
3	Beispiel Zutat relational . . . . .	25
4	Beispiel Operation . . . . .	27
5	Beispiel Entfernen-Operation . . . . .	30
6	IndexedDB: öffnen eines Object-Stores . . . . .	37
7	idb: öffnen eines Object-Stores . . . . .	37
8	HTML-Code der Rezepteliste . . . . .	40
9	Installation des Service Workers . . . . .	42
10	Parameter zur Installation des Service Workers . . . . .	42
11	Service Worker: Abfangen von Netzwerk-Requests . . . . .	43
12	loadUIData Methode . . . . .	43
13	Laden aller Rezepte und Zutaten . . . . .	44
14	Öffnen der IndexedDB Datenbank . . . . .	44
15	Speichern eines Objektes im ObjectStore . . . . .	45
16	Abrufen aller Objekte eines ObjectStores . . . . .	45
17	Funktion zum Generieren einer Operation . . . . .	46
18	Verarbeiten der Operationen, Teil 1 . . . . .	48
19	mapOperations Methode . . . . .	49
20	Verarbeiten der Operationen, Teil 2 . . . . .	49
21	Filtern der Operationen, Teil 1 . . . . .	50
22	Filtern der Operationen, Teil 2 . . . . .	52
23	Verarbeiten der Operationen, Teil 3 . . . . .	53
24	Aktualisieren eines Objekts . . . . .	53
25	modifyObject Methode . . . . .	54
26	Synchronisation Client . . . . .	55
27	Synchronisation Server . . . . .	56

28	Löschen überflüssiger Operationen auf dem Client . . . . .	57
29	Löschen überflüssiger Operationen auf dem Server . . . . .	57

# Literaturquellen

- Almeida, P. S., Shoker, A. & Baquero, C. (2016). Delta State Replicated Data Types. *CoRR*, *abs/1603.01529*. arXiv: 1603.01529
- Almeida, P. S., Shoker, A. & Baquero, C. (2015). Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems* (S. 62–76). Springer.
- Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*.
- Aslan, K., Molli, P., Skaf-Molli, H. & Weiss, S. (2011). C-Set : a Commutative Replicated Data Type for Semantic Stores. In *RED: Fourth International Workshop on REsource Discovery*. Heraklion, Greece.
- Ater, T. (2017). *Building Progressive Web Apps*. O'Reilly Media, Inc.
- Baquero, C., Almeida, P. S. & Shoker, A. (2017). Pure Operation-Based Replicated Data Types. eprint: 1710.04469
- Bernstein, P. A., Hadzilacos, V. & Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.
- Coulouris, G. F., Dollimore, J. & Kindberg, T. (2005). *Distributed systems: concepts and design*. pearson education.
- Kimak, S. & Ellman, J. (2015). The role of HTML5 IndexedDB, the past, present and future. doi:10.13140/RG.2.1.4528.5201
- Kleppmann, M. & Beresford, A. R. (2017). A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10), 2733–2746.
- Kulkarni, S. S., Demirbas, M., Madappa, D., Avva, B. & Leone, M. (2014). Logical physical clocks. In *International Conference on Principles of Distributed Systems* (S. 17–32). Springer.
- Martyanov, D. (2018). CRDTs in production. In *Proceedings of the 2018 Qcon*. San Francisco.
- Nédelec, B., Molli, P., Mostefaoui, A. & Desmontils, E. (2013). LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM symposium on Document engineering* (S. 37–46).

- Nicolaescu, P., Jahns, K., Derntl, M. & Klamma, R. (2015). Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *International Conference on Web Engineering* (S. 675–678). Springer.
- Oster, G., Urso, P., Molli, P. & Imine, A. (2006). Data Consistency for P2P Collaborative Editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work* (S. 259–268). CSCW '06. Banff, Alberta, Canada: Association for Computing Machinery. doi:10.1145/1180875.1180916
- Preguiça, N. (2018). Conflict-free Replicated Data Types: An Overview. *arXiv preprint arXiv:1806.10254*.
- Saito, Y. & Shapiro, M. (2005). Optimistic Replication. *ACM Comput. Surv.* 37(1), 42–81. doi:10.1145/1057977.1057980
- Shapiro, M., Preguiça, N., Baquero, C. & Zawirski, M. (2011). Conflict-Free Replicated Data Types. In X. Défago, F. Petit & V. Villain (Hrsg.), *Stabilization, Safety, and Security of Distributed Systems* (S. 386–400). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J. & Hauser, C. H. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5), 172–182.
- Van Steen, M. & Tanenbaum, A. S. (2016). A Brief Introduction to Distributed Systems. *Computing*, 98(10), 967–1009. doi:10.1007/s00607-016-0508-7
- Van Steen, M. & Tanenbaum, A. S. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall.
- Weiss, S., Urso, P. & Molli, P. (2009). Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE International Conference on Distributed Computing Systems* (S. 404–412). IEEE.

# Onlinequellen

- Erné, M. (2006). Algebraische Verbandstheorie. Zugriff 26. August 2020 unter <http://www2.iazd.uni-hannover.de/~erne/algverb/dateien/AlgVerbandstheorie.pdf>
- Feyerke, A. (2013). Designing Offline-First Web Apps. Zugriff 21. August 2020 unter <https://alistapart.com/article/offline-first/>
- Google Developers. (2020). Offline First. Zugriff 21. August 2020 unter [https://developer.chrome.com/apps/offline\\_apps](https://developer.chrome.com/apps/offline_apps)
- Google Developers. (2019). Progressive Web Apps Training. Zugriff 21. August 2020 unter <https://developers.google.com/web/ilt/pwa>
- Jaiswal, A. (2019). Understanding Service Workers and Caching Strategies. Zugriff 5. Juli 2020 unter <https://blog.bitsrc.io/understanding-service-workers-and-caching-strategies-a6c1e1cbde03>
- Long, J. (2019). CRDTs for Mortals. Zugriff 21. August 2020 unter <https://jlongster.com/s/dotjs-crdt-slides.pdf>
- Mishunov, D. (2016). True Lies Of Optimistic User Interfaces. Zugriff unter <https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interface>
- W3Techs. (2020). Usage statistics of JavaScript as client-side programming language on websites. Zugriff 13. August 2020 unter <http://w3techs.com/technologies/details/cp-javascript/>
- WHATWG. (2019). HTML Living Standard. Zugriff 5. Juli 2020 unter <https://html.spec.whatwg.org/multipage/workers.html#workers>

# Weitere Quellen

- da Silva Tavares, J. (2018). *Secure Abstractions for Trusted Cloud Computation* (Diss., NOVA University of Lisbon).
- Kemme, B. (2000). *Database replication for clusters of workstations*.

# Anhang zum Prototypen

## Wo finde ich den Prototypen?

Der Code des Prototypen befindet sich in den folgenden öffentlichen GitHub repositories:

Client: <https://github.com/llendo/crdt-app>

Server: <https://github.com/llendo/crdt-app-server>

Zusätzlich ist der Prototyp auch als gehostete Website erreichbar unter:

<https://thesis-recipebook.netlify.app/>

Der Server ist gehostet auf:

<https://crdt-app-server.herokuapp.com/>

Sowohl Client als auch Server sind in der gehosteten Version auf dem gleichen Stand wie die GitHub repositories.

## Wie setze ich den Prototypen lokal auf?

### Client

Der Client lässt sich einfach über das Aufrufen der `index.html` über einen lokalen Webserver starten. Wenn ein lokaler Server zum Synchronisieren genutzt werden soll, muss die URL in der Funktion `sync` in der Datei `crdt.js`, durch die des lokalen Servers ersetzt werden.

### Server

Zum lokalen Starten des Servers wird `node.js` benötigt (<https://nodejs.org/>).

Ist `node` installiert, muss im Ordner `crdt-app-server` einmal `npm i` über die Kommandozeile ausgeführt werden.

Anschließend kann der Server über den Befehl `node server.js` gestartet werden.

Damit die gehostete Version funktioniert, verbindet sich der Server mit einem Online-Datenbank-Cluster, auf dem sich die MongoDB Datenbank befindet. Soll eine lokale

MongoDB genutzt werden, muss in der Datei `db.config.js` der *url*: Connection String durch den Connection String einer lokalen MongoDB Datenbank ersetzt werden.



# **Erklärung über das eigenständige Erstellen der Arbeit**

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Grafiken, Skizzen, bildliche Darstellungen sowie auf Quellen aus dem Internet. Die Arbeit habe ich in gleicher oder ähnlicher Form auch auszugsweise noch nicht als Bestandteil einer Prüfungs- oder Studienleistung vorgelegt.

Ich versichere, dass die eingereichte elektronische Version der Arbeit vollständig mit der Druckversion übereinstimmt.

Lennart Ploog

Matrikelnr.: 354556

Bremen, den 28.08.2020