Platzhalter Titel

Lennart Ploog

18. August 2020

IS Medieninformatik
Fakultät 4
Hochschule Bremen

Inhaltsverzeichnis

Abkürzungsverzeichnis

CRDT Conflict-free replicated data type

SC Strong Consistency

SEC Strong Eventual Consistency

EC Eventual Consistency

CRTD Konfliktfeier Replizierter Datentyp

P2P Peer To Peer

HLC Hybrid Logical Clock

JSON JavaScript Object Notation

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

LWW Last-Writer-Wins

LWW-Register Last-Writer-Wins Register

G-Set Grow-Only Set

1 Abstract

TODO: Acronyms, Figures

2 Einleitung

2.1 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 eines anderen Nutzers, in der Zwischenzeit jedoch geändert, müssen beide Änderungen zusammengebracht, also synchronisiert werden. Der Prozess der Synchronisation ist oft aufwendig, denn zum Einen muss ermittelt werden, wo sich beide 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 Datenbanken-Technologien. Dazu gehören Datenbanken mit implementierter Synchronisation wie CloudDB oder auch Backend-as-a-Service Produkte wie Firebase oder IBM Cloudant, welche ebenfalls eine solche Funktionalität anbieten. Um zu Vermeiden, die Applikation mit suboptimalen Datenbanken-Technologien umsetzen zu müssen, verzichten viele Applikationen bei Konflikten auf eine Synchronisation und einer der beiden Nutzer verliert seine vollbrachte Arbeit.

2.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine Lösung zur Synchronisation von Daten in Offline-First Anwendungen mit Hilfe von konfliktfreien replizierten Datentypen (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.

2.3 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 im Abschnitt Verwandte Arbeiten erwähnten 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 zu 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 auch 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.

2.3.1 Prototyp

Platzhalter, genauer nach Fertigstellung des Prototypen Als Prototyp wird ein Online-Kochbuch mit folgenden Funktionen umgesetzt:

- Anlegen, Bearbeiten und Löschen von Rezepten mit Namen, Zutaten und Beschreibung
- Zugriff auf die Gleichen Rezepte von verschiedenen Clients
- "Liken" der Rezepte

3 Verwandte Arbeiten

CRDTs sind aus der Forschung an Datenstrukturen für kollaboratives Editieren entstanden. Shapiro u. a. (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 Library 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.

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

Der Anzahl an Quellen und Resourcen 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 mir keine Arbeiten über den konkreten Einsatz von CRDTs in Offline-First Applikationen bekannt.

Ziel dieser Arbeit ist es deshalb, die umfangreich erforschten Grundlagen zum Einsatz von CRDTs in einer Offline-First Applikation umzusetzen und zu ermitteln, welche besonderen Herausforderungen diese Umgebung aufweist.

4 Grundlagen

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

4.1 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. 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 dem Nutzer auch offline so viele Funktionalitäten wie möglich zur Verfugung stehen.

4.2 Service Worker

Ein Service Worker ist ein sogenannter Web Worker. Web Worker sind Skripte, die unabhängig von anderen Skripts, 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 Hypertext Transfer Protocol (HTTP)-Requests an den Server angefordert. Der Service Worker ist ein event-basiertes Skript, welches als Proxy zwischen

Client und Server agiert. Damit diese Tatsache 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 können gezielt entscheiden, welche Netzwerk-Requests auf welche Art und Weise verarbeitet werden sollen. Mithilfe dieser Funktionalität können Entwickler 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.

4.3 Caching

Es folgen einige grundlegende Caching-Strategien, mit Beispielen, für welche Art 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 Nutzer 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 Nutzer aber eine ältere Version zur Verfügung gestellt werden soll, wenn die Internetverbindung unterbrochen ist.

"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 als HTTP-Request an den Server weitergeleitet. Dies ist die bevorzugte Strategie für die meisten Requests in Offline-First Anwendungen.(Ater, 2017, Kapitel 05).

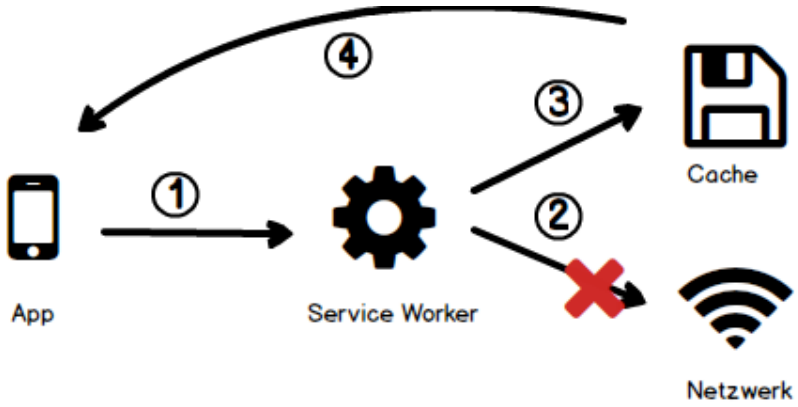

Abbildung 1: “Erst Netzwerk, dann Cache”

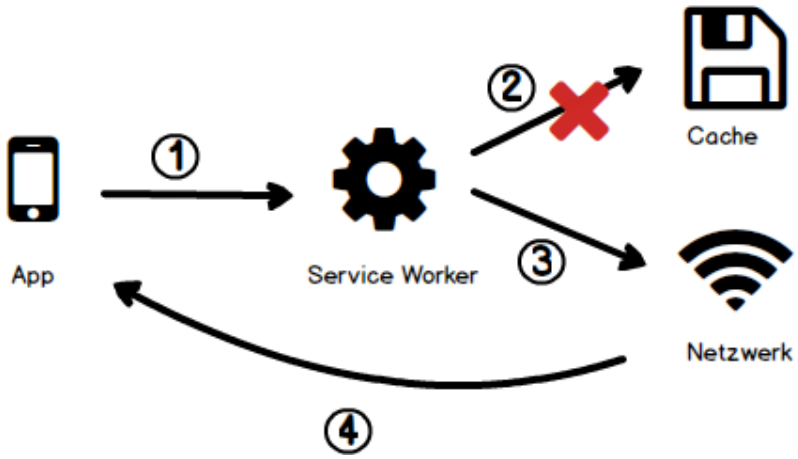

Abbildung 2: “Erst Cache, dann Netzwerk”

“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 Requests im Falle einer unterbrochenen Internetverbindung nachgeholt werden können, wenn der Nutzer wieder online ist.

“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,

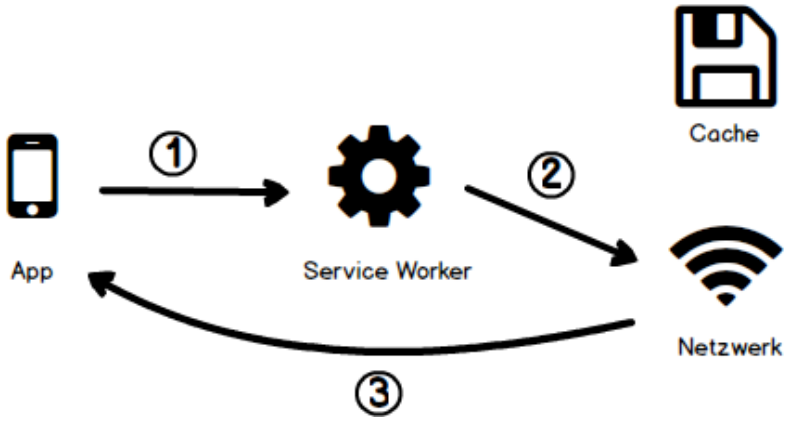

Abbildung 3: "Nur Netzwerk"

beispielsweise beim Installieren des Service Workers, mit gecached wurden.

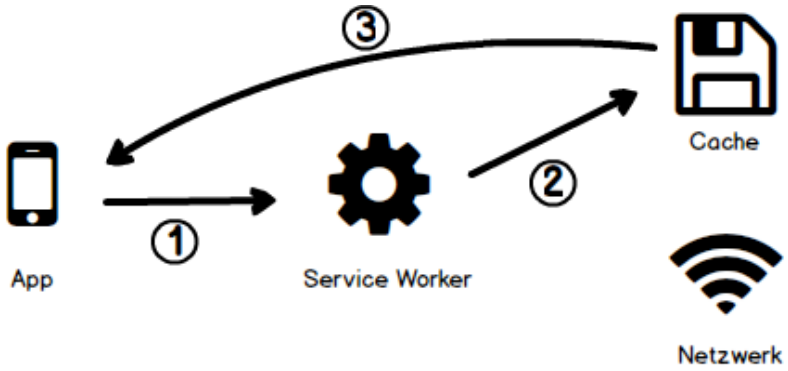

Abbildung 4: "Nur Cache"

4.4 Offline First Applikationen als verteilte Systeme

Steen und Andrew S. Tanenbaum (2016) beschreiben verteilte Systeme wie folgt: "Ein verteiltes System ist eine Sammlung von autonomen Rechenelementen, die den Benutzern als ein einziges kohärentes System erscheint."

Die folgende Liste zeigt die Charakteristika von verteilten Systemen, zusammengefasst nach Andrew S Tanenbaum und Van Steen (2007).

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 den Nutzer sind die Unterschiede zwischen den einzelnen Computern im System unersichtlich. Er nimmt die verteilten Computer als ein einzelnes System wahr.

Konsistente und einheitliche Interaktion Eine Konsequenz aus dem singulären Erscheinen ist, dass die Interaktion des Nutzers mit dem System immer gleich sein sollte, unabhängig davon, mit welcher Schnittstelle des Systems er tatsächlich interagiert.

Kontinuierliche Verfügbarkeit Das System soll dem Nutzer 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 Kernfunktionalität von Offline-First Applikationen ist die Offlinefunktionalität. Offline interagiert der Nutzer nur mit seinem Endgerät, online werden die Daten gleich an den Server geschickt. Diese Unterschiede sind für den Nutzer jedoch nicht von Belang.

Konsistente und einheitliche Interaktion Unabhängig davon, welches Endgerät der Nutzer verwendet, sollen ihm früher oder später die Änderungen aller im System aktiven Geräte angezeigt werden. Der Nutzer muss 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. Der Nutzer kann seine 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 2.1 beschriebene Synchronisation von Daten gibt es deshalb wenige beschriebene Lösungsansätze oder konkrete wissenschaftliche Arbeiten (siehe 3). 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.

4.5 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 und 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 4.4, 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, in denen Replikation angewendet wird.

4.5.1 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 und 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 und 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 (siehe 4.1), 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.

4.5.2 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 (siehe Sektion 4.4).

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. Nimmt der Nutzer eine Änderung vor, so wird diese auf seiner 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 und 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 Nutzern oder die Daten anderer Replikationen zu warten (Terry u. a., 1995).

Gerade in Offline-First Anwendungen ist die Konfliktbehandlung eine große Herausforderung, wie in Abschnitt 2.1 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.

4.5.3 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.

4.6 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 Wertes 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 (siehe **??**). 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 (siehe **??**), 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 und 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 Merge-Funktion ist anschließend in der Lage, beide Zustände zu einem zu kombinieren.

Operation-based Bei der Variante der operation-based CRDTs wird nicht der gesamte State der Objekte ausgetauscht. Stattdessen erfolgt der Austausch über einzelne Updates. Wird der State eines Objektes geändert, so wird die Operation, welche die Änderung hervorgerufen hat, gespeichert. Beim Mergen zweier Objekte werden diese Updates ausgetauscht und anschließend auf das jeweils andere Objekt

angewandt. Da die Operationen kommutativ sein müssen, ist die Reihenfolge der Updates nicht relevant.

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

4.6.1 LWW-Register

Ein Register ist ein Objekt, welches einen einzelnen Wert verwaltet. Dieser Wert kann jede vom System unterstützte Datenstruktur sein. Ein Last-Writer-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 neuste Änderung übernommen werden. Da sowohl Wert als auch Zeitstempel ausgetauscht werden, handelt es sich hier um ein state-based CRDT.

4.6.2 Grow-Only Set

Ein Set ist 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 löst dieses Problem durch das Auslassen der Entfernen-Operation. 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.

5 Konzeption

5.1 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 auf verschiedenen Wegen mit Daten interagieren.

Deshalb wird eine Applikation zum Verwalten von Rezepten entwickelt, welche es Nutzern ermöglicht, ein gemeinsames Rezeptebuch zu pflegen. Da Nutzer 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 begutachten.

5.2 Anforderungsanalyse

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

5.2.1 Funktionale Anforderungen

Zweck der funktionalen Anforderungen ist es, Verhalten und Funktionalität zu entwickelnden Systems zu beschreiben. Die formulierten Anforderungen sollen somit einen Überblick darauf geben, was das implementierte System erfüllen muss. Da das Ziel dieser Arbeit die Entwicklung einer Datenstruktur für eine Offline-First Applikation ist (siehe 2), liegt der Fokus der gesammelten Anforderungen eindeutig auf dem Zusammenspiel zwischen Offline-First und CRDTs.

Üblich in einer Anforderungsanalyse ist, die Anforderungen in "kann", "soll" und "muss" Kategorien einzuteilen. Da es sich hier um die Entwicklung eines Prototypen handelt, wird sich in diesem Teil exklusiv auf "muss" Anforderungen beschränkt. Hierbei handelt es sich um Anforderungen, welche essentiell für das System sind.

Anlegen von Rezepten Nutzer sollen Rezepte anlegen können. Ein Rezept hat einen Namen und eine Liste an Zutaten.

[FA 1] Anlegen von Zutaten Die Nutzer sollen in der Lage sein, Zutaten zu einem Rezept hinzuzufügen. Zutaten haben einen Namen und eine Mengenangabe, welche die Nutzer setzten können.

Bearbeiten und Löschen Den Nutzern soll es möglich sein, Rezepte und Zutaten zu bearbeiten und zu löschen.

[FA 2] Zugriff mit mehreren Endgeräten gleichzeitig Nutzer sollen in der Lage sein, mit mehreren Geräten gleichzeitig auf die gleiche Applikation zuzugreifen.

[FA 3] Offline-Erreichbarkeit Nutzer der Anwendung müssen in der Lage sein, die Anwendung offline zu nutzen. Der Funktionsumfang der Applikation soll dadurch nicht beeinflusst werden.

[FA 4] Lokales Speichern Die Anwendung muss über die Funktionalität verfügen, Änderungen von Nutzern lokal zu speichern, damit diese auch zur Verfügung stehen wenn keine Verbindung zum Netzwerk besteht.

Online Speichern Die Anwendungsdaten sollen nicht exklusiv lokal, sondern auch im Internet gespeichert werden können. Das Speichern der Daten erfolgt sinngemäß auf einem Server.

[FA 5] Synchronisierung von Daten Änderungen des Nutzers müssen mit dem Server synchronisiert werden können, wenn eine Verbindung zum Internet besteht. Der Nutzer sollte niemals seine offline umgesetzten Änderungen verlieren.

SEC Nachdem sich zwei Nutzer mit dem Stand des jeweils anderen Nutzers synchronisiert haben, müssen beide Applikationen den gleichen Stand haben.

[FA 6] Konfliktfreies Synchronisieren Das Synchronisieren der Daten soll ohne Konflikte erfolgen.

5.2.2 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.

[NFA 1] Anschauliche Umsetzung: Code Ein Ziel der Arbeit ist, zu ermitteln, wie CRDTs in modernen Webanwendungen umgesetzt werden können (siehe 2.1). Da sich moderne Offline-First Anwendungen leider nicht in 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.

[NFA 2] Anschauliche Umsetzung: Technologien Um das in NFA 1 formulierte Ziel der Anschaulichkeit weiter zu verfolgen, soll auch die Wahl der Technologien so erfolgen, dass diese soll die Ebene der Datenstruktur in Javascript umgesetzt werden. Auch wenn Entwicklern heutzutage zahlreiche Sprachen zur Verfügung stehen, welche zu Javascript kompilieren, bleibt Javascript aufgrund der hohen Verbreitung die für demonstrative Zwecke geeignetste Sprache.

[NFA 3] 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.

5.3 Architektur

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

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

Die Nutzung eines Service-Workers (siehe 4.2) ermöglicht das zwischenspeichern der gesamten Client-seitigen 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 des Nutzers Daten an den Server zu schicken.

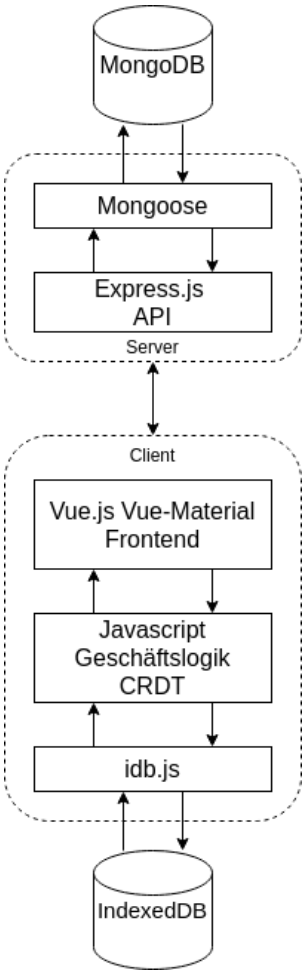

Abbildung 5: “Architektur des Prototypen”

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 library “(”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 eine einzelne “/sync” Route 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 server-seitige Speichern der Applikationen erfolgt auf einer MongoDB Datenbank. Ergänzend zu MongoDB wird noch die library Mongoose genutzt. Als Laufzeitumgebung zum Betreiben des Webservers dient Node.js.

5.4 Technologien

Es folgt eine kompakte Vorstellung der zur Umsetzung des Prototypen genutzten Technologien. Auch wird erläutert, warum die einzelnen Technologien verwendet werden.

5.4.1 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 Übersetzten lässt, ist eine Implementierung in der "Grundsprache" am besten dazu geeignet, die Anforderung eine anschaulichen und verständlichen Umsetzung zu erfüllen.

5.4.2 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, welches diese Interaktionen vereinfacht, an.

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.

Ein Feature 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 component-library vue-material genutzt, um Zugriff auf einige vorgefertigte Komponenten wie Buttons und Eingabefelder zu haben.

5.4.3 IndexedDB

IndexedDB ist eine Browser-API welche das clientseitige Speichern von Daten im Browser ermöglicht. Das bedeutet, dass über IndexedDB gespeicherte Daten 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 ObjectStores 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.

5.4.4 idb

Die Event- und Callback-basierte Schnittstelle, welche IndexedDB bietet, führt dazu, dass sehr komplexer Code geschrieben werden muss um simple Operationen an der Datenbank durchzuführen (Kimak und Ellman, 2015). Um dieses Problem zu vermeiden und guten, lesbaren Code zu schreiben wird bei der Umsetzung des Prototypen auf die library 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 5.1 zeigt das erzeugen eines Object-Stores über die gewöhnliche IndexedDB API, in Listing 5.2 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 5.1 IndexedDB: öffnen eines Object-Stores

```
function initDB() {
    var request = indexedDB.open("example-db", 1);
    request.onsuccess = function (evt) {
        db = request.result;
    };

    request.onupgradeneeded = function (evt) {
        var objectStore =
            evt.currentTarget.result.createObjectStore("example-
            store", { keyPath: "id", autoIncrement: true });
        }
    };
};
```

Listing 5.2 idb: öffnen eines Object-Stores

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

5.4.5 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 5.3 erläutert, nur eine Tabelle für das Speichern von Operationen verwalten. Als Beispiel wird hier eine MongoDB Datenbank mit Mongoose als Object Data Modeling library verwendet. Die Wahl fiel auf MongoDB

weil es eine populäre Datenbank ist, funktionale Kriterien spielten bei der Wahl der Datenbank keine Rolle.

5.4.6 Sonstiges

UUIDv4, google Roboto font, google material icons

5.5 Entwurf der Nutzeroberfläche

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

5.6 Datenstruktur

Diese Sektion beschreibt die Planung der Datenstruktur für den Prototypen. Erster Schritt dieser Planung ist es, zu ermittelt, 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 und die Datenstruktur schließlich auf die Applikation zugeschnitten sein soll 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.

Abbildung 6: "Hauptbildschirm"

Abbildung 7: "Übersicht Zutaten"

5.6.1 Grundlagen der Datenstruktur

Die in Sektion 5.2 formulierten Anforderungen formulieren, 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 5.3 zeigt die 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 5.3 Beispiel Rezept verschachtelt

```
var exampleRecipe = {
    _id: "re_ckdiw1bxw002411w66m7mol85s",
    name: "Tomatensalat",
    ingredients: [
        {
            name = "Tomaten",
            measure: "500g"
        },
        {
            name: "Zwiebeln",
            measure: "1"
        },
        {
            name: "Olivenöl",
            measure: "3 EL"
        },
        {
            name: "Essig",
            measure: "1 EL"
        }
    ]
}
```

Ein solches Modell ist nicht unüblich und gerade in JavaScript Anwendungen, weil Objekte in diesen oft verschachtelt sind, häufig praktisch. Beim Einsatz von CRDTs jedoch, lohnt es sich die Datenstruktur so flach wie möglich zu gestalten. Wir erinnern uns – Ein CRDT baut auf normalen Datenstrukturen auf. Wenn es die Möglichkeit gibt, die Komplexität der ausgehenden Datenstruktur zu reduzieren, bedeutet dass auch eine Reduzierung an Komplexität für die zu implementierenden CRDT-Funktionalitäten.

Eine Option, die Datenstruktur abzuflachen besteht darin, die in Listing 5.3 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 5.4 zeigt ein Rezept in der relationalen Modellierung, Listing 5.5 eine dazugehörige Zutat. Die Verbindung zwischen Rezepten und Zutaten wird nun über die

Eingenschaft “recipe” der Zutaten definiert. Hier wird die ID des Rezepts hinterlegt, welchem die Zutat zugehörig ist.

Listing 5.4 Beispiel Rezept relational

```
var exampleRecipe = {
    _id: "re_ckdiw1bxw002411w66m7mol85s",
    name: "Tomatensalat"
}
```

Listing 5.5 Beispiel Zutat relational

```
var exampleIngredient = {
    _id: "in_ckdpnyi0c00081w6j7q4nwbg6",
    name: "Tomaten",
    measure: "500g",
    recipe: "re_ckdiw1bxw002411w66m7mol85s"
}
```

5.6.2 Erweiterung zum CRDT

Währen sich bei konventionellen Datentypen nicht viel daran ändert, ob eine Zutat über “recipe.ingredient”, wie es in Listing 5.3 der Fall wäre, oder “ingredient”, siehe Listing 5.4 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 was 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 in Sektion 5.2 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	Datenstruktur Anforderung
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 Zutatensammlung.
Bearbeiten und Löschen	Bearbeitung der Eigenschaften Rezeptname, Zutatenname, Zutatenmenge. Entfernen eines Rezeptes aus der Rezeptesammlung. Entfernen einer Zutat aus der Zutatensammlung.

Die Umsetzung im Prototypen erfolgt durch ein "Operation-based" CRDT. Wie in Sektion 4.6 erläutert, werden alle Änderungen, welche an den Applikationsdaten erfolgen, einzeln als sogenannte Operationen abgespeichert. Wie Rezepte und Zutaten werden auch die Operationen über die IndexedDB API lokal im Browser gespeichert.

Listing 5.6 zeigt eine Operation, welche beim Umbenennen des in Listing 5.4 entworfenen Rezeptes 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 5.6 Beispiel Operation

```
var operation = {
    _id:"op_ckdpnyar800061w6jtko9ew8k",
    store:"recipe",
    object: "re_ckdiw1bxw002411w66m7mol85s",
    key: "name",
    value: "Tomaten-Paprika-Salat",
    timestamp:1597133338100,
}
```

5.6.3 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 von einander 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-Writer-Wins (LWW) Regeln behandelt. Abbildung 8 zeigt, wie entschieden wird, ob eine eingehende Operation angewendet wird. Zuerst wird überprüft, ob es für eine eingehende Operation, welche angewandt werden soll, bereits eine lokale Operationen gibt, welche den selben Wert betreffen. 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 angewandt und anschließend, wie vorgesehen, zu den lokalen Operationen hinzugefügt.

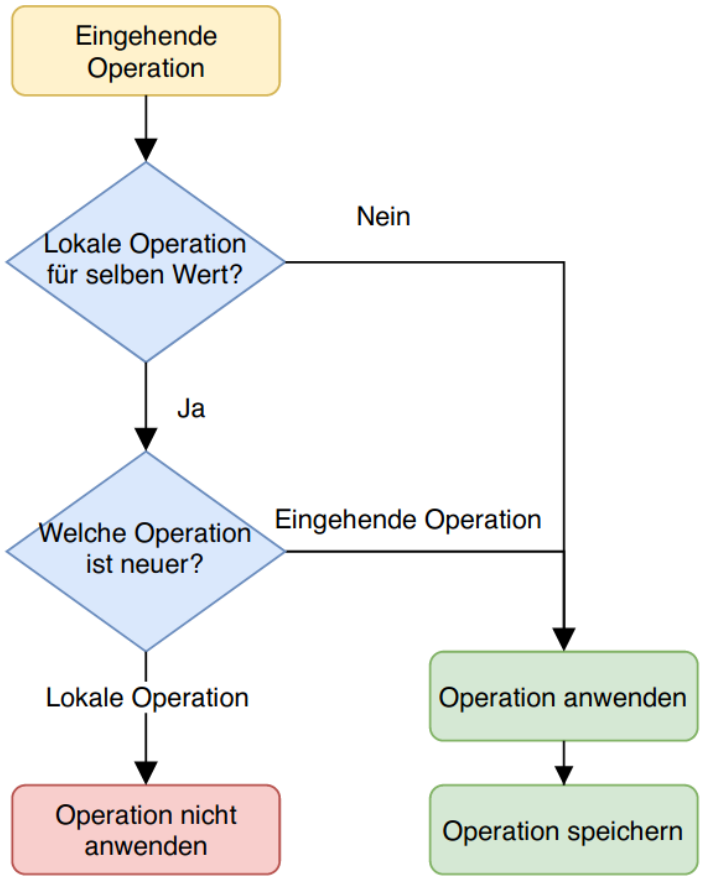

Abbildung 8: "Bearbeiten einer eingehenden Operation"

5.6.4 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 (siehe 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 Schnittstellen zur Datenstruktur ist (siehe 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 angewandt werden. Somit wäre erneut das Kommutativgesetz verletzt und die Applikationsdaten könnten sich nach dem Synchronisieren unterscheiden.

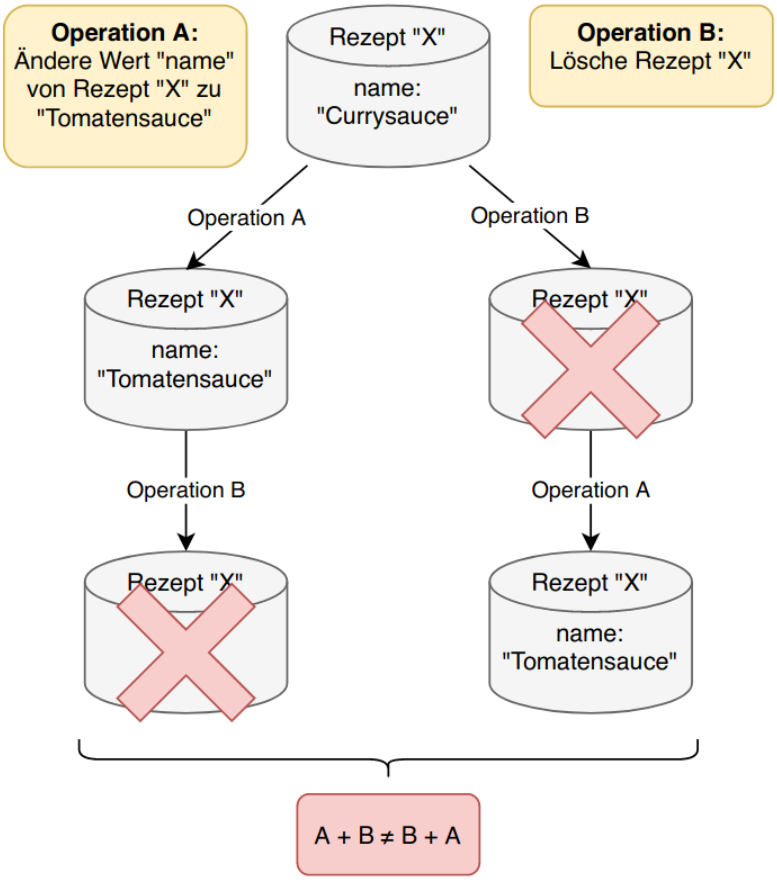

Abbildung 9: "Fehlerfall beim Entfernen eines Rezeptes"

Im Prototyp werden die Rezepte- und Zutatensammlungen deshalb als G-Set (siehe Sektion 4.6.2) implementiert. Dies bedeutet, dass 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”, Englisch für Grabstein, 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 5.7 Beispiel Entfernen-Operation

```
var operation = {
    _id:"op_ckdpnyar800051w6jocpy5sjm",
    store:"recipe",
    object: "re_ckdiw1bxw002411w66m7mol85s",
    key: "tombstone",
    value: "1",
    timestamp:159713335680,
}
```

Listing 5.7 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.

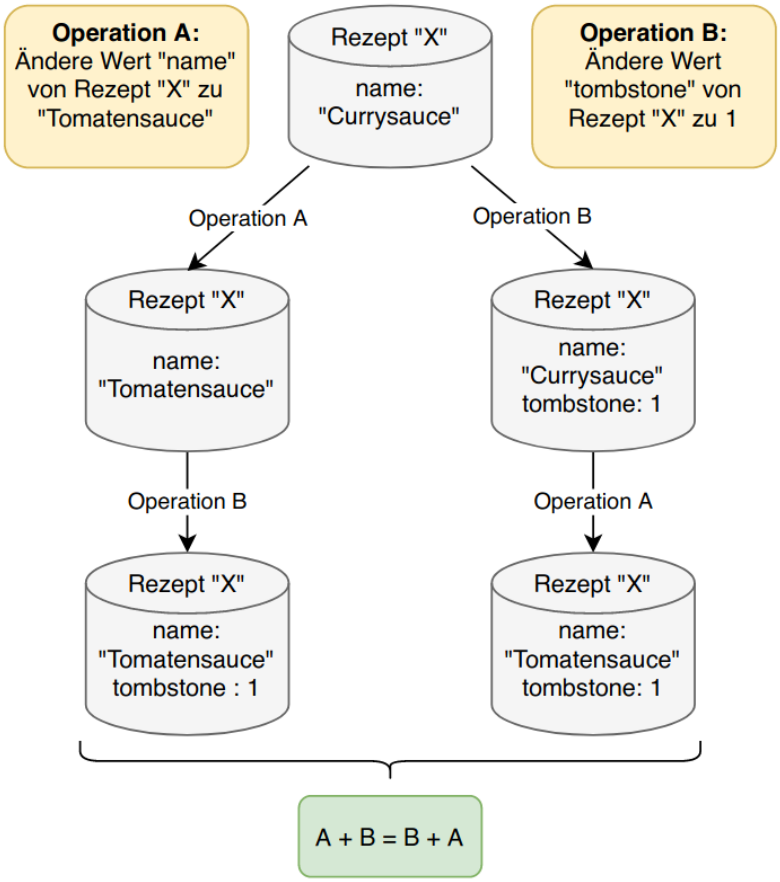

Abbildung 10: "Korrektes Entfernen eines Rezeptes"

5.7 Server

Diese Sektion beschreibt, wie der Server in der Architektur genutzt wird, um für das Verteilen von Operationen zu sorgen.

5.7.1 API

5.7.2 Datenbank

5.8 Schnittstellen

Literatur

[1] Marc Shapiro u. a. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Hrsg. von Xavier Défago, Franck Petit und Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 386–400. ISBN: 978-3-642-24550-3.

[2] M. Kleppmann und A. R. Beresford. "A Conflict-Free Replicated JSON Datatype". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), S. 2733–2746.

[3] Gérald Oster u. a. "Data Consistency for P2P Collaborative Editing". In: *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. CSCW '06. Banff, Alberta, Canada: Association for Computing Machinery, 2006, S. 259–268. ISBN: 1595932496. DOI: 10.1145/1180875.1180916. URL: https://doi.org/10.1145/1180875.1180916.

[4] Stéphane Weiss, Pascal Urso und Pascal Molli. "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, S. 404–412.

[5] Brice Nédelec u. a. "LSEQ: an adaptive structure for sequences in distributed collaborative editing". In: *Proceedings of the 2013 ACM symposium on Document engineering*. 2013, S. 37–46.

[6] WHATWG. *HTML Living Standard*. 2019. URL: https://html.spec.whatwg.org/multipage/workers.html#workers (besucht am 05.07.2020).

[7] Aayush Jaiswal. *Understanding Service Workers and Caching Strategies*. 2019. URL: https://blog.bitsrc.io/understanding-service-workers-and-caching-strategies-a6c1e1cbde03 (besucht am 05.07.2020).

[8] Tal Ater. *Building Progressive Web Apps*. O'Reilly Media, Inc., 2017.

[9] Maarten Steen und Andrew S. Tanenbaum. "A Brief Introduction to Distributed Systems". In: *Computing* 98.10 (Okt. 2016), S. 967–1009. ISSN: 0010-485X. DOI:

10.1007/s00607-016-0508-7. URL: https://doi.org/10.1007/s00607-016-0508-7.

[10] Andrew S Tanenbaum und Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007, S. 2–3.

[11] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. 1999. Kap. Vorwort.

[12] Yasushi Saito und Marc Shapiro. "Optimistic Replication". In: *ACM Comput. Surv.* 37.1 (März 2005), S. 42–81. ISSN: 0360-0300. DOI: 10.1145/1057977.1057980. URL: https://doi.org/10.1145/1057977.1057980.

[13] George F Coulouris, Jean Dollimore und Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[14] Bettina Kemme. "Database replication for clusters of workstations". ETH Zurich, 2000.

[15] Philip A. Bernstein, Vassos Hadzilacos und Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5. URL: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx.

[16] Douglas B Terry u. a. "Managing update conflicts in Bayou, a weakly connected replicated storage system". In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), S. 172–182.

[17] Nuno Preguiça. "Conflict-free Replicated Data Types: An Overview". In: *arXiv preprint arXiv:1806.10254* (2018).

[18] Dmitry Martyanov. "CRDTs in production". In: *Proceedings of the 2018 Qcon*. San Francisco, 2018.

[19] W3Techs. *Usage statistics of JavaScript as client-side programming language on websites*. 2020. URL: http://w3techs.com/technologies/details/cp-javascript/ (besucht am 13. 08. 2020).

[20] Stefan Kimak und Jeremy Ellman. "The role of HTML5 IndexedDB, the past, present and future". In: Dez. 2015. DOI: 10.13140/RG.2.1.4528.5201.