Platzhalter Titel

Lennart Ploog

24. 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

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

CORS Cross-Origin Resource Sharing

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 (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?

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 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 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 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.

Mit Woot (Oster u. a. 2006), Logoot (Weiss, Urso und Molli 2009) und LSEQ (Nédelec u. a. 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 mir keine wissenschaftlichen Arbeiten über den konkreten Einsatz von CRDTs in Offline-First Applikationen bekannt. Einen wichtigen Anhaltspunkt für die Nutzung in Offline-First Applikationen bietet der Vortrag "CRDTs for Mortals" (**Online\acp {CRDT}ForMortals**), in welchem über den Einsatz von CRDTs im Client-Server Modell gesprochen wird. In dem Vortrag geht es jedoch um die Entwicklung einer komplett lokalen Applikation.

Ziel dieser Arbeit ist es deshalb, die umfangreich erforschten Grundlagen zum Einsatz von CRDTs in einer Offline-First Webanwendung 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 (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 dem Nutzer auch offline so viele Funktionalitäten wie möglich zur Verfügung stehen (Feyerke, 2013).

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

In dieser Sektion werden einige grundlegende Caching-Strategien beschrieben, mit Beispielen, für welche Art von 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.

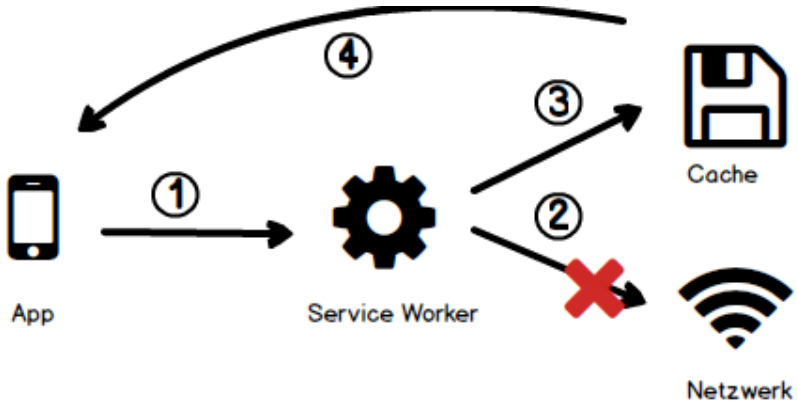

Abbildung 1: Caching-Strategie: “Erst Netzwerk, dann Cache”

“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).

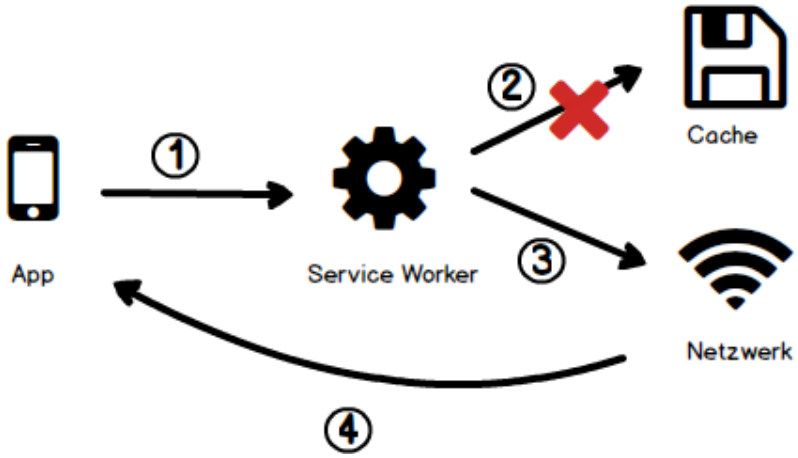

Abbildung 2: Caching-Strategie: “Erst Cache, dann Netzwerk”

“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.

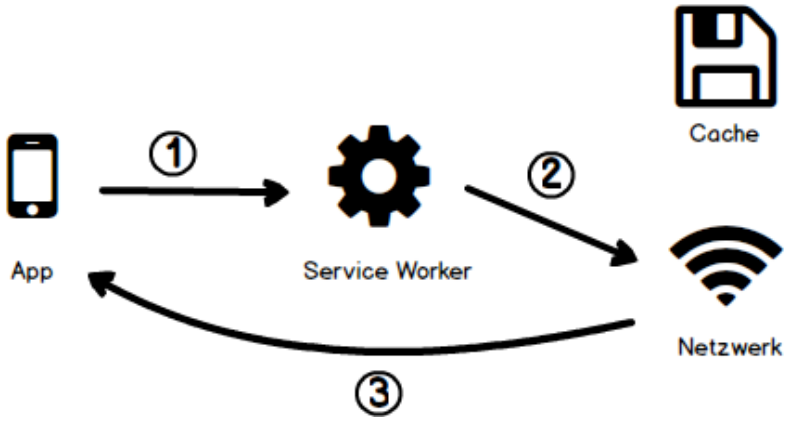

Abbildung 3: Caching-Strategie: "Nur Netzwerk"

"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.

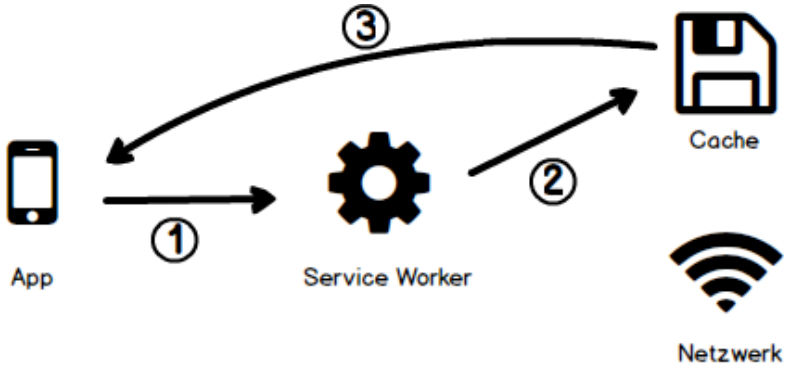

Abbildung 4: Caching-Strategie: "Nur Cache"

4.4 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 den Benutzern als ein einziges kohärentes System erscheint."

Die folgende Liste zeigt die Charakteristika von verteilten Systemen, zusammengefasst nach Van Steen und Tanenbaum (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 (Google Developers, 2020). 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 (vgl. Sektion 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 (vgl. 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 (vgl. 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 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 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 (vgl. Abschnitt **??**). 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 3), 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 Implementierung eines state-based CRDT 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[1] gebildet, dessen obere Schranke den aktuellen Wert des CRDTs darstellt. (Almeida, Shoker und 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 u. a., 2011, S. 9), was bedeutet, dass ein state-based CRDT ein operation-based CRDT emulieren kann und umgekehrt.

4.6.1 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-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.

[1] "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)

4.6.2 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 (Silva Tavares, 2018, S. 43).

4.6.3 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 und 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 u. a., 2011, S.7).

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 Abschnitt 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 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 (vgl. 2), 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 sollen Rezepte anlegen können. Ein Rezept hat einen Namen und eine Liste an Zutaten.

[A 02] 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.

[A 03] Bearbeiten und Löschen Den Nutzern soll es möglich sein, Rezepte und Zutaten zu bearbeiten und zu löschen.

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

[A 05] Offline-Erreichbarkeit Nutzer 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 Nutzern 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 des Nutzers müssen mit dem Server synchronisiert werden können, wenn eine Verbindung zum Internet besteht. Der Nutzer sollte niemals seine offline umgesetzten Änderungen verlieren.

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

[A 10] 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.

[A 11] Anschauliche Umsetzung: Code Ein Ziel der Arbeit ist, zu ermitteln, wie CRDTs in modernen Webanwendungen umgesetzt werden können (vgl. 2.1). 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.

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.

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 (vgl. 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.

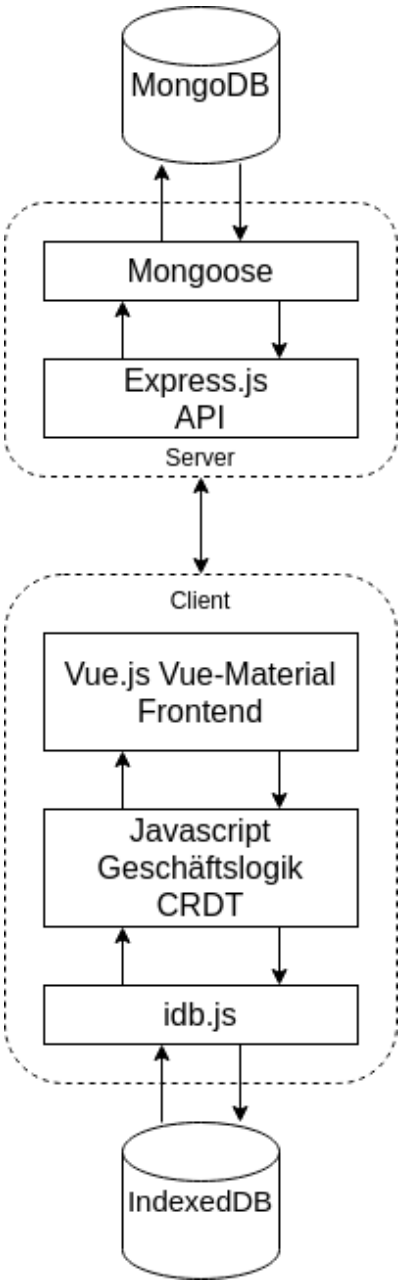

Abbildung 5: Architektur des Prototypen

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-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 server-seitige 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 6.1.

5.4 Entwurf der Nutzeroberfläche

Aus den funktionalen Anforderungen (Abschnitt 5.2.1) 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.

Abbildung 6: Entwurf Nutzeroberfläche Hauptbildschirm

Abbildung 7: Entwurf Nutzeroberfläche Zutaten

5.4.1 Optimistic UI

Wenn der User eine Änderung über die Nutzeroberfläche einleitet, bekommt er 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).

5.5 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.

5.5.1 Grundlagen der Datenstruktur

Die in Sektion 5.2 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 5.1 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 5.1 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 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 4.6 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 5.1 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 5.2 zeigt ein Rezept in der relationalen Modellierung, Listing 5.3 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 5.2 Beispiel Rezept relational

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

Listing 5.3 Beispiel Zutat relational

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

5.5.2 Erweiterung zum CRDT

Während sich bei konventionellen Datentypen nicht viel dadurch ändert, ob eine Zutat über "recipe.ingredient", wie es in Listing 5.1 der Fall wäre, oder "ingredient", vgl. Listing 5.2 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 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.

Die Umsetzung im Prototypen erfolgt durch ein "Operation-based" CRDT. Da beide möglichen Formen gleichwertig sind (vgl. Abschnitt 4.6) wurde diese Entscheidung getroffen weil "Operation-based" CRDTs unter Umständen eine geringere Nutzlast für

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 Zutatensammlung
Bearbeiten und Löschen	Bearbeitung der Eigenschaften Rezeptname, Zutatenname, Zutatenmenge Entfernen eines Rezeptes aus der Rezeptesammlung Entfernen einer Zutat aus der Zutatensammlung

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 4.6).

Wie in Sektion 4.6 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 5.4 zeigt eine Operation, welche beim Umbenennen des in Listing 5.2 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 5.4 Beispiel Operation

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

5.5.3 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-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 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 angewandt und anschließend, wie vorgesehen, zu den lokalen Operationen hinzugefügt. Dies verhindert auch das mehrfache Anwenden derselben Operation, wo-

durch das Verarbeiten eingehender Operationen idempotent ist. Dies ist eine wichtige Eigenschaft für das CRDT (vgl. Abschnitt 4.6).

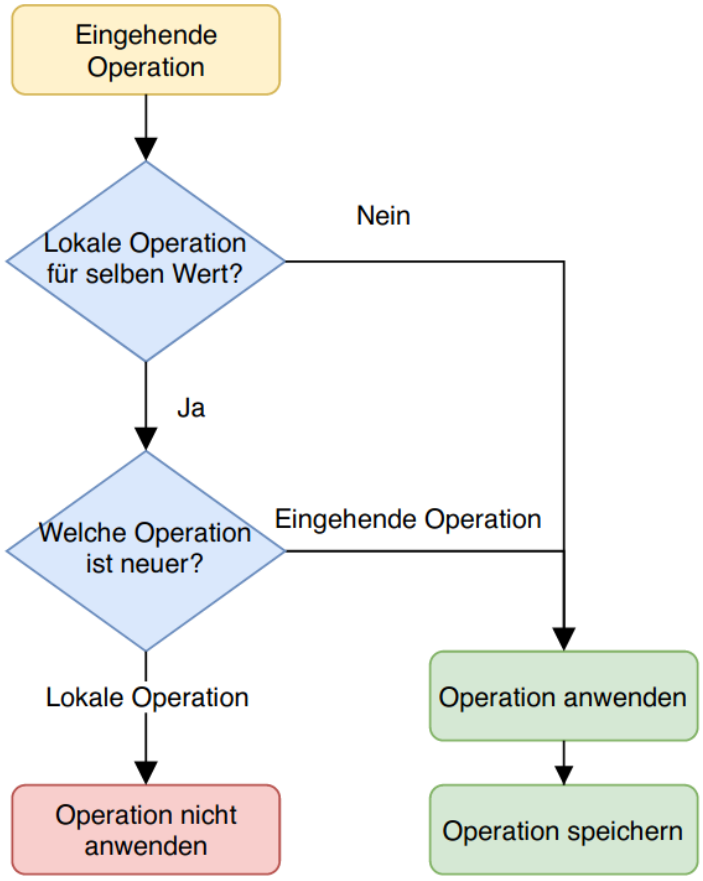

Abbildung 8: Bearbeiten einer eingehenden Operation

5.5.4 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.

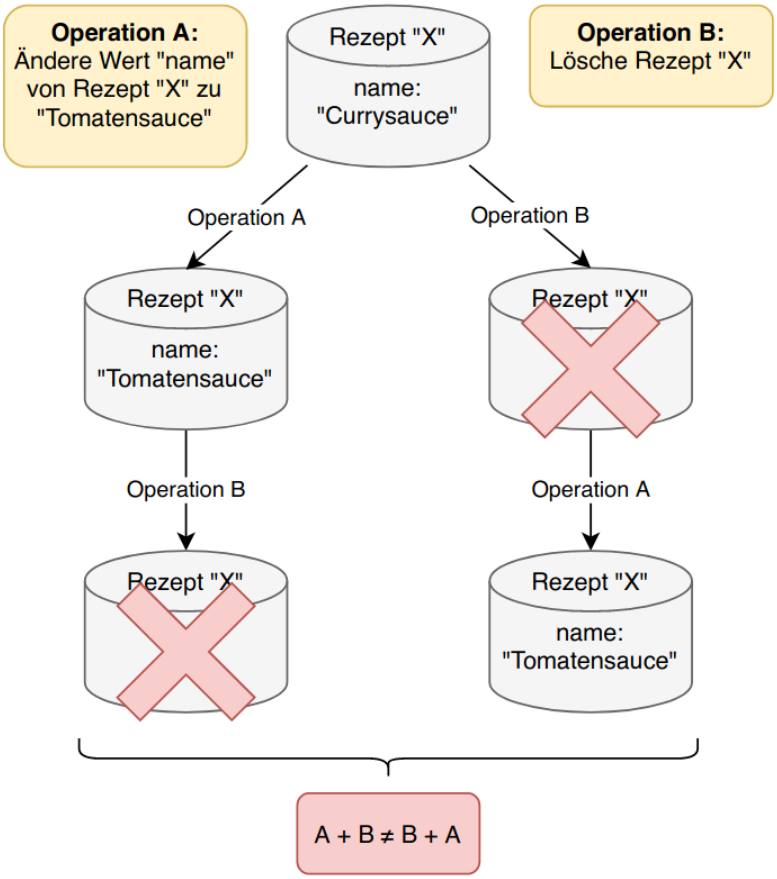

Abbildung 9: Fehlerfall beim Entfernen eines Rezeptes

Im Prototyp werden die Rezepte- und Zutatensammlungen deshalb als G-Set (vgl. Sektion 4.6.3) 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 5.5 Beispiel Entfernen-Operation

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

Listing 5.5 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.

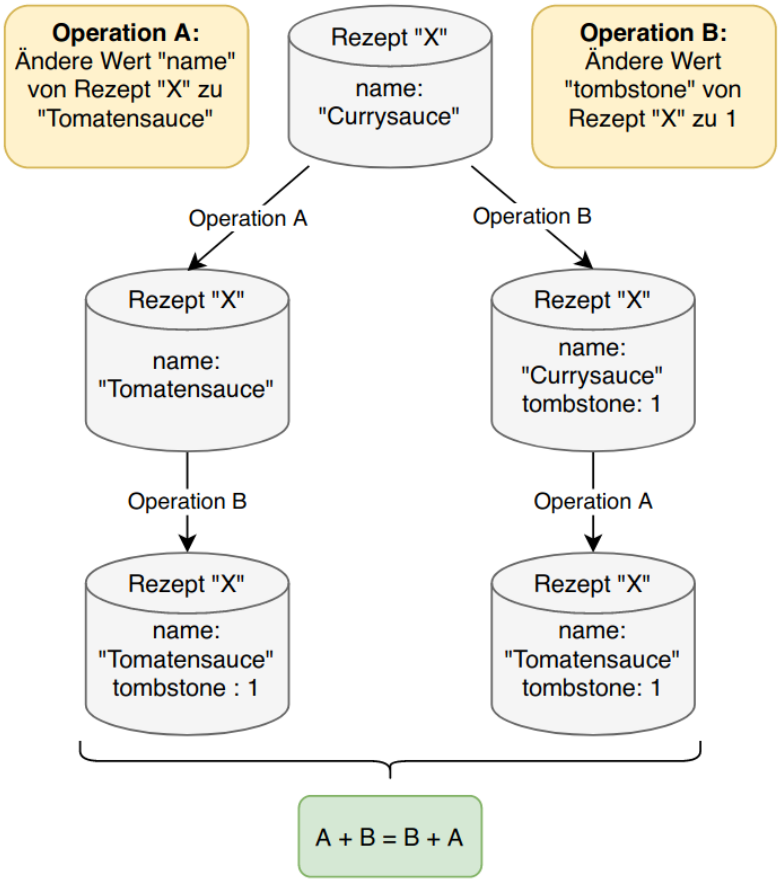

Abbildung 10: Korrektes Entfernen eines Rezeptes

5.6 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.

5.6.1 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. Sektion 4.6). Deshalb dient der Server in der Umsetzung des

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

5.6.2 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.

5.6.3 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.

5.6.4 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.

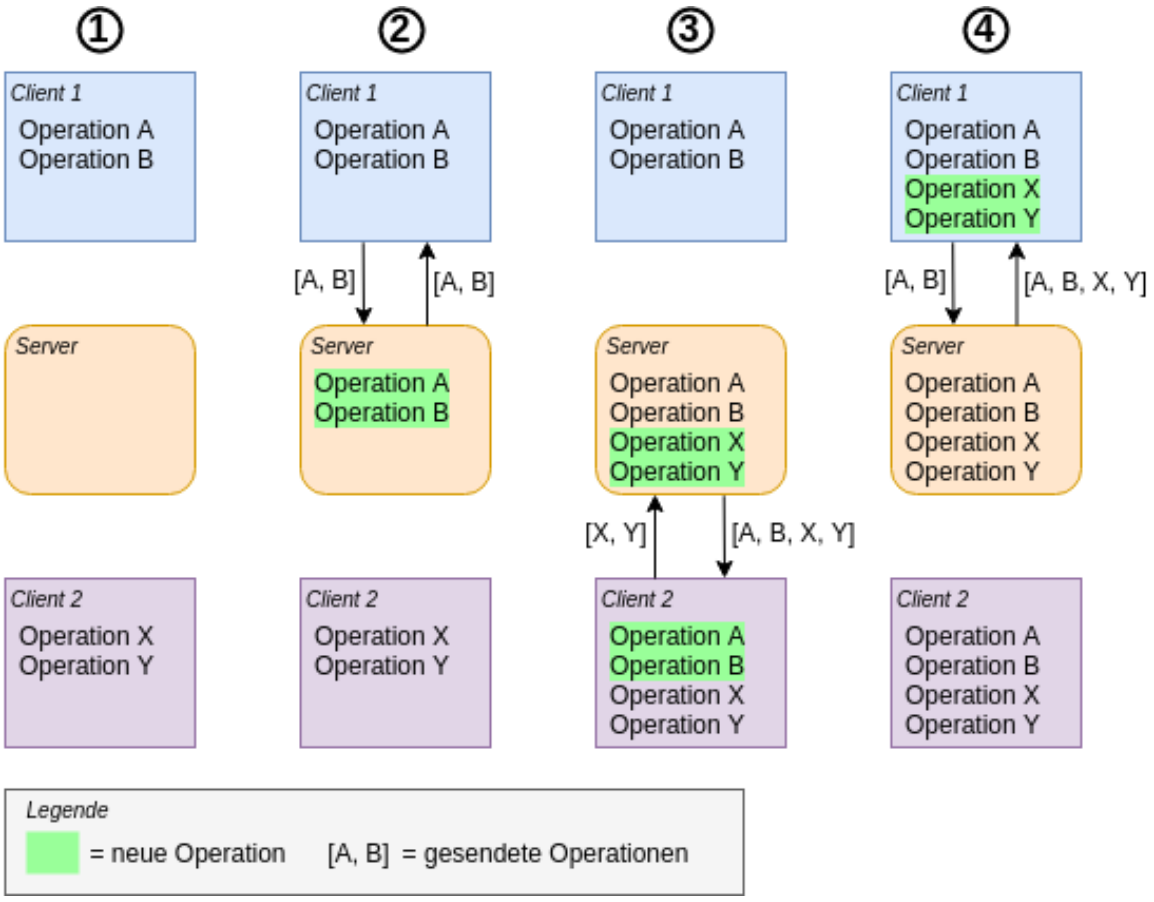

Abbildung 11: Synchronisieren über den Server

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 5.5.3 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.

6 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.

6.1 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.

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

6.1.2 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.

6.1.3 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.

6.1.4 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 und 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 6.1 zeigt das Erzeugen eines Object-Stores über die gewöhnliche IndexedDB-API, in Listing 6.2 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 6.1 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 6.2 idb: öffnen eines Object-Stores

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

6.1.5 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 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 Bibliothek verwendet. Die Wahl fiel auf MongoDB, weil es eine populäre Datenbank ist. Funktionale Kriterien spielten bei der Wahl der Datenbank keine Rolle.

6.1.6 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-Fingerabduck, 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.

6.1.7 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.

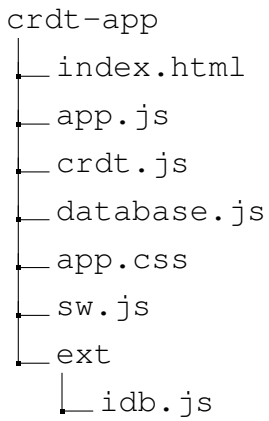

Abbildung 12: Ordnerstruktur des Clients

6.2 Nutzeroberfläche

Die Nutzung von Vue.js ermöglicht die Erzeugung sogenannter Vue Instanzen. Diese Instanzen dienen dem Template als eine Art interaktives Viewmodel (vgl. Sektion 6.1.2). 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 6.3 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 6.3 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 6.3 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.

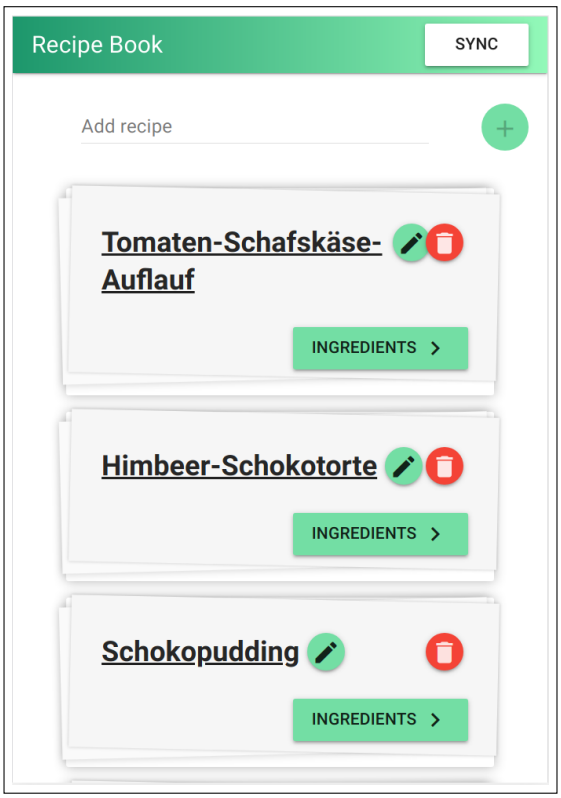

Abbildung 13: Hauptbildschirm

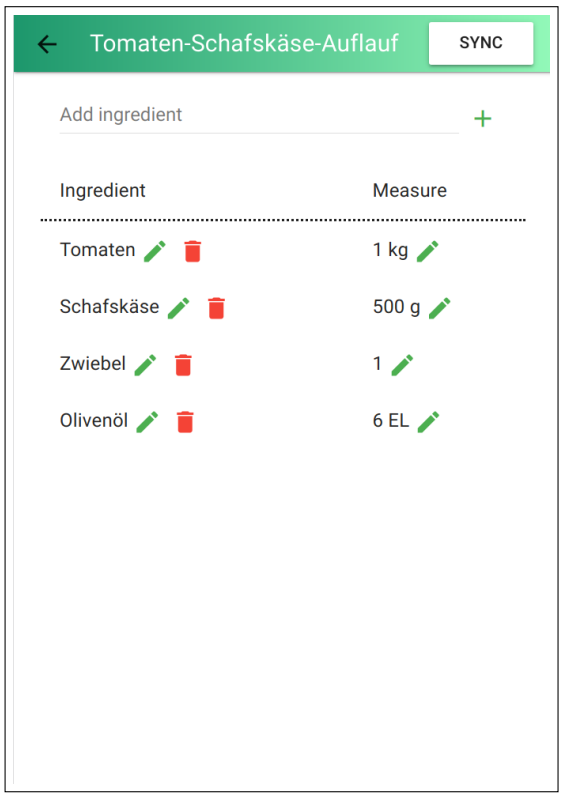

Abbildung 14: Übersicht Zutaten

6.3 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 6.4). Die zu speichernden Ressourcen werden dabei als Parameter angegeben (vgl. Listing 6.5). Die Installation des Service Workers findet immer dann statt, wenn die Website geöffnet wird und noch kein Service Worker vorhanden ist.

Listing 6.4 Installation des Service Workers

```
self.addEventListener("install", function (event) {
  event.waitUntil(
    caches.open(cachename).then(function (cache) {
      console.log("cache opened");
      return cache.addAll(urlstocache);
    }),
  );
});
```

Listing 6.5 Parameter zur Installation des Service Workers

```
var cachename = 'recipebook-v0.1';
var urlstocache = [
  'index.html',
  'app.js',
  'crdt.js',
  'database.js',
  [...]
];
```

Das Bereitstellen der Dateien aus dem Cache erfolgt über das Abfangen von Netzwerk-Requests. Das *fetch* Event, für welches der in Listing 6.6 dargestellte Event Listener registriert wird, feuert jedes Mal, wenn eine Ressource im Anwendungsbereich[1] des Service Workers aus dem Netzwerk abgerufen werden soll. Dies bezieht sich sowohl auf Dokumente innerhalb des Anwendungsbereiches, wie die bei der Initialisierung gecacheten 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 6.6 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

[1] Im Fall des Prototypen umfasst der Anwendungsbereich die gesamte Applikation

(vgl. Sektion 4.3) 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 6.6 Service Worker: Abfangen von Requests

```
self.addEventListener("fetch", function (event) {
  console.log(event.request.url);
  if (event.request.url == "https://crdt-app-server.herokuapp.com
     /api/sync") {
    event.respondWith(fetch(event.request));
  } else {
    event.respondWith(
      caches.match(event.request).then(function (response) {
        return response || fetch(event.request);
      }),
    );
  }
});
```

Im Prototypen ist dieses Verhalten wie folgt umgesetzt: Beim Aufrufen der Webseite werden alle lokal gespeicherten Applikationen aus der Datenbank geladen (vgl. Listing 6.7) und in zwei lokalen Variablen, *recipes* und *ingredients*, der *app* Instanz gespeichert (vgl. Listing 6.8).

Listing 6.7 loadUIData Methode

```
loadUIData: function (dataName) {
  return new Promise ((resolve, reject) => {
    return getAllFromStore(dataName).then(loadedObjects => {
      this[dataName] = loadedObjects.reverse();
      resolve('Updated ' + dataName + 'UI Components');
    });
  });
  }
```

Listing 6.8 Laden aller Rezepte und Zutaten

```
loadAllUIData: async function () {
  return Promise.all([
    this.loadUIData('recipes'),
    this.loadUIData('ingredients')
  ]).catch( e => {
    console.log(e)
  }).then((messages) => {
    messages.forEach(message => {console.log(message)});
  })
  }
```

6.4 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 5.5). Die Datensätze werden jeweils in einem eigenen ObjectStore in der Datenbank gespeichert. Listing 6.9 zeigt die Variable "idbPromise", welche bei allen Zugriffen auf die Datenbank genutzt wird. Wie in Sektion 6.1.4 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 6.9 Öffnen der IndexedDB Datenbank

```
var idbPromise = idb.open("crdt-app", 2, function (upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
    case 1:
      upgradeDb.createObjectStore("operations", { keyPath: "_id"
          });
      upgradeDb.createObjectStore("recipes", { keyPath: "_id" });
      upgradeDb.createObjectStore("ingredients", { keyPath: "_id"
          });
  }
});
```

Somit können anschließend Zugriffe wie das Speichern von Objekten (vgl. Listing 6.10) oder das Abrufen aller Objekte eines ObjectStores (vgl. Listing 6.11) erfolgen. Der Ablauf folgt dabei stets dem gleichen Muster. Zu Beginn wird die Datenbank mit dem "idbPromise" 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 6.10 Speichern eines Objektes im ObjectStore

```
function saveObject(storeName, object) {
  idbPromise.then((db) => {
    let transaction = db.transaction(storeName, "readwrite");
    let store = transaction.objectStore(storeName);
    return store.put(object);
  });
}
```

Listing 6.11 Abrufen aller Objekte eines ObjectStores

```
function getAllFromStore(storeName) {
  return idbPromise.then((db) => {
    let transaction = db.transaction(storeName, "readonly");
    let store = transaction.objectStore(storeName);
    return store.getAll();
  });
}
```

6.5 Generieren der Operationen

Wie in Sektion 5.5.2 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 5.5.4) und das Löschen eines Objekts über das Bearbeiten des *tombstone* Werts erfolgt, können alle drei Anforderungen über eine Methode erfüllt werden, die in Listing 6.12 aufgeführt ist.

Listing 6.12 Funktion zum Generieren einer Operation

```
function generateUpdateOperation(store, Id, key, value) {
  let targetStore = store;
  let targetObject = Id;
  let targetKey = key;
  let targetValue = value;
  let timestamp = Date.now();
  console.log("Changed " + targetObject + "property " + targetKey
      + "to " + targetValue + "in " + targetStore,);
  return {
    store: targetStore,
    object: targetObject,
    key: targetKey,
    value: targetValue,
    timestamp: timestamp,
    _id: "op_" + cuid(),
  };
}
```

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.

6.6 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 angewandt werden können. Schlussendlich werden die gefilterten Operationen Angewendet.

6.6.1 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 *processOperations* Methode mit den lokalen Operationen verglichen werden müssen. Wie Listing 6.13 zeigt, erfolgt das Laden der lokalen Operationen mit der im vorherigen Abschnitt (vgl. 6.4) beschriebenen Funktion *getAllFromStore*.

Listing 6.13 processOperations, 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 6.14) 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* Objekts, 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 neuste 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 6.14 mapOperations Methode

```
function mapOperations(localOperations, newOperations) {
  return new Promise((resolve, reject) => {
  let mappedOperations = new Map();
  let sortedLocalOperations = localOperations.sort((a, b) => {
    return b.timestamp - a.timestamp;
  });
  newOperations.forEach((newOperation) => {
    let mostRecentMatchingLocalOperation =
       sortedLocalOperations.find(
    (localOperation) =>
      localOperation.store === newOperation.store &&
      localOperation.object === newOperation.object &&
      localOperation.key === newOperation.key,
    );
    mappedOperations.set(newOperation,
       mostRecentMatchingLocalOperation);
  });
  if (mappedOperations.size > 0) {
    console.log("Mapped new operations to local operations");
    resolve(mappedOperations);
  [...]
```

6.6.2 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 6.15).

Listing 6.15 processOperations, Teil 2

```
[...]
.then((mappedOperations) => {
  return filterOperationsToApply(mappedOperations);
})
[...]
```

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. 8). Das

vorherige Gegenübermappen der eingehenden und lokalen 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 eingehenden zu lokalen Operationen während des Filterns durchzuführen, die Konsequenz wäre jedoch verschachtelter, schwer lesbarer Code.

Listing 6.16 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 Interation der for-Schleife wird über die Zeitstempel überpfüft, 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 6.16 Filtern der Operationen, Teil 1

```
function filterOperationsToApply(mappedOperations) {
  return new Promise((resolve, reject) => {
    let recipeOperationsToApply = [];
    let ingredientOperationsToApply = [];
    Array.from(mappedOperations.keys()).forEach((newOp) => {
    let localOp = mappedOperations.get(newOp);
    if (
      !localOp ||
      newOp.timestamp > localOp.timestamp
    ) {
      if (newOp.store == "recipes") {
      recipeOperationsToApply.push(newOp);
      } else ingredientOperationsToApply.push(newOp);
    }
    [...]
```

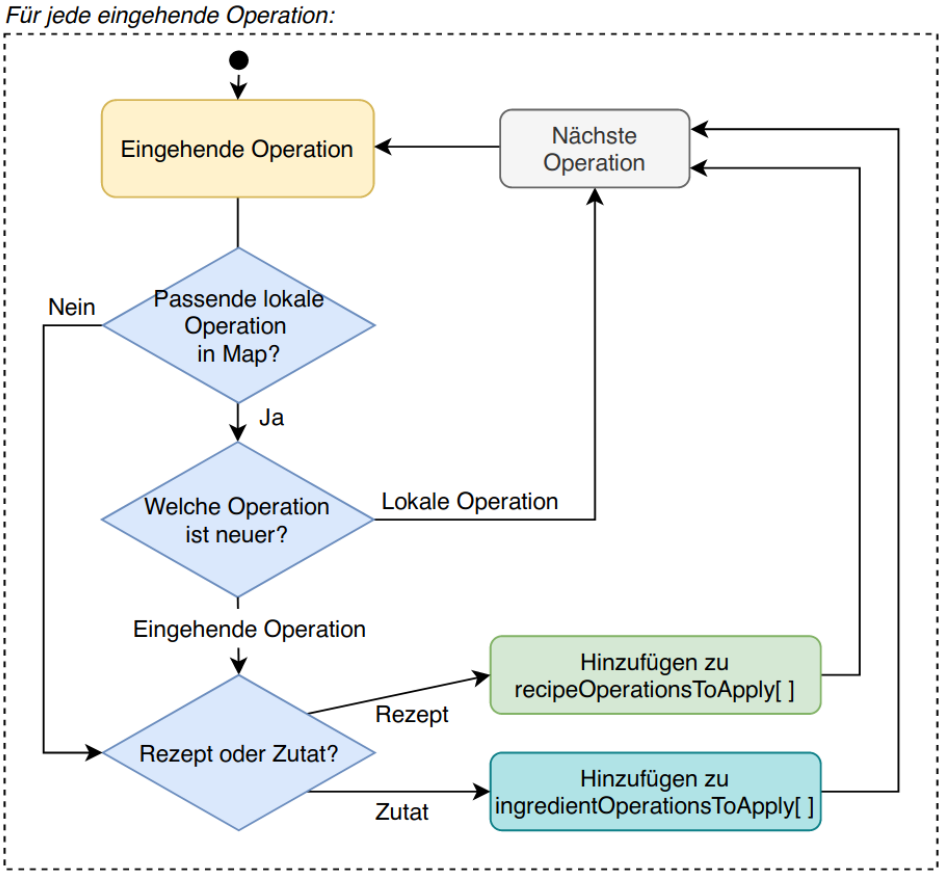

Abbildung 15: Filtern der Operationen

Nachdem alle eingehenden Operationen die for-Schleife durchlaufen haben, werden die beiden Arrays *recipeOperationsToApply* und *ingredientOperationsToApply* in der Map, *operationsToApply*, gespeichert (vgl. Listing 6.17). 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[2] zum Verarbeiten der Operationen nahtlos fortgeführt werden, ohne Kompromisse zum nutzen mehrerer Rückgabewerte treffen zu müssen.

[2]Aneinanderreihung mehrerer Promises über das wiederholte Aufrufen der .then Methode des jeweils vorherigen Promises

Listing 6.17 Filtern der Operationen, Teil 2

```
[...]
let operationsToApply = new Map([
    ["recipes", recipeOperationsToApply],
    ["ingredients", ingredientOperationsToApply],
  ]);
  if (
    operationsToApply.get("recipes").length > 0 ||
    operationsToApply.get("ingredients").length > 0
  ) {
    resolve(operationsToApply);
[...]
```

6.6.3 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.

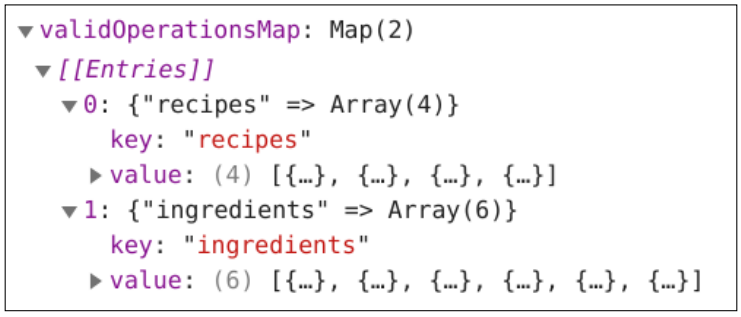

Abbildung 16: *validOperationsMap* im Chrome-Debugger

Listing 6.18 zeigt die Fortsetzung der *processOperations* Methode durch Verschachtelung

zweier *Promise.all*[3] Methoden. Was auf den ersten Blick komplex aussieht, ist in der Praxis relativ einfach.

Listing 6.18 Verarbeiten der Operationen, Teil 3

```
[...]
.then((validOperationsMap) => {
  return Promise.all(
    Array.from(validOperationsMap.keys()).map((storeName) => {
      return Promise.all(
        validOperationsMap.get(storeName).map((validOperation) =>
            {
            return updateObject(validOperation, storeName);
[...]
```

Die erste *Promise.all* Methode bekommt für jeden Schlüssel in der *validOperations-Map* 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 *updateObejct* Methode aufgerufen wird.

Listing 6.19 updateObject Methode, Teil 1

```
function updateObject(operation, storeName) {
  return getSingleObjectFromStore(storeName, operation.object)
    .then((objectToUpdate) => {
      return modifyObject(objectToUpdate, operation);
    })
    .then((modifiedObject) => {
      return saveObject(storeName, modifiedObject);
    });
}
```

In *updateObject*6.19 wird das Objekt, auf dass die Operation angewendet wird, zuerst aus der Datenbank geladen. Dies ist nötig, weil es nicht möglich ist einzelne Werte

[3]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

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 6.20) aufruft.

Listing 6.20 modifyObject Methode

```
function modifyObject(object, operation) {
  return new Promise(function (resolve, reject) {
    if (object == undefined) {
      object = {
        _id: operation.object,
      };
    }
    var modifiedObject = object;
    modifiedObject[operation.key] = operation.value;
    saveObject("operations", operation);
    resolve(modifiedObject);
  });
}
```

Es kann vorkommen, dass das Objekt, welches Modifiziert werden soll, noch nicht existiert (vgl. Sektion 5.5.2). Ist dies der Fall, gibt das Laden des Objektes im vorherigen Schritt *undefined* zurück. Für diesen Fall, 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 in der Wert des in der Operation definierten Schlüssels mit dem neuen Wert überschrieben wird. Die Operation wird nun noch über die *saveObject* Funktion (vgl. Sektion 6.4) in den lokalen Operationen gespeichert.

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

6.7 Synchronisieren

Zum Synchronisieren werden Operationen zwischen Client und Server über eine API (siehe Sektion 5.3) ausgetauscht. In dieser Sektion wird beschrieben, wie diese Funktionalität auf Client und Server umgesetzt ist.

Das Synchronisieren beginnt auf Seite des Clients mit dem Abrufen aller lokalen Operationen (vgl. 6.21). Sind die Operationen erfolgreich abgerufen, werden sie über den $\backslash sync$ Endpunkt der API, an den Server geschickt.

Listing 6.21 Synchronisation Client

```
function sync() {
  return getAllFromStore("operations")
    .then((operations) => {
      return fetch("https://crdt-app-server.herokuapp.com/api/
          sync", {
        method: "POST",
        body: JSON.stringify(operations),
        headers: {
          "Content-Type": "application/json",
        },
      });
    })
    .then((res) => {
      return res.json();
    })
    .then((data) => {
      return processOperations(data);
    })
    .catch((e) => console.log(e));
}
```

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. 5.6.1). Über die MongoDB Funktion *insertMany* können alle Operationen auf einmal in die Datenbank geschrieben werden (vgl. Listing 6.22). 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 Response an den Client geschickt. Auf diesem werden die eingehenden Operationen nun mit der *process-Operations* (siehe 6.6) Methode verarbeitet.

Listing 6.22 Synchronisation Server

```
exports.processOperations = (req, res) => {
  const operations = req.body;
  [...]
  .then(() => {
    return Operation.insertMany(operations, { ordered: false });
  })
  .catch((e) => {
    console.log(e);
  })
  .then(() => {
    return Operation.find({});
  })
  .then((operations) => {
    return res.send(operations);
  })
[...]
```

6.8 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 die selbe 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 durch die aktuelle durchsetzen, was bedeutet, dass die ältere Operation irrelevant. In diesem Fall wird die alte Operation gelöscht.

Auf dem Client findet das Löschen alter Operationen während des Filterns der Operationen statt (vgl. 6.6.2). In diesem Schritt werden die lokalen Operationen schließlich ohnehin mit den eingehenden Operationen verglichen.

Listing 6.23 Löschen überflüssiger Operationen auf dem Client

```
[...]
  if (
    !!recentMatchingOperation &&
    mappedOperation.timestamp > recentMatchingOperation.timestamp
  ) {
    deleteOperation(recentMatchingOperation._id);
  }
[..]
```

6.9 Zusammenfassung

7 Evaluation

In diesem Kapitel werden die funktionalen- und nicht-funktionalen Anforderungen reflektiert und auf Funktionalität und Vollständigkeit überprüft.

7.1 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 Existenz 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.

7.2 Zugriff mit mehreren Endgeräten zugleich

Mehrere Nutzer 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 macht. Es sollte immer individuell entschieden werden, welche Art der Lösung für gleichzeitige Änderungen des selben Wertes am besten zu den Anforderungen der Applikation passt.

7.3 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. Ein Interessanter Ansatz wäre es, Synchronisation auch über andere Wege als den Server zu erlauben. Das Synchronisieren könnte schließlich auch von Client zu Client Erfolgen, da das Austauschen der lokalen Operationen alles ist, was dafür nötig ist.

7.4 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.

7.5 Synchronisierung von Daten

7.6 SEC

7.7 Konfliktfreies Synchronisieren

7.8 Anschauliche Umsetzung

7.9 Unabhängigkeit von Datenbank-Technologien

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] Google Developers. *Progressive Web Apps Training*. 2019. URL: https://developers.google.com/web/ilt/pwa (besucht am 21. 08. 2020).

[7] Alex Feyerke. "Designing Offline-First Web Apps". In: *A List Apart* (Dez. 2013). URL: https://alistapart.com/article/offline-first/.

[8] WHATWG. *HTML Living Standard*. 2019. URL: https://html.spec.whatwg.org/multipage/workers.html#workers (besucht am 05. 07. 2020).

[9] 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).

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

[11] Maarten Van 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.

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

[13] Google Developers. *Offline First*. 2020. URL: https://developer.chrome.com/apps/offline_apps (besucht am 21.08.2020).

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

[15] 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.

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

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

[18] 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.

[19] 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.

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

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

[22] Marcel Erné. *Algebraische Verbandstheorie*. Leibniz University Hannover, 2006.

[23] Paulo Sérgio Almeida, Ali Shoker und Carlos Baquero. "Efficient state-based crdts by delta-mutation". In: *International Conference on Networked Systems*. Springer. 2015, S. 62–76.

[24] Joana da Silva Tavares. "Secure Abstractions for Trusted Cloud Computation". Diss. NOVA University of Lisbon, 2018.

[25] Carlos Baquero, Paulo Sérgio Almeida und Ali Shoker. "Pure Operation-Based Replicated Data Types". In: (Okt. 2017). eprint: 1710.04469. URL: https://arxiv.org/abs/1710.04469.

[26] Khaled Aslan u. a. "C-Set : a Commutative Replicated Data Type for Semantic Stores". In: *RED: Fourth International Workshop on REsource Discovery*. Heraklion, Greece, Mai 2011. URL: https://hal.inria.fr/inria-00594590.

[27] Denys Mishunov. "True Lies Of Optimistic User Interfaces". In: *Smashing Magazine* (2016). URL: https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interface.

[28] 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).

[29] 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.