Ravensburg-Weingarten University of Applied Science

Bachelor Thesis
Angewandte Informatik
Sommersemester 2021

Fakultät Elektrotechnik und Informatik

Handling von großen Datenmengen bei begrenzter Bandbreite in Single Page Progressive Web Apps für Cloud Computing Anwendungen

Autor:
Dennis Erdelean

Professor:
Prof. Dr.-Ing. Thorsten Weiss

11. August 2021

Bearbeitungszeit

15.02.2021 - 15.08.2021

1. Prüfer

Prof. Dr.-Ing. Thorsten Weiss

2. Prüfer

Prof. Dr. rer. nat. Martin Zeller

Autor

Dennis Erdelean
Robert-Koch-Weg 24
88250 Weingarten

Studiengang

Angewandte Informatik, Bachelor

Immatrikulationsnummer

26806

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Datum, Ort

Unterschrift

Zusammenfassung

Diese wissenschaftliche Arbeit soll Strategien untersuchen, in wie weit es möglich ist, Webanwendungen offline zu nutzen, Daten mit einem Server abzugleichen und damit keine verringerte User Experience zu generieren. Die größten Herausforderungen in der Entwicklung offline funktionsfähiger Applikationen, bei denen eine Internetverbindung geschäftsentscheidend ist, zeigen sich vor allem in der Synchronisierung von Daten von einem Client zu einem Server und vice versa, sowie das Management von Konflikten, welche beim Bearbeiten eines Datensatzes durch mehrere Nutzer enstehen können. Es gilt diese Problemstellungen genauer zu beleuchten.

Vorgestellt wird eine Webapplikation als Single-Page-App, bei der es unabhängig von der Internetverbindung möglich ist, nahezu die volle Funktionalität beizubehalten und keine Dateninkonsistenzen zu erzeugen. Das Entwicklungskonzept der Progressive Web App, unter Verwendung von Cloud Computing, bietet sich hier als ein vielversprechender Ansatz an.

Für den Offlinezugriff auf die Webapplikation trägt die sogenannte Service Worker API bei, welche Webseiten dabei unterstützt auch offline zu funktionieren. Für ausgelagerte Datenpflege schafft Cloud-Computing zusätzlich Abhilfe. Das Datenhandling erfolgt mit Apollo GraphQL.

Es werden Lösungen präsentiert, wie auch bei mangelnder oder gar fehlender Konnektivität auf die Webapplikation zugegriffen werden kann, wie Daten im Internetbrowser gespeichert werden und wie man den lokalen Zustand der Progressive Web App mit einem entfernten Server synchronisiert.

Inhaltsverzeichnis

Legende

Kursiv: Kursive Wörter sind aus dem Englischen übernommen und weltweit anerkannte Fachbegriffe. Ebenso sind Markennamen kursiv hinterlegt.
Fettgedruckt-kursiv: Begriffe die fettgedruckt-kursiv sind, befinden sich im Glossar mit enstprechender Bedeutungserklärung.

1 Einleitung

Problemstellung

Die Zahl der Internetnutzung durch mobile Geräte überstieg 2016 zum ersten Mal die der Desktop-Computer[1]. Im Gegensatz zu Desktop-Computer, welche permanent kabelgebunden, physischen Zugriff zum Internet haben, ist die Konnektivität mobiler Geräte auch in dicht vernetzten Gebieten keine Garantie. Trotz all dem, entwickeln wir größtenteils immer noch Applikationen unter der Annahme, jeder habe eine große und immer verfügbare Breitbandverbindung. Unglücklicherweise führt das zu schlechter ***User Experience***, bis hin zu erheblichen Einbußen für den Erfolg von Unternehmen. In den meisten Fällen ist eine Applikation bei fehlender Verbindung nicht mehr nutzbar. Möchte man Daten zu einem Server hochladen, verändern oder löschen, während man offline ist, gibt es bei herkömmlichen Webapplikationen kaum Ansätze wie damit umgegangen werden soll.

Motivation und Zielsetzung

Die Motivation dieser Bachelor-Thesis enstand vor aufgrund des Mangels an umgesetzen offlinefähigen Applikationen bei denen der Quelltext frei einsehbar ist. Insbesondere im Hinblick auf die Datensynchronisation mangelt es an Lösungsansätzen.

Aus der Beobachtung vieler bekannter Webapplikationen des Marktes weltweit, ist eindeutig zu erkennen, dass diese nicht mit dem Offline-first Gedanken entwickelt wurden, von dem eigentlich schon 2008 in einer Veröffentlichung des ***World Wide Web Consortium (W3C)*** die Rede war. In diesem Artikel[2] wurden verschiedene Eigenschaften von *HTML5* vorgestellt, welche es ermöglichen, Webapplikationen zu erstellen die offline arbeiten können. Auch die Vorstellung der 2017 eingeführten Technologie unter dem Marketingnamen, *Progressive Web App*, von Alex Russell[3], scheint kein flächendeckendes Umdenken hervorgerufen zu haben. Eine scheinbar revolutionierende Technologie, der langsam aber sicher immer mehr Aufmerksamkeit geschenkt wird.

Besonders in einer weiteren Veröffentlichung des anerkannten *W3C*, in dem es um die besten Vorgehensweisen mobiler Webentwicklung geht, wird dem Entwickler empfohlen, angemessene Speichermöglichkeiten für den Client zu nutzen und diesem entsprechend vorgestellt.

Wenn Entwickler Applikationen mit dem Gedanken entwicklen würden, dass eine Internetverbindung nur eine Erweiterung anstatt einer Vorraussetzung wäre, würde der Fokus sich besimmt in Richtung verbesserter *User Experience* bewegen.

Mit der Einführung von Single-Page-Applikationen populärer *frameworks*, wie *React.js*, *Vue.js* und *Angular*, ist es Softwareentwicklern nun möglich dynamische Webseiten zu erstellen und somit mehr Logik im Client zu beherbergen. Damit können Teile einer Webseite geladen werden, ohne dem Projekt zusätzliche Serverlogik hinzuzufügen.

Ein weiteres geschicktes Hilfsmittel ist das Cloud-Computing, mit dem ebenso der eigene Server massiv entlastet werden kann.

Schaut man sich nun diese verfügbaren Werkzeuge an, scheint man sehr gut für die Entwicklung einer offlinefähigen Applikation gewappnet zu sein. Was uns aber zum nächsten Problem führt, ist die Synchronisation der Daten vom Client zum Server und umgekehrt. Hauptproblematik bei der Entwicklung von Offline Web Apps ist schlichtweg die korrekte Synchronisation der Daten zu einem Server und das Erzeugen einer einheitlichen Konsistenz der Daten für alle Teilnehmer. Was passiert wenn ein Nutzer Daten zu einem Server hochladen möchte und der Server momentan nicht erreichbar ist ? Wie geht man damit um wenn mehrere Nutzer gleichzeitig einen Datensatz ändern und einer der beiden Nutzer dabei offline war ? Wie kann man sich sicher sein, dass die Daten auch wirklich publiziert wurden und auch alle Nutzer diese Updates bekommen ? Genau diese Fragen gilt es in

dieser Arbeit zu erörtern. Im nächsten Abschnitt wird genauer auf diese Fragen eingegangen und entsprechende Lösungsansätze präsentiert.

Einsatzgebiete

Die Einsatzgebiete mit der vorgestellten wissenschaftlichen Arbeit, belaufen sich vor allem auf Anwendungen in denen es von hoher Wichtigkeit ist, jederzeit auf Applikationsdaten zugreifen zu können. Um einige zu nennen:

- **Medizinsektor** - Gesundheitsinformationsysteme profitieren immens von Applikationen welche offline funktionieren. Diese Systeme werden in vielen Fällen in Notfallsituationen genutzt bei denen das Personal nicht viel Zeit hat. Diese Arbeiter müssen in der Lage sein Patienenprofile schnell und zuverlässig abzurufen. Die Verfügbarkeit von Daten sind hier entscheidend.
- **Öffentlicher Dienst** - Informationen wie Gesetze, Fahrpläne oder Regulatorien welche den Bürgern eines Volkes immer zur Verfügung stehen sollten. Wenn man bedenkt, dass man den Großteil dieser Informationen unterwegs am Smartphone abruft, oft unter schlechten Netzwerkbedingungen, sollten vor allem diese Information offline abrufbar sein.
- **Mobile Dienstleistungen** - Logistik, Transport, Liefer und Postservice sind Bereiche bei denen ein Großteil der Arbeit mobil gehandhabt wird und unabhängig von Netzwerkbedingungen funktionieren muss.
- **Geo Services** - Anwendungen bei denen Geodaten verwendet werden, sollten generell offlinefähig sein. Diese Applikationen werden meistens unterwegs genutzt, wo es zu immer zu Netzwerkproblemen führen kann.
- **Entwicklungsländer** - In diesen Regionen ist oft eine Internetverbindung ein Luxusgut und wenn jemand eine besitzt, ist diese höchstwahrscheinlich nicht zuverlässig. Vor allem diese Regionen profitieren sehr von Applikationen die eine Internetverbindung nur als ein Bonus bzw. eine Erweiterung ansehen.
- **Trading und Auktionen** - In diesen Geschäftsbereichen ist es von hoher Wichtigkeit Offlinefunktionalitäten einzubauen. Man stelle sich vor, es wird eine wichtige Auktion getätigt oder ein Handel abgeschlossen und genau im Moment des Erwerbs kommt es zu einem Netzwerkausfall. Es muss für solche Szenarien eine Lösungen geben wie den Zeitpunkt des Kaufs mit einem Zeitstempel zu versehen, offline zu speichern und bei wiederkehrender Internetverbindung die Aktion auszuführen.

Aufbau und Struktur

Beginnend mit der Motivation und dem eigentlichen Hauptziel dieser wissenschaftlichen Arbeit, wird dann in den weiteren Kapiteln auf gundlegende bekannte Konzepte eingegangen, um ein Grundwissen zu erlangen. Kapitel 2.2 beschäftigt sich mit bekannten Entwurfsmustern der Datensynchronisation. Das darauffolgende Kapitel erläutert die angewandten Konzepte für die Implementierung des Demonstrators in Kapitel 4. In Kaptitel 5 werden Performanceergebnisse des Demonstrators evaluiert. Das letzte Kapitel der wissenschaftlichen Arbeit führt zu einer Diskussion und dem Fazit, indem sich damit auseinandergesetzt wird, wie ausgereift der Demonstrator und die aktuellen Konzepte sind. Besonders der Blick in die Zukunft steht dabei im Fokus.

2 Datensynchronisation

Die Hauptaufgabe bei Anwendungen unter unzuverlässigen Netzwerkbedingungen, ist die Datensynchronisation. Im Folgenden werden verschiedene Strategien vorgestellt, wie eine Applikation synchronisiert werden kann.

2.1 Datensynchronisationsentwurfsmuster

1995 wurden erstmals im Buch "Design Patterns, Elements of Reusable Object-Oriented Software", 23 *Design Patterns* vorgestellt. Das sind die allgemein bekannten Software Design Entwurfsmuster, die bis heute noch ihre Gültigkeit finden. Der bekannte Architekt und Entwurfstheoretiker Christopher Alexander sagt, “Jedes Muster beschreibt ein Problem, welches immer und immer wieder in unserer Umgebung auftritt und beschreibt den Kern der Lösung zu diesem Problem auf eine Art und Weise, so dass man diese Lösung über eine Millionen Mal benutzen kann ohne dies doppelt getan zu haben ”[4]. Was Christopher Alexander über Gebäude und Städte sagte, trifft genau so auch auf das Entwerfen von Software zu. Nehmen wir an, man erkennt beim Programmieren immer wiederkehrende Probleme und entwickelt über die Zeit Herangehensweisen wie man diese Probleme lösen kann. Da jedoch nicht jedes Programmierproblem exakt auf die Zeile gleich ist, greift man jedes Mal zur den erlernten Herangehensweisen und wendet es auf die Probleme an. Diese Lösung ist jedoch zum gegebenen Zeitpunkt nicht dokumentiert und öffentlich anerkannt. Irgendwann kommt man dann eventuell auf die Idee, dass diese Lösung der Öffentlichkeit sehr nützlich sein könnte. Das sind die ersten Schritte zur Geburt eines *Design Pattern* zu deutsch, Entwurfsmuster. Man sollte beachten, dass solche Muster nicht als rein kopierbare Lösungen betrachtet werden sollten. Es geht vielmehr darum, für einen Satz an bekannten Problemen, entsprechende Lösungen zu entwicklen, welche auf diese Probleme anwendbar sind. Dadurch muss das Rad nicht neu erfunden werden und man spart sich enorm viel Zeit bei der Problemlösung.

Das Werk *"Data Synchronization Pattern in Mobile Application Design"*[5] von Zach McCormick und Douglas Schmidt erschien im Jahre 2012. In dieser wissenschaftlichen Arbeit wurde zum Ersten Mal ein Katalog an verschiedenen Datensynchronisationsentwurfsmuster vorgestellt. Auch wenn das schon einige Zeit her ist, finden die beinhalteten Konzepte immer noch ihre Gültikgeit. Die Wahrheit ist aber auch, dass Frontend-Entwickler heute viel mehr Möglichkeiten haben. Vor allem durch das Konzept der *Progressive Web App*, für die nur ein Internetbrowser auf dem Gerät installiert sein muss, enstehen neue Möglichkeiten aber genauso auch Probleme.

Nachfolgend werden die verschiedenen bekannten Datensynchronistationsentwurfsmuster erläutert und in Kontext heutiger Anwendungen gebracht. Datensynchronisationsentwurfsmuster werden in 3 Kategorien aufgeteilt:

- **Synchronisationsmechanismen**
- **Datenspeicher**
- **Verfügbarkeit**

2.2 Entwurfsmuster für Datensynchronisationsmechanismen

Pattern für Datensychnronisationsmechanismen versuchen die Frage zu beantworten, "Wann soll eine Applikation, Daten zwischen einem Endgerät und einem entferntem System, wie einem Cloud Server, synchronisieren ?". Unglücklicherweise gibt es keine generelle Lösung die diese Frage beantworten

kann. Faktoren wie Netzwerkverfügbarkeit, Aktualität der Daten und *User Interface Design* spielen hier eine entscheidende Rolle.

Asynchrone Datensynchronisation

Eine der größten und wohl wichtigsten Herausforderungen von Softwareentwicklern ist es, schnellen Datenzugriff zu ermöglichen. *Responsiveness* und Latenz sind zwei Hauptelemente, welche das entscheiden. Eine langsam reagierende Anwendung führt zu einer schlechten *User Experience*. Auch wenn zum Beispiel eine Formulareingabe sehr schnell reagiert, der Nutzer aber dafür lange warten muss bis die Daten geladen sind, ensteht meistens Frustration beim User. Aus diesem Grund ist es wichtig, dass die Applikation nicht blockiert ist, wenn eine Synchronisation erfolgt. Wie man im Diagramm sehen kann, wird ein Synchronisationsevent ausgelöst sobald die Anwendung für die Nutzung bereit ist und kehrt dann sofort wieder in den Arbeitsmodus zurück.

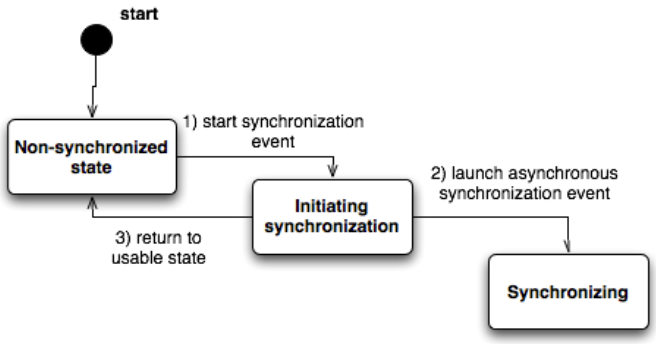

Abbildung 1: *Asynchrone Datensynchronisation*[6]

Die Vorteile dieser Lösung sind offensichtlich. Die Verfügbarkeit der Anwendung während der Synchronisation. Das ist die Stärke einer im Hintergrund laufenden Synchronisation. Die Nachteile hingegen sind beachtlich, denn es können Inkonsistenzen bei Zugriff meherer Nutzer auf einen Datensatz enstehen. Genauso wie, wenn die Größe der Datenmenge nicht bekannt ist, was zu einer Netzwerküberlastung führen kann.

Tatsächlich ist es schwierig für diese Kategorie enstprechende Beispiele zu nennen, denn sie ist sehr anhängig vom Geschäftszweck. Zum Beispiel gäbe es da den Anwendungsfall einer E-Kommerz Applikation, welches aus Kategorien und Produkte besteht. Die Komponente "Kategorienïst sehr statisch und kann daher auf Applikationslevel gespeichert werden. Die Produkte hingegen sind dynamisch. Werden Produkte geladen oder wenn der *infinity scroll* Ansatz verfolgt wird, würde sich das asynchrone Laden der Daten anbieten.

Synchrone Datensynchronisation

Für manche Applikationen ist es notwendig, dass erst bestimmte Daten verfügbar sein müssen, damit die Applikation benutzt werden kann oder darf. Das können am meist genutzte Datensätze oder Echtzeitdaten sein. Im folgenden Diagramm sieht man wie die Anwendung in einen Zustand geht, bei dem der User warten muss, bis die Synchronisation abgeschlossen ist, um mit der Anwendungn zu interagieren.

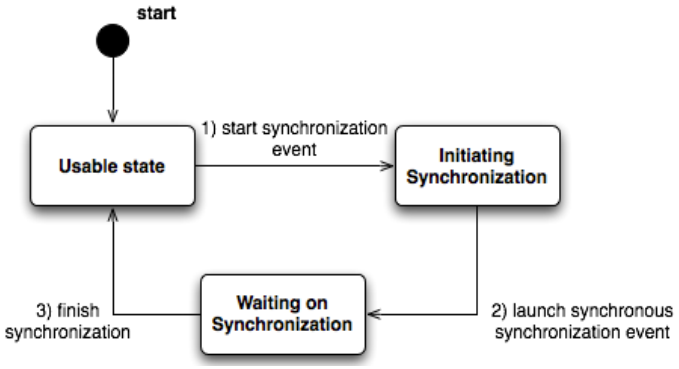

Abbildung 2: *Synchrone Datensynchronisation*[7]

Die Vorteile dieser Lösung sind die einfache Handhabung solch einer zustandsbasierten Variante. Doch im Gegenzug verringert das die User Interaktion mit der Applikation.

Ein gutes Beispiel für eine synchrone Datensynchronisation wäre eine Anwendung in der stets Nutzerrechte geprüft werden müssen, wie ein kostenpflichtiges Abonnement. Somit wäre es hier angebracht, vollen Zugriff auf die App zu blockieren, bis der Accountstatus des Nutzers überprüft wurde. Ein anderes Beispiel wäre die Verwaltung von Produkten in einem Online Shop seitens eines Administrators, da hier die Genauigkeit der Daten von hoher Wichtigkeit ist.

2.3 Entwurfsmuster für Datenspeicher und Verfügbarkeit

Abseits der Frage "Wann soll eine Applikation, Daten zwischen einem Endgerät und einem entferntem System, wie einem Cloud Server, synchronisieren ?", die im vorherigen Kapitel erörtert wurde, gibt es eine andere vergleichbar wichtige Frage zu beantworten. "Wie sollen die Daten gespeichert werden?", um maximale Verfügbarkeit der Daten zu erreichen. Für manche Applikatioen reichen ein Bruchteil der Daten aus, um lauffähig zu werden. Für andere hingegen trifft dies nicht zu, da eventuell mit Echtzeitdaten gearbeitet wird.

Die Kategorie der Datenspeicherung und Verfügbarkeit besitzt 2 Entwurfsmuster.

Partieller Speicher

Alle clientseitigen Programme haben Ressourcenprobleme in Form von Netzwerkbandbreite, physischer Speicher oder Plattformlimitierungen. Serverseitig sind die Probleme jedoch viel einfacher zu lösen, da man Dinge wie Bandbreite, Speicher und Prozessor erhöhen kann. Unglücklicherweise gibt es diese Möglichkeiten im Client nicht. Aus diesem Grund muss eine Software so entwickelt werden, damit alle gängigen Endgeräte eine zufriedenstellende *User Experience* aufweisen. Die Intention beim partiellen Speicher ist es die Daten zu synchronisieren und nur zu speichern, wenn diese benötigt werden.

Die nachfolgende Illustration zeigt die Sequenz eines Zugriffs auf Daten, welche sich nicht im Applikationspeicher befinden. Werden die Daten empfangen, sendet das Datenzugriffsobjekt keine Netzwerkanfragen mehr an den Server und greift stattdessen auf den internen Speicher zu.

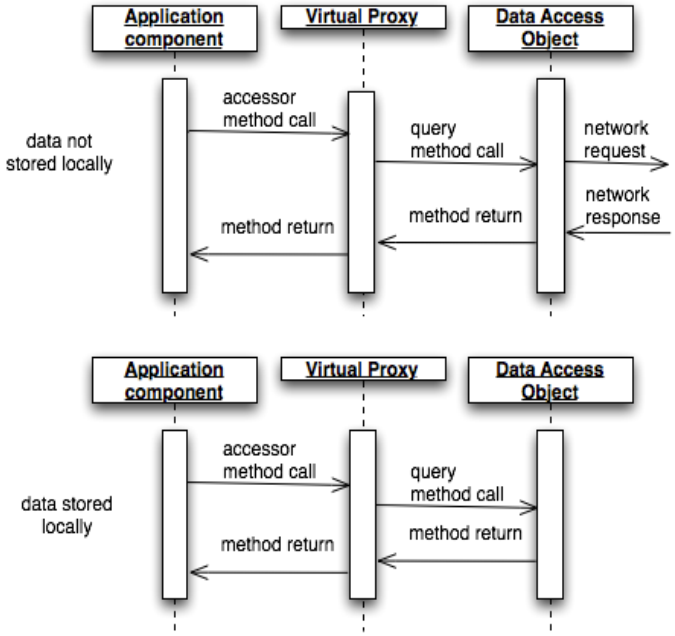

Abbildung 3: *Partieller Speicher*[8]

Dieses *Pattern* ist sehr vorteilhaft bei begrenztem Speicher und hat noch den Vorteil unterschiedliche Mengen an Daten synchronisieren zu können. Die Hauptprobleme jedoch sind Netzwerkprobleme, insbesondere die Anzahl der Netzwerkanfragen, um Daten zu empfangen sowie die Datentransferrate, was die *User Experience* beeinträchtigen kann. Ein bekanntes Beispiel wäre ein Geoinformationssystem wie GoogleMaps[9], welches auf Wunsch schon besuchte bzw. geladene Routen oder Kartenfragmente für jede Sitzung der Anwendung speichert.[10]

Kompletter Speicher

Im Gegensatz zum partiellen Speicher, liegt die Absicht des kompletten Speicher, die Daten zu synchronisieren und zu speichern noch bevor die Daten gebraucht werden. Dies führt zu besseren Antwort-und Ladezeiten. Partieller Speicher lädt Daten auf Anfrage, was in Situationen in denen keine Netzwerkverbindung besteht und die Applikation trotzdem funktionieren muss, nicht möglich ist.

Hier gibt es einen Unterschied zwischen 2 Arten von Aktionen:

- Synchronisation
- Datenabruf

Das Diagramm gibt einen Überblick des Entwurfsmusters Kompletter Speicher.

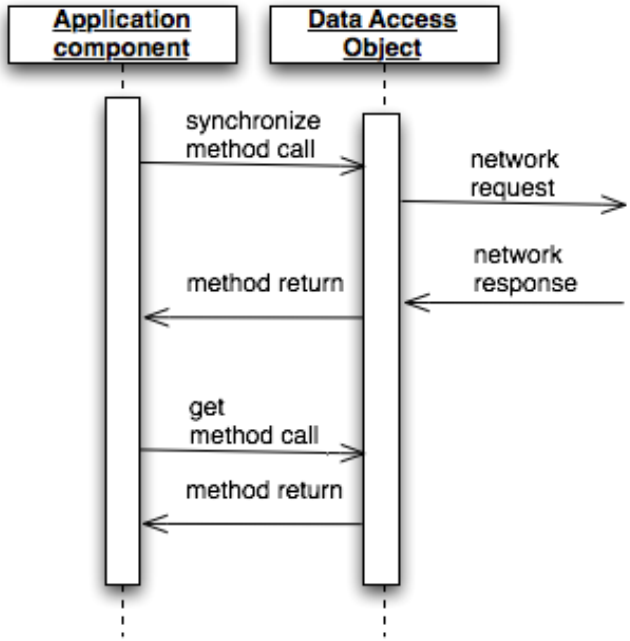

Abbildung 4: *Kompletter Speicher*[11]

Ein Synchronistationsereigniss sendet eine Netzwerkanfrage und gibt Daten zurück. Das Entwurfsmuster besagt, alle Daten auf einmal zu synchronisieren und bei einer späteren Datenabfrage diese Daten lokal zurückzugeben. Der Vorteil liegt klar in der geringeren Abhängigkeit der Netzwerkverfügbarkeit. Doch auch dieses *Pattern* ist nicht ohne Schwächen. Hier muss man beachten wie groß die zu synchronisierende Daten sind und ob diese auf dem Endgerät gespeichert werden können. Nachteil ist die inherente Ladezeit der Synchronisation.

Als Beispiel könnte man sich eine Applikation vorstellen, bei welcher man eine Stadt über ein Suchfeld suchen kann und als Ergebniss relevate Information über diese Stadt bekommt. Diese Daten müssen nur einmal vom Server geladen werden, können dann lokal gespeichert werden und sind dann jederzeit ohne Netzwerkanfragen zu senden abrufbar. Viele weitere Beispiele wären Dateihosting Anbieter wie *Dropbox*, *Google Drive* oder *Evernote*, welche die Möglichkeit bieten mit den Daten auch offline zu arbeiten.

2.4 Entwurfsmuster für Datentransfer

Datentransferentwurfsmuster setzen sich mit der Fragestellung auseinander, wie man den Datentransfer der Synchronistation so optimiert, damit die Menge der Datensätze minimal ist. Wie schon erwähnt ist die Netzwerkverfügbarkeit in einigen Ländern, Orten und vor allem auf mobilen Geräten immer noch ein Problem. Daher gilt es besonders diese Entwurfsmuster näher zu betrachten.

Voller Transfer

Wie der Name schon sagt, werden beim vollen Transfer alle Daten geladen. Der illustrierte Ablauf zeigt den Prozess.

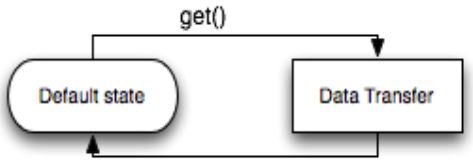

Abbildung 5: *Voller Transfer*[12]

Der größte Vorteil ist die Einfachheit dieses Ansatzes. Einfachheit kommt aber selten ohne Nebenwirkungen davon. Ein möglich auftretender Nachteil ist die Redundanz an Daten. Wenn sich nur ein kleiner Teil der Daten ändert oder gar nicht, wird unnötig Bandbreite genutzt. Was wenn in einer riesigen Liste mit Datensätzen, in einem Datesatz sich nur eine Information ändert ? Mit dem Entwurfsmuster des vollen Transfers wird dann nicht nur der eine Datensatz aktualisiert sondern die Komplette Liste an Daten. Eine offensichtliche Ressourcenverschwendung findet statt.

Auch hier gibt es verschiedene Beispiele. Ein passendes Beispiel wo dieser Ansatz Sinn macht wäre eine Onlinezeitung, welche täglich die Top 10 Nachrichten des Tages veröffentlicht. Bedeutet, dass sich die Top 10 Artikel nicht ändern. Bei Datenänderung, ändert sich somit auch die die komplette Liste an Daten. Ein weiteres Beispiel wäre ein Applikationsfehler. Anstatt den Fehler manuell zu beheben kann die Applikation neu gestartet werden und lädt somit alle Daten neu. Hiermit kann zusätzlicher aufwendiger Quellcode verhindert werden.[13]

Zeitstempeltransfer

Da in den meisten Fällen ein voller Transfer, Ressourcen verschwendet gibt es die Möglichkeit eines zeitbasierten Verfahrens. Speziell wenn sich ein Datensatz seit der letzten Synchronisation nicht geändert hat, kann es zu redundanten Datentransfers kommen. Das nachfolgende Diagramm zeigt eine komplexere Logik für einen Datentransfer durch Nutzung eines Zeitstempels.

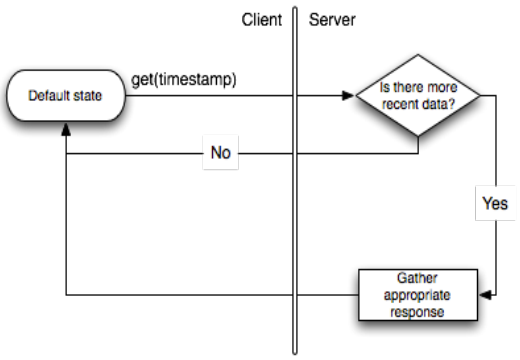

Abbildung 6: *Zeitstempeltransfer*[14]

Der Client initiiert eine Anfrage mit einem Zeitstempel. Der Server kann nun anhand des Zeitstempels prüfen ob er Daten an den Client zurückgeben muss.

Der Vorteil ist eine effizientere Resourcennutzung im Gegensatz zum Ansatz des vollen Transfers. Generell kommt ein beaknntes Problem beim Zeitstempeltransfer auf, nämlich Datenlöschungen. Gelöst wird dieses Problem durch *soft deletes*, welches im Kapitel Delta Query 4.4.2 erläutert wird. Diesen Ansatz verfolgt auch der erste Demonstrator dieser wissenschaftlichen Arbeit, in dem in Kapitel 4 eingegangen wird. Beispiele für zeitbasierte Verfahren sind im Allgemeinen Applikation, welche mit historischen Daten arbeiten, wie ein soziales Netzwerk, Aktivitätstracker und viele andere Anwendungen welche mit zeitlich basierte Daten nutzen.

Mathematischer Transfer

Was wenn für den Geschäftszweck ein voller Transfer nicht sinnvoll ist und zusätzlich dazu nicht alle Strukturen und Daten der Anwendung nicht anhand einen Zeitstempel synchronisiert werden können ? Der mathematische Transfer kann hier als Erweiterung des zeitbasieren Verfahrens agieren. Die Intention hinter diesem Ansatz ist es Zeit für den Datentransfer einzusparen und mehr Zeit der Entwicklung einer mathematischen komplexen Methode zu widmen, wie zum Beispiel die Berechnung einer Prüfsumme teile großer Datenmenge oder der entwickelte Algorithmus von Trachtenberg, Starobinski und Agarwal[15].

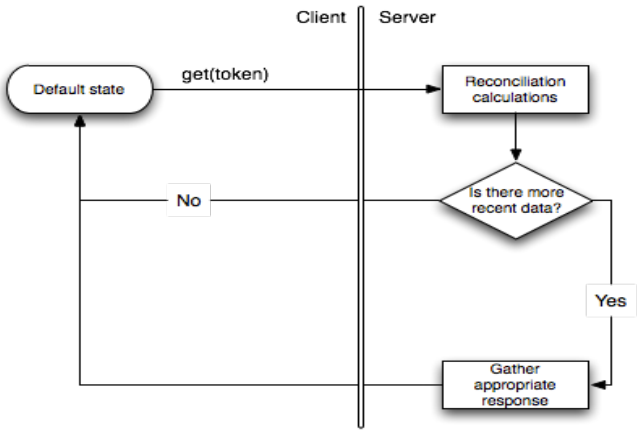

Abbildung 7: *Mathematischer Transfer*[16]

Der Flowchart ist ähnlich zu dem des Zeitstempeltransfer, mit dem Zusatz der Berechnung von Unterschieden in Datensätzen. Beim zeitbasierten Verfahren findet grundsätzlich nur ein zeitlicher Vergleich statt, doch mit dem mathematischen Tranfer würde das eine viel aufwendigere Kalkulation sein, welche als separatren Prozess angesehen werden sollte.

Für diesen Ansatz gibt es auch einige Beispiele, beginnend bei einem Geoinformationssystem wo eine Kennungen von Kartenfragmente als *token* angezeit werden können oder noch viel komplexere Systeme wie Video Streaming Dienste, welche Berechnungen wie Summe aus kompletten Differenzen oder quadratische Summen von Differenzen, um ein optimale Kodierung für einen neuen ***frame*** zu finden.

2.5 Publisher/Subscription Entwurfsmuster

Ein weiteres hilfreiches und perfomantes Konzept ist das *Publisher/Subscription* Entwurfsmuster. Dies ist ein *Pattern* im Bereich des asynchronen Nachrichtenaustauschs. Eine veröffentlichte Nachricht wird nur an die Empfänger gesendet, welche sich dafür interessieren. Ein sogenannter *publisher* abonniert ein bestimmtes Thema, über das er stets informiert werden will und wird dann jederzeit über Neuigkeiten informiert. Dieses Konzept hat den entscheidenden Vorteil, dass ein Nutzer Daten nicht explizit abfragen muss, sondern immer wenn neue Daten verfügbar sind, diese auch empfängt.

2.6 Delta Replikation

Um näher auf das Entwurfsmuster des Zeitstempeltransfers 2.4 und des mathematischen Transfers 2.4 einzugehen, wird das Konzept der Delta Replikation[17] [18] herangezogen. Delta Replikation transferiert Daten von einer lokalen Datenbank zu einer zentralen bzw. entfernten und bietet Datentransformationsmöglichkeiten an. Eine gültige Delta Replikation wird durch die Kompatibilität der Datenbankschemas der Quelle und des Ziels bestimmt.

Bei Anwendungen, welche eine hohe Bandbreite voll ausnutzen müssen und eine Vielzahl an Besuchern vermerken, ist es sinnvoll Applikationsdaten nicht nur zentral zu speichern sondern auch dezentral in einer Clientdatenbank, um den Server zu entlasten und dem Nutzer perfomante Antwortzeiten zu bieten. Da die Daten in der zentralen Datenbank sich ständig ändern, muss es eine automatische Verteilung der neuen Daten an die verschiedenen dezentralen Datenbanken geben. Die einzige Vorraussetzung für eine Delta Replikation ist, dass das Datenbankschema der zentralen Datenbank die gleiche Struktur wie das der lokalen aufweist.

Die Vergangenheit weist eine nennenswerte Bemühung wissenschaftlicher Projekte im Bereich der Datenintegration auf. Im Allgemeinen ist das Ziel der Datenintegration, Daten verschiedener Quellen unter Anwedung eines globalen Datenmodells zu kombinieren und das Auffinden und Auflösen von Schema und Datenkonflikte, damit eine homogene und vereinte Ansicht verügbar gemacht wird[19]. Das bedeutet es werden nicht mehr Daten ausgeliefert als benötigt und angefragt.

Vorteile dieses Ansatzes sind unter anderem:

- sehr schelle Antwortzeiten
- Daten von temporär unerreichbaren Quellen können dem Nutzer immer noch präsentiert werden
- Zusätzliche Datenquellen können jederzeit hinzugefügt werden

Apollo GraphQL und die beiden Frameworks *Offix* und *Graphback*, implementieren das Konzept der Delta Replikation. Möglich gemacht wird dies mit Aufbau eines lokalen und entfernten Datenspeicher, welche durch die *GraphQL Schema Types* die gleiche Datenbankschemastruktur haben. Im Kapitel 4 dann mehr dazu.

Je nach Geschäftszweck gilt es nun Werkzeuge zu finden, welche diese Konzepte in Programmcode umsetzen können. Im Fall einer Applikation die bei begrenzter Bandbreite agiert und große Datenmengen handhaben muss, bieten sich einige dieser Konzepte an. Vor allem das Ansätze der asynchronen Datensynchronisation gepaart mit dem Zeitstempeltransfer, Delta Replikation und *Subscriptions* wirken vielversprechend, um eine perfomante und ausfallsichere Anwendung zu entwickeln.

3 Angewandte Konzepte und Technologien

Gesucht wird eine universell einsetzbare Webapplikation, welche Offline-Funktionalitäten besitzt und große Datenmengen handhaben kann. Der Demonstrator dieser wissenschaftlichen Arbeit sollte beliebte Technologien einer weitreichenden und aktiven Community verwenden. In den nachfolgenden Abschnitten werdendie Technologien vorgestellt.

3.1 Offline-First

Der Ansatz *Offline-first* startet mit dem Gedanken eine Webapplikation zuerst einmal völlig offline zu entwickeln und dann sukzessive Online Eigenschaften hinzuzufügen. *Offline-first* ist ein relativ neues Desgin Paradigma für Webapplikationen. Im Jahre 2012 wurde dieses Konzept zum ersten Mal aufgegriffen. Ein Artikel davon stammt von Joe Lambert mit dem Titel *Öffline First – A better HTML5 User Experience"*[20]. In diesem Artikel hebt er heraus, dass *Offline* eine Funktion ist und diese schon zu Beginn der Entwicklungsphase berücksichtigt werden soll. Lambert präsentiert 3 Richtlinien aus technischer Sicht, für die Entwicklung von *Offline-first* Applikationen.

- **Die Logik der Applikation sollte sich vom Server zum Client verschieben** - Der Server wird nur als Datenspeicher und als Kommunikation mittels ***JSON*** zwischen Client und Server genutzt
- **Kreation einer clientseitigen Abstraktionsschicht** - Der Client sollte die gleichen ***API***-Endpunkte wie im Server haben
- **Datenschicht** - besagt welche Daten vom Server koordiniert und im Internetbrowser zwischenspeichert werden

Alex Feyerke verfolgt ähnliche Ansätze wie Lambert für die technischen Aspekte einer *Offline-first* Anwendung. Er erwähnt hierbei, dass die Anwendungslogik größtenteils im Browser läuft und dass für eine volle Offline-Erfahrung, der Client alle Daten im Internetbrowser speichern sollte, um eine Synchronisation zum Server möglich zu machen. Feyerke sieht jedoch *User Experience* als größeres Problem, bei dem die Industrie noch keine passenden Entwurfsmuster entwickelt hat. Im Folgenden werden verschiedene Szenarien aufgelistet in denen eine Änderung der Internetkonnektivität für Probleme sorgt:

- Für den Nutzer ist es frustrierend wenn sie den Zugriff auf Daten verlieren, sobald sie Offline sind. Das Mindeste was die Applikation tun sollte, ist die Daten wenigstens in einem lesbaren Modus zu halten. Noch besser wäre es wenn die Daten auch Offline modifizierbar sind.
- Offline sollte nicht als ein Fehler behandelt werden. Wenn der Nutzer etwas Offline erstellen möchte, sollte die Applikation in der Lage zu sein das Erstellte, lokal zu speichern und den Nutzer darüber informieren, dass die Daten zu einem späteren Zeitpunkt gesendet werden, anstatt eine Fehlermeldung anzuzeigen. Der User sollte zu jederzeit die nötigsten Informationen sehen können.
- Es ist unvermeidbar, dass es Konflikte und zu Dateninkonsistenzen zwischen Versionen der Daten komm. Vor allem wenn die Anwendung es zulässt, Daten simultan auf verschiedenen Geräten zu editieren. Die Anwendung sollte daher eine einfache und benutzerfreundliche Benutzeroberfläche bereitstellen, um Konflikte zu beseitigen.

- Ein weiteres Problem taucht bei Anzeigen von Nachrichten oder Produkten in chronologischer Reihenfolge. Wenn die Applikation Nachrichten in der Reihenfolge in der sie gesendet wurden anzeigt, sind diese leicht zu bermerken. Verwirrend wird es dann aber wenn eine Nachricht eine Antwort auf eine viel ältere Nachricht war. Auf der anderen Seitem, wenn die Applikation die Nachrichten in einer chronologischen Reihenfolge anzeigt ist die Sequenz der Daten wichtig. Eine kürzlich gesendete Nachricht ist daher nicht immer auch die Neueste und taucht an Stellen auf, an der der Nutzer sie nicht erwartet.

Ein weiterer Autor fasst mit seinem Werk *"Good offline design from the user's perspective"*[21], die wichtigsten Punkte ebenfalls sehr gut zusammen:

- Stelle dem User so viel Inhalte wie möglich zu Verfügung wenn er offline ist
- Der User sollte unabhängig von der Internetverbindung in der Lage sein den Inhalt anzusehen und bearbeiten zu können
- Nutze benutzerfreundliche Fehlermeldungen
- Der User sollte die Aufgabe die er begonnen hat auch beenden können, selbst wenn er währenddessen die Verbindung verliert
- Kann die Applikation einen Konflikt nicht automatisch auflösen, gebe dem Nutzer Möglichkeiten wie er den Konflikt auflösen kann
- Speicher die für den User wichtigsten Informationen zuerst zwischen
- In einem *empty state* (nicht abgefangene Situation im Programmcode), zeige dem User was er als nächstes tun kann
- Behalte den Status der Applikation über mehrere Sitzungen
- Skaliere die Netzwerknutzung auf die verfügbare Bandbreite
- Lösche niemals den Zwischenspeicher, es sei denn der Nutzer hat es explizit angefordert

Einige dieser Prinzipien beziehen sich nicht direkt auf gute Offline-Erfahrung. Manche davon sind einfach nur gute *User Experience*, sowie die Entscheidungsmöglichkeiten bei einem *empty state*. Trotzdessen assistieren all diese Prinzipien die Szenarien die Feyerke wahrgenommen hat.

3.2 Progressive Web App

Es gibt verschiedene Arten von Applikationen welche bestimme Zwecke verfolgen. Die zwei wohl bekanntesten sind native Applikationen und Webapplikationen. Eine native App ist eine Anwendung, die entwickelt wurde, um auf einer bestimmten Plattform oder einem bestimmten Endgerät zu arbeiten. Aus diesem Grund können native Apps mit den auf der jeweiligen Plattform installierten Betriebssystemfunktionen interagieren und diese nutzen. Um mit dem jeweiligen Bestriebssystem interagieren zu können, ist daher auch eine bestimmte Programiersprache nötig.

Eine Web App ist eine Anwendung, die auf beliebigen Endgeräten oder Browsern funktioniert. Darum wird die App unabhängig vom Betriebssystem programmiert. Im Gegensatz zu einer nativen App, ist es möglich mit einer einzigen App auf mehreren Endgeräten zu arbeiten. Webapplikationen werden mit der Programmiersprache *JavaScript* programmiert welche *HMTL* so manipuliert um visuelle Ergebnisse im Web Browser zu erzielen.

In den letzten Jahren hat sich die Nutzung nativer mobiler Applikationen im Gegensatz zu mobilen Webapplikation vergrößert. In Sachen Performance, Verlässlichkeit und User Interaktion sind Webb Apps immer noch schwächer. Oft sehen sich Unternehmen und Entwickler dazu gezwungen native mobile Apps augfrund der Limitierungen der Web Apps zu entwickeln. In Sachen Reichweite hingegen sind Web Apps stärker als native Apps. Native Applikationen nutzen auch mehr Ressourcen und benötigen mehr Zeit zum entwickeln, genau wie das Warten und Verbreiten auf spezfische Plattformen. Vor allem der Prozess zum Veröffentlichen einer nativen App, zum Beispiel auf dem Android Play Store oder dem App Store von Apple ist ein langwieriger Prozess. Beide Marktplätze haben Design und Datenschutzrichtlinien die eingehalten werden müssen. Ein weiterer Nachteil nativer Applikationen sind die Schritte bis zur eigentlichen Nutzuung. Speicherplatz überprfüfen, herunterladen, installieren und dann letztendlich benutzen. Eine Studie veröffentlichte, dass eine App ca. 20% der Kunden durch die Schritte vom ersten Kontakt des Users mit der App und der Beginn der Nutzung verliert[22].

Beide Plattformen haben ihre Vor- und Nachteile, daher gibt es ein Bedürfniss eine Plattform zu entwickeln welche beide Welten vereint. Eine *Progressive Web App* ist diese Plattform. Sie nutzt die besten Vorzüge einer Webapplikation und einer nativen Applikation. Zum ersten Mal 2016 wurde das Konzept in San Francisco bei der Google I/O Konferenz vorgestellt[23]. Es adressierte genau die Problembereiche der Web Apps. Eine *Progessive Web App* auch *PWA* genannt kann auf allen Plattformen genutzt werden, sieht aus wie eine native App und reduziert die Kosten für die Entwicklung nativer Applikationen.

Progressive Web App Applikationen sind keine neue Technologie oder Framework sondern eher eine Art beste Praktiken des Webs um dem Nutzer das gleiche Gefühl einer nativen App zu geben. Hauptkomponenten sind Web App Manifest, Push Benachrichtigungen und *Service Worker*. PWA's können mit einem Klick auf dem *Home screen* installiert werden. Ein Nutzer bekommt die volle Erfahrung einer nativen Applikation mit Vollbildschirm. Der wohl wichtigste Aspekt ist, dass eine *PWA* auch fffline funktioniert. Folgende Punkte indentifiziert *Google* als eine *PWA*:

- **Progressive** - Eine PWA funktioniert für alle Nutzer in allen Internetbrowsern was durch die Design-Philosophie ***Progressive Enhancement***möglich gemacht wird
- **Responsive** - Das *User Interface* passt in allen möglichen Formaten von Endgeräten, sei es ein Tablet, Desktop oder sonstiges
- **Discoverable** - Über das W3C *web app manifest* und dem *Service Worker* Registrierungsgültigkeitsbereich, identifizieren Suchmaschinen eine *PWA* als Applikation. Das erhöht die Wahrscheinlichkeit die App über eine Suchmaschine zu finden
- **App-like** - Über das Designkonzept der App-Shell Architektur wird die Applikationsfunktionalität von dem Inhalt getrennt und nutzt so wenig wie mögliche Seitenaktualisierungen um eine native App Erfahrung zu realisieren
- **Connectivity independent** - Der *Service Worker* macht es möglich die App Offline auch mit fehlender Konnektivität zum Internet, funktionieren zu lassen
- **Fresh** - Immer auf dem aktuellsten Stand dank dem *Service Worker* Update Prozess
- **Safe** - *HTTP* und *TLS* sind erforderliche damit eine *PWA* funktioniert und schützt davor, dass Inhalte nicht modifiziert werden können
- **Re-engagable** - wie bei nativen Applikationen und bringt Nutzer dazu sich wieder an der PWA zu betätigen

- **Installable** - Erlaubt es User, die App auf dem den *Home screen* zu installieren und zu starten ohne die Mühe eines App Stores
- **Linkable** - Teilen der Applikation erfolgt einfach über die *URL*.[24]

3.3 Single-Page-App

Webapplikationen werden grundsätzlich in 2 Kategorien unterteilt.

- *Single-Page Application*
- *Multi-Page Application*

Mit dem Beginn des Webs begann auch die *Multi-Page Application*. Jede Seitennavigation oder Mausklick in der Applikation führt zu einer Netzwerkanfrage an den Server der mit der entsprechenden Seite antwortet. Diese Applikationen haben mehrere Ebenen an *User interface* und können sehr groß werden. Solche Webseiten sind meistens statische Webseiten. Vorteile einer *Multi-Page Application*:

- Guter Ansatz wenn der Nutzer einen visuellen Wegweiser braucht um zu wissen wohin er auf der Webseite gelangen kann
- sehr *SEO* freundliche, es können auf jeder Seite Schlüsselwörter, *meta tags* und Bilder eingebettet werden
- Aufschlussreiche Datenanalyse der Webseitenbesucher. Jede Seite kann beispielsweise Informationen über Aufenthaltsdauer und Häufigkeit des Besuchs der Seite liefern.

Nachteile von *Multi-Page Application*:

- Aufwendige Entwicklung - Viele Seiten bedeutet viele Funktionen was zu größerem Aufwand führt. Die Entwicklungskosten steigen mit Anzahl der Seiten
- Performanceeinußen durch Belastung des Servers - Der Großteil der Logik geschieht im Server und die Seiten werden ständig neu geladen. Bei großen Nuterzahlen der Webseite benötigt man eine komplexe und perfomante serverseitige Infrakstruktur, was sich nicht jedes Unternehmen, geschweige denn Privatperson leisten kann.
- Kostspielige Wartung - Web App Werkzeuge sind schneller veraltet als man denkt. Parallel dazu werden ständig neue Programmiersprachen, *frameworks* und Bibliotheken eingeführt oder zumindest neue Versionen davon. Die Wahl des passenden Technologiestacks kann hierbei geschäftsentscheidend sein.

*Single-Page Application*arbeiten im Browser und benötigen keine Seitenaktualisierungen um zu funktionieren. Die bekanntesten Webseiten heutzutage nutzen dieses Konzept. Beispiele wäre hier, *Gmail*, *Google Maps*, *Facebook* und noch viele mehr. Single-Page Applications haben das Hauptziel eine hervorragende *User Experience* zu schaffen. Die Applikation besteht aus nur aus einer Webseite und alle anderen Inhalte werden über *JavaScript* geladen. Die Seiten kommen nicht vom Server sondern werden im Internetbrowser berechnet und ausgegeben. Vorteile einer *Single-Page Application*sind:

- Sehr schnelle visuelle Antwortzeiten, *(HTML, CSS, JavaScript)* werden nur einmal über die Laufzeit der Applikation geladen. Nur die Daten müssen immer wieder geladen werden
- Es ist nicht nötig Quellcode zu schreiben der Seiten im Server generiert

- Entwicklung kann sofort auch ohne Server begonnen werden
- Der Quellcode kann gut in den Entwicklertools eines Browsers gedebugged werden
- Es ist einfacher mobile Applikationen zu entwickeln da der serverseitige Code für Web und native Applikationen genutzt werden kann
- Große Unternehmen entwickeln aktiv an verschieden *frameworks* für *Single-Page Application*
- Mittlerweile wird jede beliebige Programmiersprache unterstützt, was mit dem Projekt *Web Assembly*[25] möglich gemacht wurde, indem es eine beliebige Programmiersprache entgegen nimmt und diese in *JavaScript* kompiliert

Nachteile einer *Single-Page Application* sind:

- Initialer Download ist relativ langsam da die *frameworks* viel Speicher benötigen und der Download schließlich am Client stattfindet
- *JavaScript* muss aktiviert sein
- Geringere Sicherheit da viel Logik im Client behandelt wird

React.js

React.js ist ein quelloffenes Web *framework* von Facebook für *Single-Page Application* zum Entwicklen von Benutzeroberflächen. *React* ist sehr flexibel und kann für UI-Projekte mit jeder Größe und Plattform verwendet werden, einschließlich mobiler, Web- und Desktop-Entwicklungen. Entwickler verwenden *React* häufig für kleine und mittlere Projekte. Gleichermaßen nutzen viele weltbekannte Unternehmen *React* bei ihren Produkten, z. B. *Netflix*, *Dropbox*, *Khan Academy*, *Reddit*, *Imgur* und viele andere. Die Ergebnisse der jährlichen Umfrage von *Stack Overflow* liefern, dass *React.js* hinter *jQuery* das meist genutze Web *framework* für Entwickler überhaupt ist. *React* ist komponentenbasiert und die UI-Komponenten können in verschiedenen Projekten immer wieder verwendet werden können.

Ionic

Ionic ist ein quelloffenes *web framework* zu Erstellung von *Progressive Web Apps*, Desktop und native mobile Apps auf Basis von *HTML5, CSS* und *JavaScript/Typescript*. Das Ziel von *Ionic* ist es plattformübergreifende Benutzeroberflächen zu entwickeln. Das bedeutet, eine Code Basis funktioniert auf allen Plattformen wie *ios*, *Android*, *Windows* und *Blackberry*. *Ionic* ermöglicht es auch auf native Betriebssystemfunktionen zuzugreifen um das gleiche Gefühl wie bei einer nativen Applikation zu bekommen. Die entwickelten Webapplikationen können sogar über die App Stores der jeweiligen Plattform vertrieben werden.

3.4 Cloud-Computing

Cloud-Computing stellt IT-Ressourcen von Anwendungen bis zu Rechenzentren über das Internet bereit, meistens auf Basis nutzungsabhängiger Gebühren. Generell einfach als "die Cloud"bezeichnet. Angebot und Nutzung dieser IT-Ressourcen erfolgt in der Regel über eine Programmierschnittstelle *API* für Anwender über eine Webseite oder App. Das *National Institute of Standards and Technology* erwähnt 5 Charakteristiken die Cloud-Computing ausmachen:

- *On-demand self-service* - Ein User kann ohne Interaktion Leistungen bereitstellen lassen. Wie eine Art Selbstbedienung
- *Broad network access* - Leistungen sind über das Netzwerk erreichbar welche die Nutzung mit Endgeräten fördern
- *Resource-pooling* - Die Computerresourcen des Anbieters werden zusammengefasst um mehrere Nutzer bedarfsgerecht bedienen zu können. Der Nutzer hat keine Kontrolle darüber woher genau er die Resourcen bezieht
- *Rapid elasticy* - Aus Sicht des Nutzers scheinen die verfügbaren Computerresourcen unbegrenzt und können jederzeit in beliebiger Höhe angepasst werden.
- *Measured service* - Die Nutzung der Leistung kann überwacht und gesteuert werden um die Transparenz für den Nutzer wie auch den Anbieter gewährleisten zu können[26]

Die Dienstleistungen des Cloud-Computings sind folgende:

- *Infrastructure as a Service (IaaS)* - IaaS bietet Zugriff auf Anfrage zu grundlgenden Rechenressourcen wie physische und virtuelle Server, Netzwerke und Speicher über das Internet
- *Software as a Service (SaaS)* - auch bekannt als cloudbasierte Software oder Cloud-Anwendungen, ist eine Anwendungssoftware die in der Cloud zur Verfügung gestellt wird und auf die man über einen Webbrowser, Desktop oder eine API zugreift, um diese nutzen zu können. In den meisten Fällen zahlen Anwender eine monatliche oder jährliche Gebühr. Der Anwender muss sich nicht um Updates oder Datenverlust kümmern. Die Daten können nach einem Absturz des Gerätes oder Verlust der Daten auf dem Gerät, jederzeit wieder von der Cloud heruntergeladen/abgerufen werden. *SaaS* ist heute das primär bereitgestellte Modell für die meiste kommerzielle Software.
- *Platform as a Service (SaaS)* - Richtet sich in erster Linie an Entwickler und stellt diesen Hardware, Software, Infrastruktur und Entwicklertools zur Verfügung. Dabei fallen das kostenintensive und zeitraubende Warten der Systeme für die Entwickler weg.
- *Serverless Computing* - ist ein Modell das alle Verwaltungsaufgaben der serverseitigen Infrastrukutr an den Cloud-Provider auslagert, so dass sich Entwickler ganz auf den Code und die Geschäftslogik der Anwendung konzentrieren können

Hinzu kommen noch verschiedene Arten von Cloud-Computing

- *Public Cloud* - ist ein Angebot eines frei zugänglichen Providers, der seine Dienste offen über das Internet für jeden zugänglich macht. Bekannte Anbieter wie, Google Cloud, Amazon Web Services, Microsoft Azure oder IBM Cloud haben einen Kundenstamm der mehrere Millionen umfasst. Der globale Markt für Public Cloud Computin ist in den letzten Jahren rasant gewachsen und Analysten prognostizieren, dass sicher dieser Trend fortsetzen wird. Der Branchenanalyst Gartner sagt voraus, dass die weltweiten Public-Cloud-Umsätze 330 Milliarden US-Dollar bis Ende 2022 übersteigen werden[27].
- *Private Cloud* - Aus Gründen von Datenschutz und IT-Sicherheit ziehen es Unternehmen häufig vor, ihre IT-Dienste weiterhin selbst zu betreiben und ausschließlich ihren eigenen Mitarbeitern zugänglich zu machen. Werden diese in einer Weise angeboten, dass der Endnutzer

im Unternehmen cloud-typische Mehrwerte nutzen kann, wie beispielweise eine skalierbare IT-Infrastruktur oder installations- und wartungsfreie IT-Anwendungen, die über den Webbrowser in Anspruch genommen werden können, dann spricht man von einer *Private Cloud*. Häufig werden diese Mehrwerte aber in so bezeichneten IT-Infrastrukturen nicht oder nur teilweise erreicht.

- *Hybrid Clouds* - Mit *Hybrid Clouds* werden Mischformen dieser beiden Ansätze bezeichnet. So laufen bestimmte Services bei öffentlichen Anbietern über das Internet, während datenschutzkritische Anwendungen und Daten im Unternehmen betrieben und verarbeitet werden. Die Herausforderung liegt hier in der Trennung der Geschäftsprozesse in datenschutzkritische und -unkritische Workflows. Voraussetzung ist eine saubere und konsequente Klassifizierung der im Unternehmen vorhandenen und verarbeiteten Daten.

- *Community Clouds* - Einen eher seltenen Spezialfall stellen die so genannten *Community Clouds* dar. So werden Cloud-Infrastrukturen bezeichnet, die von mehreren Unternehmen – z. B. im Kontext eines Projekts – genutzt und bereitgestellt werden, um gemeinsam auf bestimmte Dienste zugreifen zu können. Diese sind jedoch nicht öffentlich verfügbar, sondern beschränken sich auf einen definierten Nutzerkreis.

MongoDB

MongoDB ist eine universelle, dokumentbasierte, verteilte Datenbank für die moderne Anwendungsentwicklung und die Cloud, die in puncto Produktivität höchsten Ansprüchen gerecht wird"[28]. Die Daten werden in einem *JSON*-ähnlichen Dokumentenformat gespeichert. *MongoDB* wirbt dafür, dass diese Herangehensweise als ***NoSQL-Datenbank***, aussagekräftiger und leistungsfähiger als eine Zeilen/Spalten-Modell macht. Bei der Implementierung eines Demonstrators in Kapitel 44 wird MongoDB als „Cloud-Datenbank“ genutzt. Eine „Cloud-Datenbank“ kann zwei verschiedene Formen haben: eine traditionelle Datenbank bzw. eine *NoSQL*-Datenbank, die in einer virtuellen Cloud-Maschine ausgeführt wird (eine Public-, Private- oder Hybrid-Cloud-Plattform), oder ein vollständig verwaltetes Datenbank-as-a-Service-Angebot (DBaaS) eines Cloud-Anbieters. Der Betrieb einer eigenen, selbst verwalteten Datenbank in einer Cloudumgebung unterscheidet sich eigentlich nicht vom Betrieb einer herkömmlichen Datenbank. Cloud-SaaS-Angebote sind hingegen das natürliche Datenbank-Äquivalent zu Software-as-a-Service (SaaS): Man zahlt nach Bedarf und nur für das, was man nutzt, und überlässt dem System alle Details der Bereitstellung und Skalierung entsprechend der Nachfrage und zur Leistungsoptimierung.

Native Cloud-Datenbanken werden immer beliebter – ganz egal, ob ein Team bereits Software mit einer Cloudinfrastruktur entwickelt oder veraltete Anwendungen in die Cloud migriert.

Moderne Database-as-a-Service-Plattformen ermöglichen über einheitliche *API's* und Treiber einen einfachen und kontrollierten Zugriff von Cloud- und Nicht-Cloud-Systemen und vereinfachen so den Zugriff auf wichtige Ressourcen. Insbesondere Microservice-Architekturen profitieren von zentralisierten und leicht zugänglichen Datenbankressourcen, da viele Anwendungen auf Daten zugreifen und diese gemeinsam nutzen müssen.

Während in vielen Fällen Cloud-*SQL*-Datenbanken zum Einsatz kommen, verbessert die Flexibilität moderner Cloud-*NoSQL*-Datenbanken die Agilität im Datenmanagement und in der Softwareentwicklung dramatisch. Es gibt keine Ausfallzeiten für Systemupgrades, die Neuausrichtung von Clustern oder die Bereitstellung von schnellerer Hardware oder gar Änderungen an Schemata und Strukturen.

3.5 Browsertechnologien

Bei der Entwicklung von Webapplikationen gibt es einige Browsertechnologien welche die Applikationsentwicklung fördern. Zum einen stellt sich zu Beginn die Frage, wie die Daten am Client gespeichert werden sollen. Die gängigen Internetbrowser bieten alle folgende Speichermechanismen.

- **Cookies**
- **Session Storage**
- **Local Storage**
- **Web SQL**
- **IndexedDB**

Cookies

Das Web nutzt intensive das *HTTP/S* Protokoll um Nachrichten über das Netzwerk zu versenden. Da *HTTP/S* ein zustandloses Protokoll ist und somit keine Verbindungsübergreifende Informationen speichert braucht es einen Weg Informationen über eine Verbindung bzw. Sitzung hinweg zu speichern. Das ist mit Cookies möglich. Sie können über mehrere Sitzungen Informationen für eine bestimmte Webseite speichern. *Cookies* haben auch ein Gültigkeitsdatum und verfallen je nach Einstellung. Möchte man zum Beispiel eine User-ID für einen Nutzer speichern oder generell Authentifikationsinformationen werden diese über HTTP bei jeder Anfrage mitgesendet. Im *HTTP header* darf jedoch das `HttpOnly` *flag* nicht gesetzt werden. Ist es gesetzt können Cookies nicht verwendet werden. Das Speicherlimit eines *Cookie* beläuft sich auf 4 Kilobyte.

Local Storage

Local Storage ist ein teil der *Web Storage API* und ein synchroner Schlüssel-Werte Paar Speicher im Browser. *Local Storage* speichert Informationen ebenso über Sitzungen hinweg. Informationen aus *Local Storage* können nur über *JavaScript* gehandhabt werden. Der Zugriff auf *Local Storage* läuft synchron, das bedeutet, dass bei einer Schreiboperation der *DOM* und somit die Webseite geblockt wird. Das Speicherlimit von *Local Storage* liegt bei 5 Megabyte. *Local Storage* wird häufig für Authentifikationsinformationen wie zum Beispiel *Json Web Token* verwendet.

Session Storage

*Session Storage*ist identisch zu *Local Storage* und auch teil der *Web Storage API* bis auf die Tatsache, dass Informationen nur so lange gespeichert werden wie auch der Browser Tab geöffnet ist. Das Öffnen eines neuen Browser Tab erzeugt eine neue Sitzung.

Cache

Der *Cache* im Browser ist ein Puffer-Speicher mit dem bereits abgerufene Ressourcen auf dem Computer des Benutzers lokal als Kopie aufbewahrt werden können. Im *Cache* wird jeder Internetadresse *(URL)* der zuletzt bekanngewordene Inhalt zugeordnet. Bei jedem Webseitenaufruf wird zuerst im *Cache* überprüft ob diese bereits vorhanden ist. Somit muss keine Netzwerkanfrage an den Server erfolgen und Antwortzeiten stark reduziert werden. Hingegen kann es aber passieren, dass im *Cache* gespeicherte Daten veraltet sein können wenn Aktualisierungen der Webseite erfolgt sind. Dann

kann es zu einem der härtesten Problemen in der Informatik kommen, der *cache invalidation*, der Ungültigkeit des *Cache*. Daher ist vorsicht geboten wenn Programmierarbeiten mit dem Browser Cache stattfinden.

Der Browser Cache wird z.b. sehr stark von *Apollo GraphQL* genutzt indem dort Applikationsdaten gespeichert werden können. Generell fungiert der Browser *Cache* dazu, Anfrage und Antwortpaare zu speicher. Für mehr Informationen stellt *MDN Web Docs* eine umfangreiche Dokumentation der *Cache API* zusammen, um mit dem *Cache* zu interagieren[29].

Web SQL

Web SQL ist eine auf *SQLite* basierende Datenbanktechnologie die mit JavaScript abgefragt werden kann. Momentan wird die aktive Pflege nicht mehr vorrangestrieben . *WebSQL* war auf dem Weg zu einer W3C *Recommendation* zu werden, aber die Arbeit an der Spezifikation wurde gestoppt. Alle an der Technik interessierten Browserhersteller hätten dieselbe Implementierung, *SQLite*, verwendet. Um aber die Standardisierung voranzutreiben, bedürfe es mehrerer, unabhängiger Implementierungen der Spezifikation[30].

IndexedDB

IndexedDB ist eine asynchrone clientseitige Datenbank wobei jeder Eintrag ein indiziertes Schlüssel-Werte Paar ist. Während Session Storage und Local Storage nur eine geringe Anzahl an Daten speichern kann, eignet sich *IndexedDB* für große Datenmengen. Der maximale Browserspeicher ist dynamnisch und basiert auf dem Speicher der Festplatte des Computers. Das globale Limit ist 50% des Festplattenspeichers. *IndexedDB* blockiert nicht den *DOM* und somit nicht die Applikation bei Lese oder Schreiboperationen. Für komplexere Webapplikationen bei der große Datenmengen gehandhabt werden, ist *IndexedDB* die einzig sinnvolle Wahl.

Service Worker

Ein *Service Worker* ist ein asybchroner ereignisgesteuerter Hintergrundprozess, welcher sich zwischen einer Webapplikation, dem Browser und dem Netzwerk befindet. Ein *Service Worker* kommt in Form einer *JavaScript*-Datei und kontrolliert die Web Applikation, in dem es Netzwerkanfragen abfangen kann, Ressourcen zwischenspeichern und Seitgennavigationen modifizieren kann. Der *Service Worker* agiert als separater Prozess der Applikation, hat keinen Zugriff auf den *DOM* und blockiert somit nicht den *DOM*. Die Intention hinter einem *Service Worker* ist es, eine effektive Offline-Erfahrung zu kreieren, um angemessene Aktionen bei fehlender Internetverbindung auszuführen. Einsatzgebiete eines *Service Worker* sind unter anderem, Hintergrunddatensynchronisation, Antworten auf Anfragen anderer Quellen, Reagieren auf Push-Benachrichtigungen, Vorabrufen von Ressourcen und viele mehr. Hauptsächlich liegt der Fokus eines *Service Worker* auf Performance und Offline-Funktionalitäten. Es können Netzwerkanfragen in bestimmten Situationen vermieden werden und vor allem die Webapplikation kann bei komplexen Berechnungen entlastet werden. Der *Service Worker* wird häufig in einer *Progressive Web Application* genutzt.

Workbox

Workbox ist eine quelloffene Sammlung an *JavaScript*-Bibliotheken von *Google* für *Progressive Web Apps*. Sie vereinfacht den Umgang mit einem *Service Worker* und bietet zahlreiche Hilfsfunktionen um einen produktionsfertigen *Service Worker* zu entwicklen. Beim Entwickeln einer *React-*

Applikation über *Create React App*[31] wird *workbox* mitgeliefert und in nur wenigen Schritten ist eine komplette Webapplikation für den Offlinezugriff bereit.

3.6 Server/Client

Das Client-Server-Modell ist ein Architekturkonzept zur Verteilung von Diensten und Aufgaben in einem Netzwerk. Dienste werden von Servern bereitgestellt und können von Clients genutzt werden. Mehrere Clients greifen generell auf einen Server zu. Aufgaben eines Servers ist es einen bestimmten Dienst lokal oder über das Netzwerk zur Verfügung zu stellen. Der Server nimmt Anfragen von Clients die einen Dienst anfordern an und schicken den Dienst in der Anwortnachricht zurück.

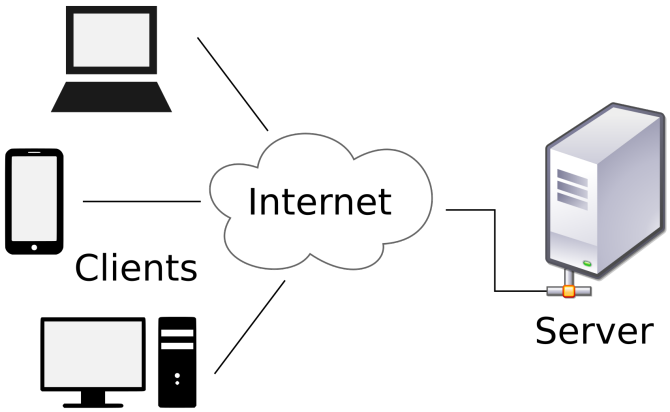

Abbildung 8: *Client-Server-Modell*

Oft sind auf einem Hardware-System mehrere Server-Programme gleichzeitig installiert und aktiv. Deshalb werden die Rechner selbst oft als Server bezeichnet. Rechner wie Desktop-Computer, die hauptsächlich Clientfunktionen ausführen, werden analog dazu Clients genannt. Die Begriffe Server und Client sind nach dem Client-Server-Modell jedoch nicht für physische Hardwaresysteme zu verwenden, sondern beziehen sich auf die logische Funktion eines Programms. Auf einem Rechner können sowohl Client- als auch Server-Programme implementiert sein. Vor allem in TCP/IP-Netzwerken werden viele Anwendungen auf Basis des Client-Server-Modells betrieben. Typische Anwendungsbeispiele sind der Zugriff auf Webseiten über *HTTP/S* oder die Abwicklung des E-Mail Verkehrs. Von Vorteil bei diesem Konzept ist die zentrale Verwaltung der Dienst und Ressourcen sowie die zentrale Datenhaltung. Der Nachteil hingegen besagt das Wort ßentralßchon. Alles wird zentral verwaltet was bei einem Ausfall des Servers für einen kompletten Dienstausfall sorgt. Man ist abhängig von einer Instanz, man verlässt sich nur auf eine Quelle. Genau so wie die Server mit genügend Bandbreite ausgestattet sein müssen, um alle Clients bedienen zu können. Der Betrieb und die Bereitstellung eines Servers ist mit erhöhtem zeitlichen und finanziellen Aufwand verbunden und erfordert spezifisches Know-How. Doch genau diese genannten Nachteile können mit dieser vorgestellten wissenschaftlichen Arbeit sehr minimiert werden.

TypeScript

TypeScript ist eine quelloffene Programmiersprache die auf *JavaScript* basiert und diese, um statische Typdefinitionen erweitert. Gültiger *JavaScript* Code ist also auch gültiger *TypeScript* Code. *JavaScript* und *TypeScript* werden hauptsächlich genutzt um mit dem DOM zu interagieren und *HMTL* so zu manipulieren, um visuelle Ergebnisse in Form einer Benutzeroberfläche zu präsentieren. Seit 8 Jahren ist *JavaScript* unangefochten die am meist genutzte Programmiersprache[32]. Im Gegenzug ist jedoch bei einer Umfrage für Entwickler der beliebtesten Programmiersprachen, *TypeScript* auf Nummer 2 und *JavaScript* auf Platz 10. *TypeScript* gewinnt von Jahr zu Jahr an Beliebtheit[33]. Der Quellcode kann mit *TypeScript* besser strukturiert und dokumentiert werden, was vor allem bei sehr komplexen Anwendungen hilfreich ist.

Node.js

Node.js ist eine quelloffene plattformübergreifende und serverseitige *JavaScript* Umgebung welche in der *V8 Engine* läuft und *JavaScript* ausserhalb eines Internetbrowsers ausführt. Mit Einführung von *Node.js* wurde es zum ersten Mal möglich *JavaScript* nicht nur für die clientseitige Entwicklung zu nutzen sondern auch serverseitig. Möglich gemacht wurde das durch Ryan Dahl der es geschafft hat die *V8 Engine* aus dem Chrome Browser zu extrahieren und somit nicht mehr an den Browser gekoppelt zu sein. *Node.js* verfolgt den Ansatz mit nur einer Programmiersprache *(JavaScript)*-*Fullstack*-Applikationen zu entwickeln und somit sowohl serverseitig als auch clientseitig nur eine Programmiersprache zu benutzen.

3.7 Apollo GraphQL

GrapQL ist eine quelloffene Datenabfragesprache und Serverumgebung für APIs, die den Clients genau die Daten zurückgeben die sie auch angefragt haben. *GrapQL* ist ein neuer *API*-Standart der eine effizientere, mächtigere und flexiblere Alternative zu *REST* zur Verfügung stellt. Entwickelt von *Facebook* und aktuell von einer riesigen Community an Unternehmen und Entwicklern aus der ganzen Welt. *GraphQL* nutzt das Konzept des *decalarative data fetching* in dem ein Nutzer genau die Daten empfängt die er auch angefragt hat. Anstatt meherer Server-Endpunkte, welche eine festgelegte Strukur zurückgeben, hat *GrapQL* nur einen Endpunkt und gibt die Struktur an Daten zurück die der Client haben möchte. Folgende Illustrationen zeigen das Konzept und die Alternative zur herkömmlichen *REST*-Architektur:

Abbildung 9: *REST*[34]

Das Beispiel soll ein soziales Netzwerk darstellen. Wie zu sehen ist, gibt es hier 3 Endpunkte. `/users/id` gibt die Userdaten eines Users zurück, `/user/<id>/post` gibt alle Posts eines Users zurück und `/user/<id>/followers` gibt eine Liste aller Personen die einem folgen zurück. Mit *REST* muss also für jeden Endpunkt eine Netzwerkanfrage gesendet werden. Zusätzlich dazu werden auch mehr Informationen als benötigt zurückgesendet. Was ist wenn man den Endpunkt `/users/<id>/followers` anfragt aber nur die id's aller Followers haben möchte ? Das ist mit *REST* schlichtweg nicht möglich. Man bekommt alle Userdaten einer Person zurückgesendet. *GraphQL* löst genau dieses Problem.

Mit *GraphQL* kann man dem *GraphQL* Enpunkt genau sagen was man haben möchte. In diesem Fall werden die Namen der letzten 3 Follower des Users Mary zurückgegeben.

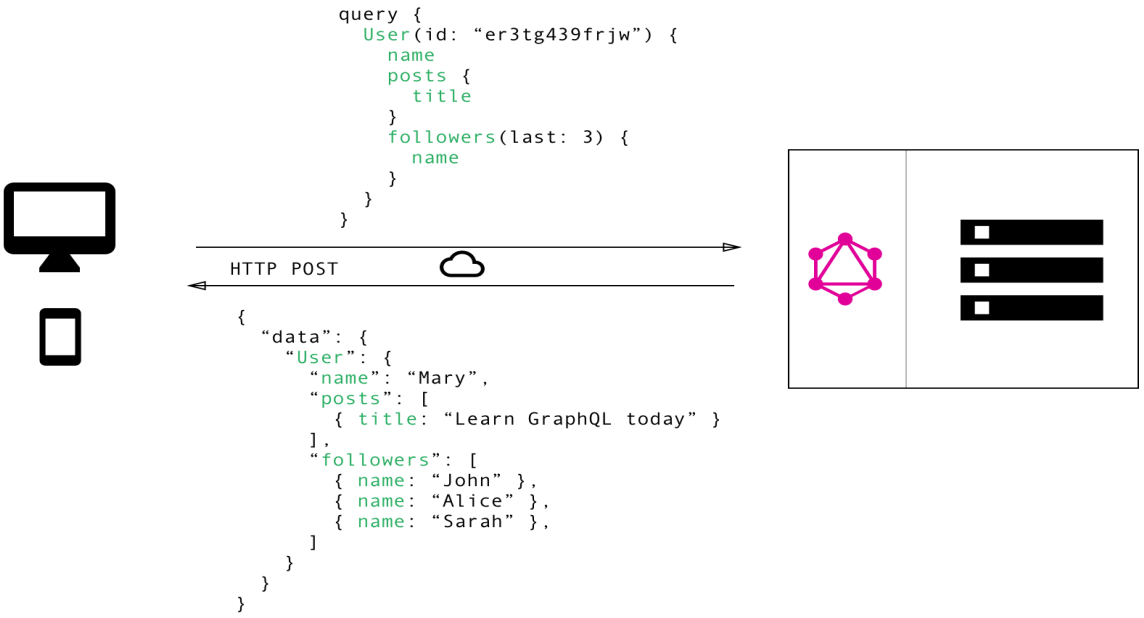

Abbildung 10: *GraphQL*[35]

Das geläufigste Problem von *REST* ist das *over* und *underfetching*. Beim *overfetching* lädt der Client mehr Daten als benötigt herunter. Beim *undefetching* sendet ein Endpunkt zu wenige Informationen an den Client zurück. Der Client muss zusätzliche Anfragen schicken um die benötigten Daten zu empfangen. Man stelle sich vor ein Client lädt eine Liste an Elementen herunter, muss dann aber noch eine Netzweranfrage schicken um die Informationen pro Element zu bekommen. Dieses Problem wird auch das n+1 Problem genannt[36].

GrapQL nutzt ein starkes Typensystem für die Möglichkeiten einer *API* zu definieren. Alle Typen einer*API* sind in form eines *schema* welches die *GraphQL Schema Definition Language (SDL)* nutzt. Das Schema ist die Verbindung zwischen einem Client und Server, um zu definieren wie ein Client auf die Daten zugreift. Ist das Schema einmal definiert, können server-und clientseitige Entwickler ohne weitere Kommunikation arbeiten, da beide sich der Struktur der Daten die über das Netzwek gesendet werden, bewusst sind.

Mit *API* ist offensichtlich das Augenmerkmal auf Performance gerichtet. *Facebook* musste sich nämlich eine Lösung überlegen wie sie mit den großen Mengen an Datenumgehen umgehen, welches ihr soziales Netzwerk täglich produziert.

3.7.1 Apollo GraphQL Client Abfragestrategien

Da in dieser wissenschaftlichen Arbeit ein Großteil der Arbeit clientseitig geleistet wird und *GraphQL* als Datenmanagement genutzt wird, ist es wichtig sich erst ein Mal mit den Möglichkeiten zu beschäftigen, wie Daten abgefragt werden können. *GrapQL* bietet eine Palette an Möglichkeiten wie Daten gehandhabt werden sollen. Mit den vielen Möglichkeiten Daten abfragen zu können, kann es schnell vorkommen, dass Entwickler suboptimale Anfragen erzeugen oder sich auf Standarteinstellungen verlassen, die eventuell unerwartete Ergebnisse liefern.

Im diesem Abschnitt werden die meisten Anfrageoptionen zusammengefasst die *GrapQL* anbietet und auf Entwurfsmuster abgebildet. Die folgenden Beispiele nutzen den puren *Apollo JavaScript*

Client zur Veranschaulichung der Entwurfsmuster. Natürlich können diese Optionen und Entwurfsmuster auch *framework*-spezifisch mit *Angular*, *Vue*, *React* oder anderen nativen Clients umgesetzt werden.

Apollo Cache

Eine Anforderung für bessere Effizienz ist es, *Apollo normalized cache* zu nutzen. Der *Cache* führt dazu die Anzahl der Anfragen an den Server zu minimieren, indem schon vorhandene Daten aus dem *Cache* benutzt werden. Dieser kann folgendermaßen konfiguriert werden:

```
const cache = new InMemoryCache();

const client = new ApolloClient({
  link: new HttpLink(),
  cache
});
```

Der *Cache* ist solange im Speicher wie auch die Applikation läuft. Bei App Neustarts wird er wieder entfernt und neu initialisiert. Um den *Cache*che im permament im Speicher zu halten gibt es die Bibliothek `apollo-cache-persist`[37].

Für Anfragen gibt es bei *Apollo Client* 2 Möglichkeiten. Die Funktion `query` und die Funktion `mutate`. Beiden Funktionen können wichtige Parameter übergeben werden, um mit dem Cache und dem Netzwerk zu interagieren.

```
client.query({
  ...
  fetchPolicy: ?,
  errorPolicy: ?
  ...
  });
```

```
client.mutate({
  ...
  opimisticResponse: ?,
  update: ?,
  refetchQueries: ?
  ...
  });
```

3.7.2 GraphQL Query Caching

Für die Funktion `client.query()` gibt mehrere Parametermöglichkeiten, wobei die wichtigsten diese sind:

- *fetchPolicy* - wie soll die Systemkomponente (z.b. eine React-Komponente) mit dem Cache interagieren

- *errorPolicy* - wie sollen Netz-und *GraphQL*-Fehler behandelt werden.

Die Standarteinstellung für `fetch-policy` ist `client.query(fetchPolicy: 'cache-first')`. Folgende Optionen akzeptiert `fetch-policy`:

- *cache-first* - Mit dieser Einstellung sendet Apollo eine Netzwerkanfrage aus und nutzt den *Cache* für weitere Anfragen. Das bedeutet, dass wenn sich Daten am Server verändern, ändert das nichts im *Cache*. Diese Option ist optimal wenn eine große Anzahl an Daten geladen werden welche sich selten verändern. Beispielsweise bei einer Applikation mit allen Namen aller Städte aus einem Land. Diese Daten werden einmal geladen und änder sich nicht mehr. Die Daten sind dann für die komplette Laufzeit der Applikation verfügbar. Bei jeder weiteren Leseoperation kann dann aus dem Cache gelesen werden ohne eine Netzweranfrage an den Server zu senden.

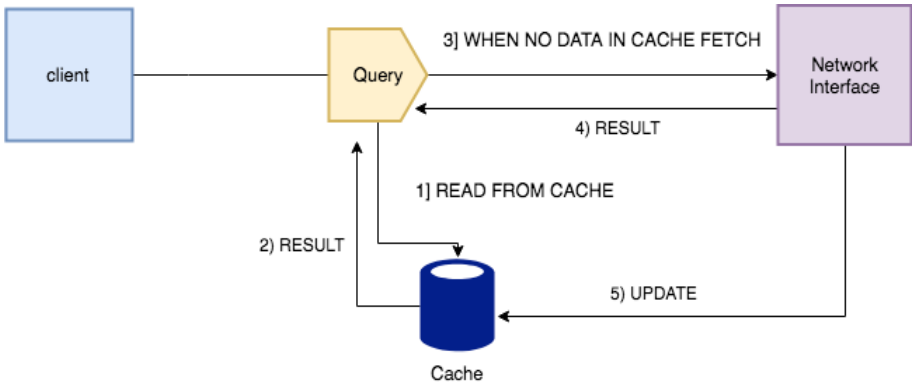

Abbildung 11: *fetchPolicy: 'cache-first'*

- *cache-and-network* - Diese Option gibt die Antwort direkt aus dem *Cache*, falls vorhanden und sendet dazu noch eine Anfrage an den Server. Sobald die Anntwort vom Server zurückkommt, wird der *Cache* damit überschrieben. Egal ob die Daten im *Cache* gefunden wurden oder nicht wird immer eine Anfrage an den Server gesendet. Diese Einstellung eignet sich für Nutzer die eine schnelle Anwort erhalten wollen, während die Daten in Konsistenz mit dem Server gehalten werden, jedoch auf Kosten zusätzlicher Netzwerkanfragen. Weiter eignet sich diese Option in den meisten Fällen bei denen man sich leisten kann, mit jeder Anfrage den Server zu erreichen.

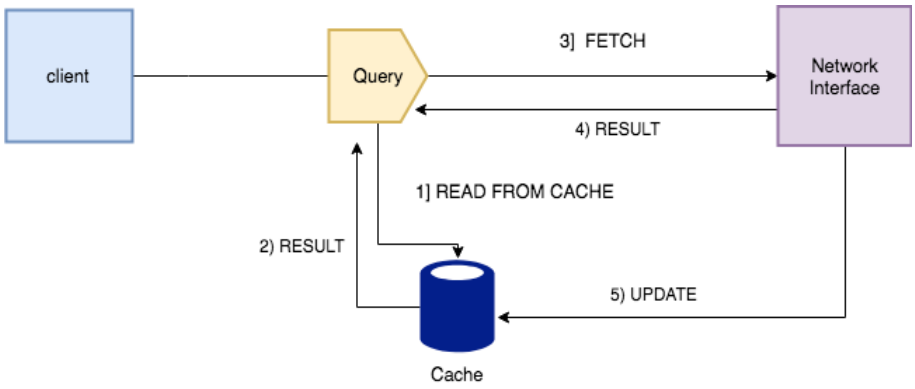

Abbildung 12: *fetchPolicy: 'cache-and-network'*

- *network-only* - *network-only* sendet immer eine Anfrage an den Server und aktualisiert den Cache. Der Cache ist damit also immer auf dem neuesten Stand und bekommt immer neue Daten vom Server. Der Cache wird mit dieser Option niemals ausgelesen. Der Nutzer sieht erst die Ergebnisse wenn die Daten vom Server zurückkommen.

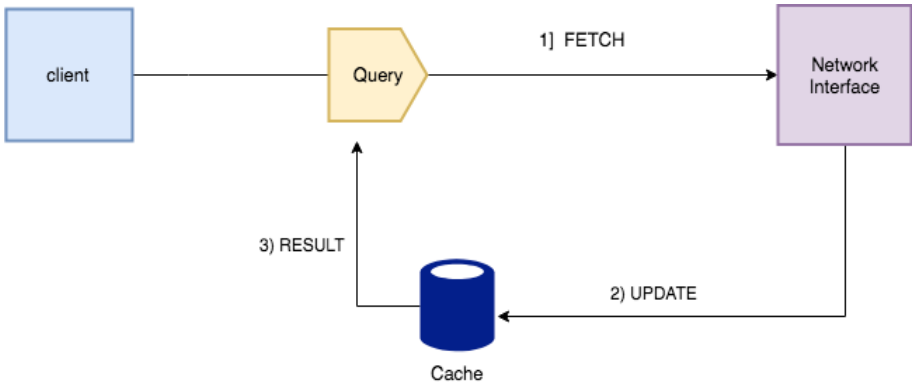

Abbildung 13: *fetchPolicy: 'network-only'*

- *cache-only* - Bei dieser Abfragestrategie werden immer nur Daten aus dem Cache gelesen. Netzwerkanfragen werden nie ausgesendet. Fall die Daten nicht im Cache gefunden ensteht eine Fehlermeldung. Mit dieser Einstellung gelangt man schnell zu Ergebnissen mit der Prämisse Dateninkonsitenz zum Server zu haben.

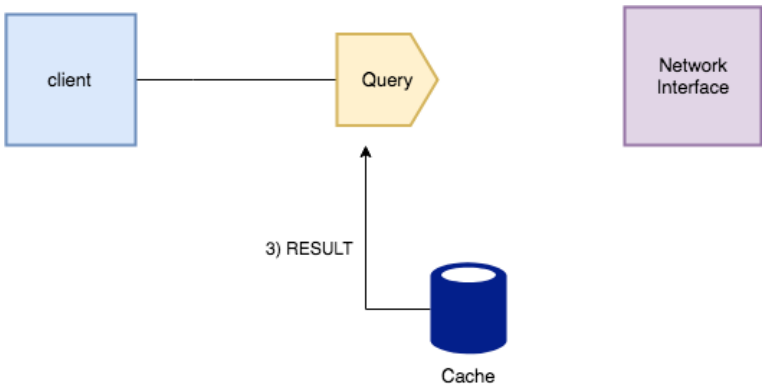

Abbildung 14: *fetchPolicy: 'cache-only'*

Mit diesem Wissen kann man sich jetzt dem Thema der Cachemanipulation widmen.

3.7.3 Cache Entwurfsmuster

Ein inkorrekter konfigurierter *Cache* kann Nutzern ungültige Daten liefern, was zu kritischen Fehlern führen kann. Bei Entwicklung einer Applikation in der ein *Cache* genutzt wird, muss ein Entwickler zusätzliche Vorrkehrungen treffen. Das Ziel ist es dabei zu jederzeit immer die aktuellsten Daten bereitzustellen.

REST-Ansatz NetworkOnly

```
if(ONLINE)
    nur Netzwerk anfragen
else
    nur Cache anfragen
```

Dieses Entwurfsmuster stellt sicher, dass immer die aktuellsten Daten vorhanden sind. Clients senden immer eine Netzwerkanfrage, wenn eine Internetverbindung besteht. Schlägt die Anfrage fehl oder es besteht keine Internetverbindung, wird auf den *Cache* zugegriffen, welcher dank der *network only* Strategie auf dem neuesten Stand ist. Dieses Entwurfsmuster eignet sich in Situationen in denen man immer die aktuellsten Daten zu Verfügung haben will, die Applikation auch Offline funktioniert und man fortgechrittene Themen wie *Subscriptions* vermeiden will. Der Nachteil dieses Ansatz ist die intesive Netzwerkbefragung und zusätzlich dazu sieht ein Client die Ergebnisse nicht unmittelbar.

Wiederauffüllung des Cache mit Subscription

```
beim Start: nur Netzwerk anfragen
später: Daten abonnieren
nur Cache anfragen
```

In Applikationen in denen Netzwerkanfragen an den Server minimiert werden sollen und *Subscriptions* Anwendung finden, ist die Strategy *CacheRefill* ein guter Ansatz. Intention ist es den *Cache* als Hauptdatenquelle zu betrachen. Danach können Entwickler alternative *Cache*-Wiederauffülungsstrategien wie, *Pooling* oder *Subscriptions* konfigurieren.

Abgesehen von *Subcriptions* kommt es trotzdem vor, dass Clients Offline gehen. Diese Strategy funktioniert in Anwendungen wie Chat-Programme in denen oft der *Cache* für gute *User Experience* genutzt wird. *Subscriptions* würden aber nur beispielsweise bei einem geöffneten Chatfenster funktionieren, deshalb wird *network only* immer noch benötigt.

Wiedernutzung des Cache

Wenn man mit einer ***Master-Detail-Ansicht*** arbeitet können Daten immer für die Master-Ansicht vorgeladen und der Cache für die Detailansicht wiederverwendet werden. In Spezialfällen können dann noch extra Details vom Netzwerk angefragt werden. Dieses Entwurfsmuster wird als eine einfache Abfrageoptimierung angesehen.

Optimistic fetch

Optimistic fetch ist eine Variante der *Online network only* Strategy.

```
nur Cache anfragen
if(ONLINE)
    nur Netzwerk anfragen
```

Optimistic fetch versucht immer als Erstes die Antwort einer Abfrage vom *Cache* zu lesen, aber falls eine Internetverbindung bestehen sollte, wird der *Cache* im Hintergrund, mit der Antwort vom Server überschrieben (In React würde zu dieser Zeit die Komponente noch einmal aktualisieren). Falls die angeforderten Daten sich im lokalen Cache befinden, werden mit diesem Konzept alle UI-spezifischen Anfragen funktionieren. Ebenfalls wird der *Cache* stetig mit den Daten vom Server konsistent gehalten, wenn dieser erreichbar ist.

Direkte Cachemanipulation

Apollo Client erlaubt es direkt mit dem Cache zu interagieren, um diesen als Datenbank zu nutzen. In den meisten Fällen jedoch ist dies dank *refetchQueries* nicht nötig, das die betroffenen Abfragen aktualisiert. *refetchQueries* benötigt jedoch viel Aufwand und kann sehr spezifisch werden. Aus diesem Grund bietet Apollo und andere *frameworks* Cacheaktualisierungsfunktionen mit denen direkt auf den Cache zugegriffen werden kann. Das Projekt *Offix*, widmet sich der Implementierung genau dieser Strategien.[38]

Offix

Offix Datastore ist ein voll kompatibler *Apollo GraphQL*-Client für *JavaScript/TypeScript* Anwendungen. Es erweitert den einen *Apollo GraphQL*-Client um Offlinemöglichkeiten mit Echtzeitsynchronisation und Datenspeicherung. *Offix* speichert alle Daten lokal und stellt einen *GraphQL*-Replikationsmechanismus zu einem *GraphQL*-Server zur Verfügung. Damit sind die Daten sowohl offline als auch online verfügbar. Alle Datensätze werden lokal gespeichert und bei Veränderungen an Datensätzen bei fehlender Konnektivität, werden diese in eine Warteschlange gereiht bis

die Konnektivität wieder hergestellt ist. *Offix* verfügt über Synchronisation mit *GraphQL CRUD*-kompatiblen-Servern, was eine optimale Anbidung an *Graphback* ist.

Graphback

Graphback ist ein Framework für einen *Apollo GraphQL*-Server, welches aus Datenbankmodellen eine produktionsfertige *API* generiert. *Graphback* nutzt *GraphQL* und die *GraphQLCRUD Spec*[39] um einen *CRUD*-fertigen Server mit Datensynchronisationsmöglichkeiten zu erstellen. Anhand der *GraphQL* Typen, repräsentiert in der GraphQL-Schema-Definitionssprache, welche auf dem Datenbankmodell basiert, generiert *Graphback* die *GraphQL-API* und die clientseitigen Dokumente.

4 Umsetzung und Implementierung eines Demonstrators

Für die Implementierung des vorgestellten Demonstrators wird ein fortgeschrittenes Verständnis von *Apollo GraphQL*, *TypeScript* und Datenbanktechnologien vorrausgesetzt.

Dieses Kapitel behandelt eine gründliche Erklärung einer quelloffenen Applikation von *aerogear*, mit einigen Abänderungen, umfunktioniert zu einer *Progressive Web App*. Der Link zum Quellcode befindet sich im Anhang[40].

Der Demonstrator ist eine ToDo-App, bei der man eine Aufgabe erstellen, bearbeiten, löschen und als fertig markieren kann. Konnte ein Eintrag nicht über das Netzwerk gesendet werden, wird die fehlgeschlagene Operation im Browserspeicher gehalten, in der Komponente Offline Changes angezeigt und bei Wiedererlangung der Internetverbindung an den Server gesendet. Jeder Eintrag wird mit einem Zeitstempel, der Uhrzeit der Erstellung oder Bearbeitung versehen. Besteht keine Internetverbindung, wechselt die Farbe des grün hinterlegten Online-Symbol, zu rot. Genauso wechselt die Farbe wieder auf bei wiederkehrender Verbindung auf grün und die fehgeschlagene Operation kann über das Netzwerk gesendet werden.

4.1 Anforderungen an die Applikation

Der Demonstrator sollte folgende Anforderungen haben:

- **Responsiveness** - trotz handling großer Datenmengen, sollte die Benutzeroberfläche über akzeptable Antwortzeiten verfügen
- **Datenverfügbarkeit** - Applikationsdaten sollten zu jederzeit verfügbar und möglichst aktuell sein
- **Datenkonsistenz** - Möglichst alle Daten die in der Datenbank gespeichert sind, sollten in Synchronisation mit dem Datenspeicher im Client sein

Der User soll folgende Möglichkeiten haben:

- sich die Applikation auf das mobile Gerät installieren zu lassen
- die Applikation öffnen zu können ohne zur *URL* des Web Browsers navigieren zu müssen
- die Applikationsdaten fffline lesen und bearbeiten zu können
- Benachrichtigungen bei Konfliktsituationen zu erhalten

4.2 Server

Die Initialisierung und der Start des Servers folgen in der Datei `/server/index.ts`. In dieser Demo werden 10 000 Datensätze in die *MongoDB* Datenbank geladen. Authentifikation ist momentan in dieser Demo nicht konfiguriert.

Für den Aufbau des Server wird das *framework Graphback* genutzt. Annotationen sind der Kern der *Graphback* Datenbankmodell Definitionssyntax. Sie weisen *Graphback* darauf hin ob ein *Schema type*, Teil des Datenmodells, sprich eine Tabelle in einer relationalen Datenbank oder eine *collection* in *MongoDB* ist.

```
"""
@model
@datasync
"""
type Task {
  """@id"""
  id: ObjectID!
  ...
}
```

Um Datensynchronisationsmöglichkeiten zu aktivieren, benötigt man zusätzlich die Annotation `@datasync` in der Datei der *GraphQL schema types*.

Die Annotation `@datasync` sorgt dafür Datensynchronisation zu ermöglichen, die dem Schema bestimmte Informationen hinzufügt.

```
"""
@model
@datasync
"""
```

Wenn ein *GraphQL*-Client sich in einer Umgebung mit schlechter Internetverbindung befindet, sind die Daten die im *Cache* zwischengespeichert werden sehr schnell nicht mehr auf dem neuesten Stand. Es ist daher zwingend erforderlich eine Möglichkeit zu finden, über Datenänderungen informiert zu werden, um bei wiederhergestellter Verbindung auf dem neuesten Stand zu sein. Diesen Mechanismus bietet *Graphback* mit `@datasync` an.

`@datasync` aktiviert die Funktionalität der *Delta Query*. *Delta Query* bietet die Möglichkeit an, Daten abzufragen von denen der Client in der Zeitspanne in der er offline war keine Kenntniss hat. Dadurch können zu einem späteren Zeitpunkt Konflikte entdeckt und aufgelöst werden. Sie erlauben einem Nutzer Anfragen zu senden, die in Form einer Sequenz, Zeitstempel oder einer Serie von Datumsangaben zurückgegeben werden. Diese partiellen Daten werden zum Einen, in Daten die schon bekannt sind separiert und zum Anderen in ein sogenanntes Delta, welches noch zum Client gesendet werden muss. Eine *Delta Query* liest nicht das ganze Objekt aus sondern nur einen Teil davon, um Ressourceneffizient zu agieren. Unter anderem, wenn die Geschäftslogik Löschoperationen (*delete mutation*) benötigt, versichert `@datasync` noch, dass gelöschte Objekte eine bestimmte Zeit in der Datenbank verweilen umd Konflikte zu beseitigen, bei der ein Client längere Zeit offline war. Die Standarteinstellung liegt bei 2 Tagen. Dazu mehr in Abschnitt **Konflikthandling** 2.2.

Die bisher vorgestellten Vorgänge sind nötig, um eine *API* für den Typ Task zu generieren. Es

werden dadurch *queries*, *mutations*, *subscriptions* und *resolver* für diesen Typ generiert, damit ein vollständiger *CRUD-Prozess* genutzt werden kann. Um diesen vollständigen *CRUD*-Prozess zu erstellen, benötigt man einen Server. Dieser befindet sich in `/server/src/graphql.ts`. Der Server ist ein *Apollo Express Server*. Mit der Funktion `createApolloServer()` der *Apollo* Server erstellt. Die Funktion `buildGraphbackAPI()`, die aus der *Graphback*-Bibliothek importiert wird, generiert uns den kompletten *CRUD*-Prozess. Sie nimmt 2 Argumente entgegen. Zum einen die Schema-Definitionen, die durch die Funktion `loadSchemaSync()` geladen wurden und ein Objekt welches Konfigurationen der eigentlichen *CRUD*-Generierung und dem enstprechenden Datenbankanbieter beinhaltet.

```
const { typeDefs, resolvers, contextCreator } = buildGraphbackAPI(modelDefs, {
        serviceCreator: createCRUDService(),
        dataProviderCreator: createDataSyncMongoDbProvider(db),
        plugins: [
            new DataSyncPlugin()
        ]
    });
```

In der Datei `/server/src/crudServiceCreator.ts` befinden sich all die Konfigurationen um den *CRUD*-Prozess zu entwicklen. Diese Demo nutzt *GraphQL subscriptions* damit jeder Client über Datenänderungen im Server nahezu in Echtzeit benachrichtigt wird. Um diese Funktionalität zu nutzen, muss der in der Konfiguration eine Instanz der *subscription* erzeugt und übergeben werden:

```
import { PubSub } from 'graphql-subscriptions';

export function createCRUDService(globalServiceConfig?: CRUDServiceConfig) {
    let pubSub = new PubSub();

    return (model: ModelDefinition, dataProvider: DataSyncProvider):
    ↪  GraphbackCRUDService => {
        const serviceConfig: CRUDServiceConfig = {
            pubSub,
            ...
        }
        ...
```

Wichtig für die zukünftige Datensynchronisation sind zwei Funktionen aus der *Graphback*-Bibliothek, welche Attribute des Konfigurationsobjektes sind, die wir mit:

```
import { DataSyncPlugin, createDataSyncMongoDbProvider } from
↪  "@graphback/datasync"
```

einbinden.

4.3 Client

Für die Implementierung des Clients wird das *framework Offix* genutzt. Prinzipiell funktioniert *Offix* wie *Graphback*. Es wird hierbei sozusagen ein lokaler *Apollo*-Server im Client erstellt, der mit einem serverseitigen *Apollo*-Server kommuniziert. Eine lokale *API* wird dementsprechend generiert, um *GraphQL*-Anfragen an einen *GraphQL*-Server zu schicken und parallel an den internen Browserspeicher *IndexedDB*. Die lokale *IndexedDB*-Datenbank hat bis zum Zeitpunkt fehlender Konnektiviät alle Daten im Speicher. *Offix* generiert mit dem `offix/cli` Konsolentool über den Paketmanager *npm*, *GraphQL queries* und *mutations*, basierend auf dem vorhandenen *GraphlQL*-Schema des Servers. Die Ausgabe der Generierung befindet sich in `/client/src/generated.ts`

Um mit dem *Apollo*-Server zu kommunizieren benötigt man einen *Apollo*-Client. Die Konfiguration für den Aufbau des *Apollo* Clients mit Offlinefunktionalität durch *Offix*, befindet sich in der Datei: `/client/src/config/clientConfig.ts`

```
const cache = new InMemoryCache({
  cacheRedirects: {
    Query: {
      getTask: (_, { id }, { getCacheKey }) => getCacheKey({ __typename: 'Task',
      ↪  id }),
    },
  },
});

export const clientConfig: ApolloOfflineClientOptions = {
  link: splitLink
  cache: cache,
  conflictListener: new ConflictLogger(),
  mutationCacheUpdates: globalCacheUpdates,
  networkStatus: new CapacitorNetworkStatus(),
  conflictProvider: new TimeStampState(),
  inputMapper: {
    deserialize: (variables: any) => {
      return (variables && variables.input) ? variables.input : variables;
    },
    serialize: (variables: any) => {
      return { input: variables }
    }
  }
};
```

Das Objekt `clientConfig` beinhaltet alle nötigen Informationen um einen Offline Client zu erstellen. Mit dem ersten Attribut `link` gibt man an welche *Apollo links* man verwenden möchte. In diesem Fall `splitLink` mit der *Apollo* Link mitgeteilt wird, dass wir die Kommunikation anhand der *GraphQL* Operation aufteilen. Somit werden alle *subscriptions* über *WebSockets* und alle *queries* über *HTTP* gesendet. WebSockets könnten dafür genutzt werden alle Operationen auszuführen, sollte doch in den meisten Fällen nur *HTTP* für *queries* nutzen. Der Grund hierfür ist, dass *queries* keine zustandsorientierte Verbindung benötigen, was das Senden über *HTTP* effizienter und skalierbarer macht, falls keine *WebSocket*-Verbindung bereits besteht. Im Abschnitt 4.4 wird das Konzept *Apollo*

Link näher erläutert.

Das zweite Attribut ist `cache`. Standardmäßig werden *queries* und Ergebnisse von Mutations mit `InMemoryCache` im Browser-Cache gespeichert. Durch das zusätzliche Argument `cacheRedirects` wird dem *Apollo*-Client mitgeteilt, dass Abfragen einzelner Tasks, direkt aus dem *Cache* gelesen werden sollen. Dadurch werden weitere Netzwerkanfragen vermieden.

Das nächste Attribut `ConflictListener` ist ein *Interface*, welches bei Konflikten andere Teile des Systems darüber informiert.

Ein wichtiges Attribut des `clientConfig`-Objektes ist, `mutationCacheUpdates` mit der Funktionalität für jede *mutation* den *Cache* zu aktualisieren und zusätzlich bei Seitenaktualisierungen und App-Neustarts, fehlgeschlagene Operationen wiederherzustellen.

Das darauffolgende Attribut `networkStatus` verschafft Auskunft über den Offline-und Onlinestatus der Applikation.

`conflictProvider` ist ein *Interface* welches Informationen beinhaltet, ob ein Objekt modifiziert wurde und ob daraus ein Konflikt resultiert.

Das letzte Attribut *inputMapper* sind 2 Funktionen die sicherstellen, dass die Eingabevariablen immer an die *root mutation* weitergegeben werden.

Ein Blick in die Datei `ApolloClientConfig.ts` der `offix-client`-Bibliothek zeigt die Optionen die einen *Apollo*-Client entscheidend erweitert:

```
export declare class ApolloOfflineClientConfig {
    httpUrl?: string;
    offlineQueueListener?: ApolloOfflineQueueListener;
    conflictStrategy: ConflictResolutionStrategy;
    conflictProvider: ObjectState;
    networkStatus?: NetworkStatus;
    terminatingLink: ApolloLink | undefined;
    cacheStorage: PersistentStore<PersistedData>;
    offlineStorage: PersistentStore<PersistedData>;
    conflictListener: CompositeConflictListener;
    mutationCacheUpdates?: CacheUpdates;
    cachePersistor?: CachePersistor<object>;
    link?: ApolloLink;
    inputMapper?: InputMapper;
    cache: any;
    retryOptions: {
        delay: {
            initial: number;
            max: number;
            jitter: boolean;
        };
        attempts: {
            max: number;
        };
    };
    constructor(options?: ApolloOfflineClientOptions);
}
```

Folgende Attribute sind hier hervorzuheben:

`offlineQueueListener` ist ein *Interface* das auf Offline-und Onlineereignisse reagiert und dadurch werden ab dem Zeitpunkt des Offlinezustand, *mutations* in Form einer Warteschlange im Browserspeicher eingereiht.

Damit kommen wir auch schon zum nächsten Attribut `offlineStorage` welches die Operationen `getItem`, `setItem` und `deleteItem` zur Verfügung stellt um mit dem Browserspeicher *IndexedDB* zu interagieren.

Ein weiter wichtig zu erwähndendes Attribut ist `cachePersistor` aus der Bibliothek `apollo-cache-persist` . Dieses Tool ermöglicht es den *Apollo Cache*, persistent in *IndexedDB* oder *LocalStorage* zu speichern. *IndexedDB* hält ein komplettes Abbild des *Apollo Cache* im Browserspeicher. Genauso wird auch jede Cacheoperation registriert und der Browserspeicher entsprechend sofort aktualisiert. Dies ist die essentielle Funktionalität, bei der die Applikation zu jederzeit den *Apollo Cache* und somit alle Applikationsdaten zu Verfügung hat.

Die Konfiguration des *Apollo Offline*-Client hat noch eine bedeutende Option, *retryOptions*. Diese Eigenschaft stammt aus der offiziellen *Apollo Link*-Bibliothek `@apollo/client/link/retry`. Sie wird benutzt um zu bestimmen, wie oft eine Operation wiederholt werden soll wenn sie fehlschlägt. Schlägt also eine Netzwerkanfrage fehl, wird diese Anfrage solange wiederholt wie in den Optionen angegeben. Dies ist nützlich für Szenarien in denen der Client online ist, ein Fehler ensteht und die Netzwerkanfrage nicht gesendet werden kann. wenn es nur einen sehr kurzen Verbindungsausfall gibt, kann damit ein Absturz der Applikation und mehrere Offlineoperationen verhindert werden.

Um diese Konfiguration in *React* zu nutzen, wird in der Datei `/client/src/AppContainer.tsx` eine *Apollo*-offline-Instanz erstellt und der Instanz die entsprechenden Konfigurationen übergeben. Damit *Offix* in *React* genutzt werden kann, werden `ApolloOfflineProvider` und `ApolloOfflineClient`, um die `App.tsx` Komponente gewickelt. Das Objekt welches die Instanz beinhaltet, wird den beiden Komponenten als *prop* übergen und somit besitzt *Offix* nun vollen Zugriff auf die Applikation.

Die Bibliothek `offix-scheduler` welche auf den ersten Blick nicht ersichtlich ist, wird bei der Kreierung der *ApolloOfflineClient*-Instanz automatisch mitübergeben. Dies ist wahrscheinlich die wichtigste Systemkomponente. Die Klasse *OffixScheduler* reiht fehlgeschlagene Operationen bei fehlender Konnektivität in eine Warteschlange und sendet diese wieder bei wiederkehrender Internetverbindung. `offix-scheduler` nutzt standartmäßig *IndexedDB* als Offlinespeicher. Zusätzlich wird ein *OfflineQueueListener* registriert um auf Ereignisse der Warteschlange zu reagieren. In der Datei `OffixScheduler.ts` befindet sich die Logik für die Umsetzung einer Warteschlange.

In der Datei `ApolloOfflineClient.d.ts` aus der `offix-client` Bibliothek sieht man wie *OffixScheduler* mit der Eigenschaft `scheduler` als Option miteingebunden wird:

```
1  export declare class ApolloOfflineClient extends
   ↪  ApolloClient<NormalizedCacheObject> {
2      ...
3      scheduler: OffixScheduler<MutationOptions>;
4      offlineStore?: ApolloOfflineStore;
5      conflictProvider: ObjectState;
6      conflictListener: CompositeConflictListener;
7      networkStatus: NetworkStatus;
8      queue: ApolloOfflineQueue;
9      mutationCacheUpdates?: CacheUpdates;
10     initialized: boolean;
11     inputMapper?: InputMapper;
12     constructor(options: ApolloOfflineClientOptions);
```

```
13      ...
14  }
```

Zu erwähnen wäre noch die Option `queue`. Sie ist die Datenstruktur in der die *GraphQL mutation*, also die fehlgeschlagenen *GraphQL*-Operation im Offlinemodus, gespeichert werden.

4.4 Datensynchronisation und Konflikthandling

Apollo Link

Für das Verständnis vom Datenfluss einer *Apollo GraphQL*-Applikation, ist es von Vorteil sich das Konzept *Apollo Link* näher zu bringen. Vor allem im Hinblick auf Applikationen welche offline funktionieren sollen, bietet dieses Konzept Möglichkeiten an Offlineszenarien zu behandeln.

Apollo Link ist für die Kontrolle von *HTTP*-Anfragen zuständig. Mit *Apollo Link* kann man den Datenfluß zwischen einem *Apollo* Client und einem *GraphQL*-Server individuell anpassen. Strukturiert ist *Apollo Link* in Form von einer Kette von Link-Objekten die sequentiell aufgerufen werden.

Man kann sich einen Link vorstellen wie eine *middleware* für eine *GraphQL*-Operation, welche die Operation modifizieren und vor Seiteneffekten bewahren kann. Verschiedene Links können kombiniert werden, wobei jeder Link für sich eine andere Aufgabe besitzen kann. Jeder Link in der Kette wendet Logik auf eine *GraphQL*-Operation an und ruft den nächsten Link in der Kette auf.

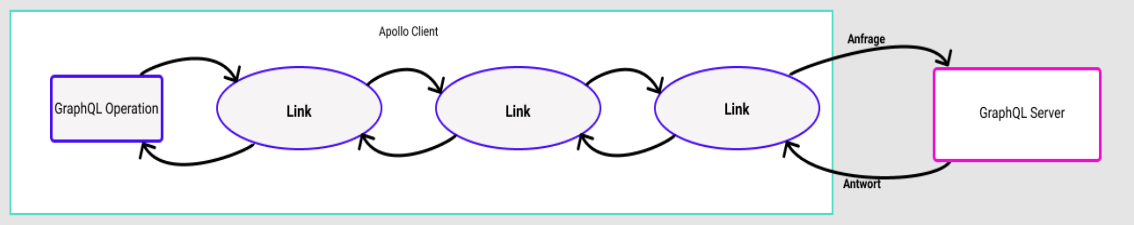

Abbildung 15: *Apollo Link Kette*

Man stelle sich zum Beispiel in der obigen Illustration vor, dass der erste Link Fehler behandelt, der zweite Link einen *HTTP*-Header für Authentifikation hinzufügt und der finale Link auch *terminating link* genannt, die Anfrage letztendlich über *HTTP* an das gewünschte Ziel sendet. Jeder Link hat somit eine bestimmte Aufgabe die zu einem reibungslosen Datenfluss führen sollte.
In dieser Demo würde die *Apollo Link*-Kette wie folgt aussehen:

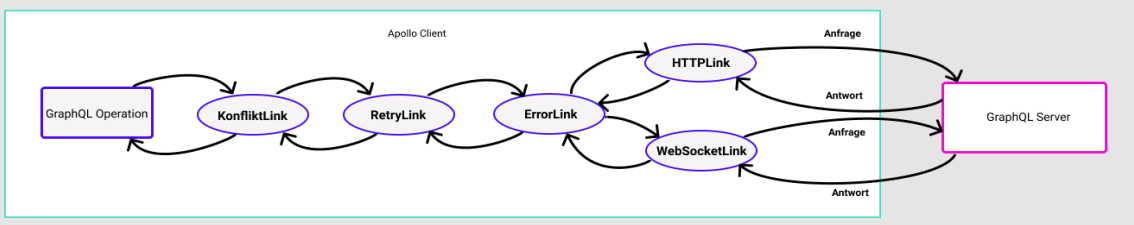

Abbildung 16: *Apollo Link Kette Demo*

Ein entscheidender Link, nämlich der Konfliktlink kommt hinzu. Schlägt eine Netzwerkanfrage fehl, liest der Konfliktlink die Errornachricht aus und entscheidet ob ein Konfliktauflösungsprozess gestartet werden muss. Zusätzlich dazu, kann man sehen wie vom *ErrorLink* zwei Links ausgehen. Dort wird anhand der *GraphlQL*-Operation entschieden ob diese über den *HTTP* oder *WebSocket-Link* gesendet wird. Eine *GraphQL query* wird über den *HTTP Link* und eine *mutation* über den *WebSocketLink* gesendet.

4.4.1 CRUD Prozess

Folgende Operationen sind in der Applikation möglich:

- Task hinzufügen
- Task löschen
- Task editieren
- Task öffnen
- Alle Tasks laden

Für diese Operationen werden *React Hooks* benutzt, die uns die Bibliothek `react-offix-hooks` zu Verfügung stellt. Grundsätzlich, um einen Datensatz in *GraphQL* zu erstellen, wird eine *GraphQL mutation* ausgeführt. Dies geschieht normalerweise mit der Funktion `client.mutate()`. *Offix* stellt eine Funktion bereit, welche eine *mutation* auch zulässt, wenn der Client offline ist. Die Funktion `useOfflineMutation()` erweitert *Apollos* `mutate`-Funktion mit Extra-Funktionalitäten.

Die *React*-Komponente um einen Task hinzuzufügen, befindet sich in er Datei `/client/src/pages/AddTaskPage.tsx` . Der Funktion `useOfflineMutation()` werden 2 Argumente übergeben. Zum einen die *GraphQL*-Operation die augeführt werden soll `createTask` und die dafür entsprechenden Optionen `mutationOptions.createTask` .

AddTaskPage.tsx:

```
  const [createTaskMutation] = useOfflineMutation(createTask,
  ↪  mutationOptions.createTask);
```
```

In der Datei `/client/src/helpers/mutationOptions.ts` sind die entsprechenden Optionen vorhanden um eine *offline mutation* auszuführen. Folgende Optionen führen zu einer *mutation* in der ein Task hinzugefügt wird:

```
export const createTask = {
  updateQuery: findTasks,
  returnType: 'Task',
  mutationName: 'createTask',
  operationType: CacheOperation.ADD,
  returnField: 'items'
};
```
```

Das erste Attribut `updateQuery` gibt an was genau aktualisiert werden soll. In diesem Fall `findTasks` *query*, welche die komplette Liste an Tasks im Cachespeicher hat. `returnType` ist der *schema type* um den es geht, der Typ des Ergebnisses der *mutation*-Operation. `mutationName` ist der Name der *mutation*.

Das nächste Attribut definiert den *operation type*, um entsprechende Aktualisierungen im *Cache* durchzuführen. Um dieses wichtige Attribut zu verstehen, muss man sich zuerst bewusst werden, wie eine *mutation* funktioniert.

Wenn eine *mutation* mehrere Einträge modifiziert, kreiert oder entfernt, wird der *Apollo Cache* nicht automatisch aktualisiert. Das zu beheben benötigt eine sogenannte Update-Funktion, welche beim Aufruf von `useMutation`, im Fall dieser Demo, `useOfflineMutation` mitgegeben werden kann. Der Sinn einer Update-Funktion ist, den *Cache* mit den neuen Resultaten vom Server abzugleichen. Bedeutet, wenn man einen Task hinzufügt, aktualisiert man manuell den *Cache* im Client mit dieser Update-Funktion. Dementsprechend gibt es für die 3 *CRUD*-Operationen jeweils eine Update-Funktion welche den *Cache* aktualisieren:

```
export const globalCacheUpdates = {
  createTask: getUpdateFunction(createTask),
  updateTask: getUpdateFunction(updateTask),
  deleteTask: getUpdateFunction(deleteTask),
}
```

Über die Bibliothek `offix-cache` werden diese eingebunden.

Subscriptions

Sobald ein Datenbankeintrag oder Änderung stattfindet, werden alle Clients über *WebSockets* darüber informiert. Das bedeutet, dass auch bei jeder *Subscription* wieder der *Cache* aktualisiert werden muss. Aus diesem Grund werden der *Apollo*-Client-Funktion `subscribeToMore` bestimmte *Subscription* Optionen übergeben, welche den *Cache* stets aktualisieren. Diese Prozedur folgt in der *React*-Komponente **TaskPage.tsx**:

```
  const { loading, error, data, subscribeToMore } = useQuery(findTasks, {
    fetchPolicy: 'cache-first'
  });

  const isOnline = useNetworkStatus();

  useEffect(() => {
    if (!subscribed) {
      subscribeToMore(subscriptionOptions.add);
      subscribeToMore(subscriptionOptions.edit);
      subscribeToMore(subscriptionOptions.remove);
      setSubscribed(true);
    }
  }, [subscribed, setSubscribed, subscribeToMore])
```

Um den Cache so schnell wie möglich zu aktualisieren, wird es so gehandhabt, dass durch *useEffect* also bei jedem *render* der Komponente, jeweils die die entsprechende Funktion aufgerufen wird.

Ein entscheidendes Problem taucht jedoch bei Ipdate-Funktionen auf. Ein *Apollo Client* hält alle *mutation*-Parameter im Cachespeicher. Bei einem App-Neustart, sprich einer Seitenaktualisierung oder vom Offline zum Onlinestatus, gehen aber alle *mutation* Parameter verloren, somit auch die wichtigen Update-Funktionen die den *Cache* und die UI updaten. Daher muss es eine Lösung geben wie man alle Infomationen immer noch im Speicher halten kann und der Cachespeicher damit nicht verloren geht.

Optimistic UI

Für unmittelbare visuelle Ergebnisse wird *Optimistic UI* genutzt. *Optimistic UI* ist ein Entwurfsmuster, mit dem Resultate einer *mutation* simuliert werden und das *User Interface* mit diesen Resultaten aktualisiert, noch bevor die Antwort vom Server empfangen werden. Wenn die Resultate vom Server zurückkommen, wird die sogenannte *optimistic response* mit der Antwort vom Server ersetzt. Für *Offline-first*-Applikationen ist dies ein optimales Konzept, denn nachdem eine Operation ausgeführt wird, geht es vor allem darum dem Nutzer sofortige Ergebnisse zu liefern auch wenn keine Internetverbindung besteht. Die Bibliothek `offix-cache` macht dies möglich indem über die Update-Funktionen *Optimistic UI* genutzt wird.

Folgendes Problem ensteht generell bei Applikationen die online sowohl als auch offline funktionieren sollen. Man stelle sich vor, eine *mutation* erzeugt ein Objekt. Dieses Objekt wird dann lokal in einer Warteschlange gespeichert, um bei wiederkehrender Verbindung von dort aus wieder gesendet zu werden. Das Offline erstellte Objekt besitzt nun aber keine servergenerierte ID. Somit bekommt es eine temporärere clientgenerierte ID. Wenn dieses Objekt danach Offline bearbeitet wird, hat der Server kein Wissen darüber welches Objekt bearbeitet wurde. Wenn die *mutation* erfolgreich ist und die Antwort vom Server kommt, müssen alle Referenzen zu diesem Objekt in der Warteschlange mit der ID des Servers aktualisiert werden. Mit der Funktion `replaceClientGeneratedIDsInQueue` in der Datei `optimisticResponseHelper.ts` im Modul `offix-client` wir mit Hilfe der *optimistic response* das Problem gelöst. Bei erfolgreicher Antwort vom Server wird die Client-ID und Server-ID an den Client gesendet, in der Warteschlange nach der Client-ID gesucht und mit der Server-ID aktualisiert. Erst dann kann der *Cache* mit der neuen *ServerID* geupdated werden.

Offline Datenfluss

createTask

1. In der Datei `/client/src/pages/AddTaskPage.tsx` wird `useOfflineMutation(createTask, mutationOptions.createTask)` aufgerufen. Dadurch wird mit dem *GraphQL*-Client, die *GraphQL*-Operation durch alle *Apollo Links* weitergereicht und schließlich mit dem *HTTPLink* versucht eine Netzwerkanfrage zu senden. die Funktion `client.offlineMutate()` in der Datei `useOfflineMutation.ts` der `react-offix-hooks` Bibliothek führt die *GraphQL-mutation* dann aus.

2. Ein *GraphQL mutation*-Objekt mit dem Attribut `startDate` als Zeitstempel und einer temporären ID (*optimistic ID*) wird erstellt.

3. Der *Apollo Cache* wird mit dem *mutation*-Objekt, also der *optimistic response* aktualisiert.

4. *Apollo Cache* wird durch `apollo-cache-persist` in *IndexedDB* repliziert.

5. Task wird zur Komponente `OfflineQueuePage.ts` der Benutzeroberfläche hinzugefügt

6. Kurz darauf ensteht ein Fehler. Denn bei netzwerkrelevanten Fehlern gibt Apollos ursprüngliche `client.mutate` Funktion ein *promise* mit einem *Error* zurück, so auch `client.offlineMutate` mit dem Unterschied den *Error* mit diesem *promise* zu referenzieren und die Information mitzugeben, dass diese *GraphQL*-Operation offline abgeschickt wurde.

```
      client.offlineMutate({
        mutation,
        ...options,
        ...mutateOptions,
        variables: mutateVariables
        })
        .then(response => {
          onMutationCompleted(response as ExecutionResult<TData>,
          ↪  mutationId);
          resolve(response as ExecutionResult<TData>);
        })
        .catch(err => {
          onMutationError(err, mutationId);
          reject(err);
        });
```

7. `OffixScheduler.ts` der Bibliothek `offix-scheduler` welcher in `client/src/ApolloOfflineClient.ts` intialisiert wurde wird nun aktiv. In der Funktion `execute()` wird geprüft ob die Applikation sich im Offlinemodus befindet und somit die *GraphQL*-Operation in *IndexedDB* in einer Warteschlange gespeichert. Daraufhin folgt der der wichtige Schritt, dass ein neues *promise* erzeugt wird, welches mit der fehlgeschlagenen Operation zusammen in die Warteschlange eingereiht wird. Zuletzt wird dann über die Klasse *OfflineError* ein *Error* erstellt indem das *promise* dieser Klasse übergeben wird.
OffixScheduler.ts:

```
        public async execute(options: T): Promise<any> {
            if (this.online) {
              return this.executor.execute(options);
            } else {
              // GraphQL Operation in Warteschlange einreihen
              const queueEntry = await this.queue.enqueueOperation(options);

              // promise erzeugen welches resolved/rejected wenn die
              ↪  Operation erfolgreich war
              const mutationPromise =
              ↪  this.queue.buildPromiseForEntry(queueEntry);

              // Error mit Referenz zum promise
              throw new OfflineError(mutationPromise as any);
            }
          }
```

8. Ein individueller *OfflineEventListener* wird für diese *mutation* registriert und darauf gewartet bis das *promise* duch das Onlineereignis *fullfilled* wird.

   Pseudocode zur Veranschaulichung:

```
try {
  await client.offlineMutate(options)
} catch(error) {
  if(error.offline) {
    const result = await error.watchOfflineChange()
    console.log('Wieder Online ! mutation erfolgreich empfangen', result)
  }
}
```

   Lässt man sich den Error in der Konsole mit `console.log(error)` ausgeben ensteht folgende Ausgabe:

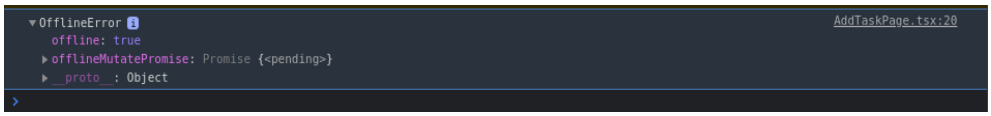

Abbildung 17: *Offline Error*

   Wie man sehen kann ist das *promise* im *pending state* und wartet darauf *fullfilled* zu werden.

9. Wenn der Client wieder online ist, ensteht ein Online-Event, der *OfflineQueueListener* wird benachrichtigt und alle Operationen in der Warteschlange werden der Reihenfolge nach abgeschickt
10. Datensatz erreicht den Server und prüft ob es einen Konflikt gibt
11. Datenbankeintrag wird in *MongoDB* mit einem Zeitstempel der Speicherung erstellt
12. Task wird über *WebSockets* an alle Clients gesendet
13. Beim Client welcher die Anfrage abgeschickt hat, wechselt bei erfolgreicher Antwort vom Server, das *promise* auf *fullfilled*
14. *optimistic response* im Client wird mit der Antwort vom Server überschrieben. Somit wird das lokale Objekt mit der Antwort vom Server ersetzt und die temporäre ID mit der des Servers überschrieben
15. *Apollo* Cache-Updates mit der Antwort vom Server werden ausgeführt.
16. Task wird aus der *IndexedDB*-Warteschlange gelöscht
17. Das Abbild des *Apollo*-Cache im Browserspeicher *IndexedDB* wird ebenso mit dem neuen Datensatz aktualisiert
18. Benutzeroberfläche wird aktualisiert

deleteTask

Eine *delete mutation* wenn offline, läuft nahezu identisch ab. Die einzigen Unterschiede sind, dass bei einer Offlinelöschung, der Task von der Benutzeroberfläche erst enfernt wird, wenn die Antwort vom Server zurückkommt. Generell, wenn der Client online ist und einen Datensatz löscht, findet ein sogenannter *soft delete* statt. Das bedeutet, dass ein Task nicht direkt gelöscht, sondern nur als gelöscht markiert wird. Hierbei wechselt dann der Status des Attributes `_deleted = false` auf `_deleted = true`. Solche Datensätze verweilen noch 2 Tage in der Datenbank um eventuell Konflikte zu beseitigen, bei denen ein Client sehr lange offline war und möglicherweise so einen schon gelöschten Datensatz bearbeitet.

updateTask

Der einzige Unterschied zu einer *createTask mutation* ist die Veränderung des Attributes `updatedAt`, welche der Client bei der Bearbeitung des Datensatz ändert. Genauso ändert der Server, wenn wieder online, dieses Attribut mit dem Zeitsptempel der Speicherung in der Datenbank. Dazu noch ausführlichere Informationen im Kapitel Konflikhandling 4.4.3.

App Neustart

Beim Aktualisieren der Seite, sprich einem Neustart der App oder auch nur beim Aufrufen der Seite mit allen Tasks, wird in der Komponente *TaskPage.tsx*, jedes Mal die Funktion `useQuery(findTasks, fetchPolicy: 'cache-and-network')` aufgerufen. Hiermit wird die *GraphQL query* `findTasks` gesendet, um alle Tasks zu laden. Die Anfragestrategie ist mit `cache-and-network` so eingestellt, dass parallel zu einer Netzwerkanfrage auch eine Anfrage an den *Cache* gesendet wird. Da die Applikation offline ist, schlägt die Netzwerkanfrage fehl und die Daten werden aus dem *Cache* geladen. Dies ist möglich, weil `apollo-cache-persist` mit der lokalen Browserdatenbank *IndexedDB* verknüpft ist und somit alle Daten von dort gelesen werden. Zusätzlich ist der *Service Worker* dazu da, alle Seiten der Applikation im Cache-Speicher zu halten. Jede Seitennavigation in der App wird durch den *Service Worker* möglich gemacht. In der Datei `client/src/serviceWorker.ts` geschieht dies mit der Open-Source-Bibliothek *workbox* von *Google*. `create-react-app` liefert diese Konfiguration automatisch mit. Der *Service Worker* muss nur in der Datei `index.ts` mit `serviceWorker.register()` aktiviert werden. Hier muss aber dringend beachtet werden ob der Internetbrowser auch *Service Worker* unterstützt[41].
Damit ist es nun möglich jede Seite der Applikation im Offlinemodus zu nutzen.

4.4.2 Datensynchronistation

Delta Query

GraphQL Subscriptions sind perfekt in Situationen bei denen Nutzer sofortige Updates erhalten wollen. Sie sind sehr flexibel und Entwickler haben somit eine direkte Anbindung für serverseitige Datenänderungen. *Subscriptions* liefern aber nur Updates an Clients welche abonniert und online sind. Um bei verpassten *Subscriptions* aufgrund eines Neustarts der App oder fehlender Internetverbindung, trotzdem noch Updates zu bekommen, kann der Client das Konzept der *Delta Query* nutzen. Der Sinn einer *Delta Query* ist es, dass der Client nach einer Offlineperiode nicht alle Datensätze erneut laden muss, sondern nur die Änderungen von denen er nichts mitbekommen hat.

Wie bereits erwähnt kann sich *Graphback* an dem Konzept der *Delta Query* bedienen. Hierbei werden die Daten in 2 Teile separiert. Server-und clientseitig bekannte konfliktfreie Daten und Daten

die der Client noch noch nicht gesehen hat aber speichern muss.

Das Problem jedoch ist, dass nicht jeder Datensatz *Delta Query* unterstützt. Bei jeder Löschung wird ein Datensatz von der Datenbank enfernt, was bedeutet, dass das Sortieren und Separieren des Datensatzes um Updates zu erlangen, den Datensatz unbrauchbar macht.

Gelöst wird dieses Problem unter anderem durch *soft deletes*:

- Datensätze werden nicht in der Datenbanktabelle gelöscht
- Eine separate Tabelle nur für Löschsoperationen und zusammenführen der Ergebnisse, wenn Daten zum Client gesendet werden
- Weitere separate Tabelle um Unterschiede abzufragen

Diese Konzepte sind sehr weit verbreitet und werden insbesondere bei Applikationen die große Datenmengen handhaben, verwendet. Vor allem wenn die Daten im Client auch bei fehlender Konnektivität verfügbar sein müssen. Aus diesen Gründen hat *Graphback Delta queries* sowie serverseitige Konfliktausflösung in die *GraphQL CRUD Spec* miteingebaut.

Das Package `@graphback/datasync` welche mit der in Kapitel 4.2 beschriebenen Annotation `@datasync` genutzt wird, ist eine Referenzimplementation für *Delta queries* und Konfliktauflösung, Erweiterungen der *GraphQL CRUD Spec*. *DataSync* erreicht dies, indem ein spezieller Typ einer *query* zur Verfügung gestellt wird, eine sogenannte *Delta Query*, welche die existierenden *resolver* um einen serverseitigen Konfliktauflösungsmechanismus erweitern. Eine *Delta Query* hilft einem Client dabei zu erfahren, was sich in der Zeitspanne in der er offline war, verändert hat.
Hier ein Beispiel einer *Delta Query* mit der entsprechenden Antwort des Servers:

Delta Query:

```
query syncTasks {
  syncTasks(lastSync: "1590679886048"){
    items{
     _id
      description
      _pdatedAt
      _deleted
    }
    lastSync
  }
}
```

Antwort der Delta Query:

```
{
  "data": {
    "syncTasks": {
      "items": [
        {
          "id": "5ee0a1da7f1f39313744185a",
          "description": "First!"
          "updatedAt": "1591852693075",
          "_deleted": "true"
        },
        {
          "id": "5ee0a67345beff3862220be4",
          "description": "Second"
          "updatedAt": "1591780979988",
          "_deleted": "false"
        }
      ],
      "lastSync": "1591852700920",
      "lastSync": "1591852700920",
    }
  }
}
```

Mit der *Delta Query* `query syncTask(lastSync: "1590679886048")` antwortet der Server mit einer Liste aktuellster Versionen der Datensätze, seit dem `lastSync` Zeitstempel. Die sogenannte Deltaliste. Intern wird in der Datenbank die `updatedAt`-Eigenschaft mit dem `lastSync` verglichen, um zu sehen welche Datensätze verändert wurden. Das Attribut `deleted` auf `true` gibt an, dass dieser Datensatz seit der letzten Synchronisation `lastSync: 1590679886048` gelöscht wurde. Die Antwort beinhaltet auch noch ein `lastSync`-Zeitstempel der für die nächste *Delta Query* `syncTask(lastSync: ...)` genutzt werden kann. In der Beispielantwort ist zu sehen, dass der erste Task `"Saugen!"` gelöscht und der zweite Task `"Putzen!"` modifiziert wurde.

Nun gibt verschiedene Möglichkeiten *Delta Query* zu implementieren. Die in der Demo genutze Variante bietet die Möglichkeit in einem bestimmten Zeitintervall eine solche *Delta Query* zu senden und sendet ebenfalls eine *Delta Query* bei wieder erlangter Internetverbindung. Der aktuelle Quellcode der Demo zeigt wie der Server Information hinzufügt um *Delta Queries* nutzen zu können. Entscheidend hierfür ist die Datei `DatasyncMongoDBDataProvider.ts` der Bibliothek `@graphback/datasync` . Für jedes Datenbankmodell werden *Delta queries* erstellt. Das bedeutet für jedes *GraphQL*-Schema wird auch ein *resolver* erzeugt. Dies erfolgt in der Datei `DataSyncPlugin.ts` der Bibliothek `@graphback/datasync` mit der Funktion `transformSchema()` . Zusammenfassend fügt diese Funktion zusätzlich die Zeitstempel als Attribut `lastSync` hinzu, welche die Bestätigung für den Server ist, dass der Client die Veränderungen eingebunden hat. Hinzugefügt werden noch die Attribute `_deleted` und `updatedAt`. Im Falle eines schon gelöschten Datensatzes, gibt `updatedAt` Auskunft darüber, wann dieser gelöscht wurde und bei Modifikation eines Datensatzes, wann dieser verändert wurde.

Zusätzlich zu den hinzugefügten Attributen erzeugt die Funktion neue *GraphQL*-Schema-Typen für die *Delta queries* welche folgendermaßen aussehen:

```
type TaskDeltaList {
  items: [TaskDelta]!
  lastSync: String
}

type TaskDelta {
  # @id
  id: ObjectID!
  title: String!
  description: String!
  status: TaskStatus
  type: String
  priority: Int
  public: Boolean
  startDate: DateTime
  payload: JSON

  # @createdAt
  # @db(type: 'timestamp')
  createdAt: String

  # @updatedAt
  # @db(type: 'timestamp')
  updatedAt: String
  _deleted: Boolean
```

Weiter wird in der Funktion `createResolvers()` dann für jedes Datenbankmodell ein *Delta Query Resolver* hinzugefügt:

```
public createResolvers(metadata: GraphbackCoreMetadata): IResolvers {
    const resolvers: IResolvers = {
```

```
      Query: {},
      Mutation: {},
      Subscription: {}
    }

    const models = metadata.getModelDefinitions()

    for (const model of models) {
      if (isDataSyncModel(model)) {
        this.addDeltaSyncResolver(model, resolvers.Query as IFieldResolver<any,
        ↪  any>)
      }
    }

    return resolvers
  }
```

Mit der If-Anweisung wird zuerst geprüft ob es sich bei dem Model um ein *DataSync Model* handelt. Es wird geprüft ob das Model die Annotation `@datasync` besitzt. Erreicht ein Datensatz den Server, wird in der Datei `DatasyncMongoDBDataProvider.ts` der Bibliothek `@graphback/datasync` mit der Funktion `checkForConflicts()` auf Konflikte geprüft. In der Datei befinden sich auch die *MongoDB*-Datenbankoperationen.

Konflikte werden nur bei einer Löschung oder Bearbeitung eines Datensatzes benötigt. Bei Kreierung eines Datensatzes wird nur das Attribut `_deleted` auf `false` gesetzt.

Zweiter Demonstrator

Zu Veranschaulichung einer Implementation von *Delta queries* wird eine zweiter Demonstrator hinzugezogen[42]. Der Quelltext befindet sich ebenso im Anhang.

In der Datei `/src/config.ts` in welcher der *Offix Datastore* intitalisiert wird, kann man einstellen in was für einem Intervall eine *Delta Query* an den Server zu senden.

```
export const datastore = new DataStore({
  dbName: "offix-datasync",
  replicationConfig: {
    client: {
      url: "http://localhost:5400/graphql",
      wsUrl: "ws://localhost:5400/graphql",
    },
    delta: { enabled: true, pullInterval: 30000 },
    mutations: { enabled: true },
    liveupdates: { enabled: true }
  }
});
```

Wie im obigen Quellcode zu sehen ist, ist das Intervall auf 30000 eingestellt, angegeben in Millisekunden.

Je nach Geschäftszweck und Performanceabsichten müssen Überlegungen angestellt werden, wie oft eine solche Anfrage an den Server abgeschickt werden soll. Ist die Rede von einer Echtzeitapplikation, welche kleine Datenpakete in sehr kurzen Abständen sendet, bietet sich an das Intervall zu verkürzen. Für den Zweck bei großen Datenmengen würde ein Zeitintervall oder ein kurzes Zeitintervall eher weniger Sinn machen und daher wäre die Option, nur bei wiederkehrender Konnektivitivät solch eine *Delta Query* zu senden.

4.4.3 Konflikthandling

Das `@graphback/datasync` *package* besteht aus dem Datensynchronisations-Schema-Plugin und kompatible Datenbankquellen. In dieser Bibliothek befinden sich auch die Datensynchronisationsstrategien für den *Offix GraphQL*-Client mit Offlinefunktionalität. Genauer gesagt fungiert das *package* dazu das *Offix*-Framework mit einer geeigneten *GraphQLCRUD-API* zu erweitern.

Die Applikation erlaubt es Nutzer Daten offline zu modifizieren. Dies kann zu Konflikten führen. Ein Konflikt findet statt wenn zum Beispiel mehrere Nutzer versuchen den gleichen Datensatz zu modifizieren aber einer der beiden Nutzer davon offline ist. Die Applikation muss diese Konflikte auflösen können. Dies wird in 2 Phasen geregelt:

- **Konfliktentdeckung**
- **Konfliktauflösung**

Konfliktentdeckung

Workflow um Konflikte zu entdecken:

- **Eine Mutation erfolgt** - Ein Client versucht durch eine *GraphQL mutatio*, ein Objekt auf dem Server zu löschen oder modifizieren
- **Den Objektstatus lesen** - Der Server liest des Status des aktuellen Objektes, welcher der Client versucht zu modifizieren
- **Konfliktentdeckung** - Der Server vergleicht das aktuelle Objekt mit den den gesendeten Daten des Clients, um zu sehen ob ein Konflikt besteht

Die Konfliktentdeckung geschieht mit dem `DataSyncPlugin` und dem `createDataSyncMongoDbProvider` aus der `@graphback/datasync` Bibliothek, welche der Funktion `buildGraphbackAPI()` übergeben werden. In dem Plugin werden die *resolver* und *Delta queries* erstellt und in der Datei *DataSyncPlugin.ts* erfolgt die Konfliktentdeckung über die Anbindung der *MongoDB*-Datenbank. Momentan bietet *Graphback* nur ein Plugin für *MongoDB* an. Sobald der Server einen Konflikt entdeckt, schickt er einen Konflikterror in Form eines *JSON* an den Client wo dann der Konflikt aufgelöst wird.

Struktur der Konfliktnachricht:

```
"extensions": {
    "code": "INTERNAL_SERVER_ERROR",
    "exception": {
        "conflictInfo": {
            "serverState": {
               "id": "5eedae1367d72e2192561723",
```

```
            "text": "AlreadyUpdatedTitle",
            "_deleted": "false",
            "createdAt": "1592634899084",
            "updatedAt": "1592634899084"
          },
          "clientState": {
            "id": "5eedae1367d72e2192561723",
            "text": "ClientSideUpdate",
            "updatedAt": "1592634898093"
          }
        }
    }
}
```

Konflikte können auf 2 Arten entdeckt werden:

1. **Versions/Zeitbasiert**: Jedes Objekt besitzt Attribute wie `createdAt` und `updatedAt` als Ganzzahlwert. Ein Konflikt ensteht wenn es zeitliche Differenzen zwischen einem Client und dem Server gibt. Das bedeutet, wenn ein Client einen Datensatz bearbeitet, ändert sich das Attribut `updatedAt` welches der Server bei jeder Netzwerkanfrage prüft.
2. **Hashbasiert**: Dieser Mechanismus prüft das komplette Objekt, welches vom Client modifiziert wurde. Hierbei werden die Hashes beider Objekte verglichen. Sind diese nicht identisch, ist dies ein Indiz dafür, dass ein Client das Objekt verändert hat.

Diese Demo nutzt die versions/zeitbasierte Variante.

Pseudocode zur Veranschaulichung

```
const { conflictHandler } = require("@graphback/datasync")

type Task {
    id: ObjectID
    title: String
    updatedAt: Date
}

type Mutation {
    updateTask(title: String!, updatedAt: Date): Task
}

const serverData = db.find(clientData.id);
const conflict = conflictHandler.checkForConflicts(serverData, clientData);

if(conflict) {
    sendConflictToClient();
    throw ConflictError();
```

```
19 }
20
21 db.save(clientData.id, clientData);
```

Konfliktauflösung

Workflow um Konflikte zu lösen:

1. **Eine Mutation erfolgt** - Ein Client versucht durch eine *GraphQL mutation*, ein Objekt auf dem Server zu löschen oder zu modifizieren
2. **Den Objektstatus lesen** - Der Server liest des Status des aktuellen Objektes, welcher der Client versucht zu modifizieren
3. **Konfliktentdeckung** - Der Server vergleicht das aktuelle Objekt mit den den gesendeten Daten des Clients, um zu sehen ob ein Konflikt existiert.
4. **Konfliktauflösung** - Der Client versucht den Konflikt aufzulösen und schickt eine neue Anfrage an den Server in der Hoffung, dass die Daten nun konfliktfrei sind.

Die Konfliktauflösung beinhaltet folgende Punkte:

- Sie geschieht am Client
- Besitzt einen zusätzlichen Rückgabewert zum *GraphQL*-Kontext jeder *mutation*
- Zusätzliche Metadaten innerhalb der *GraphQL schema*-Typen (wie zum Beispiel ein Attribut `updatedAt`) je nach hashbasierter oder versionsbasierter Konfliktentdeckung
- Serverseitige *resolver* um die Konfliktnachricht zurück an den Client zu senden

Konfliktsituationen

- **Aktuellster Datensatz** - Wenn beispielsweise ein Client A offline ist und einen Datensatz um 15.27 Uhr bearbeitet und Client B online aber diesen Datensatz online um 14.27 Uhr bearbeitete, nimmt der Server nur den veralteten Datensatz von Client B wahr. Der Server hat keine Chance Datensätze zu erkennen die offline bearbeitet werden. Sobald Client A wieder Online ist, muss also der offline bearbeitete Datensatz von Client A als aktuellester erkannt werden.
- **Bereits gelöschter Datensatz** - Versucht ein Client einen gelöschten Datensatz im Offlinemodus zu bearbeiten, kann es zu einem Konflikt kommen. Es kann passieren, dass ein Client längere Zeit offline ist, in dieser Zeit aber der Datensatz von einem anderen Client schon gelöscht wurde. Der Client hat trotzdem die Möglichkeit den Datensatz zu bearbeiten, da er Offline und nicht auf dem neuesten Stand ist. Sobald der Client wieder online ist, muss der Server den Client benachrichtigen, diesen Datensatz zu löschen.
- **Lokaler Konflikt** - Ist ein Client online und bearbeitet einen Datensatz der genau in diesem Moment aktualisiert wurde, muss der Client darüber informiert werden, dass ein Update stattgefunden hat und der Client muss danach gefragt werden ob er diesen Datensatz dennoch senden möchte. Dadurch kann eine zusätzliche Netzwerkanfrage vermieden werden.

Konfliktauflösung Server

createTask

Der Server braucht sich bei Erstellung eines Task nicht um Konfliktauflösung zu kümmern. Alles was er benötigt ist die Information über die Reihenfolge der Datensätze oder Zeitstempel.

updateTask

Bei jedem wird in der Datei `DatasyncMongoDBDataProvider.ts` mit der Funktion `checkConflicts()` in der *MongoDB*-Update-Funktion auf Konflikte geprüft. Die Funktion nimmt den Datensatz und den *GraphQL*-Kontext als Argumente entgegen. Der Datensatz wird nun in der *MongoDB* Datenbank gesucht und das Ergebnis wird dann in der Variable `queryResults` gespeichert. Die Konfliktauflösung geschieht folgendermaßen:

```
if (queryResult) {
      queryResult[idField.name] = queryResult._id;
      if (
        queryResult[fieldNames.updatedAt] !== undefined &&
        clientData[fieldNames.updatedAt].toString() !==
        ↪ queryResult[fieldNames.updatedAt].toString()
      ) {
        queryResult[fieldNames.updatedAt] =
        ↪ queryResult[fieldNames.updatedAt].toString()

        return { serverState: queryResult, clientState: clientData };
      }

      return undefined;
    }

    throw new NoDataError(`Could not find any such documents from
    ↪ ${this.collectionName}`);
```

Es wird nun erst einmal geprüft ob solch ein Datensatz existiert. Ist das der Fall, wird der Datensatz des Clients mit dem Ergebnis `queryResult` des Servers über das Attribut `updatedAt` nach zeitlichen Unterschieden zwischen Server und Client gesucht. Es wird geprüft ob der Zeitstempel des Clients äquivalent mit dem des Servers ist. Wurde eine zeitliche Inkonsistenz gefunden, bekommt die Funktion ein Objekt als Rückgabewert, in dem der aktuelle Status des Clients und des Servers festgehalten werden. Dies ist die Konfliktnachricht für den Client.

```
public async update(data: any, context: GraphbackContext): Promise<Type> {
    const conflict = await this.checkForConflicts(data, context);

    if (conflict !== undefined) {
      throw new ConflictError(conflict);
    }

```

```
8
9       return super.update(data, context);
10    }
```

Mit `throw new ConflictError(conflict)` wird der Konflikt dann an den Client gesendet, damit er diesen auflösen kann. Sobald der Client den Konflikt versucht hat aufzulösen, startet die Prozedur von vorne und der Server prüft ob die Daten nun konfliktfrei sind. Sind sie es, wird der Datensatz in der Datenbank mit `return super.update(data, context)` gespeichert.

deleteTask

Eine ähnliche Prozedur findet auch bei Löschung eines Datensatzes statt. Mit dem Unterschied, dass der Datensatz nicht wirklich gelöscht wird sondern ein *soft delete* durchgeführt wird. Der enstprechende Datensatz wird mit `data._deleted = true` als gelöscht in der Datenbank aktualisiert und nicht endgültig gelöscht.

Konfliktauflösung Client

Für das Handling verschiedener Konflikte im Client ist hauptsächlich die Bibliothek `offix-conflicts-client` zuständig. Generell sieht der Mechanismus wie folgt aus. Ein Client erstellt, editiert oder löscht einen Datensatz. Dieser Datensatz erhält einen Zeitstempel mit dem Zeitpunkt der Erstellung mit `startDate` und bei Bearbeitung oder Löschung einen Zeitstempel mit `updatedAt`. Erreicht der Datensatz irgendwann den Server, prüft dieser auf Konflikte. Exisitiert ein Konflikt, wird eine Konfliktnachricht wird vom Server an den Client gesendet.

Der *ErrorLink* in der *Apollo Link*-Kette gibt die Konfliktnachricht an den Konfliklink weiter und wird dann in der Datei `offix/client/src/apollo/conflicts/ConflictLink.ts` weiterverarbeitet. Insbesondere die Funktion `conflictHandler()` ist dafür zuständig:

```
1  private conflictHandler(errorResponse: ErrorResponse): Observable<FetchResult> {
2      const { operation, forward, graphQLErrors } = errorResponse;
3      const data = this.getConflictData(graphQLErrors);
4      const individualStrategy = this.strategy || UseClient;
5      if (data && operation.getContext().returnType) {
6        const base = operation.getContext().conflictBase;
7        const conflictHandler = new ConflictHandler({
8          base,
9          client: data.clientState,
10         server: data.serverState,
11         strategy: individualStrategy,
12         listener: this.listener,
13         objectState: this.config.conflictProvider as ObjectState,
14         operationName: operation.operationName
15       });
16       const resolvedConflict = conflictHandler.executeStrategy();
17       if (resolvedConflict) {
18         operation.variables = this.config.inputMapper ?
           ↪  this.config.inputMapper.serialize(resolvedConflict):
           ↪  resolvedConflict;
```

```
      }
    }
    return forward(operation);

  }
```

Die Konfliktnachricht des Servers befindet in der *GraphQL*-Errornachricht. Die genaue Konfliktnachricht wird extrahiert und eine Instanz des Konflikthandlers erstellt. Folgende Parameter nimmt die Instanz entegegen:

- `base` - Daten vor Client und Serveränderungen
- `client` - Clientseitige Datenveränderungen, *GraphQL mutation*-Variablen
- `server` - Serverseitige Änderungen seit der letzten Bearbeitung von welcher der Client wusste
- `strategy` - Die Strategie um den Konflikt zu lösen
- `objectState` - *Interface* welches den Status des aktuellen Objektes handhabt
- `listener`- Konfliktlistener um Client mit Konfliktinformationen zu benachrichtigen
- `operationName` - Name der *GraphQL*-Operation

Mit Kreierung dieser Instanz wird im Konstruktor der Klasse direkt die Funktion `checkConflict()` aufgerufen welche die server-und clientseitigen Veränderungen berechnet. Und zwar wir mit der ersten *for*-Schleife:

```
for (const key of Object.keys(client)) {
    if (base[key] && base[key] !== client[key]) {
      if (!this.ignoredKeys.includes(key)) {
        this.clientDiff[key] = client[key];
      }
    }
  }
```

Jeder Wert des `base` Objektes wird mit dem `client` Objekt verglichen und auf Ungleicheit geprüft. Gibt es eine Ungleichheit, wird diese in der Variable `clientDiff` gespeichert.
In der zweiten *for*-Schleife

```
 for (const key of Object.keys(this.options.client)) {
    if (base[key] && base[key] !== server[key]) {
      if (!this.ignoredKeys.includes(key)) {
        this.serverDiff[key] = server[key];
        if (this.clientDiff[key]) {
          this.conflicted = true;
        }
      }
    }
  }
```

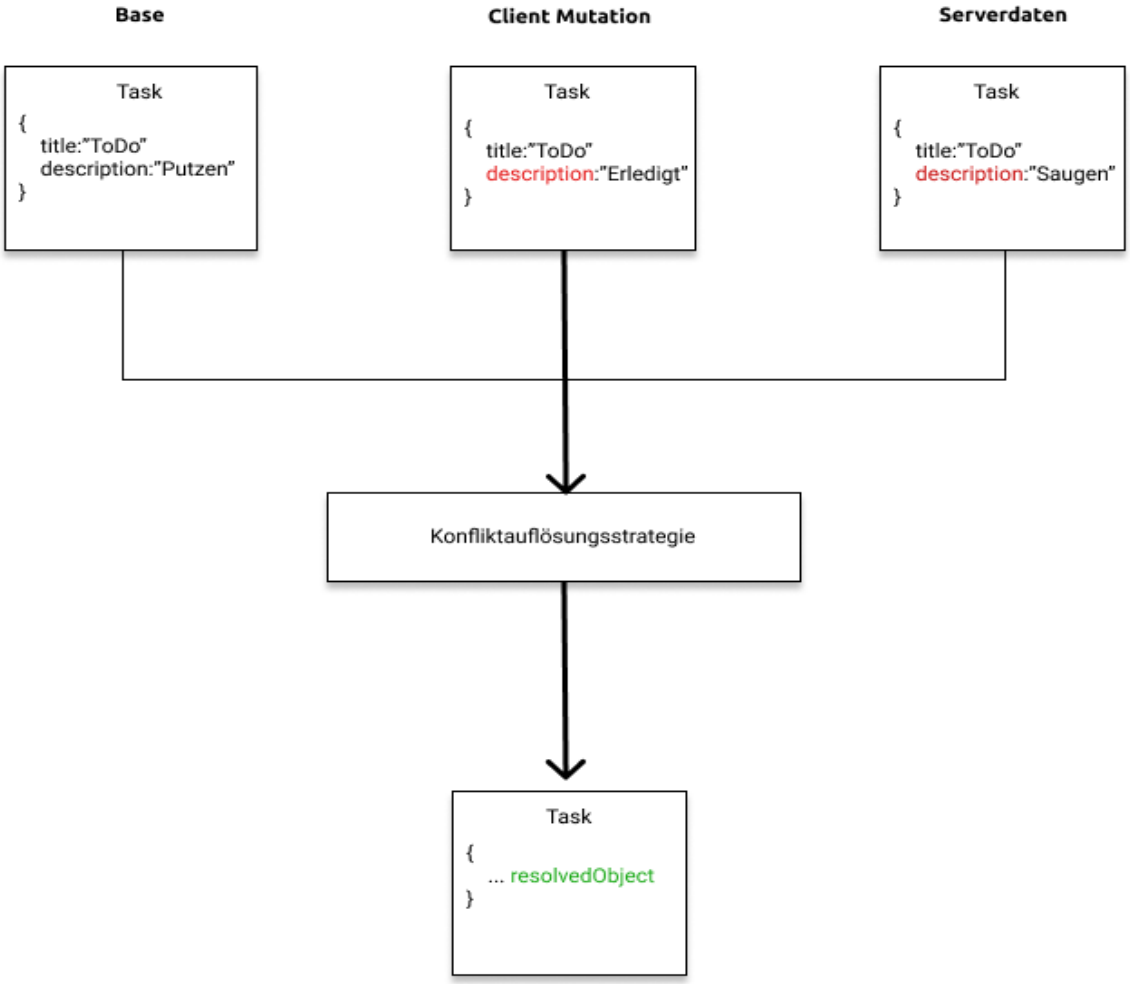

Abbildung 18: *Konfliktauflösung*

geschieht das Gleiche, nur dass das `base` Objekt mit dem `server` Objekt verglichen wird. Ein Konflikt ensteht, wenn ein Wert zwischen `base` und `server` Ungleich ist und genau dieser Wert auch in dem vorherig gespeicherten `clientDiff` Objekt existiert. `this.conflicted = true` signalisiert dann, dass ein Konflikt herrscht.

Basierend darauf wird im nächsten Schritt die Konfliktstrategie mit `conflictHandler.executeStrategy()` im *ConflictLink* ausgeführt. Dort wird der Status des Clients mit `assignServerState(client, server)` des Servers überschrieben. Die Datei *conflictStrategy.ts* führt dies durch:

```
1 public assignServerState(client: any, server: any): void {
2     client.updatedAt = server.updatedAt.toString();
3   }
```

Schlussendlich wird der aufgelöste Konflikt den *Listener* übergeben und der Konfliktlogger in der Datei `client/src/helpers/ConflictLogger.ts` gibt entweder die Meldung aus, dass ein Konflikt herrscht oder der Konflikt erfolgreich beseitigt wurde. Die Ausgabe erfolgt über die Konsole.

Im nachfolgenden Screenshot ist eine beispielhafte Konsolenausgabe zu sehen, wenn Client A einen Datensatz offline löscht und Client B darauffolgend versucht diesen Datensatz zu editieren. Client B löst den Konflikt auf und folgende Ausgabe erscheint dann in der Konsole:

Abbildung 19: *Konfliktlogger Konsolenausgabe*

Das `data`-Objekt ist die `base`, also die Daten vor Client und Serveränderungen, in diesem Fall Test 66, das `server`-Objekt sind serverseitigen Änderungen seit der letzten Bearbeitung, von welcher der Client wusste, Test 66 und das `client` -Objekt, welches Client B online zu Test 55 bearbeitet hat. Wie man sieht unterscheiden sich die Zahlenwerte beim Attribut `updatedAt` und anhand dessen wird der Konflikt aufgelöst.

Die gleiche Prozedur findet auch statt, wenn Client A einen Datensatz offline bearbeitet und danach Client B diesen online bearbeitet.

4.5 Progressive Web App

Anforderungen einer Progressive Web App

Damit eine *PWA* installierbar wird und somit erst zu einer Progressive Web App wird gelten für verschiedene Internetbrowser unterschiedliche Kriterien[43] [44][45][46][47]. Die Mindestanforderung aller Browser zusammen beläuft sich also auf:

- Die Web App sollte nicht schon installiert sein

- Muss über *HTTPS* geliefert werden - in Falle dieser Demo über *localhost*, was für Entwicklungszwecke auch zugelassen wird
- Benötigt ein *Web app manifest*
- Registrierung eines *Service Worker* - das bei der Erstellung einer *React*-Applikation durch *create-react-app* mitgeliefert wird und mit dem Befehl `serviceWorker.register()` registriert wird
- *Responsive design* welches mit *Ionic* möglich gemacht wird

Web App Manifest

Das *Web app manifest* ist eine *JSON*-Datei, die dem Internetbrowser befiehlt, wie sich die *Progressive Web App* verhalten soll wenn diese auf dem Desktop oder mobilen Gerät installiert wird. Eine typische *manifest*-Datei beinhaltet den Applikationsnamen, welches Icon benutzt werden soll und die Displaygröße und unter welcher URL die geöffnet werden soll, wenn die App gestartet wird.

Google Chrome, *Microsoft Edge*, *Mozilla Firefox*, *Apple Safari*, *UC Browser*, *Opera* und der *Samsung*-Browser unterstützen das *Web app manifest*.

Die *manifest* Datei kann jeden beliebigen Namen annehmen, gängig ist jedoch `manifest.json` und wird aus dem Wurzelverzeichniss der Web App bedient.
Die Datei setzt sich wie folgt zusammen:

```
{
  "short_name": "PWA",
  "name": "GraphQL DataSync PWA",
  "icons": [
    {
      "src": "/assets/icons/favicon.ico",
      "type": "image/x-icon",
      "sizes": "64x64 32x32 24x24 16x16",
      "purpose": "any maskable"
    },
    {
      "src": "/assets/icons/logo192.png",
      "type": "image/png",
      "sizes": "192x192",
      "purpose": "any maskable"
    },
    {
      "src": "/assets/icons/logo512.png",
      "type": "image/png",
      "sizes": "512x512",
      "purpose": "any maskable"
    },
     {
      "src": "/assets/icons/android-chrome-192x192.png",
      "type": "image/png",
      "sizes": "192x192",
```

```
      "purpose": "any maskable"
    },
     {
      "src": "/assets/icons/android-chrome-512x512.png",
      "type": "image/png",
      "sizes": "512x512",
      "purpose": "any maskable"
    }
  ],
  "start_url": ".",
  "scope": ".",
  "display": "fullscreen",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

short_name

Eine *manifest* Datei benötigt zumindest das Attribut `short_name` oder `name`. `short_name` ist der Anzeigename im *Home Screen*. `name` ist der Anzeigename im Browser nach der Installation.

icons

Wenn ein User die *PWA* installiert kann für den *home screen*, *splash screen*, *task switcher* und *app launcher* das Logo der App genutz werden. Das Attribut `icons` ist ein *Array* aus Bilderobjekten. Jedes Objekt beinhaltet das `src`, `sizes` und `type` Attribut. Um *maskable icons* zu nutzen wird die Einstellung `'purpose': 'any maskable'` benötigt. Manche Icons werden von *Android* nicht unterstützt und bekommen dann einen weißen Hintergrund. Mit dieser Einstellung wird der weiße Hintergund mit dem gewünschten Logo gefüllt[48]. Für *Chrome* müssen mindestens die Größen 192x192 und 512x512 Pixel angegeben werden. *Chrome* skaliert dann die *icons* für das entsprechende Gerät.

start_url

`start _url` teilt dem Browser mit, wo die Applikation starten soll. Das verhindert, dass währensd die App zum *Home Screen* hinzugefügt wird, nicht von der aktuell besuchten Seite startet. Die *URL* sollte einen User direkt in die Applikation leiten und nicht zur Homepage.

background_color

Diese Eigenschaft wird für den *splash screen* genutzt, wenn die Applikation zum ersten Mal auf einem mobilen Gerät gestartet wird.

background_display

Hiermit wird bestimmt wie groß der Anzeigebildschirm sein soll. Es gibt 4 Optionen:

- `fullscreen` - Die Webapplikation nimmt die Größe des verfügbaren Bildschirmbereichs ohne sichtbare Browserelemente ein. Das gibt das Gefühl einer nativen mobilen App

- `standalone` - Die App läuft in einem separaten Browserfenster und sieht aus wie eine eigenständige Applikation. Browserelemente wie die Adressleiste sind hierbei nicht sichtbar
- `minimal-ui` - Ähnlich wie `standalone` nur dass die *buttons* für die Navigation einer Webseite sichtbar bleiben
- `browser` - Einstellung wie bei einer normalen Webseite

scope

`scope` sind alle *URL*-Adressen die der Browser berücksichtigen soll. `scope` kontrolliert die *URL*-Struktur, welche alle Einstiegs-und Austiegspunkte der Web App umfasst. `start_url` muss sich innerhalb von `sope` befinden. Wird `scope` nicht in einer `masnifest`-Datei angegeben, ist die Standarteinstellung der Pfad von der die Web App aus geliefert wird. Mit `.` oder \ist das Wurzelverzeichniss gemeint und somit werden alle darunterliegenden Adressen berücksichtigt.

theme_color

Die Farbe der *tool bar* des Browser wird hiermit bestimmt. *theme_ color* sollte mit der Farbe im `meta` *tag* des Dokumentenkopfs `head` übereinstimmen.

In der Datei `/client/public/index.html` wird die *manifest*-Datei eingebunden

```
1 <link rel="manifest" href="/manifest.json" />
```

5 Evaluation

5.1 UI

Für den die Tests der Funktionaliäten der Applikation, sprich ob die Benutzeroberfläche korrekt funktioniert, werden folgende Geräte und Browser herangezogen.

- **Chrome**, **Firefox**, **Edge** - Google Pixel 3 (Android), Apple iPhone 12 (ios), Desktop (Ubuntu Linux)
- **Safari** - Apple iPhone 12 (ios)

Es wurden *Google Chrome*, *Mozilla Firefox*, *Microsoft Edge* und *Apple Safari*, da diese Internetbrowser sich die größten Marktanteile teilen[49].

Testergebnisse

Android	Chrome	Firefox	Edge
Installationsaufforderung	✓	x	x
Installierbar	✓	✓	✓
Offlinefähig	✓	✓	✓
UI fehlerfrei	✓	✓	✓

Tabelle 1: Testergebnisse Google Pixel 3

ios	Chrome	Firefox	Edge	Safari
Installationsaufforderung	x	x	x	x
Installierbar	x	x	x	✓
Offlinefähig	✓	✓	✓	✓
UI fehlerfrei	✓	✓	✓	teilweise

Tabelle 2: Testergebnisse Apple iPhone 12

Desktop	Chrome	Firefox	Edge	Safari
Installationsaufforderung	✓	x	✓	x
Installierbar	✓	x	✓	x
Offlinefähig	✓	✓	✓	✓
UI fehlerfrei	✓	✓	✓	✓

Tabelle 3: Testergebnisse Linux Ubuntu und MacOS Catalina

Android

Wie man aus der Tabelle 1 herauslesen kann, bietet der Internetbrowser *Chrome* von *Google* den besten Support auf dem Google Pixel 3, ein Android-Gerät. *Chrome* ist auch der einzige der 3 Browser der beim ersten Besuch der Webseite einen Vorschlag macht die *PWA* zum *Home screen* hinzuzufügen.

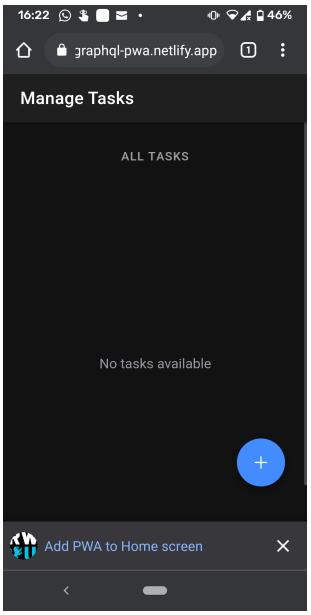

Abbildung 20: *Chrome Installationsauforderung*

In allen 3 Browser funktioniert die Applikation offline, ist auf dem *Home screen* installierbar und kann fehlerfrei im Vollbildschirmmodus verwendet werden. Um bei den Browser von *Microsoft* und *Mozilla* die *PWA* zum *Home screen* hinzuzufügen muss man in die Browsereinstellungen gehen und es dort manuell auswählen.

ios

ios es bietet nur bei seinem Betriebssystemeigenen Browser *Safari* die Möglichkeit die Applikation zum *Home screen* hinzuzufügen. Das einzige Problem dabei ist jedoch die korrekte Übernahme des *icon* der *PWA*.

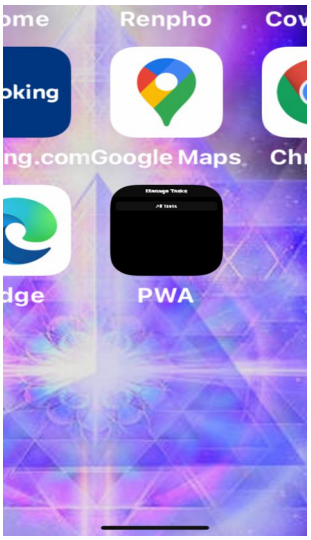

Abbildung 21: *ios home screen icon*

Wie zu sehen ist, wird hier als *icon* die Seite verwendet, die man zur Zeit des Hinzufügens zum *Home screen* besucht hat.

Desktop

Die Testergebnisse der Tabelle 3 zeigen, dass *Chrome* auch auf dem Desktop vollen Support für *PWAs* liefert. Auch *Microsoft* bietet volle Unterstützung. *Safari* und *Firefox* hingegen bieten keine Möglichkeit die Webapplikation auf den Desktop zu installieren.

Bei den Testergebnissen ist vor allem die Offlinefähigkeit herauszuheben, die von allen Geräten und Browser unterstützt wird. Die PWA funktioniert bei allen fehlerfrei und auch bei fehlender Internetverbindung. Die Resultate zeigen auch, dass Apple und *Firefox* weniger Support als *Chrome* und *Edge* aufweisen. Bei beiden Internetbrowser ist es nicht wirklich möglich eine native mobile Erfahrung zu schaffen. *Mozilla* kündigte sogar an den Destkop-Support komplett einzustellen[50].

5.2 Performance

Da bei dieser wissenschaftlichen Arbeit und allgemein bei einer *Progressive Web App* davon ausgegangen wird, dass eine Internetverbindung nur eine Erweiterung und somit ein Server nur selten erreicht werden kann, liegt der Fokus auf der clientseitigen Performance. Jede Applikation ist unterschiedlich und Performanceanfoderungen sind nicht immer offensichtlich. Aus diesem Grund wurden Entwurfsmuster zu Datenzugriff und Datenabfrage vorgestellt, an denen man sich richtet und so Optimierungen in Sachen Performance vollbringen kann. Vor allem müssen Anforderungen der Applikation eingiebig erörtert werden. Es ist wichtig zu wissen unter was für Bedingungen Nutzer die Applikation nutzen. Nur anhand dieser Kriterien kann dann eine für den geschäftszweck optimale Anwendung konzipiert werden.

Ein klassischer Kompromiss hinsichtlich der Performance ist Latenz gegen Durchsatz. Auch Nebenläufigkeit ist ein Faktor. Bei einer *PWA* sollte das Augenmerkmal auf darauf gerichtet werden so schnell wie möglich visuelle Ergebnisse zu bekommen, die Applikation funktionstüchtig zu bekommen und dann weniger abhängig vom Server bzw. der Internetverbindung zu sein. Insbesondere der

initiale Download der Applikationsdaten um danach auch offline direkt mit den Daten gearbeitet werden kann.

Für die Testergebnisse zur Performance der Applikation wird unter anderem das vom *Chrome*-Browser eigene Werkzeug *Lighthouse* verwendet.

Lighthouse

Einer der bekanntesten Metriken zurzeit ist der *Lighthouse Score*. Mit *Lighthouse* ist es möglich die Performance, beste Vorgehensweisen, Erreichbarkeit, Suchmaschinenoptimierung und Progressive Web Apps, einer Webseite zu untersuchen. Lighthouse simuliert eine erste Netzwerkanfrage ohne zwischengespeicherte Daten unter Netzwerkbendingungen wie 3G. Wichtig für diese wissenschaftliche Arbeit ist die Kategorie Performance, auf die nachfolgend eingegangen wird.

In der Kategorie Perfomance liegt der Fokus von *Lighthouse* auf 6 Metriken:

1. *First Contentful Paint* - Indiziert die Zeit bis ein erster Text oder Bild für den Nutzer sichtbar wird
2. *First Meaningful Paint* - Gibt an wann der Hauptinhalt der Webseite sichtbar wird
3. *Speed Index* - Gibt Auskunft darüber wie schnell der Inhalt der Seite geladen wird
4. *Time to interactive* - Die Zeit bis der User mit der Webapplikation interagieren kann
5. *First CPU Idle* - Liefert die Zeit bei der die Aktivität des Hauptthread der App am geringsten ist, um Nutzereingaben entgegenzunehmen
6. *Estimated Input Latency* - bezieht sich auch auf das vorherige Ergebnis, das in einem Zeitfenster von 5 Sekunden gemessen wird. Liegt die Latenz über 50 Millisekunden, nehmen User eine Webseite als langsam wahr

Die Tests wurden unter *localhost* und somit im eigenen Heimnetzwerk getestet. *Lighthouse* befindet sich im *Google Chrome*-Browser in den *Developer tools* und kann von dort aus gestartet werden.Unter der Einstellung *Device* kann ausgewählt werden ob man den Test für ein Desktopgerät durchführen will oder ein emuliertes mobiles Gerät. Für den Desktoptest wird ein *Lenovo* T450s Notebook, mit 16 GB RAM und einen i7 Prozessor unter *Linux Ubuntu* genutzt. Für den mobilen Test wird das Gerät *Motorola Moto G4* genutzt.

Test 1 Desktop

Der erste Test wurde ohne Applikationsdaten durchgeführt.

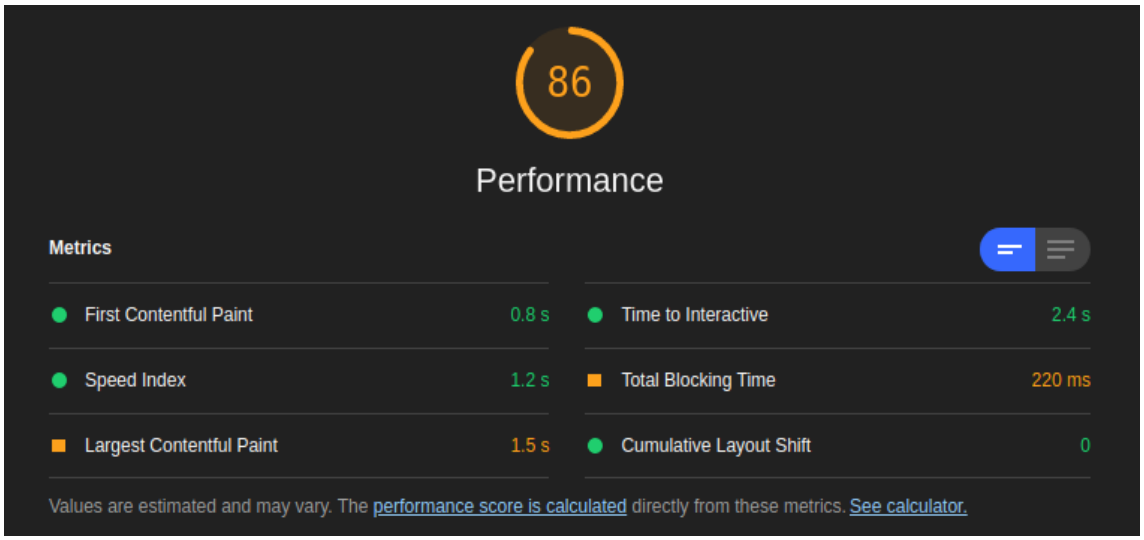

Abbildung 22: *Lighthouse Score Desktop ohne Datensätze*

Test 1 Mobil

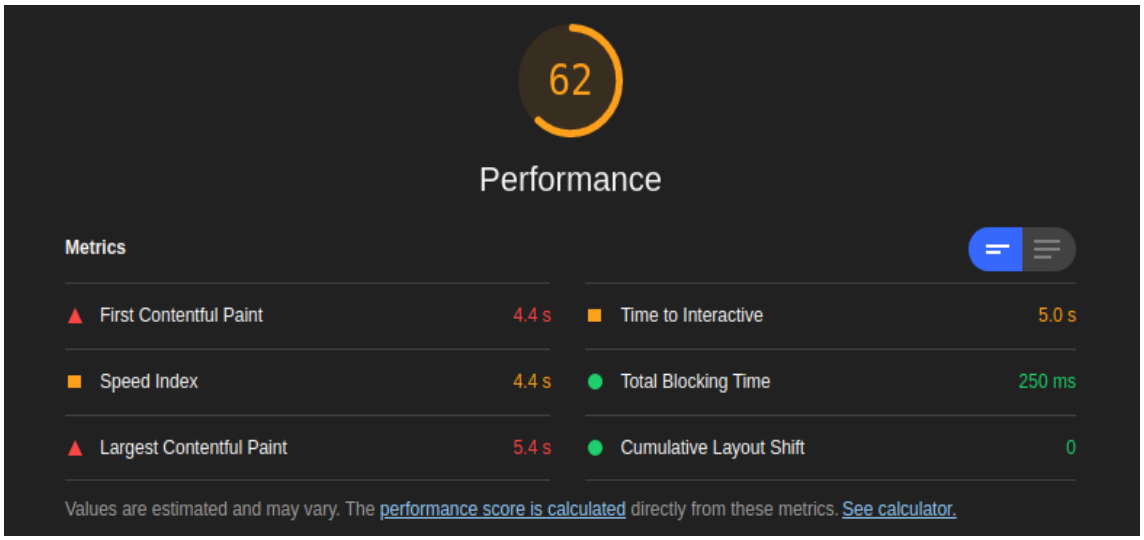

Abbildung 23: *Lighthouse Score mobil ohne Datensätze*

Test 2 Desktop

Der zweite Test wurde mit 10.000 Datensätzen ausgeführt.

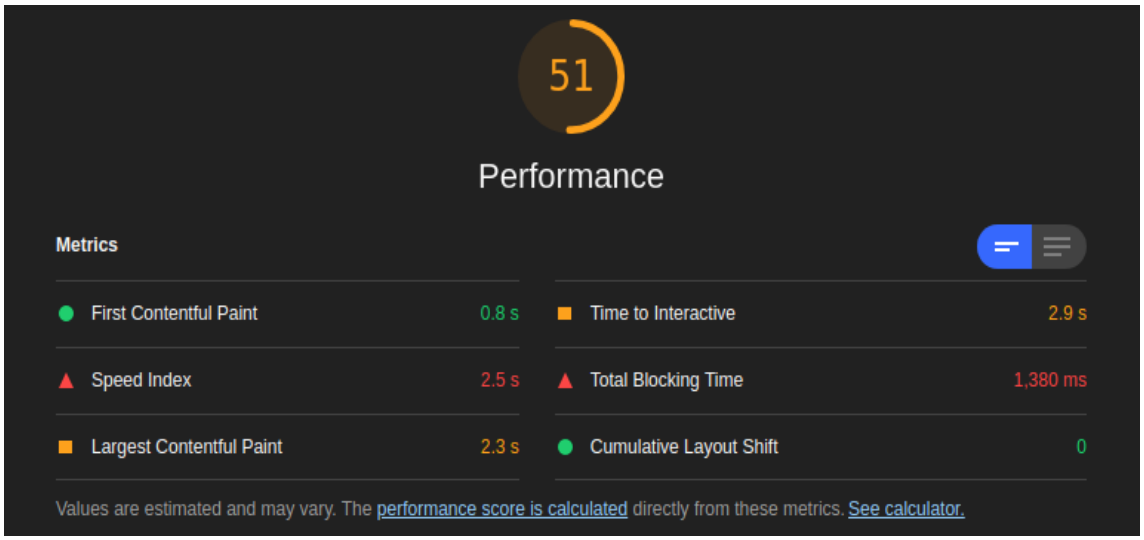

Abbildung 24: *Lighouse Score Desktop 10.000 Datensätze*

Test 2 Mobil

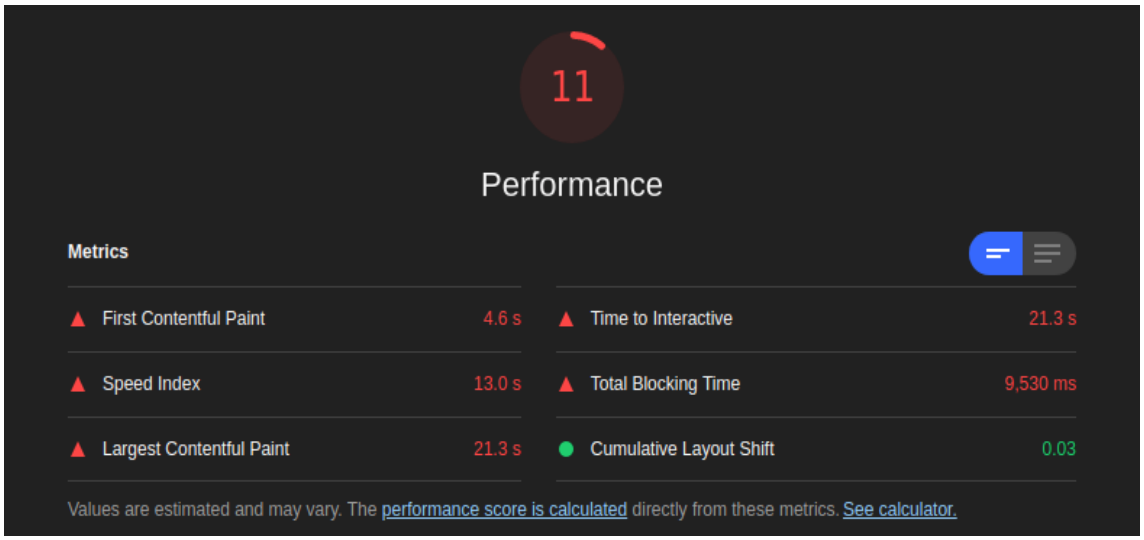

Abbildung 25: *Lighouse Score mobil 10.000 Datensätze*

Test 3 Desktop

Der dritte Test mit 100 000 Datensätzen.

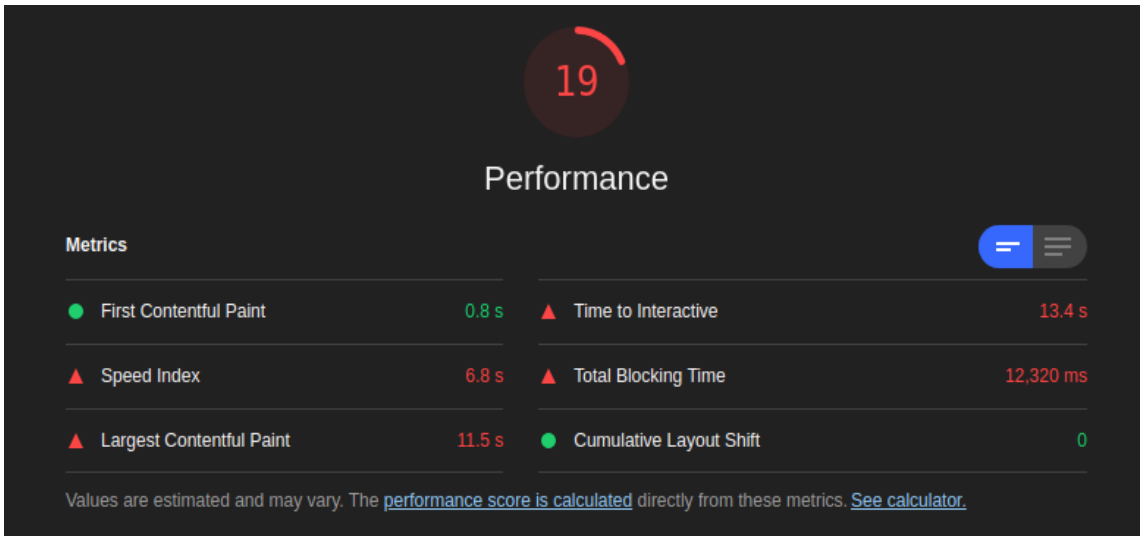

Abbildung 26: *Lighouse Score Desktop 100.000 Datensätze*

Test 3 Mobil

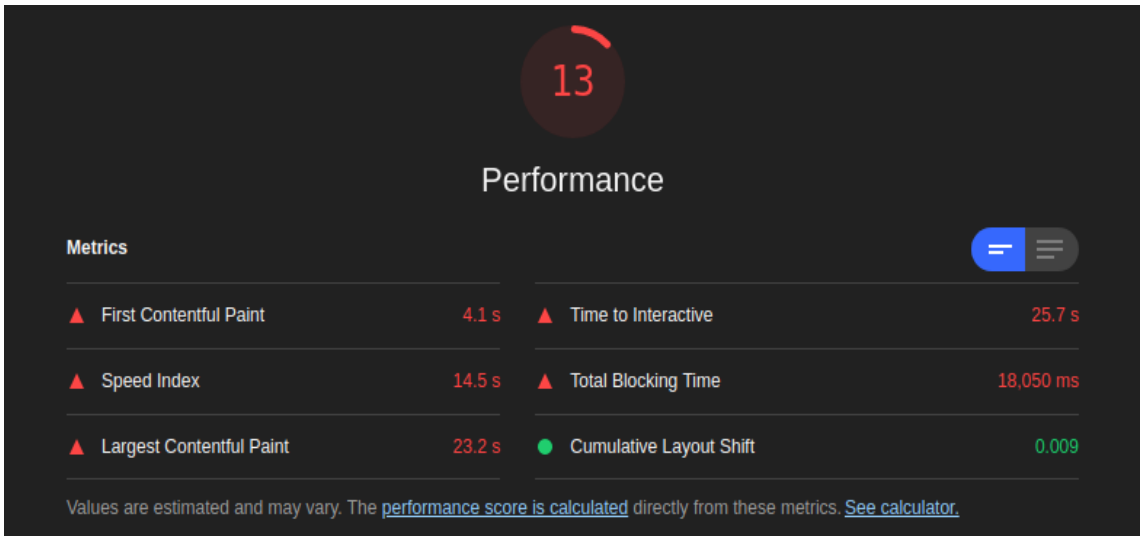

Abbildung 27: *Lighouse Score mobil 1.000.000 Datensätze*

Wie zu sehen ist, verschlechtert sich die Punktezahl mit Anzahl der Datensätze. Interessant ist jedoch, dass beim Test auf einem mobilen Gerät mit 100.000 Datensätzen die Performance besser als der Test mit der geringeren Anzahl an Datenätzen von 10.000 ist. Wie sehr man sich auf diesen Test verlassen kann steht noch zur Debatte. Denn abseits von diesen Tests es wurden noch 3 aufeinander folgende Tests mit gleicher Anzahl an Datensätzen und gleichen Bedingungen durchgeführt und erstaunlicherweise lieferte Lighthouse jedes Mal ein anderes Ergebnis. Google erwähnt jedoch in

einem ausführlichen Bericht[51], dass die Tests sehr gute Richtwerte sind, doch es Faktoren gibt welche die Punktzahl beeinflußen. Wie zum Beispiel

1. Werbung die in dem Moment des Tests eingeblendet wird
2. Wechselnde Routen beim Internetverkehr
3. Testen auf verschiedenen Geräten
4. Browser-Erweiterungen
5. Antivirus Software

Wobei 1., 3. und 5. ausgeschlossen werden kann, da im Heimnetzwerk getestet wurde. Genauso kann 4. ausgeschlossen werden, weil die Tests im privaten Modus (Ikognito-Fenster) durchgeführt wurden.

Bei jedem Test wird genau dokumentiert wie lange eine bestimmte Metrik benötigt und Lightouse gibt auch Vorschläge wie diese Metriken explizit verbessert werden können.

Initialer Datendownload

Die erste Netzwerkanfrage bei der 10.000 Datensätze der Applikation vom Server geladen werden und somit alle verfügbaren Applikationsdaten, dauert je nach Latenz zwischen 2-5 Minuten. Bei Download von 100.000 Datensätzen benötigt die Applikation ca. 34 Minuten.

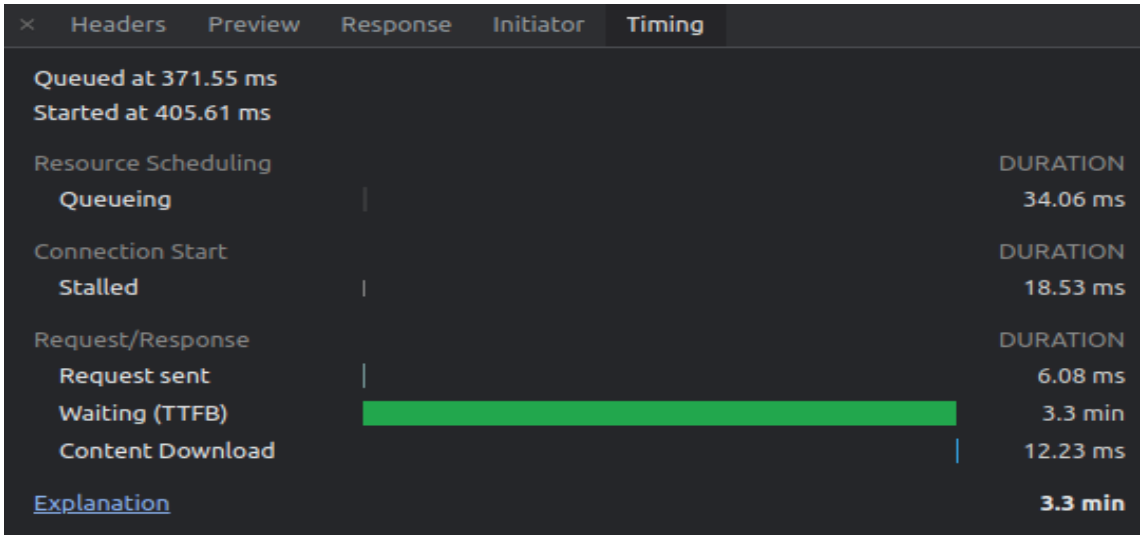

Abbildung 28: *Laden von 10.000 Datensätzen*

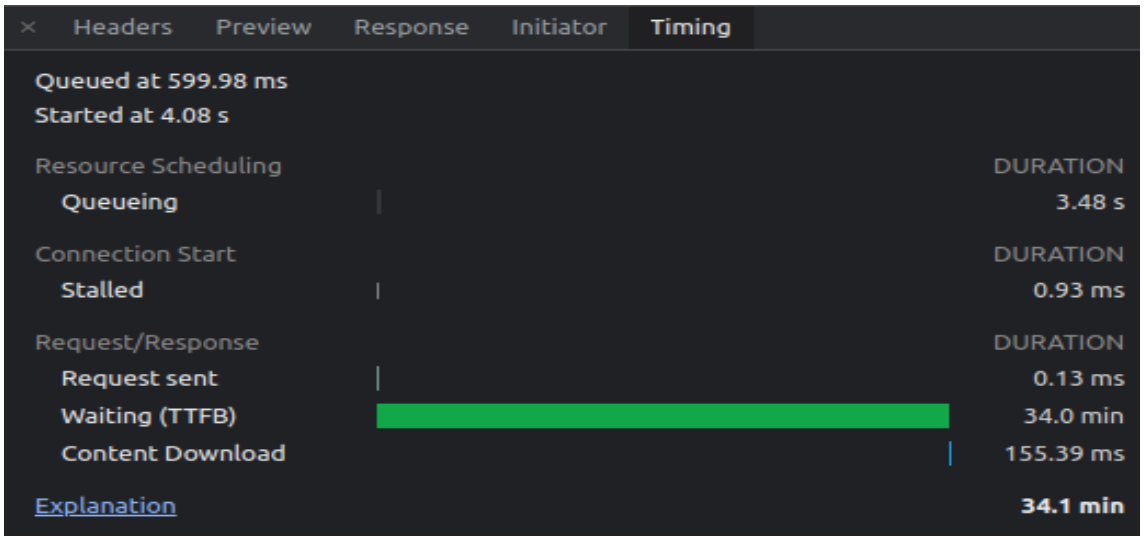

Abbildung 29: *Laden von 1.000.000 Datensätzen*

Die Angabe *(TTFB) Time to first byte* ist das Indiz dafür wie gut ein Server ansprechbar ist. *TTFB* misst die Zeit vom Zeitpunkt der Netzwerkanfrage des Clients, bis hin zum ersten *Byte* welches der Browser empfängt. Bei beiden Tests ist die Zeitspanne für viele Geschäftsanwendnungen nicht akzeptabel. Hier kann man daraus schließen, dass serverseiteige Optimierungen vorgenommen werden müssen. Denn die Angabe *Content Download* liefert akzeptable Zeiten im Millisekundenbereich. Das ist die Zeit vom Zeitpunkt des Empfangs des ersten *Byte* bis hin zum Download in den Browser.

Um genauere Gründe für die Dauer der initialen Downloads der Applikationsdaten zu finden, müssten mehrere und tiefergehende Tests durchgeführt werden. Vor allem im Bezug auf eine konkrete Geschäftsanwedung würde das mehr Aufschluss geben. Dann könnte man auch zum Beispiel die Performancegewinne von *Delta queries* messen. Wie schon erwähnt, nennt *Google* verschiedene Faktoren welche den Test der Performance beeinflussen. Hier könnten noch verschiedene Cloud-Computing Anbieter herangezogen werden. Eventuell beeinträchtigt die Ladezeit auch die Tatsache ob für einen Cloud-Service bezahlt wird oder nicht. Der Standort der Rechenzentren ist natürlich auch ein Faktor der die Ladezeit beeinträchtigt. Der *MongoDB*-Server für diesen Demonstrator befindet sich in Belgien und wird über den Cloud Anbieter *Google Cloud* bedient. Eine wahrscheinlich perfomantere Variante wäre die kostenpflichtige Variante mit einem Server aus Frankfurt über den Cloud Anbieter *Amazon Web Services*.

5.3 Responsiveness

Als nächstes gilt es zu erörtern wie ansprechbar die Applikation ist, nachdem alle Applikationsdaten geladen wurden und die Anwendung funktional bedient werden kann.

10.000 Datensätze

Bei der Verwendung der Applikation mit 10.000 Datensätzen git es für das Löschen, Entfernen, Bearbeiten und Anzeigen eines Task kaum Verzögerung. Die Operationen werden unmittelbar ausgeführt. Auch bei einer Seitenaktualisierung/Neustart der App gibt es kaum eine Verzögerung.

100.000 Datensätze

Bei der Verwendung der Applikation mit 100.000 Datensätzen kommt es bei allen Operatonen zu einer Verzögerung von 5-15 Sekunden. Ein App-Neustart beläuft sich auf ca. 10 Sekunden.

Schlussfolgernd ist zu sagen, dass der initiale Datendownload sehr viel Zeit in Anspruch nimmt und nicht für jede Geschäftsanwendung genutzt werden kann. Bedient man die App mit 10 000 Datensätzen bekommt man eine sehr gute *User Experience* sowohl online als auch offline. Denn durch die Anwendung von *Optimistic Response* ist das Ansprechverhalten nicht vom Netzwerk abhängig. Bedient man die App mit 100 000 Datensätzen sinkt das Ansprechverhalten drastisch. Diese Anzahl an Datensätzen führt zu einer schlechten *User Experience*. Der Grund für das schlechte Ansprechverhalten könnte daraus resultieren, dass die kompletten Applikationsdaten als einzelnes Objekt im Browserspeicher gespeichert werden. *apollo-cache-persist* führt dies durch um den *Apollo Cache* im Speicher zu halten. Es wird immer wieder erwähnt, dass *IndexedDB* eine asynchrone *API* ist welche den Hauptthread nicht blockiert. Das widerspricht sich mit der Tatsache, dass *IndexedDB* bei Speicherung eines Objektes den *structured clone algorithm* anwendet und dieser Algorithmus geschieht im Hauptthread, was bedeutet, je größer das Objekt ist, desto länger wird der Haupthread geblockt. Deshalb bietet es sich an jeden Datensatz als einzelnes Objekt mit einem Index abzuspeichern. Lese-und Schreiboperationen sollten nicht größer sein als die Daten auf die zugegriffen wird. Eine Aufteilung der Datensätze in mehrere Teile kann hier ein großen Vorteil bringen. Optimierungen könnten noch zusätzlich mit einem *Service Worker* duchgeführt werden. Indem der *Service Worker* alle Lese-und Schreib tätigt. Da der *Service Worker* in einem eigenen Thread abläuft kann die Auslagerung der *IndexedDB*-Operationen den Haupthread entlasten.

Ein anderer Grund für die schlechte *Responsivness* könnte die Tatsache sein, dass jeder Schreiboperation doppelt ausgeführt wird. Zu erst im *Apollo Cache* und dann durch die Bibliothek *apollo-cache-persist*, die bei jeder Schreiboperation ein komplettes Abbild des *Cache* in *IndexedDB* speichert. Vielleicht ist genau diese Prozedur zu rechenintensiv für größeren Datenmengen.

Ein zusätzlicher Faktor durch Nutzung von *IndexedDB* sind die Performanceunterschiede in verschiedenen Internetbrowser. Eine Studie aus 2017[52] von Ala'a Al-Shaikh und Azzam Sleit zeigte, dass der Internet Explorer die beste Performance unter den Browser, *Chrome*, *Firefox* und *Opera* hatte. Der Unterschied zwischen dem *Google Chrome*-Browser mit einer Gesamtperformance von 57,14 prozent und dem *Microsoft Internet Explorer* von 96,43 Prozent ist signifikant. Auch Owen Nolan zeigte die Performanceunterschiede in verschiedenen Browser[53].

Wie bereits erwähnt wird damit geworben, dass *IndexedDB* den *DOM* nicht blockiert. Erstaunlicherweise kann man das so pauschal nicht sagen. Nolan Lawson hat sich damit eingehend beschäftigt und Performancetests mit den gängigen Browsern, *Firefox*, *Safari*, *Chrome* und *Edge* durchgeführt[53]. Er kam zum Ergebniss, dass *IndexedDB* eine sehr komplizierte *API* hat und in viele Fällen den *DOM* doch blockiert. *IndexedDB* soll immer noch keine ausgereifte Lösung sein, doch trotzdem die beste aus Sicht der Performance. Vor allem in Kombination mit Nutzung eines *Service Worker*, wird der *DOM* gar nicht geblockt.

Erwähnenswert ist auch die Strategie zum Anzeigen der Daten. Es muss auf den Geschäftszweck abegstimmt werden, wie viele Daten der Nutzer zu einer bestimmten Zeit laden kann. Vor allem die Unterstützung von *Apollo GraphQL* durch *pagination* ist sehr hilfreich. Ursprünglich arbeitete man bei *pagination* so, dass bei Anfordern von neuen Daten einer Liste eine Netzwerkanfrage geschickt wurde. Nun kann man dies mit dem *Apollo Cache* so konfigurieren, dass die Daten direkt lokal vom *Cache* geladen werden.

Letzenendes ist es nicht trivial ganzheitliches Performancetests durchzuführen. Es bedingt ein explizite Geschäftsanwendung, Tests in der Produktion mit vielen Clients und sehr viel andere Tests in dem man herkömmlichen Anwendungen miteinander vergleicht.

PRPLE

Mit Einführung der *Progressive Web Apps* wurde auch das sogenannte *PRPLE* Entwurfsmuster vorgestellt. Das Ziel des Entwurfsmusters ist es, Webseiten so schnell wie möglich interaktiv zu machen.

- **Push/Preload** - Die kritischen Ressourcen, um erste visuelle Ergebnisse zu bekommen als erstes vom Server schicken
- **Render** - Die erste Route der Webapplikation als erstes rendern
- **Preload** - verbleibende Ressourcen vorladen
- **Lazy load** - andere Routen und Ressourcen nachladen

Durch dieses Entwurfsmuster kann der *Lighthouse Score* zusätzlich positiv beeinflusst werden[54].

6 Diskussion

Die vorgestellen Demonstratoren sind Applikationen mit relativ einfachen Funktionen und arbeiten nur mit einzelnen textbasierten *(JSON)* Daten. Das im Titel der wissenschaftlichen Arbeit verwendete "Handling von großen Datenmengen", bezieht sich auf die Anzahl an Datensätzen und nicht auf komplexere Datenstrukturen wie Bilder, Audio oder Video. Nichtsdestotrotz können viele der vorgestellten Konzepte genauso für Applikationen mit solchen Datenstrukturen umgesetzt werden. Es müssten jedoch andere Tests und Evaluationen durchgeführt werden.

Die hauptsächlich vorgestellte Applikation bietet die Möglichkeit der *Delta queries*, ist aber nicht als solche umgesetzt. Aus diesem Grund wurde eine zweite Applikation hinzugezogen, um das Konzept aufzuzeigen. Beide Demonstratoren verwenden als Datenbanktechnologie *MongoDB*. Daher können die Ergebnisse nur in Kontext mit diesem Datenbankanbieter und speziellen Server in Belgien in Verbindung gebracht werden.

Die zuerst vorgestellte Webanwendung, ist mit der Anfragestrategie an den Server standartmäßig so eingestellt, dass beim jedem Aufruf der Hauptkomponente mit den angezeigten Tasks, eine Frage an den Server und parallel an den *Cache* gesendet wird. Für Anwendungen die große Datenmengen handhaben, ist das nicht effizient. Es würden nämlich jedes Mal alle Datensätzen beim Server angefragt. Aus diesem Grund wurde die Applikation so eingestellt, dass immer zuerst der *Cache* abgefragt wird, bevor eine Netzwerkanfrage an den Server gesendet wird. Ein weiteres Problem ergibt sich beim Testen der Anwendung im Offlinemodus. Möchte man 2 Clients simulieren, sprich 2 verschiedene Tabs oder gar 2 verschiedene Browserfenster, werden die getätigten Operationen, trotzdem über die *WebSockets* an den anderen Client gesendet. Das liegt daran, dass im Heimnetzwerk *localhost* getestet wurde und die *WebSockets* somit die Nachricht trotzdem abschicken, weil alle Tabs im selben *Google Chrome* Prozess ablaufen. Wenn die Applikation über eine Domain mit Webadressen geliefert wird, ensteht dieses Problem nicht mehr.

Die Evaluation mit den Ergebnissen der Tests ergab, dass bei Anstieg der Datensätz die Applikation immer weniger ansprechbar wurde. Mit 10 000 Datensätzen läuft die Webapplikation sehr flüßig und erfüllt fast alle gestelleten Anforderungen. Bei 100 000 Datensätzen ist die Webapplikation kaum nutzbar und hat sehr langsame Antwortzeiten. Wie schon erwähnt könnte das draus resultieren, dass *IndexedDB* den *DOM* blockiert. Unglücklicherweise ist es mir nicht gelungen genau zu erschließen warum der initale Download aller Datensätze so viel Zeit in Anspruch nimmt. Genau so ist es mir nicht ersichtlich geworden warum die App ab 100 000 Datensätze dermaßen langsam reagiert. Es müsste mehr Forschung in dieser Hinsicht betrieben werden. Vor allem in Hinblick auf den Browserspeicher *IndexedDB* und die Bibliothek *apollo-cache-persist*.

Im Vorfeld dieser wissenschaftlichen Arbeit wurde noch eine zusätzliche Applikation mit *JavaScript* und *React.js* entwickelt[55], ohne *GraphQL*, Datensynchronistation und Konflikthandling. Es wurde ein herkömmlicher *REST*-Ansatz und *Redux* für das Datenmanagement genutzt. Für eine trivialere Interaktion mit dem Browserspeicher *IndexedDB* wurde die Bibliothek *dexie.js* implementiert. Bei dem Test mit 100 000 Datensätzen lief die Applikation sehr flüssig. Der initiale Download belief sich auf ca. 5-10 Sekunden und alle Lese-und Schreiboperationen wurden unmittelbar ausgeführt. Um weitere Aussagen treffen zu können, müsste der Applikation noch ein Synchronisations-und Konfliktmechanismus hinzugefügt werden und mehrere Tests durchgeführt werden. Im Anhang befindet sich der dazugehörige Quelltext.

Es war die Rede davon, dass es Ineffizient ist mehrere Datensätze in einem einzelnen Objekt zu speichern, aufgrund des *structured clone algorithm* welcher den *DOM* blockiert. Die im Vorfeld entwickelte Applikation hat genau diesen Ansatz vermieden und jeden Datensatz einzeln mit eigenem Index in *IndexedDB* gespeichert. Das Resultat ist eine Applikation die selbst mit 100.000 Datensätzen ein sehr gutes Ansprechverhalten hat. Vielversprechend wäre eine Applikation, welche die Vorzüge der Reduktion der Netzwerkanfragen durch *GraphQL* nutzt und das Speichern aller Datensätze als einzelne Einträge durch *dexie.js* in *IndexedDB*. Lässt man eventuell den *Apollo Cache* weg und führt jede Lese-und Schreiboperation direkt in *IndexedDB* aus, könnte die Performance optimiert werden.

Aufschlussreich waren die Unterstützung der verschiedenen Internetbrowser für *Progressive Web Apps*. *Firefox* und *Safari* weisen geringeren Support für dieses Konzept auf. Das Ziel der *Progressive Web Apps* ist es sowie nah wie möglich an die *User Experience* einer nativen Applikation zu kommen. Vor allem *Apple* lässt es in dem Sinne nicht vollkommen zu, weil es nicht möglich ist sich die Webapplikation auf dem *Home screen* installieren zu lassen und man sich somit kein Vollbildschirrm generieren lassen kann. Offensichtlich ist es, dass *Apple* einen großen Umsatz sowie Marktanteil an nativen Applikationen besitzt. Es wird viel in die Forschung und Entwicklung nativer Applikationen für die Plattformen *MacOS* und *ios* betrieben. *Progressive Web Apps* würden für Umdenken und Umstrukturierung sorgen.

7 Fazit

Webapplikationen mit dem *Offline-first* Gedanken zu entwickeln ist herausfordernd. *Offline-first* ist ein relativ neues Paradigma, mit dem Applikationen zuerst unter der Annahme entwickelt werden, es gäbe keine Verbindung zum Internet und dann wird eine Anwendung schrittweise mit Onlinefunktionalitäten erweitert. Im Endeffekt entwickelt man ein komplexes verteiltes System.

Diese wissenschaftliche Arbeit untersuchte in wie weit es möglich ist, eine Webapplikation offline zu nutzen, große Datenmengen zu einem Server zu synchronisieren und dabei trotzdem eine gute *User Experience* beizubehalten. Es wurden aktuellste Technologien der Industrie wie *TypeScript*, *React.js*, *Apollo GraphQL* genutzt und Konzepte der Datensynchronisation vorgestellt. Umgesetzt wurde dies anhand zweier Demonstratoren durch die *GraphQL*-Bibliotheken *Offix* und *Graphback*, mit denen es möglich ist, Datensynchronisation und Konfliktauflösung zu realisieren. Zusätzlich dazu wurde im Vorfeld ein dritter Demonstrator ohne *GraphQL* entwickelt, mit dem es trotz 100.000 gespeicherten Datensätzen im Client, keine verringerte *User Experience* gab. Datensynchronisation und Konfliktauflösung konnten mit dem ersten Demonstrator aufgezeigt werden. Für das Konzept der *Delta Query*, das Daten abfragt welche in Offlineperioden verpasst wurden, konnte ein zweiter Demonstrator herangezogen werden.

Aufgrund der schlechten Perfomanceergebnisse bei Datenhaltung und Download von 100.000 Datensätzen des ersten Demonstrators, wurde der im Vorfeld ohne *GraphQL* entwickelte Demonstrator wieder herangezogen und erwähnt. Fakt ist, dass der erste Demonstrator im aktuellen Zustand Probleme beim Handling größerer Datenmengen hat. In erster Linie ist der initale Datendownload für 10 000 und 100 000 Datensätze nicht akzeptabel. Die Bedienung der App mit 10.000 Datensätzen hingegen ist flüssig.

Generell müssten mehrere Tests durchgeführt werden. Mehr Variationen mit der Anzahl an Datensätzen, Verwendung der Applikation in der Produktion durch eine große Zahl an Nutzer, Serveranbindung an verschiedene Cloud-Anbieter und die Gegenüberstellung von zwei Applikationen, bei der eine davon mit *Apollo GraphQL* und die andere mit einem herkömmlichen *REST*-Ansatz entwickelt wird.

Die vorgestellten Technologien und Konzepte wirken jedoch sehr vielversprechend. Insbesondere *Service Worker API*, *IndexedDB*, *Apollo Link* und das Konzept der *Delta Query* bieten Entwicklern starke Werkzeuge, um perfomante, zuverlässige und offlinefähige Applikationen zu entwickeln.

Die Anforderungen einer *Progressive Web App* sind ein sehr guter Richtwert für die Entwicklung von offlinefähigen Webapplikationen. Auch wenn *Apple* geringeren Support als andere Internetbrowser-Anbieter für *Progressive Web Apps* bietet, verändert dies nicht die Tatsache, dass die Entwicklung einer *PWA* auf Performance und plattformübergreifende Nutzung ausgelegt ist und damit der Grundstein für jede Webapplikation sein sollte.

Es hat sich herausgestellt, dass es beim Entwickeln offlinefähiger Applikationen entscheident ist, den Geschäftszweck genau zu erötern, um dann die Webanwendung dementsprechend darauf abzustimmen. Eine generelle Lösung exsitiert unglücklicherweise nicht dafür. Es gibt eine unbestimmte Anzahl an Situationen im Programmcode, die vor allem bei offlinefähigen Webapplikationen nicht abgefangen werden können. Diese *edge cases* werden meistens über Jahre hinweg bekannt. Deswegen ist es unabdingbar die Anforderungen der Applikation genau festzulegen, um viele der Situationen einzugrenzen.

8 Ausblick

Bei der Technologierecherche wurden unter anderem auch andere Implementierungen der Industrie in Betracht gezogen. Das Ziel jedoch war es, eine Implementierung zu finden, bei der, der Quellcode frei einsehbar ist und eine aktive Community daran arbeitet. Alternative kostenpflichtige Implementierungen im Bereich der Offlineapplikationen wären zum Beispiel, *MongoDB Realm* oder *Dexie Cloud*. Ein sehr interessanter Ansatz verfolgt auch die kostenpflichtige Variante von *Amazon* durch *AWS AppSync*. *AWS AppSync* nutzt einige Konzepte in dieser vorgestellten wissenschaftlichen Arbeit, wie zum Beispiel *Delta Query* und setzt auch das Datenhandling mit *GraphQL* um.

Zu erwähnen wären noch Kosten-und quellfreie Alternativen wie *Meteor.js* und *workbox*. Vor allen Dingen *workbox* von *Google* ist eine mächtige Bibliothek, die eine Sammlung an Werkzeugen ist, um vor allem mit der *Service Worker API* und den Browsertools zu interagieren. Der im Vorfeld genutze Demonstrator hat sich eingehend mit *workbox* beschäftigt und wird eventuell in einer weiteren wissenschaftlichen Arbeit genauer untersucht.

Da eine offlinefähige Webapplikationen stark von der Speicherung der Daten im Client und somit im Internetbrowser ist, sollte in Zukunft noch mehr Forschung in dieser Richtung betrieben werden. *IndexedDB* scheint momentan die einzig sinnvolle Lösung zu sein, gößere Datenmengen im Internetbrowser zu speichern. *IndexedDB* sollte aus diesem Grund auch noch genauer untersucht werden.

Der Markt bietet einige Angebote für Lösungen, bei denen der Hauptfokus auf Offlinefähigkeit mit korrekter Datensynchronisation liegt. In wie weit all diese Lösungen große Datenmengen handhaben können, müsste noch erforscht werden.

Mit der vorgestellten wissenschaftlichen Arbeit wurde unter anderem gezeigt, wie wichtig Offlinefähigkeit und Dezentralisierung sein können. Offlinefähige Webapplikationen sind nämlich genau das, dezentral. Es sollte ein Umdenken in Unternehmen und bei Entwicklern geben, die anerkennen müssen, dass mit dem *Offline-first* Gedanken, ausfallsichere, zuverlässigere und perfomantere Webapplikationen entwickelt werden können.

Glossar

API Eine API (Application Programming Interface) ist Quellcode der es ermöglicht, dass Softwaresysteme miteinander kommunizieren können. Sie ist die Schnittstelle zur Außenwelt, somit können Entwickler mit Programmcode, Services anderer Entwickler und Unternehmen in ihre eigene Applikation einpflegen . 14

frame Ein frame im Video und Animationsbereich sind individuelle Bilder in einer Sequenz von Bildern welche sich dann zu einem flüßigen Ablauf entwickeln und somit ein Video enstehen lässt. 12

JSON JavaScript Object Notation ist ein leichtgewichtiges Dateninteraktionsformat. Es besteh aus einer Sammlung aus Schlüssel-Werte Paaren. Sie wird größtenteils für den Datenaustausch zwischen einem Server und Client genutzt . 14

Master-Detail-Ansicht Eine Master-Detail-Ansicht oder Master-Detail-Beziehung bedeutet, dass zur Hauptinformation noch weitere Informationen dazugehören. In einer Tabelle mit Daten, ist die Masteransicht die primär sichtbaren Daten und die Detailansicht die dazugehörigen Details nach einem Klick auf die Masteransicht . 31

NoSQL-Datenbank Diese Typen von Datenbanken sind speziell für Anwendungen optimiert, die große Datenmengen, geringe Latenz sowie flexible Datenmodelle erfordern. Die Datensätze werden in Form von JSON gespeichert. 20

Progressive Enhancement Progressive Enhancement ist eine Design-Philosophie bei der es darum geht, so vielen Nutzer*innen wie möglich die grundlegenden Inhalte und Funktionen einer Website zur Verfügung zur stellen. Darüber hinaus wird Nutzer*innen mit einem modernen Browser, der den dafür notwendigen Code ausführen kann, das bestmögliche Erlebnis geboten[56] . 16

User Experience Ein frame im Video und Animationsbereich sind individuelle Bilder in einer Sequenz von Bildern welche sich dann zu einem flüßigen Ablauf entwickeln und somit ein Video ensteht. 4

World Wide Web Consortium (W3C) Das World Wide Web Consortium ist eine internationale Community, bei der Vollzeitangestellte, Mitgliedorganisationen und die öffentliche Arbeit zusammen Web Standards entwickeln . 4

Literaturverzeichnis

[1]statcounter, *Desktop vs Mobile vs Tablet Market Share Worldwide.* (2021) https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/2016.

[2]I. H. W. W. G. Anne van Kesteren, *Offline Web Applications.* (2008) https://www.w3.org/TR/offline-webapps/.

[3]A. Russell, *Progressive Web Apps: Escaping Tabs Without Losing Our Soul.* (2015) https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/.

[4]A. Christopher, „A Pattern Language: Towns, Buildings, Construction.“, in (Oxford University Press, 1977) Kap. x.

[5]Z. M. D. Schmidt, *Data Synchronization Patterns in Mobile Applications Design.* (2012) https://www.dre.vanderbilt.edu/~schmidt/PDF/PatternPaperv11.pdf.

[6]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 3.1 Asynchronous Data Synchronization Visual Explanation.

[7]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 3.2 Synchronous Data Synchronization Visual Explanation.

[8]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 4.1 Partial Storage Visual Explanation.

[9]Google, *Offlinekarten.* (2021) https://support.google.com/maps/answer/6291838?co=GENIE.Platform=Android&hl=de.

[10]M. Kirchner, *Eager Acquisition.* (2002) http://kircher-schwanninger.de/michael/publications/EagerAcquisition.pdf.

[11]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 4.2 Complete Storage Visual Explanation.

[12]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 5.1 Full Transfer Visual Explanation.

[13]M. Kirchner, *Lazy Acquisition.* (2002) http://kircher-schwanninger.de/michael/publications/LazyAcquisition.pdf.

[14]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 5.2 Timestamp Visual Explanation.

[15]S. A. Ari Trachtenberg David Starobinski, *Fast PDA Synchronization Using Characteristic Polynomial Interpolation.* (2002) http://people.bu.edu/staro/infocom02pda.pdf.

[16]Z. M. D. Schmidt, „Data Synchronization Patterns in Mobile Applications Design.“, in (Vanderbilt University, 2012) Kap. 5.3 Mathematical Transfer Visual Explanation.

[17]U. o. S. Tonci Dadic, *Delta Replication.* (2005) https://www.bib.irb.hr/507928/download/507928.Dadic_Eurocon2005.pdf.

[18]SAP, *Delta Replication.* (2016) https://patentscope.wipo.int/search/de/detail.jsf?docId=US219985786&tab=PCTDESCRIPTION.

[19]U. o. Z. Patrick Ziegler, *User-Specific Semantic Integration of Heterogeneous Data: What Remains to be Done?.* (2004) https://files.ifi.uzh.ch/hkunz/techreports/TR-2004/ifi-2004.01.pdf.

[20]J. Lambert, *Offline First – A better HTML5 User Experience.* (2012) https://www.joelambert.co.uk/article/offline-first-a-better-html5-user-experience/.

[21]D. Sauble, *Offline First Web Development* (Packt, 2015).

[22]G. Cselle, *Tales of Creation.* (2012) https://medium.com/gabor/every-step-costs-you-20-of-users-b613a804c329.

[23]G. I/O, *Progressive Web Apps across all frameworks.* (2016) https://www.youtube.com/watch?v=srdKq0DckXQ.

[24]G. Addy Osmani, *Getting Started with Progressive Web Apps.* (2015) https://developers.google.com/web/updates/2015/12/getting-started-pwa.

[25]M. M. Google Apple, *Bringing the Web up to Speed with WebAssembly.* (2017) https://dl.acm.org/doi/pdf/10.1145/3062341.3062363.

[26]N. I. of Standards und Technology, *The NIST Definition of Cloud Computing.* (2011) https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[27]N. I. of Standards und Technology, *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019.* (2019) https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g.

[28]MongoDB, *Die Datenbank für moderne Anwendungen.* (2021) https://www.mongodb.com/de-de.

[29]M. MDN Web Docs, *Cache.* (2021) https://developer.mozilla.org/en-US/docs/Web/API/Cache.

[30]W. World Wide Web Consortium, *Web SQL Database.* (2021) https://www.w3.org/TR/webdatabase/.

[31]Facebook, *Create React App.* (2021) https://github.com/facebook/create-react-app.

[32]S. Overflow, *Stack Overflow Developer Survey 2020.* (2020) https://insights.stackoverflow.com/survey/2020#most-popular-technologies.

[33]NPM, *Download statistics for package typescript.* (2021) https://npm-stat.com/charts.html?package=typescript&from=2010-01-28&to=2021-07-27.

[34]R. MacPherson, *The Fullstack Tutorial for GraphQL.* (2021) https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

[35]R. MacPherson, *The Fullstack Tutorial for GraphQL.* (2021) https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

[36]A. Ignjatovic, *Understanding and fixing N+1 query.* (2020) https://medium.com/doctolib/understanding-and-fixing-n-1-query-30623109fe89.

[37]A. GraphQL, *apollo-cache-persist.* (2021) https://github.com/apollographql/apollo-cache-persist.

[38]W. Trocki, *Data Query Patterns for Apollo GraphQL client.* (2019) https://medium.com/@wtr/data-query-patterns-for-graphql-clients-af66830531aa.

[39]aerogear, *delta queries.* (2020) https://graphqlcrud.org/docs/next/spec-datasync/delta-queries.

[40]aerogear, *AeroGear DataSync Starter.* https://github.com/aerogear/datasync-starter.

[41]B. S. Adam Connors, *Datenbank für browserunterstützende Technologien.* (2021) https://caniuse.com/.

[42]aerogear, *react-datastore.* https://github.com/aerogear/offix/tree/master/examples/react-datastore.

[43]P. LePage, *https://web.dev/install-criteria/.* https://web.dev/install-criteria/.

[44]M. MDN Web Docs, *How do you make an app A2HS-ready?.* https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Add_to_home_screen#how_do_you_make_an_app_a2hs-ready.

[45]S. D. Hub, *Ambient Badging.* https://hub.samsunginter.net/docs/ambient-badging/.

[46]Dev.Opera, *Installable Web Apps and Add to Home screen.* https://dev.opera.com/articles/installable-web-apps/.

[47]U. Plus, *Add to Homescreen.* https://plus.ucweb.com/docs/pwa/docs-en/zvrh56.

[48]T. S. Tiger Oakes, *Adaptive icon support in PWAs with maskable icons.* https://web.dev/maskable-icon/.

[49]statcounter, *Browser Market Share Worldwide.* https://gs.statcounter.com/browser-market-share.

[50]entwickler.de, *Firefox beendet PWA-Support für den Desktop.* https://entwickler.de/javascript/firefox-beendet-pwa-support-fur-den-desktop.

[51]Google, *Lighthouse Metric Variability and Accuracy.* https://docs.google.com/document/d/1BqtL-nG53rxWOI5ROOpItSRPowZVnYJ_gBEQCJ5EeUE/edit.

[52]A. S. Ala'a Al-Shaikh, *Evaluating IndexedDB Performance on Web Browsers.* (2015) https://www.researchgate.net/publication/320614525_Evaluating_IndexedDB_Performance_on_Web_Browsers/link/59f05ae6aca272cdc7ca25e4/download.

[53]N. Lawson, *IndexedDB, WebSQL, LocalStorage – what blocks the DOM?.* (2015) https://nolanlawson.com/2015/09/29/indexeddb-websql-localstorage-what-blocks-the-dom/.

[54]H. Djirdeh, *Apply instant loading with the PRPL pattern .* (2018) https://web.dev/apply-instant-loading-with-prpl/.

[55]D. Erdelean, *PWA.* (2021) https://github.com/DeLean/PWA/tree/non-graphql.

[56]M. MDN Web Docs, *Progressive Enhancement.* (2021) https://developer.mozilla.org/de/docs/Glossary/Progressive_Enhancement.

Abbildungsverzeichnis

Anhang

1. Demonstrator

Der erste Demonstrator wurde auf Basis dieses Quellcodes entwickelt:
https://github.com/aerogear/datasync-starter

2. Demonstrator

https://github.com/aerogear/offix/tree/master/examples/react-datastore

3. Demonstrator

Der 3. Demonstrator wurde unter einer *MIT*-Lizenz auf *GitHub* geladen:
https://github.com/DeLean/PWA/tree/non-graphql