Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Masterarbeit

# Entwicklung einer deklarativen Offline-First Applikation unter dem Backend-as-a-Service Paradigma

**Kevin Twesten**

4twesten@informatik.uni-hamburg.de
Studiengang Wirtschaftsinformatik
Matr.-Nr. 6714157
Fachsemester 6

| | |
|---|---|
| Erstgutachter: | Prof. Dr. –Ing. Nobert Ritter |
| Zweitgutachter: | Dr. –Inform. Fabian Panse |
| Abgabe: | 22.06.2017 |

# Inhaltsverzeichnis

# 1 Einleitung

(Web-)Applikation stellen im Bereich der Anwendungsprogramme eine essenzielle Revolution dar. Denn seit ihrer Entwicklung müssen Anwendungen nicht länger umständlich auf den eigenen lokalen Geräten installiert werden, sondern können einheitlich über den Webbrowser der Wahl angesteuert und visualisiert werden. Dass die Voraussetzung für die Nutzung dieser (Web-)Applikation in einer stabilen Netzwerkverbindung liegt, kann hierbei jedoch leicht in Vergessenheit geraten. Doch gerade durch die ansteigende Nutzung von mobilen Endgeräten, wie Smartphones und Tablets und den damit verbundenen, notwendigen Bedarf einer mobilen Internetverbindung, wird eine unangenehme Wahrheit stetig offensichtlicher. Die Verfügbarkeit von dauerhaft stabilem Internet kann nicht immer gewährleistet werden.

Aus diesem Grund entstand vor einigen Jahren ein neuer Ansatz bei der Entwicklung von (Web-)Applikation. Dieser ist vor allem durch die steigende Nutzung von mobilen Endgeräten zu begründen und wird als „Offline-First“ bezeichnet. Ziel dieses Ansatzes ist es, die Verwendung von (Web-)Applikation auch dann zu ermöglichen, wenn temporär keine stabile Netzwerkverbindung hergestellt werden kann. Eine besondere Herausforderung hierbei, ist die Entwicklung eines zuverlässigen Storage Konzeptes zur Sicherstellung einer konsistenten Datenbasis. Diese Materie wird durch die vorliegende Masterarbeit mit dem Titel „Declarative Offline-First Application Development under the Backend-as-a-Service Paradigm“ thematisiert.

## 1.1 Motivation

Das Internet – ein Netzwerk aus unzähligen, autonomen Systemen, welche es ermöglichen, Informationen wie nie zuvor zu verteilen und zur Verfügung zu stellen. Ein Medium, dass die Möglichkeiten von Organisationen und Individuen so stark beeinflusst hat und noch immer prägt, wie kaum ein Phänomen zuvor. Die Online-Studie von ARD und ZDF aus dem Jahr 2015, bei welcher 1800 Personen ab 14 Jahren befragt wurden, hat ergeben, dass 63,1% der Deutschen, das Internet täglich nutzen. Das ist ein Zuwachs von 8,5% gegenüber dem Vorjahr. Hierbei nimmt vor allem die Nutzung von mobilen Geräten kontinuierlich zu. Bereits 55% der Befragten geben an, das Internet auch mobil zu nutzen [AZ15]. Bei der Betrachtung der steigenden Verbreitung von mobilen Endgeräten (Smartphones, Tablets etc.) wird dieser Trend noch deutlicher. Im Jahr 2015 waren 7,9 Milliarden mobiler Endgeräte im Umlauf. Dies entspricht einem erneuten Anstieg um 600 Millionen Geräte.

Neben der steigenden Verbreitung von mobilen Endgeräten, nimmt auch der Datenverkehr pro Gerät stark zu. Im Jahr 2015 ist der durchschnittlich monatliche Datenverkehr pro Smartphone um 43% auf 929 MB gestiegen [CISCO15]. Diese Entwicklung lässt darauf

schließen, dass (Web-)Applikationen im zunehmenden Maße datenzentriert agieren. Auch die Verteilung und Bereitstellung dieser Daten auf unterschiedliche Geräte, hat im Bereich der (Web-)Applikationen eine hohe Priorität. In den letzten Jahren hat daher das Paradigma „Backend-as-a-Service“ zunehmend an Bedeutung gewonnen. Hierbei werden Anwendungslogik und Datenhaltung als Cloud-Service bereitgestellt. Backend-as-a-Service fungiert demnach als Middleware, welche (Web-)Applikationen via APIs mit dem Cloud-Service verbindet [Tech15]. Das Internet spielt daher unter Anderem für die Bereitstellung von Daten eine zentrale Rolle. Hierbei wird allerdings besonders in dem Bereich der mobilen Nutzung eine Problematik deutlich. Denn um auf die Daten im Netzwerk zugreifen zu können, muss das Gerät mit eben diesem verbunden sein. Dies kann bei mobilen Endgeräten, im Gegensatz zu stationären Desktop-Computern, nicht immer gewährleistet werden. Die Notwendigkeit einer vorhandenen Netzwerkverbindung, stellt daher einen Flaschenhals für die Benutzbarkeit von (Web-)Applikationen dar.

Auch die Qualität und die Stabilität von vorhandenen Netzwerkverbindungen variiert bei mobilen Geräten stark und beeinflusst somit aktiv die Benutzbarkeit von (Web-)Applikationen. Im Jahr 2015 besaßen global betrachtet 52% aller mobilen Endgeräte lediglich 2G als Verbindungsgeschwindigkeit. Dies entspricht einer durchschnittlichen Übertragungsgeschwindigkeit für Smartphones von 7,5 Mbps [CISCO15]. Eine zusätzliche Restriktion wird durch die limitierten Ressourcen von mobilen Endgeräten bedingt. Die begrenzte Batteriekapazität beispielsweise, hemmt die dauerhafte Aktivierung von mobilen Daten und somit auch die kontinuierliche Verfügbarkeit einer Netzwerkverbindung. Aufgrund der aufgezeigten Gegebenheiten, ist es im mobilen Bereich erstrebenswert, dass (Web-)Applikationen auch ohne vorhandene Netzwerkverbindung, sprich offline, benutzbar sind. Dieses Paradigma wird als „Offline-First-Applications“ bezeichnet. Hierbei handelt es sich um ein innovatives Thema, welches die klassische Client-Server-Kommunikation erweitert. Eine Offline-First-Applikation ist so konzipiert, dass eine temporäre Netzwerkverbindung ausreicht, um die (Web-)Applikation verwenden zu können. Hierbei steht in der Regel, lediglich ein Teilbereich der Applikation offline zur Verfügung. Offline-Updates werden clientseitig durch einen Storage Provider vorgehalten. Sobald eine Netzwerkverbindung besteht, werden „Client-Daten“ und „Server-Daten“ zusammengeführt und eventuelle Konflikte aufgelöst [Ho13a]. Der Fokus liegt demnach deutlich auf der verbindungsunabhängigen Verfügbarkeit von (Web-)Applikationen.

Das Thema der Masterarbeit befasst sich speziell mit diesem Paradigma. Hierbei soll speziell ein deklaratives Offline-First (DOF) Storage-Konzept für die JavaScript-API von *Orestes* konzipiert und implementiert werden. Dieses Konzept beinhaltet konkrete Lösungen im Bereich lokale Datenanfragen, Aktualisierung von lokalen Zuständen, Synchronisierung und Konfliktauflösung.

## 1.2 Problemstellung

Zur Verdeutlichung der zugrundeliegenden Thematik dieser Masterarbeit, muss zunächst die Mechanik der klassischen Webkommunikation erläutert werden. Generell sind für die Kommunikation zwei Parteien notwendig, der Client und der Server. Hierbei stellt der Server diverse Dienste zur Verfügung, welche wiederum von dem Client genutzt werden können. Der Server übernimmt demnach die Rolle des Diensterbringers, wohingegen der Client als Dienstnutzer fungiert [SS12].

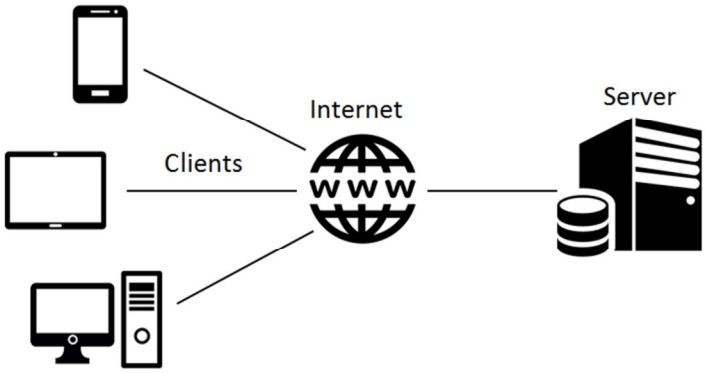



Abb. 1: Client-Server-Architektur

Bevor jedoch eine Kommunikation zwischen Client und Server zustande kommen kann, muss die (Web-)Applikation dem Client zur Verfügung stehen. Dies kann zum einen über die Anfrage von Ressourcen (HTML, CSS, JavaScript etc.) via URL-Aufruf geschehen oder im Falle von nativen Apps über den Download aus einem „App-Store“ [Ho13b]. Hierbei ist zu beachten, dass der Client bei der Kommunikation mit dem Server, die Rolle des Initiators übernimmt. Dies geschieht beispielsweise durch das Senden von neuen Daten zum Server oder durch das Anfragen von neuen Inhalten oder Daten vom Server [Ho13b].

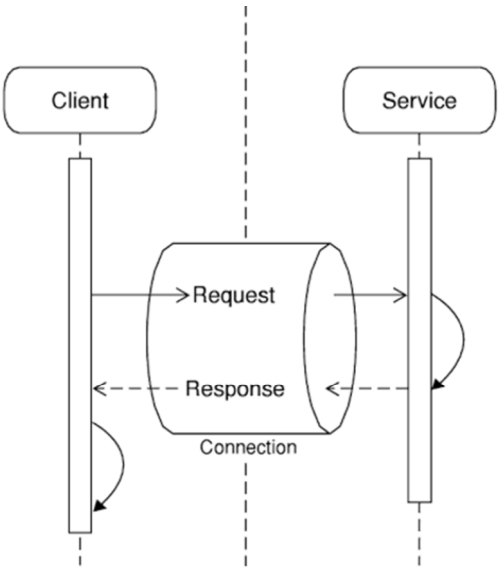



Abb. 2: Request-Response-Pattern [Da12]

Das Request-Response-Pattern ist das meist verbreitetste Pattern für die Kommunikation zwischen Client und Server. Es stellt eine synchrone Kommunikation dar und wird demnach verwendet, wenn der Client eine Antwort ohne größere Verzögerung erwartet. Hierbei sendet der Client eine Anfrage (Request) und wartet solange, bis der Server die entsprechende Antwort (Response) zurücksendet [Da12].

Diese Mechanik der klassischen Webkommunikation zeigt deutlich, dass das Internet eine zentrale Rolle bei der Verteilung der einzelnen Anfragen spielt. Wie die Motivation allerdings bereits verdeutlicht, ist die Vorstellung einer dauerhaften und stabilen Internetverbindung im Bereich von mobilen Endgeräten derzeit und auch in naher Zukunft eine Utopie. Daher besteht das Problem darin, die Benutzung von (Web-)Applikationen auch dann zu ermöglichen, wenn eine Verbindung zum Internet nicht vollständig gewährleistet werden kann. Es muss eine Funktionalität entwickelt werden, bei welcher das Fehlen einer Netzwerkverbindung nicht zu einem Absturz führt. Vielmehr sollte die (Web-)Applikation einen minimalen Grad an Benutzbarkeit niemals unterschreiten [Fe13]. Um dieses Problem zu bewältigen, ist es notwendig, sowohl Ressourcen der (Web-)Applikation (HTML, CSS, JavaScript etc.), als auch die Datenobjekte lokal zu speichern [La12]. Der Client muss demnach befähigt werden, die (Web-)Applikation autark, sprich ohne die Kommunikation mit dem Server, zu betreiben. Dieses Konzept stellt besonders für datengetriebene (Web-)Applikationen eine weitere Herausforderung dar. Denn durch das lokale Zwischenspeichern von Daten, werden Inkonsistenzen der verschiedenen Datenstände gefördert. Dies gilt im Besonderen, wenn die Datenbasis durch mehrere Benutzer veränderbar ist. Daher ist die Konzipierung eines geeigneten Storage-Konzeptes zur Erkennung von Datenkonflikt, ein großer Bestandteil eines Offline-First-Konzeptes [Fe13].

## 1.3 Zielsetzung

Die Masterarbeit trägt den Titel „Declarative Offline-First Application Development under the Backend-as-a-Service Paradigm“ und hat die Konzeptionierung und Implementierung eines deklarativen Offline-First (DOF) Storage-Konzeptes zum Ziel. Dieses Konzept soll für die JavaScript-API von *Orestes* entwickelt werden.

Nach erfolgreicher Implementierung des Artefaktes, soll dieses anhand ausgewählter Anwendungsfälle (*use cases*) evaluiert werden. Hierbei steht der Vergleich mit den zuvor definierten funktionalen Anforderungen im Vordergrund.

## 1.4 Methodik und Vorgehen

Nach Österle et. al. entsteht ein Artefakt der gestaltungsorientierten Wirtschaftsinformatik in den Phasen Analyse, Entwurf und Evaluation [Ös10]. Hierbei wird in der Analysephase das Phänomen erfasst und dokumentiert. Die Entwurfsphase nutzt die dokumentierte

Analyse, um Konzepte zur Problemlösung zu entwickeln und diese zu implementieren. Zur Überprüfung des entwickelten Artefaktes dient abschließend die Evaluationsphase. In der vorliegenden Masterarbeit dient die Analyse-Phase zum einen für die Beschreibung der JavaScript-API von *Orestes*. Zum anderen werden Kriterien für den Einsatz eines geeigneten clientseitigen Storage Providers formuliert. Diese Kriterien werden anschließend für die Auswahl eines geeigneten Storage Providers (*Web Storage*, *IndexedDB*, *WebSQL*) überprüft. Die Entwurfsphase unterteilt sich in die Phasen Konzeptionierung und Implementierung. Während der Konzeptionierungsphase werden Lösungen für folgende Bereiche entwickelt:

- **Lokale Anfrageverarbeitung**: Persistente Speicherung von Datenobjekte in einem lokalen Storage Provider (Online-Modus). Gezieltes Anfragen dieser Datenobjekte durch MongoDB-Anfragen (Offline-Modus).

- **Offline Updates**: Aktualisierung des lokalen Zustands im Falle einer fehlenden Netzwerkverbindung. Die Änderungen müssen hierbei für eine spätere Zusammenführung vermerkt werden.

- **Synchronisierung**: Möglichkeit der Zusammenführung des lokalen und remote Datenbestandes. Hierbei sollen alle CRUD Operationen berücksichtigt werden.

- **Konfliktauflösung**: Handler zur Konfliktauflösung bei nebenläufigen, in Konflikt stehenden Änderungen. Zur Konfliktauflösung sollen mehrere Strategien angeboten werden.

Während der Implementierungsphase werden anschließend die konzipierten Lösungen umgesetzt. Diese Umsetzung wird zum Abschluss der Ausarbeitung, anhand von ausgewählten Anwendungsfällen evaluiert.

## 1.5 Related Works

Bei der Auseinandersetzung mit dem Offline-First-Ansatz, fungiert das Thema Caching als fundamentaler Einstiegspunkt. In der Informatik bezeichnet Caching das Speichern von Daten in einem so genannten *Cache*. Hierbei stellt der *Cache* einen sehr schnellen und verhältnismäßig kleinen Speicher dar, welcher als Puffer fungiert. In diesem werden Daten temporär zwischengespeichert, um die Zugriffszeiten auf Daten drastisch zu reduzieren. Diese Performancesteigerung ist insbesondere bei häufig angefragten Datenstrukturen zu beobachten [Dr07]. Im Web-Bereich wird dieses Konzept als *Web Caching* bzw. *HTTP Caching* bezeichnet. Hierbei werden Webressourcen wie HTML-Dateien, JavaScript oder Bilder zwischengespeichert, um die Latenz beim Laden der Website zu verringern. Die Webressourcen können dabei, wie beispielwiese beim *Browser Cache*, lokal auf dem

Endgerät des Benutzers zwischengespeichert werden, oder wie im Fall eines *Transparent-Proxy-Cache*, auf einem Server im Netzwerk [SP17]. Der entscheidende Nachteil dieser Caching-Technologie ist, dass zwischengespeicherte Ressourcen nur in Verbindung mit einer vorhandenen Netzwerkverbindung aufgerufen werden können. Durch die Entwicklung von *HTML5*, wird der Offline-First-Ansatz zum ersten Mal ernsthaft berücksichtigt. Der Application Chache (*AppCache*) ermöglicht es im Gegensatz zum *Web Cache*, Webseiten bzw. (Web-)Applikation auch ohne Netzwerkverbindung, sprich offline, zu verwenden. Dabei besitzt der *AppCache* jegliche Performancevorteile des *Web Caching* und bietet zusätzlich die Möglichkeit, das gewünschte Caching-Verhalten mittels einer Manifest-Datei zu spezifizieren [RH17].

In der Literatur existieren allerdings auch zahlreiche kritische Stimmen bezüglich des *AppCache*. [Ar12] beschreibt beispielsweise, dass der *AppCache* die zwischengespeicherten Informationen nicht nur im Falle einer fehlenden Netzwerkverbindung verwendet („fallback“), sondern vielmehr immer zunächst den gecachten Inhalt lädt und erst anschließend prüft, ob aktuellere Inhalte auf dem Server vorhanden sind. Sollte dies der Fall sein, ist ein neu laden der Webseite seitens des Nutzers erforderlich. Des Weiteren müssen Ressourcen explizit im Manifest angegeben werden, ansonsten werden diese nicht zwischengespeichert und somit auch nicht angezeigt. Neben diesen, werden noch einige andere Designschwächen des *AppCache* beschrieben. Aufgrund der fragwürdigen Reife des *AppCache*, gilt seit kurzem das Konzept des *Service Worker* als ernsthafte Alternative. Ein *Service Worker* ist ein auf JavaScript basierender, programmierbarer Netzwerk-Proxy. Dieser horcht auf die Anfragen (Request) und befähigt daher zur vollen Kontrolle über die Anfrageverarbeitung [Ga17]. Dies ermöglicht ein situationsspezifisches Antwortverhalten. Beispielsweise kann auf diesem Wege eine instabile Netzwerkverbindung erkannt werden und entsprechend mit Daten aus dem *Cache* reagiert werden. Die Verarbeitung erfolgt hierbei asynchron. Die großen Vorteile des *Service Worker* liegen daher in erster Linie in dessen Flexibilität bezüglich der Anfrageverarbeitung und der Unterstützung der Offline-Funktionalitäten. Ein Beispiel aus der Praxis, welches die Vorteile des *Service Worker* nutzt, sind die so genannten Progressive Web Apps (PWA). Eine PWA ist eine (Web-)Applikation, welche im mobilen Bereich ähnliche Charakteristika besitzt wie native Apps. So kann die (Web-)Applikation auf dem mobilen Endgerät des Nutzers installiert werden und simuliert dadurch eine native App. Durch den Einsatz von *Service Worker* und Caching, ist die PWA optimal für den Offline-Betrieb geeignet [Fi16].

Der innovative Charakter des Themas „Offline First“ zeigt sich vor Allem bei der Suche nach passenden Frameworks. Zwar existieren einige Einzellösungen in Bezug auf „Offline First“ wie *PouchDB* [PDB17] und *localForage* [LF17] aber eine „All-in-One-Lösung“ ist kaum zu finden. Eine dieser wenigen Lösungen ist das Hoodie-Framework [Ho17]. Dieses Framework bietet eine Frontend API, mit welcher die eigene App kommuniziert, um

beispielsweise Daten lokal zu speichern. Zur Kommunikation mit der Server-Datenbank, wird eine node.js Backend-Logik verwendet. Als lokalen Datenspeicher nutzt *Hoodie* die open-source Datenbank *PouchDB*. Zum Synchronisieren der lokalen Daten mit der Server-Datenbank, wird *CouchDB* [CDB17] eingesetzt. Neben diesem Framework existiert zurzeit kein anderes, öffentlich zugängliches Offline-First-Framework.

# 2 Technische Grundlagen und Konzepte

Dieses zweite Kapitel thematisiert die technischen Grundlagen und Konzepte, welche für das Verständnis dieser Masterarbeit notwendig sind. Zu Beginn wird das Paradigma Backend-as-a-Service behandelt. Anschließend wird das Prinzip von Offline-First-Applikationen beschrieben. Daran anlehnend findet eine Vertiefung dieses Prinzips durch die Technologien *Web-Caching* und *Service Worker* statt.

## 2.1 Backend-as-a-Service

Unter Backend-as-a-Service versteht man eine in der Cloud gehostete Backend-Infrastruktur, welche es Entwickler in den Bereichen App- und Mobile-Web-Entwicklung ermöglicht, ein individuelles Backend zu konfigurieren [HD14]. Hierbei wird unter anderem das Persistieren der Daten und die Skalierung von Anfragen direkt vom jeweiligen Anbieter übernommen. Der eigene Entwicklungsprozess wird durch Features, wie das Schema-Management oder die Integration für soziale Netzwerke deutlich verschlankt.

Zwar ist Backend-as-a-Service neben den bereits bekannteren Cloud-Diensten Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) und Software-as-a-Service (SaaS), eine noch recht neue Form in diesem Bereich, jedoch zeigt sich bereits jetzt dessen Potential. Das Unternehmen marketsandmarkets hat 2012 das globale Marktvolumen von Backend-as-a-Service mit 216 Millionen Dollar beziffert. Laut dieser Studie wird dieses globale Marktvolumen bis zum Jahr 2020 auf 28 Milliarden Dollar ansteigen [MAM16]. Typische Anbieter im Bereich Backend-as-a-Service sind *Kinvey* [Ki17], *apiOmat* [Ap17] und *Firebase* [Fi17]. Im Rahmen dieser Masterarbeit wird mit dem Anbieter *Baqend* [Ba17] kooperiert. Dieser unterscheidet sich im Vergleich zu den vorherig genannten Anbietern, vor Allem durch Performancevorteile im Bereich der Ladezeiten.

## 2.2 Das Offline-First-Prinzip

Offline-First ist ein Paradigma der Webentwicklung, welches das Ziel verfolgt, die Benutzbarkeit von (Web-)Applikationen unabhängig vom Verbindungsstatus zu gewährleisten. Aus diesem Grund werden (Web-)Applikationen so konzipiert, dass der Offline-Zustand als Standardfall angenommen wird, welcher die Funktionalität nicht beschränkt [Te16]. Hierbei wird unter offline nicht nur das Fehlen einer Netzwerkanbindung verstanden, sondern auch dessen instabiler Charakter berücksichtigt. Denn selbst wenn eine Verbindung besteht, ist diese häufig zu unbeständig, um eine vollkommene User Experience (UX) zu gewährleisten.

Damit eine (Web-)Applikation auch ohne Netzwerkverbindung funktionieren kann, müssen sowohl die Struktur der App (HTML, CSS, JavaScript etc.), als auch die Datenobjekte im Frontend gespeichert werden [La12]. Auf diese Informationen kann anschließend offline zurückgegriffen werden. Hierzu bauen erste Architekturentwürfe ([Ro10], [Al14]) auf die Technologie des *AppCache* auf, welcher sich allerdings aufgrund seiner Designschwächen [Ar12] als unvorteilhaft herausstellt. Aus diesem Grund wird vermehrt der *Service Worker* (siehe Kapitel 2.4) als Standardlösung empfohlen. Mit diesem ist es erstmals möglich, den Request einer Applikation gezielt abzufangen und dessen Response zu speichern. Die folgende Illustration visualisiert eine Beispielarchitektur für die Umsetzung von Offline-First-Funktionalitäten mittels *Service Worker*.

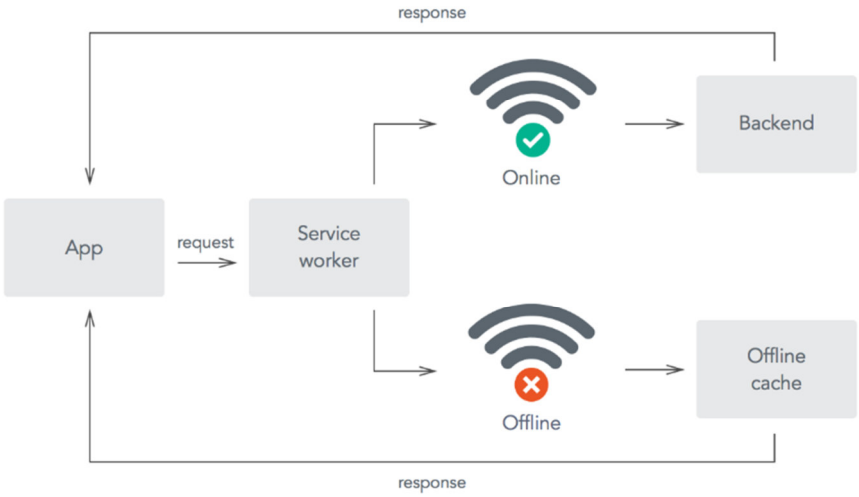



Abb. 3: Beispielarchitektur Offline-First mittels Service Worker [Ch15]

Für das clientseitige Zwischenspeichern der Informationen werden grundsätzlich zwei Mechanismen unterschieden. Zum einen das Web-Caching (siehe Kapitel 2.3) für das Vorhalten von Web-Ressourcen (HTML, CSS, JavaScript etc.) und zum anderen ein lokaler Datenspeicher (siehe Kapitel 3.2) für die persistente Speicherung von Datenobjekten. Auch die Unterscheidung zwischen online und offline stellt sich als nicht triviales Problem dar. Denn Lösungen wie *navigator.onLine* oder *window.addEventListener('online', function())*, kennen lediglich die strikte Trennung zwischen online und offline. Häufig ist jedoch gerade der Anwendungsfall interessant, bei dem zwar eine Netzwerkverbindung besteht, diese jedoch einen dermaßen instabilen Charakter aufweist, dass aus Gründen der Performance, bevorzugt auf vorgehaltene Informationen (beispielsweise aus einem *Cache*) zurückgegriffen werden soll. Abschließend muss ebenfalls auf die Herausforderung der Konsistenzsicherung hingewiesen werden. Denn dadurch, dass die Informationen an physikalisch unterschiedlichen Orten gespeichert sind, ergibt sich das Potential von Inkonsistenzen. Dieses Potential wird noch gesteigert, sobald die Applikation und somit auch die Daten, von unterschiedlichen Nutzern verwendet bzw. bearbeitet werden kann. Ein Mechanismus zur Konfliktauflösung ist hierbei unumgänglich [Fe13].

## 2.3 Web-Caching

Nach [We01] handelt es sich beim *Caching* um einen Mechanismus zum Speichern von kürzlich verwendeten Informationen, um deren Wiederverwendung zu ermöglichen. Zum Speichern wird ein so genannter *Cache* verwendet. Dieser stellt einen sehr schnellen Nebenspeicher dar, welcher Abfragen äußert effizienten beantwortet [OLD17]. Dabei ist es unerheblich, ob eine bestimmte Information erneut angefragt wird oder nicht. Daher sind *Caches* nur dann von Vorteil, wenn der Aufwand für das Speichern von Informationen geringer ist, als deren erneutes Anfragen bzw. Berechnen.

Im Kontext des Internets wird dieser Mechanismus als *Web-Caching* bezeichnet. Hierbei ist es üblich, mindestens einen *Cache* innerhalb des Netzwerkes zwischen Client und Server einzusetzen. Als Client-Cache fungiert der in vielen Browsern integrierte Browser-Cache. Dieser *Cache* ermöglicht das Zwischenspeichern von Web-Ressourcen wie HTML-Dokumente und JavaScript-Dateien. Limitiert ist diese Art des Caches dadurch, dass nur der lokale Client von den Informationen des Browser-Caches profitiert. Denn die Web-Ressourcen werden nur dann aus dem Browser-Cache geladen, wenn der Client dieselbe Seite mit demselben Browser erneut besucht. Das *Proxy Caching* [Hu99] hingegen wird innerhalb des Netzwerkes zwischen Client und Server eingesetzt und kann daher gespeicherte Web-Ressourcen an beliebig viele Clients ausliefern. Durch den Einsatz von Web-*Caching* wird vor Allem die Latenz drastisch verringert. Hierdurch können erhebliche Performancevorteile generiert werden. Auch im Bereich der Bandbreite hat das Web-*Caching* einen positiven Effekt. Denn dadurch, dass Informationen aus dem *Cache* geladen werden können, verbraucht die Anfrage und deren spezifische Applikation eine geringere Bandbreite. Dies kann positive Nebeneffekte bezüglich der Performance von nebenläufigen Applikationen zur Folge haben. Des Weiteren sinkt durch die verringerte Anzahl an Anfragen die Auslastung auf den Servern.

Sobald der *Cache* eine Request erhält, wird geprüft, ob sich der entsprechende Ressource im *Cache* befindet. Sollte dies nicht der Fall sein, wird von einem *cache miss* gesprochen und der Request wird zum Server durchgereicht. Ein *cache miss* tritt dann auf, wenn die Web-Ressource nie zuvor angefragt wurde, oder diese beispielsweise mit *no-store* (nicht für *Caching* freigegeben) gekennzeichnet ist. Sollte sich die angefragte Web-Ressource in dem *Cache* befinden, könnte es sich um einen *cache hit* handelt. Zuvor muss allerdings validiert werden, ob die gespeicherte Information *fresh* (gültig) oder *stale* (veraltet) ist. Sollte letzteres der Fall sein, wird eine Validierungsanfrage an den Server gesendet, um einen gültigen Response zu erhalten. Ein möglicher Indikator für das Erkennen von veralteten Web-Ressourcen im *Cache*, ist die Cache-Control-Direktive *max-age.* Mittels dieser kann ein Zeitraum in Sekunden angegeben werden, in welchem die Web-Ressource nach deren Auslieferung, als gültig betrachtet werden kann. Neben diesen existieren noch weitere

Cache-Control-Direktiven [CCD17], welche das Cache-Verhalten bei Requests bzw. Responses spezifizieren.

Eine besondere Herausforderung beim Web-Caching ist die Sicherstellung von Konsistenz. Denn dadurch, dass Web-Server nur begrenzt über Mechanismen zur Gültigkeitsprüfung (*freshness*) implementieren, kann es durchaus vorkommen, dass veraltete Informationen an den Client weitergeben werden [We01]. Um dieses Problem zu überwinden, existieren die verschiedensten Validierungsmechanismen [RFC2616].

## 2.4 Service Worker

Der *Service Worker* [W3C15] ist eine einfache JavaScript-Datei, welche die Kommunikation zwischen der Applikation und dem Netzwerk kontrolliert. Hierdurch kann das (Lade-)Verhalten einer Applikation, durch das Abfangen und eventuelle Manipulieren der Netzwerkanfragen, situationsabhängig gesteuert werden. Die dadurch gewonnene Kontrolle ist vor Allem bei der Bereitstellung von Offline-Funktionalitäten von großem Vorteil. Denn durch den Einsatz des *Service Worker*, kann eine fehlende oder instabile Netzwerkverbindung erkannt und bei Bedarf mit Informationen aus einem *Cache* kompensiert werden. Da der *Service Worker* separat von der Applikation ausgeführt wird, ist dieser nicht für den Benutzer sichtbar und kann auf keine DOM-Elemente der Applikation zugreifen. Aus Sicherheitsgründen kann der *Service Worker* nur in Verbindung mit HTTPS eingesetzt werden.

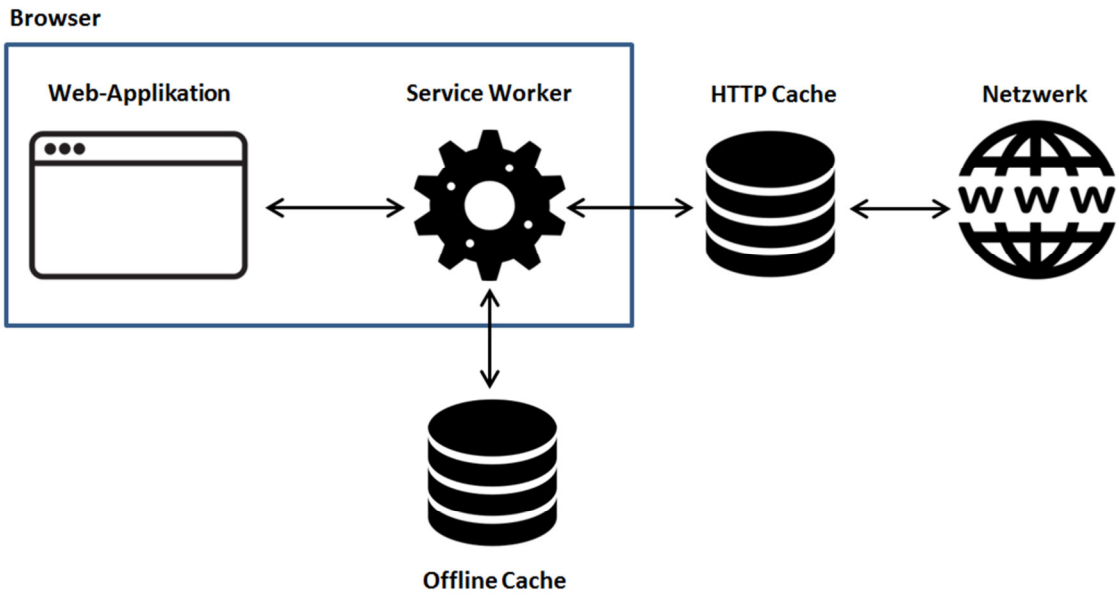



Abb. 4: Anfrageverarbeitung mit Service Worker

Die obige Abbildung visualisiert die Anfrageverarbeitung einer Webapplikation, mit dem Einsatz eines *Service Worker*. Hierbei werden wie bereits angedeutet, die Anfragen der Webapplikation durch den *Service Worker* abgefangen und können anschließend entweder

durch Informationen aus dem Netzwerk oder durch zuvor zwischengespeicherte Ressourcen beantwortet werden. Der Umfang der vorgehaltenen Informationen kann hierbei selbstständig bestimmt werden. Hierzu verwendet der *Service Worker* die *Cache API* [CA17], welche einen Speichermechanismus für Request/Response Objekte bereitstellt. Um die Funktionalitäten des *Service Worker* nutzten zu können, muss dieser zuvor innerhalb der Webapplikation registriert werden. Dies kann durch den folgenden Programmcode umgesetzt werden:

```
navigator.serviceWorker.register('sw.js').then(function(reg){
      // …
}).catch(function(err){
      // …
});
```

Innerhalb der Registrierung kann ebenfalls der Skopus festgelegt werden, welcher durch den *Service Worker* kontrolliert werden soll. Als Standard gilt der Pfad, in welcher der *Service Worker* abgelegt ist. Sollte der *Service Worker* sich demnach im Pfad */foo/sw.js* befinden, wird jede Seite innerhalb des Verzeichnisses */foo/* durch den *Service Worker* kontrolliert. Um einen angemessenen Grad der Kontrolle zu erreichen, ist der *Service Worker* in der Lage, bestimmte Ereignisse zu erkennen und entsprechend dem individuell programmierten Verhalten, auf das Ereignis zu reagieren. Hierfür stehen die verschiedensten *EventListener* innerhalb der *Service Worker API* zur Verfügung. Einer dieser *EventListener* kann dem folgenden Code-Beispiel entnommen werden.

```
self.addEventListener('fetch', function(event){
      // …
});
```

Dieses Fetch-Ereignis, wird durch jede Anfrage einer vom *Service Worker* kontrollierten Seite ausgelöst. Dadurch ist es möglich, das Antwortverhalten für bestimmte Anfragen situationsspezifisch zu manipulieren. Neben dieser Möglichkeit zur Anfragemanipulation, spielt auch der Lebenszyklus des *Service Worker* eine große Rolle. Denn dieser ist komplett unabhängig vom Lebenszyklus der Webapplikation. Grundsätzlich können Seiten einer Webapplikation nur dann durch den *Service Worker* kontrollieren werden, wenn diese geladen werden, nachdem der *Service Worker* registriert (*active*) ist. Das bedeutet, dass Anfragen von bereits geladenen Seiten, nicht von einem kürzlich registrierten *Service Worker* überwacht werden können. Die Seite muss zwangsläufig neu geladen werden, um unter die Kontrolle des *Service Worker* zu gelangen. Ein wenig anders verhält es sich, wenn die Logik des *Service Worker* während der Laufzeit der Webapplikation verändert wird. Hierbei reicht ein simples Aktualisieren (neu laden) der Seite nicht aus, um die neue Funktionalität des *Service Worker* zu aktivieren. Zwar erkennt der Browser das

Vorhandensein einer neuen Version des *Service Worker*, diese wird allerdings solange vom Status *installing* auf *waiting* gesetzt, bis keine Seite der Webapplikation die vorherige Version nutzt. Die Seite muss also entweder geschlossen werden oder es muss auf eine andere Seite navigiert werden, welche nicht vom *Service Worker* kontrolliert wird. Alternativ kann durch das Ausführen der Methode *skipWaiting()*, die Aktivierung der neuen Version erzwungen werden. Dadurch kann sichergestellt werden, dass nur eine Version des *Service Worker* zur selben Zeit aktiv ist.

Zusammenfassend kann der *Service Worker* als eine recht junge Technologie bezeichnet werden, welche allerdings bereits in diesem frühen Stadium der Entwicklung, grundlegende Mechanismen zur Überwachung und Manipulation von Netzwerkanfragen bereitstellt. Dadurch stellt der *Service Worker* eine ernsthafte Alternative zum durchaus umstrittenen *AppCache* dar.

# 3 Analyse

Das folgende Kapitel spiegelt die Analysephase dieser Arbeit wieder. Da das zu implementierende Artefakt dieser Arbeit, für die JavaScript-API von *Orestes* konzipiert ist, wird zunächst der technische Aufbau von *Orestes* thematisiert. Anschließend findet ein Vergleich von verschiedenen lokalen Speichermechanismen satt. Den Abschluss bildet die Auswahl des für die Implementierung geeigneten Storage Provider.

## 3.1 Orestes

*Orestes* [Or17] ist eine im Fachbereich Informatik der Universität Hamburg entwickelte Middleware-Architektur zur persistenten Datenverarbeitung von objektorientierten Informationen durch ein REST/HTTP Kommunikationsprotokoll. Motiviert ist die Entwicklung der *Orestes* Architektur und der zugehörigen Anwendungslogik durch die beschränkte Verwendbarkeit von Datenbanksystemen in einer Cloud-Umgebung. Vor allem die hohe Netzwerklatenz bei Anfragen von mobilen Clients und die fehlenden Mechanismen für eine horizontale Skalierung bei der Anfragebearbeitung, zählen hierbei zu den größten Problemen [GBR14]. Aber auch der Wunsch nach einer statuslosen und zugleich einheitlichen Kommunikation bei Datenbankzugriffen, begründet die Entwicklung von *Orestes* [Ge14].

Da die thematische Ausrichtung dieser Masterarbeit ausschließlich ein grundlegendes Verständnis von *Orestes* voraussetzt, wird dessen Funktionalität im Folgenden lediglich rudimentär erläutert. Neben den hier beschriebenen Mechanismen, werden andere von *Orestes* beinhalteten Erweiterungen wie beispielsweise *Polyglot Persistence* [SGR15] oder *Cache Sketch* [Ge15] nicht thematisiert. Eine vollständige Dokumentation, sowie zahlreiche Publikationen können unter [Or17] und [OV17] eingesehen werden.

### 3.1.1 Architektur

Eine vereinfachte Darstellung der *Orestes* Architektur kann der Abbildung 5 entnommen werden. Clientseitig bilden Persistenz-APIs wie *Java Data Objects* [JDO17] oder *Java Persistence API* [JPA17], ihre Operationen auf die REST/HTTP API von *Orestes* ab. Serverseitig werden REST-Aufrufe durch eine beliebige Anzahl an Orestes-Servern, auf das skalierbare Datenbanksystem abgebildet. Hierbei kann jedes beliebige Datenbanksystem, welches CRUD-Operation unterstützt, mit der *Orestes* Middleware verknüpft werden [GBR14]. Im Netzwerk werden durch Caching-Mechanismen und Lastenverteilung, eine geringe Latenz und ein hohes Maß an horizontaler Skalierung bei der Anfrageverarbeitung gewährleistet. Für eine ausführliche Darstellung der Architektur von *Orestes*, steht die Anlage 1 zur Einsicht zur Verfügung.

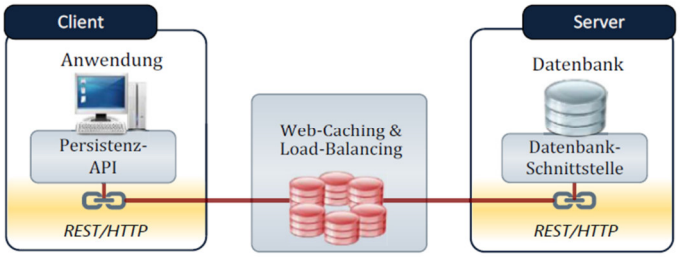



Abb. 5: Vereinfachte Architektur von Orestes [GB13]

### 3.1.2 REST API

In der *Orestes* REST/HTTP API werden verfügbare Ressourcen wie *transaction*, *query* oder *schema* eindeutig durch URLs identifiziert. Mittels der HTTP-Methoden *GET* (abfragen), *PUT* (anlegen/ersetzten), *POST* (aktualisieren) und *DELETE* (löschen), kann mit diesen Ressourcen interagiert werden. Ein mögliches Beispiel einer solchen clientseitigen Interaktion, ist das Abfragen eines bestimmten Nutzerprofils. Der Aufruf wird durch die Persistenz-API ausgeführt: *profiles.find(id)*. Dieser wird dann in einen einheitlichen REST-Aufruf umgewandelt: *GET db/profiles/id*. Einen Ausschnitt einer solchen Ressourcenstruktur, zeigt die folgende Abbildung. Standardmäßig werden Ressourcen in *Orestes* als JSON-Objekte repräsentiert und müssen aus Gründen der Konsistenzsicherung, Versionsnummern beinhalten.

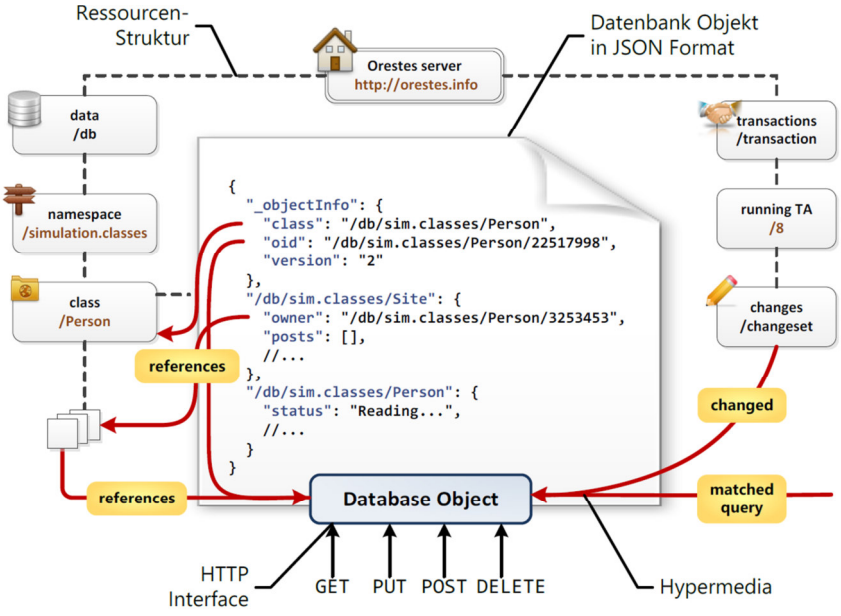



Abb. 6: Ausschnitt einer Ressourcenstruktur der Orestes REST/HTTP API [GBR14]

### 3.1.3 Schema Management

Um die Nutzbarkeit der Datenbank für global verteilte Nutzer zu gewährleiten, unterstützt die *Orestes* Middleware ein serverseitiges Schema-Management. Ein wesentlicher Vorteil von expliziten Schemas ist die Vermeidung von Datenverfälschungen durch die

Möglichkeit der Typenprüfung. In *Orestes* bildet ein Schema Feldnamen auf Typen ab. Diese Typen können entweder *primitive* (String, Boolean, Integer, Float, Date, GeoPoint), eine *reference* (object id), eine *collection* (Set, Map, List), ein *embedded object* (definiert durch ein Schema) oder ein *JSON* (Array, Object) sein [Ge14]. Hierbei hat jeder Orestes-Server das aktuelle Schema in seinem Speicher und eventuelle Änderungen des Nutzers, werden automatisch an alle anderen Orestes-Server kommuniziert. Der Zugang zu einem spezifischen Schema, kann durch *Access Control Lists* (ACL) auf einen bestimmten Nutzer oder einer assoziierten Rolle beschränkt werden.

### 3.1.4 Caching und Skalierung

Eines der Ziele von *Orestes* ist die Verringerung der Netzwerklatenz bei Datenbankzugriffen. Um dies zu realisieren, nutzt *Orestes* die vorhandene Caching-Infrastruktur im Netzwerk, um bereits vorhandene Ressourcen zwischen zu speichern. Denn in *Orestes* werden alle ausgelieferten Ressourcen für eine bestimmte Zeitspanne, die so genannte *TTL* (time to live), als cachebar gekennzeichnet. Für diese Zeitspanne gelten diese Objektkopien als gültig und werden nach Ablauf der Zeit verworfen. Dadurch besteht die Möglichkeit, die Ressourcen näher am Client zu positionieren. Bei einer erneuten Anfrage können die benötigen Ressourcen somit direkt aus dem Zwischenspeicher geladen werden, ohne die Notwendigkeit einer Kommunikation mit den Datenbankserver. Basierend auf ihrer Lage im Netzwerk, unterscheidet *Orestes* zwischen sechs Typen von Caches. Der *Client Cache* kann direkt in die Applikation eingebettet werden. Der *Server Cache*, der *Reverse Proxy Cache* und der *CDN Cache* werden direkt von den Orestes-Servern kontrolliert. Neben diesen wird außerdem der *Forward Proxy Cache* und der *Web Proxy Cache* verwendet [GBR14].

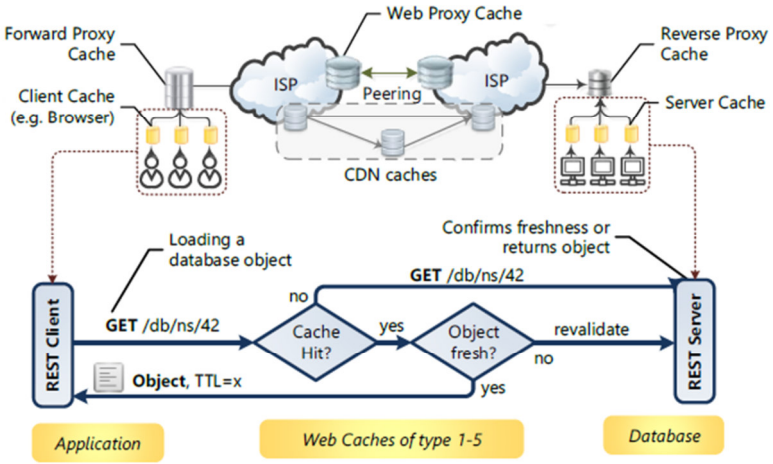



Abb. 7: Typen von Web-Caches und deren Verhalten in Orestes [GBR14]

Abbildung 7 visualisiert neben den sechs Typen von *Cache*, auch den Prozess eines *Cache* bei der Verarbeitung einer Objekt-Anfrage. Hierbei fragt der Client mittels einer REST-Anfrage, über eine eindeutige URL ein Objekt an. Sollte der *Cache* das angefragte Objekt

nicht beinhalten, wird die Anfrage direkt an die Datenbank weitergeleitet. Falls sich das Objekt allerdings im *Cache* befindet, wird die gecachte Kopie des Objektes, auf dessen Gültigkeit (Konsistenz) geprüft. Sollte die Kopie gültig sein, wird das Objekt direkt und ohne Kommunikation mit einem Orestes-Server an den Nutzer ausgeliefert. Im Falle, dass die Kopie nicht mehr gültig ist, fragt der *Cache* eine Revalidierung beim Orestes-Server an. Dieser sendet anschließend das konsistente Objekt mit der aktuellen Versionsnummer zurück. Dieses Verfahren ermöglicht die drastische Verkürzung von Latenzzeiten.

Die horizontale Skalierung von Datenbankanfragen wird durch den Einsatz von *Load Balancing* [Bo01] erreicht. Die Anfrage wird hierbei zur Verarbeitung auf eine beliebige Anzahl an Servern verteilt und erhöht dadurch die Bearbeitungsgeschwindigkeit. Voraussetzung hierfür ist die Statuslosigkeit des HTTP–Protokolls. Denn nur so können Anfragen verteilt werden, ohne dass etwaige Zustände zwischengespeichert werden müssen.

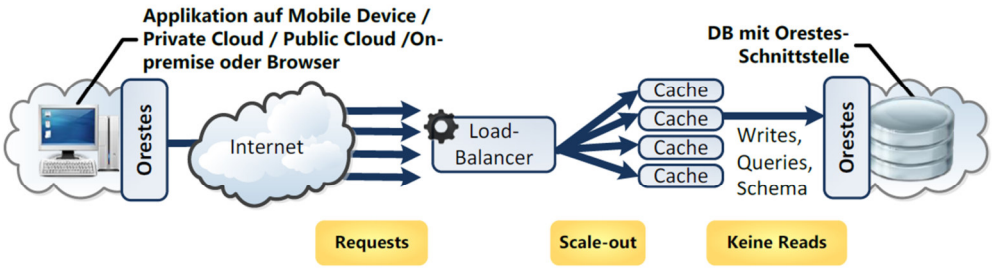



Abb. 8: Horizontale Skalierung bei Orestes [Or17]

### 3.1.5 Cache Konsistenz

Ein wichtiger Aspekt bei der Verarbeitung von Objekten aus einem *Cache*, ist das Zusichern von deren Konsistenz. Dies ist besonders bei verteilten Caching-Infrastrukturen eine große Herausforderung. Zu diesem Zweck wird für die *Orestes* Middleware ein auf dem Konzept des Bloom Filters basierender Mechanismus eingesetzt. Diese *bloomfilterbasierte Cache Kohärenz* [BF17] ermöglicht es die Verantwortung der *Cache-Invalidation* vom Server zum Client zu verlagern [G214]. Ein Bloom Filter [Bl70] ist eine probabilistische Datenstruktur, mit deren Hilfe effizient festgestellt werden kann, ob ein Objekt in einem bestimmten Set vorkommt. Hierbei ist zu beachten, dass der Bloom Filter aus einem Bit-Array besteht und lediglich die Repräsentation von Objekten als Bit enthält. Im speziellen Fall von *Orestes* repräsentiert dieses Set alle Objekte, welche sich in einer bestimmten Zeitspanne verändert haben. Dadurch, dass ein solcher Bloom Filter innerhalb der *Orestes* Middleware verteilt ist, kann der Client diesen vor jedem Transaktionsbeginn abrufen. Sobald der Client eine Anfrage zum Laden eines Objektes stellt, kann anhand des Bloom Filters prüft werden, ob sich die Repräsentation des gewünschten Objektes in dem globalen Set befindet. Sollte dies nicht der Fall sein, kann das Objekt bedenkenlos aus dem *Cache* geladen werden, da es

mindestens so aktuell ist wie der Bloom Filter selbst. Sollte sich die Repräsentation jedoch in dem Set befinden, ist das ein Indiz dafür, dass sich das Objekt verändert hat und eine Revalidierung durch den Orestes-Server stattfinden muss. Anschließend wird das aktualisierte Objekt wieder in den *Cache* geladen [GBR14]. Die nachfolgende Abbildung veranschaulicht diesen Prozess.

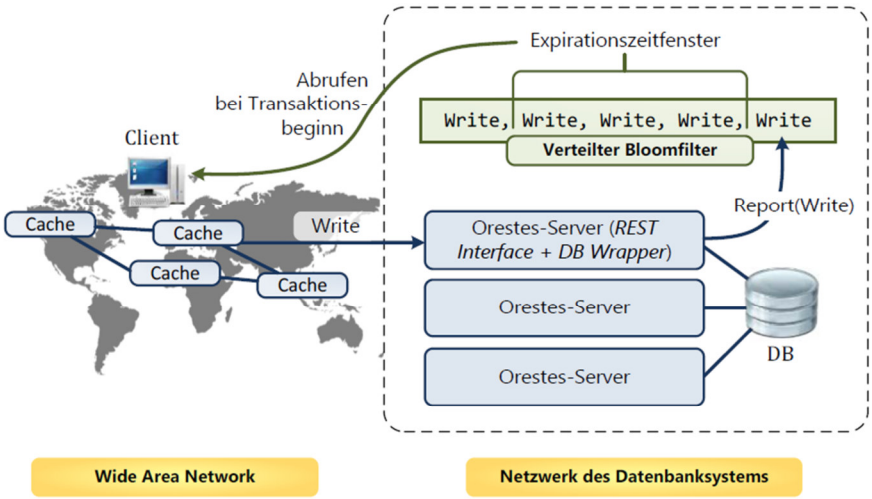



Abb. 9: Bloomfilterbasierte Cache Kohärenz in Orestes [GB13]

## 3.2 Storage Provider

Im Folgenden findet die Auswahl eines geeigneten Storage Providers für die angestrebte Offline-First-Applikation statt. Die Methodik für die Auswahl ist angelehnt an den Softwareauswahlprozess von [Gr01]. Hierfür werden in einem ersten Schritt, Kriterien für einen qualifizierten Storage Provider aufgestellt und gewichtet. Anschließend werden drei potenzielle Technologien zum clientseitigen Speichern von Daten kurz vorgestellt. Den Schluss bilden der Vergleich der einzelnen Technologien und schließlich deren Auswahl. Hierbei wird jede Technologie hinsichtlich der formulierten Kriterien geprüft und bewertet. Die Entscheidung zur Auswahl wird demnach entsprechend der höchsten Bewertung getroffen.

### 3.2.1 Bewertungskriterien

Für die ausstehende Auswahl eines geeigneten Storage Providers zur clientseitigen Speicherung von Daten, werden hier die zugrundeliegenden Entscheidungskriterien formuliert. Formulierungsgrundlage sind die in Kapitel 2 beschriebenen technischen Grundlagen. Nachfolgend werden die sieben Bewertungskriterien aufgelistet.

1. *Kompatibilität mit Service Worker*: Der Storage Provider muss mit der Technologie der *Service Worker* kombinierbar sein.

2. *Strukturierter Datenspeicher*: Durch eine strukturierte Datenspeicherung, sollen gezielte Suchanfragen an den Storage Provider gestellt werden können.
3. *Transaktionsmanagement*: Durch das Verwenden von Transaktionen, sollen Konflikte bei simultanen Datenanfragen behandelt werden können.
4. *Große Datenmengen*: Um Limitationen bei der Skalierung von Datenmengen zu vermeiden, muss der Storage Provider große Datenmengen verarbeiten können.
5. *Komplexe Queries*: Es sollte möglich sein, komplexe Queries (Filter, Sortierung etc.) an den Storage Provider zu stellen.
6. *Asynchrone Kommunikation*: Zur Vermeidung von blockierenden Prozessen bei der Datenverarbeitung, muss die Kommunikation asynchron stattfinden.
7. *Aktive Weiterentwicklung*: Die Technologie des Storage Providers muss durch eine breite Community gepflegt und eingesetzt werden.

Um die individuelle Bedeutsamkeit der formulierten Auswahlkriterien zu visualisieren, wird eine Gewichtung dieser durchgeführt. Hierfür kommt die Methode der Bewertungsmatrix aus der DIN SPEC 1041 [DIN10] zum Einsatz. Das Ergebnis dieser Gewichtung kann der folgenden Abbildung entnommen werden.

Punkteverteilung:
2 : 0 Zeilenkriterium wichtiger als Spaltenkriterium

1 : 1 Zeilenkriterium gleich wichtig wie Spaltenkriterium

0 : 2 Zeilenkriterium weniger wichtig wie Spaltenkriterium

G = Gewichtungsfaktor (normiert)

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Σ | G |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Kompatibilität mit SW | Strukturierter Datenspeicher | Transaktionsmanagement | Große Datenmengen | Komplexe Queries | Asynchrone Kommunikation | Aktive Weiterentwicklung | | |
| 1 | Kompatibilität mit SW | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 12 | 0,25 |
| 2 | Strukturierter Datenspeicher | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 4 | 0,08 |
| 3 | Transaktionsmanagement | 0 | 1 | 1 | 2 | 1 | 0 | 2 | 7 | 0,14 |
| 4 | Große Datenmengen | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 3 | 0,06 |
| 5 | Komplexe Queries | 0 | 1 | 1 | 2 | 1 | 0 | 1 | 6 | 0,12 |
| 6 | Asynchrone Kommunikation | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 12 | 0,25 |
| 7 | Aktive Weiterentwicklung | 0 | 2 | 0 | 1 | 0 | 1 | 1 | 5 | 0,10 |
| | Σ | | | | | | | | 49 | 1 |

Abb. 10: Gewichtung der Bewertungskriterien

### 3.2.2 Web Storage

Die *Web Storage API* [W3C16] ist eine seit *HTML5* verfügbare Technologie, welche das clientseitige Speichern von Daten ermöglicht. Ziel der Einführung ist unter anderem die Überwindung der technischen Unzulänglichkeiten von Cookies [Ba11]. Denn mit der *Web Storage API* werden Daten nicht länger bei jedem Server-Request, sondern nur noch bei speziellen Anfragen verwendet. Dies steigert die Sicherheit und die Performance im Vergleich zu Cookies entscheidend. Ein weiterer Vorteil ist die vergrößerte Speicherkapazität von mindestens 5 MB. Im *Web Storage* werden die Daten als Key-Value-Paar abgespeichert. Hierbei ist zu beachten, dass sowohl der Schlüssel-, als auch der entsprechende Wert-Parameter, ausschließlich als Zeichenkette hinterlegt werden kann. Etwaige andere Datenformate müssen demnach vor dem Speichern in das String-Format überführt werden. Abhängig von der intendierten Spanne der Datenpersistenz, kann in der *Web Storage API* zwischen zwei verschiedenen Attributen gewählt werden. Der *sessionStorage* speichert die Daten solange, bis die Sitzung (Session) des Nutzers endet, sprich bis das Fenster im Browser geschlossen wird. Der *localStorage* hingegen, hält die Daten solange vor, bis diese explizit gelöscht werden. Daten im *localStorage* sind demnach im Browser auch fensterübergreifend verfügbar. Beide Attribute besitzen allerdings dieselben Methoden. So können Daten beispielsweise mit der Methode *setItem()* in das Storage-Objekt hinterlegt und mit *getItem()* wieder aufgerufen werden.

### 3.2.3 WebSQL

Die *WebSQL API* [W3C10] ist eine von *HTML5* separierte Spezifikation, welche Methoden zur Manipulation von clientseitigen Datenbanken mittels *SQL* beinhaltet. Hierbei hat jeder Client ein festes Set an assoziierten Datenbanken. Jede Datenbank in diesem Set hat einen Namen und eine gegenwertige Version. Innerhalb der *WebSQL API* besteht lediglich die Möglichkeit, eine dieser Datenbanken zu manipulieren. Ein Löschen der Datenbank ist innerhalb der API nicht möglich. Die Kommunikation findet hierbei asynchron statt. Die API enthält die folgenden drei Kernmethoden:

- *openDatabase*: Diese Methode erzeugt das Datenbank-Objekt. Hierfür wird entweder eine vorhandene Datenbank angesprochen, oder eine neue Datenbank erstellt.
- *transaction*: Dient zur Kontrolle von SQL-Transaktionen. Dadurch kann auf die Ausführung einer SQL-Abfrage situationsabhängig (commit oder roll-back) reagiert werden.
- *executeSQL*: Diese Methode dient zur Ausführung von SQL-Abfragen.

Die große Schwäche der *WebSQL API* liegt in dessen mangelnder Standardisierung. Denn alle bekannten Implementierungen sind mittels *SQLite* [SQL17] umgesetzt und hemmen dadurch das Standardisierungspotenzial.

### 3.2.4 IndexedDB

Bei der *IndexedDB API* [W3C17] handelt es sich um einen clientseitigen Speicher, welcher das strukturierte Ablegen von großen Datenmengen ermöglicht. Ähnlich der *WebSQL API* hat jeder Client ein festes Set an assoziierten Datenbanken. Jede dieser Datenbanken besitzt ebenfalls einen Namen und eine gegenwertige Version. Jedoch werden die Informationen in diesem Fall nicht in relationalen Tabellen gespeichert, sondern innerhalb der Datenbanken in eine beliebige Anzahl an *object stores* abgelegt. Dadurch stellt die *IndexedDB* eher einen Vertreter des NoSQL-Ansatzes [NS17] dar. Ein *object store* besitzt einen eindeutigen Namen und enthält eine Liste von Einträgen, welche die gespeicherten Daten beinhalten. Ein Eintrag in dieser Liste verfügt über einen Schlüssel und einen Wert. Dabei kann das Schlüsselattribut vom Typ *number*, *date*, *string*, *binary* oder *array* sein. Das Werteattribut kann neben primitiven Datentypen, auch *file objects* und sogar *blob objects* abbilden. Die unterschiedlichen Einträge in einem *object store* sind nach dem Schlüsselattribut aufsteigend sortiert und durch die vorgeschriebene Eindeutigkeit des Schlüsselattributes werden Datenduplikationen ausgeschlossen. Ein charakteristisches Merkmal der *IndexedDB* ist die Unterstützung von Indizes. Hierbei referenzieren die jeweiligen Indizes auf verschiedene Attribute von Einträgen in einem *object store*. Die Indizes werden automatisch erzeugt und bei Änderungen an dem referenzierten Attribut ebenfalls angepasst. Dieser Mechanismus ermöglicht eine hohe Performance für Suchanfragen auf spezielle Attribute eines Datenobjektes. Die Kommunikation mit der Datenbank findet immer über *transactions* statt. Dadurch werden Konflikte bei simultanen Anfragen auf identische Datenobjekte unterbunden. Hierzu besitzen Transaktionen immer eine definierte Lebensdauer und einen festen Skopus. Die *IndexedDB* API bietet sowohl die synchrone, als auch die asynchrone Kommunikation an. Jedoch wird zurzeit lediglich die asynchrone Kommunikation von den gängigen Webbrowsern unterstützt.

### 3.2.5 Auswahl der Technologie

Nach der in den vorigen Abschnitten durchgeführten Vorstellung der drei möglichen Technologien, werden diese nachfolgend anhand der gewichteten Entscheidungskriterien bewertet. Hierbei wird untersucht, ob die jeweilige Technologie das formulierte Kriterium erfüllt. Sollte dies der Fall sein, wird die Gewichtung des entsprechenden Kriteriums, der Technologie angerechnet. Für das Nichterfüllen eines Kriteriums wird dementsprechend keine Gewichtung verrechnet. Anschließend werden die aggregierten Gewichtungen zwischen den verschiedenen Technologien verglichen und jenes mit der höchsten Summe ausgewählt. In den folgenden Abbildungen sind die Bewertungen der drei Technologien *Web Storage*, *WebSQL* und *IndexedDB* visualisiert.

| **Web Storage** | Bemerkungen | Punkte |
|---|---|---|
| Kompatibilität mit SW | Keine Kompatibilität mit Service Worker | 0 |
| Strukturierter Datenspeicher | Lediglich Listeneinträge mit Key-Value-Struktur | 0 |
| Transaktionsmanagement | Kein Transaktionsmanagement möglich | 0 |
| Große Datenmengen | Nur geeignet für kleine Datenmengen | 0 |
| Komplexe Queries | Keine Unterstützung für Queries | 0 |
| Asynchrone Kommunikation | Synchrone Kommunikation | 0 |
| Aktive Weiterentwicklung | Eingestuft als Webstandard | 0,10 |
| Σ | | **0,10** |

Tab. 1: Bewertung der Kriterien Web Storage

| **WebSQL** | Bemerkungen | Punkte |
|---|---|---|
| Kompatibilität mit SW | Keine Kompatibilität mit SW | 0 |
| Strukturierter Datenspeicher | Datenspeicherung in relationalen Tabellen | 0,08 |
| Transaktionsmanagement | Transaktionsmanagement vorhanden | 0,14 |
| Große Datenmengen | Keine Begrenzung vorhanden | 0,06 |
| Komplexe Queries | SQL-Queries möglich | 0,12 |
| Asynchrone Kommunikation | Asynchrone Kommunikation | 0,25 |
| Aktive Weiterentwicklung | Weiterentwicklung 2010 eingestellt | 0 |
| Σ | | **0,65** |

Tab. 2: Bewertung der Kriterien WebSQL

| **IndexedDB** | Bemerkungen | Punkte |
|---|---|---|
| Kompatibilität mit SW | API zum Einbinden vorhanden | 0,25 |
| Strukturierter Datenspeicher | Sortierte Datenablage in Listen und Unterstützung von Indizes | 0,08 |
| Transaktionsmanagement | Transaktionsmanagement vorhanden | 0,14 |
| Große Datenmengen | Keine Begrenzung vorhanden | 0,06 |
| Komplexe Queries | Gewünschte Query-Spreche muss zusätzlich bereit gestellt werden | 0 |
| Asynchrone Kommunikation | Synchrone und Asynchrone Kommunikation | 0,25 |
| Aktive Weiterentwicklung | Eingestuft als Webstandard | 0,10 |
| Σ | | **0,88** |

Tab. 3: Bewertung der Kriterien IndexedDB

Nach Betrachtung der einzelnen Kriterien wird deutlich, dass die *Web Storage API* lediglich für kleine Datenmengen geeignet ist. Speziell für Anwendungen, welche synchron kommunizieren und keine Notwendigkeit für einen strukturierten Datenspeicher haben, ist diese API eine geeignete Alternative. Daher kann die *Web Storage API* als eher leichtgewichtige Lösung angesehen werden. Die *WebSQL API* erfüllt technisch nahezu alle formulierten Kriterien, allerdings ist diese Technologie nicht mit dem *Service Worker* kompatibel. Des Weiteren muss beachtet werden, dass diese API aufgrund mangelnden Standardisierungspotenzials, seit 2010 als überholt eingestuft wird [W3C10]. Komfortabler gestaltet sich die Bewertung für die *IndexedDB API*. Diese bietet nicht nur die gewünschten technischen Möglichkeiten, sondern wird zusätzlich von der Technologie der *Service Worker* unterstützt. Lediglich die Unterstützung für eine geeignete Query-Sprache ist nicht in der API enthalten. Dies kann allerdings durch das Einbinden einer solchen Sprache überwunden werden. Für die technische Reife spricht außerdem die Einstufung als Webstandard. Aus diesen Gründen wird die *IndexedDB API* als Storage Provider für die angestrebte technische Umsetzung ausgewählt.

# 4 Konzeptionierung

Im folgenden Kapitel findet der Konzipierung der einzelnen Mechanismen für das zu implementierende Artefakt statt. Hierzu werden anfangs die Anforderungen an die Implementierung festgehalten. Anschließend wird eine Begründung für die Umsetzung innerhalb des *Baqend SDK* abgegeben. Darauf folgend wird die Technologie für das gezielte Anfragen von lokalen Daten und eine mögliche Umsetzung der Offline Erkennung vorgestellt. Den Abschluss dieses Kapitels bildet die Erläuterung der Mechanismen für Offline-Updates, der Synchronisierung und den einzelnen Konfliktauflösungsstrategien.

## 4.1 Anforderungen an die Implementierung

Im folgenden Abschnitt sind die Anforderungen für das zu implementierende Artefakt formuliert. Diese Anforderungen dienen im vorliegenden Kapitel der Konzipierung als Entwurfsgrundlage und werden im späteren Kapitel der Evaluation, einer qualitativen Überprüfung unterzogen. In Anlehnung an [Ba13] sind die nun folgenden Anforderungen nach den Kategorien funktional und qualitativ differenziert.

| **Funktionale Anforderungen** | | |
|---|---|---|
| **Nr.** | **Name** | **Beschreibung** |
| 1 | Kompatibilität (Kapitel 4.2) | Das Artefakt muss mit der von *Baqend* eingesetzten Technologie kompatibel sein. |
| 2 | Lokale Datenspeicherung (Kapitel 4.3) | Datenobjekte sollen in einem lokalen Speichermedium abgespeichert werden können. |
| 3 | Offline Erkennung (Kapitel 4.4) | Das Artefakt muss in der Lage sein, eine fehlende Netzwerkverbindung zu erkennen (Offline-Modus). |
| 4 | Offline Anfragen (Kapitel 4.3) | Lokale Datenobjekte sollen im Online-Modus gesammelt werden und im Offline-Modus mittels MongoDB-Queries abrufbar sein. |
| 5 | Online-/Offline-Updates (Kapitel 4.5) | Lokale Daten müssen sowohl im Online-, als auch in Offline-Modus veränderbar sein. |
| 6 | Offline-Update-Erkennung (Kapitel 4.5) | Die Veränderung von lokalen Daten im Offline-Modus muss identifiziert werden können. |
| 7 | Synchronisierung (Kapitel 4.6) | Es muss die Möglichkeit bestehen, die im Offline-Modus geänderten Datenobjekte, mit den lokalen Daten zusammenzuführen. Des Weiteren muss ein Set definiert werden |

| | | |
|---|---|---|
| | | können, welches jene Abfragen enthält, die im Online-Modus automatisch den lokalen Datenbestand aktualisieren. |
| 8 | Konflikterkennung (Kapitel 4.7) | Konflikte in der Beschaffenheit von lokalen und remote Daten müssen erkannt werden. |
| 9 | Konfliktauflösung (Kapitel 4.7) | Das Artefakt muss über Konfliktauflösungsstrategien verfügen. |

Tab. 4: Funktionale Anforderungen

| **Qualitative Anforderungen** | | |
|---|---|---|
| **Nr.** | **Name** | **Beschreibung** |
| 1 | Konfliktfreie Datenbestände | Lokal durchgeführte Änderungen an Datenobjekten, müssen nach erfolgreicher Synchronisierung konfliktfrei mit den entsprechenden remote Datenobjekten sein. |
| 2 | Reproduzierbarkeit | Das Ausführen eines Queries mit identischen Filter Operatoren führt lokal, im Vergleich zu remote, zum gleichen Ergebnis. |
| 3 | Online Performance | Die Performance im Online-Modus soll durch den Einsatz der Offline Technologie nicht negativ beeinflusst werden. |
| 4 | Nutzerfreundlich | Die Komplexität bei der Benutzung der Baqend Technologie, soll durch die zusätzlichen Offline Funktionalitäten nicht gesteigert werden. |

Tab. 5: Qualitative Anforderungen

## 4.2 Service Worker vs. Baqend SDK

Im Kapitel 2.4 dieser Ausarbeitung wird die Technologie des *Service Worker* eingeführt, dessen Kompatibilität mit Offline-Funktionalitäten unbestritten ist. Gerade in Anwendungsfällen mit statischen Webressourcen wie HTML- oder CSS-Dateien, stellt sich der *Service Worker* als äußert vorteilhaft heraus. In diesem speziellen Fall der Zielsetzung, muss dessen Einsatz allerdings etwas kritischer interpretiert werden. Denn aufgrund der Tatsache, dass in erster Linie das Verarbeiten und Bereitstellen von Datenobjekten, Ziel dieser Ausarbeitung ist, steht die Technologie des *Service Worker* differenzierteren Herausforderungen gegenüber. Anders als bei der Benutzung eines Ressourcen-Caches, bei welchem zumeist ein Abgleich der angefragten Ressource mittels der *Cache API* [CA17] ausreicht, ist die Kommunikation, Synchronisierung und Konfliktauflösung mit einem

lokalen Storage Provider deutlich komplexer. Mit steigender Komplexität, wächst die Größe der *Service Worker* Datei entsprechend an. Dies hat einen negativen Einfluss auf die Performance beim Registrieren des *Service Worker* und beeinträchtigt dadurch negativ die User Experience. Da die angestrebte Implementierung mit der Zielsetzung von *Baqend* [Ba17] übereinstimmen soll und diese ausdrücklich jegliche Hemmung in Bezug auf Performance zu minimieren ersucht, stellt der Einsatz eines *Service Worker* einen Interessenskonflikt dar.

Eine weitere Problematik ergibt sich durch den Skopus des *Service Worker*. Wie bereits in Kapitel 2.4 beschrieben, kann der *Service Worker* nur jene Request erkennen, welche sich in den zuvor definierten Skopus befindet. Als Standard gilt hierbei der Pfad in welchem der *Service Worker* abgelegt ist. Im Hinblick auf die Technologie von *Baqend* funktioniert dies solange unproblematisch, bis sich die Hostdomain der Applikation, von der Datenbankdomain unterscheidet. Denn in diesem Fall werden die Anfragen mittels *IFrame* durchgeführt und können dadurch nicht vom *Service Worker* erfasst werden. Die folgende Abbildung veranschaulicht den Prozess dieser Datenbankkommunikation.

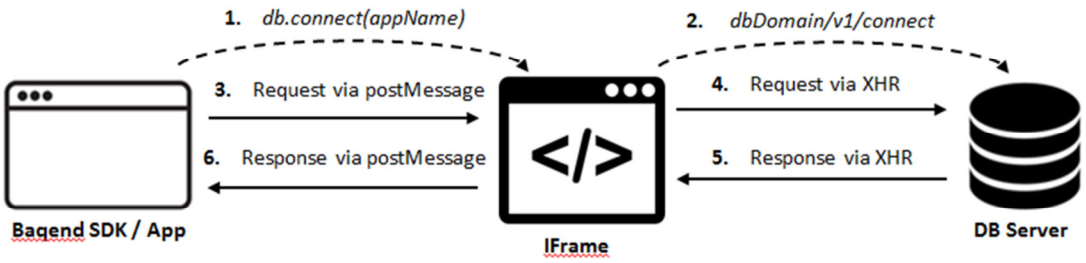



Abb. 11: Datenbankkommunikation via IFrame

Um eine Verbindung mit dem Datenbank-Server herzustellen, muss die Connect-Methode am Datenbank-Objekt aufgerufen werden. Wird hierbei der Name der Applikation als Parameter übergeben, wird automatisch ein *IFrame* erstellt. Dieses verbindet sich anhand des übergebenen Namens mit dem Datenbank-Server und lädt unter dem Verzeichnis *v1* eine *connect* Datei. Diese ist für das Bearbeiten von Anfragen über das *IFrame* verantwortlich. Hierbei wird der als JSON codierte Request mittels *postMessage* an das *IFrame* übergeben. Innerhalb dessen wird die Anfrage decodiert und mittels *XHR* an den Datenbank-Server weitergeleitet. Der entsprechende Response wird anschließen mittels selbiger Übertragungsart, vom Server über das *IFrame* zur Applikation übermittelt. Durch diesen Mechanismus ermöglicht es *Baqend*, App und Datenbank auf verschiedenen Domains zu hosten. Wie bereits beschrieben ist dies nicht mit dem Skopus-Ansatz des *Service Worker* kompatibel.

Aufgrund der aufgezeigten Probleme hinsichtlich der Verarbeitung und Bereitstellung von Datenobjekten und bezüglich der bedenklichen Kompatibilität des *Service Worker* mit

Teilen der von *Baqend* eingesetzten Technologien, wird entschieden, die angestrebte Offline-Funktionalität nicht mittels *Service Worker* zu implementieren. Vielmehr soll die Funktionalität direkt in das *Baqend SDK* integriert werden.

## 4.3 Lokale Anfrageverarbeitung mittels minimongo

Eine zentrale Anforderung an das zu implementierende Artefakt, ist die Möglichkeit der lokalen Anfrageverarbeitung für den Fall, dass keine Verbindung zur Remotedatenbank hergestellt werden kann. Im Kapitel 3.2.5 dieser Arbeit wird hierzu *IndexedDB* als lokaler Datenspeicher ausgewählt. Dieser ermöglicht es allerdings lediglich, Einträge mittels einem zuvor bekannten Schlüssel (*key*) bzw. eines Schlüssel-Bereichs (*key range*) auszulesen. Das Ziel ist es jedoch, gezielte Anfragen (*queries*) mittels Filter Operatoren durchzuführen. Aufgrund der Tatsache, dass *Baqend* die dokumentenorientierte NoSQL-Datenbank *MongoDB* [MDB17] zur Datenverwaltung einsetzt, muss der lokale Datenspeicher die Syntax dieser Technologie ebenfalls interpretieren können. Hierzu soll das Node.js-Modul *minimongo* [MM17] eingesetzt werden, welches einen Teil der *MongoDB API* implementiert und gleichzeitig die Datenverwaltung mittels *IndexedDB* ermöglicht. Das Modul ist eine aus dem Jahr 2014 durchgeführte Abspaltung (*fork*) des gleichnamigen Meteor-Moduls [Me17]. Das Modul kann über den NPM Befehl *npm install minimongo* installiert und mittels *require('minimongo')* eingebunden werden. Anschließend besteht die Möglichkeit, beliebig viele Objektinstanzen der gewünschten JavaScript-Objekte (*IndexedDb*, *LocalStorageDb*, *WebSQLDb*, *MemoryDb*, *RemoteDb* und *HybridDb*) zu erzeugen. Nachfolgend kann ein Code-Beispiel zur Initialisierung eines *IndexedDB*-Objektes entnommen werden.

```
var minimongo = require('minimongo');
var IndexedDB = minimongo.IndexedDb;
db = new IndexedDB({namespace: 'testDB'}, function() {
      // …
});
```

Da *minimongo* die Daten in *collections* [MC17] abspeichert, muss vor jeder Anfrage bzw. Operation die entsprechende *collection* mittels der *addCollection(name)* Methode angesprochen werden. Zum Abfragen von Datensätzen bietet *minimongo* die *find()* Methode. Diese erwartet als Übergabeparameter eine Abfragebedingung (*query selector*) und optional den Limit- bzw. Sortierparameter. Neben dieser Methode existiert auch die *findOne()* Methode. Diese erwartet ausschließlich eine Abfragebedingung (*query selector*) und liefert nur den ersten Datensatz, welcher die Bedingung erfüllt, aus. Die Abfragebedingung kann mittels Filter Operatoren in Form von *{<field1>: {<operator1>: <value1>}, …}* formuliert werden. Die nachfolgende Tabelle enthält alle für *Baqend* relevanten Filter Operatoren, welche von *minimongo* unterstützt werden. Die

Beschreibungen der jeweiligen Operatoren sind aus der Dokumentation von *MongoDB* [MDB17] ins Deutsche übersetzt.

| Name | Beschreibung |
|---|---|
| *$eq* | Findet alle Dokumente, bei denen der angegebene Wert exakt mit dem Wert in dem angegebenen Feld übereinstimmt. |
| *$ne* | Findet alle Dokumente, bei denen der angegebene Wert nicht mit dem Wert in dem angegebenen Feld übereinstimmt. |
| *$gt* | Findet alle Dokumente, bei denen der angegebene Wert größer als der Wert in dem angegebenen Feld ist. |
| *$gte* | Findet alle Dokumente, bei denen der angegebene Wert größer oder gleich dem Wert in dem angegebenen Feld ist. |
| *$lt* | Findet alle Dokumente, bei denen der angegebene Wert kleiner als der Wert in dem angegebenen Feld ist. |
| *$lte* | Findet alle Dokumente, bei denen der angegebene Wert kleiner oder gleich dem Wert in dem angegebenen Feld ist. |
| *$in* | Findet alle Dokumente, bei denen das angegebene Array-Feld einen Wert aus dem angegebenen Array besitzt. |
| *$nin* | Findet alle Dokumente, bei denen das angegebene Array-Feld keinen Wert aus dem angegebenen Array besitzt. |
| *$exists* | Findet alle Dokumente, bei denen das angegebene Feld einen Eintrag besitzt. |
| *$all* | Findet alle Dokumente, bei denen das angegebene Array-Feld, alle Werte aus dem angegebenen Array besitzt. |
| *$mod* | Führt eine Module-Operation aus und findet alle Dokumente, bei denen das angegebene Feld dem Ergebnis dieser Operation entspricht. |
| *$regex* | Findet alle Dokumente, bei denen das angegebene Feld der angegebene Regular Expression entspricht. |
| *$size* | Findet alle Dokumente, bei denen das angegebene Array-Feld der angegebenen Größe entspricht. |
| *$and* | Joint Abfrage-Bedingungen mit einem logischen *and* und findet alle Dokumente, welche alle Bedingungen erfüllen. |
| *$or* | Joint Abfrage-Bedingungen mit einem logischen *or* und findet alle Dokumente, welche mindestens eine Bedingung erfüllen. |
| *$nor* | Joint Abfrage-Bedingungen mit einem logischen *nor* und findet alle Dokumente, welche keine der Bedingungen erfüllen. |

Tab. 6: MongoDB Filter Operatoren in minimongo

Neben den aufgeführten Filter Operatoren unterstützt *minimongo* ebenfalls das von *Baqend* angebotene Tiefenladen von Objekten. Die entsprechende Tiefe kann durch Angabe des *depth* Parameters spezifiziert werden. Lediglich die Angabe eines *offsets* und die *count*

Methode werden von *minimongo* nicht unterstützt. Ziel ist es, den Datenbestand des lokalen Datenspeichers während des Online-Modus zu speisen. Aus diesem Grund werden die Ergebnisse einer Abfrage im Online-Modus nicht nur an den Client ausgeliefert, sondern ebenfalls in die lokale Datenbank eingefügt bzw. überschrieben. Dadurch wird erreicht, dass die lokalen Datenobjekte mindestens genau so aktuell sind, wie deren letzte im Online-Modus durchgeführte Abfrage der Serverversion.

Zum Durchführen von CRUD-Operationen kann zum einen die *upsert()* Methode verwendet werden. Diese führt sowohl das Einfügen (*insert*), als auch das Aktualisieren (*update*) von Datenobjekten durch. Da für eingefügte Datenobjekte intern eine eigene Id erzeugt wird, muss vor dem Aktualisieren eines Datenobjektes, dieses zunächst durch die *find()* bzw. *findOne()* Methode geladen werden. Hierbei kann das gewünschte Objekt mittels der externen Id geladen und modifiziert werden, um dieses anschließend an die *upsert()* Methode zu übergeben. Simultan verhält es sich beim Löschen von Datenobjekten. Die zuständige *remove()* Methode benötigt ebenfalls das geladene interne Datenobjekt. Nach erfolgreichem Löschen eines Datenobjektes wird dieses allerdings nicht physisch aus der internen Datenbank gelöscht, sondern lediglich dessen Status auf gelöscht gesetzt. Datenobjekte mit diesem Status werden bei Anfragen nicht mehr berücksichtigt.

In Kapitel 3.2.1 dieser Ausarbeitung wird thematisiert, dass die von *Baqend* eingesetzte Technologie *Orestes*, Anfragen mit den entsprechenden Abfragebedingung mittels REST-Aufruf durchführt. Ziel soll es sein, Anfragen für den Fall, dass keine Verbindung zur Remotedatenbank hergestellt werden kann, durch lokale Datenobjekte zu kompensieren. Hierfür werden die spezifischen Abfragen an die *find()* Methode von *minimongo* weitergereicht. Das Ergebnis dieser Abfrage wird anschließend an die Applikation ausgeliefert. Dadurch können Objektanfragen auch ohne bestehende Netzwerkverbindung beantwortet werden.

## 4.4 Offline Erkennung

Ein essentieller Bestandteil eines Offline-First Storage Konzeptes, ist das Erkennen der aktuellen Netzwerkverbindung. Demnach muss die Möglichkeit bestehen, zwischen online und offline zu unterscheiden. Wie bereits im Kapitel 2.2 dieser Arbeit beschrieben, liegt eine besondere Herausforderung hierbei in dem Erkennen einer unbefriedigenden Netzwerkanbindung. Sprich der Umstand, in welchem technisch gesehen eine Verbindung besteht, diese allerdings eine dermaßen niedrige Qualität und Beständigkeit aufweist, dass sie eigentlich als nicht vorhanden eingestuft werden muss. Aus Gründen der zeitlichen Restriktion dieser Arbeit, wird darauf verzichtet, eine solch ausgereifte Funktionalität zu implementieren. Das umzusetzende Artefakt verwendet zum Unterscheiden von Online- und Offline-Modus, das am *navigator* registrierte Attribut *onLine*, welches einen booleschen Ausdruck repräsentiert. Der für die Synchronisierung (siehe Kapitel 4.6) entscheidende

Wechsel von Offline- zu Online-Modus, wird mittels den beiden Events *online* und *offline* [OOE17] erkannt. Hierzu soll das Konzept der *observable objects* [RxJS17] genutzt werden. Dadurch kann sich der Nutzer auf den Wechsel zwischen online und offline subskribieren (*subscribe*) und somit auf eben diesen Wechsel horchen. Dies ermöglicht das Ausführen von Methoden, welche speziell für eine der beiden Netzwerkstatus konzipiert sind. Auch wenn eine erweiterte Funktionalität dieser Erkennung nicht Teil des Artefakts dieser Arbeit ist, wird im Folgenden dennoch eine mögliche Umsetzung erläutert.

Eine eher simplere Bereitstellung der Funktionalität zum Erkennen von unbefriedigender Netzwerkverbindung besteht im Einbinden von entsprechenden Bibliotheken wie *Offline.js* [OJS17]. Diese bieten eine breite Spanne an Funktionalitäten, wie beispielsweise das Informieren von Nutzern, sobald eine schlechte Netzanbindung besetzt oder das Erfassen von Ajax-Anfragen während unzureichender Verbindung und deren Wiederholung, sobald die Netzanbindung wieder besteht. Derartige Bibliotheken bieten allerdings oftmals mehr Funktionalität, als das eigene Artefakt benötigt und vergrößern das Projekt daher unnötig. Eine schlankere aber auch aufwändigere Lösung ist das Konzipieren einer eigenen Offline-Methode. Diese könnte beispielsweise einen *XHR-Request* an den entsprechenden Server senden und auf dessen Response warten. Sollte die Antwort länger als die in der Logik festgesetzte Toleranz in Anspruch nehmen, wird eine nicht vorhandene Netzanbindung angenommen und die jeweilige Offline-Funktionalität angestoßen.

Die beiden fortgestellten technischen Umsetzungen sind nur zwei von vielen Möglichkeiten, eine detailliertere Offline Erkennung zu integrieren bzw. zu implementieren. Wie bereits beschrieben wird das Artefakt dieser Arbeit jedoch lediglich die strikte Trennung von online und offline unterstützen.

## 4.5 Offline Updates

Die in diesem Kapitel beschriebenen Mechanismen zur Identifizierung von Offline-Updates werden während der Synchronisierung (siehe Kapitel 4.6) dazu verwendet, um Konflikte zu erkennen und die Konfliktauflösung (siehe Kapitel 4.7) zu unterstützen. Nachfolgend wird hierfür zwischen INSERT- bzw. UPDATE-Operationen und DELETE-Operationen unterschieden.

Zur Gewährleistung einer zufriedenstellenden Offline-First-Erfahrung, ist es erforderlich, dass Datenobjekte auch ohne vorhandene Netzwerkverbindung erstellt, modifiziert und gelöscht werden können. Die Herausforderung liegt hierbei darin, die entsprechenden Datenobjekte so zu markieren, dass diese für das spätere Zusammenführen von unterschiedlichen Objektquellen eindeutig identifiziert werden können. Um dies zu bewältigen, existiert unter anderem das Konzept einer *Dirty Flag* [IBM17]. Hierbei handelt es sich in der Regel um ein boolesches Attribut, welches auf wahr gesetzt wird, sobald der

Nutzer eine Veränderung an dem entsprechenden Datenobjekt vorgenommen hat. Sobald die Änderungen gespeichert sind, wird das boolesche Attribut auf unwahr gesetzt. Dadurch können Änderungen, welche noch nicht persistent gespeichert sind, zuverlässig erkannt und behandelt werden.

Der Nachteil des Konzeptes der *Dirty Flag* besteht in der Ladegeschwindigkeit der gekennzeichneten Datenobjekte. Denn da die Kennzeichnung lediglich an dem Objekt selbst vorgenommen wird und diese wiederrum in verschiedenen *collections* abgespeichert sind, müssen zwangsweise alle *collections* durchgelaufen werden, um alle relevanten Datenobjekte mittels gezielten Abfragen zu identifizieren. Dies kann bei einer steigenden Zahl an *collections*, zu einer negativen Beeinflussung der Ablaufgeschwindigkeit führen. Aus diesem Grund wird bei dem in dieser Arbeit umzusetzenden Artefakt, auf das Konzept der *Dirty Flag* verzichtet. Zur Kennzeichnung von abgeschlossenen INSERT- bzw. UPDATE-Operation, soll daher eine eigenständige *collection* gepflegt werden. Hierzu werden Referenzen auf die angelegten bzw. veränderten Datenobjekte, in einer speziell für diesen Zweck erstellten *collection*, in dem lokalen Datenspeicher vorgehalten. Die Referenz wird vor dem Anlegen/Bearbeiten mittels der Id erstellt und nach Abschluss der Operation in der *Collection* gespeichert. Für das Durchführen von DELETE-Operationen wird dieses Konzept ebenfalls verwendet. Die in diesen beiden *collections* enthaltenden Referenzen können anschließend durch eine gezielte Iteration ausgelesen und behandelt werden. Nach erfolgreicher Synchronisierung der einzelnen Datenobjekte innerhalb dieser *collections*, können die Referenzen aus der jeweiligen *collection* entfernt werden.

## 4.6 Synchronisierung

Nachdem im vorherigen Abschnitt erläutert wird, wie offline Updates innerhalb des lokalen Datenspeichers gekennzeichnet werden, thematisiert dieses Kapitel den Mechanismus zur Zusammenführung von Offline- und Online-Daten. Genauer wird hierzu eine kurze Einleitung zur generellen Funktionsweise gegeben und anschließend der Ablauf der Synchronisierung in einzelnen Schritten erläutert.

Bei der Konzipierung eines Synchronisierungs-Mechanismus stellt sich die Frage, welcher Zustand der jeweiligen Datenbestände nach erfolgreicher Synchronisierung bestehen soll. Laut der in Kapitel 4.1 formulierten Anforderung, müssen lokal durchgeführte Änderungen nach Durchführung der Synchronisierung konfliktfrei mit den entsprechenden Serverdaten sein. Als eine zentrale Entwurfsentscheidung bezüglich des Umfangs an zu synchronisierenden Daten wird festgelegt, dass lediglich jene Datenobjekte, welche offline angelegt bzw. aktualisiert wurden, mit den entsprechenden Versionen auf dem Server zusammengeführt werden sollen. Demnach gilt nach erfolgreicher Synchronisierung, dass alle lokal behandelten Datenobjekte, die Ausprägung der Serverversionen wiederspiegeln. Der Umfang an Datenobjekte bleibt hierbei jedoch

derselbe, denn es werden keine neuen Daten vom Server in den lokalen Datenspeicher übertragen. Für das Zusammenführen von nicht veränderten Datenobjekten und für das Bereitstellen von neuen Serverdaten, ist ein von der Synchronisierung losgelöstes Konzept angedacht. Hierzu wird das im Kapitel 4.4 beschriebene Verhalten der *observable objects* genutzt. Dadurch können Veränderungen an dem neu konstruierten Online-Attribut des Datenbankobjektes wahrgenommen werden. Somit kann ein Set an Abfragen definiert werden. Diese werden im Falle einer vorhandenen Netzwerkverbindung zum Server gesendet und aktualisieren dadurch automatisch die lokalen Datenobjekte. Um lokale Änderungen hierbei nicht zu überschreiben, wird dieser Mechanismus erst nach abgeschlossener Synchronisierung angestoßen. Da eine Trennung von INSERT- bzw. UPDATE-Operationen und DELETE-Operationen stattfindet, existieren zwei getrennte Mechanismen für das Synchronisieren von angelegten/veränderten und gelöschten Datenobjekten. Abbildung 12 visualisiert eine vereinfachte Darstellung dieser beiden Synchronisierungsmechanismen in Form eines Aktivitätendiagramm. Eine detaillierte Darstellung beider Mechanismen können den Anlagen 2 (Offline-INSERT/-UPDATE) und 3 (Offline-DELETE) entnommen werden.

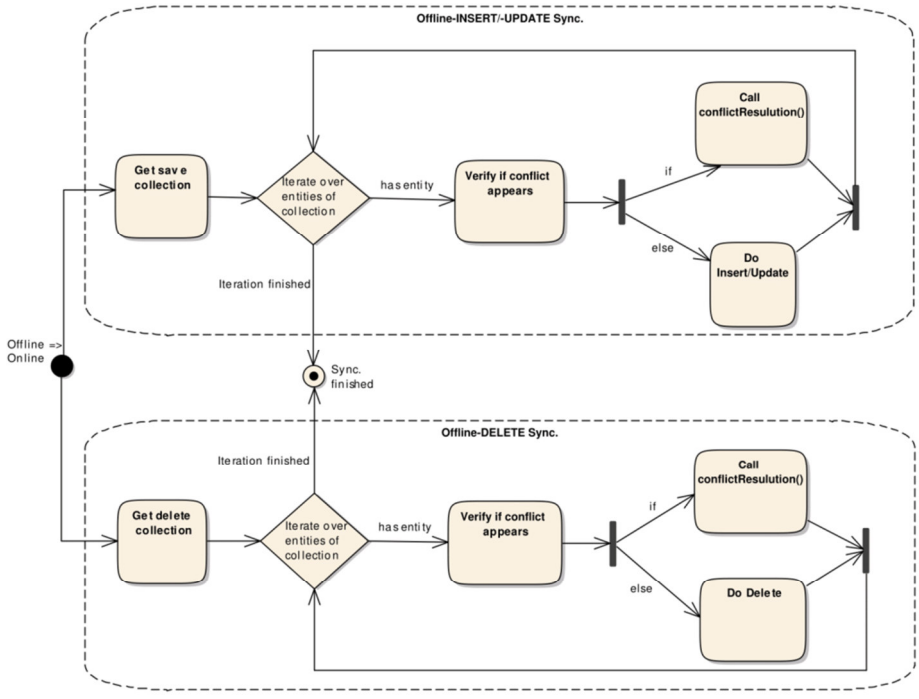



Abb. 12: Sync.-Mechanismus vereinfacht dargestellt

Wie die obige Abbildung zeigt, ist der Auslöser für beide Mechanismen der Wechsel des Verbindungsstatus von offline zu online. Beendet ist die Synchronisation, sobald alle relevanten Datenobjekte beider Mechanismen auf Konflikte überprüft wurden und mit den Daten des Servers übereinstimmen. Nachfolgend werden die Abläufe für beide Mechanismen genauer aufgelistet.

**Offline-INSERT/-UPDATE Mechanismus:**

1. Lade die *collection* von im Offline-Modus gespeicherten Datenobjekte mittels *db.addcollection('savedObjectsCollection')* und speichere diese in der Variable *collection*.

2. Iteriere über das Array von Einträgen in der *collection* und speichere den Eintrag der Iteration in der Variable *entity*.

3. Überprüfe, ob die *entity* bereits das Attribut *version* besitzt.

4. Fall 1: Das Attribut besitzt keine *version*. Das bedeutet, dass dieses Objekt im Offline-Modus eingefügt wurde. Sende einen Insert-Request mit der zugehörigen Tabelle und der *entity* als Übergabeparameter an den Orestes-Server.

5. Fall 1.1: Der Insert-Request des Datenobjektes liefert keinen Fehler zurück. Das Einfügen konnte durchgeführt werden.

6. Fall 1.2: Der Insert-Request des Datenobjektes liefert ein Fehler mit dem Status *Object already exists* zurück. Dass bedeutet, dass serverseitig bereits ein Objekt mit der *entity.Id* existiert. Daher besteht ein Konflikt, welcher durch den Aufruf der entsprechenden Methode gelöst werden muss.

7. Fall 2: Das Attribut besitzt eine *version*. Das bedeutet, dass dieses Objekt im Offline-Modus aktualisiert wurde. Sende einen Update-Request mit der zugehörigen Tabelle, der *entity.Id* und der *entity* als Übergabeparameter an den Orestes-Server. Die Version der *entity* wird als Match-Header angehängt.

8. Fall 2.1: Der Update-Request des Datenobjektes liefert keinen Fehler zurück. Das bedeutet, dass die Version des lokalen Objektes mit der Version des serverseitigen Objektes übereinstimmt und daher das Aktualisieren durchgeführt werden konnte.

9. Fall 2.2: Der Update-Request des Datenobjektes liefert ein Fehler mit dem Status *Object out of date* zurück. Dass bedeutet, dass serverseitig eine aktuellere Version des gleichen Datenobjektes existiert. Daher besteht ein Konflikt, welcher durch den Aufruf der entsprechenden Methode gelöst werden muss.

10. Lösche abschließend die Referenz auf die *entity* aus der *savedObjectsCollection*.

**Offline-DELETE Mechanismus:**

1. Lade die *collection* von im Offline-Modus gelöschten Datenobjekte mittels *db.addcollection('deletedObjectsCollection')* und speichere diese in der Variable *collection*.

2. Iteriere über das Array von Einträgen in der *collection* und speichere den Eintrag der Iteration in der Variable *entity*.

3. Sende einen Delete-Request mit der Tabelle und der *entity.Id* als Übergabeparameter an den Orestes-Server. Die Version der *entity* wird als Match-Header angehängt.

4. Fall 1: Der Delete-Request des Datenobjektes liefert keinen Fehler zurück. Das bedeutet, dass die Version des lokalen Objektes mit der Version des serverseitigen Objektes übereinstimmt und daher das Löschen durchgeführt werden konnte.

5. Fall 2: Der Delete-Request des Datenobjektes liefert ein Fehler mit dem Status *Object out of date* zurück. Dass bedeutet, dass serverseitig eine aktuellere Version des gleichen Datenobjektes existiert. Daher besteht ein Konflikt, welcher durch den Aufruf der entsprechenden Methode gelöst werden muss.

6. Lösche abschließend die Referenz auf die *entity* aus der *deletedObjectsCollection*.

Beide Mechanismen der Synchronisierung nutzen intensiv die Funktionsweise der *Baqend* Methoden *delete()* und *save()*. Die genaue Funktionalität dieser Methoden kann im Guide von *Baqend* [Ba17] nachgelesen werden. Aufgrund der Tatsache, dass die Synchronisierung jedes einzelnen Datenobjektes, erst nach dessen erfolgreicher Zusammenführung mit den remote Daten als abgeschlossen angesehen wird, können Fehlertypen wie App-Abstürze oder Netzwerkabbrüche keine nachhaltige Dateninkonsistenz erzeugen. Sollte der Mechanismus der Synchronisierung durch einen dieser Fehlertypen zum Abbruch kommen, sind alle Datenobjekte, welche bereits durch die Iteration innerhalb der Synchronisierung behandelt wurden, als konfliktfrei anzusehen. Diejenigen Datenobjekte, welche nicht durch diese Iteration gelaufen sind, werden erst nach einem erneuten Wechsel des Netzwerkstatus von offline zu online und den damit verbundenen Anstoß der Synchronisierung, mit den Serverdaten verglichen.

## 4.7 Konfliktauflösung

Bei der Zusammenführung von Datenobjekten aus unterschiedlichen Datenquellen besteht grundsätzlich die Gefahr, dass die Konsistenz der Datenstrukturen verletzt wird. Hierbei geht es vor allem darum, die Vollständigkeit und Korrektheit der Datenstrukturen zu gewährleisten [NH02]. Konflikte in der unterschiedlichen Beschaffenheit von gleichen Datenobjekten, müssen demnach zuverlässig erkannt und behoben werden.

Zur Identifikation von in Konflikt stehenden Datenobjekten, soll wie bereits im vorherigen Kapitel beschrieben, die vorhandene Logik von *Orestes* verwendet werden. Diese stützt sich vor Allem auf den Vergleich von Objektversionen durch den Einsatz von Versionsnummern. Hierbei darf ein Objekt nicht durch ein anderes Objekt mit einer geringeren Version verändert bzw. verdrängt werden. Eine Anfrage (Insert, Update oder Delete), welche dieses Kriterium nicht erfüllt, wird durch einen Fehler mit entsprechendem Fehlerstatus beantwortet. Sollte also beispielsweise eine UPDATE-Operation an einem Serverobjekt mit höherer Versionsnummer angefragt werden, wird ein Fehler mit dem Status *Object out of date* zurückgeliefert. Falls ein solches Szenario während der Synchronisierung auftritt, besteht ein Konflikt, welcher durch geeignete Konfliktlösungsstrategien überwunden werden muss. In Anlehnung an [Go15] werden zur Konfliktbewältigung die folgenden drei Lösungsstrategien verfolgt:

- **Local copy**: Das lokale Datenobjekt wird den Änderungen am serverseitigen Datenobjekt vorgezogen. Als Folge dessen wird das serverseitige Datenobjekt durch das lokale Datenobjekt überschrieben (Standardverhalten).

- **Server copy**: Das serverseitige Datenobjekt wird den Änderungen am lokalen Datenobjekt vorgezogen. Als Folge dessen wird das lokale Datenobjekt durch das serverseitige Datenobjekt überschrieben (Optional).

- **Individual callback**: Die Konfliktauflösung findet durch die vom Nutzer übergebene Callback-Funktion statt. Das Ergebnis ist von der individuellen Implementierung dieser Funktion abhängig (Optional).

Nachdem durch die hier formulierten Konfliktlösungsstrategien jenes Datenobjekt ermittelt ist, welches als Sieger aus dem Konflikt hervorgeht, wird die entsprechende Operation durchgeführt. Für das Beispiel der UPDATE-Operation würde dies bedeutet, dass in Annahme eines Sieges des lokalen Datenobjektes, das serverseitige Objekt durch die lokale Version überschrieben wird. Durch die formulierten Lösungsstrategien soll eine ausgewogene Verteilung von festen und individuellen Verhaltensmustern erreicht werden.

# 5 Implementierung

Das vorliegende Kapitel thematisiert die Umsetzung des in Kapitel 4 skizzierten Entwurfs der Implementierung. Hierzu findet zunächst eine Erläuterung der grundlegenden Struktur statt. Anschließend werden in den verschiedenen Unterkapiteln, die wichtigsten Bestandteile des Artefaktes anhand deren Codestruktur vorgestellt. Ziel ist es, einen grundlegenden Überblick über das implementierte Artefakt zu schaffen.

## 5.1 Grundlegende Struktur

Aufgrund der in Kapitel 4.2 dargestellten Limitationen bezüglich einer Umsetzung mittels *Service Worker*, ist das Artefakt in das *Baqend SDK* integriert. Die in dieser Arbeit entwickelte Programmlogik befindet sich hierbei in der Klasse *OfflineService*. Die Umsetzung ist in *JavaScript* [JS17] geschrieben und entspricht den ES6 Standards [ES17]. Als Drittanbieter-Software wird *minimongo* [MM17] eingesetzt, welches als MongoDB-Wrapper für den lokalen Datenspeicher *IndexedDB* dient. Funktional bietet die implementierte Klasse neben der Kommunikation mit dem lokalen Datenspeicher, auch die Synchronisierung mit dem Remoteserver und bei Bedarf die Auflösung von Konflikten.

Im Folgenden kann eine Auflistung der umgesetzten Methoden der Klasse *OfflineService* eingesehen werden. Eine detaillierte Beschreibung der zentralen Methoden wird in den jeweiligen Unterkapiteln dieses Kapitels vorgenommen. Außerdem kann der vollständige Programmcode der Anlage 4 entnommen werden.

- *constructor()*: Initialisierung der notwendigen Variablen und Zuweiseung des Attributes *online*, welches ein *observable object* darstellt und Veränderungen in der Netzwerkverbindung kommuniziert (siehe Kapitel 5.3).

- *isOnline()*: Gibt Auskunft über den aktuellen Netzwerkstatus in Form eines booleschen Ausdrucks.

- *saveDataLocally()*: Speichert das übergebene Objekt in die entsprechende *collection* des lokalen Datenspeichers (siehe Kapitel 5.4).

- *deleteLocalEntity()*: Löscht das übergebene Objekt aus der entsprechenden *collection* des lokalen Datenspeichers (siehe Kapitel 5.4).

- *loadLocalEntity()*: Lädt das übergebene Objekt aus der entsprechenden *collection* des lokalen Datenspeichers (siehe Kapitel 5.4).

- *queryLocalData()*: Stellt eine Anfrage an den lokalen Datenspeicher mittels der übergebenen Attribute *query*, *limit* und *sort* (siehe Kapitel 5.2).

- *synchronizeSavedData()*: Synchronisiert die im Offline-Modus erstellten bzw. aktualisierten Datenobjekte mit dem Remoteserver (siehe Kapitel 5.5).

- *synchronizeDeletedData()*: Synchronisiert die im Offline-Modus gelöschten Datenobjekte mit dem Remoteserver (siehe Kapitel 5.5).

- *errorHandling()*: Erkennt Konflikte bei der Kommunikation mit dem Remoteserver und führt nach deren Auflösung (siehe *conflictResolution()*) die entsprechende Logik aus (siehe Kapitel 5.6).

- *conflictResolution()*: Löst Konflikte mittels der vom Nutzer eingestellten Strategie auf (siehe Kapitel 5.6).

Die Integration in die Logik des *Baqed SDK* findet mittels einer Weiche zwischen Online- und Offline-Verhalten statt. Hierbei wird durch die bereitgestellte Methode *isOnline()* festgestellt, ob die ursprüngliche Logik des SDK oder die des neuen *OfflineService* ausgeführt werden muss. Wenn also beispielsweise eine Anfrage zu einem spezifischen Datenobjekt gestellt wird, entscheidet das SDK, ob der Remoteserver erreichbar ist und ruft im Falle einer fehlenden Netzwerkverbindung die Methode *queryDataLocally()*, anstatt die Anfrage an den Remoteserver zu stellen.

## 5.2 Lokale Anfrageverarbeitung

Die hier vorgestellte Methode *queryLocalData()* dient der Beantwortung von MongoDB-Queries an den lokalen Datenspeicher. Die Anfrage wird hierbei als Zeichenkette formatiertes JSON-Objekt übergeben. Zusätzlich zur Anfrage muss das Sortierkriterium übergeben werden, dessen Format dem der Anfrage entspricht. Außerdem werden das Limit und der Name der *collection* als Zeichenkette erwartet.

```
queryLocalData(query, sort, limit, entityClass) {
    let localQueryResult;
    let promise = new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            limit = limit < 0 ? 0 : limit;
            localDb.addCollection(entityClass, () => {
                localDb[entityClass].find(JSON.parse(query), {sort: JSON.parse(sort)}, {limit: JSON.parse(limit)}).fetch((result) => {
                    localQueryResult = result;
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });
    });

    return promise.then(() => localQueryResult);
};
```

Abb. 13: Methode zum Anfragen von lokalen Datenobjekten

Die Methode besitzt die lokale Variable *localQueryResult*, welcher im Programmverlauf das Ergebnis der Abfrage zugewiesen wird. Da die Kommunikation mit dem lokalen Datenspeicher asynchron stattfindet, wird das Konzept der *Promises* [Pr17] eingesetzt, welches es ermöglicht, auf die Terminierung der Logik zu warten. Innerhalb des *Promises* wird zunächst eine Verbindung mit dem lokalen Datenspeicher hergestellt. Nachdem sichergestellt ist, dass das Limit nicht kleiner als 0 ist, wird die *collection* anhand des übergebenen Bezeichners geladen. An dieser wird anschließend eine Anfrage mittels *find()* gestellt, welche die übergebenen Parameter *query*, *sort* und *limit* verwendet. Da diese Parameter als Zeichenketten übergeben werden, muss mittels *JSON.parse()*, eine Umwandlung in das ursprüngliche JSON-Objekt durchgeführt werden. Sollte die Anfrage keinen Fehler verursachen, wird wie bereits angedeutet, das Ergebnis dieser Anfrage in der lokalen Variable *localQueryResult* gespeichert und das *Promise* erfüllt (*resolve()*). Im Falle eines Fehlers wird das *Promise* zurückgewiesen (*reject()*). Abschließend gibt die Methode die lokale Variable *localQueryResult* zurück, welche entweder einem Ergebnis oder dem Wert *undefined* entspricht.

## 5.3 Offline Erkennung

Neben der Methode *isOnline()*, welche den aktuellen Netzwerkstatus der Web(-Applikation) wiedergibt, wird im Konstruktor des Artefaktes, das Attribut *online* initialisiert. Dieses Attribut ist ein *observable object*, auf welches gehorcht werden kann (*db.online.subscribe()*), um Veränderungen der Netzwerkverbindung zu identifizieren. Dadurch wird die Möglichkeit eingeräumt, differenzierte Mechanismen in Abhängigkeit der entsprechenden Ausprägungen des *online* Attributes auszuführen. Ein Beispiel hierfür kann das Festlegen von verschiedenen Anfragen sein, welche im Falle einer vorhandenen Netzwerkverbindung ausgeführt werden sollen, um hierdurch den lokalen Datenspeicher zu aktualisieren.

```
constructor(entityManager) {
    this.entityManager = entityManager;
    this.entityManagerFactory = entityManager.entityManagerFactory;

    this.online = new Observable((observer) => {
        if(window) {
            window.addEventListener('load', () => {
                window.addEventListener('offline', (e) => {
                    observer.next(false);
                });
                window.addEventListener('online', (e) => {
                    this.synchronizeData().then(() => {
                        observer.next(true);
                    });
                });
            });
            observer.next(navigator.onLine);
        }
    });
}
```

Abb. 14: Konstruktor der Klasse OfflineService

Zunächst wird geprüft, ob das *window* Attribut am Dokument verfügbar ist. Sollte dies der Fall sein, werden nachdem das Laden der Ressourcen abgeschlossen ist, zwei verschiedene *EventListener* an dem Dokument registriert. Zum einen der Offline-Listener, welcher dem *observable object* den Wert *false* (nicht online) zuweist. Zum anderen der Online-Listener, welcher zunächst den Mechanismus der Synchronisierung (siehe Kapitel 5.5) anstößt und nach Abschluss dessen, dem *observable object* den Wert *true* (online) zuweist. Dadurch kann gewährleistet werden, dass jegliche Logik, welche im Online-Modus durchgeführt wird, erst nach erfolgreicher Synchronisierung mit dem Server angestoßen wird. Dies dient dem Zweck, das Überschreiben von nicht synchronisierten Offline-Updates zu unterbinden.

## 5.4 CRUD Operationen

Neben dem gezielten Anfragen von Datenobjekte aus dem lokalen Datenspeicher, besteht ebenfalls die Möglichkeit, CRUD (create, read, update, delete) Operationen im Offline-Modus durchzuführen. Hierzu stehen die Methoden *loadLocalEntity()*, *saveDataLocally()* und *deleteLocalEntity()* zur Verfügung. Diese werden folgend genauer beschrieben.

```
loadLocalEntity(entity, entityClass) {
    let localLoadResult;
    let promise = new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(entityClass, () => {
                localDb[entityClass].findOne({ _id: entity.id }, {}, (result) => {
                    localLoadResult = result;
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });
    });

    return promise.then(() => localLoadResult);
}
```

Abb. 15: Methode zum Laden von lokalen Datenobjekten

Die Methode *loadLocalEntity()* ist für das Laden von lokalen Datenobjekten verantwortlich und erwartet das Datenobjekt (*entity*) und den Namen der *collection* (*entityClass*) als Übergabeparameter. Der lokalen Variable *locaLoadResult* wird im Programmverlauf das Ergebnis der Ladeoperation zugewiesen. Auch in dieser Methode, wird für die Kommunikation mit dem lokalen Datenspeicher, das Konzept der *Promises* eingesetzt. Innerhalb des *Promises* wird zunächst eine Verbindung mit dem lokalen Datenspeicher hergestellt. Anschließend wird die *collection* anhand des übergebenen Bezeichners geladen. An dieser wird sodann eine Anfrage mittels *findOne()* gestellt, welche das gewünschte Datenobjekt anhand der *id* der übergebenen *entity* sucht. Sollte die Anfrage keinen Fehler verursachen, wird das Ergebnis der Anfrage in der lokalen Variable *locaLoadResult* gespeichert und das *Promise* erfüllt (*resolve()*). Im Falle eines Fehlers wird das *Promise* zurückgewiesen (*reject()*). Abschließend gibt die Methode die lokale Variable *locaLoadResult* zurück, welche entweder einem Ergebnis oder dem Wert *undefined* entspricht.

```
saveDataLocally(entity, entityClass, offlineSave) {
    let promise = new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(entityClass, () => {
                entity._id = entity.id;
                localDb[entityClass].upsert(entity, () => {
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });

    });

    return promise.then(() => {
        if(offlineSave) {
            return this.saveDataLocally(entity, savedObjectsCollection, false);
        }
    });
};
```

Abb. 16: Methode zum Speichern von lokalen Datenobjekten

Die Methode *saveDataLocally()* behandelt das Einfügen und das Aktualisieren von lokalen Datenobjekten. Als Übergabeparameter wird das Datenobjekt (*entity*), der Name der *collection* (*entityClass*) und ein boolescher Ausdruck erwartet. Letzterer dient der Identifikation von Speichervorgängen, welche im Offline-Modus stattgefunden haben. Nachdem eine Verbindung zu dem lokalen Datenspeicher hergestellt und die entsprechende *collection* geladen ist, wird das Attribut *_id* der *entity*, gleich der vom Server generierten *id* gesetzt. Dies muss zwingend ausgeführt werden, da der lokale Datenspeicher, die Datenobjekte anhand des Attributes *_id* identifiziert. Für den Fall, dass dieses Attribut nicht gesetzt ist, wird lokal eine eigene Ausprägung generiert. Dadurch sind für dasselbe Datenobjekt unterschiedliche Bezeichner vorhanden, was die Komplexität der Kommunikation zwischen lokalem und remote Datenspeicher unnötig steigert. Auf Grund dessen, wird durch diese Logik die Einheitlichkeit zwischen unterschiedlichen Datenspeichern gewährleistet.

Anschließend wird die übergebenen *entity* durch die *upsert()* Methode eingefügt bzw. aktualisiert. Sollte die Operation keinen Fehler verursachen, wird das *Promise* erfüllt (*resolve()*) und andernfalls zurückgewiesen (*reject()*). Nachdem die komplette Logik innerhalb des *Promise* abgeschlossen ist, wird geprüft, ob es sich bei der jeweiligen Speicheroperation um eine im Offline-Modus durchgeführte Operation handelt. Sollte dies der Fall sein, wird die entsprechende *entity* zusätzlich in der *savedObjectsCollection* gespeichert. Dies dient dazu, die im Offline-Modus gespeicherten Datenobjekte für den Mechanismus der Synchronisierung zu vermerken. Dadurch können diese später mit dem Remoteserver zusammengeführt werden.

```
deleteLocalEntity(entity, entityClass, offlineDelete) {
    let promise = new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(entityClass, () => {
                localDb[entityClass].remove(entity.id, (res) => {
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });
    });

    return promise.then(() => {
        if(offlineDelete) {
            if(entity.version) {
                return this.saveDataLocally(entity, deletedObjectsCollection, false);
            } else {
                return this.deleteLocalEntity(entity, savedObjectsCollection, false);
            }
        }
    });
}
```

Abb. 17: Methode zum Löschen von lokalen Datenobjekten

Für das Löschen von lokalen Datenobjekten kann die Methode *deleteLocalEntity()* verwendet werden. Als Übergabeparameter wird das Datenobjekt (*entity*), der Name der *collection* (*entityClass*) und der boolesche Ausdruck *offlineDelete* erwartet. Letzterer dient der Identifikation von Löschvorgängen, welche im Offline-Modus stattgefunden haben. Auch hier wird zunächst eine Verbindung mit dem lokalen Datenspeicher und der relevanten *collection* hergestellt. Anschließend wird das übergebene Datenobjekt anhand dessen *id* aus dem lokalem Datenspeicher gelöscht. Hierbei wird das Objekt allerdings nicht physikalisch verworfen, sondern lediglich dessen Status auf *removed* gesetzt. Sollte die Operation keinen Fehler verursachen, wird das *Promise* erfüllt (*resolve()*) und andernfalls zurückgewiesen (*reject()*).

Im Anschluss an die im *Promise* ausgeführte Logik, wird geprüft, ob es sich bei der jeweiligen Löschoperation um eine im Offline-Modus durchgeführte Operation handelt. Falls dies der Fall sein sollte, wird die *entity* dahingehend untersucht, ob diese eine *version* besitzt. Denn wenn dies nicht zutrifft, wurde das Datenobjekt im Offline-Modus erzeugt und im Laufe der Methode wieder gelöscht. Dies bedeutet, dass die während des Offline-Speicherns vermerkte Referenz in der *savedObjectsCollection,* wieder entfernt werden kann, da diese *entity* nicht länger relevant für den Mechanismus der Synchronisierung ist. Sollte allerdings eine *version* vorhanden sein, muss die Löschoperation durch eine Referenz auf das Datenobjekt in der *deletedObjectsCollection* vermerkt werden. Denn in diesem Fall ist diese Operation relevant für die spätere Synchronisierung.

## 5.5 Synchronisierung

Das Zusammenführen von lokalen und serverseitigen Datenobjekten ist eine zentrale Aufgabe des implementierten Artefaktes. Die hierfür zuständigen Methoden *synchronizeSavedData()*, *synchronizeDeletedData()* und *errorHandling()* werden beim Wechsel des Netzwerkstatus von offline zu online angestoßen. Im Folgenden werden diese drei Methoden genauer betrachtet.

```
synchronizeSavedDate() {
    return new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(savedObjectsCollection, () => {
                localDb[savedObjectsCollection].find({}).fetch((result) => {
                    if(result) {
                        let mapPromises = result.map((entity) => {
                            let bucket = this.getBucketOfEntity(entity);
                            let key = this.getKeyOfEntity(entity);
                            let msg;
                            delete entity._id;
                            if(entity.version) {
                                msg = new message.ReplaceObject(bucket, key, entity)
                                    .ifMatch(entity.version);
                            } else {
                                msg = new message.CreateObject(bucket, entity);
                            }

                            return this.entityManager.send(msg).then((response) => {
                                return this.saveDataLocally(response.entity, bucket, false).then(() => {
                                    return this.deleteLocalEntity(entity, savedObjectsCollection, false);
                                });
                            }, (e) => {
                                return this.errorHandling(e, entity, 'save').then(() => {
                                    return this.deleteLocalEntity(entity, savedObjectsCollection, false);
                                })
                            });
                        });

                        Promise.all(mapPromises).then(() => resolve());
                    } else {
                        resolve();
                    }
                });
            });
        });
    });
}
```

Abb. 18: Methode zum Synchronisieren von offline gespeicherten Daten

Die grundlegende Struktur der beiden Methoden *synchronizeSavedData()* und *synchronizeDeletedData()* ist nahezu identisch. Beide sind in einem *Promise* implementiert, um die Möglichkeit zu bieten, auf deren Terminierung zu warten. In beiden Fällen wird eine Verbindung zum lokalen Datenspeicher hergestellt und anschließend die entsprechende *collection* (*savedObjectsCollection* bzw. *deletedObjectsCollection*) geladen. Sobald alle Einträge der jeweiligen *collection* geladen sind, werden diese innerhalb einer *map()* Operation durchlaufen. Die Logik innerhalb dieser Iteration ist jedoch für beide Methoden zum Teil unterschiedlich. Beide Methoden benötigen den Namen der *collection* (*bucket*) und den Bezeichner (*key*) des Datenobjektes der jeweiligen Iteration. Im Falle der *synchronizeSavedData()* Methode wird ebenfalls das Attribute *_id* der *entity* gelöscht, da dieses lediglich lokal zur Identifikation genutzt wird und dem Datenschema des Servers nicht bekannt ist. In der Kommunikation mit dem Server besteht nun der entscheidende

Unterschied zwischen beiden Methoden. Während die *synchronizeSavedData()* Methode anhand der Ausprägung des Attributes *version* erkennt, ob es sich bei dem gespeicherten Objekt um einen *insert* oder einem *update* handelt, verarbeitet die *synchronizeDeletedData()* Methode ausschließlich *delete* Operationen. Entsprechend der Ausprägung wird die *Message* für den Server erstellt (beispielsweise *CreateObject()* für ein *insert*) und gegebenenfalls mit einem *ifMatch()* Header versehen. Dieser dient dem Vergleich der Versionen von lokalen und remote Daten.

```
synchronizeDeletedDate() {
    return new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: appName },  () => {
            localDb.addCollection(deletedObjectsCollection, () => {
                localDb[deletedObjectsCollection].find({}).fetch((result) => {
                    if (result) {
                        let mapPromises = result.map((entity) => {
                            let bucket = this.getBucketOfEntity(entity);
                            let key = this.getKeyOfEntity(entity);

                            return this.entityManager.send(new message.DeleteObject(bucket, key).ifMatch(entity.version)).then(() => {
                                return this.deleteLocalEntity(entity, deletedObjectsCollection, false);
                            }, (e) => {
                                return this.errorHandling(e, entity, 'delete').then(() => {
                                    return this.deleteLocalEntity(entity, deletedObjectsCollection, false);
                                });
                            });
                        });

                        Promise.all(mapPromises).then(() => resolve());
                    } else {
                        resolve();
                    }
                });
            });
        });
    });
}
```

Abb. 19: Methode zum Synchronisieren von offline gelöschten Daten

Nachdem die *Message* an den Server gesendet wurde, wird festgestellt in welcher Weise der Server auf die Anfrage reagiert. Sollte kein Fehler vom Server ausgeliefert werden, konnte die jeweilige Operation ohne Konflikt ausgeführt werden. In diesem Fall wird die Referenz des Datenobjektes der aktuellen Iteration, aus der jeweiligen *collection* gelöscht. Andernfalls wirft der Server einen Fehler. Zur Verifizierung, um welche Art von Konflikt es sich handelt, wird die Methode *errorHandling()* gerufen.

```
errorHandling(error, entity, operation) {
    let bucket = this.getBucketOfEntity(entity);
    let key = this.getKeyOfEntity(entity);

    return new Promise((resolve, reject) => {
        if(error.status === StatusCode.OBJECT_OUT_OF_DATE || error.status === StatusCode.CONFLICT) {
            this.entityManager.send(new message.GetObject(bucket, key)).then((response) => {
                if(this.conflictResolution(entity, response.entity) === response.entity) {
                    this.saveDataLocally(response.entity, bucket, false).then(() => resolve());
                } else {
                    if(operation === 'save') {
                        delete entity.version;
                        this.entityManager.send(message.ReplaceObject(bucket, key, entity)).then((response) => {
                            this.saveDataLocally(response.entity, bucket, false).then(() => resolve());
                        });
                    } else if(operation === 'delete') {
                        this.entityManager.send(new message.DeleteObject(bucket, key)).then(() => resolve());
                    }
                }
            }, (e) => {
                if (e.status === StatusCode.OBJECT_NOT_FOUND && operation === 'save') {
                    if(this.conflictResolution(entity, 'remoteObj') === entity) {
                        this.entityManager.send(new message.CreateObject(bucket, entity)).then(() => resolve());
                    } else {
                        this.deleteLocalEntity(entity, bucket, false).then(() => resolve());
                    }
                } else if(e.status === StatusCode.OBJECT_NOT_FOUND && operation === 'delete') {
                    resolve();
                } else {
                    reject();
                }
            });
        } else {
            reject();
        }
    });
}
```

Abb. 20: Methode zum Verifizieren von Konflikten

Die Methode *errorHandling()* ist sowohl für die Konfliktverarbeitung von gespeicherten, als auch gelöschten Datenobjekten verantwortlich. Dementsprechend wird neben dem Fehler (*error*) und dem Datenobjekt (*entity*) auch die *operation* als Übergabeparameter erwartet. Letztere dient zur Identifikation, um welche Operation (*save* oder *delete*) es sich handelt. Generell wird zwischen zwei Fehlerszenarien unterschieden. Zum einen, ob die Versionsnummer des lokalen Datenobjektes nicht mit der Serverversion übereinstimmt (für *update* und *delete*) oder ob das gewünschte Objekt bereits auf dem Server existiert (für *insert*). Und zum anderen, ob das gewünschte Datenobjekt nicht mehr auf dem Server existiert (für *update*). Im ersten Fall muss zunächst das Serverobjekt durch *Message.GetObject()* geladen werden, um es innerhalb der Konfliktauflösung (siehe Kapitel 5.6) zu verwenden. Entsprechend dem Ergebnis der Konfliktauflösung und der Ausprägung der *operation* wird entweder das lokale Datenobjekt durch das Serverobjekt überschrieben oder das Serverobjekt aktualisiert bzw. gelöscht.

Der Fehlerfall, dass das Serverobjekt nicht mehr vorhanden ist, hat lediglich für *save* Operationen eine ausführlichere Betrachtung zur Folge. Denn falls eine *delete* Operation diesen Konflikt verursacht, entspricht dies keinem Konflikt im eigentlichen Sinne, da die gewünschte Operation bereits dem Serverstand entspricht. Solle es sich allerdings nicht um eine *delete* Operation handeln, wird mittels der *conflictResolution()* Methode (siehe Kapitel 5.6) entschieden, ob die lokale Änderung durchgeführt werden kann oder nicht. Sollte die lokale Änderung gegenüber der Serverversion gewinnen, wird das lokale Objekt auf dem Server neu angelegt. Im gegensätzlichen Fall, wird das lokale Datenobjekt gelöscht und dadurch die Serverversion vorgezogen.

Nach Durchlauf der Konfliktverifizierung und der damit verbundenen Logik, wird die nächste Iteration der jeweiligen *collection* angestoßen und der Prozess startet von neuem. Sobald jede *entity* behandelt wurde, gilt der Mechanismus der Synchronisierung als abgeschlossen und alle *Promises* als erfüllt.

## 5.6 Konfliktauflösung

Die *conflictResolution()* Methode des Artefaktes dient dazu, programmatisch zu entscheiden, welches der beiden übergebenen Objekte vorgezogen werden muss. Es handelt sich hierbei demnach, um eine zur Konfliktauflösung geeignete Methode. Die Strategie, wie ein spezifischer Konflikt gelöst werden kann, hängt stark von der Konfiguration des Nutzers ab. Dieser hat nämlich die Möglichkeit zu entscheiden, welche Strategie bei der Auflösung von Konflikten verfolgt werden soll. Hierzu existiert am *db* Objekt des *Baqend SDK* die Methode *configure()*. An dieser wurde das neue Attribut *conflictResolution* erstellt, welches entweder eine Zeichenkette oder eine individuelle Funktion enthalten kann. Die Konfiguration kann daher mittels *db.conigure({conflictResoltion})* durchgeführt werden.

```
conflictResolution(localObj, remoteObj) {
    if(this.entityManagerFactory.conflictResolution) {
        if (typeof this.entityManagerFactory.conflictResolution === "function") {
            return this.entityManagerFactory.conflictResolution(localObj, remoteObj);
        } else if(this.entityManagerFactory.conflictResolution === 'remote') {
            return remoteObj;
        } else {
            return localObj;
        }
    } else {
        return localObj;
    }
}
```

Abb. 21: Methode zum Auflösen von Konflikten

Die Methode überprüft zunächst, ob eine entsprechende Konfiguration durch den Nutzer vorgenommen wurde. Sollte dies nicht der Fall sein, ist das Standardverhalten bei der Auflösung von Konflikten, dass lokale Änderungen vorgezogen werden. Falls eine Konfiguration seitens des Nutzers vorgenommen wurde, wird untersucht, ob es sich hierbei um die Zeichenkette *remote* handelt. Diese zieht serverseitige Änderungen den lokalen Änderungen vor. Eine weitere Möglichkeit ist die Definition eines individuellen *callback* zur Konfliktauflösung. In jedem Fall gibt die Methode entweder das lokale oder das serverseitige Datenobjekt zurück.

# 6 Evaluation

Zur Einordnung der Güte des implementierten Artefaktes, findet im Folgenden eine qualitative Evaluation der im Kapitel 4.1 formulierten funktionalen Anforderungen statt. Hierzu wird zunächst der Aufbau der Studie (Fallstudie) beschrieben und anschließend die verschiedenen Anwendungsfälle definiert. Nachdem das für die Studie eingesetzte Szenario und die Testdurchführung genauer erläutert sind, werden die Ergebnisse der Fallstudie vorgestellt.

## 6.1 Aufbau der Studie

Für die Evaluation des in dieser Arbeit implementierten Artefaktes, wird die Methode der Fallstudie (*case study*) verwendet. Nach [RH09] wird unter der Fallstudie eine empirische Forschungsmethode verstanden, welche ein spezifisches Phänomen innerhalb dessen natürlichem Umfeld untersucht. Hierzu werden zuvor formulierte Forschungsfragen, anhand von einem oder mehreren definierten Anwendungsfällen (*use cases*) geprüft und anschließend ausgewertet. Die Grundlage der Untersuchung bilden die im Kapitel 4.1 definierten funktionalen Anforderungen. Hierbei werden die Anforderungen „Kompatibilität“ und „Offline Erkennung“ lediglich implizit untersucht. Grund hierfür ist der fundamentale Charakter dieser Funktionalitäten. Sprich die Erfüllung dieser wurde bereits während der Implementierung ausgiebig getestet. Die restlichen funktionalen Anforderungen werden explizit durch Forschungsfragen evaluiert. Hieraus können demnach die folgenden Fragen abgeleitet werden:

- *RQ1:* Werden Anfragen im Offline-Modus erwartungsgemäß durch den lokalen Datenspeicher beantwortet?

- *RQ2:* Werden CRUD Operationen im Einzelbenutzersystem korrekt und konsistenzfrei zwischen lokalem und remote Datenspeicher synchronisiert?

- *RQ3:* Werden CRUD Operationen im Mehrbenutzersystem korrekt und konsistenzfrei zwischen lokalem und remote Datenspeicher synchronisiert?

Zur Überprüfung dieser Forschungsfragen dienen die im nachfolgenden Kapitel 6.2 thematisierten Anwendungsfälle. Diese besitzen neben einer expliziten Beschreibung, außerdem einen erwarteten Zustand, welcher erfüllt sein muss, nachdem der Anwendungsfall abgeschlossen ist. Hierbei wird bei der Beurteilung des Erfüllungsgrades zwischen *erfüllt* und *nicht erfüllt* unterschieden. Um die Testumgebung so natürlich wie möglich zu konzipieren, verwendet das Testszenario (siehe Kapitel 6.3) das in dieser Arbeit erweiterte *Baqend SDK*, welches unter anderem die Kommunikation im dem Datenspeicher

im Offline-Modus beinhaltet. Für das Zwischenspeichern von statischen Webressourcen (HTML, CSS, JavaScript etc.), wird der in der Masterarbeit [De17] konzipierte *Service Worker* eingesetzt. Eine Simulation des Offline-Modus, findet anhand der browserinternen Funktion statt (siehe Kapitel 6.4).

## 6.2 Anwendungsfälle

Die Formulierung von anschaulichen Anwendungsfällen, ist die Grundlage für eine möglichst realitätsnahe Untersuchung der funktionalen Eigenschaften des implementierten Artefaktes. Da die Datenobjekte von (Web-)Applikationen in der Regel nicht nur von einem, sondern vielmehr von einer Vielzahl an Nutzern konsumiert und verändert werden können, sollen die in diesem Kapitel definierten Anwendungsfälle neben dem Einzelbenutzersystem, auch das Mehrbenutzersystem adressieren. Dadurch soll eine Simulation der in der Realität vorherrschenden Bearbeitungsfrequenz stattfinden.

Zum besseren Verständnis der einzelnen Anwendungsfälle, sind diese in einer gleichbleiben Struktur veranschaulicht. Hierbei ist zunächst eine kurze Beschreibung des Anwendungsfalles gegeben. Nachdem die beteiligten Akteure genannt sind, werden die essenziellen Schritte des jeweiligen Anwendungsfalles angegeben. Abschließend enthält jeder Anwendungsfall ein individuelles Ergebnis, welches nach der erfolgreichen Durchführung als erwarteter Zustand verstanden wird. Der folgende Anwendungsfall bezieht sich auf die erste Forschungsfrage und deckt die Anforderungen „lokale Datenspeicherung“ und „Offline Anfragen“ ab. Die folgende Tabelle veranschaulicht diesen ersten Anwendungsfall.

| | |
|---|---|
| Anwendungsfall | Anfrage an ein Datenobjekt A durch den Nutzer 1 im Offline-Modus. |
| Akteure | Nutzer 1 |
| Essenzielle Schritte | 1. Nutzer 1 lädt die Web(-Applikation) im Online-Modus.<br>2. Nutzer 1 fragt das Datenobjekt A im Online-Modus an.<br>3. Nutzer 1 wechselt in den Offline-Modus.<br>4. Nutzer 1 fragt das Datenobjekt A im Offline-Modus an. |
| Erwartetes Ergebnis | Die ausgelieferte Offline-Repräsentation des Datenobjektes A ist identisch der zuletzt angefragten Online-Repräsentation. |

Tab. 7: Anwendungsfall - lokale Anfrage im Offline-Modus

Neben der Verarbeitung von Anfragen im Offline-Modus, ist das Erzeugen und Verändern von lokalen Datenobjekten eine wesentliche Anforderung an das implementierte Artefakt. Daher thematisiert der nachfolgende Anwendungsfall die Durchführung von CRUD Operationen auf ein lokales Datenobjekt im Offline-Modus. Hierbei wurden aus Gründen der Kompaktheit, die einzelnen CRUD Operationen unter einem Anwendungsfall zusammengefasst. Bei der Durchführung der Untersuchung, werden jedoch die Operationen CREATE, READ, UPDATE und DELETE getrennt voneinander vollzogen und protokolliert.

Der Anwendungsfall simuliert die Gegebenheiten in einem Einzelbenutzersystem und bezieht sich daher auf die zweite Forschungsfrage. Speziell werden hierbei die funktionalen Anforderungen „Online-/Offline-Updates“, „Offline-Update-Erkennung“ und „Synchronisierung“ untersucht. Da das Herbeiführen von Konflikten während der CRUD Operationen intendiert ist, werden ebenfalls die Anforderungen „Konflikterkennung“ und „Konfliktauflösung“ geprüft. Ein detaillierter Aufbau dieses Anwendungsfalles kann der Tabelle 8 entnommen werden.

| | |
|---|---|
| Anwendungsfall | Offline CRUD Operation auf ein Datenobjekt A im Einzelbenutzersystem und anschließender Wechsel in den Online-Modus. |
| Akteure | Nutzer 1 |
| Essenzielle Schritte | 1. Nutzer 1 führt eine CRUD Operation auf das Datenobjekt A durch.<br>2. Die CRUD Operation wird persistent im lokalen Datenspeicher gespeichert.<br>3. Nutzer 1 wechselt in den Online-Modus.<br>4. Die Synchronisierung von Datenobjekt A ist erfolgreich |
| Erwartetes Ergebnis | Die im Offline-Modus durchgeführte CRUD Operation muss entsprechend der Konfliktauflösungsstrategie, im Online-Modus von Nutzer 1 konsistent dargestellt werden. |

Tab. 8: Anwendungsfall - CRUD Operation im Einzelbenutzersystem

Wie bereits der vorherige Anwendungsfall, bezieht sich auch der folgende auf die Ausführung von CRUD Operationen auf ein lokales Datenobjekt. Auch hier liegt der Schwerpunkt auf der konsistenten Synchronisierung dieser Veränderungen. Jedoch wird

bei diesem Anwendungsfall nicht die Konsistenzsicherung in einem Einzelbenutzersystem, sondern vielmehr die Zusammenführung von verschiedenen Datenobjekten in einem Mehrbenutzersystem simuliert. Hierbei ist der Mechanismus der Synchronisierung aufgrund der verteilten Datenbestände differenziert zu betrachten und stellt daher eine größere Herausforderung dar. Dieser Anwendungsfall bezieht sich auf die dritte Forschungsfrage, untersucht jedoch die identischen funktionalen Anforderungen, wie bereits der vorherige Anwendungsfall. Die Tabelle 9 veranschaulicht wie bereits die vorherige Tabelle eine kompakte Beschreibung der einzelnen CRUD Operationen.

| | |
|---|---|
| Anwendungsfall | Offline CRUD Operation auf ein Datenobjekt A im Mehrbenutzersystem und anschließender Wechsel in den Online-Modus. |
| Akteure | Nutzer 1, Nutzer 2 |
| Essenzielle Schritte | 1. Nutzer 1 führt eine CRUD Operation auf das Datenobjekt A durch.<br>2. Die CRUD Operation wird persistent im lokalen Datenspeicher gespeichert.<br>3. Nutzer 1 wechselt in den Online-Modus.<br>4. Die Synchronisierung von Datenobjekt A ist erfolgreich<br>5. Nutzer 2 fragt Datenobjekt A im Online-Modus an.<br>6. Nutzer 2 wechselt in den Offline-Modus. |
| Erwartetes Ergebnis | Die im Offline-Modus durchgeführte CRUD Operation muss entsprechend der Konfliktauflösungsstrategie, sowohl im Online- als auch im Offline-Modus von Nutzer 2 konsistent dargestellt werden. |

Tab. 9: Anwendungsfall - CRUD Operation im Mehrbenutzersystem

Durch die in diesem Kapitel formulierten Anwendungsfälle kann sichergestellt werden, dass jegliche im Kapitel 4.1 definierten funktionalen Anforderungen einer expliziten bzw. impliziten Prüfung unterzogen werden können. Hierbei sind die jeweiligen Anwendungsfälle durch individuell zu erwartende Ergebnisse gekennzeichnet, was die Bestimmung eines Erfüllungsgrades der einzelnen Fälle ermöglicht. Diese bilden in der Summe das Ergebnis dieser Evaluation.

## 6.3 Testszenario

Um die im Kapitel 6.2 formulierten Anwendungsfälle durchführen zu können, muss ein dem natürlichen Umfeld entsprechendes Testszenario konzipiert werden. Dieses sollte so gestaltet sein, dass alle relevanten Operationen der jeweiligen Anwendungsfälle, innerhalb des Szenarios ausgeführt werden können. Aus diesem Grund wird eigens für diese Ausarbeitung, eine (Web-)Applikation umgesetzt, welche das um das implementierte Artefakt erweiterte *Baqend SDK* verwendet und demnach mit *Baqend* Technologie umgesetzt ist. Diese (Web-)Applikation stellt den Grundstein für das Testszenario dar und wird im Folgenden vorgestellt.

Die (Web-)Applikation *Trillo* [Tr17] ist eine leichtgewichtige Implementierung zum Erstellen und Verwalten von individuellen Aufgaben, welche auf dem *Angular Framework* [An17] basiert. Beim Aufrufen der Applikation werden alle bereits erstellten Aufgaben geladen und nach deren Priorität sortiert angezeigt. Eine nutzerspezifische Ansicht ist nicht vorhanden, sprich jeder Nutzer kann die Aufgaben jedes anderen Nutzers laden und bearbeiten. Eine Veranschaulichung der Benutzeroberfläche von *Trillo*, kann der nachfolgenden Abbildung entnommen werden.

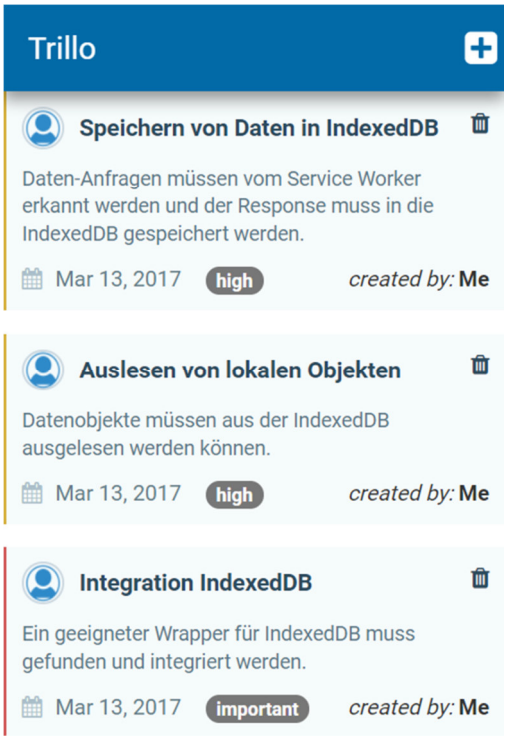



Abb. 22: Trillo App als Grundstein des Testszenarios

Neben dem Laden von bereits existierenden Aufgaben, besteht die Möglichkeit, neue Aufgaben anzulegen. Hierzu steht das Plus-Symbol in der Menüleiste zur Verfügung. Durch einen Klick auf dieses Symbol, öffnet sich eine Eingabemaske, in welcher der Nutzer die entsprechenden Informationen für die neue Aufgabe hinterlegen kann. Nach

erfolgreichem Speichern der Aufgabe, erscheint diese in der bereits angesprochenen Übersicht. Neben dem Erstellen von Aufgaben, können diese auch bearbeitet werden. In der rudimentären Version dieser Ausarbeitung, besteht jedoch lediglich die Möglichkeit, die Priorität einer spezifischen Aufgabe zu verändern. Durch einen Klick auf die entsprechende Aufgabe, wechselt die Priorität von *done* zu *in progress* und umgekehrt. Letztlich kann eine Aufgabe durch einen Klick auf das Mülleimer-Symbol unwiderruflich gelöscht werden und verschwindet dadurch sofort aus der Übersicht.

Die hier vorgestellte (Web-)Applikation *Trillo* dient einzig dem Zweck, die im vorherigen Kapitel formulierten Anwendungsfälle zu überprüfen. Bei der Konzipierung steht daher die Funktionalität gegenüber der Benutzerfreundlichkeit im Vordergrund. Es wird außerdem gewährleistet, dass alle CRUD Operationen innerhalb dieses Testszenarios durchgeführt werden können.

## 6.4 Testdurchführung

Die Durchführung der Evaluation umfasst das Testen der definierten Anwendungsfälle (siehe Kapitel 6.2) innerhalb des implementierten Testszenarios (siehe Kapitel 6.3), welches die (Web-)Applikation *Trillo* darstellt. Da die (Web-)Applikation bei *Baqend* gehostet ist, muss ein passender Webbrowser für die Durchführung des Tests ausgewählt werden. Hierzu kommt der Browser *Google Chrome* in der Version 58 zum Einsatz. Der Grund für die Auswahl dieses speziellen Browsers, liegt zum einen in der persönlichen Präferenz aber zum größten Teil, an dem von Browser unterstützen Netzwerk-Tab. Dieser soll während der Testdurchführung, wichtige Informationen bezüglich des Speicherverhaltens der (Web-)Applikation liefern und somit als Monitoring-Werkzeug dienen. Das Wechseln der Netzwerkverbindung während eines Tests, wird ebenfalls über die im Netzwerk-Tab verfügbare Konfiguration vorgenommen.

Zu Beginn jedes Tests wird überprüft, ob keine Datenobjekte im lokalen Datenspeicher zwischengespeichert sind und keinerlei Referenzen im *cache* des Browsers vorgehalten werden. Auch der *Service Worker* wird zu Beginn jedes Tests erneut registriert. Dadurch wird sichergestellt, dass die Testergebnisse der einzelnen Anwendungsfälle, nicht durch vorherige Informationen bzw. Konfigurationen verfälscht werden. Während der Tests werden die in den jeweiligen Anwendungsfällen dokumentierten essenziellen Schritte ausgeführt. Hierbei wird anhand der im Netzwerk-Tab enthaltenen Informationen untersucht, ob die (Web-)Applikation ein erwartetes Speicherverhalten ausweißt. Hierzu werden im speziellen die lokalen Datenbestände während der Testdurchführung beobachtet und eventuelle Abweichungen von dem erwarteten Verhalten festgehalten. Ein Beispiel einer solchen Ansicht kann der Abbildung 23 entnommen werden. In dieser werden die acht Datenobjekte der (Web-)Applikation, welche in der *IndexedDB* des Webbrowsers vorgehalten sind, visualisiert.

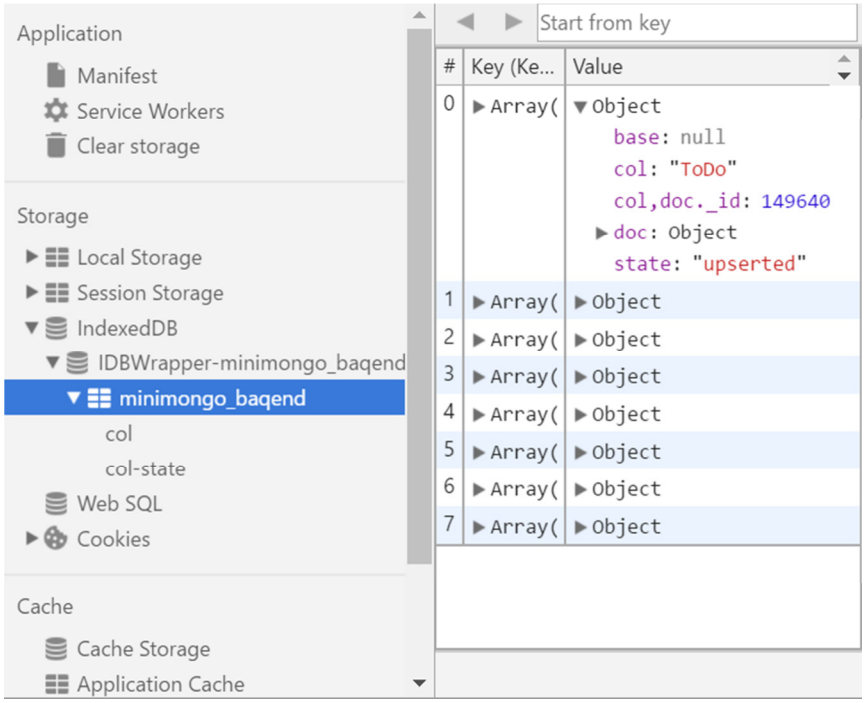



Abb. 23: Netzwerk-Tab zum Monitoring des Speicherverhaltens

Zum Ende eines jeden Tests findet ein Vergleich zwischen dem erwarteten und dem im Testszenario vorhandenen Ergebnis statt. Je nachdem, ob eine Abweichung festgestellt werden kann oder nicht, wird der Anwendungsfall als *erfüllt* oder *nicht erfüllt* eingestuft. Die Ausprägung dieser Einstufung beeinflusst zugleich die Beantwortung der zugehörigen Forschungsfrage. Die Durchführung der Evaluation gilt als abgeschlossen, sobald jegliche Anwendungsfälle eine Untersuchung innerhalb des Testszenarios durchlaufen haben.

## 6.5 Ergebnispräsentation

Nach der gewissenhaften Untersuchung der Anwendungsfälle innerhalb des Testszenarios, werden im Folgenden die erzielten Ergebnisse vorgestellt. Hierfür findet eine separate Bewertung der einzelnen Anwendungsfälle bzw. Forschungsfragen statt. Die erste Forschungsfrage behandelt die persistente Speicherung von Datenobjekten in einem lokalen Datenspeicher und die Möglichkeit der gezielten Anfrageverarbeitung.

*RQ1:* Werden Anfragen im Offline-Modus erwartungsgemäß durch den lokalen Datenspeicher beantwortet?

Für die Bewertung dieser Forschungsfrage spielt unter anderem das Speicherverhalten des lokalen Datenspeichers eine essentielle Rolle. Die persistente Speicherung der Datenobjekte ist nicht nur für die erste, sondern auch für die folgenden Forschungsfragen eine Grundvoraussetzung. Das ausgiebige Testen dieser Funktionalität hat gezeigt, dass das Speicherverhalten den Erwartungen voll entspricht. Neben dem lokalen Speichern von

Datenobjekten, thematisiert die erste Forschungsfrage vor Allem das Verarbeiten von Anfragen im Offline-Modus. Die Untersuchung stützt sich hierzu auf eine bewusste Differenzierung von Anfragen, durch die im Kapitel 4.3 aufgezeigten Filter Operatoren. Durch das hierbei erreichte breite Spektrum an Anfragen konnte beweisen werden, dass die entsprechenden Operationen durch den lokalen Datenspeicher unterstützt werden. Auf Basis dieser Ergebnisse, wird die erste Forschungsfrage als *erfüllt* eingestuft.

*RQ2:* Werden CRUD Operationen im Einzelbenutzersystem korrekt und konsistenzfrei zwischen lokalem und remote Datenspeicher synchronisiert?

Die zweite Forschungsfrage beinhaltet das Ausführen von CRUD Operation in einem Einzelbenutzersystem. Nach einem anschließendem Wechsel in den Online-Modus und der damit verbundenen Synchronisierung mit dem Server, müssen die Datenobjekte konfliktfrei und konsistent vorgehalten sein. Die Untersuchung dieses Anwendungsfalles hat gezeigt, dass bei jeder CRUD Operation der erwartete Zustand, nach erfolgreicher Synchronisierung erreicht ist. Dadurch kann auch die zweite Forschungsfrage als *erfüllt* beantwortet werden.

*RQ3:* Werden CRUD Operationen im Mehrbenutzersystem korrekt und konsistenzfrei zwischen lokalem und remote Datenspeicher synchronisiert?

Wie bereits die vorherige, thematisiert auch diese Forschungsfrage das Ausführen von CRUD Operationen im Offline-Modus. Der zentrale Unterschied liegt in der Anzahl der beteiligten Nutzer. Denn die dritte Forschungsfrage soll das Verhalten in einem Mehrbenutzersystem untersuchen. Die Testdurchführung hat gezeigt, dass bei der Durchführung von INSERT, UPDATE und READ Operationen, keinerlei Konsistenzverluste zwischen verschiedenen Nutzeransichten entstehen. Das Artefakt verhält sich bei dieser Art von Operationen demnach wie erwartet. Differenziert muss dies allerdings bei der Durchführung von DELETE Operationen bewertet werden. Die Untersuchung hat gezeigt, dass diese Art von Operationen nicht wie erwartet synchronisiert wird. Genauer handelt es sich hierbei um den Fall, dass ein spezifisches Datenobjekte in dem lokalen Datenspeicher eines Nutzers A vorhanden ist und dieses anschließend, von einem Nutzer B durch eine DELETE Operation vom Server gelöscht wird. Das entsprechende Datenobjekt wird hierdurch aus dem lokalen Datenspeicher des Nutzer B und vom Server gelöscht. Da dieser Vorgang allerdings nicht im Server referenziert wird, können andere Nutzer nicht nachvollziehen, dass ein Löschen von Datenobjekte stattgefunden hat. Das entsprechende Datenobjekt verbleibt demnach im lokalen Speicher des Nutzers A und verursacht dadurch inkonsistente Datenansichten zwischen Online- (Serverdaten) und Offline-Modus (lokale Daten). Aufgrund dieses nicht erwarteten Ergebnisses, muss die dritte Forschungsfrage als *nicht erfüllt* eingestuft werden.

Zur Lösung des in der dritten Forschungsfrage aufgetretenen Defizites bezüglich DELETE Operationen im Mehrbenutzersystem bieten sich mehrere Implementierungen an. Zum einen könnte die Serverlogik bei DELETE Operationen in der Form angepasst werden, dass nach erfolgreicher Löschung eines Datenobjektes, eine Referenz auf das entsprechende Objekt vorgehalten wird, welche anschließend durch verschiedene Clients zur Konsistenzsicherung der eigenen lokalen Datenbestände genutzt werden kann. Eine weitere, etwas kompliziertere aber auch elegantere Lösung, liegt in der Kombination aus der *TTL* (time to live) des Datenobjektes und dem *Bloomfilter* der Infrastruktur von *Baqend*. Hierbei werden alle Datenobjekte innerhalb des lokalen Datenspeichers vor dem Mechanismus der Synchronisierung mittels einer *Dirty Flag* [IBM17] gekennzeichnet. Während der Synchronisierung wird diese *Dirty Flag* von jedem Objekt entfernt, welches innerhalb des Mechanismus behandelt wird. Dadurch verbleiben nach der Synchronisierung alle Datenobjekte, welche nicht aktualisiert wurden und dadurch potentiell veraltet sind. Von diesen Datenobjekten wird anschließend anhand der Header-Informationen die *TTL* bestimmt. Ist diese abgelaufen wird eine Revalidierung des Datenobjektes angestoßen. Sollte dies nicht der Fall sein, wird die Frische des Datenobjektes im *Bloomfilter* überprüft. Sollte das Datenobjekt nicht mehr frisch sein (*stale*), wird ebenfalls eine Revalidierung angestoßen. Andernfalls gilt das lokale Datenobjekt als frisch (*fresh*). Durch diesen zusätzlichen Mechanismus ist ebenfalls die Konsistenz von DELETE Operationen gegeben.

Die vorliegenden Ergebnisse der Evakuation konnten aufzeigen, dass die definierten Forschungsfragen und damit verbunden, auch die erhobenen Anforderungen an das implementierte Artefakt, zum größten Teil erfüllt sind. Lediglich der Anwendungsfall bezüglich DELETE Operationen im Mehrbenutzersystem, muss durch eine zusätzliche Implementierung erweitert werden.

# 7 Schluss & Ausblick

Durch wachsende Präsenz im Web und der damit verbundenen Variation an Angeboten, steigt die Relevanz von (Web-)Applikationen stetig an. Hierbei konnte aufgezeigt werden, dass die Verfügbarkeit einer Netzwerkverbindung in der Regel die Voraussetzung zur Nutzung dieser Angebote ist. Eine solch konstante und in der Qualität hochwertige Verbindung, kann allerdings in den seltensten Fällen gewährleistet werden. Dies betrifft vor Allem mobile Endgeräte wie Smartphones oder Tablets. Zur Überwindung dieses Hindernisses, kommt immer häufiger das Konzept des Offline-First zum Einsatz. Hierbei werden (Web-)Applikationen derart konzipiert, dass eine fehlende Netzwerkverbindung durch zuvor vorgehaltene Daten bzw. Strukturen kompensiert werden kann. Aus diesem Grund ist es das Ziel dieser Masterarbeit, ein deklaratives Offline-First (DOF) Storage-Konzept zu konzipieren und zu implementieren.

Funktional liegt der Schwerpunkt hierbei zum einen in der persistenten Speicherung von Datenobjekten und der lokalen Verarbeitung von MongoDB-Anfragen auf eben diese Daten. Zum anderen soll der lokale Datenbestand durch Offline-Updates manipuliert werden können und anschließend durch eine geeignete Konfliktauflösungsstrategie mit dem Server zusammengeführt werden. Außerdem ist die Kompatibilität mit der vorhandenen Technologie von *Baqend* eine zentrale Anforderung an das implementierte Artefakt. Nach einer Beschreibung der für diese Ausarbeitung grundlegenden technischen Themen, wurde die von *Baqend* eingesetzte Technologie untersucht und dokumentiert. Darauf folgend fand die Auswahl eines geeigneten lokalen Datenspeichers statt. Hierzu wurden drei mögliche Kandidaten identifiziert und anhand zuvor definierten Kriterien bewertet. Als Ergebnis dieser Bewertung, stützt sich die Implementierung auf *IndexedDB* als lokalen Datenspeicher.

Die anschließende Konzeptionierung, dient der Dokumentation der gedanklichen Struktur des implementierten Artefaktes. Hierbei wurde aus gegebenen Gründen (siehe Kapitel 4.2) zunächst entschieden, die Logik des implementierten Artefaktes nicht im *Service Worker*, sondern im SDK von *Baqend* zu implementieren. Als Wrapper für die erfolgreiche Verarbeitung von MongoDB-Anfragen, kommt *minimongo* zum Einsatz. Hierdurch kann gewährleistet werden, dass fast jede der von *Baqend* unterstützen Filter Operatoren, auch im neu implementierten Offline-Modus zur Verfügung stehen. Ferner wurde der Mechanismus von Offline-Updates genauer spezifiziert. Hierzu werden jegliche im Offline-Modus durchgeführten CRUD Operationen, im lokalen Datenspeicher referenziert und können dadurch, zu einem späteren Zeitpunkt identifiziert werden. Die konzipierte Synchronisierungslogik nutzt diese Referenzen, um durch eine optionale Konfliktauflösung, lokale und remote Datenobjekte miteinander zu vereinen.

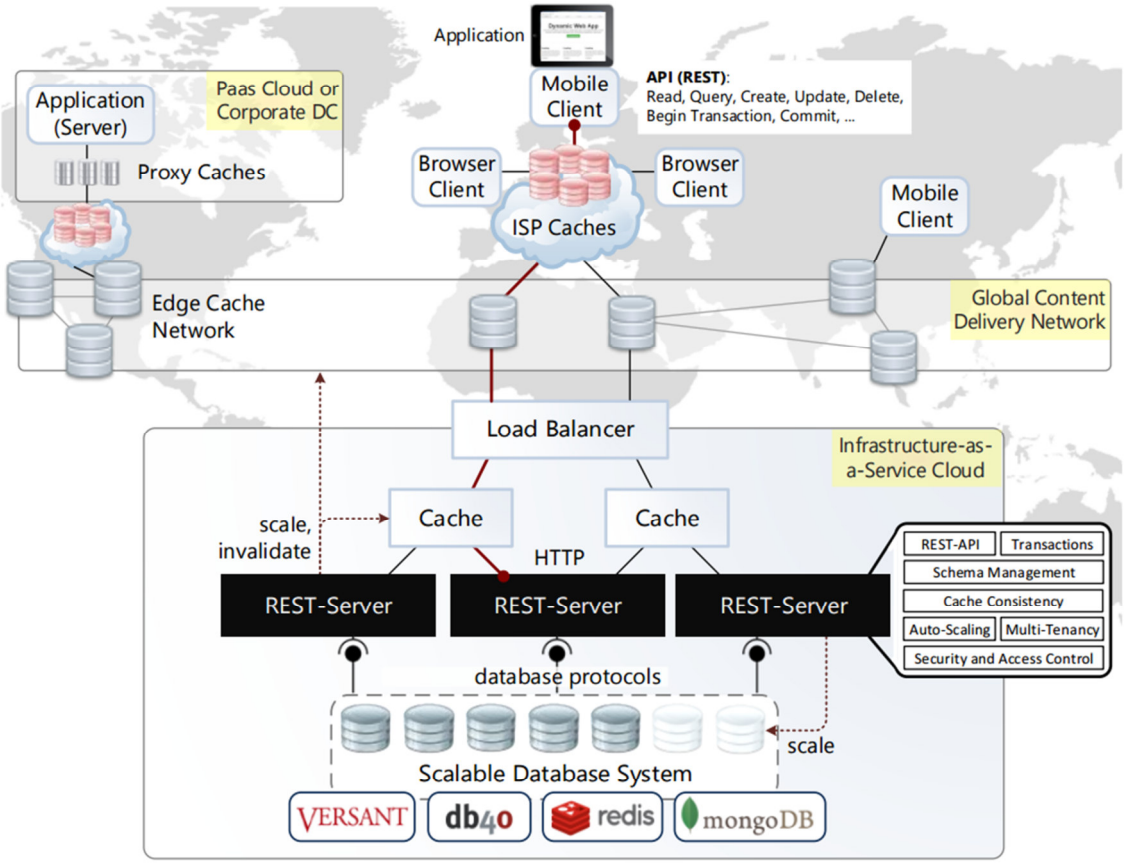
Nach einer ausgiebigen Phase der Implementierung, fand die Evaluation des implementierten Artefaktes statt. Hierbei war es das Ziel, die definierten funktionalen Anforderungen in angemessene Forschungsfragen zu transformieren und diese anschließend, innerhalb einer Fallstudie zu untersuchen. Die hierfür formulierten Anwendungsfälle, wurden daher anhand einer ausschließlich für die Evaluation dieser Masterarbeit implementierten (Web-)Applikation (*trillo*) untersucht. Das Ergebnis dieser Untersuchung zeigt, dass das in dieser Masterarbeit implementierte Artefakt nahezu alle zuvor formulierten Anforderungen erfüllt. Lediglich die im Mehrbenutzersystem durchgeführten DELETE Operationen benötigen die Implementierung zusätzlicher Logik, um das gewünschte Verhalten zu erzielen.

Hinsichtlich der Weiterentwicklung des implementierten Artefaktes, sind neben dem Ausbau von DELETE Operationen im Mehrbenutzersystem, noch weitere Erweiterungen denkbar. Hierbei ist vor allem die Vertiefung der Offline Erkennung anzumerken. Diese könnte wie bereits in Kapitel 4.4 beschrieben, nicht nur die strikte Trennung zwischen online und offline unterstützen, sondern vielmehr auch die Qualität der Netzwerkverbindung beurteilen. Diese Erweiterung ist vor Allem im Bereich der mobilen Endgeräte als sinnvoll zu benennen, da ein inkonstanter Verbindungsstatus hier oftmals vorkommt. Eine weitere Erweiterung könnte in der Auslagerung der Synchronisierungslogik in einen *Service Worker* liegen. Denn laut [Ar15] verfügt der *Service Worker* über einen Mechanismus, welcher es ermöglicht, Logik im Hintergrund auszuführen. Durch diesen *background sync* könnte die Synchronisierung mit dem Server auch dann durchgeführt werden, wenn die (Web-)Applikation gar nicht aktiv durch den Nutzer angesteuert wird. Hierdurch kann die Frequenz der Synchronisierung deutlich erhöht werden, ohne dass diese den Betrieb der (Web-)Applikation hemmt bzw. verlangsamt. Hieraus ergibt sich der Vorteil, dass der Nutzer stetig aktuelle Datenobjekte in seinem lokalen Datenspeicher vorhält, ohne die (Web-)Applikation aktiv anstuern zu müssen. Sollte der *Service Worker* aus Gründen fehlender Kompatibilität mit dem Endgerät nicht zur Verfügung stehen, könnte die bisherige Logik im *Baqend SDK* als Rückhalt (*fallback*) genutzt werden.

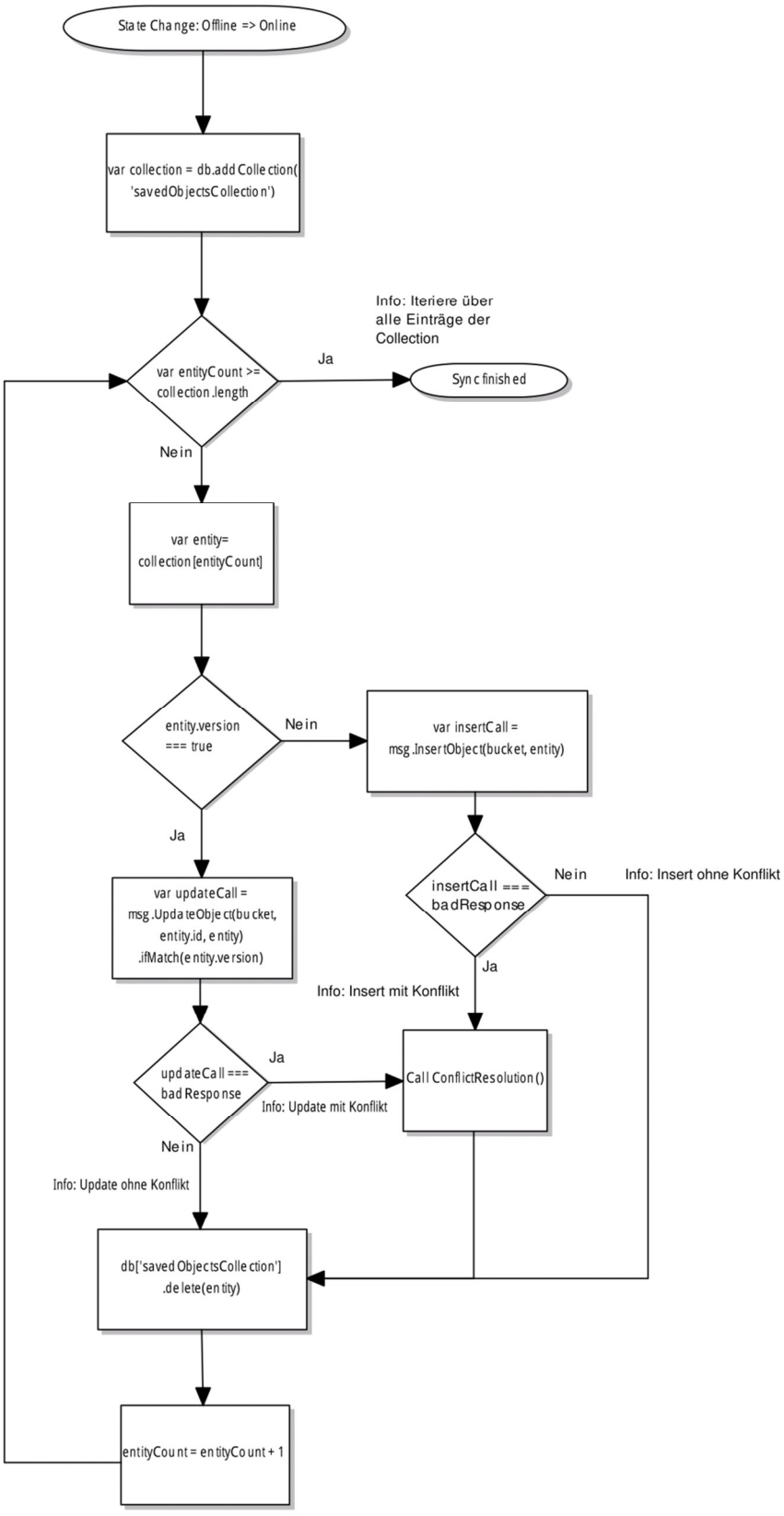
Abgesehen von diesen möglichen Erweiterungen, kann das in dieser Masterarbeit implementierte Artefakt als erfolgreicher Ausbau des *Baqend SDK* angesehen werden. Denn durch diese Erweiterung ist die Möglichkeit gegeben, die vorhandene Technologie auch ohne Verbindung zum Netzwerk zu verwenden. (Web-)Applikationen, welche mit dem erweiterten *Baqend SDK* entwickelt werden, bietet dadurch automatisch einen nahezu vollständigen Offline-First-Ansatz.

# A Anhang

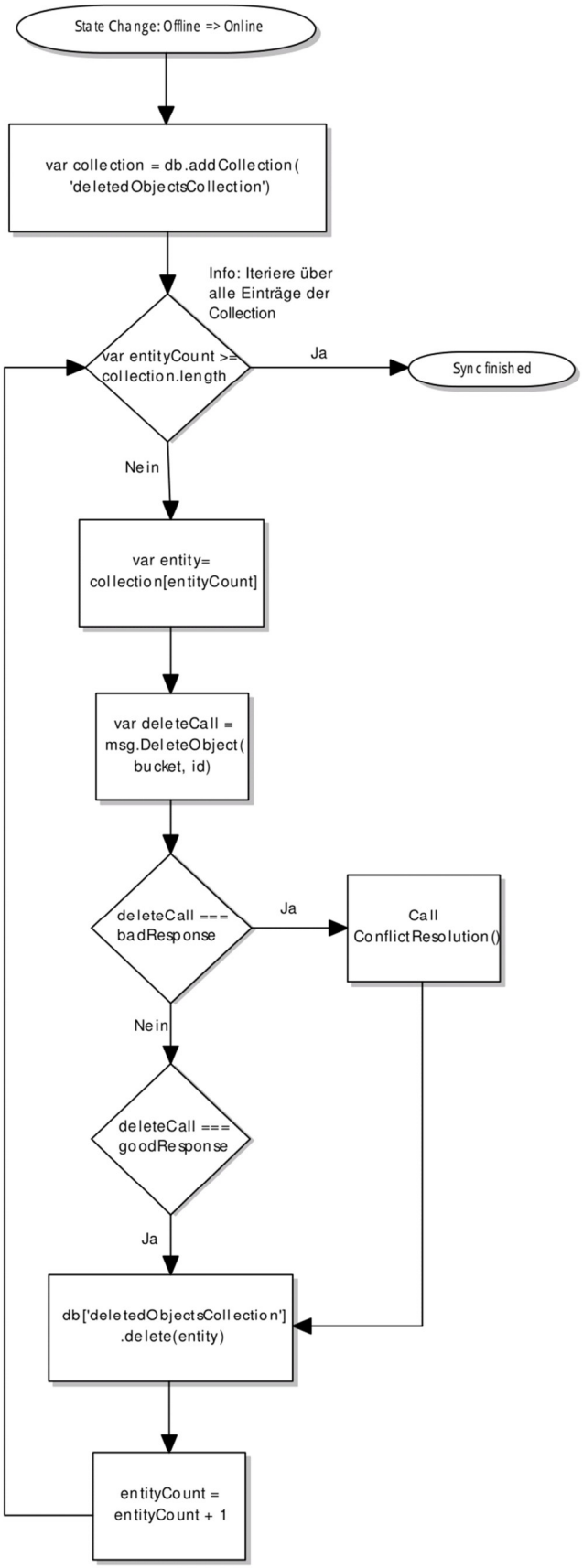
## A.1 Orestes Architektur



## A.2 Flussdiagramm Synchronisierung Offline-INSERT/-UPDATE



## A.3 Flussdiagramm Synchronisierung Offline-DELETE



## A.4 Codeübersicht der Klasse OfflineService

```
'use strict';
const Observable = require('../observable');
const message = require('../message');
const Message = require('../connector/Message');
const StatusCode = Message.StatusCode;

const minimongo = require('minimongo');
const IndexedDb = minimongo.IndexedDb;

let savedObjectsCollection = 'savedObjectsCollection';
let deletedObjectsCollection = 'deletedObjectsCollection';

/**
 * @alias OfflineService
 */
class OfflineService {
    /**
     * @param {EntityManager} entityManager The EntityManager which of this offlineService instance
     */
    constructor(entityManager) {
        this.entityManager = entityManager;
        this.entityManagerFactory = entityManager.entityManagerFactory;

        this.online = new Observable((observer) => {
            if(window) {
                window.addEventListener('load', () => {
                    window.addEventListener('offline', (e) => {
                        observer.next(false);
                    });
                    window.addEventListener('online', (e) => {
                        this.synchronizeData().then(() => {
                            observer.next(true);
                        });
                    });
                });
                observer.next(navigator.onLine);
            }
        });
    }

    /**
     * method to verify whether the app is online or offline
     */
    isOnline() {
        return navigator.onLine;
    }

    /**
     * method to save (insert/update) objects to the local database
     * @param {Object} entity
     * @param {String} entityClass
     * @param {Boolean} offlineSave
     */
    saveDataLocally(entity, entityClass, offlineSave) {
        let promise = new Promise((resolve, reject) => {
            let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
                localDb.addCollection(entityClass, () => {
                    entity._id = entity.id;
                    localDb[entityClass].upsert(entity, () => {
                        resolve();
                    }, (e) => {
                        reject();
                    });
                });
            });

        });

        return promise.then(() => {
            if(offlineSave) {
                return this.saveDataLocally(entity, savedObjectsCollection, false);
            }
        });
    };
```

```
/**
 * method to delete objects of the local database
 * @param {Object} entity
 * @param {String} entityClass
 * @param {Boolean} offlineDelete
 */
deleteLocalEntity(entity, entityClass, offlineDelete) {
    let promise = new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(entityClass, () => {
                localDb[entityClass].remove(entity.id, (res) => {
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });
    });

    return promise.then(() => {
        if(offlineDelete) {
            if(entity.version) {
                return this.saveDataLocally(entity, deletedObjectsCollection, false);
            } else {
                return this.deleteLocalEntity(entity, savedObjectsCollection, false);
            }
        }
    });
}

/**
 * method to load objects from the local database
 * @param {Object} entity
 * @param {String} entityClass
 */
loadLocalEntity(entity, entityClass) {
    let localLoadResult;
    let promise = new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(entityClass, () => {
                localDb[entityClass].findOne({ _id: entity.id }, {}, (result) => {
                    localLoadResult = result;
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });
    });

    return promise.then(() => localLoadResult);
}

/**
 * method to query data of the local database
 * @param {String} query
 * @param {String} sort
 * @param {String} limit
 * @param {String} entityClass
 */
queryLocalData(query, sort, limit, entityClass) {
    let localQueryResult;
    let promise = new Promise((resolve, reject) => {
         let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            limit = limit < 0 ? 0 : limit;
            localDb.addCollection(entityClass, () => {
                localDb[entityClass].find(JSON.parse(query), {sort: JSON.parse(sort)}, {limit: JSON.parse(limit)}).fetch((result) => {
                    localQueryResult = result;
                    resolve();
                }, (e) => {
                    reject();
                });
            });
        });
    });

    return promise.then(() => localQueryResult);
};

synchronizeData() {
    let savedDatePromise = this.synchronizeSavedDate();
    let deletedDataPromise = this.synchronizeDeletedDate();

    return Promise.all([savedDatePromise, deletedDataPromise]);
}
```

```
synchronizeSavedDate() {
    return new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: this.getAppName() }, () => {
            localDb.addCollection(savedObjectsCollection, () => {
                localDb[savedObjectsCollection].find({}).fetch((result) => {
                    if(result) {
                        let mapPromises = result.map((entity) => {
                            let bucket = this.getBucketOfEntity(entity);
                            let key = this.getKeyOfEntity(entity);
                            let msg;
                            delete entity._id;
                            if(entity.version) {
                                msg = new message.ReplaceObject(bucket, key, entity)
                                    .ifMatch(entity.version);
                            } else {
                                msg = new message.CreateObject(bucket, entity);
                            }

                            return this.entityManager.send(msg).then((response) => {
                                return this.saveDataLocally(response.entity, bucket, false).then(() => {
                                    return this.deleteLocalEntity(entity, savedObjectsCollection, false);
                                });
                            }, (e) => {
                                return this.errorHandling(e, entity, 'save').then(() => {
                                    return this.deleteLocalEntity(entity, savedObjectsCollection, false);
                                })
                            });
                        });

                        Promise.all(mapPromises).then(() => resolve());
                    } else {
                        resolve();
                    }
                });
            });
        });
    });
}

synchronizeDeletedDate() {
    return new Promise((resolve, reject) => {
        let localDb = new IndexedDb({ namespace: appName },  () => {
            localDb.addCollection(deletedObjectsCollection, () => {
                localDb[deletedObjectsCollection].find({}).fetch((result) => {
                    if (result) {
                        let mapPromises = result.map((entity) => {
                            let bucket = this.getBucketOfEntity(entity);
                            let key = this.getKeyOfEntity(entity);

                            return this.entityManager.send(new message.DeleteObject(bucket, key).ifMatch(entity.version)).then(() => {
                                return this.deleteLocalEntity(entity, deletedObjectsCollection, false);
                            }, (e) => {
                                return this.errorHandling(e, entity, 'delete').then(() => {
                                    return this.deleteLocalEntity(entity, deletedObjectsCollection, false);
                                });
                            });
                        });

                        Promise.all(mapPromises).then(() => resolve());
                    } else {
                        resolve();
                    }
                });
            });
        });
    });
}

 /**
  * method to verify which object wins when a conflict occurs
  * @param {Object} localObj
  * @param {Object} remoteObj
  */
 conflictResolution(localObj, remoteObj) {
     if(this.entityManagerFactory.conflictResolution) {
         if (typeof this.entityManagerFactory.conflictResolution === "function") {
             return this.entityManagerFactory.conflictResolution(localObj, remoteObj);
         } else if(this.entityManagerFactory.conflictResolution === 'remote') {
             return remoteObj;
         } else {
             return localObj;
         }
     } else {
         return localObj;
     }
 }
```

```
    /**
     * method to analyse and handle errors
     * @param {Error} error
     * @param {Object} entity
     * @param {String} operation
     */
    errorHandling(error, entity, operation) {
        let bucket = this.getBucketOfEntity(entity);
        let key = this.getKeyOfEntity(entity);

        return new Promise((resolve, reject) => {
            if(error.status === StatusCode.OBJECT_OUT_OF_DATE || error.status === StatusCode.CONFLICT) {
                this.entityManager.send(new message.GetObject(bucket, key)).then((response) => {
                    if(this.conflictResolution(entity, response.entity) === response.entity) {
                        this.saveDataLocally(response.entity, bucket, false).then(() => resolve());
                    } else {
                        if(operation === 'save') {
                            delete entity.version;
                            this.entityManager.send(message.ReplaceObject(bucket, key, entity)).then((response) => {
                                this.saveDataLocally(response.entity, bucket, false).then(() => resolve());
                            });
                        } else if(operation === 'delete') {
                            this.entityManager.send(new message.DeleteObject(bucket, key)).then(() => resolve());
                        }
                    }
                }, (e) => {
                    if (e.status === StatusCode.OBJECT_NOT_FOUND && operation === 'save') {
                        if(this.conflictResolution(entity, 'remoteObj') === entity) {
                            this.entityManager.send(new message.CreateObject(bucket, entity)).then(() => resolve());
                        } else {
                            this.deleteLocalEntity(entity, bucket, false).then(() => resolve());
                        }
                    } else if(e.status === StatusCode.OBJECT_NOT_FOUND && operation === 'delete') {
                        resolve();
                    } else {
                        reject();
                    }
                });
            } else {
                reject();
            }
        });
    }


    /**
     * method to get the key of a specific entity
     * @param {Object} entity
     */
    getKeyOfEntity(entity) {
        return entity.id.substr(entity.id.lastIndexOf('/') + 1);
    }

    /**
     * method to get the bucket of a specific entity
     * @param {Object} entity
     */
    getBucketOfEntity(entity) {
        return entity.id.split('/')[2];
    }

    getAppName() {
        return this.entityManager._connector.origin;
    }
}

module.exports = OfflineService;
```

# Literaturverzeichnis


[Al14] Allsopp, J.: *Offline First - HTML5 technologies for a faster, smarter, more engaging web.* Online unter: http://www.webdirections.org/offlineworkshop/ibooksDraft.pdf. 2014

[An17] Angular 2. Online unter: https://angular.io/. Zugriff: 02.06.2017

[Ap17] Apiomat. Online unter: https://apiomat.com/. Zugriff: 03.04.2017

[Ar12] Archibald, J.: *Application Cache is a Douchebag*. Online unter: http://alistapart.com/article/application-cache-is-a-douchebag. 2012

[Ar15] Archibald, J.: *Introducing Background* Sync. Online unter: https://developers.google.com/web/updates/2015/12/background-sync#what_could_i_use_background_sync_for. 2015

[AZ15] ARD und ZDF: *Online-Studie 2015*. Online unter: http://www.ard-zdf-onlinestudie.de/index.php?id=540. 2015

[Ba11] Barth, a.: *HTTP State Management Mechanism*. Online unter: https://tools.ietf.org/html/rfc6265. 2011

[Ba13] Bass, L. et al. Software architecture in practice (3. ed.). Upper Saddle River, NJ [u.a.]: Addison-Wesley/Pearson. 2013

[Ba17] Baqend. Online unter: https://www.baqend.com/. Zugriff: 03.04.2017

[BF17] Bloom filter library. Online unter: https://github.com/Baqend/Orestes-Bloomfilter. Zugriff: 30.01.2017

[Bl70] Bloom, B.: *Space/time trade-offs in hash coding with allowable errors***.** In: Communications of the ACM. Volume 13. 1970

[Bo01] Bourke, T.: *Server Load Balancing*. Beijing [u.a.]: O'Reilly. 2001

[CA17] Cache API. Online unter: https://developer.mozilla.org/de/docs/Web/API/Cache. Zugriff: 20.03.2017

[CCD17] Cache-Control-Direktiven. Online unter: https://developer.mozilla.org/de/docs/Web/HTTP/Headers/Cache-Control. Zugriff: 24.03.2017

[CDB17] CouchDB. Online unter: http://couchdb.apache.org/. Zugriff: 15.01.2017

[Ch15] Chenkie, R.: *Creating Offline-First Web Apps with Service Workers*. Online unter: https://auth0.com/blog/creating-offline-first-web-apps-with-service-workers/. 2015

[CISCO15] CISCO: *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015–2020 White Paper*. Online unter: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html. 2015

[Da12] Daigneau, R.: *Service design patterns: Fundamental design solutions for SOAP/WSDL and RESTful Web services*. Upper Saddle River, NJ [u.a.]: Addison-Wesley. 2012

[De17] Decken, S.: *CacheWorker: Caching Arbitrary Resources without Staleness*. Masterarbeit am Arbeitsbereich VSIS. Universität Hamburg. 2017

[DIN10] DIN SPEC 1041: *Outsourcing technologieorientierter wissensintensiver Dienstleistungen*. 2010

[Dr07] Drepper, U.: *What Every Programmer Should Know About Memory*. Online unter: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf. 2007

[ES17] Es6. Online unter: http://es6-features.org/#Constants. Zugriff: 18.05.2017

[Fe13] Feyerke, A.: *Designing Offline-First Web Apps.* Online unter: http://alistapart.com/article/offline-first. 2013

[Fi16] Firtman, M.: *High Performance Mobile Web: [best Practice for Optimizing Mobile Web Apps]*. O´Reilly. 2016

[Fi17] Firebase. Online unter: https://firebase.google.com/. Zugriff: 03.04.2017

[Ga17] Gaunt, M.: *Service Workers: An Introduction*. Online unter: https://developers.google.com/web/fundamentals/getting-started/primers/service-workers. 2017

[GB13] Gessert, F.; Bücklers: *ORESTES: ein System für horizontal skalierbaren Zugriff auf Cloud-Datenbanken*. Online unter: http://orestes.info/assets/files/Informatiktage.pdf. 2013

[GBR14] Gessert, F.; Bücklers, F.; Ritter, N.: *ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency*. In CloudDB 2014. 2014

[Ge14] Gessert, Felix et al.: *Torwards a Scalable and Unified REST API for Cloud Data Stores*. Online unter: : https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/55. 2014

[Ge15] Gessert, Felix et al.: *The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management*. Online unter: https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/55. 2015

[Go15] Go Y. et al.: *Reliable, Consistent, and Efficient Data Sync for Mobile Apps*. Online unter: https://www.usenix.org/system/files/conference/fast15/fast15-paper-go.pdf. Zugriff: 14.05.2017

[Gr01] Gronau, Norbert: *Industrielle Standardsoftware: Auswahl Und Einführung*. München [u.a.]. Oldenbourg. 2001

[HD14] Heise Developer. *BaaS: Was bedeutet Backend as a Service*. Online unter: https://www.heise.de/developer/artikel/BaaS-Was-bedeutet-Backend-as-a-Service-2412568.html. 2014

[Ho13a] Hoodie: *Say Hello to Offline First*. Online unter: http://hood.ie/blog/say-hello-to-offline-first.html. 2013

[Ho13b] Hoodie: *Offline First and the circle of web*. Online unter: http://hood.ie/blog/offline-first-and-the-circle-of-web.html. 2013

[Ho17] Hoodie Framework. Online unter: http://hood.ie/. Zugriff: 15.01.2017

[Hu99] Huston, G.: *Web Caching*. The Internet Protocol Journal - Volume 2, No. 3. 1999

[IBM17] IBM: *Dirty Flag*. Online unter: https://www.ibm.com/support/knowledgecenter/SSS28S_3.5.0/com.ibm.form.webform.overview.doc/i_wfsws_c_differences_dirty_flag.html. Zugriff: 30.04.2017

[JDO17] Java Data Objects. Online unter: http://www.oracle.com/technetwork/java/index-jsp-135919.html. Zugriff: 30.01.2017

[JPA17] Java Persistence API. Online unter: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. Zugriff: 30.01.2017

[JS17] JavaScript. Online unter: https://www.javascript.com/. Zugriff: 18.05.2017

[Ki17] Kinvey. Online unter: https://www.kinvey.com/. Zugriff: 03.04.2017

[La12] Lambert, J.: *Offline First – A better HTML5 User Experience*. Online unter: http://www.joelambert.co.uk/article/offline-first-a-better-html5-user-experience. 2012

[LF17] localForage. Online unter: https://github.com/localForage/localForage. Zugriff: 15.01.2017

[MAM16] Marketsandmarkets: *Cloud/Mobile Backend As A Service (BAAS) Market by Service Type - Global Forecast to 2020*. Online unter: http://www.marketsandmarkets.com/Market-Reports/mobile-backend-as-a-service-mbaas-market-813.html. 2016

[MC17] Meteor Minimongo Colloections. Online unter: http://docs.meteor.com/api/collections.html#Mongo-Collection. Zugriff: 23.04.2017

[MDB17] MongoDB. Online unter: https://www.mongodb.com/de. Zugriff: 22.04.2017

[Me17] Meteor Minimongo. Online unter: https://guide.meteor.com/collections.html. Zugriff: 23.04.2017

[MM17] Minimongo. Online unter: https://github.com/mWater/minimongo. Zugriff: 23.04.2017

[NH02] Neumann, F.; Häussler, H.: *Declarative data merging with conflict resolution*. In: Proceedings of the International Conference on Information Quality (IQ), Cambridge. 2002

[NS17] NoSQL Databases. Online unter: http://nosql-database.org/. Zugriff: 12.02.2017

[OJS17] Offline.js. Online unter: http://github.hubspot.com/offline/docs/welcome/. Zugriff: 05.05.2017

[OLD17] Oxford Living Dictionaries: *Cache*. Online unter: https://en.oxforddictionaries.com/definition/cache. Zugriff: 20.03.2017

[OOE17] Online- und Offline-Events. Online unter: https://developer.mozilla.org/de/docs/Online_and_offline_events. Zugriff: 05.05.2017

[Or17] Orestes API. Online unter: http://orestes.info. Zugriff: 29.01.2017

[Ös10] Österle, H.: *Gestaltungsorientierte Wirtschaftsinformatik: Ein Plädoyer für Rigor und Relevanz*. Deutschland infowerk ag. 2010

[OV17] Orestes VSIS. Online unter: https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/55. Zugriff: 03.02.2017

[PDB17] PouchDB. Online unter: https://pouchdb.com/. Zugriff: 15.01.2017

[Pr17] Promise. Online unter: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise. Zugriff: 19.05.2017

[RFC2616] Hypertext Transfer Protocol -- HTTP/1.1. Online unter: https://www.ietf.org/rfc/rfc2616.txt. 1999

[RH09] Runeson, P.; Höst, M.: *Guidelines for conducting and reporting case study research in software engineering*. In: Empirical Software Engineering. Vol. 14, Issue 2. Springer Netherlands 2009

[RH17] Research Hub: *Browser Cache vs. HTML5 Application Cache*. Online unter: http://researchhubs.com/post/computing/web-application/browser-cache-vs-html5-application-cache.html. 2017

[Ro10] Rouget, P.: *offline web applications*. Online unter: https://hacks.mozilla.org/2010/01/offline-web-applications/. 2010

[RxJS17] RxJS: *Observable Objects*. Online unter: http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html. Zugriff: 12.05.2017

[SGR15] Schaarschmidt, M.; Gessert, F.; Ritter, N.: *Towards Automated Polyglot Persistence*. Online unter: https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/55. 2015

[SP17] Silicon Press: *Web Caching Technology Brief*. Online unter: http://www.silicon-press.com/briefs/brief.webcaching/brief.pdf. 2017

[SQL17] SQLite. Online unter: http://www.sqlite.org/. Zugriff: 10.02.2017

[SS12] Schill, A.; Springer, T.: *Verteielte Systeme: Grundlagen und Basistechnologien* (2. Aufl.). Berlin [u.a.]. Springer Verlag. 2012

[Te16] Teixeria, P.: *Build More Reliable Web Apps with Offline-First Principles*. Online unter: https://thenewstack.io/build-better-customer-experience-applications-using-offline-first-principles/. 2016

[Tech15] Techopedia: *Backend-as-a-Service (BaaS)*. Online unter: https://www.techopedia.com/definition/29428/backend-as-a-service-baas. 2015

[Tr17] Trillo App. Online unter: https://trillo.app.baqend.com/#/. Zugriff: 01.06.2017

[W3C10] W3C: *Web SQL Database*. Online unter: https://www.w3.org/TR/webdatabase/. 2010

[W3C15] W3C: *Service Worker*. Online unter: https://www.w3.org/TR/service-workers/ 2015

[W3C16] W3C: *Web Storage (Second Edition)*. Online unter: https://www.w3.org/TR/webstorage/#refsRFC2119. 2016

[W3C17] W3C: *Indexed Database API 2.0*. Online unter: https://w3c.github.io/IndexedDB. 2017

[We01] Wessels, D. *Web Caching*. 1. ed. Beijing [u.a.]: O'Reilly. 2001

# Abbildungsverzeichnis



# Tabellenverzeichnis

# Glossar

| | |
|---|---|
| ACL | Eine ACL (Access Control List) legt fest, in welchem Umfang einzelne Benutzer und Systemprozesse Zugriff auf bestimmte Objekte haben. |
| API | APIs (Application Programming Interface) dienen in der Informatik der vereinheitlichten und strukturierten Datenübergabe zwischen Programmen und Programmteilen. |
| Backend | Als Backend wird der Teil eines IT-Systems bezeichnet, der sich mit der Datenverarbeitung im Hintergrund beschäftigt – der Data Layer. |
| Client | Unter einem Client wird Soft- oder Hardware verstanden, die bestimmte Dienste von einem Server in Anspruch nehmen kann. |
| Cloud | Cloud Computing beschreibt die Bereitstellung von IT-Infrastruktur und IT-Leistungen wie beispielsweise Speicherplatz, Rechenleistung oder Anwendungssoftware als Service über das Internet. |
| CRUD | Das Akronym CRUD umfasst die grundlegenden Datenbankoperationen Create, Read oder Retrieve, Update und Delete oder Destroy. |
| DOM | Ein Browser analysiert ein HTML-Dokument und erstellt daraus im Arbeitsspeicher das Document Object Model, also eine Repräsentation dieses Dokuments, auf die man beispielsweise mit JavaScript zugreifen kann. |
| Framework | Im Software-Engineering ist ein Framework ein modernes Rahmenwerk, das dem Programmierer den Entwicklungsrahmen für seine Anwendungsprogrammierung zur Verfügung stellt und damit die Software-Architektur der Anwendungsprogramme bestimmt. |
| Frontend | Bei einem IT-System bezeichnet das Frontend die Presentation Layer, also den Teil eines IT-Systems, der näher am Anwender ist. Es ist das Gegenteil des Backend. |
| JSON | JavaScript Object Notation (JSON) ist ein textbasiertes, von Menschen lesbares Datenaustauschformat, das für die Darstellung einfacher Datenstrukturen und Objekte in browserbasiertem Code verwendet wird. |
| Latenz | Latenz meint im übertragenen Sinne die Zeit zwischen einem Reiz und der daraus folgenden Reaktion. In der Technik wird häufig auch das Synonym Signallaufzeit verwandt. |
| Middleware | Mit Middleware ist eine zusätzliche Schicht in einer komplexeren Software-Struktur gemeint, deren Aufgabe es ist, die Zugriffsmechanismen auf unterhalb angeordnete Schichten zu vereinfachen und die Details deren Infrastruktur nach außen hin zu verbergen. |

| | |
|---|---|
| MongoDB | MongoDB ist eine Open-Source-Datenbank, die ein dokumentenorientiertes Datenmodell verwendet. |
| Proxy | Ein Proxy ist eine Kommunikationsschnittstelle in einem Netzwerk. Er arbeitet als Vermittler, der auf der einen Seite Anfragen entgegennimmt, um dann über seine eigene Adresse eine Verbindung zur anderen Seite herzustellen. |
| Request | Request (Anfrage) ist eine Anfrage des Clients an den Server, welcher nach der Verarbeitung des Requests mit einer Response (Antwort) reagiert. |
| Response | Antwort eines Servers auf eine Anforderung eines Clients. |
| Server | Ein Server ist ein Computerprogramm oder ein Computer, der Computerfunktionalitäten wie Dienstprogramme, Daten oder andere Ressourcen bereitstellt, damit andere Computer oder Programme („Clients“) darauf zugreifen können, meist über ein Netzwerk. |
| User Experience | Unter User Experience (UX), das sich als Nutzungserfahrung oder Nutzungserlebnis übersetzen lässt, versteht man die erlebte und gefühlte Qualität der Interaktion eines Nutzers insbesondere im Bereich digitaler Medien. |
| (Web-)Applikation | Eine Webanwendung ist ein Anwendungsprogramm nach dem Client-Server-Modell. Anders als klassische Desktopanwendungen werden Webanwendungen also nicht lokal auf dem Rechner des Benutzers installiert und dort ausgeführt. |

# Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.
Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit einer Einstellung meiner Abschlussarbeit in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 20.06.2017

Kevin Twesten