Datensynchronisation und Offline-Nutzung für webbasierte mobile Informationssysteme

Diplomarbeit

Ausgeführt zum Zweck der Erlangung des akademischen Grades
Dipl.-Ing. für technisch-wissenschaftliche Berufe

am Masterstudiengang Digitale Medientechnologien an der Fachhochschule St. Pölten, **Vertiefungsrichtung Mobiles Internet**

von
Kerstin Blumenstein, BSc
dm101504

Erstbegutachter/in und Betreuer: FH-Prof. DI Dr. Grischa Schmiedl
Zweitbegutachter: FH-Prof. DI Markus Seidl

St. Pölten, 01.06.2012

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der vom Begutachter bzw. der Begutachterin beurteilten Arbeit überein.

Ort, Datum	Unterschrift

Kurzfassung

Der Zugriff auf das Internet über mobile Endgeräte hat im letzten Jahrzehnt enorm an Bedeutung gewonnen. In Österreich nutzten 2011 37% der Bevölkerung das Internet über mobile Geräte (vgl. WUNDERMANN PXP GmbH, 2011) – Tendenz steigend. Daraus lässt sich ein steigendes Interesse an Anwendungen für mobile Endgeräte ableiten.

Betrachtet man die Nutzungsorte von Smartphones, wird deutlich, dass ein mobiles Gerät immer und überall dabei ist und auch genutzt wird – ob zu Hause oder unterwegs mit öffentlichen Verkehrsmitteln. Gerade dort – auf dem Weg quer durchs Land oder in Gebäuden – bekommt der mobile Internetuser jedoch immer wieder Probleme mit der Netzwerkverbindung. Eine flächendeckende Netzabdeckung für jeg-liche mobile Netzwerkverbindung wird wohl für immer utopisch bleiben. Das Surfen im Internet ist bspw. bei einer langen Autofahrt somit nicht durchgängig möglich. Dieses Problem unterstreicht den Wunsch nach (Web-) Anwendungen, die auch ohne Internetverbindung genutzt werden können. Während dies bei nativen Applikationen üblich ist, sind Webanwendungen dieser Art Raritäten. Jedoch ist die Umsetzung und Anwendung solch einer Applikation erst mit der Entwicklung entsprechender JavaScript-APIs im Umfeld von HTML 5 realisierbar.

Die vorliegende Arbeit untersucht die Problematik der Datensynchronisation und Offline-Nutzung für webbasierte mobile Informationssysteme. Zwei Untersuchungsszenarien sind in diesem Zusammenhang zu unterscheiden:

1. Die Nutzbarmachung mobiler Webapplikationen trotz fehlender bzw. unzuverlässiger Internetverbindung
2. Die Verbesserung der Antwortzeiten (vor allem bei schlechten Verbindungen)

Bezugnehmend auf diese Szenarien werden Lösungsansätze anhand relevanter Design Pattern und Methoden erarbeitet sowie technologische Hilfsmittel betrachtet, die bei der Offline-Datenhaltung und Datensynchronisation unterstützen.

Basierend auf Erkenntnissen aus der Implementierung einer Anwendung, welche die Anforderungen für Szenario 1 erfüllt, werden Empfehlungen abgegeben, die EntwicklerInnen bei der erfolgreichen Umsetzung einer offline-fähigen Webanwendung unterstützen.

Abstract

In the past decade, accessing the Internet via mobile devices has grown in importance. In Austria in 2011, 37% of all Austrians used the Internet via mobile devices (WUNDERMANN PXP GmbH, 2011) – the highest percentage so far. These numbers show that there is a great potential for applications for mobile devices.

Smartphones are used at all times and everywhere - whether at home or on public transport. When travelling in the countryside or in buildings users of mobile Internet encounter problems with network connections. There is no comprehensive coverage for any mobile network connection. Therefore Internet access, for example during a long car ride, is not consistently available. This problem emphasises the need for (web) applications that can be used without an Internet connection. While this is common for native applications, web applications of this type are rare. However, such applications require the development of appropriate HTML 5 compatible JavaScript APIs.

This thesis explores the issues of data synchronisation and offline usage of web based mobile information systems. Two problem domains are distinguished from each other in this context:

1. The activation of mobile web applications in spite of no or disaffected Internet access
2. The improvement of response times (especially without consistent network connections)

Possible solutions are explained based on design patterns and methods as well as considering tools assisting in offline data management and data synchronisation.

Based on findings from the implementation of an application that meets the requirements for problem domain 1, recommendations to developers are given for building an offline-enabled web application.

Inhaltsverzeichnis

1. Einleitung

„Der mobile Zugang zum Web macht noch keine Revolution aus, es sind die Funktionen, die über die reine Webnutzung hinausgehen, die eine Revolution ausmachen können.“

(Alby, 2008, p. 47)

Die Popularität des mobilen Internets ist im letzten Jahrzehnt rasant gestiegen. In Österreich hat diese Entwicklung im Dezember 1999 begonnen, als WAP[1] durch die Mobilkom Austria eingeführt wurde (vgl. mobilkom Austria, 2000). Allerdings kann man erst mit dem Markteinstieg des Apple iPhone[2] von einer Akzeptanz der Nutzung des Internets über mobile Geräte sprechen. Ausschlaggebend war zum einen das bereits 2006 eingeführte HSDPA[3], welches höhere Datenübertragungsraten zuließ (vgl. mobilkom Austria, 2006), und zum anderen das Gerät selbst. Nur drei Jahre später – im Jahr 2011 – nutzten bereits 37% der Österreicher das mobile Internet. (vgl. WUNDERMANN PXP GmbH, 2011, p. 12). Zahlen aus dem Nachbarland Deutschland[4] verdeutlichen diese Entwicklung. Während 2006 nur 6% der Bevölkerung das Internet mobil nutzten, waren es 2010 bereits 28,4%. Einer Prognose von PwC – der in Deutschland führenden Wirtschaftsprüfungs- und Beratungsgesellschaft – zur Folge werden im Jahr 2015 mehr als die Hälfte (rund 54%) der Bevölkerung das Internet über mobile Geräte nutzen. (vgl. Hermann et al., 2011; pdwb, n.d.; PwC, 2010; PwC & Wilkofsky Gruen Associates, 2011)

Diese Zahlen zeigen das Potential von Anwendungen für mobile Endgeräte. Jedoch sollten die Nutzungsorte von mobilen Geräten beachtet werden. Eine Studie von Kaikkonen (2008, p. 6) zeigt, dass europäische NutzerInnen das mobile Internet in den Anfängen der Nutzung über mobile Geräte vorwiegend zu Hause einsetzten. Unterwegs, während des Fahrens mit öffentlichen Verkehrsmitteln oder während des Wartens auf diese sowie auf Arbeit oder im Büro wurden als weitere relevante Nutzungssituationen angeführt. Die selben Orte bilden auch 2012 die Top-4 der beliebtesten Orte zur Nutzung von Smartphones (vgl. Ipsos & MMA, 2012). Dabei zeigt sich deutlich, dass der Smartphone-User immer und überall erreichbar sein möchte – ob zu Hause im Gebäude oder im Zug unterwegs quer durchs Land.

1.1. Problemstellung

Jane et al. (2009) resümieren im Jahr 2009, dass die mobile Umgebung mit beschränkter Bandbreite und beschränkter Gerätestärke (Verarbeitungsgeschwindigkeit, Speicherkapazität, Arbeitsspeicher) sowie lückenhafter Konnektivität einher geht. Die kabellose Kommunikation mit Handheld-Computern birgt jedoch den Vorteil, immer und überall auf alle denkbaren Informationen zugreifen zu können. Verglichen mit kabelgebundenen Netzwerken ist die Performance einer kabellosen Verbindung derzeit jedoch deutlich niedriger.

1 Wireless Application Protocol (kurz WAP)

2 Das erste Modell des iPhone ist seit März 2008 auf dem österreichischen Markt erhältlich.

3 High Speed Downlink Packet Access (kurz HSDPA)

4 Aus Österreich liegt ein Vergleich dieser Art nicht vor.

Auch 2011 geben noch 15% der Befragten eine zu langsame Internetverbindung als Grund an, warum sie das Internet am Mobiltelefon nicht nutzen (vgl. Accenture, 2011, p. 33).

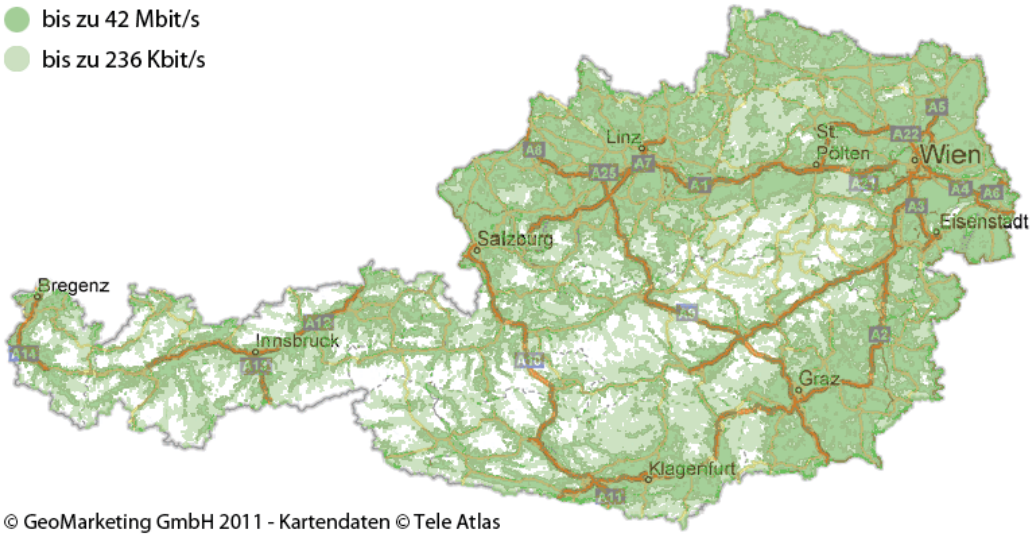

Abbildung 1: Netzabdeckung der A1 Telekom Austria AG, Stand April 2012 (Quelle: A1 Telekom Austria AG, 2012)

Die Netzabdeckung der A1 Telekom Austria AG (Abbildung 1) zeigt, dass die Nutzung von HSDPA (bis zu 42 Mbit/s, Dunkelgrün) nicht flächendeckend möglich ist. Die Großräume Wien, St. Pölten und Eisenstadt sowie Salzburg, Linz, Klagenfurt und Graz sind sehr gut ausgestattet. Es sind jedoch auch sehr viele hellgrüne und weiße Bereiche zu sehen, in denen nur eine langsame (bis zu 236 Kbit/s) oder gar keine Internetverbindung zur Verfügung steht.

Tatsächlich zeigt sich meist, dass in den Ballungsräumen eine sehr gute Netzabdeckung vorherrscht. Wohnt man am Land oder fährt mit dem Auto quer durchs Land, kommt es häufig zu Empfangsproblemen. HSDPA steht selten durchgängig zur Verfügung. Das Gleiche gilt, wenn man im Wald oder innerhalb von Gebäuden unterwegs ist, bspw. in den Katakomben einer Kirche: Es ist vorbei mit der schnellen mobilen Internetverbindung.

Die Verbindung bricht ohne Vorwarnung ab und das Surfen im Internet oder Nutzen einer Anwendung, die auf eine Internetverbindung angewiesen ist, ist nicht mehr möglich. Ruft ein User in diesem Moment Informationen ab, bleibt diese Anfrage erfolglos. In solchen Fällen wünscht man sich, (Web-) Anwendungen auch ohne Internetverbindung nutzen zu können. Bisher ist man dies nur von nativen Applikationen gewohnt.

Um die User Experience in solchen Fällen zu verbessern, ermöglichen die zukünftige HTML-Spezifikation HTML 5 durch den Application Cache und das im HTML 5-Umfeld entstehende Web Storage API sowie das API für lokale Datenbanken (Web SQL und Indexed Database API) auch für Webanwendungen eine Offline-Nutzung. Ob und wie diese Erweiterungen in realen Szenarien mit beschränkter Konnektivität tatsächlich sinnvoll genutzt werden können, muss jedoch erst gezeigt werden.

1.2. Gegenstand und Ziel der Arbeit

Die vorliegende Diplomarbeit soll EntwicklerInnen mit grundlegenden Kenntnissen in der Webentwicklung die Problematik der Datensynchronisation und Offline-Nutzung für mobile Informationssysteme im Webbereich näher bringen.

Die allgemeine Aufgabe eines Informationssystems ist, die richtige Information zur richtigen Zeit am richtigen Platz zur Verfügung zu stellen. (vgl. Stahlknecht & Hasenkamp, 2002, p. 397) In diesem Zusammenhang stehen für mobile Informationssysteme vier Schlagwörter: anything, anytime, anywhere und anyhow. Mobile Informationssysteme geben den BenutzerInnen die Möglichkeit, jegliche Art von Information, Produkt oder Serviceleistung zu jedem Zeitpunkt an jedem beliebigen Ort dieser Welt in einer adäquaten Form individuell zugeschnitten abzurufen. (vgl. Dengler, Henseler & Zimmermann, 2001, p. 385)

Die Ausgangssituation eines mobilen Informationssystems stellt sich wie folgt dar: Auf einem Server befinden sich viele Daten, die in einer Applikation potentiell verwendet und entweder komplett oder in einem Sub-Set auf dem Client gehalten werden sollen. Ziel ist es, den Datenzugriff der Applikation während der Nutzung in schlecht oder nicht versorgten Gebieten zu minimieren oder gar ganz zu vermeiden und somit Verzögerungen bei der Bedienung zu reduzieren. In gut versorgten Gebieten kann hingegen automatisch oder auf Anfrage eine Aktualisierung der gecachten Daten durchgeführt werden.

Behandelt werden Applikationen, die aktuelle Daten nutzen wollen und daher eigentlich auf eine funktionierende Datenverbindung angewiesen sind. Zwei Nutzungsszenarien sind in diesem Zusammenhang relevant und werden im Folgenden durch die Definition von Personas beschrieben:

1. Kein bzw. nur kurzzeitiger Internetzugriff vorhanden

Person A wohnt und arbeitet als Informatik-LehrerIn in Österreich. Die Person fährt zur Weiterbildung auf eine Pädagogen-Konferenz nach Deutschland. Für diese Konferenz steht eine mobile Webapplikation zur Verfügung, in der das gesamte Konferenzprogramm abgebildet ist. Person A möchte diese Anwendung auf ihrem Smartphone auch auf der Konferenz nutzen, sie weiß allerdings nicht, wann und in

welchem Umfang sie in Deutschland Internetzugriff haben wird. Sie ist deshalb darauf angewiesen, alle benötigten Informationen bereits in Österreich auf das Smartphone zu laden.

Die Applikation muss beim ersten Aufruf alle benötigten Ressourcen und Daten auf dem Smartphone speichern. Es ist davon auszugehen, dass während des Auslandaufenthaltes lediglich Updates möglich sind, wenn Person A z.B. im Hotel die Möglichkeit hat, WLAN zu nutzen. Der Großteil der Interaktion mit der Applikation findet ohne Netzwerkverbindung statt.

2. Antwortzeiten verbessern

Person B ist oft im Zug unterwegs. Während der Fahrt surft sie gerne und viel mit ihrem Smartphone. Jedoch steht ihr nicht immer die beste Verbindung zur Verfügung. Sie schätzt es, wenn die Antwortzeiten von Webapplikation dennoch kurz sind.

Ziel einer Applikation für dieses Szenario sollte es sein, Inhaltsdaten vorzuladen, um die Antwortzeiten zu verringern. Mögliche Eingrenzungen der Daten anhand des Kontextes, in dem sich Person B befindet, sind in diesem Fall von Vorteil. Dies könnten z.B. die GPS-Position, Datum und Uhrzeit oder personenbezogene Daten sein.

Im Rahmen dieser Arbeit werden folgende Fragestellungen detailliert untersucht:

Welche Szenarien und Möglichkeiten gibt es, Applikationen durch Caching-Techniken auch bei langsamen Verbindungen oder komplett ohne Netzwerkverbindung nutzen zu können?

Wie kann eine Webanwendung nach Szenario 1 umgesetzt werden?

Sind die Möglichkeiten, die HTML 5 bietet, geeignet, um die Anforderungen zu erfüllen?

Welche Probleme können bei der Umsetzung von webbasierten Offline-Anwendungen auftreten und warum?

1.3. Aufbau der Arbeit

Der Aufbau dieser Arbeit ist in Abbildung 2 dargestellt.

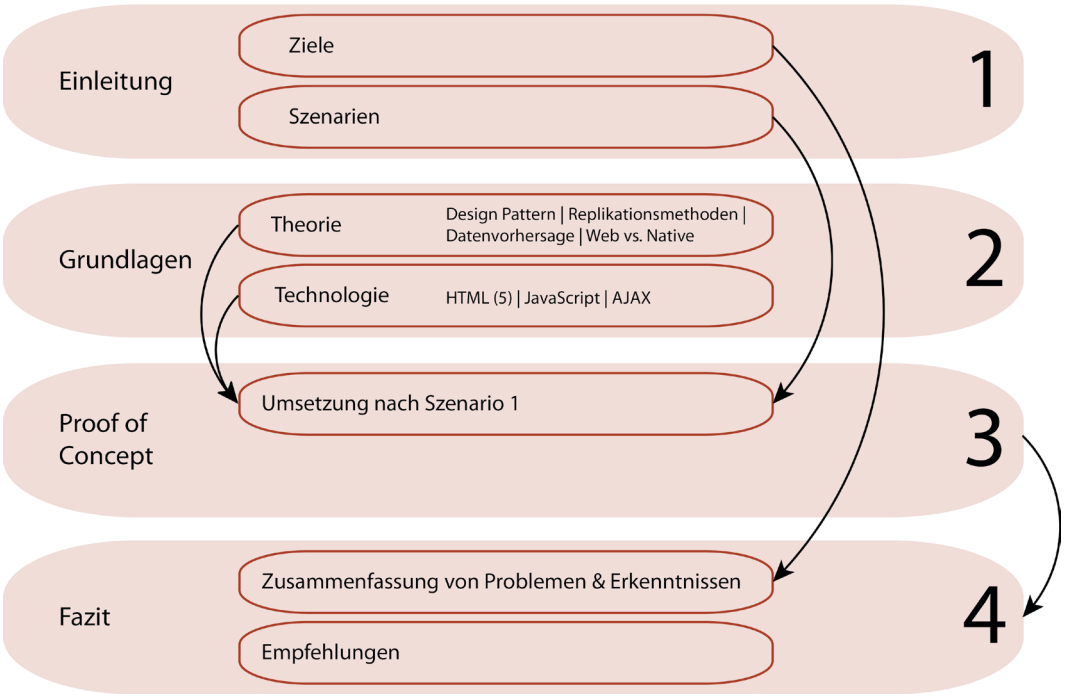

Abbildung 2: Aufbau der Arbeit

Kapitel 2 behandelt die Grundlagen zur Datensynchronisation für mobile Informationssysteme. Durch eine Literaturrecherche werden Lösungsansätze anhand relevanter Design Pattern und Methoden erarbeitet sowie technologische Hilfsmittel betrachtet, die bei der Offline-Datenhaltung und Datensynchronisation unterstützen.

Anschließend werden in Kapitel 3 anhand einer Beispielanwendung im Sinne eines Proof of Concepts die zur Verfügung stehenden Technologien auf deren Praxistauglichkeit mit aktuellen Smartphones getestet. Dabei konzentriert sich der Proof of Concept auf die Umsetzung einer Webanwendung nach Szenario 1 (vgl. Kapitel 1.2).

In Kapitel 4 erfolgt die Beantwortung der in Kapitel 1.2 aufgestellten Fragen. Zum Abschluss werden die gewonnen Erkenntnisse und Probleme in Empfehlungen zusammengefasst.

1.4. Definitionen

In der Arbeit wird zwischen Local Cache of Documents und Local Cache of Data unterschieden.

Der **Local Cache of Documents** beinhaltet alle zur Darstellung der Applikation benötigten Ressourcen (bspw. HTML-, JavaScript-, CSS- und Bilddateien). Dieser Cache muss nur lesbar sein.

Im **Local Cache of Data** werden alle Daten einer Applikation gehalten. Relevant sind hier alle Daten, die zum Betreiben der Anwendung notwendig sind. Der Local Cache of Data muss sowohl gelesen als auch geschrieben werden können.

2. Grundlagen zur Datensynchronisation für mobile Informationssysteme

Das zweite Kapitel behandelt den theoretischen und technologischen Teil. Begriffe wie Synchronisation und Replikation werden geklärt. Außerdem wird die Recherche zu relevanten Ansätzen im Umfeld der Datensynchronisation und Design Pattern dokumentiert. Zudem werden Strategien zur Datenvorhersage und Standards wie HTML und JavaScript sowie relevante APIs im Umfeld von HTML 5 aufgezeigt, die für das Thema Datensynchronisation in Webapplikationen von Nutzen sind.

2.1. Datenreplikation und –synchronisation

2.1.1. Datenreplikation

Die Erzeugung von redundanten Datenobjekten wird Replikation genannt. Es können dabei beliebig viele Datenobjekte an unterschiedlichen Orten erstellt werden. Die Kopie des Datenbestandes trägt dabei den gleichen Informationsgehalt wie das Original. Replizierte Datenobjekte werden als Replikate bezeichnet. Bei einer Replikation werden sowohl Original als auch Kopie als Datenbasis verwendet. Im Gegensatz dazu steht die Migration, bei welcher das Original gelöscht wird. (vgl. Ameling, 2009, p. 7f)

Ameling unterscheidet in (2009, p. 8f) zwischen partieller und vollständiger Replikation. Bei der vollständigen Replikation werden alle Daten repliziert. Dies kann bspw. die gesamte Datenbank sein. Partielle Replikation bedeutet, dass nur eine Teil- bzw. Untermenge der Daten repliziert werden.

Sowohl für den Local Cache of Documents als auch für den Local Cache of Data (vgl. Kapitel 1.4) sind vollständige und partielle Replikationen denkbar. Werden bspw. nur die Folgeseiten in einer bestimmten Linktiefe vorgeladen, kann eine partielle Replikation des Local Cache of Documents zum Einsatz kommen. Wird eine Applikation zur vollständigen Offline-Nutzung wie im Proof of Concept (siehe Kapitel 3) umgesetzt, so ist eine vollständige Replikation zielführend.

Im Bereich der verteilten Datenbanksysteme (vgl. Kapitel 2.2.5) wird Replikation eingesetzt, um die Verfügbarkeit zu erhöhen. Außerdem soll eine Steigerung der Zuverlässigkeit und ein schneller lokaler Zugriff auf das Datenbanksystem erreicht werden. (vgl. Ozsu & Valduriez, 2011)

Die Datenreplikation für mobile Informationssysteme verfolgt ähnliche Ansätze. Ziel ist es zum einen das Antwortzeitverhalten zu verbessern und somit einen schnelleren lokalen Zugriff zu erreichen, zum anderen aber auch die Offline-Verfügbarkeit zu gewährleisten und damit die Verfügbarkeit zu erhöhen.

2.1.2. Datensynchronisation

Der Vorgang des Abgleichs oder der Aktualisierung von Datenobjekten wird als Synchronisation bezeichnet. (vgl. Lehner, 2003, p. 183) Die Synchronisation erfolgt immer von einem Rechner zu einem oder mehreren anderen. Es gibt einen Rechner, der die Kontrolle der Synchronisation übernimmt, eine entsprechende Synchronisationsnachricht erstellt und diese Nachricht an alle Rechner versendet, auf denen sich Replikate der geänderten Daten befinden (siehe Abbildung 3). Dieser Rechner wird als Sender (Master) bezeichnet. Dementsprechend erfolgt eine Synchronisation immer von einem Sender zu einem oder mehreren Empfängern (Slave). (vgl. Ameling, 2009, p. 7ff)

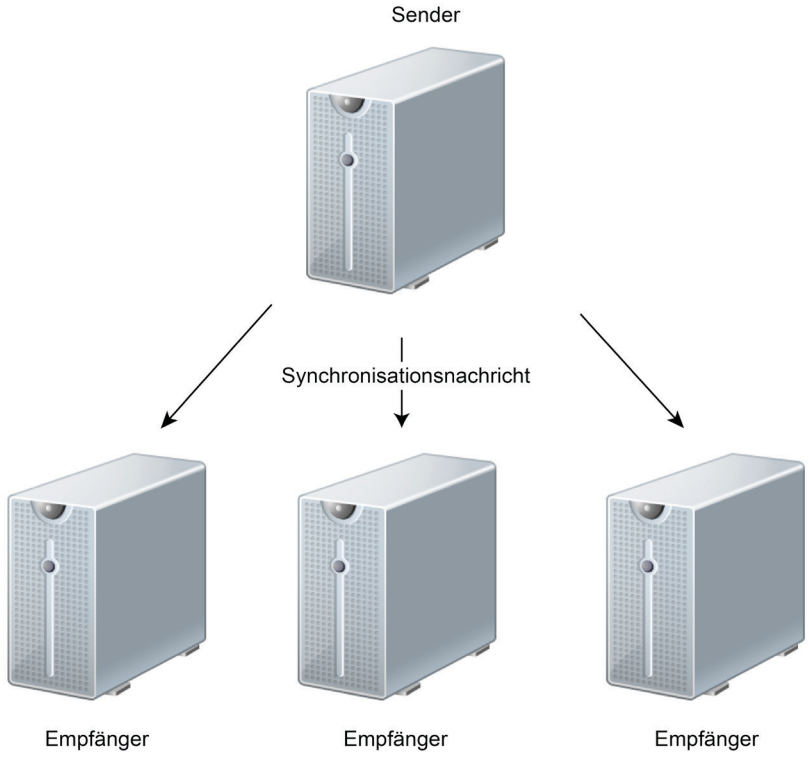

Abbildung 3: Synchronisationsvorgang

Ameling (2009, p. 7ff) betrachtete die Synchronisation in Datenbanksystemen. Überträgt man dieses Wissen auf Webapplikationen, erkennt man, dass der Impuls zur Synchronisation aufgrund des Übertragungsprotokolls HTTP[5] nicht direkt vom Webserver dem eigentlichen Sender ausgehen kann. Da der Webserver seine Clients nicht kennt, muss der Impuls vom Client, also dem eigentlichen Empfänger, ausgehen. Dieser sendet an den Server (Sender) eine entsprechende Anfrage und empfängt daraufhin die Daten zur Synchronisation. Dieser Vorgang ist in Abbildung 4 dargestellt.

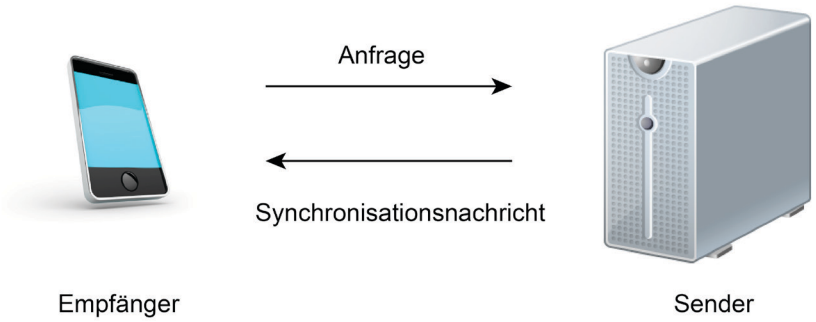

Abbildung 4: Synchronisationsvorgang bei Webapplikationen

5 Das Hypertext Transfer Protocol (kurz HTTP) ist ein zustandsloses Protokoll. Zwischen Client und Server werden Nachrichten (Anfrage vom Client an den Server und Antwort vom Server auf die Anfrage eines Clients) gesendet.

Eine Webanwendung, die Web Sockets (Näheres dazu in Kapitel 2.7.4.6) einsetzt, kann sehr wohl dem in (Ameling, 2009, p. 7ff) vom Autor beschriebenen Muster folgen.

Pataky (2005, p. 40ff) beschreibt in seiner Arbeit Synchronisationsarten anhand von Datenbanken. Er unterscheidet in unidirektionale und bidirektionale Synchronisation.

Unidirektional bedeutet, dass alle Daten nur in eine Richtung synchronisiert werden. Änderungen werden nur von der zentralen Datenbank an die Replikate weiter gegeben. Der Abgleich des Local Cache of Documents (vgl. Kapitel 1.4) wird unidirektional erfolgen. Änderungen werden nur an den Ressourcen des Webservers getätigt. Clients haben nur Leserechte.

Bei einer **bidirektional**en Synchronisation wird in beide Richtungen abgeglichen. Es können demnach unterschiedliche Datenbestände bestehen. Die Änderungen müssen bei einem Synchronisationsvorgang berücksichtigt werden. Es kommt zu einem Konflikt, wenn der gleiche Datensatz auf unterschiedlichen Endgeräten bearbeitet wird. Deshalb müssen Regelungen getroffen werden, die diesen Konflikt auflösen. Nach Pataky (2005, p. 40ff) sehen diese Regelungen wie folgt aus:

- Es wird der aktuellere Wert der beiden Varianten gewählt. Dieser kann bspw. anhand eines Zeitstempels ermittelt werden. (vgl. auch Optimistic Offline Lock, Kapitel 2.3.4)
- Die Datensätze werden zum Ändern ausgecheckt und können somit nur vom auscheckenden Rechner bearbeitet werden. Erst wenn die Datensätze wieder eingecheckt werden, sind diese für weitere Bearbeitungen freigegeben. (vgl. auch Pessimistic Offline Lock, Kapitel 2.3.5)

Der Local Cache of Data (vgl. Kapitel 1.4) muss sowohl les- als auch schreibbar sein. Damit kommt es zur bidirektionalen Synchronisation. Konflikte können auftreten, wenn Änderungen an den gleichen Daten an mehreren Endgeräten zur selben Zeit vorgenommen werden. Sofern die Datenbankreplikation des mobilen Informationssystems als Master-Slave (vgl. Kapitel 2.4.1) implementiert ist, kommt eine unidirektionale Synchronisation zum Einsatz.

2.2. Relevante Ansätze im Umfeld der Datensynchronisation

Es gibt zahlreiche Anwendungen, bei denen Daten synchronisiert werden. Alle anzuführen würde den Rahmen dieser Diplomarbeit sprengen. Dennoch sollen der Vollständigkeit halber einige Herangehensweisen kurz zusammengefasst werden. In weiterer Folge werden relevante Anwendungen und Ansätze näher beleuchtet sowie Rückschlüsse für die Datensynchronisation in mobilen Informationssystemen gezogen.

Ein frühes System zur Datensynchronisierung ist das Coda File System. Satyanarayanan et al. (1990) beschreiben dieses System, das auf dem Client-Server-Modell[6] basiert. Coda protokolliert lokale Dateimodifizierungen in einem Client Modification Log, um nicht jede Änderungen an den Server senden zu müssen und so Bandbreite zu sparen.

Kao et al. (2009) beschreiben eine Middleware für mobile Geräte mit folgenden Spezifikationen:

- NutzerInnen haben Zugriff auf vordefinierte Webinformationen ohne Netzwerkverbindung,
- Offline-Daten werden am mobilen Gerät gespeichert und
- auch wenn der Webbrowser nicht Google Gears[7] unterstützt, können Webseiten offline aufgerufen werden.

Die Flying Templates wurden von Tatsuori & Suzumura (2009) entwickelt. Dabei werden HTML-Templates erstellt, die in weiterer Folge am Client gespeichert und durch Einsetzen der Inhalte in die Platzhalter der Templates zur Generierung des User Interfaces genutzt werden. Der Client hält demnach die Programm-Ressourcen lokal. Die Flying Templates basieren auf dem Design Pattern Template View[8].

Gonçalves & Leitão (2009) entwickelten einen adaptiven Mechanismus zum Vorladen von Daten. Das System basiert auf Heuristiken, um die Situation, in welcher der User sich gerade befindet, auszuwerten und so dem Client, dynamischen Inhalt liefern zu können.

---

6 Das Client-Server-Modell sieht vor, dass innerhalb eines Netzwerkes der Client einen Dienst vom Server anfordern kann. Der Server verarbeitet die Anfrage und sendet gegebenenfalls ein Ergebnis zurück. (vgl. Chantelau & Brothuhn, 2009, p. 40)

7 Google Gears unterstützt als Browsererweiterung u.a. die Offline-Fähigkeit von Webapplikationen. Google Gears wird in Kapitel 2.2.1 näher betrachtet.

8 Die Template View ist ein Design Pattern für Webpräsentationen. Bei der Erstellung einer statischen HTML-Seite werden Markierungen eingefügt. Bei Anforderung der Seite werden diese bspw. durch Ergebnisse einer Datenbankabfrage ersetzt. (vgl. Fowler, 2003, p. 388ff)

Aufbauend auf dem MVC-Modell[9] beschäftigen sich La et al. (2011) grundsätzlich mit der Architektur von mobilen Applikationen. In ihrem System gibt es View, Controller und Model auf der Clientseite sowie Controller und Model auf der Serverseite. Außerdem nutzen sie client- und serverseitige Datenbanken.

2.2.1. Plucker und WebToGo

Die ersten ernsthaften Ambitionen, Webseiten offline verfügbar zu machen, können mit dem Durchbruch des Betriebssystems Palm verbunden werden. Eine dieser Anwendungen ist Plucker.

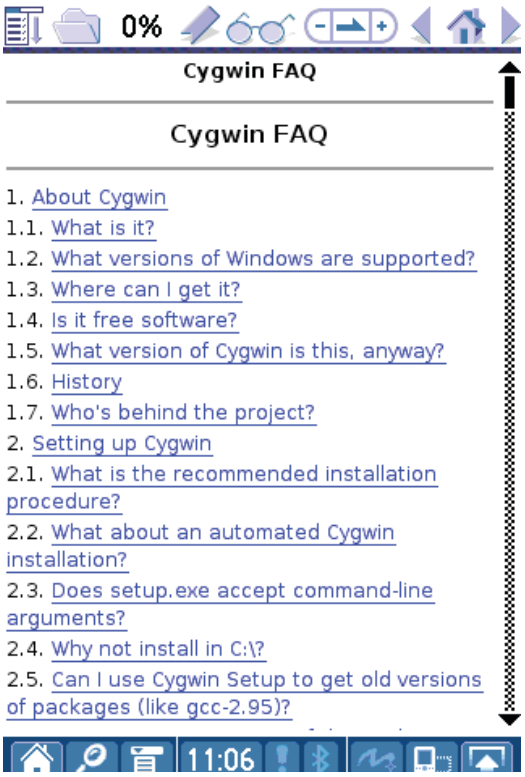

Abbildung 5: Plucker-Viewer auf dem mobilen Gerät (Quelle: Plucker, 2010a)

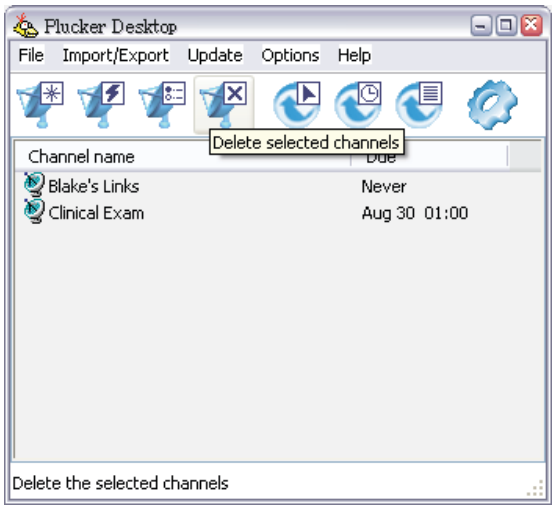

Abbildung 6: Plucker-Desktop (Quelle: Plucker, 2003)

Plucker ist ein Offline Web- und eBook-Reader für PDAs und andere mobile Geräte, die auf Palm OS und Windows Mobile basieren. Mit Plucker-Desktop (Abbildung 6) wählen die BenutzerInnen am Rechner aus, welche Webseiten am mobilen Gerät gelesen werden sollen. Diese werden von Plucker konvertiert, komprimiert und auf das mobile Gerät transferiert, damit diese im Plucker-Viewer (Abbildung 5) gelesen werden können. Plucker kann mit unterschiedlichen Quellen arbeiten – RSS, PDF, Textdateien, HTML etc. (Plucker, 2005, 2010b)

9 Model-View-Controller (kurz MVC) wird von Fowler (2003, p. 367ff) als Design Pattern für Webpräsentationen angeführt. Es unterscheidet drei Rollen (Model = Datenmodell, View = Präsentation, Controller = Programmsteuerung) und zerlegt damit die Interaktion mit dem User Interface in drei verschiedene Rollen.

WebToGo bietet im Gegensatz zu Plucker sowohl Online- als auch Offline-Browsing. Es beinhaltet eine WebSync-Funktion, mit der Internetseiten, die über die Internetverbindung des PCs geladen wurden, auf den PDA per HotSync[10] übertragen werden. Es ist möglich, Channels zu abonnieren oder eigene zu erstellen. Die Linktiefe innerhalb einer Website kann definiert werden. Außerdem ist es möglich, Links zu externen Seiten verfolgen zu lassen und somit externe Links vorzuladen. (vgl. Pakalski, 2002; WebToGo Mobiles Internet GmbH, 2003, p. 22ff)

Einstellungsmöglichkeiten, wie sie die BenutzerInnen von Plucker und WebToGo hatten, könnten bei der Datensynchronisation von mobilen Informationssystemen ebenfalls Berücksichtigung finden. So kann dem User die Möglichkeit gegeben werden, Einfluss auf die vorzuladenden Daten zu nehmen.

2.2.2. Google Gears

Das Google Gears API wurde 2007 veröffentlicht und ist eine Open Source Schnittstelle, die u.a. die Offline-Fähigkeit von Webapplikationen unterstützt. Die dazugehörige Browsererweiterung konnte auf Desktop-Rechnern, Laptops und Handheld-Geräten genutzt werden. (vgl. Kao et al., 2009, p. 339; Vaughan-Nichols, 2010)

Das Gears API enthielt u.a. die Module Local Server und Database. Local Server unterstützte durch lokales Zwischenspeichern das Offline-Halten von Applikationsressourcen (HTML, JavaScript, Bilder etc.). Das Modul Database erlaubte das lokale Speichern von Daten in einer durchsuchbaren relationalen Datenbank. Weitere Module waren Geolocation zur Bestimmung der Position des Users, HttpRequest für AJAX-Funktionalität, Canvas zur Bildmanipulation und Worker Pool, der es ermöglichte, JavaScript-Code asynchron ablaufen zu lassen. (vgl. Google, n.d.-a, n.d.-b)

Die Weiterentwicklung des Gears API wurde eingestellt. HTML 5 konnte jedoch von den Erfahrungen des Google Gears-Teams profitieren. In diesem zukünftigen HTML-Standard wird bspw. das Cache Manifest (siehe Kapitel 2.7.1.1) implementiert, das vergleichbar mit dem Gears-Modul Local Server ist. (vgl. Boodman, 2011)

Google Gears besteht aus zwei Teilen: dem Browser-Plugin und der API-Sammlung. Das API wird mit JavaScript angesprochen. Abbildung 7 stellt den Aufbau einer mit Google Gears implementierten Applikation dar.

10 HotSync ist ein Programm zur Synchronisation von Daten zwischen einem PDA und einem PC.

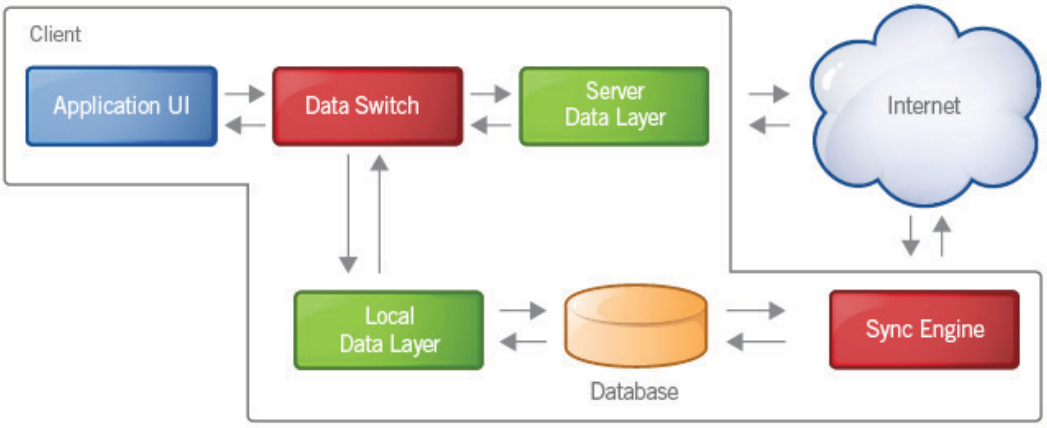

Abbildung 7: Aufbau einer Applikation mit Google Gears (Quelle: Google, n.d.-b)

Google behandelt in der Google Gears-Dokumentation Themenbereiche, die relevant für die Datensynchronisation in mobilen Informationssystemen sind.

2.2.2.1. Modal vs. Modeless Application

Bei einer **modal**en Applikation wechselt der User bewusst zwischen den Modi Online und Offline. Die Applikation kommuniziert mit dem Server, wenn sie online ist. Ist sie offline, wird der lokale Speicher genutzt. Die Daten werden synchronisiert, wenn zwischen den Modi gewechselt wird. Diese Art der Applikation ist einfach zu implementieren. Als nachteilig stellt sich heraus, dass eine User-Interaktion erforderlich ist. Die NutzerInnen müssen bewusst zwischen den Modi wechseln. Wird der Wechsel vergessen, ist es bspw. nicht möglich, die Applikation offline zu nutzen, da die Daten nicht lokal gespeichert wurden. Dies stellt auch ein Problem dar, wenn die Netzwerkverbindung unterbrochen wird. (vgl. Google, n.d.-b)

Eine **modeless** Application arbeitet in der Annahme, dass sie keine Netzwerkverbindung hat. Dementsprechend weiß auch der User nicht, wann er offline ist. Die Anwendung nutzt den lokalen Speicher, wann immer es geht. Ist der Server verfügbar, werden Datensynchronisationen im Hintergrund (Background-Sync, vgl. Kapitel 2.2.2.2) vorgenommen. Als Vorteil dieser Art der Anwendung stellt sich die bessere User Experience heraus. Die NutzerInnen müssen nicht auf die Netzwerkverbindung achtgeben oder den Modus wechseln. Außerdem arbeitet die Applikation auch bei unterbrochener Netzwerkverbindung reibungslos. Der entscheidende Nachteil ist die Komplexität der Implementierung und der damit zusammenhängende Testaufwand. Der Synchronisationsprozess im Hintergrund darf nicht zu viele Ressourcen in Anspruch nehmen. (vgl. Google, n.d.-b)

2.2.2.2. Manuelle Synchronisation vs. Synchronisation im Hintergrund

Eine Datensynchronisation wird nötig,

1. wenn NutzerInnen Änderungen im Offline-Status machen,
2. wenn Daten verteilt sind und von externen TeilnehmerInnen geändert werden können sowie
3. wenn Daten von einer externen Quelle bspw. einem Feed kommen.

Dabei gibt es zwei Arten der Synchronisation: manuelle Synchronisation und Hintergrund-Synchronisation (Background Sync).

Bei der **manuellen Synchronisation** entscheidet der User, wann synchronisiert wird. Diese Synchronisationsmethode kann einfach implementiert werden, in dem alle veränderten Daten zum Server gesendet werden und beim Offline-Gehen eine neue Kopie der Daten vom Server geladen wird. Für diese Methode müssen die Daten klein genug zum Download sein, um diese in einer angemessenen Zeit zu übertragen. Außerdem muss der User dem System bewusst melden, dass er offline geht. Da jedoch die NutzerInnen vor allem in einem mobilen Szenario nicht immer über den Status ihrer Netzwerkverbindung Bescheid wissen, stellt sich diese Methode als problematisch dar. Des Weiteren könnten die User vergessen, zu synchronisieren bevor sie offline gehen.

Im Gegensatz dazu steht der **Background Sync**. Hier synchronisiert eine Sync Engine (vgl. Abbildung 7) innerhalb der Anwendung kontinuierlich die Daten zwischen lokalem Speicher und dem Server. Dies kann laut Google (N.d.-b) über einen Server-Ping[11] in regelmäßigen Abständen oder über einen Server-Push[12] erfolgen. Durch eine Synchronisation im Hintergrund stehen die Daten auch dann zur Verfügung, wenn sich der User entscheidet, offline zu gehen oder die Verbindung verliert. Die Performance wird vor allem bei langsamer Internetverbindung erhöht. Als Nachteile lassen sich anführen, dass die Sync Engine zusätzliche Ressourcen verbraucht und die Online Experience des Users durch die Verarbeitungen im Hintergrund verlangsamt. (vgl. Google, n.d.-b)

Die Entscheidungen, ob modal oder modeless und manuelle oder Hintergrund-Synchronisation, sind grundlegend für die Umsetzung des Proof Of Concept in Kapitel 3.

2.2.3. Where Store

Stuedi et al. (2010) entwickelten einen Location-basierten Datenspeicher für Smartphones. Die wichtigste Eigenschaft ist die Nutzung der Location History des Telefons, um zu bestimmen, welche Daten lokal repliziert werden. Aus dem Anwendungsver-

11 Ein Server-Ping ist eine Anfrage beim Server. In Webanwendungen erfolgt dieser über einen XMLHttpRequest (siehe Kapitel 2.7.3).

12 Beim Server-Push sendet der Server Daten, z.B. sobald sich in der Datenbank etwas verändert. Diese Variante der Implementation kann mit Hilfe von Web Sockets implementiert werden (siehe Kapitel 2.7.4.6).

halten einer Person in der Vergangenheit lässt sich gut eine Prognose darüber ableiten, welche Informationen sie in Zukunft benötigen wird.

Studi et al. verfolgen mit dem Caching der Daten im Smartphone das Ziel, die Datenzugriffszeiten sowie die Wahrscheinlichkeit der Nichtverfügbarkeit von Daten aufgrund fehlender Netzwerkverbindung zu reduzieren.

Where Store nutzt ein Filtered Replication System (vgl. Kapitel 2.4.2). Ein Filter stellt das Set von Elementen basierend auf der aktuellen und vorhergesagten Location des Users zusammen, auf das in naher Zukunft zugegriffen werden soll.

Abbildung 8 zeigt die Architektur des Frameworks Where Store. Es handelt sich um ein Client-Server-System. Der Client läuft auf dem Mobiltelefon und der Server läuft im Netzwerk (Cloud). Der Where Store Client auf dem Telefon besteht aus zwei Teilen: dem Location Service und dem Replication Subsystem. Der Location Service stellt Informationen über die zukünftige Position auf dem Mobiltelefon bereit. Das Replication Subsystem hält die Daten in einem lokalen Speicher bereit, die mit der Cloud synchronisiert werden. Der Serverteil von Where Store enthält ebenfalls ein Replication Subsystem, jedoch kein Location Service. Die Applikation selbst interagiert mit Where Store über eine Konfigurationsdatei. Diese definiert, welche Elemente bei einer bestimmten Position lokal gespeichert werden sollen. (vgl. Studi et al., 2010)

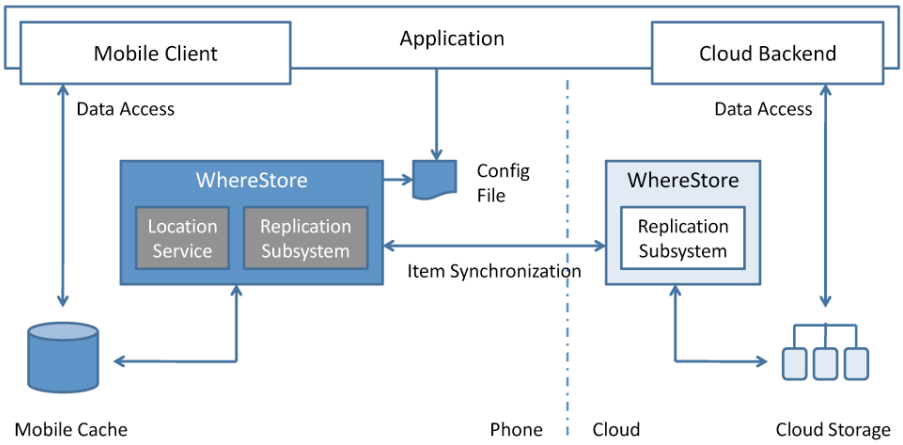

Abbildung 8: Architektur von Where Store (Quelle: Studi et al., 2010)

Der Location Service basiert auf StarTrack[13]. Auf dem mobilen Gerät läuft der StarTrack Client. Dieser lokale Service erfasst periodisch die aktuelle Position des Users (z.B. über GPS) und leitet sie an den StarTrack Server weiter, der sich als Service in der Cloud befindet. Der StarTrack Server verarbeitet die Positionsdaten zu Tracks, welche die Route des Users repräsentieren. (vgl. Ananthanarayanan, Haridasan, Mohomed, Terry & Thekkath, 2009)

13 StarTrack ist ein Framework, das ein Set von Operationen zur Verfügung stellt, um die Entwicklung und Verwendung Track-basierter Anwendungen zu vereinfachen. (vgl. Ananthanarayanan, Haridasan, Mohomed, Terry & Thekkath, 2009)

Zur Vorhersage der zukünftigen Position des Users wird als Datenstruktur ein Place Transition Graph verwendet. Dieser Graph wird aus den Tracks erstellt, die der Star-Track Service zur Verfügung stellt, und ist ein möglicher Weg Location-Vorhersagen zu erstellen. (vgl. Stuedi et al., 2010)

Besonders mobile Informationssysteme, die auf Basis der Position Informationen bereitstellen, profitieren von der Art der Vorhersage und des Vorladens, wie es das Application Framework Where Store umsetzt. Anhand der aktuellen Position der NutzerInnen lassen sich die naheliegenden und damit zukünftigen Positionen ermitteln und dementsprechend die passenden Inhalte vorladen.

2.2.4. Sync Kit

Benson et al. (2010) stellen in ihrem Paper Sync Kit vor. Sync Kit[14] ist eine Bibliothek basierend auf JavaScript und Python, die es ermöglicht clientseitige HTML 5-Speicher mit einer serverseitigen Datenbank zu synchronisieren. Das Toolkit bietet dabei verschiedene Strategien zum Synchronisieren relationaler Datenbanktabellen zwischen Webbrowser und Webserver durch eine clientseitige Template-Bibliothek.

Die Ausgangssituation von Benson et al. war, dass Applikationsentwickler den Cache manuell managen müssen. Dabei müssen die Abfragen modifiziert werden, um einen Vorteil aus der clientseitigen Datenhaltung ziehen zu können. Außerdem muss Code geschrieben werden, um herauszufinden, ob die gespeicherten Ergebnisse am Client noch aktuell sind sowie die gespeicherten Ergebnisse mit den Updates des Servers zusammengefügt werden können.

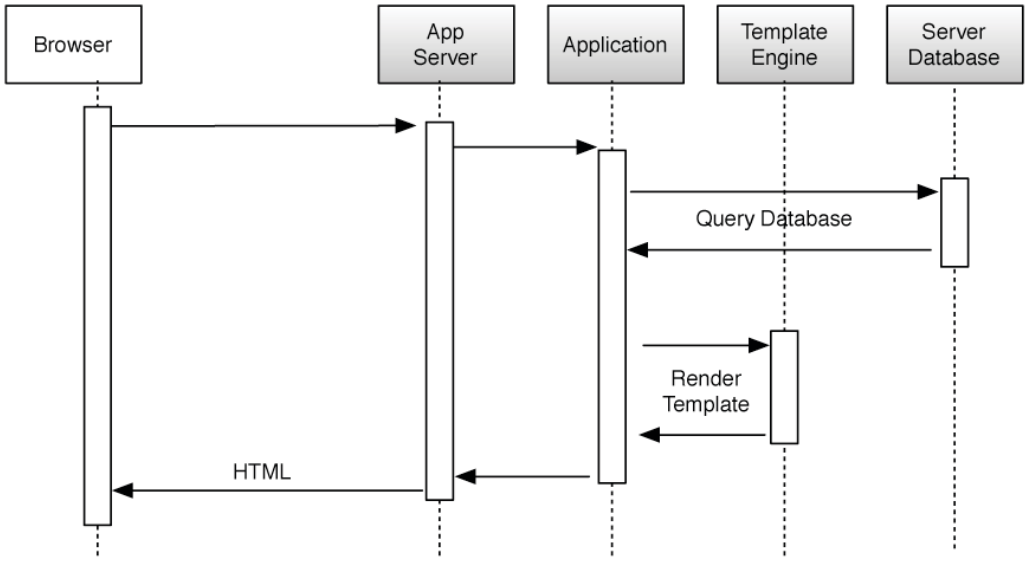

Abbildung 9: Traditioneller Rendervorgang einer Webseite (Quelle: Benson et al., 2010, p. 124)

14 Die Sync Kit-Bibliothek steht unter https://github.com/synckit/synckit zur Verfügung.

Abbildung 9 verdeutlicht einen traditionellen Rendervorgang (weiße Box = Clientseite, graue Box = Serverseite). Dabei fordert der Webbrowser die Webseite vom Server an. Die gesamte Daten- und Template-Zusammenstellung erfolgt auf der Server-Seite. Der Webbrowser erhält HTML und stellt dieses dar.

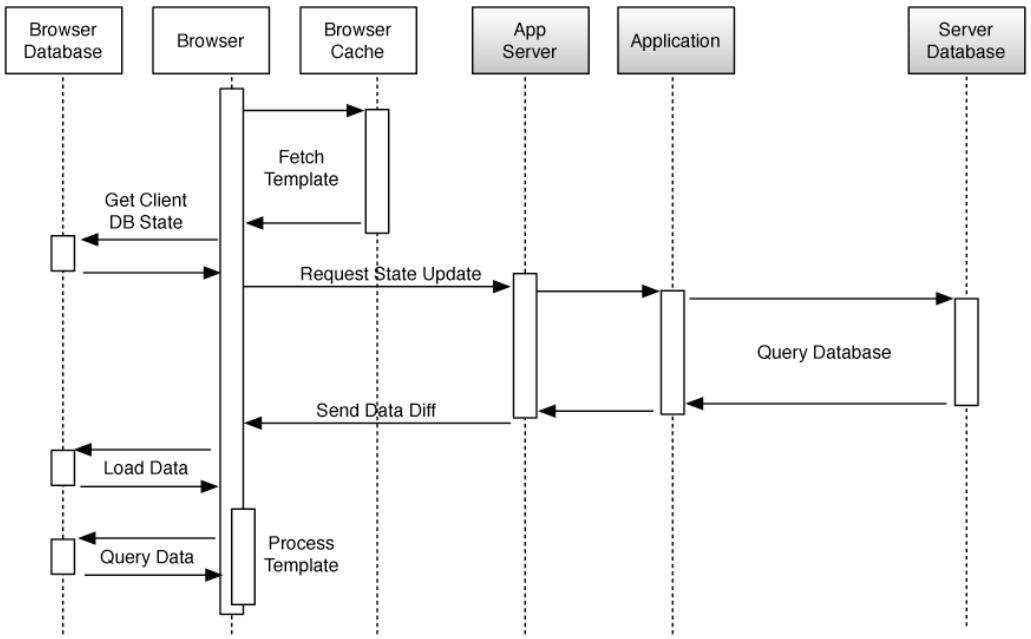

Abbildung 10: Rendervorgang einer Webseite mit Sync Kit (Quelle: Benson et al., 2010, p. 124)

In Abbildung 10 ist zum Vergleich der Rendervorgang einer Webseite mit Sync Kit dargestellt. Hierbei erfolgt die Verarbeitung der Daten auf der Clientseite. Vom Server werden bei Aufruf lediglich die Datenänderungen angefordert. Das Laden der Daten erfolgt von der clientseitigen Datenbank genauso wie das Laden in das Template.

Listing 1: Sync Kit Template für eine Liste von Blog-Einträgen (Quelle: Benson et al., 2010, p. 122)

```
<section id="blog-posts"
   data-query="SELECT title, author, body FROM entry_data
               ORDER BY lastModified DESC LIMIT 10"
   data-as="Entry">
      <article itemscope itemtype="Entry">
         <h2 itemprop="title"></h2>
         By <span class="author" itemprop="author"></span>
         <div class="contents" itemprop="body"></div>
      </article>
</section>
```

Listing 1 zeigt, wie ein Sync Kit Template aussehen kann. Es besteht aus HTML 5 versehen mit Tag-Attributen, die definieren, welche Daten von der clientseitigen Daten-

bank in das Template geladen werden. Im `section`-Element wird bspw. über das Attribut `data-query` die SQL Query zur Abfrage der Datenbank definiert.

In (Benson et al., 2010) zeigen die Autoren, dass clientseitiges Rendering trotz ausgiebiger Nutzung von JavaScript und den Overheads durch clientseitige Datenbankzugriffe keinen negativen Einfluss auf die clientseitige Performance hat. Für datenintensive Webseiten bietet Sync Kit signifikante Performance-Verbesserungen.

2.2.5. Verteilte Datenbanken (Distributed Database)

Eine verteilte Datenbank ist eine Sammlung von mehreren, logisch miteinander verknüpften Datenbanken, die über ein Computernetzwerk verteilt sind. (vgl. Ozsu & Valduriez, 2011, p. 3)

Aus dem Bereich der verteilten Datenbanken sind bezugnehmend auf das Thema dieser Diplomarbeit besonders die Delivery Modes, das Synchronisationsintervall und die Kommunikationsmethoden von Interesse.

2.2.5.1. Delivery Modes

Es wird in pull-only, push-only und hybrid unterschieden.

Im Fall **pull-only** erfolgt der Datentransfer vom Server zum Client. Gestartet wird dieser mit einem Pull des Clients. Wenn die Anfrage des Clients beim Server ankommt, antwortet der Server mit der angefragten Information. Neue Daten oder Updates werden also nicht automatisch an den Client ausgeliefert, sondern nur nach explizierter Anfrage durch den Client.

Bei **push-only** wird der Datentransfer vom Server zum Client ohne eine spezifische Anfrage des Clients durch einen Server-Push initiiert.

Die **hybride** Variante kombiniert Client-Pull und Server-Push.

(vgl. Ozsu & Valduriez, 2011, p. 5f)

Eine normale Webbrowser - Webserver Verbindung über das HTTP-Protokoll lässt ausschließlich den pull-only Modus zu. In Verbindung mit HTML 5 und dem Web Socket API (Näheres dazu in Kapitel 2.7.4.6) ist auch der push-only bzw. der hybride Delivery Mode in einem mobilen Informationssystem umsetzbar.

2.2.5.2. Synchronisationsintervall

Die Synchronisation kann periodisch, unter bestimmten Bedingungen sowie ad-hoc (oder irregulär) erfolgen.

In einer **periodisch**en Synchronisation werden Daten in regemäßigen Intervallen von Server zu Client gesendet. Dies kann sowohl beim Pull als auch beim Push sinnvoll sein.

Bei der **bedingten Auslieferung** werden nur Daten vom Server gesendet, wenn bestimmte Bedingungen am Client erfüllt sind. Eine einfache Bedingung kann bspw. das Erreichen eines bestimmten Zeitpunktes sein. Zumeist wird diese Art der Synchronisation bei push-only oder der hybriden Variante genutzt.

In einem pull-basierten System wird meist eine **ad-hoc** oder irreguläre Synchronisation implementiert. Die Daten werden ad-hoc vom Server gepullt, wann immer die Clients anfragen.

(vgl. Ozsu & Valduriez, 2011, p. 6)

Für die Datensynchronisation in mobilen Informationssystemen kommen alle drei Synchronisationsintervalle in Frage.

2.2.5.3. Kommunikationsmethode

Kommunikationsmethoden beschreiben Verfahren der Auslieferung der Daten zum Client. Es wird in unicast und one-to-many unterschieden.

Bei einer Kommunikation über **unicast** sendet der Server die Daten zu einem einzelnen Client (Punkt-zu-Punkt).

Im Fall **one-to-many** sendet der Server die Daten an viele Clients (Multicast- oder Broadcast-Protokoll). Diese Kommunikationsmethode ist jedoch für eine Webanwendung eines mobilen Informationssystems nicht relevant.

(vgl. Ozsu & Valduriez, 2011, p. 7)

2.3. Design Pattern

Design Pattern in der Software-Entwicklung gehen auf Entwurfsmuster in der Architektur zurück. Alexander et al. definieren in (Alexander, Ishikawa & Silverstein, 1977), dass ein Muster ein wiederkehrendes Problem beschreibt, das in unserer Umwelt auftritt. Ein Pattern erklärt den Kern der Lösung. (vgl. Fowler, 2003, p. 15; Gamma, Helm & Johnson, 2001, p. 3)

Alexander war Architekt und definierte 1977 Entwurfsmuster für Gebäude. Die Definition lässt sich jedoch auch gut für Software verwenden. Ein Pattern ist als Ratschlag zu verstehen. Der Fokus liegt auf gebräuchlichen und bewährten Lösungen für ein oder mehrere wiederkehrende Probleme. Ein Schlüsselaspekt bei der Anwendung von Entwurfsmustern ist, dass die Lösungen nie blind anwendbar sind. Bei der Benutzung sind Anpassungen an die jeweils gegebene Aufgabenstellung notwendig. (vgl. Fowler, 2003, p. 24)

Im folgenden Abschnitt werden relevante Design Pattern für die Umsetzung von mobilen Informationssystemen mit Datensynchronisationen beschrieben.

2.3.1. Lazy Load

> *„Ein Objekt, das nicht alle benötigten Daten enthält, aber weiß, wie sie zu bekommen sind.“*
> (Fowler, 2003, p. 227)

2.3.1.1. Problem und Lösung

Ein Objekt aus einer Datenbank soll in den Arbeitsspeicher geladen werden. Für eine spätere Performance ist es gut, wenn alle Objekte, die mit diesem Objekt verbunden sind, mitgeladen werden. Dies kann unter Umständen dazu führen, dass eine riesige Auswahl an verbundenen Objekten geladen werden muss. Wenn tatsächlich nur einige Objekte benötigt werden, kann dieses Vorgehen das Leistungsverhalten der Anwendung beeinträchtigen. (vgl. Fowler, 2003, p. 227)

Lazy Load unterbricht den Ladeprozess an einer bestimmten Stelle. Es wird eine Markierung in die Objektstruktur eingefügt, damit die Daten erst bei Bedarf nachgeladen werden können. Damit wird das Laden der Objekte verzögert, bis diese benötigt werden. (vgl. Fowler, 2003, p. 228; van Zyl, Kourie, Coetzee & Boake, 2009, p. 151)

2.3.1.2. Einsatz im mobilen Informationssystem

Lazy Load (dt. etwa verzögertes Laden) also das Nachladen bei Bedarf kommt für die Implementierung des Ladevorganges des Local Cache of Data sowie Local Cache of Documents in Frage.

2.3.2. Eager Load

Den gegenteiligen Ansatz zum Lazy Load Pattern verfolgt Eager Load. Von Eager Load wird gesprochen, wenn alle Komponenten sofort einsatzbereit gemacht werden. Dabei werden auch alle Abhängigkeiten geladen. Diese Art des Vorgehens kann beim Laden Wartezeiten verursachen und dazu führen, dass importierte Daten nicht genutzt und somit unnötige Ressourcen beansprucht werden. (vgl. van Zyl et al., 2009, p. 151; Wider, 2007, p. 26)

Nach Schwichtenberg (2012) sind folgende Entscheidungskriterien zwischen Lazy und Eager Load relevant:

- Wahrscheinlichkeit, dass die Daten benötigt werden
- Größe der Datenmenge
- Nachladen der Daten später möglich

2.3.2.1. Einsatz im mobilen Informationssystem

Bezugnehmend auf die in Kapitel 1.2 geschilderten Nutzungsszenarien ist für Szenario 1 (Kein bzw. nur kurzzeitiger Internetzugriff vorhanden) Lazy Load nicht zielführend. Für dieses Szenario müssen sämtliche Daten sowohl im Local Cache of Documents als auch im Local Cache of Data mittels Eager Load geladen werden. Alle notwendigen Daten und Ressourcen müssen auf dem Gerät zur Verfügung stehen.

Für Szenario 2 (Antwortzeiten verbessern) ist Eager Load bei großen Datenmengen besonders für den Local Cache of Data nicht relevant. Eine spezielle Technik, die in (Mason, 2010) vom Autor als Over-eager Loading bezeichnet wird, kann hier zielführend sein. Bei dieser Technik wird vorausgeahnt, welche Daten der User anfragen wird. Dementsprechend werden die notwendigen Daten vorgeladen.

Where Store (vgl. Kapitel 2.2.3) zeigt eine Umsetzung von Over-eager Loading. Im Gegensatz zu Lazy Load, wo nur bei Bedarf geladen wird, werden bei Where Store die Objekte vorgeladen, die anhand der aktuellen und zukünftigen geografischen Position nahe liegen. Es wird demnach vorausgeahnt, welche Daten für den User relevant sein können.

2.3.3. Proxy

> *„Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjektes.“*
> (Gamma et al., 2001, p. 254)

2.3.3.1. Problem und Lösung

Das Proxy Pattern ist anwendbar, sobald der Bedarf nach anpassungsfähigen und intelligenten Referenzen auf ein Objekt besteht. Es können u.a. Kosten gespart oder zusätzliche Funktionen implementiert werden, in dem bspw. die Initialisierung eines

Objektes soweit verzögert wird, bis es tatsächlich genutzt wird. (vgl. Gamma et al., 2001, p. 254ff)

Dabei können nach (Schmidt, 2006, p. 96) verschiedene Varianten implementiert werden:

1. Zugriff auf ein Objekt auf einem anderen Server (Remote-Proxy)
2. Erzeugen des Objekts beim ersten Zugriff (virtueller Proxy)
3. Durchführen von Verwaltungsaufgaben (Schutz-Proxy)
4. Ersatz für einfachen Zeiger mit zusätzlichen Aktionen bei Objektzugriff (Smart-Reference)

2.3.3.2. Einsatz im mobilen Informationssystem

Wird bei der Programmierung der Datensynchronisation eines mobilen Informationssystems wie im Proof of Concept ein Cache Manager[15] eingesetzt, kann dieser als Proxy implementiert werden. In (Floyd, Housel & Tait, 1998) beschreiben die Autoren einen Proxy als Stück Software, das in einen Client-to-Server Kommunikationspfad eingefügt wird. Dies kann sowohl auf Server- als auch auf Clientseite erfolgen.

2.3.4. Optimistic Offline Lock

> *„Verhindert Konflikte zwischen nebenläufigen Geschäftstransaktionen, indem es einen Konflikt entdeckt und die Transaktion zurücksetzt.“*
> (Fowler, 2003, p. 459)

2.3.4.1. Problem und Lösung

Ein Konflikt entsteht, wenn zwei User die gleichen Datensätze zur gleichen Zeit ändern. Das Design Pattern Optimistic Offline Lock geht davon aus, dass eine geringe Wahrscheinlichkeit für Konflikte besteht und setzt auf Konflikterkennung. Es ermöglicht damit das gleichzeitige Arbeiten an den gleichen Daten durch mehrere BenutzerInnen. Beim Optimistic Offline Lock wird anhand einer Zustands-ID überprüft, ob das Editieren eines Datensatzes nicht mit einer Änderung derselben Daten einer anderen Sitzung in Konflikt steht. (vgl. Fowler, 2003, p. 459ff)

Abbildung 11 verdeutlicht die Vorgehensweise. User 1 holt sich den Datensatz mit der Zustands-ID 1. Kurze Zeit später fragt User 2 denselben Datensatz (Zustands-ID 1) ab. Beide User editieren diesen Datensatz gleichzeitig. User 1 speichert den Datensatz zuerst. Dabei wird die Zustands-ID überprüft, zu diesem Zeitpunkt hat der Datensatz in der Datenbank die Zustands-ID 1 und auch User 1 speichert den Datensatz mit Zustands-ID 1. Mit dem erfolgreichen Speichervorgang bekommt der Datensatz die Zustands-ID 2. Kurze Zeit später speichert nun auch User 2 den Datensatz

15 Der Cache Manager entscheidet bspw. wie der Datenzugriff erfolgt. Näheres dazu in Kapitel 3.4.4.

(Zustands-ID 1). Hierbei kommt es zu einem Konflikt, da der Datensatz in der Datenbank die Zustands-ID 2 besitzt. Das Speichern von User 2 schlägt fehl.

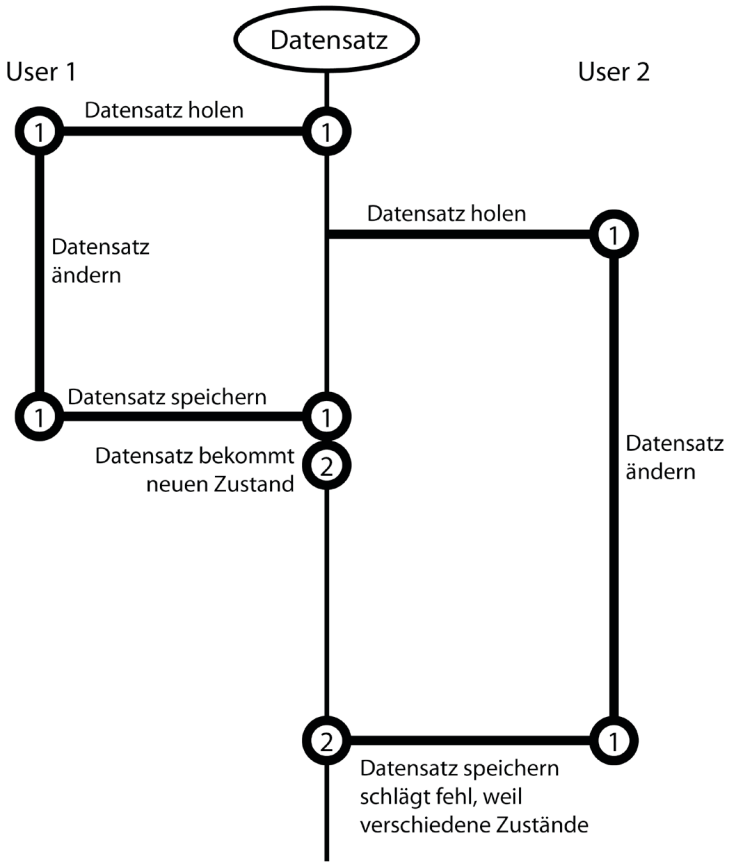

Abbildung 11: Optimistic Offline Lock (Quelle: Fowler, 2003, p. 460, modifiziert)

2.3.4.2. Einsatz im mobilen Informationssystem

Optimistic Offline Lock ist eine mögliche Implementierungsvariante zum Umgang mit Konflikten bei der Synchronisation des Local Cache of Data (vgl. Kapitel 1.4). Dabei ist gerade bei mobilen Endgeräten darauf zu achten, dass der User beim Auftreten eines Konfliktes die eingegebenen Daten nicht verliert. Das Tippen stellt sich für die BenutzerInnen von Smartphones häufig als mühsam heraus. Daher sollte es Möglichkeiten geben, um die Zustände von Datensätzen zu verbinden.

2.3.5. Pessimistic Offline Lock

> *„Verhindert Konflikte zwischen nebenläufigen Geschäftstransaktionen, indem es nur einer Geschäftstransaktion gleichzeitig den Zugriff auf die Daten erlaubt.“*
> (Fowler, 2003, p. 470)

2.3.5.1. Problem und Lösung

Pessimistic Offline Lock geht von einer hohen Wahrscheinlichkeit für das Auftreten von Konflikten beim Editieren von Datensätzen aus. Es basiert auf Konfliktvermeidung. Der Datensatz wird für das Editieren für eine Bearbeitung durch einen anderen Prozess gesperrt. Dadurch ist es nicht möglich, dass zwei oder mehrere BenutzerInnen denselben Datensatz zur gleichen Zeit verändern. (vgl. Fowler, 2003, p. 470ff)

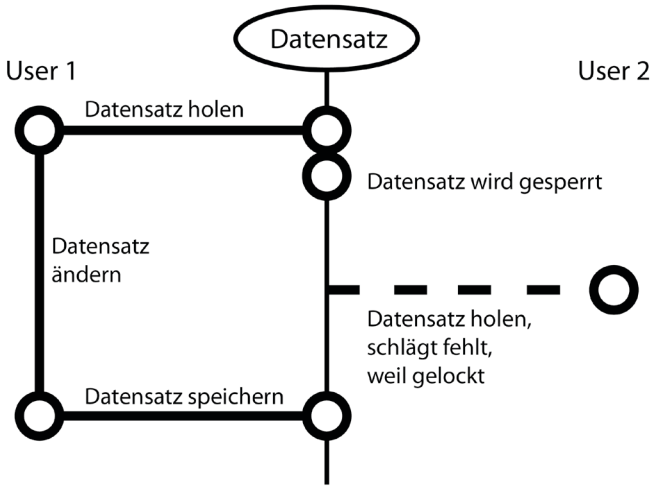

Abbildung 12: Pessimistic Offline Lock (Quelle: Fowler, 2003, p. 470, modifiziert)

Abbildung 12 zeigt die Vorgehensweise. User 1 holt sich den Datensatz zum Editieren, dabei wird der Datensatz in der Datenbank als gesperrt markiert. Deshalb ist es für User 2 solange nicht möglich diesen Datensatz zu verändern, bis User 1 die Änderungen gespeichert hat und der Datensatz entsperrt wird.

2.3.5.2. Einsatz im mobilen Informationssystem

In Bezug auf gute Usability stellt sich Pessimistic Offline Locking in Webanwendung als schwierig zu behandeln heraus. Vor allem, wenn der User einen Datensatz zum Editieren öffnet und anschließend das Browserfenster schließt, ohne den Datensatz freizugeben. Sofern keine automatische Freigabe des gesperrten Datensatzes bspw. nach einer bestimmten Zeitspanne erfolgt, können keine anderen NutzerInnen diesen Datensatz bearbeiten.

Da es in einer Webanwendung nach Szenario 1 (Kein bzw. nur kurzzeitiger Internetzugriff vorhanden) nicht vorgesehen ist, dass eine ständige Netzwerkverbindung zur Verfügung steht, ist Pessimistic Offline Lock nicht zielführend. Um Konflikte zu vermeiden, müssen theoretisch alle Datensätze, die auf dem Client gespeichert sind, in der Server-Datenbank gesperrt werden, da der User in der Offline-Phase alle Datensätze bearbeiten könnte. Damit könnte lediglich ein User diese Anwendung nutzen. Für Szenario 2 (Antwortzeiten verbessern) ist es jedoch möglich, Pessimistic Offline Lock umzusetzen.

2.3.6. Navigation Observer

2.3.6.1. Problem und Lösung

Der Navigation Observer bietet eine Lösung für die Rückverfolgung des vom User navigierten Pfades. Dabei werden die besuchten Nodes aufgezeichnet und in einer Navigation History verlinkt. Dieses Pattern wurde entwickelt, um den Navigationsprozess vom wahrnehmbaren Aufzeichnen des Prozesses zu entkoppeln. (vgl. Rossi, Garrido & Carvalho, 1996, p. 7ff; Thung, Ng, Thung & Sulaiman, 2010)

2.3.6.2. Einsatz im mobilen Informationssystem

Das Navigation Observer Pattern unterstützt bei der Implementierung eines History Tracking in ein mobiles Informationssystem. Das Pattern ist nützlich, wenn man History-Einträge des navigierten Pfades auf einfachem Weg zurückverfolgen möchte. Werden diese Informationen dem User zur Verfügung gestellt, kann dieser schneller seinen zurückgelegten Pfad finden.

2.3.7. Call Tracking

2.3.7.1. Problem und Lösung

Schnelle NutzerInnen und rechenintensive Anwendungen bringen Netzwerke und Server bis an ihr Limit. Asynchrone Interaktion ist die einzig praktikable Lösung für AJAX-Anwendungen (vgl. Kapitel 2.7.3). Da Browser nur mit einigen Anfragen gleichzeitig umgehen können, muss die Anzahl paralleler Anfragen kontrolliert werden.

Die Anzahl der gleichzeitig möglichen Verbindungen ist abhängig von der Implementierung des Browsers. Während ein Smartphone mit iOS 4 nur vier Verbindungen gleichzeitig zum selben Host aufbauen kann, unterstützt iOS 5 sechs gleichzeitige Verbindungen. Der Browser des Betriebssystems Android 4 unterstützt sechs und von Android 2.3 sogar neun gleichzeitige Verbindungen zum selben Host. Werden zum selben Zeitpunkt mehr als die angeführten Anfragen abgesetzt, so werden diese nacheinander ausgeführt. (vgl. Browserscope, 2012; Rehm, 2011)

Call Tracking beschäftigt sich mit dem Problem von parallelen XMLHttpRequest[16]-Aufrufen und wie man diese kontrollieren kann. Die Lösung besteht in der Protokollierung aller XHR. Alle Aufrufe werden in einer Queue verwaltet. Der Fortschritt jedes XHR wird protokolliert. So können die EntwicklerInnen bestimmen, wie viele XHR parallel abgesetzt werden. (vgl. Mahemoff, 2006, p. 210ff)

16 XMLHttpRequest (kurz XHR) ist ein API zum Austausch von Daten zwischen Client und Server über das HTTP-Protokoll.

2.3.7.2. Einsatz im mobilen Informationssystem

In einem mobilen Informationssystem kann Call Tracking implementiert werden. Dieses Pattern erhält jedoch erst Relevanz bei Applikationen mit hohem XMLHttpRequest-Aufkommen. In Szenario 1 (Kein bzw. nur kurzzeitiger Internetzugriff vorhanden) werden lediglich beim ersten Aufruf der Applikation bzw. bei Updates XHR benötigt, so dass ein Call Tracking nicht unbedingt notwendig ist. In Szenario 2 (Antwortzeiten verbessern) bekommt Call Tracking eine größere Relevanz, da hier Daten je nach Situation vorgeladen werden müssen und so eine größere Anzahl an XHR abgesetzt werden kann.

2.3.8. Periodic Refresh (Polling)

2.3.8.1. Problem und Lösung

Der Status vieler Webanwendungen ist von Natur aus unbeständig. Änderungen können aus einer Vielzahl an Quellen (bspw. andere NutzerInnen, externe News und Daten, Ergebnisse komplexer Berechnungen) kommen. HTTP-Anfragen können nur vom Client ausgelöst werden. Laut Mahemoff gibt es keine Möglichkeit, dass ein Server bei Datenänderungen eine Verbindung zum Client öffnet. Mahemoff schließt Comet (vgl. Kapitel 2.7.3.1) aus, da es nicht immer ideal und nicht sehr skalierbar ist. (vgl. Mahemoff, 2006, p. 215ff) Eine neue Alternative bieten Web Sockets, die in Kapitel 2.7.4.6 näher erläutert werden.

Periodic Refresh beschäftigt sich mit dem Problem, wie eine Anwendung ihre NutzerInnen über Änderungen auf dem Server informieren kann.

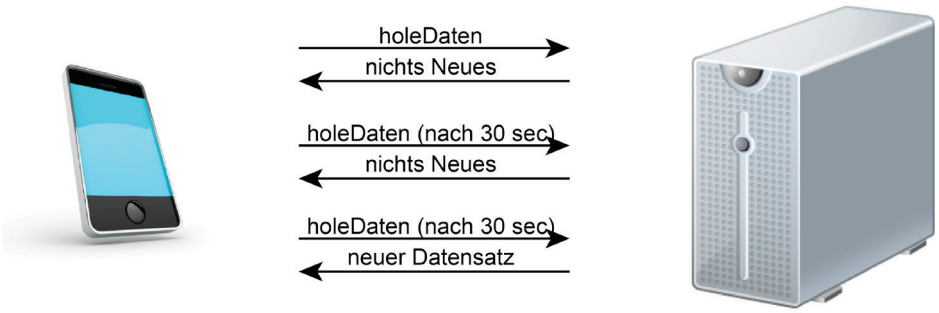

Abbildung 13: Periodic Refresh (Quelle: Mahemoff, 2006, p. 215, modifiziert)

Die Lösung wird in Abbildung 13 veranschaulicht. Der Browser sendet in einem bestimmten Intervall einen XMLHttpRequest, um neue Daten vom Server anzufordern.

2.3.8.2. Einsatz im mobilen Informationssystem

Mobile Betriebssysteme bieten Mechanismen zur Benachrichtigung (bspw. Push Notification in iOS[17]). Bisher können diese jedoch noch nicht komfortabel[18] durch Webanwendungen genutzt werden. Periodic Refresh stellt eine Variante dar, wie in einer Webapplikation Änderungen der Daten am Server erkannt werden können. Aufgrund der potentiell höheren Kosten durch die ständigen Datenübertragungen sollte diese periodische Abfrage bei einem mobilen Informationssystem nur aktiv sein, wenn der User es explizit wünscht. Zudem muss eine aktive Netzwerkverbindung zur Verfügung stehen.

2.3.9. Browser-Side Cache

2.3.9.1. Problem und Lösung

Eine Applikation soll auf eine User-Anfrage idealer Weise unmittelbar antworten. Viele User-Anfragen an die Applikation bedingen allerdings Anfragen an den Server. Aufgrund des Datentransfers und der Verarbeitung beim Server kann dessen Antwort wahrnehmbare Verzögerungen verursachen. (vgl. Mahemoff, 2006, p. 290)

Das Browser-Side Cache Pattern beschreibt, wie ein System schnell auf User-Aktionen reagieren kann. Abbildung 14 stellt das Pattern dar.

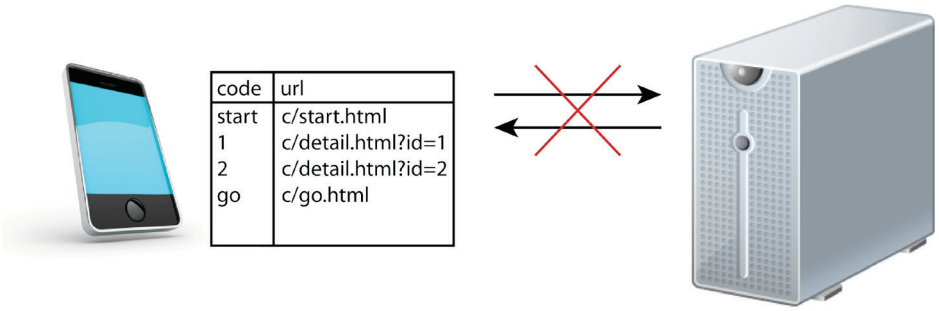

Abbildung 14: Browser-Side Cache (Quelle: Mahemoff, 2006, p. 290, modifiziert)

Serverresultate werden im Browser-Side Cache gehalten. Dies kann bspw. mit Hilfe des Proxy Patterns (vgl. Kapitel 2.3.3) umgesetzt werden.

2.3.9.2. Einsatz im mobilen Informationssystem

Ein mobiles Informationssystem sollte das Browser-Side Cache Pattern implementieren. In Szenario 2 (Antwortzeiten verbessern) ist es denkbar, Serverresultate auf

17 Der Server kann über den Apple Push Notification Service (kurz APNS) eine Nachricht an die Anwendung an die Clients senden.

18 Ein primitives Benachrichtigungssystem für Webapplikationen wäre z.B. bei Neuigkeiten in der Anwendung, eine SMS mit dem Link zur Webapplikation an das Smartphone zu senden. Voraussetzung dafür ist jedoch, dass die Rufnummer des Users bekannt ist.

Basis von Vorhersagen im Browser-Side Cache zu speichern. Für Szenario 1 (Kein bzw. nur kurzzeitiger Internetzugriff vorhanden) ist es allerdings sinnvoller, alle Daten auf dem Client zur Verfügung zu stellen – nicht nur die Resultate.

Ähnlich dem Browser-Side Cache ist das Cache Pattern für Offline HTML 5 Webapplikationen. Das Pattern wurde bei Gmail für mobile Endgeräte implementiert und toleriert unbeständige kabellose Verbindungen. (vgl. Kroeger, 2009)

Abbildung 15 zeigt eine schematische Darstellung einer Applikation, die mit dem Cache Pattern umgesetzt wird. Es wird auf Seiten des Clients, also des mobilen Gerätes, ein Cache zwischen Applikation und Server eingefügt.

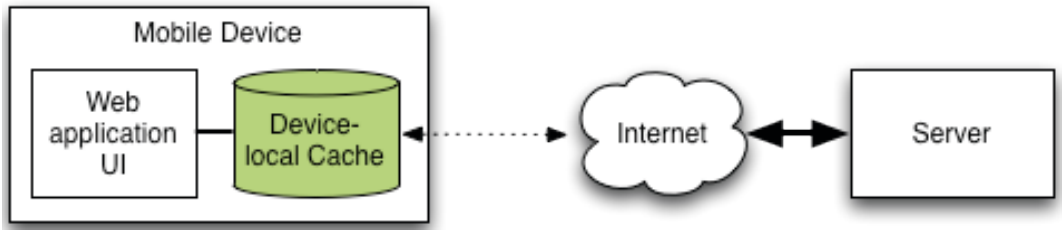

Abbildung 15: Cache Pattern für Offline HTML 5 Webapplikationen (Quelle: Kroeger, 2009)

2.3.10. Multi-Stage Download

2.3.10.1. Problem und Lösung

Besonders wenn große Bilder in Inhaltsdaten enthalten sind, kann die Übertragung viel Zeit kosten. Oft lesen User nicht den gesamten Inhalt oder erkunden einfach die Links einer Seite. Außerdem benötigen sie oft mehrere Klicks, um ein bestimmtes Element aufzufinden. (vgl. Mahemoff, 2006, p. 310ff)

Das Pattern Multi-Stage Download beschreibt, wie man die Download-Performance optimieren kann. Die Inhaltsdaten werden in mehrere Abschnitte aufgeteilt. So können die wichtigeren Daten zuerst übermittelt werden und stehen schneller zur Verfügung. Dabei wird die Webseite in Blöcke unterteilt, deren Inhalte nacheinander geladen werden. Wechselt der User die Seite, wird das Laden der restlichen Inhalte abgebrochen. (vgl. Mahemoff, 2006, p. 310ff)

2.3.10.2. Einsatz im mobilen Informationssystem

Das Multi-Stage Download Pattern ist eine mögliche Art des Ladens in den Local Cache of Data in Szenario 2 (Antwortzeiten verbessern). Sinnvoll kann es aber erst dann eingesetzt werden, wenn große Datenmengen mit bspw. großen Bilddateien angezeigt werden sollen.

2.3.11. Client-Side Dynamic Metadata Design Pattern

2.3.11.1. Problem und Lösung

Gerade bei mobilen Applikationen können EntwicklerInnen nicht auf eine konstante Kommunikation zwischen Client und Server bspw. zum Protokollieren der User-Aktionen bauen. Stamey et al. beschreiben in (2007) mit dem Client-Side Dynamic Metadata Design Pattern eine Lösung zum Dokumentieren der User-Aktionen. Es wird ein Log generiert, mit dem das User-Verhalten auf der Clientseite in einem wohldefinierten Metadatenformat protokolliert werden kann. Abbildung 16 zeigt den Programm-ablauf mit einem implementierten Client-Side Dynamic Metadata Design Pattern.

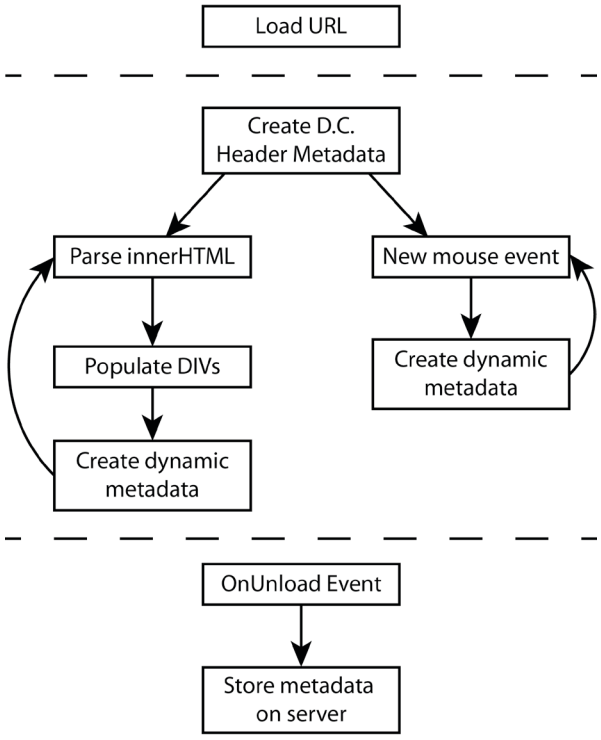

Abbildung 16: Programmablauf mit Client-Side Dynamic Metadata Design Pattern (Quelle: Stamey et al., 2007, p. 159)

Die Informationen über die User-Aktion werden in einem dynamischen Metadatenformat encodiert. Für jede aufzeichenbare Browseraktion und jeden Wechsel des Inhalts werden diese Metadaten an einen String gehangen. Dieser String wird unter Benutzung des JavaScript-Events `onUnload`[19] zum Server gesendet. (vgl. Stamey et al., 2007, p. 158f)

19 Das JavaScript-Event onUnload tritt ein, wenn der User die Webseite verlässt.

2.3.11.2. Einsatz im mobilen Informationssystem

Dieses Pattern eignet sich zum Dokumentieren der History des Users. Allerdings kommt es zu einem Problem, wenn keine Netzwerkverbindung besteht. Hier ist ein Puffern im lokalen Speicher sinnvoll. Sobald wieder eine Netzwerkverbindung besteht, werden die zwischengespeicherten Daten an den Server gesendet.

2.4. Replikationsmethoden

Im folgenden Abschnitt werden zwei Replikationsmethoden erläutert: Master-Slave Datenbank Replikation und gefilterte Replikation.

2.4.1. Master-Slave Datenbank Replikation

Das Master-Slave Prinzip als Replikationskontrollalgorithmus wurde durch C. Ellis (1977) geprägt. Abbildung 17 zeigt ein Master-Slave System. Lediglich eine Datenbank hat in diesem System das Recht, Daten zu lesen und zu schreiben (Read-Write). Alle anderen Replikate haben nur lesenden Zugriff (Read only). In diesem Fall werden alle Änderungen in einer zentralen Datenbank gespeichert und auf die Slaves verteilt. Ein Master-Slave System wird auch als primäre Backup-Replikation oder gerichtete Replikation bezeichnet. (vgl. IBM, 2001; Qui, 2010, p. 22f)

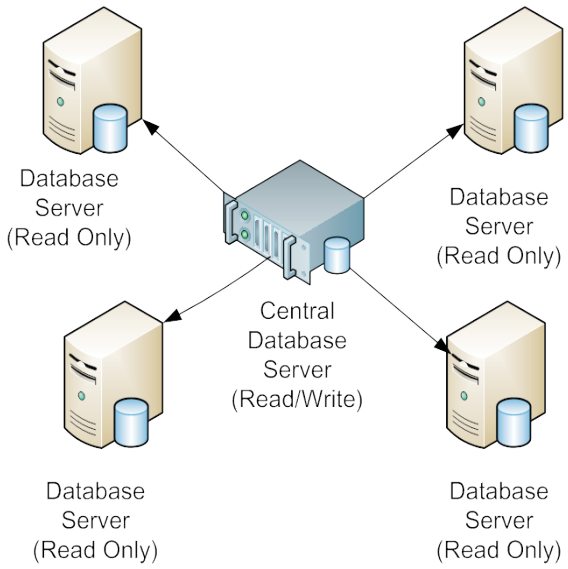

Abbildung 17: Master-Slave Replikationssystem (Quelle: Qui, 2010, p. 23)

Der Local Cache of Documents (Read only) basiert auf dem Master-Slave Prinzip. Die Ressourcen (im speziellen Fall Dateien) werden nur auf dem Server geändert und an die Clients verteilt, sobald die Webanwendung aufgerufen wird. Auch Teile des Local Cache of Data können mit Hilfe des Master-Slave Prinzips umgesetzt werden. Es bietet sich speziell die Replikation von Datenbanken an, die nicht clientseitig geändert werden.

2.4.2. Gefilterte Replikation (Filtered Replication)

Das Ziel der gefilterten Replikation ist, die Items zu speichern, die mit dem Filter übereinstimmen. Filter werden dabei genutzt, um zu beschreiben, welches Daten-Subset in der Replikation gespeichert wird. Ein Filter kann z.B. alle JPEG-Bilder, deren Geo-Tag innerhalb einer bestimmten geografischen Region liegt, selektieren. (vgl. Stuedi et al., 2010)

In Szenario 2 (Antwortzeiten verbessern) kann der Local Cache of Data über gefilterte Replikation implementiert werden. Dabei kann bspw. die Geo-Position als Filter fungieren und alle Daten selektieren, die für die aktuelle Position der BenutzerInnen relevant sind.

2.5. Methoden zur Datenvorhersage

Viele Wissenschaftler befassten sich in den letzten Jahren mit Techniken zur Vorhersage relevanter Daten. Dieser Abschnitt erhebt keinen Anspruch auf Vollständigkeit, das würde den Rahmen dieser Arbeit sprengen. Dennoch werden in den folgenden Unterkapiteln relevante Methoden zusammengefasst.

2.5.1. Pareto Prinzip

Das Pareto-Prinzip geht auf den italienischen Nationalökonom Vilfredo Pareto zurück. Er beobachtete, dass im 19. Jahrhundert 20% der italienischen Bevölkerung 80% des Landes besaßen. (vgl. Laukat, 1999; Pareto, 1935)

Dieser Effekt wurde in der Folge auf diverse wirtschaftliche und natürliche Prozesse übertragen.

> *„In einer beliebigen Menge von Elementen, die etwas bewirken sollen, bewirkt immer eine zahlenmäßig kleine Menge von Elementen den größten Effekt.“*
> (Sombart, 1967)

Firtman (2010, p. 50f) übertrug dieses Prinzip auf die Selektion des Inhaltes zur Erstellung einer mobilen Webseite. 80% der Desktop-Version interessieren den mobilen User nicht. Es kommt somit auf die Auswahl der 20% an, die den User interessieren.

Für die Datensynchronisation bei mobilen Informationssystemen kann das Pareto-Prinzip vor allem in Szenario 2 (Antwortzeiten verbessern) ein Ausgangspunkt zur Selektion der Daten bieten. Es gilt, die 20% der Daten zu selektieren und vorzuladen, die für die BenutzerInnen relevant sind. Die Selektion der Daten kann bspw. über Statistiken der meistbesuchten Inhalte, Vorlieben des Users oder Kontextabhängigkeiten (Zeit, Position etc.) erfolgen.

2.5.2. Web Prefetching

Web Prefetching ist eine Technik, welche die Wartezeit beim Laden einer Webseite reduzieren kann. Mögliche Useranfragen werden innerhalb der Leerlaufzeit zwischen zwei Anfragen vorverarbeitet. Da es sich um eine spekulative Technik handelt, kann sie negativen Einfluss auf die System-Performance haben. Ist die Vorhersage nicht akkurat genug, verbraucht diese Technik zusätzliche Ressourcen wie Netzwerkbandbreite oder Serverrechenzeit. (vgl. de la Ossa, Gil, Sahuquillo & Pont, 2007)

Web Prefetching besteht aus drei Hauptschritten:

1. Basierend auf vorherigen Erfahrungen der Userzugriffe und -vorlieben, die bspw. am Webserver in einer Logdatei protokolliert werden, wird ein Vorhersagemodell für zukünftige Datenzugriffe trainiert. Man erhält Zugriffsmuster.
2. Basierend auf den ermittelten Zugriffsmustern wird ein regelbasiertes Vorhersagemodell (Prediction Model) erstellt.
3. Eine Prefetching Engine mit dem implementierten Prediction Model entscheidet, welche Objekte vorgeladen werden.

(vgl. de la Ossa et al., 2007; Yang, Zhang & Li, 2001)

Die Vorhersage wird meist durch den Webserver erstellt wie in (Padmanabhan & Mogul, 1996). Es besteht jedoch auch die Möglichkeit, die Vorhersage durch den Webbrowser tätigen zu lassen wie in (Zhang, Lewanda, Janneck & Davidson, 2003) oder durch einen zwischen Browser und Server liegenden Proxy wie in (Fan, Cao, Lin & Jacobson, 1999).

Vorhersagealgorithmen werden nach dem Typ der Information, die angesammelt wird, und der Datenstruktur, die für die Vorhersage genutzt wird, klassifiziert. (de la Ossa et al., 2007) Beispiele dafür sind:

1. Beliebtheit des Objektes (Markatos & Chronaki, 1998)
2. Markov-Modell[20] (Padmanabhan & Mogul, 1996)
3. Struktur der Webseite (Davison, 2002)
4. Vorhersage durch teilweise Übereinstimmung (Fan et al., 1999)
5. Datamining[21] (Nanopoulos, Katsaros & Manolopoulos, 2003)
6. Genetische Algorithmen (Bonino, Corno & Squillero, 2003)

Das Vorhersagemodell Top-10 von Markatos & Chronaki (1998) basiert, wie Abbildung 18 zeigt, auf der Kooperation zwischen dem HTTP-Server und einem clientseitigen Vorhersageagenten (Prefetching Agent). Es sagt hauptsächlich statische HTML-Seiten anhand von Statistiken vorher. Der Server errechnet periodisch eine Liste der zehn populärsten Dokumente und macht diese für die Clients erreichbar. Die Clients können daraufhin die Dokumente vorladen. Dabei wird mit Hilfe der Anzahl der Anfra-

20 Das Markov-Modell ist ein Modell in der Stochastik.

21 Die Auswertung größerer Datenmengen und die gezielte, themenbezogene Datensuche wird als Data Mining bezeichnet.

gen eines Clients gewährleistet, dass die Liste nur den Clients zur Verfügung gestellt wird, die sie potentiell benötigen. Die Studie zeigt, dass Top-10 bei einer 20%-igen Erhöhung des Traffics bis zu 60% der Anfragen vorhersagen kann.

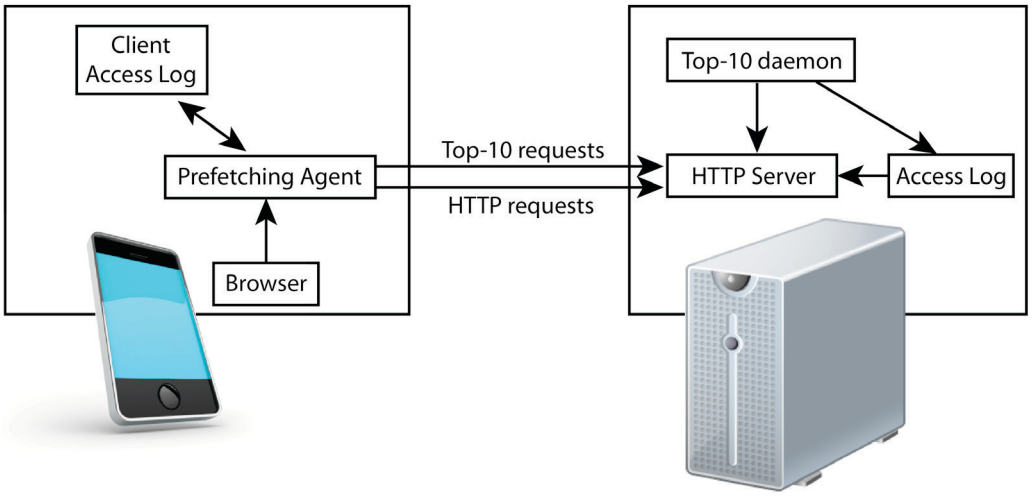

Abbildung 18: Architektur des Top-10 Vorhersagemodells (Quelle: Markatos & Chronaki, 1998, modifiziert)

In (Zhimei Jiang & Kleinrock, 1998) untersuchen die Autoren Online Prefetching in Echtzeit. Während der User surft, werden die Seiten, die er am ehesten in der nahen Zukunft anfordern wird, nach bestimmten Kriterien geladen. Das Problem wird in zwei Schritten bewältigt:

- Ermitteln der Wahrscheinlichkeit, mit der jede Datei aufgerufen wird
- Bestimmen, ob das Vorladen dieser Datei kosteneffektiv ist

Für den Server, auf dem die Datei abgelegt ist, kann ein Prefetch Threshold berechnet werden. So kann sichergestellt werden, dass die durchschnittliche Verzögerung, die durch das Vorladen entsteht, garantiert reduziert wird, solange die Zugriffswahrscheinlichkeit größer ist als der Prefetch Threshold des Servers.

Mobasher et al. (2001) vergleichen mit ARM, einer Empfehlungsmaschine (Recommendation Machine), die aktive User Session mit häufig verwendeten Itemsets in einer Datenbank und sagen so die Seite, die der User als nächstes besuchen möchte, vorher. Diese Vorhersage ist regelbasiert.

De la Ossa et al. (2007) stellen in ihrem Paper Vorhersagen beim Vorladen (Prediction at Prefetch) vor. Dabei werden nicht wie bisher in der Literatur verankert nur Vorhersagen für die Seiten erstellt, die von den NutzerInnen aktiv angefordert werden, sondern auch für die vorhergesagten Seiten. Die Ergebnisse zeigen, dass bei einer richtig konfigurierten Prediction Engine ein gutes Kosten-Nutzen-Verhältnis erreicht wird. Die Wartezeit konnte bis zu 14% reduziert werden, während der Byte-Traffic gleichzeitig um 8% stieg.

2.5.2.1. Semantische Vorhersage

In (Cheng-Zhong Xu & Ibrahim, 2003) entwickelten die Autoren eine semantisch basierte Vorhersagetechnik. Diese sagt zukünftige Anfragen auf Basis der semantischen Präferenzen der zuletzt erhaltenen Dokumente vorher. Im Gegensatz dazu steht das meist temporäre Verhältnis zwischen URL-Zugriffen. Diese Vorhersagetechnik beachtet bspw. Stichwörter in Anker-Texten von URLs.

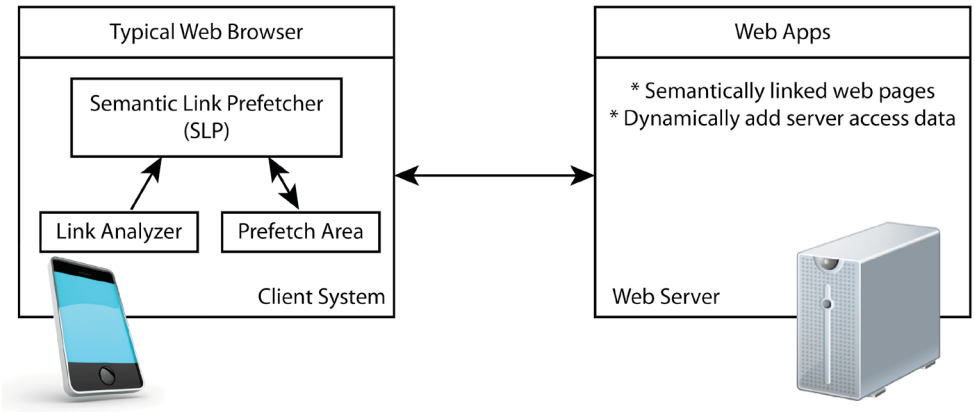

Abbildung 19: Prefetcher System Scheme (Quelle: Pons, 2006, modifiziert)

Auch Pons beschäftigte sich mit der auf Semantik basierenden Vorhersage. In (Pons, 2006) wird der Semantic Link Prefetcher (SLP) beschrieben. Abbildung 19 zeigt die Struktur eines von SLP unterstützten Systems. SLP ist als Komponente des Webbrowser am Client umgesetzt. Dieser nutzt Informationen, die mit den Hyperlinks der aktuellen Webseite verbunden sind, um vorherzusagen, welches Webobjekt vorgeladen werden soll. Eine vom Browser empfangene Webseite enthält ein Set von Hyperlinks, die der Client aufrufen könnte. Daraus identifiziert der Browser diejenigen Links, die er vorladen wird. Durch die Nutzung von semantischen Links werden dem Browser Informationen zum jeweiligen Link geliefert. So kann durch Einführung einer Rangordnung clientseitig entschieden werden, welche Links am wahrscheinlichsten als nächstes durch den User aufgerufen werden. Diese werden anschließend vorgeladen.

2.5.2.2. Dominant Pattern Graph

Khan & Qingping Tao analysierten die Strukturen von Webseiten. In Abbildung 20 bis Abbildung 23 sind die Muster dargestellt, die Khan & Qingping Tao entdeckten.

Abbildung 20: Chain-Muster
(Quelle: Khan & Qingping Tao, 2003a)

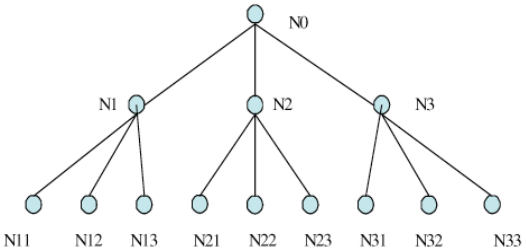

Abbildung 22: Tree-Muster
(Quelle: Khan & Qingping Tao, 2003a)

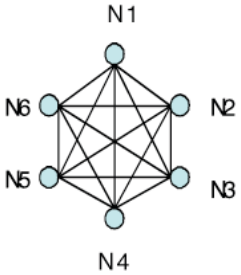

Abbildung 21: Complete Graph-Muster
(Quelle: Khan & Qingping Tao, 2003a)

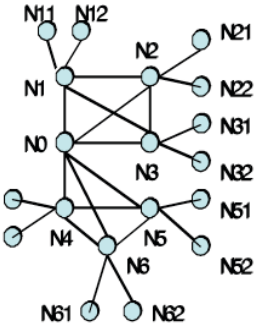

Abbildung 23: Tree with complete core-Muster
(Quelle: Khan & Qingping Tao, 2003a)

Werden diese Pattern in Vorladealgorithmen integriert, kann die Vorhersage wesentlich verbessert werden. Diese Aussagen stützen Khan & Qingping Tao mit der Entwicklung des Dominant Pattern Graph (DPG), einem Web Surfing Pattern. Jeder Link einer Webseite bekommt ein Attribut, das ihn innerhalb des DPG identifiziert. Außerdem gibt es ein `no prefetch`-Attribut, das zum Stoppen des Vorlademechanismus genutzt werden kann. (vgl. Khan & Qingping Tao, 2003a, 2003b)

2.5.2.3. Dependency Graph

Padmanabhan & Mogul (1996) beschreiben ein Vorhersagemodell, dass auf Serverwissen basiert. Es baut auf einen Algorithmus von Griffioen & Appleton (1994) auf. Das Vorhersagemodell konstruiert einen Abhängigkeitsgraphen (Dependency Graph, DG - siehe Abbildung 24), der das Muster der Zugriffe auf die verschiedenen Dateien darstellt. Ein Graph besteht aus Knoten, die jeweils eine Datei repräsentieren, sowie die Verbindungen zwischen zwei Knoten A und B, wenn B innerhalb einer bestimmten Zeitspanne nach A angefordert wurde. Die Verbindungen werden gewichtet.

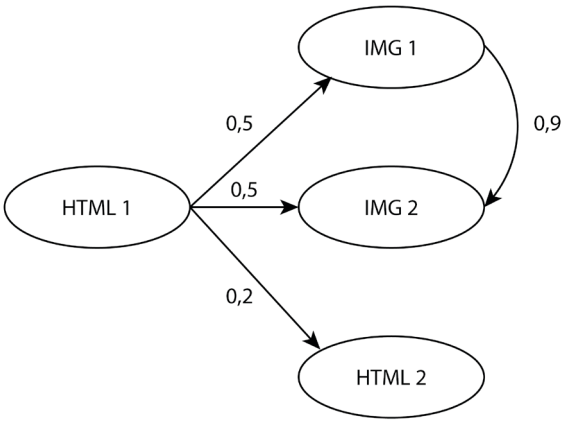

Abbildung 24: Abhängigkeitsgraph nach Padmanabhan & Mogul (Quelle: Padmanabhan & Mogul, 1996, p. 27, modifiziert)

Der Graph in Abbildung 24 zeigt: Wenn `HTML 1` aufgerufen wird, ist die Verbindung zu `IMG 1` und `IMG 2` mit 0,5 gewichtet. Wird `IMG 1` auf `HTML 1` folgend aufgerufen, ist die Verbindung zu `IMG 2` mit 0.9 gewichtet.

Der Abhängigkeitsgraph wird dynamisch vom Server aktualisiert, sobald der Server neue Anfragen erhält. Wird Knoten A aufgerufen, sendet der Server einen Hinweis zum Client, damit dieser Knoten B vorladen kann, wenn die Verbindung zwischen A und B stark gewichtet ist. Dann besteht eine gute Chance, dass B auch wirklich angefordert wird.

Mit Hilfe einer Simulation wurde dieses Vorhersagemodell getestet. Es wurden zwei Performance-Metriken gemessen – die durchschnittliche Zugriffszeit pro Datei und die partielle Erhöhung des Netzwerk-Traffic. Padmanabhan & Mogul fanden heraus, dass die durchschnittliche Zugriffszeit um 50% reduziert wurde, der Netzwerk-Traffic jedoch um 80% anstieg. Ein weiteres Ergebnis ihrer Untersuchungen war, dass aggressives Vorladen keinen signifikanten Einfluss auf die Zugriffszeit hat.

2.5.2.4. Double Dependency Graph

In (Domenech, Gil, Sahuquillo & Pont, 2006) erzeugten die Autoren unter Berücksichtigung der Charakteristiken des Webs den doppelten Abhängigkeitsgraphen (Double Dependency Graph, kurz DDG). Die Ausgangssituation stellte sich wie folgt dar: Aufgrund der großen Anzahl von eingebetteten Objekten im Web sind die meisten existierenden Vorhersagealgorithmen nutzlos, weil diese hauptsächlich die Objekte, die im HTML eingebettet sind vorhersagen, nachdem auf das HTML selbst zugegriffen wurde. Diese Vorhersagen sind nutzlos für den Client, weil keine Wartezeitreduktion erfolgt.

DDG unterscheidet zwischen Containerobjekten (HTML) und eingebetteten Objekten (z.B. Bildern), um ein Vorhersagemodell zu erstellen und die Vorhersage zu tätigen. Abbildung 25 zeigt einen mit dem DDG-Algorithmus erzeugten Graphen.

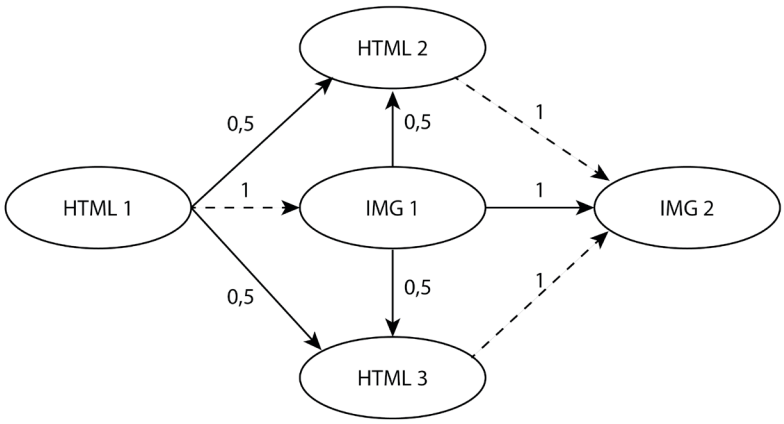

Abbildung 25: Graph eines DDG-Algorithmus (Quelle: Domenech et al., 2006)

Wie auch bei DG (vgl. Abbildung 24) besteht der Graph aus Knoten, welche die Dateien darstellen, und Verbindungen. Jedoch wird zwischen primärer und sekundärer Verbindung unterschieden. Eine Verbindung ist primär, wenn die Knoten A und B Objekte verschiedener Seiten sind. Besteht kein HTML-Zugriff zwischen A und B, ist die Verbindung sekundär und wird im Graph mit einer gestrichelten Linie dargestellt. Die Wahrscheinlichkeit jeder primären und sekundären Verbindung wird durch Dividieren des Verbindungszählers durch die Anzahl des Vorkommens des Knotens bestimmt. (vgl. Domenech et al., 2006)

Die Ergebnisse in (Domenech et al., 2006) zeigen, dass DDG bei gleicher Anzahl an Serveranfragen immer die existierenden Algorithmen bei der Reduzierung der Wartezeit zwischen 15 und 150% übertrifft.

2.6. Entwicklung für mobile Endgeräte

Werden Anwendungen für mobile Endgeräte entwickelt, steht zu Beginn die Entscheidung über die Zielplattform an. Dabei stehen zwei grundsätzliche Ausprägungen zur Verfügung – Webapplikation oder native Applikation. Als dritte Optionen gibt es die Möglichkeit einer hybriden Anwendung. In diesem Abschnitt werden die Vor- und Nachteile dieser Varianten erläutert.

2.6.1. Webapplikation

Eine Webapplikation ist eine Anwendung, die mit einem Webbrowser über das Internet (World Wide Web, kurz WWW) aufgerufen wird. Abbildung 26 zeigt den typischen Ablauf einer Anwendung, die in der Programmiersprache PHP[22] geschrieben wurde. Diese Sprache kann nicht am Client ausgeführt werden. Sie muss am Server durch einen Interpreter in bspw. HTML gewandelt werden. Der Server verarbeitet bei Anfrage (1) einer entsprechenden Datei die Daten (2) und erzeugt im PHP-Parser (3) den Code je nach Skript (meist HTML, 4). Anschließend liefert der Webserver das erzeugte Skript an den Client (Webbrowser) aus (5). (vgl. Benson et al., 2010, p. 121f; Jazayeri, 2007; Wikimedia Commons, 2009)

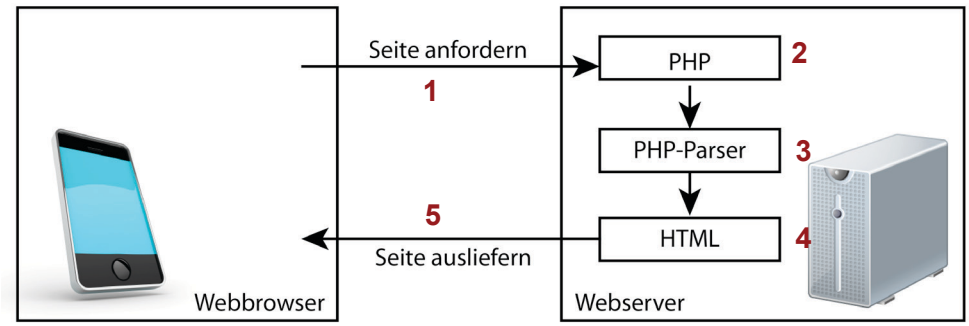

Abbildung 26: Funktionsweise von PHP
(Quelle: Thies & Reimers, 2009, p. 28, modifiziert)

Mobile Webapplikationen, die am Client lauffähig sein sollen, müssen in HTML, CSS und JavaScript geschrieben werden. Sie werden nicht auf dem Gerät installiert, ein Aufruf der entsprechenden URL genügt. (vgl. Fling, 2009, p. 75) Eine Webanwendung ist demnach nur so gut, wie der Browser, in dem sie aufgerufen wird. (vgl. Spiering & Haiges, 2010, p. 10) Die Browser wiederrum basieren auf einer Layout Engine. In Abbildung 27 ist ersichtlich, dass die Nutzung zwei mobiler Browser (Android Browser und Mobile Safari (iPhone)) gegenüber der Konkurrenz deutlich bevor-

22 Die Skriptsprache PHP wird hauptsächlich zur Programmierung von dynamischen Webseiten verwendet.

zugt wird. Diese beiden Browser basieren auf der WebKit[23] Layout Engine. (vgl. Firtman, 2010, p. 7)

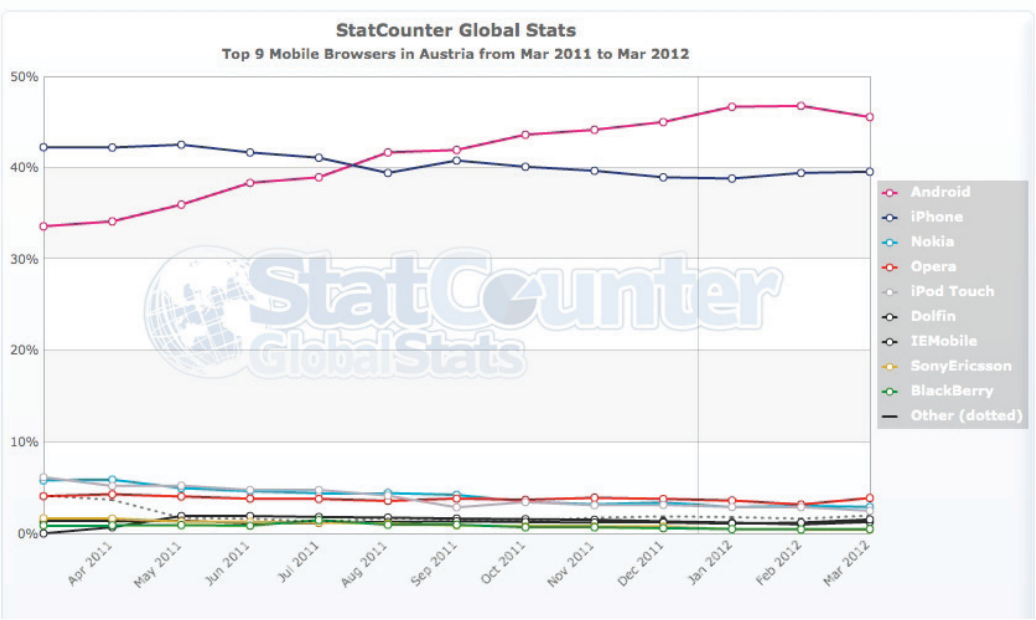

Abbildung 27: Nutzungsstatistik mobiler Browser in Österreich von März 2011 bis März 2012 (Quelle: StatCounter, 2012a)

Ein Nachteil von Webapplikationen ist bisher, dass nicht auf alle Funktionen des Gerätes zugegriffen werden kann. Nutzbar sind nur jene Funktionen, für die der jeweilige Browser eine Schnittstelle bietet. Nach einem White Paper der Global Intelligence Alliance aus dem Jahr 2010 soll es bis 2013 möglich sein, auf die in Abbildung 28 angeführten Funktionen zugreifen zu können. Bis Mai 2012 war es jedoch nur möglich auf die Geolocation und den Accelerometer (Motion detection, jedoch nur im Mobile Safari) zu zugreifen.

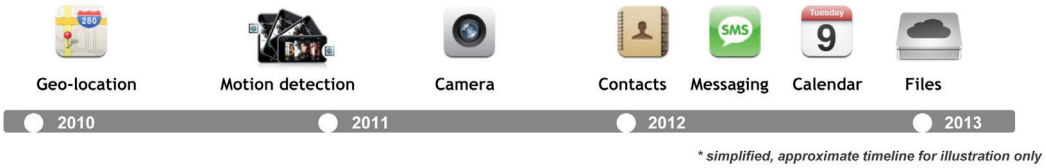

Abbildung 28: Vorhersage für den Zugriff auf Gerätefunktionen durch Webanwendungen bis 2013 (Quelle: Global Intelligence Alliance, 2010, p. 7)

Eine ausführliche Behandlung der Vor- und Nachteile von Webanwendungen findet in Kapitel 2.6.4 statt.

23 WebKit ist eine Web Content Engine für Browser und andere Applikationen. WebKit nutzt standardisierte Technologien wie HTML, CSS, JavaScript und DOM. Die Ziele des Open Source-Projektes sind: Standard-Kompatibilität, Stabilität, Performance, Sicherheit, Portabilität, Usability und einfacher Code. (vgl. Mac OS Forge, n.d.)

2.6.2. Native Applikation

Eine native Applikation läuft nicht im Browser eines Mobiltelefons, sondern als eigenständiges Programm. Sie muss vor der Nutzung heruntergeladen und installiert werden. Die native Applikation ist für ein bestimmtes Betriebssystem (bspw. iOS, Android OS oder Windows Phone) geschrieben und in derselben Form nicht auf einem anderen Betriebssystem lauffähig. (vgl. Spiering & Haiges, 2010, p. 8f)

Native Applikationen für iOS werden bspw. in der Programmiersprache Objective-C geschrieben. Für das Betriebssystem Android OS benötigen EntwicklerInnen Kenntnisse in der Programmiersprache Java.

Native Applikationen werden in der Regel über einen App Store (iTunes bei iOS, Google Play[24] bei Android oder Marketplace bei Windows Phone) vertrieben. Dabei müssen die Applikationen für den jeweiligen Store eingereicht werden und einen Genehmigungsprozess durchlaufen.

Einer der Vorteile für die NutzerInnen von nativen Applikationen ist das Auffinden der Anwendung: Der User ruft auf seinem Smartphone die entsprechende App Store-Anwendung auf und kann in dieser einfach die bevorzugte Applikation finden. Mit wenigen Klicks ist diese auf dem Gerät installiert. Die EntwicklerInnen hingegen haben bei der Programmierung nativer Anwendung, die Möglichkeit auf alle Funktionen des Gerätes zuzugreifen. Eine Auflistung von Vor- und Nachteilen nativer Applikationen ist in Kapitel 2.6.4 zu finden.

2.6.3. Hybride Applikation

Der Kern einer hybriden Anwendung ist in HTML, CSS und JavaScript geschrieben. Dieser Code wird durch eine WebView in nativen Code eingebunden. Die WebView-Klasse (in iOS bspw. UIWebView) beinhaltet die Basisfunktionalität eines Browsers (z.B. Seitendarstellung, Ausführung von JavaScript). (vgl. Luo, Hao, Du, Wang & Yin, 2011, p. 343) Dadurch kann die Funktionalität und Darstellung der Anwendung mit Webtechnologien geschrieben werden.

Während Webanwendungen auf die Schnittstellen des Gerätebrowsers angewiesen sind, um auf Gerätefunktionalitäten zuzugreifen und native Applikationen vollen Zugriff auf diese Funktionen haben, ist es aus der WebView heraus möglich, auf Funktionen des Gerätes zuzugreifen, die der Webbrowser des Gerätes nicht zur Verfügung stellt.

Eine hybride Applikation muss wie eine native Applikation vor der Nutzung auf das Gerät heruntergeladen und installiert werden. Der Vertrieb erfolgt ebenfalls über App Stores.

24 ehemals Google Market

Ein verbreitetes Framework, das bei der Erstellung hybrider Applikationen unterstützt ist Phonegap[25] von Adobe. PhoneGap stellt die WebView und ein JavaScript API zur Verfügung. Dieses JavaScript API ermöglicht es, auf gerätespezifische Funktionen (bspw. Kamera, Benachrichtigung, Dateisystem) zuzugreifen, die nicht vom Browser selbst unterstützt werden. Derzeit wird die Distribution auf sieben Plattformen (iOS, Android, Blackberry, WebOS, Windows Phone 7, Symbian, Bada) unterstützt. (vgl. PhoneGap, 2012)

2.6.4. Vergleich der drei Ansätze

Tabelle 1 zeigt eine Gegenüberstellung von Web, hybrider und nativer Applikation. Vorteile sind mit Grün hinterlegt, Nachteile mit Rot. Keine Hintergrundfarbe bedeutet, dass diese Aussage weder als Vorteil noch als Nachteil gesehen werden kann. (vgl. Charland & LeRoux, 2011, p. 50f; Kern, 2012; Schmiedl, 2010)

Tabelle 1: Gegenüberstellung von Web, hybrider und nativer Applikation

Webapplikation	**Hybride Applikation**	**Native Applikation**
Kein eigenständiges Vertriebsmodell, für kostenpflichtigen Vertrieb selbst entwickeln und warten	Eigenständiges Vertriebsmodell (App Store) für jede Plattform	Eigenständiges Vertriebsmodell (App Store) für jede Plattform
Zugriff nur auf Funktionalität, die der Webbrowser zur Verfügung stellt	Zugriff auf fast alle Gerätefunktionen	Zugriff auf alle Gerätefunktionen
Für Webbrowser geschrieben und damit Cross-Plattform einsetzbar	Für Webbrowser geschrieben und damit mit Modifizierungen Cross-Plattform einsetzbar	Für genau eine Geräteklasse geschrieben
Der selbe Code läuft auf allen Plattformen, welche die entsprechende Funktionalität unterstützen	Grundsätzlich der selbe Code, muss aber dennoch für die Unterstützung von mehreren Plattformen für jede separat kompiliert werden	Hoher Aufwand für Unterstützung von mehreren Plattformen
Keine Installation erforderlich	Installation erforderlich	Installation erforderlich

25 http://phonegap.com/

Webapplikation	**Hybride Applikation**	**Native Applikation**
Suchmaschinen können Informationen erfassen	Suchmaschinen können Informationen, die es nur innerhalb der App gibt, nicht erfassen, aber durch die Verwendung von HTML, CSS und JavaScript, ist es leicht möglich eine Webanwendung zu erstellen	Suchmaschinen können Informationen, die es nur innerhalb der App gibt, nicht erfassen
Finden über Suche oder Wissen der URL	Finden über App Store der Plattform und über Websuche (mit Verlinkung zum entsprechenden App Store)	Finden über App Store der Plattform und über Websuche (mit Verlinkung zum entsprechenden App Store)
Höherer Aufwand, um Bedienkonzepte und Empfehlungen des mobilen Plattformanbieters zu erreichen, weil Gestaltung der Oberfläche über HTML, CSS, JavaScript	Höherer Aufwand, um Bedienkonzepte und Empfehlungen des mobilen Plattformanbieters zu erreichen, weil Gestaltung der Oberfläche über HTML, CSS, JavaScript	Zur Gestaltung der Bedienoberfläche können native Elemente genutzt werden, die für die Bedienkonzepte der jeweiligen Plattform optimiert sind
Speicherung in Webspace oder lokale Speicherung im Browser	Sowohl Speicherung von Daten in lokalen Datenbanken und Dateisystemen als auch Speicherung in Webspace oder lokale Speicherung im Browser möglich	Speicherung von Daten in lokalen Datenbanken und Dateisystemen
Kommunikation über HTTP als Web-Protokoll	Kommunikation über HTTP als Web-Protokoll, aber grundsätzlich auch Kommunikation über In-/ Output Bibliotheken der jeweiligen Plattform möglich	Kommunikation über In-/ Output Bibliotheken der jeweiligen Plattform

Ein oft erwähntes Argument für native Applikationen ist die bessere Performance. Charland & LeRoux (2011, p. 49) sind der Meinung, dass dies für 3D-Spiele und Bildverarbeitung zutrifft. Bei mit Webtechnologien gut programmierten Geschäftsapplikationen ist jedoch nur ein unbedeutender oder nicht wahrnehmbarer Performance-Verlust zu verzeichnen.

Abbildung 29 zeigt eine Studie der Firmen phaydon, research+consulting GmbH, denkwerk und Interrogare (2010). In der Gunst der Smartphone NutzerInnen sind Webapplikationenen (Roter Balken) und native Applikation (Blauer Balken) nahezu gleich auf. Bei Spielen liegt die native Applikation mit 40% gegenüber 33% bei der Webanwendung vorn. Suchmaschinen und Wikipedia werden jedoch von 56% der NutzerInnen über den Browser aufgerufen. Dem gegenüber stehen 46%, die Suchmaschinen und Wikipedia über die jeweiligen nativen Applikationen nutzen.

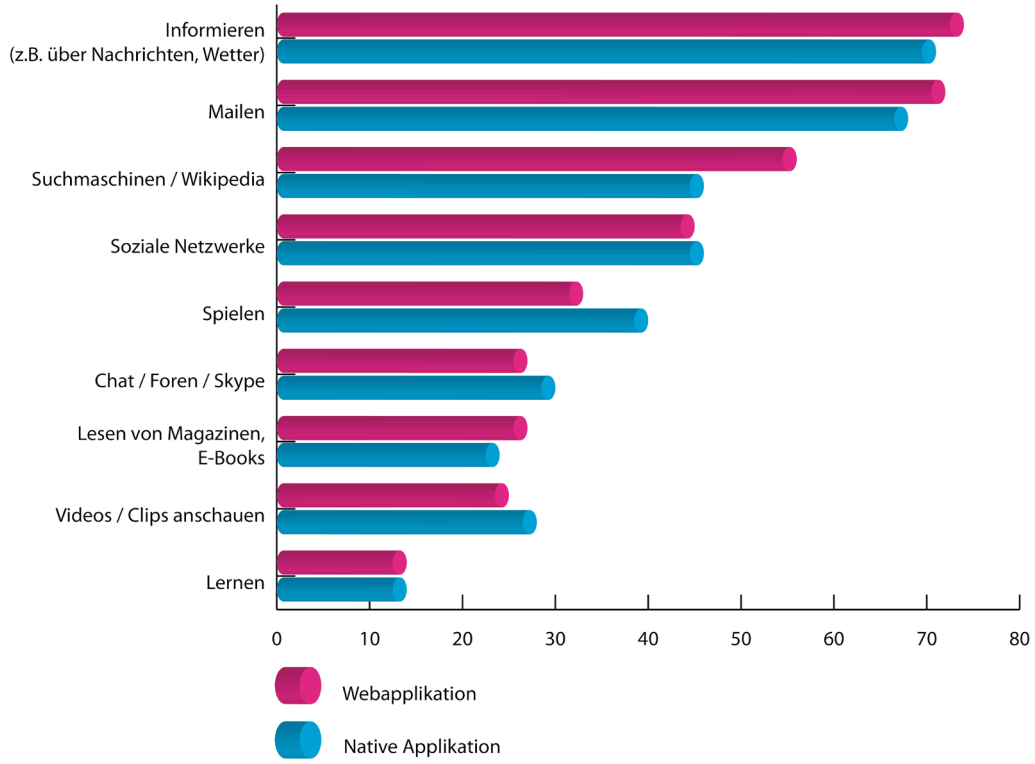

Abbildung 29: Nutzung von Smartphone-Funktionen über Apps oder Browser (Quelle: phaydon et al., 2010, modifiziert)

2.7. Clientseitige Webtechnologien

Da sich diese Arbeit mit den Möglichkeiten einer Datensynchronisation von mobilen Informationssystemen mit Webtechnologien beschäftigt, werden in diesem Abschnitt die zur Verfügung stehenden clientseitigen Technologien vorgestellt.

2.7.1. HTML

Bereits 1980 schlägt Tim Berners-Lee, ein Physiker, der bei der European Organisation for Nuclear Research (CERN) in Genf (Schweiz) arbeitet, eine erste Möglichkeit namens ENQUIRE vor, um Dokumente zu nutzen und zu teilen. ENQUIRE war bereits ein Hypertext-Programm. Man konnte Textdateien editieren, die in „nodes“ (Knoten) unterteilt waren. Zu jedem Knoten gab es eine Liste mit Links zu anderen Knoten. (vgl. Berners-Lee, 2011; SELFHTML, 2007a)

Neun Jahre später erhielt das Projekt die Unterstützung des CERN und seinen endgültigen Namen „World Wide Web“. Im Herbst 1990 schrieb Berners-Lee daraufhin die ersten Versionen der drei Säulen seines Konzepts (vgl. SELFHTML, 2007a):

- HTTP: Spezifikation für die Kommunikation zwischen Webclients und Webservern,
- URI[26]: Spezifikation für die Adressierung beliebiger Dateien und Datenquellen im Web und im übrigen Internet
- HTML[27]: Spezifikation einer Auszeichnungssprache für Webdokumente

Problemstellung für Berners-Lee bei der Entwicklung von HTML ist damals, wie Forschungsdokumente über ein Hypertext-System gemeinsam nutzbar gemacht werden können. HTML ist die 1986 veröffentlichte ISO-Norm 8879 zu Grunde gelegt. Hierbei handelt es sich um die Standard Generalized Markup Language, kurz SGML, eine computerlesbare Metasprache zum Formulieren von Regeln von Markup-Sprachen wie HTML. (Connolly, 2004; Robbins, 2008, p. 165)

1992 wird die erste Spezifikation für HTML veröffentlicht. HTML geht von einer hierarchischen Gliederung aus. Bereits in der Urversion ist jedes Element durch einen Anfangs-Tag (Opening Tag) mit Tagöffner (<) und Tagschließer (>) definiert. Einige der Elemente benutzen End-Tags (Closing Tags), die durch einen Slash (/) nach dem Tagöffner kenntlich gemacht sind. (vgl. Berners-Lee, 1992; Münz, 2008, p. 67; SELFHTML, 2007b)

Seither wurde eine Reihe von HTML-Versionen veröffentlicht. Die aktuell standardisierte Version besitzt die Nummer 4.01.

HTML 5 befindet sich derzeit im Stadium eines Working Drafts. In HTML 5 wird eine eigene Syntax formuliert, die sich am Document Object Model, kurz DOM, orientiert.

26 Universal Resource Identifier (kurz URI), universeller Quellenbezeichner

27 Hypertext Markup Language (kurz HTML)

Daraus entsteht DOM HTML. Während das allgemeine DOM ein Dokumentmodell für XML-Sprachen definiert, formuliert DOM HTML ein Dokumentmodell speziell für HTML. (vgl. Hickson & Hyatt, 2008; Münz, 2008, p. 340) Am 29. März 2012 wurde die zwölfte Version des Working Drafts von HTML 5 veröffentlicht. (vgl. Hickson, 2012a)

In HTML 5 werden neue Elemente für die inhaltliche Struktur der Dokumente eingeführt: `section`, `article`, `aside`, `header`, `footer` und `nav`. Mit `figure`, `audio`, `video`, `embed` und `canvas` werden Elemente zur Einbindung von eingebetteten (Multimedia-)Inhalten angeboten. In den neuen Elementen `details`, `summary`, `command` und `menu` ist eine stärke Orientierung von HTML 5 in Richtung Scripting und Programmierung zu erkennen. Für die Auszeichnung von Inhalten stehen in der neuen Version zusätzlich die Elemente `time`, `progress` und `meter` zur Verfügung. (vgl. Hickson, 2012a; Münz, 2008, p. 342ff)

HTML 5 bietet auch neue Möglichkeiten, um Anwendungen ohne Netzwerkverbindung auszuführen. Bisher benötigte man bspw. Browser-Plugins wie Google Gears (siehe Kapitel 2.2.2), um diese Funktionalität umzusetzen. Diese Möglichkeiten werden im folgenden Abschnitt näher beschrieben.

2.7.1.1. Cache Manifest[28]

In der Cache Manifest-Datei wird gespeichert, welche Ressourcen für die Webseite benötigt werden. Listing 2 zeigt ein Beispiel einer Cache Manifest-Datei.

Listing 2: Beispiel einer Cache Manifest-Datei

```
1 CACHE MANIFEST
  ./content/detail.html
  ./js/mobilot.js
  ./css/mobilot.css
  ./image/iconset.png
2 NETWORK:
  service.php
3 FALLBACK:
  ./image/ ./offline.jpg
```

Eine Cache Manifest-Datei muss mit `CACHE MANIFEST` (1) beginnen. Darauf folgen die Dateien, die lokal gespeichert werden sollen (Explicit Section). Außerdem kann im Bereich `NETWORK` (2) spezifiziert werden, welche Dateien nicht gespeichert (Online Whiteliste Section) und somit immer vom Server geladen werden. Zudem kann der Bereich `FALLBACK` (3) definiert werden, in dem zwischen Online- und Offline-Verfügbarkeit unterschieden wird (Fallback Section). In Listing 2 werden bei einer verfügbaren Netzwerkverbindung die Dateien im Ordner `image` geladen. Steht keine Verbindung zur Verfügung, werden die Dateien des Ordner `image` mit der Datei `offline.jpg` ersetzt.

28 Ausschnitte dieses Kapitels wurden bereits in (Blumenstein, 2012) veröffentlicht.

Die Manifest-Datei muss vom Server mit dem Content-Type `text/cache-manifest` ausgeliefert werden. Außerdem muss in der Startdatei der Webseite im HTML-Tag die Manifest-Datei verlinkt werden. Dazu wird das Attribut `manifest` gesetzt (siehe Listing 3).

Listing 3: Integration der Manifest-Datei in HTML

```
<html manifest="cache.appcache">
```

Ist das Attribut `manifest` gesetzt, werden alle GET-Anfragen zum lokalen Speicher umgeleitet. POST, PUT und DELETE werden weiterhin über das Netzwerk ausgeführt. Der Browser aktualisiert die Dateien im Manifest immer dann, wenn sich in der Manifest-Datei etwas ändert. So ist es bspw. möglich, über eine Versionsnummer das Aktualisieren der gespeicherten Dateien zu steuern.

(vgl. Hickson, 2012b; Kessin, 2011, p. 76; Spiering & Haiges, 2010, p. 227ff)

2.7.2. JavaScript

> *„Der als ECMA[29]Script (ECMA262) standardisierte Sprachkern von JavaScript beschreibt eine dynamisch typisierte, objektorientierte aber klassenlose Skriptsprache."*
> (Wikipedia, 2012)

1995 wurde von Netscape in der Version 2.0 ihres Browsers eine Sprache namens LiveScript eingeführt. Es war damit möglich, bspw. Formulareingaben vor dem Absenden zu prüfen. Die Syntax war an Java angelehnt. Laut Wenz (2010, p. 22f) wurde die Sprache aus marketingtechnischen Gründen JavaScript getauft. Mit JavaScript 1.1 war es bereits möglich, auf Bilder zuzugreifen und RollOver-Grafiken zu gestalten.

Auch Microsoft, das in den Jahren des Browserkrieges größter Konkurrent von Netscape war, bot in der Version 3 des Internet Explorers (1996) erstmals Skriptunterstützung. Aus lizenzrechtlichen Gründen wurde diese Sprache JScript genannt, wobei die Syntax praktisch identisch mit JavaScript war.

Es wird hauptsächlich clientseitig verwendet. Jedoch ist mittlerweile auch der Trend zum serverseitigen JavaScript zu bemerken. Ein Beispiel hierfür ist Node.js[30].

JavaScript läuft im Browser als Sandbox. Es ist also nur möglich, auf Objekte des Browsers zuzugreifen, nicht etwa auf das Dateisystem des Rechners.

(vgl. Kessin, 2011, p. 4f; Wenz, 2010, p. 22ff; Wikipedia, 2012)

29 European Computer Manufacturers Association (kurz ECMA)

30 http://nodejs.org/

2.7.2.1. JSON

Die JavaScript Object Notation (kurz JSON) ist ein kompaktes Format zum Datenaustausch. JSON basiert auf einer Teilmenge von JavaScript. Da es sich um ein Textformat handelt, kann es leicht von Menschen gelesen und geschrieben werden. Auf der anderen Seite ist es jedoch auch für Maschinen einfach zu parsen und zu generieren. JSON kann mit sechs Werten geschrieben werden:

1. Objekte,
2. Arrays,
3. Strings,
4. Zahlen,
5. boolsche Werte (`true` und `false`) sowie
6. der Spezialwert `null`.

Außerdem kann Whitespace (Leerzeichen, Tabulatoren, Carriage-Return und Newline-Zeichen) eingefügt werden, um bspw. die Lesbarkeit zu erhöhen.

Listing 4 kann man entnehmen, dass ein JSON-Objekt ein ungeordneter Container bestehend aus Schlüssel/Wert-Paaren ist. Der Schlüssel kann ein beliebiger String und der Wert ein beliebiger JSON-Wert sein, der selbst auch wieder ein JSON-Objekt sein kann.

(vgl. Crockford & Klicman, 2008, p. 146ff; JSON, n.d.-a, n.d.-b; Rieber, 2009, p. 658)

Listing 4: JSON-Objekt

```
var json = {
   mobidulName  :  „Kerstin's DA“,    // String
   customNav    :  [                  // Array von JSON-Objekten
      {
         text  :  „Über Mobilot“,     // String
         count :  3,                  // Zahl
         flag  :  true                // Boolscher Wert
      },
      {
         text  :  „Enter Code“,       // String
         count :  1,                  // Zahl
         flag  :  false               // Boolscher Wert
      }
   ]
}
```

Ursprünglich sollte ein JSON-Objekt mit Hilfe der `eval`[31]-Funktion eingelesen werden. (vgl. JSON, n.d.-b) Dies ist jedoch sehr unsicher, weil somit jegliches JavaScript direkt ausgeführt und leicht Code eingeschleust werden kann.

31 Die Funktion `eval` evaluiert einen String als JavaScript und führt dieses JavaScript aus.

Mittlerweile kann JSON in allen aktuellen Browsern geparst werden, ohne eine zusätzliche Bibliothek zu integrieren. Dies erfolgt über die Funktionen `JSON.parse` (String in JSON-Datenstruktur wandeln) und `JSON.stringify` (JSON-Datenstruktur in String wandeln). (vgl. Deveria, 2012a)

2.7.3. AJAX[32]

Mit HTML und CSS sind nur unzureichende Möglichkeiten zur Gestaltung von NutzerInneninteraktion vorhanden. Mit JavaScript hingegen lassen sich Inhalte generieren und nachladen. Zentrales Element ist dabei das `XMLHttpRequest`-Objekt und die Bezeichnung AJAX.

AJAX ist keine eigenständige Technologie, vielmehr vereint es mehrere Technologien:

- HTML und CSS,
- Document Object Model (DOM),
- XML und XSLT,
- XMLHttpRequest und
- JavaScript.

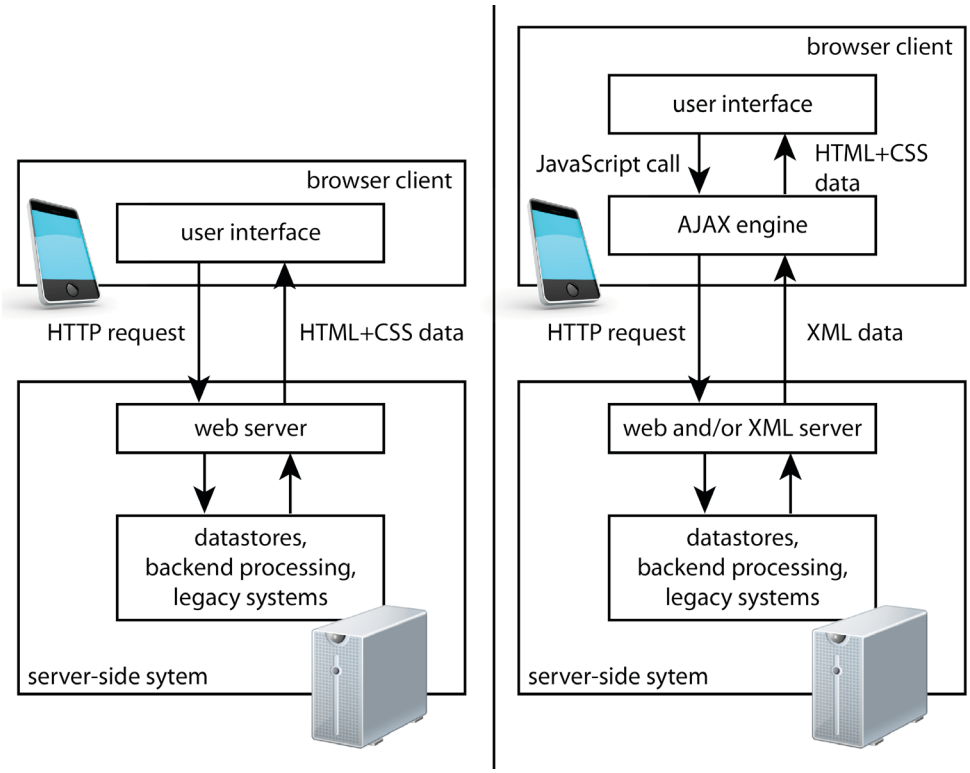

Abbildung 30: Traditionelles Modell von Webanwendungen (links) verglichen mit dem AJAX-Modell (rechts) (Quelle: Garrett, 2005 modifiziert)

Eine klassische Webanwendung arbeitet wie in Abbildung 30 (links) dargestellt. Eine User Interaktion veranlasst eine HTTP-Anfrage beim Webserver. Der Server verar-

32 Asynchronous JavaScript + XML

beitet die Anfrage und sendet eine Antwort in Form einer HTML-Seite zum Client. Bei einer Anwendung, die AJAX nutzt, wird aufgrund der User-Interaktion ein JavaScript Call zum Web-/ XML-Server durchgeführt. Der Client bekommt XML-Daten zurück, diese werden anschließend auf der Clientseite verarbeitet und das Ergebnis mit HTML und CSS dargestellt (vgl. Abbildung 30 (rechts)). (vgl. Garrett, 2005) Für diese Interaktion wird das JavaScript-Objekt `XMLHttpRequest` benötigt. Der Ablauf eines XHR ist in Listing 5 dargestellt.

Listing 5: Ablauf eines XMLHttpRequest (Quelle: van Kesteren, 2012)

```
var client = new XMLHttpRequest();
client.open(„Get“, „http://example.com/hello“);
client.onreadystatechange = function(){
  /* do something */
};
client.send();
```

Bisher war es nicht möglich, mit einem XMLHttpRequest auf ein anderes als das Origin zuzugreifen, von dem das JavaScript geladen wurde. Beim W3C wird an einem Working Draft gearbeitet, der es ermöglicht, Origin-übergreifend Requests abzusetzen. Dabei muss der Server in der Rückantwort im Header `Access-Control-Allow-Origin` mit dem Quellsystem als Parameter mitsenden. Erst dann ist es für den Browser möglich, die Antwort zu verarbeiten. (vgl. van Kesteren, 2012)

2.7.3.1. Comet

Comet ist ein Modell für Webanwendungen, bei dem eine HTTP-Anfrage abgesetzt und künstlich gehalten wird, bis die gewünschte Information vom Server vorhanden ist. Andere Bezeichnungen für Comet sind bspw. Ajax Longpolling, Ajax Push oder HTTP Streaming. (vgl. Crane & McCarthy, 2008)

Diese Methode hat jedoch einige Nachteile. Das Skript kann durch Latenzen und Limitationen des Servers bspw. durch einen Timeout abgebrochen werden. Es benötigt viele Ressourcen am Server und kann auch die Kommunikation am Client blockieren. Für moderne Webapplikationen sind Web Sockets (siehe Kapitel 2.7.4.6) die bessere Wahl.

2.7.4. JavaScript APIs im Umfeld von HTML 5

2.7.4.1. Application Cache API

Das JavaScript API dient zum Ansprechen des Offline-Speichers. Ist eine Manifest-Datei (siehe Kapitel 2.7.1.1) im HTML verlinkt, versucht der Browser alle angeführten Dateien in den Cache zu laden. Dabei kann das Application Cache API zu Hilfe

genommen werden, um bspw. Events zu behandeln und den aktuellen Status abzurufen. (vgl. Hickson, 2012b; Spiering & Haiges, 2010, p. 231ff)

Tabelle 2 beinhaltet alle Variablen und Methoden, die das Application Cache-Objekt nach der Initialisierung bereithält.

Tabelle 2: Variablen und Methoden des Application Cache (Quelle: Hickson, 2012b)

`status`	Die Variable gibt den aktuellen Status des Application Cache als Konstante (siehe Tabelle 3) zurück.
`update()`	Die Methode startet den Download-Prozess des Application Cache.
`swapCache()`	Die Methode wechselt zum aktuellen Application Cache.

Tabelle 3 stellt alle Stati dar, die der Application Cache annehmen kann.

Tabelle 3: Stati des Application Cache (Quelle: Hickson, 2012b)

`UNCACHED`	Für die aktuelle Website steht kein Cache zur Verfügung.
`IDLE`	Der Cache ist vorhanden und befindet sich im Leerlauf.
`CHECKING`	Die Manifest-Datei wird auf Aktualität geprüft.
`DOWNLOADING`	Neue Ressourcen werden heruntergeladen.
`UPDATEREADY`	Der Download der Ressourcen wurde erfolgreich abgeschlossen.

Alle Events, die das Application Cache API zur Verfügung stellt, sind in Tabelle 4 aufgelistet.

Tabelle 4: Events des Application Cache API (Quelle: Hickson, 2012b; Spiering & Haiges, 2010, p. 231ff)

`checking`	Der Browser prüft die Manifest-Datei auf Aktualität oder lädt die Datei zum ersten Mal.
`error`	Ein Fehler ist aufgetreten.
`noupdate`	Die Manifest-Datei hat sich nicht verändert.
`downloading`	Der Browser hat Veränderungen an der Manifest-Datei festgestellt und wird nun die Ressourcen laden oder lädt die Ressourcen zum ersten Mal.
`progress`	Der Browser lädt eine Ressource der Manifest-Datei herunter. Dieses Ereignis wird einmal pro Ressource aufgerufen.
`updateready`	Alle Ressourcen sind aktualisiert. Das Skript kann die Funktion `swapCache()` (vgl. Tabelle 2) nutzen, um zum neuen Cache zu wechseln.
`cached`	Alle Ressourcen sind herunterladen.

Ein Beispiel, wie das Application Cache API eingesetzt werden kann, zeigt Listing 6. Mit `window.applicationCache` wird auf den Application Cache zugegriffen (1). Der Event-Listener für das Event `updateready` (2) wird aktiv, wenn alle Ressour-

cen erfolgreich geladen wurden. Daraufhin wird die Methode `swapCache` (3) auf den Application Cache ausgeführt, um zum aktuellen Cache zu wechseln. Ein Neuladen des Fensters (4) bezweckt, dass der User automatisch mit dem neuen Cache arbeiten kann.

Listing 6: Beispiel zur Nutzung des Application Cache API

```
1  var cache = window.applicationCache;
   var updateManifest = true; // true für Intervall Update

   // Update ready
2  cache.addEventListener(‚updateready‘, function(e){
      console.log(„Cachemanager: Ressourcen fertig geladen“);
3     cache.swapCache();
      console.log(‚Cachemanger: Wechsel zu neuen Ressourcen‘);
4     window.location.reload();
   }, false);

   // Manuelles Update
5  if(updateManifest){
      setInterval(function(){
6        cache.update();
      }, 10000);
   }
```

Für die Entwicklung kann es von Vorteil sein, wenn die Applikation automatisch kontrolliert, ob der Application Cache neu geladen werden muss. Listing 6 zeigt eine mögliche Umsetzung. Über die Variable `updateManifest` wird bestimmt, ob das Intervall zum Aktualisieren aktiv sein soll. Ist dies der Fall (5) wird die Funktion `update` auf den Application Cache ausgeführt (6).

2.7.4.2. Browser Status

Der Browser Status kann mit Hilfe des `window`-Attributes `navigator.onLine` abgefragt werden. Es stehen zwei Rückgabewerte zur Verfügung (`true` und `false`). `true` bedeutet, dass der Browser keine Netzwerkverbindung haben könnte. Nimmt das Attribut den Wert `false` an, ist der Browser definitiv offline (siehe auch Tabelle 5). (vgl. Hickson, 2012b; Spiering & Haiges, 2010, p. 229)

Tabelle 5: Rückgabewerte von navigator.onLine (Quelle: Hickson, 2012b)

`true`	Der Browser könnte eine Netzwerkverbindung haben.
`false`	Der Browser ist definitiv offline.

Zudem sind im Working Draft zu HTML 5 zwei Events (`online` und `offline`) definiert, die je nach Zustand der Netzwerkverbindung ausgelöst werden. (vgl. Hickson, 2012b) Listing 7 zeigt ein Code-Beispiel zur Verwendung dieser Events.

Listing 7: Event-Listener für Online und Offline

```
window.addEventListener(„online“, function(){
   console.log(„Der Browser hat eine Internetverbindung.“);
});
window.addEventListener(„offline“, function(){
   console.log(„Der Browser hat keine Internetverbindung.“);
});
```

2.7.4.3. Web Storage

Der Web Storage erlaubt das Speichern von Schlüssel/Wert-Paaren (Key/Value) in einem Storage-Objekt. Mit den Methoden `getItem`, `setItem`, `removeItem` und `clear` kann auf dieses Objekt zugegriffen werden.

Die Verarbeitung und das Speichern der Daten findet durch ein clientseitiges Skript statt. Es werden keine Daten über HTTP-Nachrichten transportiert.

Es wird zwischen Local Storage und Session Storage unterschieden. Der Session Storage hält nur für die Dauer eines geöffneten Browserfensters und wird pro Seite und Fenster angelegt. Der Local Storage überdauert eine Session und wird pro Origin[33] angelegt. Die Größe des Storage-Objektes ist dabei auf 5MB beschränkt. Die Daten im Local Storage sind demnach auch bei einem späteren Besuch der gleichen Webseite verfügbar.

(vgl. Hickson, 2012c; Kessin, 2011, p. 49ff; Spiering & Haiges, 2010, p. 188ff; West & Pulimood, 2012, p. 82f)

Listing 8 zeigt den Zugriff auf den Local Storage. Dabei wird mit der Funktion `setItem` (1) das Schlüssel/Wert-Paar „hintergrundfarbe“ und „blau“ im Local Storage-Objekt gespeichert. Mit der Funktion `getItem` (2) wird der Wert des Schlüssels „hintergrundfarbe“ aus dem Local Storage-Objekt gelesen. `removeItem` (3) bewirkt das Löschen des Schlüssel/Wert-Paares aus dem Objekt.

Listing 8: Zugriff auf das Local Storage

```
   // set
1  localStorage.setItem(„hintergrundfarbe“, „blau“);

   // get
   var bgcolor;
2  bgcolor = localStorage.getItem(„hintergrundfarbe“);

   // remove
3  localStorage.removeItem(„hintergrundfarbe“);
```

33 Ein Origin besteht aus Schema, Host und Port. Für http://mobilot.at bedeutet dies: Schema = http://, Host = mobilot.at, Port = 80.

2.7.4.4. Web SQL Database API

Die Spezifikation zu Web SQL definiert eine Schnittstelle zu einer clientseitigen SQL[34]-Datenbank. Web SQL wird jedoch nicht mehr durch das W3C[35] weiterentwickelt und wird daher in der finalen Version von HTML 5 keine Berücksichtigung finden. Als neue lokale Datenbank wird IndexedDB (siehe Kapitel 2.7.4.5) spezifiziert. Die Web SQL-Datenbank ist jedoch derzeit die einzige native Möglichkeit lokale Datenbanken in mobilen Browsern zu erstellen (vgl. Kapitel 2.7.5). Aus diesem Grund wird sie an dieser Stelle erläutert.

Wie bei Web Storage gilt auch bei Web SQL das Origin-Prinzip. Webseiten des gleichen Origin können auf alle Daten der Datenbank zugreifen, diese ändern und löschen.

Tabelle 6 gibt einen Überblick über die Attribute und Methoden, welche das Web SQL Database API zur Verfügung stellt. Die drei Hauptmethoden sind dabei: `openDatabase` (Erstellen der Datenbank), `transaction` (Kontrolle über Transaktion) und `executeSql` (Ausführen einer SQL-Query).

Tabelle 6: Attribute und Methoden des Web SQL Database API (Quelle: Hickson, 2010)

`openDatabase()`	Erstellen der Datenbank, dabei wird entweder eine existierende genutzt oder eine neue erstellt.
`database.changeVersion()`	Versionsnummer ändern
`database.version`	Versionsnummer der Datenbank (Read only)
`database.transaction()`	Möglichkeit, eine Transaktion zu kontrollieren, um je nach Situation entweder im Erfolgsfall einen Commit oder im Fehlerfall einen Rollback auszuführen (Read-Write).
`database.readtransaction()`	Möglichkeit, eine Transaktion zu kontrollieren, um je nach Situation entweder im Erfolgsfall einen Commit oder im Fehlerfall einen Rollback auszuführen (Read only).
`transaction.executeSql()`	SQL-Query ausführen

34 Die Structured Query Language (kurz SQL) ist eine standardisierte Sprache für relationale Datenbanksysteme. Im Standard ist definiert, welche Funktionen von Datenbanken unterstützt werden müssen, die mit SQL als Abfragesprache arbeiten. (vgl. Thies & Reimers, 2009, p. 18)

35 Das World Wide Web Consortium (kurz W3C) definiert die einheitlichen Standards für das World Wide Web.

Listing 9: Öffnen einer bestehenden oder Anlegen einer neuen Datenbank

```
var database = openDatabase(„mobidul_da“, „1.0“,
  „Mobidul Kerstins DA“, 5*1024*1024);
```

Das Öffnen einer vorhandenen oder Anlegen einer neuen SQLite-Datenbank erfolgt mit der Methode `openDatabase` (siehe Listing 9). Dieser Methode können fünf Parameter übergeben werden:

1. Name der Datenbank (String)
2. Version der Datenbank (String)
3. Beschreibung (String)
4. Größe der Datenbank (Zahl), wobei 5*1024*1024 - 5 MB entspricht
5. Callback, der aufgerufen wird, wenn die Datenbank erstellt wurde (optional)

Wird der Callback als 5. Parameter nicht angegeben, behandelt das API die Versionierung selbst. Wird er genutzt, so muss in der Callback-Funktion die Methode `changeVersion` auf der Datenbank ausgeführt werden. Diese Methode erwartet folgende Parameter:

1. alte Versionsnummer (String)
2. neue Versionsnummer (String)
3. SQLTransaction-Callback (optional)
4. errorCallback für den Fehlerfall (optional)
5. successCallback für den Erfolgsfall (optional)

Das Schreiben und Lesen auf der Datenbank erfolgt innerhalb einer Transaktion. Transkationen stellen in relationalen Datenbanken abgeschlossene Einheiten dar. Sie setzen sich aus mehreren Befehlen zusammen. Entweder werden alle Befehle innerhalb einer Transaktion erfolgreich ausgeführt oder bei Fehler alle zurückgenommen. Das Datenbankobjekt stellt dafür zwei Methoden bereit: `transaction` (Read-Write) und `readtransaction` (Read only). Diese erwarten folgende Parameter:

1. SQLTransaction-Callback
2. errorCallback für den Fehlerfall (optional)
3. successCallback für den Erfolgsfall (optional)

Der erste Parameter dieser Methode ist eine Callback-Funktion, an die das Transaktionsobjekt übergeben wird. Die Methode `executeSQL` kann auf diesem Transaktionsobjekt genutzt werden, um SQL-Statements auszuführen. Dieser Methode können folgende Parameter übergeben werden:

1. SQL Statement (String)
2. Arguments (ObjectArray, optinal)
3. SQLStatement-Callback (optional)
4. SQLStatementError-Callback (optional)

(vgl. Hickson, 2010; Spiering & Haiges, 2010, p. 202f)

Ein Beispiel-Listing zeigt Listing 10. Auf der in Listing 9 geöffneten Datenbank wird die Methode `transaction` ausführt (1). Dieser wird als SQLTransaction-Callback eine Funktion übergeben, in der das Transaktionsobjekt (`tx`) zur Verfügung steht. In diesem Callback werden mit der Methode `executeSQL` drei SQL-Statements ausgeführt. In der ersten Query (2) wird die Tabelle mit dem Namen „imlabprojekt“ erstellt und in der zweiten und dritten (3) werden in die erstellte Tabelle Zeilen eingefügt. SQL Injections können bei Web SQL durch parametrisierte SQL-Strings – ein Platzhalter wird mit einem Fragezeichen (?) ausgedrückt – vermieden werden. In die ausführende Methode `executeSql` wird dafür als zweiter Parameter eine Array mit den einzufügenden Werten übergeben. SQLite kann so die Daten automatisch maskieren.

Listing 10: Erstellen und Befüllen einer Datenbank-Tabelle

```
1 database.transaction(function(tx) {
2   tx.executeSql(„CREATE TABLE IF NOT EXISTS imlabprojekt (pID INTEGER
      NOT NULL PRIMARY KEY AUTOINCREMENT, pName TEXT)“;
3   tx.executeSql(„INSERT INTO imlabprojekt (pName) VALUES (?)“,
      [„Projekname 1“]);
3   tx.executeSql(„INSERT INTO imlabprojekt (pName) VALUES (?)“,
      [„Projektname 2“]);
});
```

In Listing 11 wird mit `readtransaction` eine lesende Transaktion auf der Datenbank erstellt (1). Mit `executeSQL` wird im SQLTransaction-Callback das `SELECT`-Statement ausgeführt (2). Als dritter Parameter wird der Methode `executeSQL` eine SQLStatement-Callback-Funktion übergeben. In dieser steht neben dem Transaktionsobjekt (`tx`) das Ergebnis der Abfrage (`rs`) zur Verfügung (3). Dieses Result Set kann bspw. über eine For-Schleife ausgelesen werden (4).

Listing 11: Auslesen einer Datenbank-Tabelle

```
1 database.readtransaction(function(tx) {
2   tx.executeSql(„SELECT * FROM imlabprojekt“, [],
3     function(tx, rs){
        var result = [];
4       for (var i=0; i < rs.rows.length; i++) {
          result.push(rs.rows.item(i));
        }
        console.log(result);
      };
});
```

2.7.4.5. Indexed Database API

Mit Hilfe des Indexed Database API können JavaScript-Objekte direkt in eine indizierte Datenbank geschrieben werden. Kessin (2011, p. 61ff) beschreibt den Vorteil dieses Prinzips wie folgt: JavaScript-Objekte müssen nicht in eine SQL-Tabellenstruktur gemappt werden. Außerdem sind keine SQL Injections möglich. XSS[36] kann jedoch ein Problem darstellen. Dies kann passieren, wenn Daten über ein Textfeld eingeben und in der lokalen Datenbank gespeichert werden. Über dieses Textfeld kann der User JavaScript-Befehle verpackt in einen `script`-Tag einschleusen. Zu diesem Zeitpunkt hätte der Angreifer allerdings nur Einfluss auf die lokale Datenbank. Wird jedoch die lokale Datenbank bidirektional synchronisiert, wird der eingeschleuste Code auf die serverseitige Datenbank übertragen und in weiterer Folge auf alle Clients verteilt.

Das Origin-Prinzip von Web SQL und Web Storage greift auch bei Indexed DB. Auf eine Datenbank, die von einer Seite erstellt wurde, kann von allen Seiten desselben Origin zugegriffen werden.

(vgl. Kessin, 2011, p. 61ff; Mehta, Sicking, Graff, Popescu & Orlow, 2011)

Ein Object Store einer Indexed Database (vergleichbar mit einer Tabelle in Web SQL) kann nur in einer Version Change-Transaktion erstellt werden. Listing 12 zeigt das Erstellen des Object Store „imlabprojekt“. Dafür wird über den Befehl `window.indexedDB.open` eine Datenbank mit dem Namen „mobilot_da“ erstellt (1). Im Erfolgsfall wird `onsuccess` ausgeführt (2). Dieser Funktion wird im Objekt `e.target.result` die Datenbank übergeben (3). Über die Funktion `setVersion` wird eine Version Change-Transaktion ausgeführt (4). Diese Funktion erwartet die neue Versionsnummer als Parameter. Im Erfolgsfall (5) kann nun der Object Store erstellt werden. Im Beispiel wird zuerst überprüft, ob bereits ein Store mit demselben Namen existiert (6). Ist dies der Fall, wird dieser mit dem Befehl `deleteObjectStore` gelöscht (7). Anschließend wird mit `createObjectStore` ein neuer Object Store mit dem Namen „imlabprojekt“ erstellt (8).

36 Als Cross-Site-Scripting (kurz XSS) wird das Einschleusen von Scripting-Befehlen in eine Webanwendung bezeichnet.

Listing 12: Erstellen eines Object Store

```
1  var request = window.indexedDB.open(„mobilot_da“);
2  request.onsuccess = function(e) {
     var versionChange = 1.99;
3    var database = e.target.result;
     if (versionChange != database.version) {
4      var setVrequest = database.setVersion(versionChange);
5      setVrequest.onsuccess = function(e) {
         // Check ob objectStore schon da
6        if(database.objectStoreNames.contains(„imlabprojekt“)){
           // Lösche ObjectStore
           try{
7            database.deleteObjectStore(„imlabprojekt“);
           } catch(e){
             console.log(„ERROR DELTE: imlabprojekt“);
           }
           var store;
           try{
8            store = database.createObjectStore(
               „imlabprojekt“, {keyPath: „pID“}
             );
           }catch(e){
             console.log(„ERROR CREATE: imlabprojekt“);
           }
         }
       }
     }
   }
```

Jedes JavaScript-Objekt muss einen Schlüssel (Key) besitzen, mit dem das Objekt abgefragt werden kann. Zusätzlich können auch Sekundärschlüssel eingesetzt werden. Der Key eines Object Stores wird in der Methode `createObjectStore` als zweiter Parameter (siehe Listing 12, 8) übergeben.

Um einem Object Store Daten hinzufügen zu können (siehe Listing 13), wird eine Transaktion erstellt (1). Der Methode `transaction` wird als erster Parameter ein Array mit allen Object Stores übergeben, mit denen gearbeitet werden soll. Als zweiten Parameter erwartet die Methode eine Konstante, die besagt, welche Art von Transaktion ausgeführt werden soll (`READ_ONLY`, `READ_WRITE` oder `VERSION_CHANGE`). Im Anschluss wird der für die Transaktion relevante Object Store gewählt (2). Anschließend kann auf diesem Store die Methode `add` ausgeführt werden, um Datenobjekte hinzuzufügen (3).

Listing 13: Items zum Store hinzufügen

```
1 var trans = database.transaction([„imlabprojekt“],
    IDBTransaction.READ_WRITE);
2 var store = trans.objectStore(„imlabprojekt“);
  var data = {
    pID   :  1,
    pName :  „Projektname 1“
  };
  var request;

  try{
3   request = store.add(data);
  }catch(e){
    console.log(„ERROR ADDITEM: „ + e + „ / „ + „imlabprojekt“);
  }
  request.onsuccess = function(e) {
    // Erfolg
  }
  request.onerror = function(e) {
    console.log(„Error Adding: „, e);
  };
```

Um Items aus dem Store auszulesen (siehe Listing 14), wird ebenfalls eine Transaktion (1) und der entsprechende Object Store benötigt (2). Außerdem muss eine KeyRange bestimmt werden. Dazu stehen folgende Funktionen zur Verfügung (vgl. Mehta et al., 2011):

1. `only` – untere und obere Grenze nehmen den übergebenen Wert an
2. `lowerBound` – untere Grenze bestimmen
3. `upperBound` – obere Grenze bestimmen
4. `bound` – untere Grenze (1. Parameter) und obere Grenze (2. Parameter) bestimmen

Im Beispiel wird die untere Grenze auf 0 gesetzt (3). Damit werden sämtliche Objekte aus dem Object Store ausgelesen. Zudem wird auf dem Store ein Cursor mit der eben erstellten KeyRange geöffnet (4). Im Erfolgsfall (5) enthält das übergebene Objekt in `target.result` das Ergebnis der Abfrage (6). Um alle Ergebnisse für die KeyRange zu erhalten wird auf dem Result-Objekt die Methode `continue` ausgeführt (7).

Listing 14: Items aus dem Store auslesen

```
1  var trans = database.transaction([„imlabprojekt“],
     IDBTransaction.READ_ONLY, 0);
2  var store = trans.objectStore(„imlabprojekt“);
   // alle Daten aus dem Store holen
3  var keyRange = IDBKeyRange.lowerBound(0);
4  var cursor = store.openCursor(keyRange);
5  cursor.onsuccess = function(e) {
6    var result = e.target.result;
     if(!!result == false){
       return;
     }
     console.log(result.value);
7    result.continue();
   };
   cursorRequest.onerror = function(e) {
     console.log(„Error Adding: „, e);
   };
```

Da Indexed DB sich erst seit April 2011 (Erstellung der 1. Version des Working Drafts) in der Entwicklung befindet, ist die Umsetzung in den Browsern (bisher nur Firefox seit Version 9 und Chrome seit Version 17 (vgl. Deveria, 2012b)) stellenweise unterschiedlich und noch nicht ausgereift.

2.7.4.6. Web Sockets

HTTP ist ein zustandsloses Protokoll. Der Webbrowser stellt eine Verbindung zum Server mit einer Anfrage her, erhält eine Antwort und trennt die Verbindung wieder. Ein Web Socket ist ein klassischer TCP/IP[37]-Socket. Ein Socket wird durch den Browser geöffnet und so lange offen gelassen, bis dieser nicht mehr benötigt und explizit geschlossen wird. Ein Web Socket stellt einen bidirektionalen Echtzeit-Datenkanal dar. Im Vergleich dazu kann mit herkömmlichen HTTP-Anfragen nur ein einfaches Polling System realisiert werden. (vgl. Hickson, 2011; Hogan, 2011, p. 213ff; Kessin, 2011, p. 103ff)

Das Web Socket-Protokol baut nicht auf HTTP auf. Es definiert einen HTTP-Handshake, um zwischen einer bestehenden HTTP-Verbindung und einer Web Socket-Verbindung zu wechseln. Web Sockets versucht nicht einen Server Push-Kanal auf HTTP zu simulieren. Es definiert lediglich ein Framing-Protokol aufbauend auf TCP. (vgl. Roth, 2010)

Der Ablauf eines Web Socket-Handshake ist in Listing 15 und Listing 16 dargestellt. Der Browser sendet eine Anfrage zum Server, die ihm mitteilt, dass vom HTTP zum

37 Transmission Control Protocol / Internet Protocol (kurz TCP/IP)

Web Socket-Protokoll gewechselt werden soll. Dies wird über den Header `Upgrade` erreicht (1).

Listing 15: Header einer Anfrage des Clients (Quelle: Kaazing Corporation, 2012)

```
GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Origin: http://websocket.org
Cookie: __utma=99as
Connection: Upgrade
Host: echo.websocket.org
Sec-WebSocket-Key: uRovscZjNol/umbTt5uKmw==
1 Upgrade: websocket
Sec-WebSocket-Version: 13
```

Versteht der Server das Web Socket-Protokoll, akzeptiert er den Protokoll-Wechsel mit dem Upgrade-Header (2).

Listing 16: Header einer erfolgreiche Antwort des Servers (Quelle: Kaazing Corporation, 2012)

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Fri, 10 Feb 2012 17:38:18 GMT
Connection: Upgrade
Server: Kaazing Gateway
2 Upgrade: WebSocket
Access-Control-Allow-Origin: http://websocket.org
Access-Control-Allow-Credentials: true
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Access-Control-Allow-Headers: content-type
```

Mit der erfolgreichen Server-Antwort (Listing 16) wird die HTTP-Verbindung abgebrochen und mit einer Web Socket-Verbindung über die gleiche zugrundeliegende TCP/IP-Verbindung ersetzt. Dabei nutzt eine Web Socket-Verbindung grundsätzlich die gleichen Ports wie eine HTTP- (80) und HTTPS-Verbindung (443). (vgl. Kaazing Corporation, 2012)

Um Web Sockets nutzen zu können, wird ein Web Socket-Objekt erstellt (vgl. Listing 17). Als Parameter übergibt man die Web Socket-URL. Eine solche URL startet mit ws oder wss (für eine gesicherte Web Socket-Verbindung, die SSL nutzt).

Listing 17: Öffnen einer WebSocket-Verbindung

```
var socket = new WebSocket(‚ws://localhost:80/test/server.php’);
```

In Tabelle 7 sind alle Events und Methoden aufgelistet, die clientseitig bei einer Web Socket-Verbindung zur Verfügung stehen.

Tabelle 7: Web Sockets-Events und -Methoden

`onopen()`	Event, wenn die Verbindung zum Server hergestellt wurde.
`onmessage()`	Event, wenn eine Nachricht vom Server gesendet wurde.
`onclose()`	Event, wenn die Verbindung zum Server abbricht oder geschlossen wird.
`onerror()`	Event, wenn ein Fehler aufgetreten ist.
`send()`	Nachrichten zum Server senden
`close()`	Verbindung zum Server beenden

Ist eine Web Socket-Verbindung erfolgreich geöffnet, kann dieser Status über das Event `onopen` abgefangen werden (Listing 18, 1) und entsprechende Aktionen ausgeführt werden. Im Beispiel wird die Methode `send` aufgerufen, mit der eine Nachricht an den Server (limitiert auf valide UTF-8 Strings) gesendet wird (2). Außerdem wird ein Event-Listener für `onmessage` implementiert (3). Dieser nimmt Nachrichten des Servers entgegen und verarbeitet sie.

Listing 18: Event-Handling bei einer Web Socket-Verbindung

```
   // wenn Socket geöffnet ist
1  socket.onopen = function(){
2     socket.send(JSON.stringify({
         name: „mobilot“
      }));
   };
   // wenn eine Nachricht vom Server kommt
3  socket.onmessage = function(msg){
      var data = JSON.parse(msg.data);
      console.log(„Mobilot ID: “ + data.id);
   };
```

Für die Serverseite einer Web Socket-Implementierung sind alle gängigen Programmiersprachen denkbar, bspw. PHP, Ruby, Java oder serverseitiges JavaScript.

Als Beispiel ist in Listing 19 eine Implementierung eines Web Socket-Servers in PHP zu sehen. In PHP können für die Umsetzung eines solchen Servers die seit Version 4.1.0 enthaltenen Funktionen der Socket-Erweiterung[38] genutzt werden.

Zu Beginn wird die Laufzeit des PHP-Skriptes auf unendlich gesetzt (1). Mit `socket_create` wird der Socket erstellt (2). Diese Funktion erwartet drei Parameter:

1. Protokollfamilie = AF_INET
2. Typ = SOCK_STREAM
3. Protokoll = SQL_TCP (weil Web Sockets auf TCP/IP basieren)

Anschließend benötigt der Socket die Zuweisung der Adresse `$host` und der Portnummer `$port`, an die er gebunden sein soll (3). Das Ausführen der Funktion

38 http://www.php.net/manual/de/intro.sockets.php

`socket_listen` erreicht, dass der Socket auf eingehende Verbindungen horcht (4). Nun kann das Array mit den Socket-Verbindungen mit `socket_select` auf Änderungen überprüft werden (5). Mit `socket_accept` wird die Anfrage eines Clients akzeptiert und ein Client-Socket erstellt (6). Eine Nachricht vom Client kann daraufhin mit der Funktion `socket_recv` entgegengenommen werden (7). Die Funktion `socket_write` steht zum Versenden von Nachrichten zur Verfügung. (vgl. Schmitzer, 2011)

Listing 19: Implementierung eines Web Socket-Servers in PHP (Quelle: Schmitzer, 2011, modifiziert)

```
1  set_time_limit(0);
   $host = ‚localhost‘;
   $port = 80;

   // Socket erstellen
2  $socket = socket_create(AF_INET, SOCK_STREAM, SQL_TCP);

   // Adresse und Port binden
3  socket_bind($socket, $host, $port);

   // Lauschen
4  socket_listen($socket);

   $sockets = array($socket);
   $arClients = array();

   while(true){
      echo „Warte auf Verbindung...\r\n“;
      $sockets_change = $sockets;
5     $ready = socket_select($sockets_change, $write = null,
         $expect = null, null);
      echo „Verbindung angenommen.\r\n“;
      foreach($sockets_change as $s){
         if ($s == $socket){
            // Änderung am Serversocket
6           $client = socket_accept($socket);
            array_push($sockets, $client);
            print_r($sockets);
         }else{
            // Eingehende Nachrichten der Clientsockets
7           $bytes = @socket_recv($s, $buffer, 2048, 0);
         }
      }
   }
```

Eine Verbindung über Web Sockets bietet sich bei der Datensynchronisation in mobilen Informationssystemen an, um eine Verbindung offen zu halten, mit welcher der Server den Client über Updates am Datensatz informieren kann. Besonders in Szenario 2 bietet diese Variante einen Ressourcen schonenderen Ansatz, als regelmäßige Anfragen über AJAX absetzen zu müssen.

2.7.4.7. Web Workers

JavaScript unterstützt grundsätzlich nur einen Thread. Für kleine Applikationen ist dieser Ansatz praktisch. Mit umfangreicher werdenden JavaScript-Anwendungen stößt er jedoch an sein Limit.

JavaScript liest den Code aus einer Event-Schleife aus. In einer Warteschlange (Queue) befinden sich alle Events, die im Browser stattfinden. Immer wenn die JavaScript-Runtime im Leerlauf ist, nimmt sie das erste Event aus der Queue und startet den Handler, der mit diesem Ereignis verknüpft ist.

In den letzten Jahren wurde die Verarbeitungsgeschwindigkeit von JavaScript verbessert. In Chrome und Firefox wird JavaScript 100-mal schneller verarbeitet als in Tagen des Internet Explorer 6 – dadurch können auch mehr Events verarbeitet werden. Für die immer umfangreicheren JavaScript-Anwendungen wurde mit Google Gears die Idee des Worker Pools entwickelt. Dieser Worker Pool wurde in HTML 5 übernommen und zur Web Workers-Spezifikation, die mittlerweile aus HTML 5 ausgegliedert wurde. Ein Worker ist ein separater JavaScript-Prozess, der Berechnungen durchführt und Nachrichten zum Hauptprozess senden sowie vom Hauptprozess empfangen kann. Als einzige Methode der Interprozesskommunikation steht für Web Worker der Messaging-Mechanismus von HTML[39] bereit.

Zum Starten eines neuen Workers wird ein Worker-Objekt mit einem Parameter – dem Dateinamen – initialisiert. Damit wird ein Worker vom Quellcode der entsprechenden Datei erstellt (siehe Listing 20, 1).

Listing 20: Starten eines Web Workers (Quelle: Kessin, 2011, p. 88)

```
1 var worker = new Worker(„worker.js“);
2 worker.onmessage = function(event){
    console.log(event.data);
  };
3 worker.postMessage(„Hello World“);
```

Der Browser wird den Worker laden, den Code ausführen und danach auf Events warten. Ein solches Ereignis ist das `message`-Event, um Daten zum Worker zu senden. Dazu sendet der Haupt-Thread einen String mit der Methode `postMessage`

39 Cross-Document Messaging erlaubt es, Skripten, die auf verschiedenen Origins liegen, Nachrichten hin- und herzusenden. (Hickson, 2012d)

(3). Die Daten werden im `event.data`-Feld gehalten und der Worker kann mit dem Event `onmessage` die Daten entgegen nehmen (2).

Objekte und Interfaces, die im Web Worker verfügbar sind, sind in Tabelle 8 aufgelistet. Zusätzlich dazu sind String, Array und Date verfügbar. Außerdem kann JSON (siehe Kapitel 2.7.2.1) genutzt werden.

Tabelle 8: Im Web Worker verfügbare Objekte und Interfaces

`WorkerNavigator`	Objekt repräsentiert die Identity und den Zustand des Browsers
`WorkerLocation`	Alle Eigenschaften dieses Objektes (`href`, `protocol`, `host`, `hostname`, `port`, `pathname`, `search`, `hash`) sind Read only.
`self`	Worker-Objekt
`importScripts()`	Methode zum Importieren von anderen Skripten
`XMLHttpRequest`	Interface für AJAX-Methoden
`setTimeout()`	Methode zur Erstellung eines Timeouts
`setInterval()`	Methode zur Erstellung eines Intervalls
`close()`	Methode zum Beenden des Worker-Prozesses

Demnach kann auch nicht auf das DOM zugegriffen werden, Elemente sind somit nicht direkt manipulierbar.

(vgl. Hickson, 2012e; Hogan, 2011, p. 229f; Kessin, 2011, p. 85ff)

2.7.5. Unterstützung in Browsern

Tabelle 9: Unterstützung der in Kapitel 2.7 angeführten Technologien (Quelle: Deveria, 2012b, 2012c, 2012d, 2012e, 2012f)

	Mobile Safari (iOS 5.0)	Android Browser (Android 4.0)
Cache Manifest	ja	ja
Application Cache API	ja	ja
Browser Status	teilweise	teilweise
Web Storage	ja	ja
Web SQL Database API	ja	ja
Indexed Database API	nein	nein
Web Sockets	teilweise	nein
Web Workers	ja	nein
XMLHttpRequest	ja	ja

Tabelle 9 zeigt die Unterstützung der im Kapitel 2.7 angeführten Technologien durch relevante Browser. Die Auswahl beschränkt sich auf die neusten Versionen, der am meisten genutzten Browser (vgl. Abbildung 31).

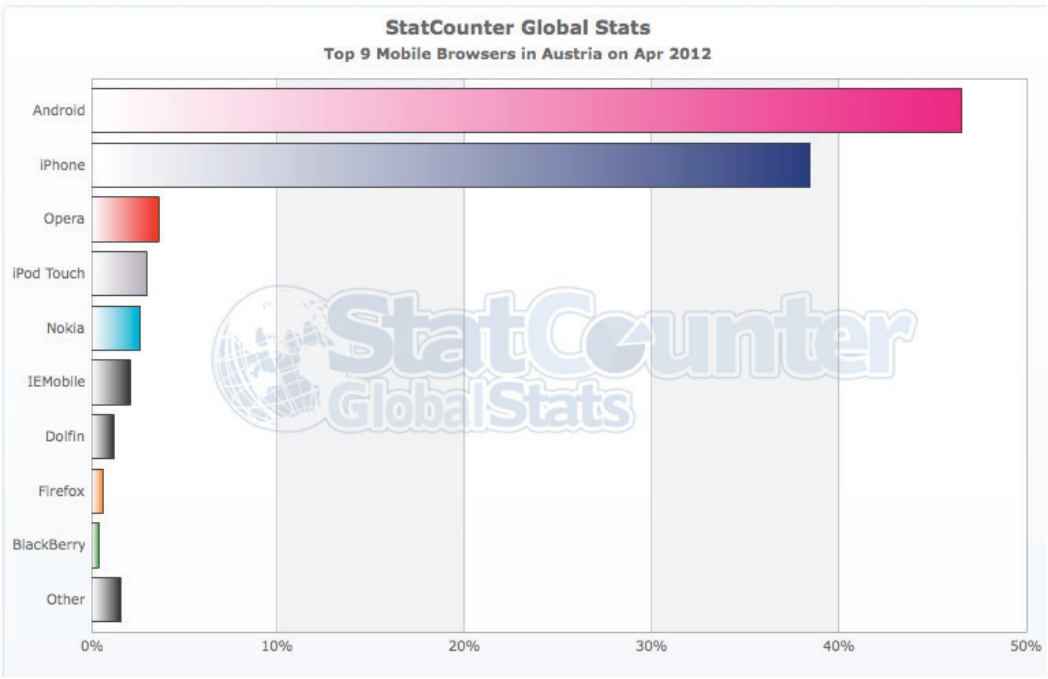

Abbildung 31: Nutzungsstatistik mobiler Browser in Österreich im April 2012 (StatCounter, 2012b)

Laut StatCounter (2012b) werden der Mobile Safari und der Android Browser von insgesamt 87,9% der BenutzerInnen verwendet. Beide Browser basieren auf WebKit und ähneln sich in der Unterstützung der Technologien. Rückschlüsse auf andere WebKit-basierte Browser (Blackberry Browser, WebOS-Browser) sind demnach möglich. Ausgeschlossen wird jedoch der Internet Explorer auf Windows Phone 7, da dieser auf der Layout Engine Trident basiert.

3. Proof of Concept

Kapitel 3 dokumentiert eine Beispielanwendung im Sinne eines Proof of Concept und zeigt, wie es möglich ist, eine mobile Webanwendung ohne Netzwerkverbindung nutzbar zu halten.

3.1. Anforderungen an die Applikation

Das Ziel der Beispielanwendung ist die Umsetzung des in Kapitel 1.2 angeführten Szenario 1. Es ist kein bzw. nur kurzzeitiger Internetzugriff vorhanden. Dafür muss die Anwendung beim ersten Aufruf alle benötigten Ressourcen und Daten auf dem Smartphone speichern. Es ist davon auszugehen, dass lediglich sporadische Updates des Inhaltes möglich sind. Der Großteil der Interaktion mit der Applikation findet ohne Netzwerkverbindung statt.

Als Basis für diese Entwicklung wird eine existierende Anwendung genutzt, die auf dem Framework Mobilot (Version beta 1.0) basiert. Folgende Features sollen integriert werden:

- Offline-Fähigkeit: Die Webanwendung soll nach erfolgter Initialisierung ohne Netzwerkverbindung genutzt werden können. Dies gilt auch für den parametrisierten Aufruf der Webanwendung durch eine URL. Mit Mobilot kann dies bspw. durch die Übergabe eines Codes durch Scannen eines QR-Codes mit einer externen Applikation erfolgen.
- Lokale Datenhaltung: Sämtliche Daten (Inhaltsdaten und Ressourcen) sollen lokal auf dem mobilen Geräte gespeichert und automatisch synchronisiert werden.
- Private History: Die Aktionen des Users sollen auch ohne Netzwerkverbindung dokumentiert und bei verfügbarer Netzwerkverbindung an den Webserver gesendet werden.

3.2. Genutzte Design Pattern, Methoden und Technologien

Die Anwendung wird als **modeless** Application (vgl. Kapitel 2.2.2.1) implementiert. Sie arbeitet grundsätzlich lokal. Die Prüfung auf eine bestehende Internetverbindung wird nur durchgeführt, wenn diese explizit erforderlich ist. Die BenutzerInnen müssen nicht aktiv eingreifen, um zwischen Online- und Offline-Nutzung zu unterscheiden.

Der gesamte Synchronisationsprozess erfolgt im Hintergrund (**Background Sync**, vgl. Kapitel 2.2.2.2). Der User muss in den Prozess nicht eingreifen.

Als Delivery Mode wird wie bei Webanwendungen üblich die **pull-only** Variante umgesetzt (vgl. Kapitel 2.2.5.1). Der Datentransfer vom Server zum Client wird mit einem Pull des Clients gestartet.

Es werden die drei in Kapitel 2.2.5.2 beschriebenen Synchronisationsintervalle implementiert: **periodisch**, **bedingte Auslieferung** und **ad hoc**. In diesem Zusammenhang wird Periodic Refresh (vgl. Kapitel 2.3.8) für das periodische Synchronisationsintervall umgesetzt.

Mit Hilfe des Cache Manifest erfolgt das Laden der Ressourcen in den Local Cache auf Documents und in den Local Cache of Data nach dem **Eager Load** Pattern (vgl. Kapitel 2.3.2). Dabei werden alle Dateien und Daten sofort geladen. Das gegensätzliche Pattern Lazy Load (siehe Kapitel 2.3.1) ist in der Umsetzung für Szenario 1 nicht zielführend, da in dieser Anwendung alle Daten und Ressourcen auf dem Client benötigt werden, um die Offline-Nutzung zu realisieren. In diesem Zusammenhang wird ebenfalls ein **Browser-Side Cache** (vgl. Kapitel 2.3.9) umgesetzt. Es werden alle Daten, die Resultate einer Anfrage beim Server sind, auf dem Client gespeichert.

Der Cache Manager wird nach dem **Proxy** Design Pattern (vgl. Kapitel 2.3.3) realisiert. Er kontrolliert u.a. den Zugriff auf die Datensätze, in dem die Abfrage der Daten über den Cache Manager umgeleitet werden und dieser entscheidet, woher die Datensätze abgerufen werden (lokal oder über Netzwerkverbindung).

Innerhalb dieser Beispielanwendung sind keine Updates von Daten der Datenbank vom Client zum Server vorgesehen. Deshalb wird als Replikationsmethode die **Master-Slave** Datenbank Replikation (vgl. Kapitel 2.4.1) implementiert. Dabei stellt die serverseitige Datenbank den Master und die clientseitigen Datenbanken die Slaves dar. Änderungen sind nur an der serverseitigen Master-Datenbank relevant und werden bei einer Synchronisation an alle Clients (Slaves) übergeben. Die Kommunikation erfolgt daher **unidirektional** (vgl. Kapitel 2.1.2). Es ist somit nicht erforderlich, einen Locking Mechanismus (vgl. Optimistic Offline Lock (Kapitel 2.3.4) und Pessimistic Offline Lock (Kapitel 2.3.5)) zu implementieren.

Strategien zur Datenvorhersage (vgl. Kapitel 2.5) sind für die Umsetzung dieses Szenarios nicht relevant.

Technologisch gesehen wird die Anwendung als Webanwendung ausgeführt. Dabei kommen clientseitig die Sprachen HTML (vgl. Kapitel 2.7.1), JavaScript (vgl. Kapitel 2.7.2) und CSS zum Einsatz. Die Synchronisation wird mit AJAX (vgl. Kapitel 2.7.3) umgesetzt.

Aus dem Umfeld von HTML 5 werden das Cache Manifest (vgl. Kapitel 2.7.1.1) und das damit verbundene Application Cache API (vgl. Kapitel 2.7.4.1) eingesetzt, um die Ressourcen des Local Cache of Documents auf dem Client zu speichern. Außerdem werden Browser Status (vgl. Kapitel 2.7.4.2), Web Storage (im Speziellen Local Storage, vgl. Kapitel 2.7.4.3) und Web SQL Database API (vgl. 2.7.4.4) verwendet.

Als JavaScript-Bibliothek wird xui (vgl. Kapitel 3.2.1) genutzt. Zudem kommen MD5-Hashes (vgl. 3.2.2) zum Einsatz, um Änderungen im Cash Manifest und an der Datenbank zu erkennen.

Serverseitig wird der Webservice in PHP implementiert. Der Datenaustausch zwischen Client und Server erfolgt mit JSON (vgl. Kapitel 2.7.2.1). Als Datenbanksprache kommt sowohl client- als auch serverseitig SQL zum Einsatz.

3.2.1. xui

xui[40] ist eine sehr schlanke JavaScript-Bibliothek zum Erstellen von HTML 5 Webanwendungen für mobile Endgeräte. 2008 wurde mit der Entwicklung von xui begonnen. Es bestand das Bedürfnis nach einem soliden DOM Framework, welches die Latenzen und Charakteristiken des mobilen Webs berücksichtigt. Laut (xui contributers, 2010a) ist xui das kleinste Framework, das alle Geräte der mobilen Landschaft unterstützt. Nach eigenen Angaben werden WebKit-Browser, der Internet Explorer Mobile und der Blackberry Browser unterstützt. xui diktiert keine Seitenstruktur und kein Widget-Modell. Es bietet u.a.

- einfaches Selektieren von DOM-Elementen
- Manipulation des DOM
- Eventhandling für Klick, Touch-Events, Gesten und Orientation-Change
- Animationen, Transformationen und Übergänge
- Einfacher Umgang mit XMLHttpRequest

(vgl. xui contributers, 2010a, 2010b, 2010c, 2010d, 2010e, 2010f, 2010g)

Da die JavaScript-Bibliothek bereits im Mobilot-Framework genutzt wird, wurde für die Implementierung der Beispielanwendung auf Funktionen dieser Bibliothek zurückgegriffen.

3.2.2. MD5-Hashes

Der Message-Digest Algorithmus 5 (kurz MD5) erstellt aus einer beliebigen Nachricht einen 128-Bit-Hashwert. Dargestellt wird diese Prüfsumme als 32-stellige Hexadezimalzahl. In (Rivest, 1992) wird dieser Algorithmus ausführlich beschrieben.

Zum Generieren von MD5-Hashes kommt in dieser Anwendung eine JavaScript-Bibliothek von Joseph Myers[41] zum Einsatz. Sie wird u.a. eingesetzt, um Datensätze zu erkennen, bei denen Synchronisationsbedarf besteht (siehe Kapitel 3.4.4.2).

40 http://xuijs.com/

41 http://www.myersdaily.org/joseph/javascript/md5-text.html

3.3. Das Framework Mobilot[42]

Mobilot ist ein Framework, welches das Umsetzen von mobilen kontextsensitiven Systemen erleichtert. Es können Anwendungen erstellt werden, die auf die GPS-Position der NutzerInnen, die aktuelle Zeit, einen Code (bspw. QR-Code[43]) und definierbare Regeln reagieren. Mobilot, der **mobi**le Pi**lot**, bietet einen Zugang zu unterschiedlichsten Modulen, die Mobidule. Ein Mobidul ist derzeit nur auf Smartphones verwendbar und für diese optimiert.

Das erweiterbare System eröffnet einen intuitiven Zugang zu Informationen. Dabei wird der Zugriff auf diese Informationen den BenutzerInnen erleichtert. Wird ein QR-Code mit der Kamera am Smartphone eingescannt oder der Anwendung erlaubt, die aktuelle Position zu ermitteln, erhalten die NutzerInnen in Abhängigkeit von Code, Zeit und Position die zur Situation passenden Informationen. Inhalte können dabei Texte, Bilder, Videos aber auch Karten sein. Im Grunde sind alle beliebigen Webinhalte darstellbar. Außerdem ist die Anbindung an bestehende Content-Lieferanten wie Datenbanken und Content Management Systeme möglich.

3.3.1. Einsatzgebiete

Mobilot ist zur Unterstützung in beliebigen mobilen Szenarien verwendbar. Interaktive Städte-, Reise- und Museumsführer sind genauso wie GPS-basierte Spiele oder Location Based Services und Eventinformationssysteme auch für ProgrammiereinsteigerInnen leicht umsetzbar.

Den ersten Einsatz hatte Mobilot bei der European Researchers' Night[44] an der Fachhochschule St. Pölten im September 2011. Das Mobidul Fit für Forschung stellt ein Informationssystem dar und umfasst folgende Funktionen:

- Mobilot-Code (QR-Code) am Rücken der ForscherInnen scannen und mehr über sie erfahren
- Weitere Informationen bei den Projektständen erhalten, sich orientieren und herausfinden, was in der Nähe gerade aktuell ist
- Das aktuelle Programm eines Raumes herausfinden
- Die eigene elektronische Visitenkarte an ForscherInnen und Projekte senden, um in der Folge über sie auf dem Laufenden zu bleiben

42 Das Kapitel basiert auf (Eitler, Blumenstein & Schmiedl, 2011) und (Schmiedl, 2011). An (Schmiedl, 2011) hat die Autorin der vorliegenden Diplomarbeit mitgearbeitet, ist jedoch nicht als Autorin des Artikels aufgeführt.

43 Ein Quick-Response-Code (kurz QR-Code) ist ein zweidimensionaler Code, der zur Verschlüsselung von Informationen (z.B. URLs) dient.

44 Das Mobidul Fit für Forschung ist unter http://www.fit-fuer-forschung.eu/m erreichbar.

Abbildung 32 bis Abbildung 34 zeigen Screenshots des Mobiduls Fit für Forschung.

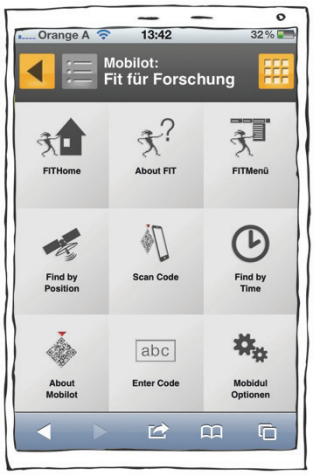

Abbildung 32: Menü des Mobiduls Fit für Forschung

Abbildung 33: Inhaltsseite einer Forscherin

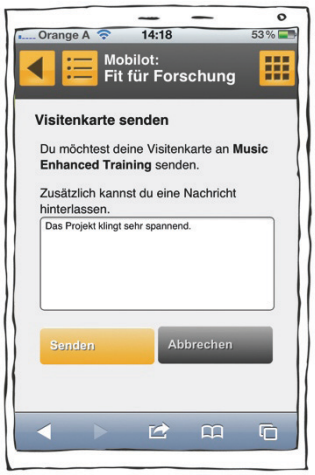

Abbildung 34: Mobilot-Seite „Visitenkarte senden

Geführte Touren

Ein weiteres Einsatzszenario ist eine geführte Tour. BenutzerInnen werden entlang eines vordefinierten Pfades geleitet. Die Geschwindigkeit der Tour ist bspw. über das Scannen von QR-Codes bzw. Code-Eingabe und / oder über die aktuelle GPS-Position steuerbar.

Location Based Games

In dieser Spielgattung wird die digitale Informationswelt mit der realen Welt verbunden und so ein interaktives Erlebnis generiert. Dafür wird die aktuelle GPS-Position der BenutzerInnen ermittelt. In Abhängigkeit dieser Position werden unterschiedliche Ereignisse ausgelöst. Es könnten bspw. neue Hinweise gegeben, weitere Aufgaben gestellt oder neue Richtungsangaben aufgedeckt werden.

Eventorganisation

Ein Mobidul kann zeit- und ortbasiert Termintafeln anzeigen. Dafür können Codes z.B. an Eingängen von Veranstaltungsräumen angebracht werden. In Abhängigkeit der aktuellen Uhrzeit werden bei Code-Eingabe andere Events angezeigt. Für Open-Air-Veranstaltung bspw. bei Festivals mit mehreren Bühnen könnte zusätzlich zur Uhrzeit auch die aktuelle GPS-Position als Einschränkung der Ergebnisse herangezogen werden.

Übersicht über Lagerbestand

Besonders relevant ist eine Übersicht über den aktuellen Lagerbestand, wenn mehrere VerkäuferInnen auf den selben Artikel zu greifen, bspw. in einem Bekleidungsgeschäft. Eine Kundin benötigt eine bestimmte Größe, die nicht mehr im Verkaufsraum

vorrätig ist. Die VerkäuferInnen erhalten durch Einscannen des Codes am Kleidungsstück unmittelbar Information über den Lagerbestand des Artikels und können so direkt Auskunft gegenüber dem Kunden geben.

3.3.2. Aufbau und Funktionsweise von Mobilot

Mobilot ist als Single Page Interface[45] realisiert. Abbildung 35 zeigt die Funktionsweise.

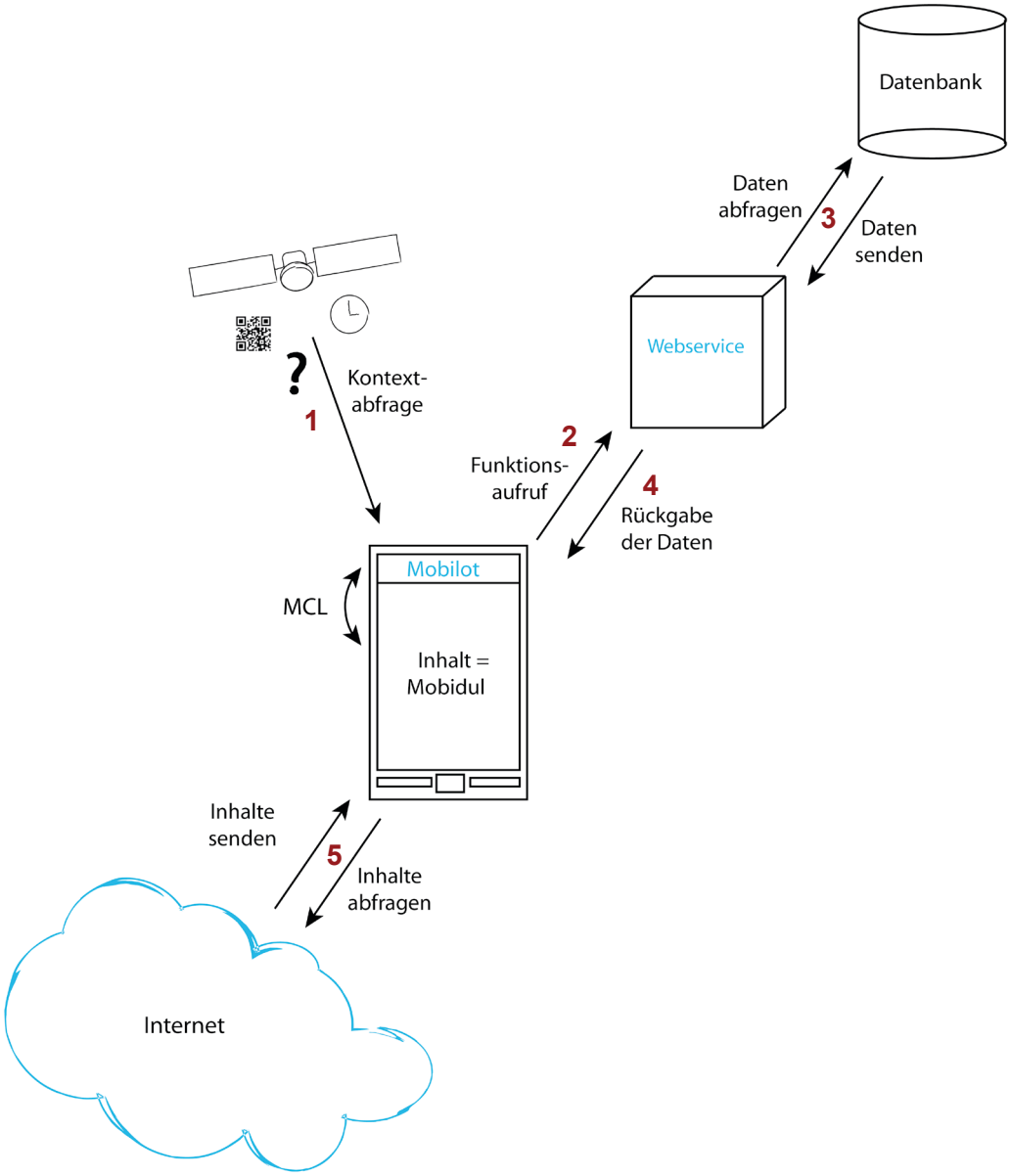

Abbildung 35: Funktionsweise des Frameworks Mobilot

45 Änderungen werden nur bei dem jeweilig zu ändernden Element gemacht, anstatt die gesamte Seite neu zu laden. Im Gegensatz dazu sind klassische Webseiten als Multi Page Interface realisiert. (vgl. Mesbah & Deursen, 2007, p. 44)

Der aktuelle Kontext – Daten, Code-Eingabe (per Tastatur oder Scan eines QR-Codes) oder die aktuelle Zeit – wird vom Handy verarbeitet (1). Mobilot überprüft die kontextspezifischen Daten und startet den entsprechenden Funktionsaufruf (2). Basierend auf den übergebenen Informationen sowie in der Datenbank hinterlegter Regeln wird die relevante Ressource identifiziert (3). Die Daten werden an das mobile Gerät gesendet (4). Die vom Server gesendeten Daten enthalten den Verweis auf den relevanten Inhalt. Dieser Inhalt wird nun vom mobilen Gerät angefordert (5). Der tatsächliche Inhalt kann von einer beliebigen Quelle im Internet, aber auch vom Webserver stammen, auf dem das Mobidul installiert ist.

Serverseitig ist bei Mobilot ein Webservice in PHP implementiert. Dieser gewährleistet die Kommunikation mit der Datenbank. Ein REST[46]-Service steht für die Kommunikation mit diesem Webservice zur Verfügung.

Der clientseitige Aufbau ist in Abbildung 36 dargestellt. Das Framework ist auf der Clientseite in HTML 5 und JavaScript geschrieben. Es besteht aus einer index.html (1), in welcher der Header (2), das Menü und alle notwendigen Seiten enthalten sind, und der dazugehörigen CSS-Datei. Das Herz von Mobilot ist eine JavaScript-Datei (3), die den Namensraum `mobilot` definiert und alle notwendigen Funktionen (bspw. zum Anzeigen der richtigen Ressource) enthält. Dazu kommen zwei JavaScript-Dateien (mobiscanner.js (4) und mobilocation.js (5)), die Mobilot als Plugins erweitern. In einer XML-Datei werden alle zur Konfiguration notwendigen Informationen hinterlegt.

Wird ein Mobidul geschrieben, muss in die in Mobilot darzustellenden Dateien (6) die JavaScript-Datei mobidul.js (7) eingebunden werden. Diese definiert einen eigenen Namensraum (`mobidul`). Die Datei gewährleistet die Kommunikation mit Mobilot (über die MCL[47]) (8). Sie stellt außerdem weitere Methoden zur Verfügung, um bspw. einfaches Lesen und Schreiben im Local Storage oder den Zugriff auf Mobilot-Variablen und Methoden zu ermöglichen. Die Kommunikation zwischen Mobilot und Mobidul erfolgt mit Hilfe des Cross Document Messaging. So ist die Kommunikation auch gewährleistet, wenn das Framework Mobilot und der Mobidul-Inhalt auf verschiedenen Origins liegen.

46 REST (Representational State Transfer) definiert, dass für Operationen wie Create, Update, Delete und Read das HTTP-Protokoll verwendet wird. (vgl. Fielding, 2000)

47 Die Mobilot Communication Language (kurz MCL) ist eine in JSON notierte Sprache, die zur Kommunikation zwischen Mobilot und Mobidul dient.

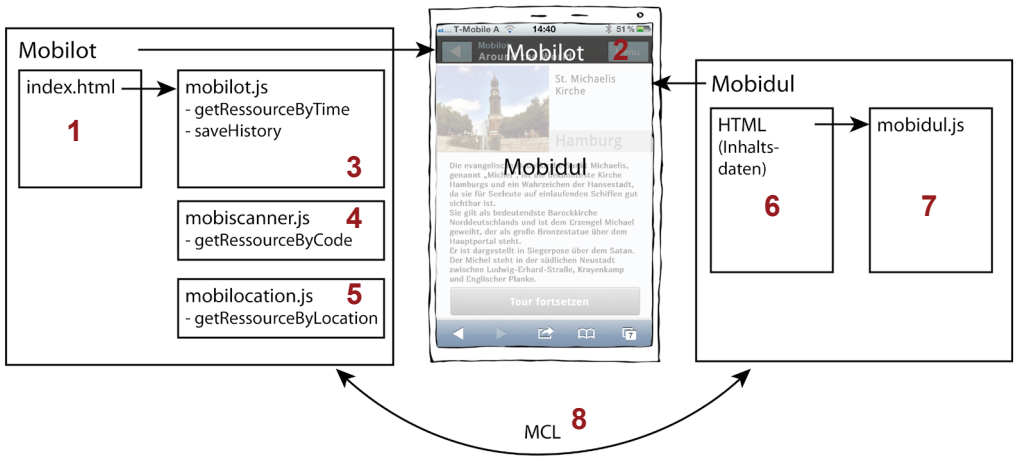

Abbildung 36: Das Framework Mobilot auf der Clientseite

3.4. Umsetzung

Abbildung 37 zeigt die Clientseite von Mobilot nach der Integration aller notwendigen Ressourcen zur Umsetzung der Beispielanwendung und verdeutlicht die Zusammenhänge der Dateien. Die Änderung im Vergleich zu Abbildung 36 sind Rot markiert.

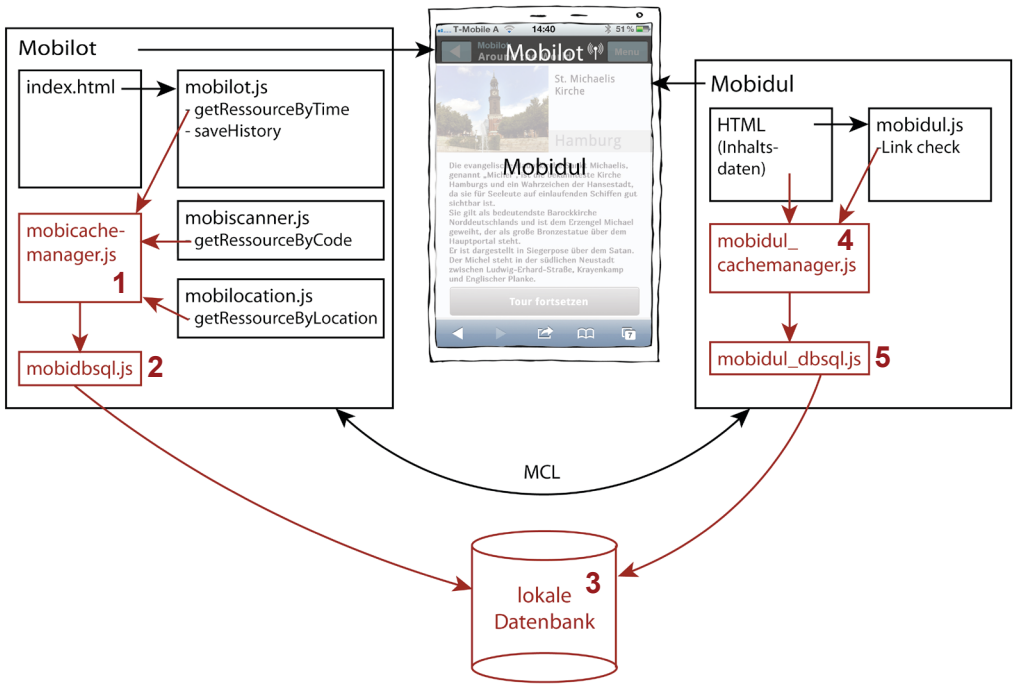

Abbildung 37: Das Framework Mobilot auf der Clientseite nach der Umsetzung

Alle Ressource-Anfragen und das Abspeichern der History gehen über den Mobilot-Cache Manager (1) (mobicachemanager.js). Dieser wiederum greift über die Datei mobidbsql.js (2) auf die lokale Datenbank (3) zu. Auf Mobidul-Seite wird ebenfalls ein

Cache Manager (mobidul_cachemanager.js) (4) implementiert. Dieser liest bspw. die Inhaltsdaten über die Datei mobidul_dbsql.js (5) aus der lokalen Datenbank (3).

Als Ausgangsbasis der in dieser Arbeit beschriebenen Offline-Erweiterung dient das zum Tag der offenen Tür der Fachhochschule St. Pölten umgesetzte Mobidul IM-Labor[48]. Das Mobidul führt durch den Interaktive Medien Park der Fachhochschule. An allen präsentierten Projekten sind QR-Codes angebracht, um über das Mobidul nähere Informationen zum Projekt zu erhalten. Abbildung 38 und Abbildung 39 zeigen Screenshots der Ursprungsversion.

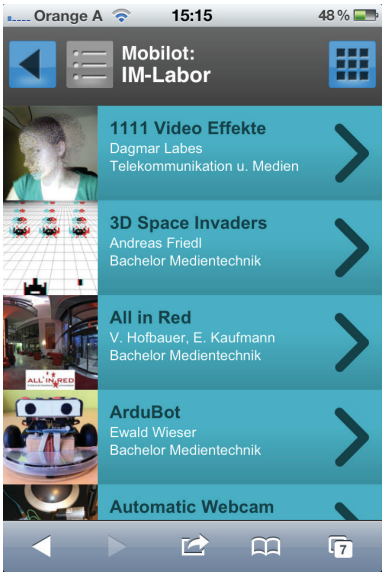

Abbildung 38: Projektübersicht des Mobidul IM-Labor

Abbildung 39: Projektdetail des Mobidul IM-Labor

Die Inhaltsdaten stehen serverseitig in einer Datenbank zur Verfügung. Zwei PHP-Skripte übernehmen die Kommunikation mit der Datenbank und die Darstellung der Inhalte in HTML. Neben Textdaten werden auch Bilddateien eingebunden.

3.4.1. Schritt 1: PHP zu JavaScript

Da PHP-Dateien serverseitig interpretiert werden müssen und nicht clientseitig verarbeitet werden können (vgl. Abbildung 26), müssen die Dateien, die lokal am Client lauffähig sein sollen, als HTML-Dateien umgesetzt werden. Die Funktionalität wird mit JavaScript realisiert.

Als Beispiel zeigen Listing 21 und Listing 22 wie im Originalcode die Inhalte des Mobiduls mit PHP geladen wurden. In der Datei `start.php` (Listing 21) wird die Funktion `getEntries` (Listing 22) ausgeführt (1). Diese Funktion stellt die Verbindung zur Datenbank her (2), führt die Datenbankabfrage durch (3) und gibt das generierte HTML aus (4). Dieser Vorgang geschieht serverseitig. Das fertige HTML wird anschließend vom Server an den Client gesendet.

48 Der Originalcode des Mobiduls IM-Labor wurde von Bernhard Grießler erstellt.

Listing 21: Auszug aus der Datei start.php (Ausgangsbasis)

```
<?php include_once(„functions.inc.php“); ?>
<!DOCTYPE html>
<html lang=“de“>
   <head>[...]</head>
1  <body> <?php getEntries(); ?> </body>
</html>
```

Listing 22: Funktion getEntries aus der Datei functions.inc.php (Ausgangsbasis)

```
function getEntries(){
   include_once(„../config.inc.php“);
   try{
2     $db = new PDO (‚mysql:host=mysql5;dbname=‘.MYSQL_DATABASE,
         MYSQL_USER, MYSQL_PASS);
   } catch(PDOException $e){
      [...]
   }
   $sql= „SELECT * from imlabprojekt ORDER BY Projektname ASC“;
3  $results=$db->query($sql);
   $results=$results->fetchAll();
   foreach ($results as $row){
4     echo „<a href=‘detail.php?id=“.$row[‚pID‘].“‘>
            <div class=‘entry‘><div class=‘ovpic‘>
               <img src=‘http://[...]“.$row[‚Foto‘].“‘/>
            </div>
            <div class=‘ovtext‘>
               <h1>“.$row[‚Projektname‘].“</h1>
               <h2>“
                  .$row[‚Name‘].
                  „<br/>“.
                  $row[‚Zusatz‘].
               „</h2>
            </div>
            <div class=‘ovarrow‘>
               <img src=‘images/arc.png‘/></div><div class=‘clear‘>
            </div>
         </div>
      </a>“;
   }
}
```

Listing 23 und Listing 24 zeigen die neu erstellten Dateien. In der Datei `imlab_overview.html` (Listing 23) wird nun eine Anfrage per AJAX (siehe Kapitel 2.7.3) abgesetzt, um die notwendige Datenbankabfrage zu realisieren (1). Diese Anfrage wird von der Datei `webservice.php` (Listing 24) verarbeitet. In dieser Datei

wird die Verbindung zur Datenbank hergestellt (2) und die Datenbankabfrage abhängig vom übergebenen GET-Parameter ausgeführt (3). Das Ergebnis der Abfrage wird als JSON (siehe Kapitel 2.7.2.1) an den Client zurückgegeben (4). Clientseitig (Listing 23) wird bei erfolgreicher Anfrage das zurückgegebene JSON-Objekt an die Funktion `render` übergeben (5), die für das Einfügen in den DOM zuständig ist.

Listing 23: Auszug aus der Datei imlab_overview.html nach dem Umschreiben zur HTML-Datei

```
<!DOCTYPE html>
<html lang="de">
   <head>
      [...]
      <script type="text/javascript">
         var pname,name,zusatz,beschreibung,foto = „";
1        xui().xhr(„webservice.php?all=all", {
               [...]
               callback: function(){
                  var json = JSON.parse(this.responseText);
5                 renderOverview(json);
               }
         });
         mobidul.init(,imlaboverview');

         function render(json){
            [...]
         }

         function showDetail(pID){
            [...]
         }
      </script>
      <link rel="stylesheet" type="text/css"
         href="css/style.css"/>
   </head>
   <body>
      <div id="content"></div>
   </body>
</html>
```

Listing 24: Datei webservice.php zur Kommunikation mit der Datenbank

```
<?php
require(„../config.inc.php“);
2 $db = connectDB();
3 if(isset($_GET[„pID“])){
    $pID = $_GET[„pID“];
    $result = $db->prepare(„SELECT * from imlabprojekt
                            WHERE pID = :projektID“);
    $result->bindParam(,:projektID‘, $pID);
    $result->execute();
    $result = $result->fetchAll();
3 }else if($_GET[„all“]){
    $result = $db->prepare(„SELECT * from imlabprojekt“);
    $result->execute();
    $result = $result->fetchAll();
  }
4 header(,Content-type: application/json‘);
  echo json_encode($result);
  ?>
```

3.4.2. Schritt 2: Ressourcen offline verfügbar machen

Im 2. Schritt werden die Dateien identifiziert, die notwendig sind, um die Offline-Verfügbarkeit der Anwendung zu gewährleisten. Darunter fallen alle HTML-, JavaScript-, CSS-, XML- und Bilddateien.

Um die Ressourcen der Webanwendung ohne Netzwerkverbindung verfügbar zu machen, wird das Cache Manifest (vgl. Kapitel 2.7.1.1) verwendet. Grundsätzlich kann diese Datei als Textdatei mit dem Content-Type `text/cache-manifest` erstellt werden. Um allerdings bei Änderung der Inhaltsdaten (bspw. neue oder andere Bilddatei) die Manifest-Datei nicht manuell ändern zu müssen, wird die Datei dynamisch mit Hilfe eines PHP-Skriptes erzeugt (siehe Listing 25).

Dieses PHP-Skript basiert auf (Stark, 2011) und wurde an die Bedürfnisse der Anwendung angepasst. Es wird durch alle Unterverzeichnisse iteriert (1). PHP- und versteckte Dateien beginnend mit „.“ werden ausgeschlossen, alle anderen werden ausgegeben (2).

Im Skript wird außerdem mit der Funktion `md5_file` ein MD5-Hash (vgl. Kapitel 3.2.2) aus jeder Datei generiert und in der Variable `$hashes` aufaddiert (3). Diese Variable wird am Ende der Manifest-Datei als MD5-Hash in einem Kommentar eingefügt (6). Dadurch kann sicher gestellt werden, dass bei Änderung einer Datei das Cache Manifest verändert wird und somit der Browser die Ressourcen neu lädt. Zusätzlich werden in der Manifest-Datei auch jene externen Ressourcen angeführt, die nicht in der selben Domain liegen (4).

Listing 25: Datei manifest.php

```
<?php
header(‚Content-Type: text/cache-manifest‘);
echo „CACHE MANIFEST\n“;
$hashes = ‚‘;

$dir = new RecursiveDirectoryIterator(„.“);
1 foreach(new RecursiveIteratorIterator($dir) as $file){
   if($file->IsFile() &&
        !strpos($file, „/.“) &&
        substr($file->getFilename(), 0, 1) !=“.“ &&
        substr($file->getFilename(), -3, 3) != „php“){
2     echo $file . „\n“;
3     $hashes .= md5_file($file);
   }
}
4 echo „http://www.mobilot.at/m/about.html\n“;
echo „http://www.mobilot.at/m/js/mobidul.js\n“;
echo „http://www.mobilot.at/m/style.css\n“;
echo „http://www.mobilot.at/m/img/Mobilot-Architektur-1024x815.png\n“;

5 echo „NETWORK:\n“;
echo „./service.php\n“;
echo „./service_json.php\n“;
echo „./content/webservice.php\n“;

6 echo „# HASH: „.md5($hashes) . „\n“;
?>
```

Bei Vorhandensein einer Manifest-Datei werden alle GET-Anfragen lokal ausgeführt. Um dies zu verhindern, werden im Bereich NETWORK die Dateien aufgelistet, die über die Netzwerkverbindung angefordert werden müssen (5). Dies sind alle PHP-Skripte, die über AJAX-Anfragen angesprochen werden.

Die Datei manifest.php wird anschließend in den HTML-Tag der index.html verlinkt (siehe Listing 26).

Listing 26: Verlinkung der Manifest-Datei im HTML-Tag der Datei index.html

```
<!DOCTYPE html>
<html lang=“de“ manifest=“manifest.php“>
```

3.4.3. Schritt 3: Lokale Datenbankanbindung schaffen

Um die Replikation der vorhandenen Datenbanktabellen auf der Clientseite zu ermöglichen, wird eine lokale Datenbank eingerichtet. Die Umsetzung erfolgt mit Web SQL (vgl. Kapitel 2.7.4.4), da das zukunftsträchtigere Indexed Database API (vgl. Kapitel 2.7.4.5) derzeit noch nicht auf mobilen Browsern unterstützt wird (vgl. Tabelle 9).

Für die Anbindung der Datenbank wird ein Mobilot-Plugin geschrieben, in dem der Namensraum `dbsql` definiert wird (Listing 27). In diesem Plugin sind alle Funktionen, die für die Kommunikation mit der Datenbank notwendig sind enthalten, bspw. zum Öffnen der Datenbankverbindung (Listing 28) und Erstellen der Tabellen (Listing 29).

Listing 27: Mobilot-Plugin für Web SQL-Datenbanken (aus mobidbsql.js)

```
mobilot.dbsql = {
   [...]
}
```

Listing 28: Öffnen der Datenbankverbindung (aus mobidbsql.js)

```
open : function() {
   var dbSize = 5 * 1024 * 1024; // 5MB
1  mobilot.dbsql.db = openDatabase(„mobidul_da“, „1.0“,
      „Mobidul Kerstins DA“, dbSize);
   console.log(„websql“);
}
```

Mit der Funktion `openDatabase` (1), die das Web SQL Database API (vgl. Kapitel 2.7.4.4) zur Verfügung stellt, wird eine Datenbank geöffnet, die den Namen „mobidul_da“ trägt, die Version 1.0 und die Beschreibung „Mobidul Kerstins DA“ besitzt und mit einer Größe von 5MB initialisiert wird. Diese Datenbank wird in der Variable `db` innerhalb des Namensraumes gespeichert.

Die Funktion `createTable` (siehe Listing 29) erwartet beim Aufruf zwei Parameter – den Namen der zu erstellenden Tabelle als String und die in der Tabelle enthaltenen Spalten als JSON-Objekt (2). Existiert die Tabelle bereits in der Datenbank, wird diese gelöscht (3). Anschließend wird das SQL Statement zum Erstellen der Tabelle in Abhängigkeit der übergebenen Parameter zusammengestellt (4). Da ein SQL Statement mit dem Web SQL Database API nur in einer Transaktion ausgeführt werden kann, wird das SQL Statement mit der Funktion `executeSql` innerhalb der Funktion `transaction` ausgeführt (5).

Listing 29: Erstellen der Tabellen (aus mobidbsql.js)

```
2  createTable : function(tableName, columns) {
     var database = mobilot.dbsql.db;
3    var sql = „DROP TABLE IF EXISTS „ + tableName;
     database.transaction(function(tx) {
       tx.executeSql(sql, [], mobilot.dbsql.onSuccess,
         mobilot.dbsql.onError);
     });
4    var sql2 = „CREATE TABLE IF NOT EXISTS „ + tableName + „ („;
     var count = 0;
     for(x in columns){
       if(count == 0){
         sql2 += x + „ „ + columns[x];
       }else{
         sql2 += „, „+ x + „ „ + columns[x];
       }
       count++;
     }
     sql2 += „)“;
5    database.transaction(function(tx) {
       tx.executeSql(sql2, []);
     });
   }
```

Innerhalb des Namensraumes `dbsql` sind außerdem Funktionen definiert, welche die Mobilot-Ressourcen-Abfragen ersetzen, wenn die Daten lokal gehalten werden. Als Beispiel ist in Listing 30 die Abfrage nach einem bestimmten Code angeführt.

Listing 30: Ressourcen-Abfrage mit der Funktion getForCode (aus mobidbsql.js)

```
1  getForCode : function(code){
     console.log(„getForCode „ + code);
     var database = mobilot.dbsql.db;
     var date = mobilot.cachemanager.getDateTime();
2    var sql = „SELECT * FROM mobilot_point [...] WHERE mobilot_point.code
       LIKE ? AND ( (? BETWEEN  mobilot_ressource.starttime AND
       mobilot_ressource.endtime) [...]“;
     database.transaction(function(tx) {
3      tx.executeSql(sql, [code, date], function(tx, rs){
         var result = [];
         var count = 0;
         for (var i=0; i < rs.rows.length; i++) {
           result.push(rs.rows.item(i));
           count++;
         }
4        mobilot.parseRessourceJSON(result, count, „code“);
       }, mobilot.dbsql.onError);
     });
   }
```

Der Funktion `getForCode` wird der abzufragende Code als String übergeben (1). Im SQL Statement werden sowohl der Code als auch die aktuelle Zeit als Fragezeichen eingefügt (2). Diese werden beim Ausführen innerhalb der Funktion `executeSql` durch die Werte, die in einem Array als zweiter Parameter übergeben werden, ersetzt (3). Sofern die Funktion `executeSql` erfolgreich ausgeführt wird, wird das Ergebnis der Abfrage aufbereitet und an die Mobilot-Funktion `parseRessourceJSON` übergeben (4). Als Parameter erwartet diese Funktion das Ergebnis der Abfrage (Array aus JSON-Objekten), die Anzahl der gefundenen Ressourcen (Zahl) und die Information, welche Funktion die Ressourcen-Abfrage ausgeführt hat (String).

3.4.4. Schritt 4: Cache Manager schreiben

Abbildung 40 zeigt die Funktionalität des Mobilot-Cache Managers. Die Hauptfunktionalitäten sind das Managen der Anfragen nach Mobilot-Ressourcen, das Managen des Datenbank-Downloads, das Managen des Ressourcen-Downloads sowie das History Logging und das Managen von Links.

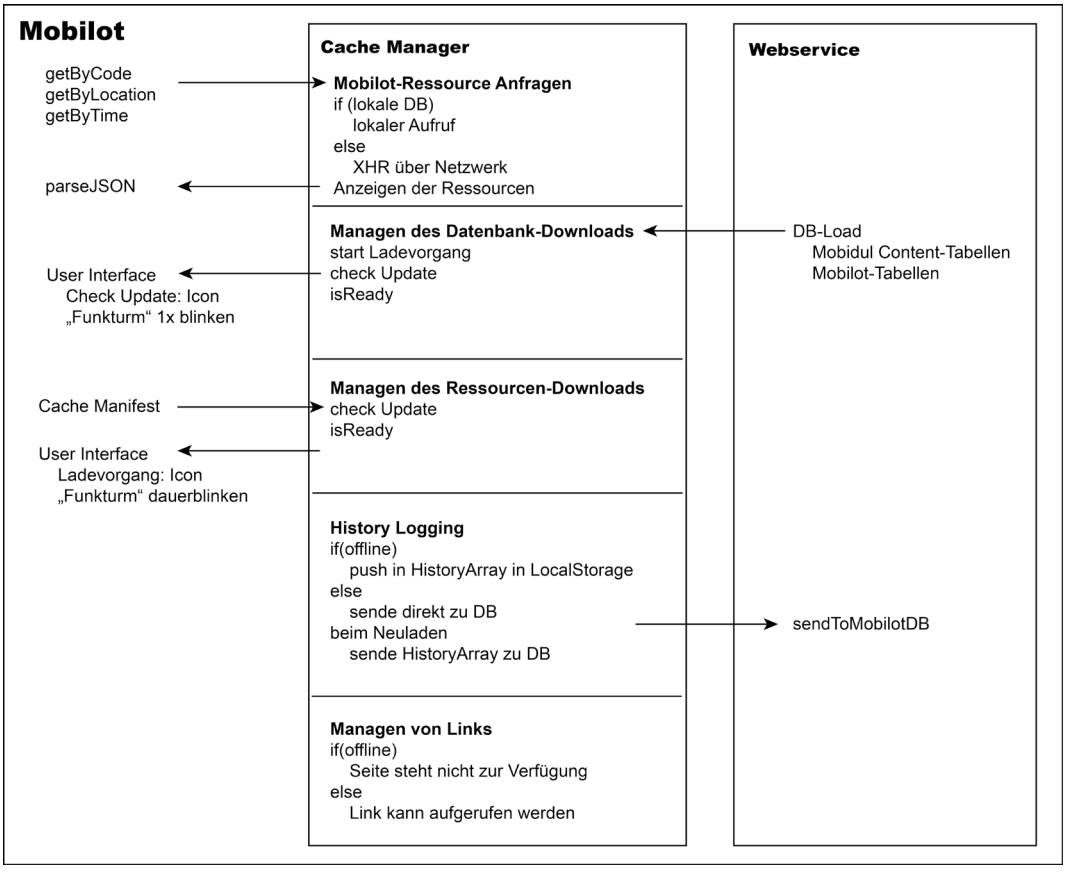

Abbildung 40: Funktionalität des Mobilot-Cache Managers

Der Mobilot-Cache Manager besteht aus zwei JavaScript-Dateien:

- Mobilot-Plugin: stellt alle notwendigen Funktionen zur Verfügung
- Datei mit der Funktion zur Initialisierung des Cache Managers

Da die Mobidule keinen direkten Zugriff auf den Mobilot-Namensraum haben, aber dennoch die Offline-Funktionalität benötigen, muss ein separater Cache Manager erstellt werden (siehe Abbildung 41). Dieser hat nur die nötigsten Funktionalitäten implementiert. Hauptfunktionalitäten sind das Laden der Inhaltsdaten und das Managen von Links.

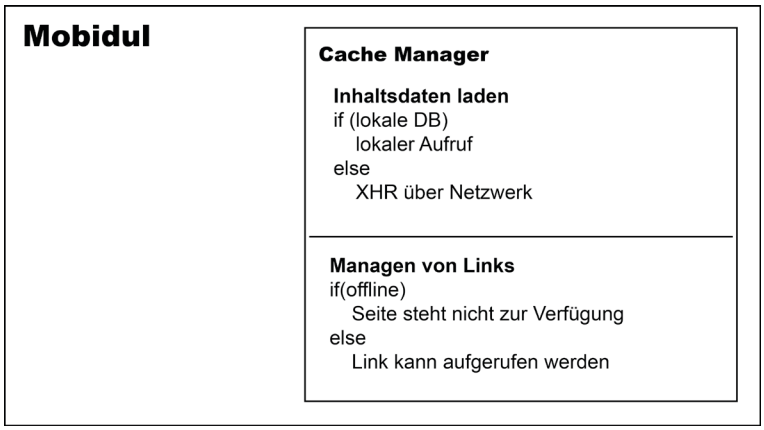

Abbildung 41: Funktionalität des Mobidul-Cache Managers

3.4.4.1. Managen des Ressourcen-Downloads

Ist eine Manifest-Datei in der Startdatei der Applikation verlinkt, übernimmt das Herunterladen der entsprechenden Ressourcen der Browser (vgl. Kapitel 2.7.1.1). Über Events des Application Cache (vgl. Kapitel 2.7.4.1) können jedoch bestimmte Zustände behandelt werden (siehe Listing 31). So kann im Fall des Startens des Downloads der Ressourcen (Event `downloading`) ein visuelles Element angezeigt werden, um dem User die Information zu geben, dass Daten geladen werden (1). In dieser Applikation wird ein Intervall genutzt, um einen blinkenden Funkturm im Header für die Dauer des Ladens der Ressourcen zu realisieren (siehe Abbildung 42 und Abbildung 43).

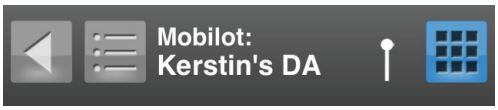

Abbildung 42: Header mit nicht aktiviertem Funkturm-Symbol

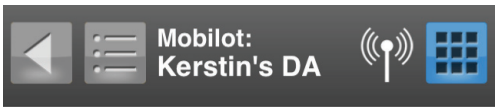

Abbildung 43: Header mit aktiviertem Funkturm-Symbol

Im Event-Listener für die Events `updateready` (alle Ressourcen sind fertig geladen) und `cached` (Ressourcen sind das erste Mal geladen) wird die Funktion `ressourcesreadyAction` aufgerufen (2), die in Listing 32 aufgeführt ist. Zum einen wird die Variable `ressourceready` auf `true` gesetzt (5) und so dem Cache Manager mitgeteilt, dass die Ressourcen fertig geladen sind. Zum anderen wird das Intervall für das visuelle Element (der blinkende Funkturm) gelöscht (6).

Außerdem behandelt die Funktion `checkManifest` (Listing 31), wann die Funktion `update` auf den Application Cache ausgeführt wird. Diese Funktion veranlasst den Browser zu überprüfen, ob sich die Manifest-Datei geändert hat. Bei der Entwicklung

erleichtert ein Intervall, das bspw. alle 10 Sekunden die Funktion auslöst, die Arbeit (3). Jedoch ist es auch im Produktivbetrieb sinnvoll, solch ein Intervall einzusetzen, vor allem wenn absehbar ist, dass sich die Dateien ändern (z.B. neue oder veränderte Bilder). Im Zusammenhang mit einer Aktualisierung der Seite nach Abschluss des Herunterladens ist so gewährleistet, dass die BenutzerInnen zeitnah die neuen Dateien zur Verfügung haben. Die Anwendung kann jedoch auch so konfiguriert werden, dass die Funktion `update` auf den Application Cache nur beim Neuladen der Seite ausgeführt wird (4).

Listing 31: Funktion checkManifest (aus mobicachemanager.js)

```
// Cache Manifest auf Änderungen überprüfen
checkManifest : function(updateManifest){
   // Event-Listener für Application Cache, Ressourcen werden geladen
   mobilot.cachemanager.cache.addEventListener(‚downloading‘,
         function(e){
      console.log(„Cachemanager: Lade Ressourcen“);
      mobilot.cachemanager.ressourcesready = false;
1     mobilot.cachemanager.setIntervalFunkturm();
   }, false);
   // Update ready
   mobilot.cachemanager.cache.addEventListener(‚updateready‘,
         function(e){
      mobilot.cachemanager.cache.swapCache();
2     mobilot.cachemanager.ressourcesreadyAction();
      window.location.reload();
   }, false);
   // Fertig mit erstem Mal laden
   mobilot.cachemanager.cache.addEventListener(‚cached‘, function(e){
2     mobilot.cachemanager.ressourcesreadyAction();
   },false);
   [...]
   // Manuelles Update
   if(updateManifest){
      setInterval(function(){
         if(mobilot.cachemanager.isOnline()){
3           mobilot.cachemanager.cache.update();
         }else{
            console.log(„Cachemanager: Check For Update Fail - Offline“);
         }
      }, 10000);
   }else{
4     mobilot.cachemanager.cache.update();
      console.log(„Cachemanager: Check For Update 1x“);
   }
},
```

Listing 32: Funktion ressourcesreadyAction (aus mobicachemanager.js)

```
// Wenn Laden der Resssourcen fertig ist
ressourcesreadyAction   :  function(){
5    mobilot.cachemanager.ressourcesready = true;

     // stop Animation
6    window.clearInterval(mobilot.cachemanager.animateFunkturm);
     mobilot.cachemanager.setFunkturm(true);

     [...]
}
```

Somit kann die Synchronisation der Ressourcen periodisch oder ad hoc erfolgen (vgl. Kapitel 2.2.5.2). Eine periodische Synchronisation findet statt, wenn das Intervall aktiv ist (Variable `updateManifest` ist `true`). In einer ad hoc-Synchronisation wird die Funktion `update` nur beim erneuten Laden der Seite ausgeführt.

3.4.4.2. Managen des Datenbank-Downloads

Bei der Initialisierung des Cache Managers werden die Inhalte der Datenbanken am Webserver über AJAX-Anfragen abgerufen (siehe Listing 33). Um Herauszufinden, ob Datensätze geändert wurden, wird aus dem Ergebnis der Abfrage ein MD5-Hash im Local Storage (vgl. Kapitel 2.7.4.3) gespeichert. Der neu erstellte MD5-Hash wird mit dem zugehörigen Wert im Local Storage verglichen (1). Sind die Werte abweichend oder ist die Variable `loadnew` auf `true` gesetzt, wird die entsprechende lokale Datenbanktabelle mit den neuen Datensätzen gefüllt. Sind die Datensätze gleich, wird die Variable `dbready` des Cache Managers direkt auf `true` gesetzt (2). Damit wird dem Cache Manager signalisiert, dass das Laden der Datenbanktabellen abgeschlossen ist.

Listing 33: Callback-Funktion der AJAX-Anfrage innerhalb der Funktion checkAllDBs (aus initCacheManager.js)

```
callback: function(){
   // check
   oldHash = typeof mobilot.getValue(„localDbHash_Content“) ===
      ‚undefined‘ ? ‚a‘ : mobilot.getValue(„localDbHash_Content“);
   newHash = md5(this.responseText);
1  if(oldHash != newHash || loadnew){
      // Nur wenn Unterschied dann Delete, Create und Insert
      [...]
   }else{
2     mobilot.cachemanager.dbready = true;
   }
}
```

Eine ständige Kontrolle der Datenbank wird durch die Funktion `checkForUpdateDB` (siehe Listing 34) zur Verfügung gestellt. Diese Funktion implementiert ein Intervall (1), das in einem definierbaren Zeitabstand (Variable `intervalCheckOnline`) kontrolliert, ob die Webanwendung eine Netzwerkverbindung hat und ob ein erneuter Check der Datenbank auf Änderungen notwendig ist. Dafür wird der Zeitpunkt des letzten Datenbank-Downloads (Variable `checkDB`) mit dem maximalen Alter, welches die Datenbank besitzen darf (Variable `alterDB`), addiert und mit der aktuellen Zeit verglichen (2). Ist der aktuelle Zeitpunkt größer als der berechnete Wert, wird eine Kontrolle der serverseitigen Datenbanken veranlasst (Funktionsaufruf `checkAllDBs`) (3).

Listing 34: Funktion checkForUpdateDB (aus mobicachemanager.js)

```
checkForUpdateDB  :  function(){
1    setInterval(function(){
        console.log(„checkOnline“);
        if(mobilot.cachemanager.isOnline()){
           // Check DB, wenn letzter Check checkDB her ist
           seconds = Math.round(new Date().getTime() / 1000);
2          if(seconds >= (mobilot.cachemanager.checkDB +
                 mobilot.cachemanager.alterDB)){
              mobilot.cachemanager.checkDB = seconds;
              console.log(„checkDBs“);
3             checkAllDBs();
4             mobilot.cachemanager.setIntervalFunkturm();
5             window.setTimeout(function(){
                 window.clearInterval(
                    mobilot.cachemanager.animateFunkturm);
                 mobilot.cachemanager.setFunkturm(true);
              },1500);
           }
           mobilot.cachemanager.setFunkturm(true);
        }else{
           mobilot.cachemanager.setFunkturm(false);
        }
    }, mobilot.cachemanager.intervalCheckOnline);
},
```

Beim Ausführen der Funktion `checkAllDBs` wird der User über ein visuelles Element über diesen Update-Versuch informiert. Dazu wird die Funktion `setIntervalFunkturm` verwendet (4), die bereits in Kapitel 3.4.4.1 genutzt wurde, um das Herunterladen der Ressourcen zu signalisieren. Im Falle des Datenbank-Update-Versuchs blinkt der Funkturm nur ein Mal. Realisiert wird dies über ein Timeout, welches nach 1,5 Sekunden das Intervall löscht (5).

3.4.4.3. Mobilot-Ressourcen Anfragen

Alle Anfragen, die bisher direkt über eine AJAX-Anfrage an den Webserver abgesetzt wurden, werden nun über den Cache Manager abgewickelt (siehe Listing 35). Dazu wird abgefragt, ob das Mobilot-Plugin und damit der Namensraum `cachemanager` exisitert (1). Ist dies der Fall, wird die entsprechende Funktion aufgerufen (2). Im Beispiel ist dies die Funktion `getRessourceByCode`. Ein Fallback, falls die JavaScript-Dateien des Cache Managers nicht inkludiert werden, ist jedoch weiterhin vorhanden (3).

Listing 35: Managen einer Mobilot-Ressourcen Anfrage (aus mobiscanner.js)

```
1 if(mobilot.cachemanager){
     console.log(„Cachemanager da“);
2    mobilot.cachemanager.getRessourceByCode(code);
3 }else{
     xui().xhr(mobilot.getRootURL()+‘/service.php?cmd=GetForCode&code=‘+
          code, {
        async: false,
        callback: function() {
           mobilot.parseRessourceXML(this.responseXML,‘code‘);
        }
     });
}
```

Der Cache Manager (siehe Listing 36) überprüft mit Hilfe der Variable `isCheckOffline` (1), ob der Browser des aufrufenden Gerätes alle notwendigen Funktionalitäten unterstützt. Diese Variable wird beim Laden des Mobiduls gesetzt. Hat die Variable den Wert `false`, wird wie gewohnt eine AJAX-Anfrage abgesetzt (5). Ist die Variable auf `true` gesetzt, wird abgefragt, ob die Datenbank fertig geladen ist (2). Ist dies nicht der Fall wird ein Timeout auf 1 Sekunde gesetzt und danach erneut überprüft (3). Erst wenn die lokale Datenbank erfolgreich geladen ist, wird die entsprechende Funktion des Mobilot-Plugins für den Datenbankzugriff aufgerufen (4). Der Zugriff auf dieses Plugin wird mit Hilfe der Funktion `whichDB` abstrahiert, um eine zukünftige Erweiterung auf andere Datenbankschnittstellen wie das Indexed Database API (siehe Kapitel 2.7.4.5) zu erleichtern. Die Funktion `whichDB` gibt entsprechend der unterstützten Datenbank den passenden Mobilot-Namensraum zurück.

Listing 36: Funktion getRessourceByCode (aus mobicachemanager.js)

```
getRessourceByCode :  function(code){
   // Wenn Offline-Verfügbarkeit
1  if(mobilot.cachemanager.isCheckOffline){
2     if(!mobilot.cachemanager.dbready){
         console.log(„set Timeout“ + code);
3        window.setTimeout(function(){[...]}, 1000);
      }else{
4        mobilot[mobilot.cachemanager.whichDB()].getForCode(code);
         mobilot.setCustomPropertyBag({„code“:code});
      }
   }else{
5     xui().xhr(mobilot.getRootURL()+
            ,/service.php?cmd=GetForCode&code=‘+code, {
         async: false,
         callback: function() {
            mobilot.parseRessourceXML(this.responseXML,‘code‘);
         }
      });
   }
}
```

3.4.4.4. History Logging

Die Funktionalität des History Loggings ist bereits in der Ausgangsversion von Mobilot enthalten. Jedoch muss es, auch während der User offline surft, möglich sein, die besuchten Seiten zu dokumentieren und diese Daten automatisch an die serverseitige Datenbank zu senden, sobald dieser wieder Internetzugang hat. Um dies zu realisieren, wird die entsprechende Funktion im Mobilot-Core verändert (siehe Listing 37). Ist das Cache Manager Plugin mit dem Namensraum `cachemanger` vorhanden, wird das Senden der Daten über den Cache Manager geregelt (1).

Listing 37: Funktion saveHistoryDB (aus mobilot.js)

```
saveHistoryDB : function (titel, url,note, type,gotByType) {
   delete ajaxdata;
   var date = new Date();
   var time = date.getFullYear() + „-“ + date.getMonth() + „-“ +
      date.getDate() + „ „ + date.getHours() + „:“ +
      date.getMinutes() + „:“ + date.getSeconds();
   var ajaxdata = {
      titel      :  titel,[...]
   }
1  if(mobilot.cachemanager){
      mobilot.cachemanager.setHistory(ajaxdata);
   }else{
      sir_send_alot([...]);
   }
}
```

Listing 38 zeigt wie die dazugehörige Funktion `setHistory` implementiert ist. Besteht keine Netzwerkverbindung (1), wird das Array, in welchem die History-Daten zwischengespeichert werden, mit Hilfe der Mobilot-Funktion `getValue` aus dem Local Storage geholt (2) und mit den entsprechenden Daten erweitert (3). Anschließend wird das Array mit Hilfe der Mobilot-Funktion `setValue` wieder in den Local Storage gespeichert (4).

Wird diese Funktion im Online-Fall aufgerufen, werden die Daten via AJAX direkt an die entsprechende Datenbank gesendet (5).

Listing 38: Funktion setHistory (aus mobicachemanager.js)

```
setHistory        :  function(ajaxdata){
   console.log(„Set History“);
   // Wenn offline unterwegs
1  if(!mobilot.cachemanager.isOnline()){
      console.log(„History im Local Storage Speichern“);
      // Wenn offline dann push in Array und save in Local Storage
2     historyArray = mobilot.getValue(„historyArray“);
3     historyArray.push(ajaxdata);
4     mobilot.setValue(„historyArray“, historyArray);
   }else{
      console.log(„Send HistoryTsoDB“);
5     mobilot.sendToMobilotDB(‚saveHistory2‘,ajaxdata, function(msg){
         [...]
         console.log(msg);
      },function(msg){
         console.log(msg);
         //fail
      });
   }
}
```

Die Funktion `saveHistoryInDB` übernimmt im Mobilot-Cache Manager das Senden der im History Array gespeicherten Daten, wenn der User die Anwendung wieder mit Netzwerkverbindung nutzt (siehe Listing 39). Dafür wird das Array aus dem Local Storage geholt (1) sowie jedes Item des Arrays via AJAX an die entsprechende Datenbank gesendet (2).

Listing 39: Funktion saveHistoryInDB (aus mobicachemanager.js)

```
// History nach Offline-Tätigkeit in Mobilot-DB speichern
saveHistoryInDB      :  function(){
   console.log(„SaveHistory“);
1  historyArray = mobilot.getValue(„historyArray“);
   console.log(historyArray);
   mobilot.setValue(„historyArray“, []);
   if(historyArray != null){
2     for(i=0; i<historyArray.length; i++){
         mobilot.sendToMobilotDB(„saveHistory2“, historyArray[i],
              function(msg){
            [...]
            console.log(msg);
         },function(msg){
            console.log(msg);
            //fail
         });
      }
   }
},
```

3.4.4.5. Managen von Links

Das Managen von Links ist notwendig, um zu verhindern, dass Seiten geladen werden, die während einer Offline-Nutzung nicht erreichbar sind.

Der einfachste Ansatz ist, die Linkadresse mit denen der Dateien zu vergleichen, die über das Cache Manifest geladen wurden, und dementsprechend zu wissen, ob diese Datei offline geladen werden kann oder nicht. Da jedoch derzeit bspw. über den Application Cache keine Möglichkeit besteht, auf die Dateinamen zuzugreifen, muss eine andere Lösung gefunden werden.

Auf Seiten des Frameworks Mobilot haben die Mobilot-Entwickler die Kontrolle und das Wissen darüber, welche Dateien im Cache Manifest enthalten sind. Demnach können gezielt die Verlinkungen, die offline nicht nutzbar sind, wie derzeit bspw. die Account-Verwaltung, mit der Funktion `checkLink` des Mobilot-Cache Managers abgefangen werden. Der Code dieser Funktion ist in Listing 40 angeführt. Es wird die Online-Verfügbarkeit kontrolliert (1) und je nach Fall die Seite geladen (2) oder eine Nachricht ausgegeben (3).

Listing 40: Funktion checkLink (aus mobicachemanger.js)

```
checkLink          :  function(url){
   console.log(„CheckLink“);
1  var check = mobilot.cachemanager.isOnline();
2  if(check){
      mainframe.attr(‚src‘,url);
      changePage(„ressourcepage“);
   }else{
3     alert(„Diese Seite steht Ihnen leider ohne Netzwerkverbindung
         nicht zur Verfügung!“);
   }
},
```

Auf Seiten des Mobiduls können die Mobilot-Entwickler keinen Einfluss darauf nehmen, wann welche Seite aufgerufen wird. Hier ist der Mobidul-Entwickler gefragt, darauf zu achten, dass alle notwendigen Dateien offline verfügbar sind. Sofern diese im selben Ordner bzw. einem Unterordner der Datei `manifest.php` liegen, werden sie dynamisch mit Hilfe dieser Datei gecacht. Werden jedoch externe Ressourcen als Links im Mobidul-Inhalt inkludiert, ist nicht mehr gewährleistet, dass diese Ressourcen auch offline zur Verfügung stehen. Deshalb werden alle Klicks auf Link-Tags (<a>) im Mobidul abgefangen (siehe Listing 41) und an die Funktion `checkLink` des Mobidul-Cache Managers übergeben (1). Diese Funktion überprüft, ob eine Netzwerkverbindung vorhanden ist (vgl. Listing 42). Ist dies der Fall, wird die aufgerufene Seite geladen (2). Wenn nicht wird den BenutzerInnen eine Information angezeigt, warum diese Seite nicht geladen werden kann (3).

Listing 41: Abfangen der Links (aus mobidul.js)

```
xui.ready(function(){
   if(mobidul.cachemanager){
      //Alle Links <a> abfangen
      xui(„a“).on(‚click‘, function(e) {
         e.preventDefault();
1        mobidul.cachemanager.checkLink(xui(this).attr(‚href‘));
      });
   }
});
```

Listing 42: Funktion checkLink (aus mobidul_cachemanager.js)

```
checkLink       : function(url){
   var check = mobidul.cachemanager.isOnline();
   if(check){
2     window.location.href = url;
   }else{
3     alert(„Diese Seite steht Ihnen leider ohne Netzwerkverbindung
         nicht zur Verfügung!“);
   }
}
```

3.4.4.6. Inhaltsdaten laden

Damit ein Mobidul offline verfügbar ist, reicht es nicht aus, sämtliche Mobilot-Ressourcen und –Daten im lokalen Speicher des Gerätes zu hinterlegen und auszulesen. Auch die Inhaltsdaten des Mobiduls, die in einer Datenbank hinterlegt sind, müssen lokal gespeichert und wieder ausgelesen werden. Das Speichern übernimmt der Mobilot-Cache Manager (vgl. Kapitel 3.4.4.2). Der Mobidul-Entwickler benötigt jedoch eine Schnittstelle, über die er die Inhaltsdaten lokal laden kann. Aus diesem Grund gibt es ein Mobidul-Plugin, welches die Funktionen zum Zugriff auf die lokale Datenbank bereitstellt (mobidul_dbsql.js). Die Funktion `getAllProjekts` (vgl. Listing 43) selektiert alle Datensätze einer Tabelle aus der Datenbank (2) und übergibt diese im Erfolgsfall an die Funktion `loadProjektListe` (3). Der Tabellenname muss als Parameter der Funktion `getAllProjekts` übergeben werden (1).

Listing 43: Funktion getAllProjekts (aus mobidul_dbsql.js)

```
1  getAllProjekts : function(tableName) {
       var database = db;
2      var sql = „SELECT * FROM „ + tableName;
       database.transaction(function(tx) {
3        tx.executeSql(sql, [], mobidul.dbsql.loadProjektListe,
           mobidul.dbsql.onError);
       });
     },
```

Die Funktion `loadProjektListe` (vgl. Listing 44) bereitet das Ergebnis der Datenbankabfrage so auf, dass es über die selbe Funktion (`render`) wie die Resultate der AJAX-Anfragen dargestellt werden kann.

Listing 44: Funktion loadProjektListe (aus mobidul_dbsql.js)

```
loadProjektListe: function (tx, rs) {
  var result = [];
  for (var i=0; i < rs.rows.length; i++) {
    result.push(rs.rows.item(i));
  }
  render(result); // definiert in der aufrufenden html-Datei
}
```

Listing 45 zeigt wie die Abfrage aus der HTML-Datei ohne das Vorhandensein einer lokalen Datenbank mit einem Fallback aussehen kann. Die Funktion `whichDB` liefert den Typ der lokalen Datenbank (1). Ist eine lokale Datenbank vorhanden, wird diese geöffnet und die entsprechenden Daten abgerufen (2). Für den Fall, dass keine lokale Datenbank vorhanden ist, wird eine AJAX-Anfrage an den Webservice abgesetzt (3).

Listing 45: Abfrage der lokalen Datenbank (aus imlab_overview.html)

```
   // Check ob lokale DB
1  var whichDB = mobidul.cachemanager.whichDB();
   if(whichDB != „“){
      // Hol aus lokaler DB
2     mobidul[whichDB].open();
      mobidul[whichDB].getAllProjekts(„imlabprojekt“);
   }else{
3     xui().xhr(„webservice.php?all=all“, {
         method:‘get‘,
         headers:[{name:‘Content-type‘, value: „application/json;“}],
         callback: function(){
            var json = JSON.parse(this.responseText);
            render(json);
         }
      });
   }
```

3.4.5. Aufgetretene Probleme

3.4.5.1. Cache Manifest-Datei

Das Laden des Local Cache of Documents wird über das Cache Manifest (vgl. Kapitel 2.7.1.1) realisiert. Bei diesem ist es nur dann möglich, ein Neuladen der Ressourcen zu erreichen, wenn die Manifest-Datei geändert wird. Dabei werden jedoch nicht nur die Dateien geladen, die sich geändert haben oder neu sind, sondern immer alle Dateien, die im Cache Manifest stehen.

Dadurch lässt sich die Bedingung aus Szenario 1, nur Updates auszuführen, lediglich beim Local Cache of Data realisieren, nicht aber für den Local Cache of Documents. Dieses Problem lässt sich programmierseitig nicht umgehen.

3.4.5.2. Keine GET-Parameter an URL möglich

Die ursprüngliche Umsetzung vom Inhalt des Mobiduls IM-Labor basierte auf einer URL-Struktur mit GET-Parametern. Dies stellte sich als Problem heraus, weil die URL, die im Cache Manifest hinterlegt ist, mit der aufgerufenen URL übereinstimmen muss. Sonst wird die Datei immer über das Netzwerk geladen. Demnach war eine Umsetzung mit GET-Parametern nicht möglich.

Zum einen musste eine Lösung für den Mobidul-Inhalt gefunden werden, um nicht für jedes Projekt eine Detailseite lokal speichern zu müssen. Zum anderen war es ebenfalls nötig, für das Framework Mobilot eine Alternative zu finden, denn der Aufruf eines bestimmten Codes, der sich bspw. hinter einem QR-Code verbirgt, erfolgte ebenfalls über GET-Parameter.

Lösung 1 – Mobilot

Es ist möglich einen Hash (#) an die URL zu hängen, der per JavaScript über `window.location.hash` ausgelesen werden kann.

Wird nun bspw. ein QR-Code mit der URL http://dm101504.students.fhstp.ac.at/da/#code=1 eingescannt, lädt bei Weiterleitung von einer Scanner-Anwendung an einem Android-Gerät der Browser immer die Webanwendung neu. An einem iOS-Gerät ist dies allerdings nicht der Fall, wenn die Datei bereits in einem Fenster geöffnet ist.

Deshalb muss eine weitere HTML-Datei erstellt werden, die per JavaScript lediglich eine Weiterleitung auf die eigentliche index.html mit dem entsprechenden Hash vornimmt (siehe Listing 46). Dementsprechend muss nun hinter einem QR-Code, der auf den Code 1 verweist, folgende URL versteckt sein: http://dm101504.students.fhstp.ac.at/da/index2.html#code=1.

Listing 46: Datei index2.html

```
<html>
   <head>
      <script>
         function init(){
            loc = window.location.hash;
            console.log(location);
            location.href = „index.html“+loc;
         }
      </script>
   </head>
   <body onload=“init()“>
   </body>
</html>
```

Lösung 2 - Mobidul Inhalt

Lösung 1 ist theoretisch auch für den Mobidul-Inhalt möglich. Durch die Struktur von Mobilot werden die Mobidul-Inhaltsseiten aber in ein iFrame geladen. Das automatische Neuladen des iFrames, wenn lediglich der Hash der URL geändert wird, funktioniert jedoch nicht.

Aus diesem Grund wird der Code, der das aufzurufende Projekt in der Datenbank identifiziert, im Local Storage gespeichert und in der Detailseite ausgelesen sowie dementsprechend die Datenbankabfrage ausgeführt. Voraussetzung dafür ist, dass der Mobilot-Code, der die Ressource in den Mobilot-Tabellen identifiziert, dem Schlüsselwert gleicht, der in der Inhaltstabelle verwendet wird.

Durch diese Änderung funktioniert jedoch die bisher in Mobilot implementierte Variante des Back-Buttons nicht mehr. Zur Identifizierung der vorherigen Seite wurde die URL verwendet. Die URL allein würde aber in diesem Fall zwar die Detailseite aufru-

fen, jedoch lediglich den als letztes im Local Storage gespeicherten Code in dieser Seite anzeigen. Nun wird zusätzlich der Code an den Titel angehangen, der ohnehin schon im entsprechenden Objekt abgelegt wurde. Bei Klick auf den Backbutton wird dieser ausgelesen und vor dem Wechsel auf die Detailseite im Local Storage hinterlegt.

3.4.5.3. Application Cache API

Es ist derzeit nicht möglich, über das Application Cache API auf die Namen und Adressen der heruntergeladenen Dateien zuzugreifen. Dies macht es unmöglich, bei dynamisch erstellten Manifest-Dateien clientseitig herauszufinden, ob die aufzurufende Datei im Cache gespeichert ist und offline aufgerufen werden kann.

Es kann lediglich der Prozess des Herunterladens einer Datei abgefangen werden, nicht aber deren Namen. Deshalb ist es auch nicht möglich, eine Liste der heruntergeladenen Dateien zu erstellen und für die weitere Verwendung bspw. im Local Storage zu speichern.

Eine denkbare Alternativlösung ist, durch das per PHP erzeugte Manifest eine Liste der eingebundenen Dateien serverseitig zu speichern und diese Liste in weiterer Folge per AJAX-Anfrage separat zu laden.

Die Mobidul-EntwicklerInnen müssen dennoch sicherstellen, dass alle Dateien, die offline verfügbar sein sollen, über das dynamisch erstellte Manifest erfasst werden. Dabei ist es auch möglich Dateien von anderen Domains einzubinden, diese müssen jedoch manuell in der Datei manifest.php eingefügt werden.

3.4.5.4. Event für Online und Offline

Im Working Draft zu HTML 5 sind Events beschrieben, die eintreten, wenn sich der Verbindungsstatus des Browsers ändert (vgl. Kapitel 2.7.4.2). Diese Events sind jedoch lediglich in Firefox und Internet Explorer implementiert und in den beiden relevanten mobilen Browsern nicht nutzbar. Solche Events würden bspw. den Umgang mit der Synchronisation der Datenbanken (vgl. Kapitel 3.4.4.2) erleichtern. Derzeit muss mit Hilfe der JavaScript-Funktion `setInterval` ein Intervall eingerichtet werden, dass in einem bestimmten Zeitraum prüft, ob eine Netzwerkverbindung existiert, um so abhängig vom Browser Status Aktionen auszuführen.

3.4.5.5. XMLHttpRequest mit xui

Die eingesetzte JavaScript-Bibliothek xui (vgl. Kapitel 3.2.1) behandelt den Status eines XMLHttpRequest nicht richtig.

Der Callback für den Erfolg wird ausgelöst, wenn der Status 200 oder 0 ist. Der Fehlerfall tritt ein, wenn der Status die Ziffern 4 oder 5 enthält. Auch wenn vor einer AJAX-Anfrage geprüft wird, ob der Browser online ist und nur in diesem Fall die

Anfrage durchgeführt wird, kann es vorkommen, dass auch im Falle einer fehlenden Internetverbindung eine Anfrage abgesetzt wird. In diesem Fehlerfall bekommt der XMLHttpRequest den Status 0, was jedoch keinen Erfolgswert darstellt. Aus diesem Grund wurde die Bibliothek so angepasst, dass nur im Fall des Status 200, was dem HTTP-Statuscode für eine erfolgreich ausgeführte Anfrage entspricht, der Callback für den Erfolgsfall ausgelöst wird. In allen anderen Fällen tritt der Fehlerfall ein.

3.5. Die Zukunft dieser Umsetzung

Das Framework Mobilot wird derzeit in parallel laufenden Diplomarbeiten architektonisch umgebaut. Die zweite Version wird nach dem MVC-Modell geschrieben und modular aufgebaut. Dadurch ist eine bessere Trennung zwischen essentiellen und nicht unbedingt notwendigen Teilen der Implementierung möglich. Zudem werden Erweiterungen einfacher zu realisieren und Teile austauschbar gestaltet sein.

Der für diese Diplomarbeit erstellte Cache Manager wird in der Version 2 von Mobilot als Plugin realisiert. Durch integrierte Optionen in den Ressourcen-Abfragen kann bspw. in Zukunft bestimmt werden, wie die Abfrage erfolgen soll.

4. Fazit

„Eines ist mobiles Internet ganz sicher nicht: ein bloßer Hype. Es ist vielmehr die wichtigste industrielle Entwicklung dieser und der kommenden Dekade.“

(Mohr, 2011)

4.1. Ergebnisse der Arbeit

Dieser Arbeit liegen folgende Fragen zu Grunde:

Welche Szenarien und Möglichkeiten gibt es, Applikationen, die aktuelle Daten nutzen wollen und daher eigentlich auf eine funktionierende Datenverbindung angewiesen sind, durch Caching-Techniken auch bei langsamen Verbindungen oder komplett ohne Netzwerkverbindung nutzen zu können?

Wie kann eine Webanwendung nach Szenario 1 umgesetzt werden?

Sind die Möglichkeiten, die HTML 5 bietet, geeignet, um die Anforderungen zu erfüllen?

Welche Probleme können bei der Umsetzung von webbasierten Offline-Anwendungen auftreten und warum?

In Kapitel 1.2 werden Untersuchungsszenarien definiert, die zwei grundlegende Anwendungsfälle für webbasierte mobile Informationssysteme unterscheiden:

1. Kein bzw. nur kurzzeitiger Internetzugriff vorhanden

Die Applikation muss beim ersten Aufruf alle benötigten Ressourcen und Daten auf dem Smartphone speichern. Der Großteil der Interaktion mit der Applikation findet ohne Netzwerkverbindung statt. Bei vorhandener Verbindung können Updates durchgeführt werden.

2. Antwortzeiten verbessern

Ziel einer Applikation für dieses Szenario ist es, Inhaltsdaten vorzuladen, um die Antwortzeiten zu verringern. Mögliche Eingrenzungen der Daten anhand des Kontextes, in dem sich der User befindet, sind in diesem Fall von Vorteil. Dies könnten z.B. die GPS-Position, Datum und Uhrzeit oder personenbezogene Daten sein.

Abbildung 44 gibt einen Überblick über die in Kapitel 2 erarbeiteten theoretischen Grundlagen zur Datensynchronisation.

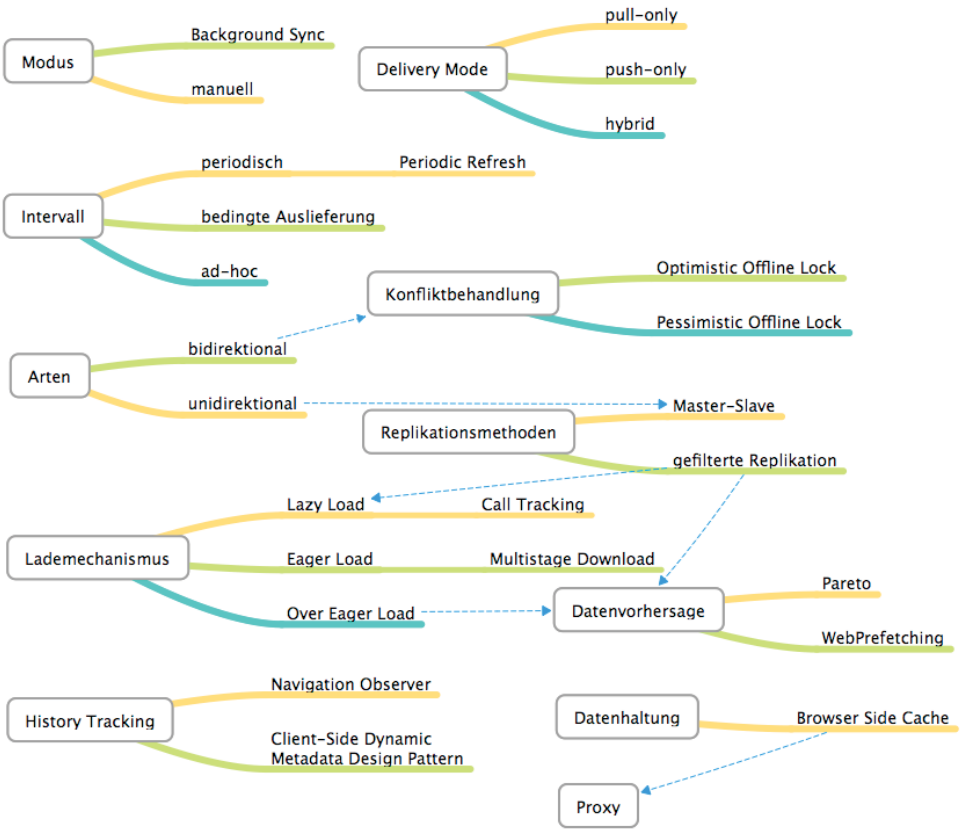

Abbildung 44: Datensynchronisation im Überblick

Vor der Implementierung eines webbasierten mobilen Informationssystems müssen grundlegende Entscheidungen getroffen werden:

- Welcher Synchronisationsmodus wird verwendet?
- Welcher Delivery Mode steht zur Verfügung?
- Welche Synchronisationsart wird gewählt?
- Ist Konfliktbehandlung notwendig?
- Welche Replikationsmethode ist relevant?
- Wie wird das Synchronisationsintervall implementiert?
- Welcher Lademechanismus ist zielführend?
- Wird eine Art der Datenvorhersage benötigt?

In weiterer Folge wird in Kapitel 3 in einem Proof of Concept gezeigt, dass mit den aktuell vorhandenen Möglichkeiten, die HTML 5 bietet, die gestellten Anforderungen aus Szenario 1 erfüllt werden können. Jedoch treten beim Arbeiten mit den JavaScript-Schnittstellen, die im Umfeld von HTML 5 entstehen, einige Probleme auf:

Zukunftsträchtiges Indexed Database API nicht nutzbar

Da das Indexed Database API bisher nicht in den relevanten Browsern (Mobile Safari und Android Browser) implementiert ist, musste die lokale Datenbank mit dem Web SQL Database API realisiert werden.

Neuladen der Ressourcen im Local Cache of Documents

Damit Ressourcen neu geladen werden, muss die Cache Manifest-Datei verändert werden. Das Cache Manifest kann jedoch auch dynamisch erstellt werden. So kann bspw. ein MD5-Hash der integrierten Dateien erzeugt und dieser als Kommentar angehangen werden.

Immer alle Dateien im Cache Manifest laden

Wird das Cache Manifest verändert, werden immer alle Dateien vom Browser neu angefordert. Dieses Problem lässt sich programmierseitig nicht umgehen, wenn man mit dem Manifest arbeitet, da es nicht möglich ist, nur einzelne Dateien neu laden zu lassen.

Keine GET-Parameter bei per Cache Manifest gecachten Dateien möglich

Die über einen GET-Request aufgerufene URL muss mit dem in der Manifest-Datei angegebenen Namen übereinstimmen. Dies wird zum Problem, wenn an einen GET-Request per „?“ Parameter angehangen werden. Sind die URLs nicht mit denselben Parameterangaben in der Manifest-Datei hinterlegt, sendet der Browser den Request zum Server. Da die Auswertung der Parameter aber nicht serverseitig, sondern clientseitig per JavaScript erfolgen soll, ist dies nicht erwünscht.

Lösung 1: Statt einen GET-Parameter mit „?“ an die URL zu hängen, kann dieser mit einem Hash (#) übergeben und per JavaScript mit `window.location.hash` ausgelesen werden.

Lösung 2: Der Wert des GET-Parameters kann bei Aufruf der entsprechenden Seite im Web Storage zwischengespeichert und auf der Zielseite wieder ausgelesen werden.

Kein Zugriff auf den Namen der gecachte Dateien durch Application Cache API

Bei dynamisch erstellten Manifest-Dateien muss eine Liste der eingebundenen Dateien serverseitig gespeichert werden. Diese kann anschließend per AJAX separat geladen werden. Die volle Kontrolle würde jedoch nur eine clientseitig erzeugte Liste mit den tatsächlich gecachten Dateien gewährleisten, die das Application Cache API zur Verfügung stellen könnte.

Keine Events für Browser Status

Um eine Änderung des Browser Status zu erkennen, muss ein Intervall definiert werden, welches den Status überprüft. Die entsprechenden Events sind in den verwendeten Browser bisher nicht implementiert.

4.2. Empfehlungen zur Umsetzung einer offline-fähigen Webanwendung

Die folgenden neun Empfehlungen sollen WebentwicklerInnen bei der erfolgreichen Umsetzung einer offline-fähigen Webanwendung unterstützen:

1. Cache Manifest nutzen

Das Cache Manifest (vgl. Kapitel 2.7.1.1) ermöglicht die Nutzung einer Anwendung ohne Internetverbindung. Dafür müssen Programmdateien, die auf der Clientseite gecacht werden sollen, in einer clientseitig interpretierbaren Sprache geschrieben sein: HTML und JavaScript.

2. Webservice für Kommunikation mit serverseitiger Datenbank

Die Kommunikation mit der serverseitigen Datenbank kann über einen Webservice realisiert werden, der per AJAX angesprochen wird.

3. Kommentar im Cache Manifest dynamisch generieren

Damit die durch das Cache Manifest gecachten Ressourcen aktualisiert werden, muss sich die Manifest-Datei verändern. Ein dynamisch generierter Kommentar, der bspw. MD5-Hashes der integrierten Dateien enthält, erleichtert diesen Update-Prozess.

4. Keine GET-Parameter bei gecachten Dateien verwenden

GET-Parameter können bei gecachten Dateien nicht verwendet werden, da der im Cache Manifest angegebene Dateiname mit dem aufzurufenden Namen übereinstimmen muss. Die Parameter können als Hash-Wert mit „#“ an die URL gehangen oder der Web Storage zum Zwischenspeichern der Parameter verwenden werden.

5. Intervall für Update des Cache Manifest (für Entwicklungsprozess)

Vor allem im Entwicklungsprozess hilft das Setzen eines Intervalls zum Updaten des Cache Manifest. Dieses Intervall beinhaltet die regelmäßige Überprüfung des Cache Manifest auf Änderungen (`update()`) und nach erfolgreichem Download den Wechsel zum aktuellen Cache (`swap()`).

6. Maximales Alter der lokalen Datenbank festlegen

Um das Synchronisationsintervall der Datenbank zu definieren, sollte das maximale Alter der lokalen Datenbank bestimmt werden. Innerhalb eines bestimmten Intervalls wird anschließend überprüft, ob dieses Alter erreicht ist und dementsprechend kontrolliert werden muss, ob Änderungen an der serverseitigen Datenbank erfolgt sind.

7. Änderungen der Datensätze mit MD5-Hash erkennen

Bei überschaubaren Datenmengen kann ein MD5-Hash des Resultats der serverseitigen Datenbankabfrage im Local Storage zwischengespeichert werden, um Änderungen an den Datensätzen zu erkennen.

8. Local Storage zum Zwischenspeichern für den Offline-Fall verwenden

Wenn keine Netzwerkverbindung besteht, müssen die für den Server relevanten Daten bzw. eingegebene User-Daten im Local Storage zwischengespeichert werden. Sobald wieder eine Verbindung verfügbar ist, sollten die entsprechenden Daten automatisiert an den Server gesendet werden.

9. Externe (nicht gecachte) Links überprüfen

Externe (nicht gecachte) Links sollten abgefangen werden, um im Offline-Fall dem User eine Nicht-Verfügbarkeits-Meldung anzeigen zu können.

Außerdem ist bei der Entwicklung von Webanwendungen mit HTML 5 zu beachten, dass HTML 5 noch nicht standardisiert ist und demnach Veränderungen (vor allem in der Implementierung durch die Browser) möglich sind. Im Mai 2011 war der Last Call, der die letzte Möglichkeit darstellte, Änderungswünsche für die Spezifikation einzubringen. Laut W3C ist der Status Recommendation für 2014 geplant. (vgl. W3C, 2011) Dieser Zeitpunkt ist jedoch nicht unbedingt gleichbedeutend mit einer vollständigen Unterstützung durch die Browser.

Dennoch zeigt diese Arbeit, dass vor allem in mobilen Webanwendungen bereits jetzt mit HTML 5 gearbeitet werden kann.

Literaturverzeichnis

Alle Internetquellen sind auf der beiliegenden CD im Ordner Internetquellen dokumentiert. Diese sind als Unterordner mit der jeweiligen Zitierbezeichnung abgelegt.

A1 Telekom Austria AG. (2012, April). Das Giganetz von A1 | A1.net. Retrieved May 8, 2012, from http://www.a1.net/hilfe-support/netzabdeckung/

Accenture. (2011). Die Chancen der mobilen Evolution. Retrieved from http://www.accenture.com/SiteCollectionDocuments/Local_Germany/PDF/Accenture-Studie-Mobile-Web-Watch-2011.pdf

Alby, T. (2008). *Das mobile Web* (1. Aufl.). München: Hanser Carl.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: towns, buildings, construction*. Oxford University Press.

Ameling, M. (2009, July 9). *Systemunterstützung für den Abgleich von Geschäftsobjekten zwischen Anwendungsservern über Web Services*. Technische Universität Dresden, Dresden. Retrieved from http://d-nb.info/1007708700/34

Ananthanarayanan, G., Haridasan, M., Mohomed, I., Terry, D. & Thekkath, C. A. (2009). StarTrack. *7th international conference on Mobile systems, applications and services* (p. 207). Presented at the MobiSys'09, Krakow, Poland: ACM Press. doi:10.1145/1555816.1555838

Benson, E., Marcus, A., Karger, D. & Madden, S. (2010). Sync kit. *19th international conferens on World wide web* (p. 121). Presented at the WWW '10, Raleigh, North Caroline, USA: ACM Press. doi:10.1145/1772690.1772704

Berners-Lee, T. (1992). Tags used in HTML. Retrieved April 29, 2012, from http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html

Berners-Lee, T. (2011). Longer Bio for Tim Berners-Lee. Retrieved April 29, 2012, from http://www.w3.org/People/Berners-Lee/Longer.html

Blumenstein, K. (2012). Cache Manifest dynamisch erstellen - Mobile Forschungsgruppe. *Mobile Forschungsgruppe*. Retrieved April 30, 2012, from http://mfg.fhstp.ac.at/development/cache-manifest-dynamisch-erstellen/

Bonino, D., Corno, F. & Squillero, G. (2003). A real-time evolutionary algorithm for Web prediction. *International Confernce on Web Intelligence* (pp. 139–145). Presented at the WI 2003, IEEE Comput. Soc. doi:10.1109/WI.2003.1241185

Boodman, A. (2011, November 3). Gears API Blog: Stopping the Gears. *Gears API Blog*. Retrieved March 10, 2012, from http://gearsblog.blogspot.com/2011/03/stopping-gears.html

Browserscope. (2012). Home - Browserscope. Retrieved May 13, 2012, from http://www.browserscope.org/?category=network&v=top

Chantelau, K. & Brothuhn, R. (2009). *Multimediale Client-Server-Systeme*. Gabler Wissenschaftsverlage.

Charland, A. & LeRoux, B. (2011). Mobile Application Development: Web vs. Native. *Queue*, *9*(4), 20. doi:10.1145/1966989.1968203

Cheng-Zhong Xu & Ibrahim, T. I. (2003). Towards semantics-based prefetching to reduce Web access latency (pp. 318–325). Presented at the SAINT'03 - Symposium on Applications and the Internet, IEEE Comput. Soc. doi:10.1109/SAINT.2003.1183065

Connolly, D. (2004). Overview of SGML Resources. Retrieved April 29, 2012, from http://www.w3.org/MarkUp/SGML/

Crane, D. & McCarthy, P. (2008). *Comet and reserve Ajax : the next-generation Ajax 2.0*. Berkeley, Calif.: Apress.

Crockford, D. & Klicman, P. (2008). *Das Beste an JavaScript : [erkunden Sie die Stärken von Java Script]*. Beijing; Cambridge; Farnham; Köln; Sebastopol; Taipei; Tokyo: O'Reilly.

Davison, B. D. (2002). Predicting web actions from HTML content. *thirteenth ACM conference on Hypertext and hypermedia* (p. 159). Presented at the HYPERTEXT '02, College Park, Maryland, USA: ACM Press. doi:10.1145/513338.513380

de la Ossa, B., Gil, J. A., Sahuquillo, J. & Pont, A. (2007). Improving Web Prefetching by Making Predictions at Prefetch. *3rd EuroNGI Conference on Next Generation Internet Networks* (pp. 21–27). Presented at the 3rd EuroNGI Conference on Next Generation Internet Networks, IEEE. doi:10.1109/NGI.2007.371193

Dengler, F., Henseler, W. & Zimmermann, H. (2001). Mobile Informationssysteme - Hard- und Softwaregestaltung im sozialen Kontext. In H. Oberquelle, R. Oppermann, & J. Krause (Eds.), *Mensch & Computer 2001* (pp. 385–386). Presented at the 1. Fachübergreifende Konferenz, Stuttgart: B.G. Teubner. Retrieved from http://mc.informatik.uni-hamburg.de/konferenzbaende/mc2001/W1.pdf

Deveria, A. (2012a). When can I use... JSON parsing. Retrieved May 7, 2012, from http://caniuse.com/#feat=namevalue-storage

Deveria, A. (2012b). When can I use... IndexedDB. Retrieved April 30, 2012, from http://caniuse.com/#feat=indexeddb

Deveria, A. (2012c). When can I use... Offline Apps. Retrieved April 30, 2012, from http://caniuse.com/#feat=offline-apps

Deveria, A. (2012d). When can I use... Web SQL Database. Retrieved April 30, 2012, from http://caniuse.com/#feat=sql-storage

Deveria, A. (2012e). When can I use... Web Storage. Retrieved April 30, 2012, from http://caniuse.com/#feat=namevalue-storage

Deveria, A. (2012f). When can I use... Web Sockets. Retrieved April 30, 2012, from http://caniuse.com/#feat=websockets

Domenech, J., Gil, J. A., Sahuquillo, J. & Pont, A. (2006). DDG: An Efficient Prefetching Algorithm for Current Web Generation. *1st IEEE Workshop on Hot Topics in Websystems and Technologies, 2006* (pp. 1–12). Presented at the HOTWEB '06, IEEE. doi:10.1109/HOTWEB.2006.355260

Eitler, T., Blumenstein, K. & Schmiedl, G. (2011). Mobilot: Architektur und technische Optionen bei der Entwicklung eines mobilen Informationssystems. *Proceedings of 4. Forum Medientechnik*. Presented at the 4. Forum Medientechnik, St. Pölten, Austria.

Ellis, C. (1977). A Robust Algorithm for Updating Duplicate Databases. *2nd Berkeley Workshop on Distributed Data Management and Computer Networks* (pp. 146–158). Berkeley, CA, USA.

Fan, L., Cao, P., Lin, W. & Jacobson, Q. (1999). Web prefetching between low-bandwidth clients and proxies. *1999 ACM SIGMETRICS International conference on Measurement and modeling of computer systems* (pp. 178–187). Presented at the SIGMETRICS '99, Atlanta, Georgia, USA: ACM Press. doi:10.1145/301453.301557

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine. Retrieved from http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Firtman, M. (2010). *Programming the mobile web*. Farnham: O'Reilly.

Fling, B. (2009). *Mobile design and development* (1st ed.). Beijing; Sebastopol Calif.: O'Reilly.

Floyd, R., Housel, B. & Tait, C. (1998). Mobile Web access using eNetwork Web Express. *IEEE Personal Communications*, *5*(5), 47–52. doi:10.1109/98.729724

Fowler, M. (2003). *Patterns für Enterprise-Application-Architekturen* (1. Aufl.). Bonn: Mitp.

Gamma, E., Helm, R. & Johnson, R. (2001). *Entwurfsmuster . Elemente wiederverwendbarer objektorientierter Software* (2. Aufl.). Addison-Wesley.

Garrett, J. J. (2005, February 18). Ajax: A New Approach to Web Applications - Adaptive Path. *Adaptive Path*. Retrieved January 3, 2012, from http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

Global Intelligence Alliance. (2010, April). Native or Web Application? How Best to Deliver Content and Services to Your Audiences over the Mobile Phone. Retrieved from http://www.globalintelligence.com/insights-analysis/white-papers/native-or-web-application-how-best-to-deliver-cont

Gonçalves, E. E. M. & Leitão, A. M. (2009). Using common Lisp to prototype offline work in web applications for rich domains (pp. 18–27). ACM Press. doi:10.1145/1562868.1562871

Google. (n.d.-a). Gears API - Google Code. *Gears API*. Retrieved March 10, 2012, from http://code.google.com/intl/de-DE/apis/gears/

Google. (n.d.-b). Architecture - Gears API - Google Code. *Gears API*. Retrieved March 10, 2012, from http://code.google.com/intl/de-DE/apis/gears/architecture.html

Griffioen, J. & Appleton, R. (1994). Reducing file system latency using a predictive approach. *USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94 (pp. 13–13). Presented at the USTC'94, Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1267257.1267270

Hermann, A., Arminger, K., Feld, T., Müller, R., Sengera, J., Tolsdorff, S., Spiegel, S. W. et al. (2011, Oktober). German Entertainment and Media Outlook 2011–2015: Onlinewerbung übernimmt Pole-Position. *PwC*. Frankfurt am Main. Retrieved from http://www.pwc.de/de/technologie-medien-und-telekommunikation/german-entertainment-media-outlook-2011.jhtml

Hickson, I. (2010, November 18). Web SQL Database. *W3C*. Retrieved April 14, 2012, from http://www.w3.org/TR/webdatabase/

Hickson, I. (2011, August 12). The WebSocket API. Retrieved April 30, 2012, from http://www.w3.org/TR/websockets/

Hickson, I. (2012a). HTML 5 - Working Draft. Retrieved April 29, 2012, from http://www.w3.org/TR/2012/WD-html5-20120329/

Hickson, I. (2012b). 5.6 Offline Web applications — HTML 5. Retrieved May 17, 2012, from http://www.w3.org/TR/html5/offline.html#offline

Hickson, I. (2012c). Web Storage. Retrieved April 30, 2012, from http://dev.w3.org/html5/webstorage/

Hickson, I. (2012d). 8 Communication — HTML 5. Retrieved May 17, 2012, from http://www.w3.org/TR/html5/comms.html

Hickson, I. (2012e, March 13). Web Workers. Retrieved April 30, 2012, from http://www.w3.org/TR/workers/

Hickson, I. & Hyatt, D. (2008). HTML 5 - Working Draft. Retrieved April 29, 2012, from http://www.w3.org/TR/2008/WD-html5-20080122/

Hogan, B. P. (2011). *HTML 5 & CSS3: Webentwicklung mit den Standards von morgen* (1st ed.). O'Reilly.

IBM. (2001). *Enterprise replication: a high-performance solution for distributing and sharing information*. San Jose, CA, USA: IBM Corporation. Retrieved from http://www.iiug.org/library/ids/whitepaper/GC27-1570-00.pdf

Ipsos & MMA. (2012). Our Mobile Planet. Retrieved May 19, 2012, from http://www.ourmobileplanet.com/de/graph/?country=at&category=DETAILS&topic=DETAILS_LOCA&stat=LOCA01&stat=LOCA02&stat=LOCA03&stat=LOCA04&stat=LOCA05&stat=LOCA06&stat=LOCA07&stat=LOCA08&stat=LOCA09&stat=LOCA10&stat=LOCA11&stat=LOCA97&wave=wave2&age=all&gender=all&active=wave

Jane, F. M. M., Ilayaraja, N., Raghav, M. A., Nadarajan, R. & Maytham, S. (2009). Cache prefetch and replacement with dual valid scopes for location dependent data in mobile environments. *11th International Conference on Information Integration and Web-based Applications & Services* (p. 364). Presented at the iiWAS '09, ACM Press. doi:10.1145/1806338.1806404

Jazayeri, M. (2007). Some Trends in Web Application Development. *Future of Software Engineering, 2007* (pp. 199–213). Presented at the FOSE '07, Minneapolis, MN: IEEE. doi:10.1109/FOSE.2007.26

JSON. (n.d.-a). JSON. Retrieved May 7, 2012, from http://www.json.org/

JSON. (n.d.-b). JSON in JavaScript. Retrieved May 7, 2012, from http://www.json.org/js.html

Kaazing Corporation. (2012). WebSocket.org | About WebSocket. Retrieved May 17, 2012, from http://www.websocket.org/aboutwebsocket.html

Kaikkonen, A. (2008). Full or tailored mobile web- where and how do people browse on their mobiles? *Proceedings of the International Conference on Mobile Technology, Applications, and Systems - Mobility '08* (p. 1). Presented at the the International Conference, Yilan, Taiwan. doi:10.1145/1506270.1506307

Kao, Y.-W., Tsai, C.-T., Chow, T.-H. & Yuan, S.-M. (2009). An offline browsing system for mobile devices (p. 338). ACM Press. doi:10.1145/1806338.1806400

Kern, F. (2012). Native App oder Web App? Teil 1: Definitionen und Entscheidungskriterien | smart digits. *Smart Digits*. Retrieved April 29, 2012, from http://www.smart-digits.com/2012/02/native-app-oder-web-app-teil-1-definitionen-und-entscheidungskriterien/

Kessin, Z. (2011). *Programing web applications in HTML 5*. Sebastopol CA: O'Reilly Media, Inc.

Khan, J. I. & Qingping Tao. (2003a). Exploiting Webspace organization for accelerating Web prefetching. *IEEE/WIC International Conference on Web Intelligence* (pp. 89–95). Presented at the WI'03, IEEE Comput. Soc. doi:10.1109/WI.2003.1241178

Khan, J. I. & Qingping Tao. (2003b). Webspace surfing patterns and their impact on Web prefetching. *2003 International Conference on Cyberworlds* (pp. 478–485). Presented at the CW'03, IEEE Comput. Soc. doi:10.1109/CYBER.2003.1253493

Kroeger, R. (2009, June 25). Gmail for Mobile HTML 5 Series: Cache Pattern For Offline HTML 5 Web Applications - The official Google Code blog. *The official Google Code blog*. Retrieved January 7, 2012, from http://googlecode.blogspot.com/2009/06/gmail-for-mobile-html5-series-cache.html

La, H. J., Ho Joong Lee & Kim, S. D. (2011). An efficiency-centric design methodology for mobile application architectures. *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications* (pp. 272–279). Presented at the WiMob, Wuhan: IEEE. doi:10.1109/WiMOB.2011.6085388

Laukat, A. (1999, February 9). Friedhof der Eliten. *Die Zeit*. Retrieved from http://www.zeit.de/1999/36/199936.pareto_.xml

Lehner, F. (2003). *Mobile und drahtlose Informationssysteme: Technologien, Anwendungen, Märkte*. Springer.

Luo, T., Hao, H., Du, W., Wang, Y. & Yin, H. (2011). Attacks on WebView in the Android system. *27th Annual Computer Security Applications Conference* (p. 343). Presented at the ACSAC '11, Orlando, Florida, USA: ACM Press. doi:10.1145/2076732.2076781

Mac OS Forge. (n.d.). The WebKit Open Source Project - WebKit Project Goals. Retrieved May 16, 2012, from http://www.webkit.org/projects/goals.html

Mahemoff, M. (2006). *Ajax design patterns* (1st ed.). Sebastopol CA: O'Reilly.

Markatos, E. P. & Chronaki, C. E. (1998). A Top-10 Approach to Prefetching on the Web. *In INET*.

Mason, A. (2010, August 18). Lazy and Eager Loading | Remote Connection. *Remote Connection*. Retrieved March 4, 2012, from http://blog.aaronmason.co.uk/lazy-and-eager-loading/

Mehta, N., Sicking, J., Graff, E., Popescu, A. & Orlow, J. (2011, June 12). Indexed Database API. *W3C*. Retrieved April 14, 2012, from http://www.w3.org/TR/IndexedDB/

Mesbah, A. & Deursen, A. (2007). An Architectural Style for Ajax. *The Working IEEE/IFIP Conference on Software Architecture, 2007* (pp. 9–9). Presented at the WICSA '07, Mumbai: IEEE. doi:10.1109/WICSA.2007.7

Mobasher, B., Dai, H., Luo, T. & Nakagawa, M. (2001). Effective personalization based on association rule discovery from web usage data. *3rd international workshop on Web information and data management* (p. 9). Presented at the WIDM '01, Atlanta, GA, USA: ACM Press. doi:10.1145/502932.502935

mobilkom Austria. (2000, February 16). GPRS und WAP: mobilkom austria hat den Sprung in die Zukunft schon genommen - GPRS und WAP: mobilkom austria hat den Sprung in die Zukunft schon genommen. Retrieved May 17, 2012, from http://www.pressetext.com/news/20000216007

mobilkom Austria. (2006, January 20). A1 startet HSDPA-Netz: Verkauf der ersten HSDPA-Datenkarten läuft an - A1 startet HSDPA-Netz: Verkauf der ersten HSDPA-Datenkarten läuft an. Retrieved May 19, 2012, from http://www.pressetext.com/news/20060120029

Mohr, N. (2011). Die Chancen der mobilen Evolution. Retrieved from http://www.accenture.com/SiteCollectionDocuments/Local_Germany/PDF/Accenture-Studie-Mobile-Web-Watch-2011.pdf

Münz, S. (2008). *Webseiten professionell erstellen. Programmierung, Design und Administration von Webseiten, m. DVD-ROM* (3., überarbeitete und erweiterte Auflage.). Addison-Wesley, München.

Nanopoulos, A., Katsaros, D. & Manolopoulos, Y. (2003). A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, *15*(5), 1155–1169. doi:10.1109/TKDE.2003.1232270

Ozsu, M. T. & Valduriez, P. (2011). *Principles of Distributed Database Systems*. Springer.

Padmanabhan, V. N. & Mogul, J. C. (1996). Using predictive prefetching to improve World Wide Web latency. *ACM SIGCOMM Computer Communication Review*, *26*(3), 22–36. doi:10.1145/235160.235164

Pakalski, I. (2002, January 16). WebToGo - Kostenloser On- und Offline-Browser für PalmOS - Golem.de. *Golem.de*. Retrieved March 10, 2012, from http://www.golem.de/0201/17796.html

Pareto, V. (1935). *The Mind and Society: Theory of derivations*. Jonathan Cape.

Pataky, H. (2005). *Integration einer lokalen Datenhaltung und Datensynchronisierung in einer Mehr-Schichten-Software Architektur* (Diplomarbeit). FH Oberösterreich, Hagenberg.

pdwb. (n.d.). Bevölkerungsentwicklung Deutschlands. Retrieved May 19, 2012, from http://www.pdwb.de/kurz_deu.htm

phaydon, research+ consulting GmbH, denkwerk, & Interrogare. (2010). *Mobile Commerce Insights 2010*. Retrieved from http://www.ibusiness.de/wrapper.cgi/www.ibusiness.de/files/jb_1045233303_1287044000.pdf

PhoneGap. (2012). Supported Features « PhoneGap. Retrieved April 29, 2012, from http://phonegap.com/about/features

Plucker. (2003). Plucker Desktop. *Plucker Desktop*. Retrieved April 18, 2012, from http://desktop.plkr.org/

Plucker. (2005, October 7). How Plucker Works - Plucker Documentation. *Plucker Documentation*. Retrieved March 10, 2012, from http://docs.plkr.org/index.php/How_Plucker_Works

Plucker. (2010a). Plucker: The best HTML and ebook reader for Palm and Windows mobile devices. *Plucker*. Retrieved April 18, 2012, from http://www.plkr.org/

Plucker. (2010b, July 27). Main Page - Plucker Documentation. *Plucker Documentation*. Retrieved March 10, 2012, from http://docs.plkr.org/index.php/Main_Page

Pons, A. P. (2006). Object prefetching using semantic links. *ACM SIGMIS Database*, *37*(1), 97–109. doi:10.1145/1120501.1120508

PwC. (2010). pwc.de: Partner im weltweiten Verbund. *pwc.de: Partner im weltweiten Verbund*. Retrieved April 28, 2012, from http://www.pwc.de/de/unternehmensinformationen/profil.jhtml

PwC & Wilkofsky Gruen Associates. (2011, Oktober). Mobiles Internet: Nutzer in Deutschland bis 2015 | Prognose. Retrieved April 14, 2012, from http://ezproxy.fhstp.ac.at:2078/statistik/daten/studie/180578/umfrage/anzahl-der-nutzer-des-mobilen-internets-in-deutschland-seit-2005/

Qui, X. (2010). *A Publish-Subscribe System for Data Replication and Synchronization Among Integrated Person-Centric Information Systems* (Master). Utah State University, Logan, Utah. Retrieved from http://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1616&context=etd

Rehm, R. (2011, January 18). XmlHttpRequest und die Worker API • Peter Kröner • Webtechnologie. Retrieved May 13, 2012, from http://www.peterkroener.de/xmlhttprequest-und-die-worker-api/

Rieber, P. (2009). *Dynamische Webseiten in der Praxis: Mit PHP 5, MySQL 5, XHTML, CSS, JavaScript und Ajax*. Hüthig Jehle Rehm.

Rivest, R. (1992). The MD5 Message-Digest Algorithm. Retrieved May 11, 2012, from http://tools.ietf.org/html/rfc1321

Robbins, J. N. (2008). *Webdesign mit (X)HTML und CSS: Ein Praxisbuch zum Einsteigen, Auffrischen und Vertiefen* (1. ed.). O'Reilly Verlag.

Rossi, G., Garrido, A. & Carvalho, S. (1996). Design patterns for object-oriented hypermedia applications. *Pattern languages of program design 2* (pp. 177–191). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. Retrieved from http://dl.acm.org/citation.cfm?id=231958.232948

Roth, G. (2010, April 26). HTML 5 Server-Push Technologies, Part 2 | Java.net. Retrieved May 17, 2012, from http://today.java.net/article/2010/04/26/html5-server-push-technologies-part-2

Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H. & Steere, D. C. (1990). Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, *39*(4), 447–459. doi:10.1109/12.54838

Schmidt, S. (2006). *PHP Design Patterns : [Entwurfsmuster für die Praxis ; deckt PHP 5.1 ab]*. Beijing [u.a.]: O'Reilly.

Schmiedl, G. (2010). Mobile Web oder Mobile App - Mobile Forschungsgruppe. *Mobile Forschungsgruppe*. Retrieved April 29, 2012, from http://mfg.fhstp.ac.at/ueberblick/mobile-web-oder-mobile-app/

Schmiedl, G. (2011). Mobilot | Fit für Forschung. Retrieved May 5, 2012, from http://www.fit-fuer-forschung.eu/mobilot

Schmitzer, D. (2011, Mai). WebSocket-Implementierung mit PHP. Retrieved May 17, 2012, from http://it-republik.de/php/artikel/WebSocket-Implementierung-mit-PHP-3817.html

Schwichtenberg, H. (2012). Eager Loading - Begriffserklärung im Entwickler-Lexikon/Glossar auf www.IT-Visions.de. *Entwickler-Lexikon*. Lexikon. Retrieved March 4, 2012, from http://www.it-visions.de/glossar/alle/6217/6217.aspx

SELFHTML. (2007a). SELFHTML: Einführung / Internet und WWW / Entstehung des World Wide Web. Retrieved April 29, 2012, from http://de.selfhtml.org/intro/internet/www.htm

SELFHTML. (2007b). SELFHTML: Einführung / Web-Technologien / HTML. Retrieved April 29, 2012, from http://de.selfhtml.org/intro/technologien/html.htm

Sombart, W. (1967). *Die drei Nationalökonomien: Geschichte und System der Lehre von der Wirtschaft*. Duncker & Humblot.

Spiering, M., & Haiges, S. (2010). *HTML 5-Apps für iPhone und Android entwickeln*. Poing: Franzis-Verl.

Stahlknecht, P., & Hasenkamp, U. (2002). *Einführung in die Wirtschaftsinformatik* (10., überarb. u. aktualis. Aufl.). Springer Berlin.

Stamey, J., Lassez, J.-L., Boorn, D. & Rossi, R. (2007). Client-side dynamic metadata in web 2.0. *25th annual ACM international conference on Design of communication* (p. 155). Presented at the SIGDOC '07, ACM Press. doi:10.1145/1297144.1297176

Stark, J. (2011). *Android-Apps mit HTML, CSS und JavaScript [mit Standard-Web-Tools zur nativen App]*. Beijing; Cambridge; Farnham; Köln; Sebastopol; Tokyo: O'Reilly.

StatCounter. (2012a). Top 9 Mobile Browsers in Austria from Mar 2011 to Mar 2012 | StatCounter Global Stats. Retrieved April 29, 2012, from http://gs.statcounter.com/#mobile_browser-AT-monthly-201103-201203

StatCounter. (2012b). Top 9 Mobile Browsers in Austria on Apr 2012 | StatCounter Global Stats. Retrieved May 1, 2012, from http://gs.statcounter.com/#mobile_browser-AT-monthly-201204-201204-bar

Stuedi, P., Mohomed, I. & Terry, D. (2010). WhereStore. *1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond* (pp. 1–8). Presented at the MCS'10, San Francisco, USA: ACM Press. doi:10.1145/1810931.1810932

Tatsubori, M. & Suzumura, T. (2009). HTML templates that fly. *18th international conference on World wide web* (p. 951). Presented at the WWW '09, Madrid, Spain: ACM Press. doi:10.1145/1526709.1526837

Thies, G. & Reimers, S. (2009). *PHP 5.3 und MySQL 5.1 : Grundlagen, Anwendung, Praxiswissen, Objektorientierung, MVC, Sichere Webanwendungen, PHP-Frameworks, Performancesteigerungen*. Bonn: Galileo Press.

Thung, P. L., Ng, C. J., Thung, S. J. & Sulaiman, S. (2010). Improving a web application using design patterns: A case study. *2010 International Symposium in Information Technology* (pp. 1–6). Presented at the ITSim, Kuala Lumpur: IEEE. doi:10.1109/ITSIM.2010.5561301

van Kesteren, A. (2012). Cross-Origin Resource Sharing. Retrieved May 1, 2012, from http://www.w3.org/TR/cors/

van Zyl, P., Kourie, D. G., Coetzee, L. & Boake, A. (2009). The influence of optimisations on the performance of an object relational mapping tool. *2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists* (pp. 150–159). Presented at the SAICSIT '09, Vynderbijl Park, South Africa: ACM Press. doi:10.1145/1632149.1632169

Vaughan-Nichols, S. J. (2010). Will HTML 5 Restandardize the Web? *Computer*, *43*(4), 13–15. doi:10.1109/MC.2010.119

W3C. (2011). W3C Confirms May 2011 for HTML 5 Last Call, Targets 2014 for HTML 5 Standard. Retrieved May 20, 2012, from http://www.w3.org/2011/02/htmlwg-pr.html.en

WebToGo Mobiles Internet GmbH. (2003). WebToGo für PalmOS Benutzerhandbuch. Retrieved from http://www.webtogo.de/Download/pdf/WTG_user-guide1.pdf

Wenz, C. (2010). *JavaScript das umfassende Handbuch ; [inkl. Ajax ; Einstieg, Praxis, Referenz ; Web.2.0: DOM, CSS, XML, Web Services ; für Einsteiger, Fortgeschrittene und Profis]*. Bonn: Galileo Press.

West, W. & Pulimood, S. M. (2012). Analysis of privacy and security in HTML 5 web storage. *Journal of Computing Sciences in Colleges*, *27*(3), 80–87.

Wider, A. (2007). *Komponentenorientierte Anwendungsentwicklung auf der .NET-Plattform* (Diplomarbeit). Technische Fachhochschule Berlin, Berlin. Retrieved from http://metrik.informatik.hu-berlin.de/grk-wiki/images/0/07/Diplomarbeit_Wider.pdf

Wikimedia Commons. (2009). File:PHP funktionsweise.png - Wikimedia Commons. *Wikimedia Commons*. Retrieved April 29, 2012, from http://commons.wikimedia.org/wiki/File:PHP_funktionsweise.png

Wikipedia. (2012). JavaScript – Wikipedia. Retrieved May 1, 2012, from http://de.wikipedia.org/wiki/Javascript

WUNDERMANN PXP GmbH. (2011). Media Use Index Österreich 2011. Retrieved from http://www.media-use-index.at/

xui contributers. (2010a). xui.js - a simple javascript library for building mobile web applications. Retrieved May 11, 2012, from http://xuijs.com/

xui contributers. (2010b). xui.js - documentation / basics. Retrieved May 11, 2012, from http://xuijs.com/docs/basics

xui contributers. (2010c). xui.js - documentation / dom. Retrieved May 11, 2012, from http://xuijs.com/docs/dom

xui contributers. (2010d). xui.js - documentation / event. Retrieved May 11, 2012, from http://xuijs.com/docs/event

xui contributers. (2010e). xui.js - documentation / fx. Retrieved May 11, 2012, from http://xuijs.com/docs/fx

xui contributers. (2010f). xui.js - documentation / style. Retrieved May 11, 2012, from http://xuijs.com/docs/style

xui contributers. (2010g). xui.js - documentation / xhr. Retrieved May 11, 2012, from http://xuijs.com/docs/xhr

Yang, Q., Zhang, H. H. & Li, T. (2001). Mining web logs for prediction models in WWW caching and prefetching. *seventh ACM SIGKDD International conference on Knowledge discovery and data mining* (pp. 473–478). Presented at the KDD'01, San Francisco CA, USA: ACM Press. doi:10.1145/502512.502584

Zhang, W., Lewanda, D. B., Janneck, C. D. & Davidson, B. D. (2003). Personalized Web Prefetching in Mozilla. Department of Computer Science and Engineering Lehigh University Bethlehem. Retrieved from http://www.peerevaluation.org/read/libraryID:649

Zhimei Jiang & Kleinrock, L. (1998). Web prefetching in a mobile environment. *IEEE Personal Communications*, *5*(5), 25–34. doi:10.1109/98.729720

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Anhang

A. Lebenslauf

Kerstin Blumenstein, BSc

Adresse:	Propst Führer Straße 3 3100 St. Pölten, Österreich
E-mail:	blumenstein.kerstin@gmail.com
Geburtsdatum:	03.03.1980
Geburtsort:	Wolfen, Deutschland
Staatsangehörigkeit:	Deutsch
Familienstand:	verheiratet

Schul- / Berufsausbildung

Studium:	Master Digitale Medientechnologien (Mobiles Internet), FH St. Pölten (A), 2010 - 2012
	Bachelor Medientechnik (Interaktive Medien), FH St. Pölten (A), 2007 - 2010
Diplomarbeit:	Datensynchronisation und Offline-Nutzung für webbasierte mobile Informationssysteme
Bachelorarbeit:	Technische und anwendungsbezogene Probleme und Besonderheiten des mobilen Internet
Ausbildung:	Mediengestalterin Bild und Ton, DAP Bitterfeld (D), 1998 - 2001
Gymnasium:	Gymnasium Wolfen-Nord (D), Abitur 1998

Berufserfahrung

Wissenschaftliche Mitarbeiterin, Schwerpunkt Mobile,
Institut für Creative \ Media / Technologies, FH St. Pölten (A), seit 2011

Lehrtätigkeit im Bachelor Medientechnik, FH St. Pölten (A), seit 2009

Studentische Assistentin, Institut für Medieninformatik, FH St. Pölten (A),
2009 - 2011

Studentische Assistentin, Ausbildungsfernsehen c-tv, FH St. Pölten (A), 2008 - 2009

Mediengestalterin Bild und Ton, RBW Fernsehgesellschaft mbH (D), 2002 - 2007

Weiterbildungen

Onlinestudium: Module „Grafik-Design“, „Bildbearbeitung“ (Photoshop) und „Verlagsgrafik“ (Quark Express), HTK Hamburg, 2005 - 2007

Sprachkenntnisse

Deutsch:	Muttersprache
Englisch:	gute Kenntnisse in Sprache und Schrift
Russisch:	Gymnasialkenntnisse (10 Jahre)

Publikationen

Blumenstein, K. & Schmiedl, G. (2011). Usability-Testing mobiler Szenarien als Sekundärtask – Vergleich technischer Ansätze. Proceedings of 4. Forum Medientechnik. Presented at the 4. Forum Medientechnik. St. Pölten, Austria.

Eitler, T., Blumenstein, K., & Schmiedl, G. (2011). Mobilot: Architektur und technische Optionen bei der Entwicklung eines mobilen Informationssystems. Proceedings of 4. Forum Medientechnik. Presented at the 4. Forum Medientechnik. St. Pölten, Austria.

Schmiedl, G. & Blumenstein, K. (2011). Trust Center Controlled Code Access Security. Proceedings of 9. Sicherheitskonferenz Krems. Presented at the 9. Sicherheitskonferenz Krems. Krems, Austria.

Schmiedl, G., Blumenstein, K. & Seidl, M. (2011). Usability Testing for Mobile Scenarios of Fragmented Attention. Proceedings of Chi Sparks 2011. Presented at the Chi Sparks 2011. Arnhem, The Netherlands.

Blumenstein, K. & Schmiedl, G. (2010). Die vier Kernprobleme der mobilen Webentwicklung. Proceedings of 3. Forum Medientechnik. Presented at the 3. Forum Medientechik. St. Pölten, Austria.

Auszeichnungen

Leistungsstipendium der Fachhochschule St. Pölten, 2009, 2010 & 2011

1. Platz Golden Wire 09 (Kategorie Interactive) für WUP, 2009

3. Platz Hoer.Spiel (Kategorie Studierende) für Hörspiel „Straßenbahnfahrt“, 2008