Offline fallback für webbasierte Kollaborations-Systeme

Masterarbeit

von

Tim van Cleef

aus

Hagen

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

September 2012

Betreuer:
Philipp Hagemeister M. Sc.

Danksagungen

In erster Linie möchte ich Philipp Hagemeister danken, der immer Zeit und ein offenes Ohr hatte und mich durch die richtigen Fragen immer vor neue Herausforderungen gestellt hat.

Desweiteren danke ich Roland Wimmer, der mir gerade während der Entwicklungszeit dadurch geholfen hat, sich meine Probleme anzuhören – alleine es jemandem zu erklären macht die Sache viel klarer.

Großer Dank gilt auch Ariane Olek dafür, dass sie die Arbeit Korrekturgelesen hat.

Zu guter Letzt möchte ich Prof. Martin Mauve dafür danken, dass ich so ein praxisnahes und aktuelles Thema behandeln durfte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1.

Einführung

1.1. Webapplikationen

Webseiten sind heutzutage nicht mehr nur statische, informative Dokumente, sondern vielmehr interaktive Applikationen, sogenannte *Webapplikationen* oder auch *Webanwendungen*. Beispiele hierfür sind Google Mail, Google Docs sowie die Kreativ-Applikationen Mockingbird und Spritepad.

Der Begriff Webapplikation bezeichnet dabei eine Software oder einen Dienst, der üblicherweise auf einem entfernten Computer (Server) im Netzwerk oder dem Internet läuft. Der Benutzer greift in der Regel über einen Webbrowser (Client) auf den Dienst zu, ohne dass zuvor die Installation einer bestimmten Software nötig wäre. Dieser Vorteil trägt dazu bei, dass webbasierte Anwendungen immer populärer werden und die klassische Desktopsoftware kontinuierlich in den Hintergrund rückt. Ein Beispiel ist die Verwaltung von Emails[web10a].

Ein weiterer Vorteil von Webanwendungen ist: selbst Computer mit wenig Rechenleistung können komplexe Aufgaben ausführen, da von diesem Computer nur Kommandos an den Server geschickt werden. Dieser führt dann die rechenintensive Operation aus und liefert das Ergebnis zurück. Im Gegensatz dazu müsste bei lokal installierter Software der Computer selbst genügend Rechenleistung haben, um die rechenintensiven Operationen durchzuführen.

Der große Nachteil, den webbasierte Anwendungen gegenüber lokal installierter Software haben, ist, dass der gesamte Code – also die gesamte Software – auf dem entfernten Rechner liegt, weshalb zwingend eine Verbindung zum Server nötig ist. Bricht diese ab oder fällt der Server aus, ist die Anwendung nutzlos.

1.2. Ziel der Arbeit

Die vorliegende Arbeit widmet sich einem möglichen Lösungsansatz für den Offline-Betrieb von Webapplikationen, indem ein *Offline Fallback* für eine bestehende Webapplikation realisiert werden soll: durch den Umbau der Applikation soll sie auch ohne Verbindung zum Server verwendbar sein. Anstatt die Webapplikation mit einer plattformabhängigen nativen Desktopsoftware zu ersetzen, soll das Offline Fallback weiterhin im Browser laufen.

Konkret wird in dieser Arbeit für die Webanwendung Adhocracy[adh] – eine Plattform zur kooperativen Normsetzung –, ein Offline Fallback implementiert, das es dem Benutzer ermöglicht die wichtigen Teile der Applikation auch offline zu nutzen.

1.3. Aufbau der Arbeit

Zunächst wird erläutert, wie Webapplikationen heutzutage in der Regel funktionieren und wie der aktuelle Stand in Bezug auf den offline Zugriff ist. Desweiteren wird Adhocracy vorgestellt. Kapitel drei bildet den Hauptteil der Arbeit: hier wird konkret erläutert, welche Schritte notwendig sind, um ein Offline Fallback für eine bestehende Webapplikation bereitzustellen. In Kapitel vier werden die Ergebnisse verschiedener Performancetests vorgestellt. Im letzten Kapitel wird ein Fazit zu den Änderungen gezogen und ein Ausblick auf zukünftige Arbeit in diesem Bereich gegeben.

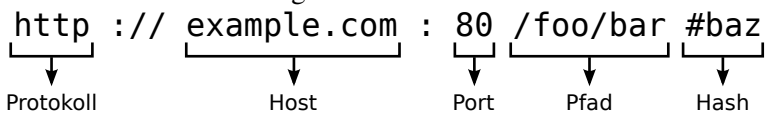
Kapitel 2.

Grundlagen

2.1. Funktionsweise von Webapplikationen

Wie in der Einleitung ($\rightarrow$ 1.1) erläutert, sind die meisten Webapplikation nach dem Client/Server Prinzip aufgebaut: der Client, in der Regel der *Browser*, schickt Anfragen an den entfernten Rechner, den *Webserver*. Dieser antwortet üblicherweise mit einer fertigen HTML[htm12a]-Seite, die der Browser verarbeitet und darstellt. Angesprochen werden Webapplikationen über ihre URL[1][url01], welche in mehrere Bereiche eingeteilt ist, wie in Abbildung 2.1 zu sehen.

Abbildung 2.1.: Aufbau einer URL



Der erste Teil der URL gibt das *Protokoll* an, über das die Kommunikation stattfindet. In den meisten Fällen handelt es sich dabei um HTTP[FGM+99a]. Für eine verschlüsselte Verbindung wird üblicherweise HTTPS[Res00] verwendet.

Der nächste Bereich, der *Host*, referenziert den entfernten Server auf dem die eigentliche Webapplikation läuft. Optional steht hinter dem Host, durch einen Doppelpunkt getrennt, noch ein spezieller *Port* – standardmäßig werden Webapplikationen aber über Port 80 angesprochen.

Dem Hostteil schließt sich der *Pfad* des aufgerufenen Dokuments an. Dieser ist im einfachsten Fall eine direkte Abbildung der Ordnerstruktur auf dem Webserver.

[1] Uniform Resource Locator - eigentlich Uniform Resource Identifier (URI). Die Begriffe werden aber im Bereich Web häufig synonym verwendet.

Optional kann hinter dem Pfadteil noch ein On-Page-Anker, der *Hash*, folgen, welcher durch eine Raute (#) eingeleitet wird. Dieser Anker veranlasst den Browser im aktuellen HTML Dokument dazu, ein Element mit dem Namen oder der ID des Ankers zu suchen und gegebenenfalls an diese Stelle zu springen.

2.1.1. Server

Serverapplikationen werden heutzutage üblicherweise nicht von Grund auf neu geschrieben, sondern basieren auf bestehenden Frameworks. Desweiteren halten sie sich an eine bestimmte Softwarearchitektur, die die Applikation in logische Bereiche aufteilt und die Aufgaben klar trennt. Die derzeit gängigste Variante im Bereich Webentwicklung ist die *Model-View-Controller* [mvc] Architektur, im Folgenden als MVC bezeichnet.

Model-View-Controller Architektur

MVC bezeichnet eine Architektur, die die Applikation in drei Bereiche einteilt. Es wird zwischen den Daten (Model), der Darstellung (Views/ Templates) und der Applikationslogik (Controller) unterschieden. Dabei besteht das Grundkonzept darin, diese Teilbereiche voneinander zu trennen, sodass wiederverwendbare Module entstehen.

Ein **Model** ist zumeist eine direkte Abbildung einer Tabelle aus der Datenbank und wird in der Applikation als Klasse realisiert. Model-Klassen bilden demnach die Schnittstelle zur Datenbank und bieten abstrakte Methoden zum Erstellen, Aktualisieren und Löschen von Daten an.

Die **Views** oder **Templates** einer MVC Applikation sind für die Darstellung der Daten zuständig: sie erstellen das HTML, bevor dies zum Client geschickt wird. Views bzw. Templates sind oft so konzipiert, dass sie eine Instanz eines bestimmten Models entgegennehmen und sich daraus die für die Darstellung benötigten Informationen zusammensuchen.

Controller bilden die logische Schicht von MVC Anwendungen. Als Schnittstelle zum Benutzer stellen sie Methoden und Funktionalitäten nach außen bereit. Ihre Hauptaufgabe besteht dabei darin, Anfragen vom Benutzer entgegen zunehmen, die korrekten Daten zu laden und sie schließlich an das richtige Template weiterzugeben.

Die Bedeutung der URL ändert sich bei MVC basierten Webanwendungen grundlegend: bisher wurde angenommen, dass der Pfadteil der URL direkt auf die Ordnerstruktur abgebildet wird. Bei

MVC Anwendungen hingegen teilt er der Applikation mit, welche Aktion ausgeführt bzw. welche Methode in welchem Controller aufgerufen werden soll. Beispielsweise würde die URL `http://adhocracy.de/instance/diskussion/show` die Applikation dazu veranlassen, die Methode `show` im Controller `instance` aufzurufen.

2.1.2. Client

Als Client einer klassichen Webapplikation gilt meistens der Browser des Benutzers, der das HTML vom Webserver empfängt, parst und anschließend darstellt. Durch das Parsen wird eine DOM-Struktur[2][dom12] aus dem HTML erstellt. Anschließend lädt er alle verlinkten Ressourcen vom Server, zum Beispiel Bilder, CSS[css10] und JavaScript. Mit Hilfe von CSS können unter anderem Schriftart und -größe sowie Hintergrundfarben angepasst werden, um das Aussehen der Webseite wie gewünscht zu modifizieren.

JavaScript ist eine Programmiersprache, die speziell für die Ausführung im Browser konzipiert wurde und heute Bestandteil fast jeder Webapplikation ist. Ihr Einsatz kann stark variieren: in den meisten Fällen wird JavaScript nur benutzt, um die Ausgaben des Browsers zu manipulieren oder kleine Animationen zu erstellen. Man kann mit JavaScript aber auch komplexe Browseranwendungen und -spiele implementieren, beispielsweise Sketchpad[skp] oder Cut the Rope[ctr].

Ein weiterer großer Einsatzbereich für JavaScript ist das Nachladen von Inhalten via XMLHttpRequests[xhr], im Folgenden *XHR* genannt. Diese speziellen JavaScript Objekte bieten eine API, die es erlaubt Daten[3] vom Server der gleichen Domain zu laden. Auf diesem Weg können Inhalte, beispielsweise Bilder oder HTML-Seiten, asynchron im Hintergrund, sprich ohne das Neuladen der gesamten Seite, abgerufen werden.

2.2. Offline Fallback: Möglichkeiten bisher

Bisher gab es – abseits von Browsercache und Cookies – keine Möglichkeit, um ein Offline Fallback für eine Webapplikation zu realisieren, die den Anforderungen und dem Ziel dieser Arbeit (→ 1.2) gerecht wird.

[2] Document Object Model: eine standardisierte Schnittstelle um Elemente eines XML-basierten Dokuments zu anzupassen. Im Bereich Webentwicklung wird sie in JavaScript verwendet um das HTML zu manipulieren.

[3] Anders als der Name vermuten lässt, können XMLHttpRequests jeden beliebigen Datentyp abrufen, nicht nur XML.

Browsercache

Der Browsercache ist ein flüchtiger Speicher, in dem Ressourcen automatisch abgelegt werden, damit sie nur einmal vom Server geladen werden müssen. Über einen HTTP Header kann der Server Einfluss darauf nehmen, wie lange die Ressourcen gecacht werden sollen. Letztendlich steht es aber dem Browser frei, ob er Ressourcen cacht oder nicht. Desweiteren ist der Zugriff auf die gecachten Ressourcen eingeschränkt, wenn die Verbindung zum Server abbricht oder der Benutzer in den Offline Modus wechselt – das Anzeigen funktioniert zwar noch, aber sobald die Seite neugeladen oder ein Link angeklickt wird, zeigt der Browser eine Fehlermeldung an. Demnach ist der Browsercache nicht für die Offline-Benutzung konzipiert und eignet sich nicht für ein Offline Fallback.

Cookies

Ein Cookie[Bar11] ist ein HTTP Header, der Platz für benutzerdefinierte Informationen bereitstellt. Er wird in der Regel vom Server erstellt und an den Client geschickt und ab diesem Zeitpunkt bei jedem Request mit übertragen. Ein Cookie kann aber auch initial im Browser in JavaScript erstellt und dann an den Server geschickt werden.

Cookies werden üblicherweise dazu eingesetzt, eine Art Beziehung zwischen Server und Browser herzustellen. Dazu setzt der Server eine SessionID, die er im Cookie ablegt, und über die er den Browser bei späteren Requests wieder identifizieren kann. Desweiteren werden Cookies verwendet, um persönliche Einstellungen auf Webseiten, wie zum Beispiel eine bevorzugte Schriftgröße, zu setzen. Für komplexere und größere Datenmengen sind Cookies allerdings nicht geeignet, da sich entgegen der Vorgaben des Standards[4] eine maximale Größe von 4 KB je Cookie in den meisten Browsern etabliert hat.

2.3. Offline Fallback: Möglichkeiten mit HTML5

HTML5[htm12b] bezeichnet eigentlich nur die Sprache HTML, wird im allgemeinen Sprachgebrauch aber auch als Sammelbegriff für alle neuen Technologien und Features verwendet, die im Bereich Web entwickelt wurden und noch werden. Von besonderer Bedeutung für diese Arbeit ist der Bereich, der sich mit dem Thema Offline Webapplikationen beschäftigt, da er Features und APIs beinhaltet, die unter anderem das dauerhafte, lokale Speichern von Ressourcen ermöglichen.

[4] Laut Standard[Bar11] ist eigentlich eine *Mindestgröße* von 4 KB je Cookie vorgesehen.

Mit Hilfe dieser und weiterer verwandter Technologien, wie zum Beispiel Web Storage, ist es nun möglich, ein Offline Fallback für eine Webapplikation zu realisieren, das den Anforderungen (→ 1.2) gerecht wird.

Einige dieser Features sind allerdings noch in aktiver Entwicklung und werden nicht von allen Browsern unterstützt. Die vorliegende Arbeit wurde daher so designed, dass auch Browser, die keine oder nur einige dieser Features kennen, die Applikation weiterhin – dann aber nur im Online-Modus – nutzen können.

2.4. Adhocracy

Adhocracy ist eine webbasierte Anwendung, die kooperative Normsetzung für Institutionen und Unternehmen ermöglicht. Es dient dazu, „Programme, Gesetzesentwürfe, Handlungsempfehlung[en], Texte o.ä. kollaborativ“[adh] zu erarbeiten. Adhocracy ist dabei in Instanzen unterteilt, über die in Form von Vorschlägen diskutiert werden kann. Diese können „in einem moderationsfreien und transparenten Prozess“[adh] bewertet und kommentiert werden.

Software

Adhocracy ist als eine reine Online Webapplikation konzipiert und umgesetzt und funktioniert nach dem klassischen Client/ Server-Prinzip. Es ist in Python geschrieben und basiert auf dem MVC Webframework Pylons[pyl]. Die Views bei Adhocracy werden in Mako[mak] geschrieben – einer Templatesprache, die speziell auf die Verwendung mit Python ausgelegt ist.

2.5. Related Work

Obwohl es die in der Arbeit verwendeten HTML5 Technologien schon einige Zeit gibt, beschränken sich die meisten Anwendungsfälle immernoch auf Demonstrationszwecke. Viele Webapplikationen setzen einzelne HTML5 Features schon ein, aber die Kombination mehrerer, um die Realisierung eines Offline Fallback für eine Webapplikation zu erreichen, in der der Benutzer auch Inhalte erstellen kann, wird nur selten angewandt[5].

[5] Der Goßteil der Offline Lösungen stützt sich derzeit noch auf Plug Ins, wie zum Beispiel Flash, Java oder direkt auf native Desktopanwendungen.

Google

Ein Vorreiter für Offline Webapplikationen ist Google – es bietet mit Gmail, Calendar und Docs für gleich drei Webapplikationen Offline-Versionen[gof] an, die sowohl den Lese- als auch den Schreibzugriff offline ermöglichen. Die Änderungen werden bei bestehender Verbindung wieder mit dem Google Server synchronisiert.

Auch Google verwendet für seine Offline Anwendungen HTML5, verfolgt dabei aber einen anderen Ansatz als diese Arbeit: der Google-Benutzer muss die Offline Nutzung explizit einrichten und eine seperate Plug-In ähnliche Anwendung installieren – es ist demnach kein Offline Fallback, von dem der Benutzer im Idealfall nichts mitbekommt.

SPWR

Ein anderes Beispiel ist *SPWR*[spw12], ein speziell auf Betriebsabläufe in Unternehmen ausgerichtetes Framework für den Umbau betriebsinterner Webapplikationen für den Offline Gebrauch.

Die Betriebsabläufe und Anwendungen in Unternehmen basieren heutzutage oft auf dem Browser-Server-Prinzip[6] und setzen daher eine kontinuierliche Verbindung zum Netz voraus – diese ist aber nicht immer gewährleistet. Daher bietet das SPWR Framework die Möglichkeit, die bestehende Webanwendung in der Art anzupassen, dass der Benutzer sie offline verwenden kann.

Das Problem, das SPWR löst, ist dem in dieser Arbeit zu Behandelnden zwar sehr ähnlich, aber der Lösungsansatz ist ein anderer. Das SPWR Framework analysiert das Benutzerverhalten und erkennt welche Teile der Applikation der Nutzer verwendet. Basierend auf diesen Informationen erstellt SPWR eine clientseitige Subapplikation, die dann offline verwendet werden kann.

Im Gegensatz zu SPWR soll in dieser Arbeit aber schreibender Zugriff auf die Daten auch lokal möglich sein, daher benötigt man Teile der Applikationslogik des Servers auch lokal im Browser.

[6] Das Browser-Server-Prinzip entspricht dem Client-Server-Prinzip mit dem Unterschied, dass davon ausgegangen wird, der Client sei immer der Browser.

Kapitel 3.

Änderungen

Um ein Offline Fallback für eine Webapplikation zu entwickeln, ist es notwendig, dass zum einen alle statischen Ressourcen, wie zum Beispiel Bilder, CSS und JavaScript, und zum anderen die Daten – die HTML-Seiten – lokal beim Benutzer vorliegen. Lokal vorliegen bedeutet dabei, dass sie so gespeichert sind, dass der Nutzer auch offline darauf zugreifen kann.

Die triviale Lösung ist demnach, alle HTML-Seiten und statischen Ressourcen dauerhaft lokal zu cachen, um sie für den Benutzer offline verfügbar zu machen. Im Verlauf dieser Arbeit wird ein HTML5 Cache-Mechanismus (→ 3.1.1) vorgestellt, der genau das ermöglicht.

Eine Eigenschaft dieses Caches lässt diesen Lösungsvorschlag aber schnell scheitern: sobald nämlich eine HTML-Seite lokal gecacht ist, wird sie nur noch aus diesem geladen und *nicht* mehr online abgerufen – selbst wenn online eine neuere Version vorliegt. Konkret bedeutet das, dass dem Benutzer nur noch veraltete Seiten gezeigt werden und die Anwendung somit unbrauchbar wird.

Um dieses Problem zu lösen, wird die Verwendung des Caches dahin gehend angepasst, dass zwar alle statischen Ressourcen, *nicht* aber die HTML-Seiten selbst gecacht werden, damit der Benutzer wieder aktuelle Applikations-Inhalte zu sehen bekommt.

Allerdings führt diese Erweiterung zu einem weiteren Problem: da keine HTML-Seiten mehr lokal gecacht werden, stehen sie dem Benutzer auch nicht offline zur Verfügung – effektiv haben wir so das Offline Fallback zerstört.

Man muss sich also überlegen, wie der Browser sowohl die HTML-Seiten lokal speichern als auch ständig für eine aktuelle Version sorgen kann. Im Rahmen dieser Arbeit wird ein Offline-Speicher (→ 3.3.3) vorgestellt, mit dem die HTML-Seiten manuell, am Cache vorbei, lokal gespeichert werden können. Hierbei müsste allerdings jede einzelne HTML-Seite der Webapplikation abgespeichert werden, was zu einer enormen Verschwendung des zur Verfügung stehenden Offline Speichers füh-

ren würde. Außerdem soll die Datenbearbeitung auch offline möglich sein, was sich als schwierig herausstellt, wenn man nur die fertigen HTML-Seiten zur Verfügung hat.

Bei der lokalen Datenspeicherung empfiehlt sich daher, besser nicht die HTML-Seiten, sondern die reinen Daten zu speichern. Das Problem dabei ist, dass heutige Webapplikationen in der Regel nur fertige HTML-Seiten ausliefern können (→ 2.1). Daher muss also das Grundkonzept heutiger Webanwendungen überdacht werden: *nicht der Server erstellt das HTML, sondern der Client (Browser)*. Der Server dient dann lediglich noch als Verteiler der reinen Daten und statischen Ressourcen. Zur Umsetzung dieses Konzepts sind Änderungen und Erweiterungen am Server notwendig. Sie umfassen dabei alle drei Bereiche der MVC Architektur:

- jedes relevante **Model** muss sich in einem vereinbarten Datenformat darstellen können (→ 3.2.1)
- je nach verwendeter Templatesprache müssen die **Views/ Templates** für die Verwendung im Browser vorbereitet werden (→ 3.2.2)
- die **Controller** müssen entscheiden können, ob sie fertige HTML-Seiten oder die reinen Daten zurückliefern sollen (→ 3.2.3)

Neben den Änderungen am Server ist parallel noch die Entwicklung eines clientseitigen *Offline Fallback Frameworks* in JavaScript unerlässlich. Dieses Framework umfasst die folgenden Aufgaben:

- die **Daten** vom Server laden (→ 3.3.1)
- das **HTML** lokal erstellen und es im Browser anzeigen (→ 3.3.2)
- die Daten **lokal speichern**, sodass sie auch offline verfügbar sind (→ 3.3.3) und dem Benutzer auch Änderungen ermöglichen (→ 3.4.3)
- die Daten **synchronisieren** und erkennen, ob der Benutzer online oder offline ist (→ 3.4)

3.1. Ressourcen und Code lokal speichern

Wie erwähnt besteht der erste Schritt darin, Ressourcen lokal zu speichern. Dazu wird ein langlebiger Cache benötigt, der die Ressourcen auch dann ausliefert, wenn keine Verbindung zum Server besteht.

3.1.1. Application Cache

Im Rahmen der HTML5 Offline Webapplications[owa12] wurde mit dem **Application Cache**[app12b] (Appcache) ein derartiger Cache standardisiert. Zuvor gab es zwar schon diverse Implementierungen, diese waren aber Browser-spezifisch und damit nicht sinnvoll einzusetzen.

Der Application Cache ermöglicht es, angegebene Ressourcen dauerhaft lokal abzuspeichern. Einmal gespeichert, bleiben sie dann solange im Cache, bis der Benutzer ihn explizit leert oder er durch die Webapplikation aktualisiert wird.

Da es sich beim Application Cache um ein HTML5 Feature handelt, stellt sich die Frage nach der Browserunterstützung, die entscheidet, ob die Verwendung des Features sinnvoll ist. Mit Ausnahme des Internet Explorer wird der Application Cache von den neueren Versionen vieler Desktop Browser implementiert[app12c]. Von den mobilen Browsern unterstützen es die beiden großen: Mobile Safari und Android (Chrome)[app12c].

Ein weiteres K.O.-Kriterium kann der Speicherplatz sein, den man für lokale Ressourcen zur Verfügung hat. Leider ist gerade der nicht genau spezifiziert und bleibt somit den Browserherstellern überlassen. In den Spezifikationen heißt es lediglich, dass „Browser den Speicherplatz begrenzen sollten“[app12a]. Dennoch stellen die aktuellen Browser mit in der Regel mindestens 5 MB für lokale Daten und Ressourcen ausreichend Speicherplatz für das Offline Fallback bei Adhocracy bereit (→ 4.3.3).

Cache Manifest

Damit der Browser erkennt, welche Ressourcen von der Webapplikation für den Offline Zugriff benötigt werden und deshalb lokal zu speichern sind, gibt es das **Cache Manifest** – eine Liste mit allen erforderlichen Ressourcen. Um es in eine HTML-Seite einzubinden, muss es direkt im öffnenden `html` Tag referenziert werden. Ein HTML5 fähiger Browser lädt das Cache Manifest anschließend vom Server und analysiert es. Die Ressourcen im Cache Manifest können dabei einem von drei Bereichen zugeordnet sein:

- `CACHE` Section
  Darin befinden sich die sogenannten *Explicit-Entries*: Ressourcen, die auf jeden Fall sofort geladen und gecacht werden sollen.

- `NETWORK` Section
  Dieser Bereich wird auch als *Online Whitelist* bezeichnet: hier referenzierte Ressourcen werden nicht gecacht, sondern immer vom Webserver abgerufen.

- `FALLBACK` Section
  Hier werden Ressourcen angegeben, die anderen Ressourcen als Fallback dienen, wenn diese nicht geladen werden konnten.

Neben den Explicit-Entries gibt es eine weitere Ressource, die auf jeden Fall gecacht wird: der sogenannte *Master-Entry*. Das ist die HTML-Seite, die das Cache Manifest referenziert. Er wird – unabhängig davon, ob er selbst im Cache Manifest aufgelistet ist oder nicht – auf jeden Fall lokal gespeichert.

Alle Ressourcen – also auch Master- und Explicit-Entry – werden, sobald sie im Application Cache gespeichert sind, immer daraus geliefert, auch wenn sich auf dem Server eine aktuellere Version befindet. Die einzige Möglichkeit, um solche Ressourcen lokal zu aktualisieren, besteht darin, den gesamten Application Cache neu zu laden. Dies passiert aber nur, wenn sich am Cache Manifest selbst etwas geändert hat. Bei jedem Request wird es vom Server geladen und mit der lokalen Kopie verglichen: wenn sie sich unterscheiden, wird der gesammte Application Cache geleert und werden alle Ressourcen neu angefordert.

Abbildung 3.1 zeigt die Verwendung des Application Cache beispielhaft: der Server liefert auf Anfrage ein HTML Dokument aus, in dessen `html` Tag ein Cache Manifest referenziert ist. Ein Browser, der den Application Cache implementiert, erkennt das `manifest` Attribut und lädt das Cache Manifest vom Server. Anschließend analysiert er es und lädt dann nach und nach alle Ressourcen herunter und speichert sie im Cache. Zuletzt legt er die HTML-Seite selbst als Master-Entry auch im Cache ab. Abbildung 3.2 verdeutlicht, dass alle zukünftigen Anfragen an eine der Ressourcen aus dem Application Cache beantwortet werden. Es wird lediglich ein Request an den Server geschickt, in dem das Cache Manifest erneut abgerufen wird.

3.1.2. Cache Manifest einbinden

Der erste, einfache Ansatz zur Verwendung des Application Cache für das Offline Fallback ist, ein Cache Manifest zu erstellen und es auf allen Seiten einzubinden, sodass es von jeder Seite ausgeliefert wird. Der Browser cacht dann die expliziten Einträge und jede Seite, die der Benutzer aufruft. So baut sich nach und nach eine offline benutzbare Version der Applikation auf.

Abbildung 3.1.: Application Cache: Ressourcen speichern

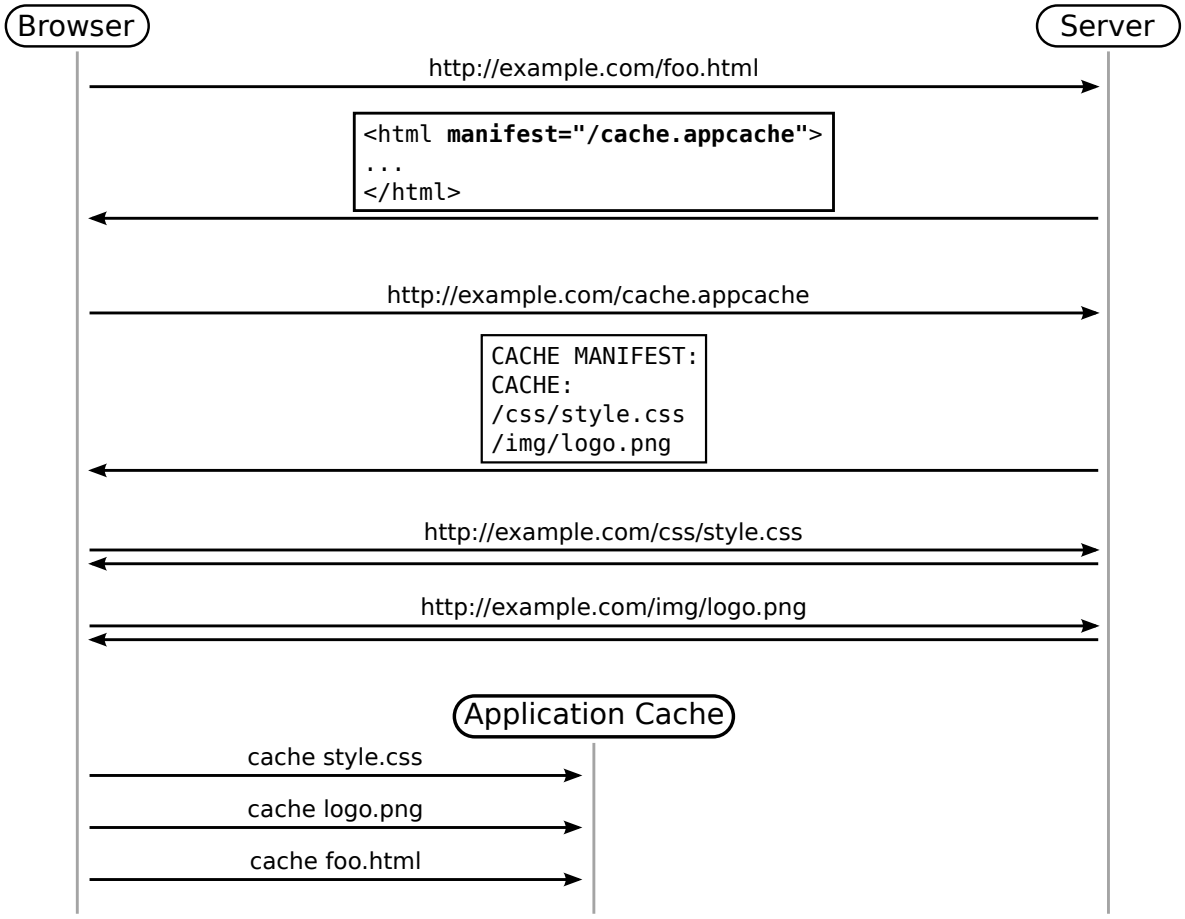
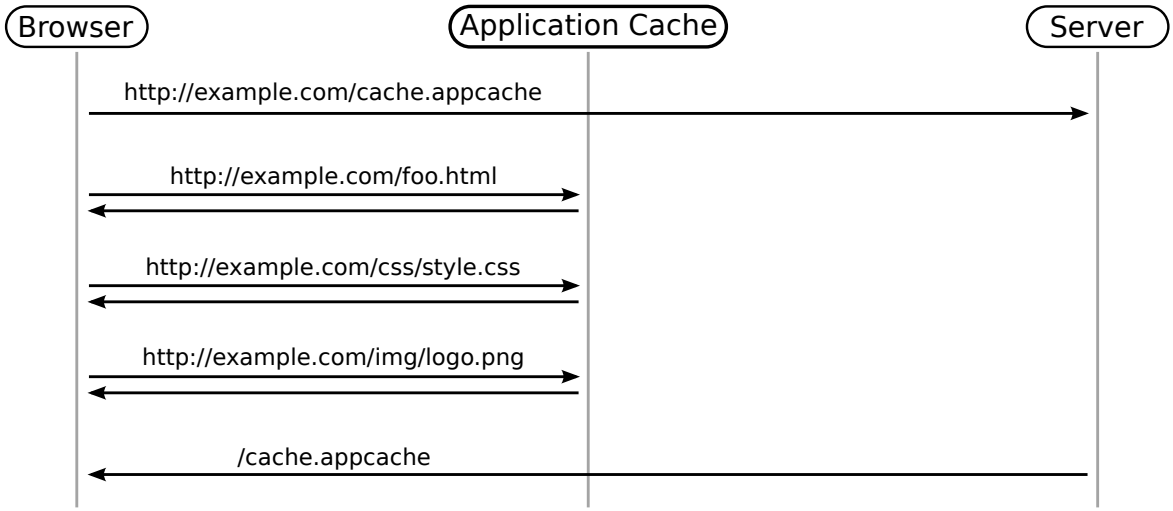


Abbildung 3.2.: Application Cache: Ressourcen ausliefern



Listing 3.1 zeigt wie ein solches Manifest beispielsweise aussehen kann. Die erste Zeile teilt dem Browser mit, dass es sich hier um ein Cache Manifest handelt. Die nächste Zeile leitet die `CACHE` Section ein. In diesem Beispiel sollen das Stylesheet `style.css`, das JavaScript `app.js` und das Icon `icon.png` auf jeden Fall gecacht werden. Diese Ressourcen werden direkt heruntergeladen und lokal gespeichert. Die `NETWORK` Section im Beipsiel stellt einen Sonderfall dar: das `*` ist ein Platzhalter und steht generell für alle Ressourcen. Im Zusammenhang mit der `NETWORK` Section bedeutet er, dass generell alle Ressourcen online abgerufen werden sollen – mit Ausnahme derjenigen, die schon im Cache sind. Würde man diese Zeile weglassen, fehlten beispielsweise alle Bilder, die nicht in der `CACHE` Section angegeben wurden oder von anderen Domains kommen.

Listing 3.1: Beispiel für ein Cache Manifest

```
1 CACHE MANIFEST
2 CACHE:
3 /css/style.css
4 /js/app.js
5 /img/icon.png
6 NETWORK:
7 *
```

Umsetzung in der Arbeit

Um Adhocracy so umzubauen, dass es das Cache Manifest auf jeder Seite ausliefert, wurde im Haupttemplate das `html` Tag um das `manifest` Attribut erweitert. Da alle weiteren Templates das Haupttemplate benutzen, referenzieren nun alle Seiten bei Adhocracy das Cache Manifest `http://adhocracy.de/cache.appcache`.

Das Cache Manifest selbst ist dabei keine richtige Datei auf dem Server, sondern wird bei jedem Request erstellt. Dazu wurde ein neuer Controller implementiert, der auf die URL des Manifests reagiert und es zusammenbaut.

Wie Listing 3.2 zu entnehmen, ist hier die Methode `get_manifest` zuständig. Zunächst schreibt sie den Header und die `NETWORK` Section. Um die statischen Ressourcen einzubinden, ruft sie anschließend mehrere Methoden auf, zum Beispiel `get_css_html`.

Da von mehreren Hundert Bildern im Adhocracy Bilderverzeichnis nur etwa zehn benötigt werden, werden die tatsächlich benutzten Bilder manuell in das Manifest geschrieben und nicht automatisch hinzugefügt, um den Appcache nicht unnötig mit mehreren MB an Ressourcen vollzuladen.

Listing 3.2: Controller, der für das Erstellen des Manifests zuständig ist

```
def get_manifest(self):
    manifest = 'CACHE MANIFEST' + \
               '\nNETWORK:' + \
               '\n*' + \
               '\nCACHE:' + \
               SourceController.get_css_html(href_only=True) + \
               SourceController.get_js_html(src_only=True) + \
               '\n/img/unicoop_logo.png' + \
               '\n/img/uni_duesseldorf_logo.gif' + \
               (...)
    return render_static(manifest, 'text/cache-manifest')
```

Mit dieser Änderung stehen die Seiten, die ein Benutzer besucht, auch offline zur Verfügung. Allerdings aktualisieren sich die Inhalte nicht mehr, da die Seiten nur noch aus dem Application Cache geliefert werden (→ 3.1.1). Besucht ein Benutzer zum Beispiel die Instanzseite `http://adhocracy.de/i/diskussion/`, landet sie als Master-Entry im Cache, sodass der Benutzer keine Änderungen (neue Vorschläge, Kommentare und ähnliches) mehr mitbekommt.

Das Problem könnte gelöst werden, indem bei jeder Änderung an einer Instanz, einem Vorschlag oder einem Kommentar auch das Cache Manifest geändert würde – zum Beispiel in Form einer hochzählenden Versionsnummer, die den Browser dann zum Neuladen alle Ressourcen im Appcache veranlassen würde (→ 3.1.1). Dies ist aber nicht Sinn eines Caches und je nach Häufigkeit der Änderungen an den Daten auch nicht praktikabel.

Es müssen also Lösungen gefunden werden, die es ermöglichen, den Appcache zu verwenden – denn der ist essentiell für ein Offline Fallback – und gleichzeitig dafür sorgen, dass der Benutzer immer die aktuellen Inhalte sieht.

3.1.3. `SETTINGS` Section

Mit der `SETTINGS` Section wurde vor Kurzem ein neuer Bereich zum Cache Manifest hinzugefügt. Sie ist ein Versuch, das Problem veralteter Seiten zu lösen, indem man dem Browser mitteilen kann, Online-Versionen von Ressourcen zu bevorzugen. Der Browser cacht die Ressourcen dann zwar, versucht aber trotzdem bei jedem Request neuere Versionen online abzurufen. Mit der Einstellung `cache-mode: prefer-online` wird dieses Verhalten aktiviert.

Der Vorteil dieser Einstellung ist, dass sie leicht einzupflegen ist und dem Benutzer wieder aktuelle Inhalte präsentiert. Allerdings ist der Sinn dieser Einstellung nicht ganz nachvollziehbar, da man sich mit ihr alle Vorteile eines Caches zu nichte macht. Angenommen, eine Webapplikation nutzt den

Application Cache, um große Bilder lokal zu cachen, damit der Server entlastet und die Datenübertragung verringert wird. Die `cache-mode` Einstellung würde in diesem Fall dafür sorgen, dass die Ressourcen zwar lokal vorliegen, sie nichtsdestotrotz bei jedem Request von Server geladen werden – der Cache wäre also hinfällig.

Aus diesem Grund und weil die `SETTINGS` Section von keinem Browser implementiert wird, kommt sie für ein Offline Fallback nicht in Frage und wird im Rahmen dieser Arbeit nicht weiter betrachtet.

3.1.4. Requests per XHR

Um also die Inhalte für den Benutzer aktuell zu halten, dürfen bestimmte HTML-Seiten *nicht* als Master-Entry (→ 3.1.1) im Cache gespeichert werden. Da das aber derzeit nicht möglich ist, kann der Lösungsansatz aktuell nur darin bestehen, zu verhindern, dass der Browser die HTML-Seiten selbst abruft und sie somit als Master-Entry in den Appcache legt.

Die erste Aufgabe des zu entwickelnden JavaScript Offline Fallback Framework besteht also darin, normale Browserrequests zu unterdrücken und die HTML-Seiten stattdessen manuell über XHR nachzuladen. Startet der Benutzer die Webanwendung, zum Beispiel über die Startseite, muss das Framework dafür sorgen, dass ab jetzt jeder Klick auf einen Link abgefangen und der eigentliche Browserrequest unterdrückt wird.

Auf diese Weise wird verhindert, dass der Browser die neue Seite aufruft und sie als Master-Entry im Appcache speichert. Allerdings muss das Framework den Request anschließend manuell an den Server schicken und die Antwort – die HTML-Seite – im Browser anzeigen.

Dieser Schritt führt zu einem Problem. Da das normale Browserverhalten komplett unterdrückt und der eigentliche Browserrequest in JavaScript erzeugt und abgeschickt wird, funktioniert zum einen die Navigation über die Vor- und Zurückbuttons nicht mehr und ändert sich zum anderen auch nicht mehr die URL in der Adresszeile. Dieser Umstand wird zunächst vernachlässigt und wird später im Abschnitt Browsernavigation (→ 3.1.8) besprochen.

Desweiteren funktioniert dieser Ansatz nur, solange der Benutzer die Webapplikation über die Startseite aufruft. Gibt er direkt eine URL zu einer Unterseite ein, die eigentlich nicht gecacht werden darf, oder kommt er über einen Deeplink, wird die HTML-Seite wieder gespeichert. Da JavaScript erst nach dem HTML geladen wird (→ 2.1.2), wird das Framework auch erst danach initialisiert. Folglich hat man keine Chance, diesen initialen Browserrequest zu unterdrücken. Im Folgenden geht also darum, Lösungen für dieses *Deeplink-Problem* vorzustellen.

Umsetzung in der Arbeit

Bei Adhocracy dürfen weder Instanz- noch Vorschlagseiten gecacht werden. Dazu wurde das Offline Fallback Framework[1] anfangs so implementiert, dass es während der Initialisierungs-Phase alle relevanten Links manipuliert.

Die Methode `prepareLinks` sucht alle Links, die mit `/i/` anfangen und registriert `interpretRequest` als Callback-Methode für das JavaScript `onclick`[2] Event. Diese unterdrückt dann zunächst den normalen Browserrequest mit Hilfe von `event.preventDefault` und ruft die HTML-Seite dann vom Server ab.

Sobald der Server antwortet, wird im Framework die Methode `handleResponse` aufgerufen, welche zunächst nur die Aufgabe hat, die empfangene HTML-Seite im Browser anzuzeigen. Da sie im Laufe der Arbeit aber deutlich komplexer (→ 3.3.1) und von mehreren Stellen des Frameworks aus aufgerufen wird, wurde sie direkt als eigenständige Methode und nicht nur als anonyme Callback-Methode implementiert.

Wenn also ein Benutzer über die Startseite auf Adhocracy zugreift, kann es trotz Appcache benutzt werden, da alle Requests über das Framework laufen. Das Deeplink-Problem gilt aber auch für Adhocracy: kommt ein Benutzer etwa direkt auf eine Instanzseite, wird diese gecacht und alle weiteren Anfragen an diese URL – normale oder über XMLHttpRequests – werden aus dem Cache beantwortet.

URL umschreiben

Zur Lösung des Deeplink-Problems kann man die URL des Requests vor dem Abschicken umschreiben, indem man einen zufällig erzeugten Parameter anhängt. Beispielsweise wird `http://example.com/cached/ressource` so zu `http://example.com/cached/ressource?timestamp=1347720141`. Da für diese neue URL kein Eintrag im Appcache vorliegt, geht der Request daran vorbei zum Server, der dann mit einer neuen HTML Version der Seite antworten kann.

Allerdings handelt es sich dabei um eine ungeeignete Lösung, da sie generell nicht verhindert, dass HTML-Seiten im Appcache landen und und ihn unnötig vollschreiben. Langfristig soll aber genau

[1] Das Framework selbst ist bei Adhocracy eine JavaScript Datei (`/js/offline_fallback.js`). Es wird zusammen mit allen anderen JavaScript Dateien ausgeliefert und im Browser initialisiert, sobald die HTML-Seite fertig geladen ist.

[2] Das `onclick` Event wird ausgelöst, wenn man mit der Maus auf einen Link klickt bzw. bei Touchscreens drauftapt oder den Link in den Fokus setzt und dann Enter drückt.

das verhindert werden, da sich HTML nicht als Datenformat für ein Offline Fallback eignet und die Kommunikation nicht mehr darüber stattfinden soll (→ 3.2).

3.1.5. Hashbang URLs

Wenn der Benutzer die Webapplikation immer über die Startseite aufrufen würde, gäbe es das Deeplink-Problem nicht. Tatsächlich gelangt man aber häufig über externe Links oder Bookmarks auf die Webseite. Daher soll mit den *Hashbang URLs* ein erster Lösungsansatz für das Deeplink-Problem vorgestellt werden, der das URL Schema so ändert, dass es für den Browser so aussieht, als befände man sich immer auf der Startseite.

Hashbang URLs machen sich die On-Page Verlinkung (→ 2.1) zunutze: die Idee ist, dass die URL eines Links in der Art abgeändert wird, dass der Browser den kompletten Pfadteil als On-Page-Anker erkennt. Aus `http://example.com/foo/bar/` wird `http://example.com/#!foo/bar/`. Klickt der Benutzer auf einen dieser Links, sucht der Browser im aktuellen HTML-Dokument ein Element mit der ID `!foo/bar/` – es wird dabei aber *kein* neuer Request abgeschickt. Auch wenn der Benutzer die URL direkt eingibt, wird der On-Page-Anker nicht mit an den Server gesendet.

Anstatt also wie bisher den Klick auf einen Link abzufangen und den Request manuell zu erzeugen, werden nun die URLs aller relevanten Links umgeschrieben. Klickt ein Benutzer nun auf einen dieser Links, ändert sich bis auf den Hash der URL nichts. Es ist also Aufgabe des Frameworks die Hash-Änderung zu erkennen, um den neuen Inhalt per XHR nachzuladen.

Vorteil dieser Methode ist es, dass die Master-Entries sogar bei Deeplinks verhindert werden, da der Browser scheinbar die ganze Zeit auf der Startseite der Webanwendung verharrt. Ein weiterer großer Vorteil ist, dass man jetzt auch kein Problem mehr mit der Browsernavigation hat, da das Konzept der On-Page-Verlinkung in den Browser integriert ist. Der Browser weiß also, dass er auch einen Eintrag im Browserverlauf erstellen muss, wenn sich der URL-Hash ändert. Damit funktioniert auch die Navigation über den Zurückbutton des Browsers wieder – das Framework muss dann nur für den richtigen Inhalt sorgen.

Neben diesen Vorteilen hat haben Hashbang URLs allerdings auch viele Nachteile, etwa, dass sie ganz offensichtlich das Konzept der On-Page-Verlinkung stört, da das Springen zu einem Element auf der Seiten über einen One-Page-Anker nicht mehr möglich ist. Desweiteren wird der Austausch von Links zwischen Benutzern der Offline Fallback Variante und Benutzern der bisherigen reinen online Variante unmöglich.

Ein weiterer großer Nachteil ist, dass ein Fehler im JavaScript dazu führt, dass die gesamte Webapplikation nicht mehr reagiert. Im Gegensatz dazu führt bei der zuvor beschriebenen Variante, in der der normale Browserrequest durch das `onclick` Event unterdrückt wird, ein Fehler im JavaScript nur dazu, dass der reguläre Browserrequest wieder ausgeführt wird.

Viele Webapplikation, etwa Twitter[twt], haben bis vor Kurzem noch das Hashbang URL Konzept eingesetzt. Auch Google setzt es heute noch für seine Nexus Produktseite[nex] ein. Allerdings wird für immer mehr Webapplikationen, aufgrund der vielen Nachteile, nach Alternativen gesucht.

Umsetzung in der Arbeit

In dieser Arbeit werden Hashbang URLs bei Adhocracy trotz ihrer Nachteile eingesetzt, da sie derzeit noch die einzige Lösungsmöglichkeit für das Deeplink-Problem bieten, die zudem noch in vielen Browsern funktioniert. Es gibt zwar alternative Ansätze (→ 3.1.6, → 3.1.7), diese funktionieren aber in nur sehr wenigen bis gar keinen Browsern.

Bei der Umsetzung für Adhocracy wurde zunächst die `prepareLinks` Methode geändert, sodass sie nun nicht mehr das `onclick` Event registriert, sondern alle relevanten Links umschreibt. Desweiteren wird während der Initialisierungsphase zusätzlich noch `hashChangeEvent` als Callback-Methode für das `window.onhashchange` Event registriert, das bei jeder URL-Hash Veränderung ausgelöst wird – also inbesondere auch dann, wenn der Benutzer auf einen Link klickt. Diese sorgt dann für das Laden des Inhalts. Die Methode `interpretRequest` wird demnach nicht mehr benötigt.

3.1.6. Cache Manifest nur noch auf der Startseite

Als zweiten Lösungsansatz für das Deeplink-Problem wird ein Umbau der Webapplikation vorgeschlagen, um das Cache Manifest *nur noch* auf der Startseite ausliefern zu lassen. Auf diese Weise verhindern wir effektiv das Erstellen ungewollter Master-Entries auf Unterseiten.

Kommt der Benutzer *nicht* über die Startseite an, machen wir den ganzen Sinn eines Offline Fallback zunichte: es wird kein Manifest mehr ausgeliefert, weshalb Ressourcen nicht gecacht und auch nicht offline zur Verfügung gestellt werden. Mit anderen Worten: es gibt *kein Offline Fallback mehr*, wenn der Benutzer über einen Deeplink zur Seite gelangt.

Das bedeutet also, dass man den Browser auf eine andere Art dazu bringen muss, das Manifest zu laden und die expliziten Ressourcen zu cachen. Dabei reicht es allerdings nicht, das eigentliche Cache

Manifest einfach mit Hilfe von XHR nachzuladen. Vielmehr muss der Browser eigenständig eine Seite aufrufen, die ein Manifest ausliefert, damit der Application Cache initialisiert wird.

Damit der Benutzer von diesen Abläufen nichts mitbekommt und wir auch nicht die aktuelle Seite verlassen müssen, wird eine Art Webseite in der Webseite nötig. In dieser könnten wir dann zum Beispiel die Startseite anfordern – die das Manifest immernoch referenziert –, um den Appcache zu initialisieren.

IFrames

IFrames bieten die Möglichkeit, vollständige Webseiten in bestehende Webseiten einzubinden[ifr] und werden häufig eingesetzt, um externe Webseiten auf der eigenen anzuzeigen. Ein Beispiel hierfür ist die Like Box von Facebook[lbf12].

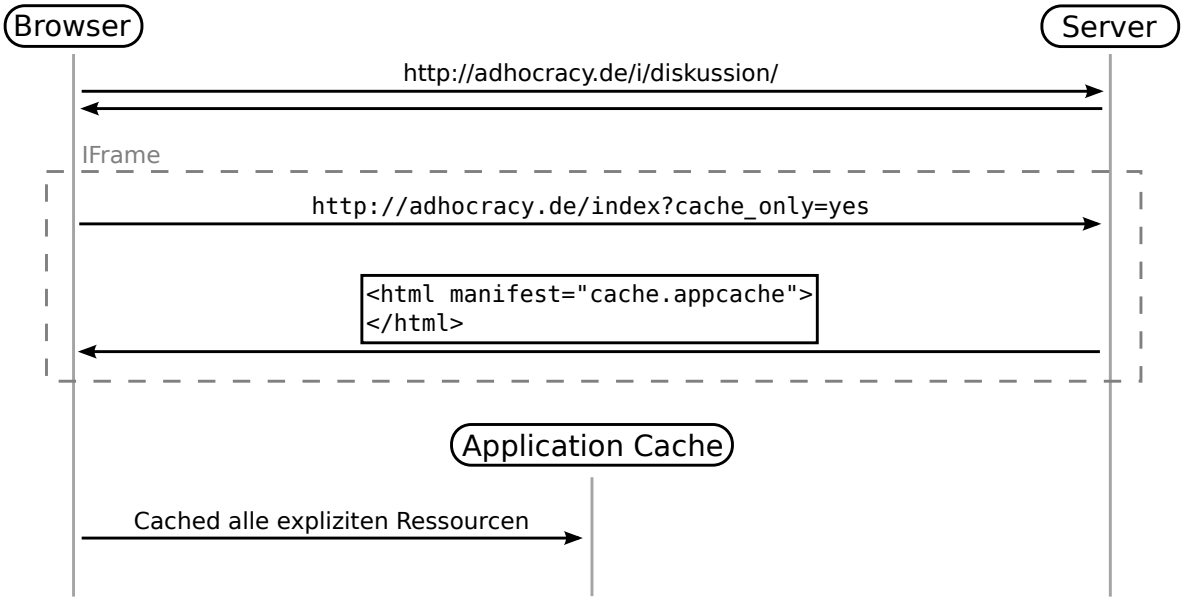
Beim Offline Fallback kann ein IFrame also verwendet werden, um den Browser dazu zu veranlassen, die Startseite und damit das Manifest zu laden und den Appcache zu erstellen. Die Idee dahinter ist, dass das IFrame nicht vom Server direkt in das HTML eingebunden wird, sondern das Framework es während der Initialisierungs-Phase erstellt und einbettet. Auf diese Weise kann zuvor auch noch geprüft werden, ob der Browser alle nötigen Vorraussetzungen zur Nutzung des Offline Fallback erfüllt.

Abbildung 3.3 verdeutlicht die Idee: ist die Hauptseite `http://adhocracy.de/i/diskussion/` fertig geladen, erstellt das Framework ein neues IFrame, setzt dessen URL auf `http://adhocracy.de/index?cache_only=yes`[3] und bettet es anschließend auf der Hauptseite ein. Mit Hilfe von CSS kann man es verstecken, sodass der Benutzer nichts bemerkt. Der Browser erkennt das IFrame und lädt die Webseite mit der angegebenen URL. Von da an läuft alles wie zuvor ab: der Browser erkennt das Manifest, erstellt den Appcache und speichert alle expliziten Ressourcen. Das Gute dabei ist jetzt, dass die Instanzseite nicht im Cache landet, da diese ja kein Manifest direkt ausgeliefert hat.

Mit Hilfe von IFrames haben wir das Deeplink-Problem gelöst und erreicht, dass HTML-Seiten nicht mehr ungewollt gecacht werden. Allerdings stößt man an dieser Stelle auf ein neues Problem: da für diese URL kein Eintrag im Appcache vorhanden ist, ruft der Browser die Seite initial immer vom Server ab. Ist dieser einmal nicht erreichbar oder der Benutzer offline, schlägt der Request fehl und der Browser zeigt die typische „Server nicht erreichbar“-Fehlermeldung an. Man bräuchte demnach

[3] Der Parameter `cache_only` veranlasst den Server dazu, eine Minimalversion der Startseite auszuliefern, in der im Grunde nichts drin steht und nur das Manifest eingebunden ist.

Abbildung 3.3.: Manifest per IFrame



einen Mechanismus, der dem Browser mitteilt, dass die angefragten Daten dieser URL woanders herkommen und er keinen Request an den Server schicken soll.

Verwendung eines Intercept

Seit der 20. Version von Google Chrome wird zur Lösung dieses Problems ein neuer Bereich im Cache Manifest behandelt: mit Hilfe der `CHROMIUM-INTERCEPT` Section kann man dem Browser mitteilen, dass Anfragen an bestimmte URLs durch bereits gecachte Ressourcen beantwortet werden sollen. Die Funktionsweise ähnelt dabei der normalen `FALLBACK` Section des Manifests (→ 3.1.1), mit dem Unterschied, dass die `FALLBACK` Section nur greift, wenn Ressourcen nicht gecacht werden konnten und der Benutzer offline ist.

Bei der `CHROMIUM-INTERCEPT` Section wird hingegen immer die Intercept-Ressource aufgerufen. Das bedeutet, man kann dort alle Seiten angeben, die kein Manifest ausliefern und kann den Inhalt für diese Seiten in der Intercept-Ressource erstellen. Listing 3.3 zeigt wie ein deratiges Manifest für Adhocracy aussehen würde: alle Instanzseiten fangen mit `/i/` an und können jetzt mit Hilfe der neuen Section auf eine spezielle Intercept Ressource geleitet werden. Diese ist dann dafür zuständig, den richtigen Inhalt für diese URL auszuliefern. Da die Änderung nicht standardisiert ist, ist zusätzlich noch eine Bearbeitung des Manifest Headers in `CHROMIUM CACHE MANIFEST` notwendig.

Listing 3.3: Beispiel für ein Cache Manifest mit CHROMIUM-INTERCEPT

```
1 CHROMIUM CACHE MANIFEST
2 ...
3 CHROMIUM-INTERCEPT:
4 /i/ return /intercept
```

Der Vorteil des `CHROMIUM-INTERCEPT` ist eindeutig: man kann jetzt auch offline auf Seiten zugreifen, die keinen Eintrag im Application Cache haben. Allerdings hat diese Vorgehensweise auch den großen Nachteil, dass sie nur in der neuesten Version von Chrome funktioniert und daher nicht für ein flächendeckendes Offline Fallback geeignet ist.

Verwenden eines Fallback

Obwohl sich `CHROMIUM-INTERCEPT` und `FALLBACK` Section stark ähneln, die `FALLBACK` Section standardisiert ist und sie von allen Browsern, die den Application Cache implementieren, unterstützt wird, eignet sich die `FALLBACK` Methode nicht für die vorliegende Arbeit, da im Rahmen des Offline Fallback die lokalen Daten verwendet werden sollen. Das nämlich ist der entscheidende Nachteil der `FALLBACK` Section: sie versucht die Ressourcen immer zunächst online abzurufen, auch wenn sie lokal vorliegen und dieser Request eingespart werden könnte. Da wir zudem davon ausgehen, dass der Server die meiste Zeit erreichbar ist, würde diese Methode unnötigen Traffic erzeugen.

Umsetzung in der Arbeit

Für das Offline Fallback bei Adhocracy wurde der vorgeschlagene Lösungsansatz (→ 3.1.6) nicht weiter verfolgt, da er sehr kompliziert umzusetzen ist und viele Änderungen am Framework verursacht hätte. Die Entwicklung des Framework war bereits weit fortgeschritten, als die `CHROMIUM-INTERCEPT` Section vorgestellt wurde. Desweiteren hätte die geringe Browserunterstützung diesen Mehraufwand nicht gerechtfertigt.

Als Proof-of-Concept dieses Lösungsansatzes wurden dennoch zwei kleine Demo Applikationen implementiert[4]. Sie bestehen im Grunde nur jeweils aus einer Startseite und zwei Unterseiten. Letztere sollen nicht im Application Cache gespeichert werden, aber trotzdem offline verfügbar sein – sie sind also vergleichbar mit den Instanzseiten bei Adhocracy. Die Applikationen laden, wie oben erläutert, das Manifest in einem versteckten IFrame nach. Während die eine Demo Applikation das

[4] Die Quellcodes der Applikationen sind auf der beiliegenden CD unter `/demos/manifest_via_iframe/intercept` und `demos/manifest_via_iframe/fallback` zu finden. Sie können auf der Webseite `http://pigsel.net/demos/manifest_via_iframe/` getestet werden.

`CHROMIUM-INTERCEPT` Konzept verwendet, benutzt die andere die normale `FALLBACK` Section, um die Unterseiten offline verfügbar zu machen.

3.1.7. Webworker

Ein dritter Lösungsansatz für das Deeplink-Problem und die damit verbundenen automatischen Master-Entries beschreibt den Einsatz von **Webworkern** im Browser.

Webworker bieten die Möglichkeit, echte Nebenläufigkeit in JavaScript zu integrieren. Im Browser ist JavaScript eigentlich so konzipiert, dass es nur in einem Thread läuft. Nebenläufigkeit wird durch den Einsatz von Callbacks erreicht bzw. simuliert. Mit dem Konzept der Webworker kann man nun JavaScripte definieren, die in einem eigenen, echten Betriebssystem-Thread laufen. Einsatzgebiete dafür sind beispielsweise Webapplikationen, die komplexe Berechnungen durchführen, dabei aber nicht das User Interface blockieren dürfen.

Im Zuge eines Bugreports des W3C[dac] wurde die Idee vorgestellt, Webworker im Bereich Offline Webapplications so einzusetzen, dass sie als lokaler Proxy für alle Requests dienen. Der Proxy entscheidet dann, ob ein Request normal ausgeführt oder an eine andere URL weitergeleitet werden soll. Alternativ kann der Proxy – das Webworker Script – auch die Daten liefern und so die Anfrage beantworten. Geht der Benutzer also zum Beispiel auf `http://adhocracy.de/i/disskusson/`, versteht der Proxy, dass diese Seite nicht gecacht werden soll, unterdrückt den Request und beantwortet die Anfrage selbst.

Diese Idee setzt allerdings eine tiefe Browserintegration voraus, da der Browser den Request an das Proxy-Skript geben müsste, noch bevor er ihn abschickt. Dieses Feature wird derzeit noch von keinem Browser unterstützt und implementiert, daher wird dieser Ansatz im Rahmen dieser Arbeit nicht weiter verfolgt.

3.1.8. Browsernavigation

Verwendet man für das Offline Fallback *nicht* den Lösungsansatz der Hashbang URLs (→ 3.1.5), müssen, wie erläutert (→ 3.1.4), die normalen Browserrequests komplett unterdrückt werden.

Dies führt zu zwei großen Problemen, auf die viele JavaScript lastige Webapplikationen stoßen:

- die URL in der Adresszeile ändert sich nicht, wenn ein Benutzer auf einen Link klickt

- die Navigation über die Vor- und Zurück-Buttons funktioniert nicht wie erwartet

Um diese beide Aufgaben kümmert sich normalerweise der Browser: sobald eine neue Seite aufgerufen wird, ändert er die Adresszeile und erstellt einen neuen Eintrag in seiner Historie. Klickt der Benutzer dann auf den Zurück-Button, wird der oberste Eintrag aus der Historie entfernt und der Browser stellt den Inhalt der vorherigen Seiten wieder her. Werden aber alle normalen Browserrequests unterdrückt, weiß der Browser nichts von dem neuen Inhalt und ein Klick auf den Zurück-Button hat nicht den erwarteten Effekt.

Geht man zum Beispiel auf den Amazon Kindle Online Reader[5], sieht man eine Übersicht seiner eBooks. Wählt man eines dieser Bücher durch Anklicken aus, wird es geladen und angezeigt. Die URL der Adresszeile ändert sich aber nicht, und auch wenn man den Zurück-Button benutzt und vermutet, so wieder zur Übersicht zurückzukehren, landet man stattdessen auf der zuvor aufgerufenen Seite.

HTML5 History API

Um für den Benutzer also wieder ein normales Verhalten der Webapplikation zu erzeugen, muss das Framework dahin gehend erweitert werden, dass es die Adresszeile anpasst und die Navigation über die Vor- und Zurück-Buttons des Browsers ermöglicht.

In HTML5 wurde mit der **History API**[his12] ein Feature vorgestellt, mit dem dies funktioniert: sie erlaubt das direkte Interagieren mit der Browser Historie über JavaScript. Das bedeutet im einfachsten Fall, dass man die Vor- und Zurück-Buttons des Browsers ansteuern kann. Man kann aber die Historie auch direkt durch Hinzufügen bzw. Ersetzen von Einträgen beeinflussen. Desweiteren kann man auf das Klicken des Zurück-Buttons reagieren.

Die History API gehört zu den am meisten verwendeten HTML5 Features und wird unter anderen von den sozialen Netzwerken Facebook[fcb] und Google+[gop] sowie vom Online Pinboard Pinterest[pin] genutzt.

Um die History API zu verwenden, muss das Framework zusätzlich zum Request noch einen Eintrag im Browserverlauf erstellen. Dazu dient die Methode `window.history.pushState`. Ihr kann man als Parameter unter anderem die Adresszeile übergeben, die im Browser angezeigt werden soll. Auf diese Weise wird ein Teil des erwarteten Browserverhaltens wieder hergestellt. Desweiteren muss das Framework einen Listener für das `window.onpopstate` Event registrieren. Dieses wird durch

[5] Ein Service von Amazon, um die Kindle eBooks im Browser zu lesen, `https://read.amazon.de`

Klicken des Zurück-Buttons ausgelöst. Hier muss das Framework dafür sorgen, dass der alte Zustand der Webanwendung wieder hergestellt wird.

In dieser Arbeit war die Verwendung der History API die bevorzugte Variante und wurde daher zunächst auch implementiert. Im Verlauf der Arbeit ist jedoch klar geworden, dass das Deeplink-Problem derzeit nur mit Hasbang URLs zu überwinden ist, woraufhin das Framework auf sie umgestellt wurde.

3.2. Server anpassen

Mit den zuvor vorgestellten Lösungsansätzen wurde erreicht, dass wir den Application Cache für statische Ressourcen nutzen können, ohne aber HTML-Seiten ungewollt mitzuspeichern. Damit ist der erste Schritt zu einem Offline Fallback getan, sodass wir uns nun der Anpassung des Servers zuwenden können, der bisher immernoch fertige HTML Ausgaben an den Client sendet.

Wie bereits erläutert, eignet sich HTML nicht als Datenformat für ein Offline Fallback, da jede mögliche HTML-Seite lokal gespeichert werden müsste, um sie offline zur Verfügung zu haben. Weiterhin ist es schwierig, Änderungen an den eigentlichen Daten vorzunehmen, wenn sie nur in Form von HTML vorliegen.

Daher soll der Server in der Weise umgebaut werden, dass er die Daten und Templates spererat ausliefern kann, damit das Framework daraus dann das HTML erstellen und es im Browser anzeigen kann (→ 3.3).

3.2.1. Model: Daten statt HTML

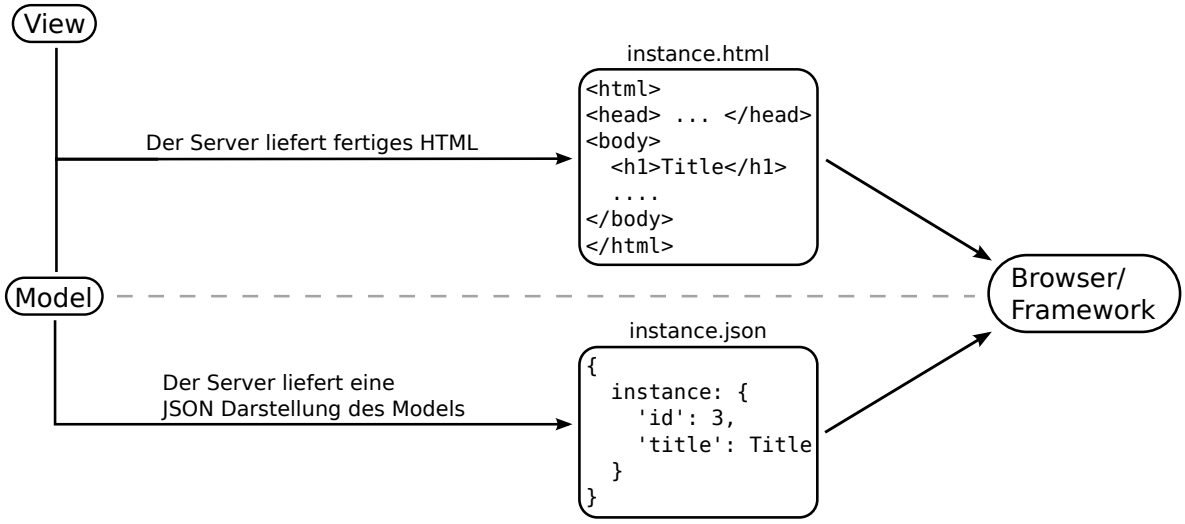
Die erste Änderung am Server betrifft die Model-Klassen, die bisher lediglich als Datencontainer dienten und in der direkten Beziehung zum Client keine große Rolle spielten. Dieser Umstand ändert sich nun, da alle relevanten Model-Klassen dahin gehend erweitert werden müssen, dass sie eine Methode bereitstellen, mit der sie sich selbst in einem festgelegten Datenformat darstellen können.

Welches Datenformat dabei für den Austausch der Daten zwischen Server und Client/ OF-Framework verwendet wird, ist im Prinzip egal. Im Bereich Webentwicklung sind die jedoch die beiden Formate Extensible Markup Language (XML)[xml] und JavaScript Object Notation (JSON)[jso] besonders populär, wobei sie natürlich sowohl Vor- als auch Nachteile mit sich bringen. Für den Datenaustausch

zwischen JavaScript und einem Server wird häufig JSON benutzt, da es platzsparender als XML ist und den großen Vorteil hat, in JavaScript direkt verwendbar zu sein, da JSON Strings eine direkte Repräsentation von JavaScript Objekten sind.

Anstatt des fertigen HTML werden also die reinen Daten an den Client geschickt. Abbildung 3.4 stellt dem derzeitigen Ablauf die neue Methode gegenüber: beim alten Ablauf (oben) erstellt die serverseitige View aus den Modeldaten das fertige HTML, welches dann zum Client geschickt wird. Beim neuen Ablauf werden hingegen die reinen Daten an den Client geschickt.

Abbildung 3.4.: Vergleich der Abläufe HTML vs. reine Daten



Umsetzung in der Arbeit

In der vorliegenden Arbeit wird als Datenformat JSON verwendet, weil es im Vergleich zu XML eine geringere Größe hat und es mehr Templatesprachen gibt, die mit JSON arbeiten können. Demnach werden alle relevanten Model-Klassen – Instanz, Vorschlag und Kommentar – so erweitert, dass sie sich selbst als JSON Dictionary darstellen können, wie Tabelle 3.1 verdeutlicht.

Die JSON Darstellung einer Instanz ist hierarchisch aufgebaut: auf oberster Ebene befinden sich alle direkten Informationen zur Instanz, gefolgt von einer Liste mit allen Vorschlägen und pro Vorschlag wieder einer Liste mit allen Kommentaren. Listing A.2 im Anhang zeigt exemplarisch, wie eine Instanz im JSON Format aussieht.

`to_mustache_dict()`	Erstellt ein JSON Dictionary aus dem Objekt. Wird von allen drei Model-Klassen implementiert. Wird vom Controller aufgerufen.
`get_proposals()`	Erstellt eine Liste mit allen Vorschlägen zu einer Instanz. Wird nur von dem Instanz-Model implementiert.
`get_comments()`	Erstellt eine Liste mit allen Kommentaren zu einem Vorschlag. Wird nur von dem Vorschlag-Model implementiert.

Tabelle 3.1.: Adhocracy Server: Daten statt HTML

3.2.2. View: Templates anpassen

Daten alleine reichen dem Client nicht aus, um daraus das HTML für die Darstellung zu erzeugen. Daher braucht das Framework des Offline Fallback neben den Daten auch die Views seperat. In den meisten Webapplikationen ist es aber derzeit nicht vorgesehen, „rohe“ Templates auszuliefern, weshalb zunächst ein Controller erstellt werden muss, der diese Aufgabe übernimmt.

Die meisten Frameworks und somit auch die Webapplikationen, die darauf aufbauen, haben das Problem, dass die eingesetzte Templatesprache sehr stark an das Framework und dessen Programmiersprache angepasst ist. Dies führt in der Regel dazu, dass in den Templates eine Menge Anwendungslogik und Quellcode enthalten ist und sie nicht nur die Daten darstellen. Wenn das der Fall ist, können die Templates nicht einfach an den Client ausgeliefert werden und müssen zunächst in eine neue Templatesprache übersetzt werden.

Diese neue Templatesprache sollte dann sowohl vom Server als auch vom Client verwendet werden können. Im Fall des Client bedeutet das, dass es einen JavaScript-Interpreter für diese Temaplatesprache geben muss. Sie sollte außerdem die Eigenschaft haben, keine Logik zu erlauben und Daten quasi nur in Platzhalter einzusetzen bzw. wenn sie Logik erlaubt, dann in einer einfachen und Programmiersprachen-unabhängigen Form. Da die meisten Webapplikationen die Ausgaben derzeit nur auf dem Server erstellen, gibt es nicht viele Templatesprachen, die diese Kriterien erfüllen.

Mustache

Die derzeit bekannteste Templatesprache ist **Mustache**[mus]. Sie kann in allen Programmierumgebungen eingesetzt werden, in denen es einen passenden Interpreter gibt. Letztere gibt es für viele Programmiersprachen, unter anderem für Python und JavaScript. Mustache definiert eine eigene Me-

tasprache, in der keine programmiersprachen-abhängige Logik erlaubt ist, sondern nur sogenannte *Tags*. Diese Tags setzen dann unter anderem die Konzepte von Variablen, If-Anweisungen und For-Schleifen um.

Das einfachste Tag ist die *Variable* – ein reiner Platzhalter. Er wird beim Übersetzen des Templates direkt mit den entsprechenden Daten befüllt. Als Daten erwartet Mustache Hashs in JSON Schreibweise. Listing 3.4 zeigt ein einfaches Beispiel eines Mustache Templates und wie die Ausgabe mit den entsprechenden Daten aussehen würde.

Listing 3.4: Beispiel für ein Mustache Template mit einem einfachen Variable-Tag

```
Template:
Hallo {{ name }}.

Eingabedaten:
{
    'name': 'Max Mustermann'
}

Ausgabe:
Hallo Max Mustermann.
```

Ein komplexerer Tag-Typ ist der sogenannte *Block*. Er wird entweder gar nicht, ein- oder mehrmals durchlaufen und entspricht etwa dem Konzept von If-Anweisungen und For-Schleifen. Wie in Listing 3.5 zu sehen ist, können innerhalb eines Blocks beispielsweise wieder einfache Variablen stehen.

Listing 3.5: Beispiel für ein Mustache Template mit einem Block-Tag

```
Template:
{{ #persons }}
    Hallo {{ name }}.
{{ /persons }}

Eingabedaten:
{
    'persons': [
        'name': 'Max Mustermann',
        'name': 'Ute Mustermann'
    ]
}

Ausgabe:
Hallo Max Mustermann.
```

```
Hallo Ute Mustermann.
```

Der Vorteil dieses Tag-Konzepts ist, dass die gleichen Mustache Templates sowohl auf dem Server als auch auf dem Client verwendet werden können – vorausgesetzt, es gibt einen Interpreter für die auf dem Server bzw. Client benutzte Programmiersprache.

Umsetzung in der Arbeit

Die aktuellen Templates bei Adhocracy eignen sich nicht für die Verwendung im Browser, da sie in Mako geschrieben sind, was speziell auf Python ausgelegt ist (→ 2.4). Zudem steckt noch viel Python Code und Anwendungslogik in den Templates, wie in Listing 3.6 zum Beispiel in Zeile 4 und 11 zu sehen ist.

Für das Offline Fallback wurden daher alle relevanten Templates in Mustache neu geschrieben. Diese Sprache wurde verwendet, da es Interpreter sowohl für Python als auch für JavaScript gibt und es direkt mit den JSON-Daten vom Server umgehen kann. Die relevanten Templates für das Offline Fallback sind unter anderem `instance_show`, `proposal_show`, `proposal_edit` und `comment_show`. Die Schwierigkeit dabei bestand darin, die gesamte Logik aus den Mako Templates sinnvoll in die Controller bzw. Models zu portieren, sodass die JSON-Daten entsprechend erstellt werden konnten.

Listing 3.7 zeigt das gleiche Template wie zuvor, angelegt in Mustache. Hier sieht man, dass es keine von Programmiersprachen abhängige Logik gibt – lediglich Platzhalter, die später gefüllt werden.

Listing 3.6: proposal_show.mako

```
1  (...)
2  <div class="mainbar">
3      <div class="top_actions aside">
4          %if can.proposal.edit(c.proposal):
5              <a class="button title edit ttip"
6                 title="${_('Edit this proposal.')}"
7                 href="${h.entity_url(c.proposal, member='edit')}">${_("edit")}</a>
8          %endif
9      </div>
10     <h2>${_("Goal and description")}</h2>
11     ${tiles.page.inline(c.proposal.description,
12                         hide_discussion =
13                             c.instance.use_norms and len(c.proposal.selections))}
14 (...)
```

Listing 3.7: proposal_show.mustache

```
1  (...)
2  <div class="mainbar">
3      <div class="top_actions aside">
4          {{#top_actions_aside}}
5              <a href="{{href}}" title="{{title}}" class="{{class}}">{{text}}</a>
6          {{/top_actions_aside}}
7      </div>
8      <h2 data-i18n="goal_and_description">Goal and description</h2>
9      <div class="tile">{{description}}</div>
10 (...)
```

3.2.3. Controller: HTML oder Daten

Der Server ist nun also in der Lage, die reinen Daten – anstelle fertiger HTML-Seiten – und die angepassten Templates seperat auszuliefern. Da aber auch Browser, die kein JavaScript können, die Webapplikation weiterhin benutzen können sollen (→ 2.3), muss der Server entscheiden können, ob er HTML oder Daten ausliefern soll.

Um diese Unterscheidung treffen zu können, gibt es zwei Möglichkeiten: der **HTTP `Accept` Header** oder die **Request URL**. Mit Hilfe des `Accept` Headers kann der Client den mime-type des erwarteten Datenformats des Antwortinhalts angeben, etwa `application/json` für JSON. Der jeweilige Controller kann daraufhin anhand dieses Headers entscheiden, was er zurückgibt.

Die zweite Möglichkeit ist, die Unterscheidung anhand der Request URL festzulegen. Man kann zum Beispiel über einen Parameter im Request entscheiden, ob man HTML oder nur die Daten erwartet. Aus `http://adhocracy.de/i/diskussion/` wird dann zum Beispiel `http://adhocracy.de/i/diskussion/?format=json`. Auch hier könnten die betroffenen Controller anhand des Formats entscheiden, was sie ausliefern sollen.

Welche der Methoden man verwendet, ist prinzipiell egal, wobei es am besten wäre, würden auf dem Server beide Methoden implementiert. Dies verursacht keinen nennenswerten Mehraufwand und verschafft außerdem Gewissheit: auch wenn man im Bereich Webentwicklung den Header eines Requests über JavaScript verändern kann, mag es zu Situationen kommen, in denen dies nicht möglich ist, beispielsweise in der Entwicklung auf mobilen Endgeräten. Der Abruf einer benutzerdefinierten URL sollte demgegenüber immer möglich sein.

Umsetzung in der Arbeit

In dieser Arbeit wurden alle relevanten Controller – Instanz, Vorschlag und Kommentar – auf diese Weise erweitert, dass sie sowohl anhand des HTTP `Accept` Header als auch des Datenformats entscheiden können, ob sie HTML oder JSON-Daten an den Client liefern sollen. Listing 3.8 verdeutlicht dies anhand der `show` Methode aus dem Instanz-Controller.

Listing 3.8: Auschnitt aus dem Instanz-Controller

```
1 def show(self, id, format='html'):
2 (...)
3     if 'application/json' in request.headers.get('accept') or 'json' in format:
4         c.mustache_values = c.page_instance.to_mustache_dict(with_proposals=True,
5                                                              with_comments=True)
6         return render_real_json(c.mustache_values)
7 (...)
```

3.3. Daten lokal verarbeiten

Zum jetzigen Zeitpunkt der Arbeit sind die Änderungen am Server abgeschlossen, sodass es im Folgenden um die Erweiterung des Offline Fallback Framework gehen soll, damit es die neuen Möglichkeiten des Servers auch ausnutzt. Es soll also nicht mehr die HTML-Seiten, sondern die Daten und Templates seperat abrufen und daraus dann das HTML erstellen.

3.3.1. Daten und Templates laden

Um an die Daten zu gelangen, kann man den `Accept Header` oder die URL des Requests ändern (→ 3.2.3). Beides stellt kein großes Problem dar, da das Framework aufgrund der Verwendung des Application Cache (→ 3.1.4) bereits alle Requests asynchron mit JavaScript abschickt und man in diesem Kontext ohne großen Aufwand auch den Header oder die URL anpassen kann.

Zu einem Datensatz – bei Adhocracy zum Beispiel einer Instanz – ist das passende Template nötig, um daraus das HTML erstellen zu können. Wie alle Ressourcen werden auch Templates über ihre URI referenziert. Das bedeutet, das Framework muss die URI des passenden Templates zu einem Datensatz kennen. Es bietet sich an, diese Information direkt mit dem Datensatz zu verknüpfen. Der Vorteil dabei ist, dass so alle Informationen zum Datensatz selbst sowie zu dessen Darstellung an einem Ort gebündelt werden. Die URI zu einem Datensatz muss dann entweder manuell in der Methode, die

diesen Datensatz auf dem Server erstellt, eingetragen werden oder man verfolgt einen allgemeineren Ansatz.

Wenn man in der Webapplikation vorgibt, dass der Name eines Templates aus dem Name der Model-Klasse und der Action des Controllers zusammen gesetzt ist – zum Beispiel `instance_show` für das Instanz Model und die Action `show` – und zudem noch alle relevanten Model-Klassen von einer Eltern-Klasse erben, kann man das Eintragen der richtigen Templates verallgemeinern. Dazu muss dann lediglich die Elternklasse eine Methode bereitstellen, welche alle Einträge erstellt.

Bei Adhocracy sind alle Model-Klassen von einer gemeinsamen Basis-Klasse vererbt. Diese könnte eine Methode bereistellen, welche beispielsweise den Template-Namen für die Action `show` wie folgt erstellt: `show_tmpl = '{0}_show.mustache'.format(type(self).__name__)`. Auf diese Weise muss man nicht in jeder Model-Klasse einzeln die URIs der Templates eintragen.

Man könnte die Verknüpfung von Datensatz zu Template auch hartcodiert in das Framework schreiben, was aber sehr unflexibel und fehleranfällig wäre: fügt man beispielsweise ein Template hinzu oder entfernt es, muss auch immer das Framework aktualisiert – also eine neue Version des Frameworks an den Client ausgeliefert – werden.

In jedem Fall muss das Template im Anschluss an die Daten über XHR geladen werden. Oft sind mehrere Templates nötig, um die Daten darzustellen, was zu einem Problem führt: aufgrund der Asynchronität von JavaScript bzw. der XMLHttpRequests kann man im Framework nicht einfach blockieren und warten bis alle Templates fertig geladen sind. Daher muss ein gesonderter **Templatehandler** entwickelt werden, der die Aufgabe hat, alle Templates zu laden und dem Framework anschließend Bescheid zu geben. Dazu bekommt dieser Handler als Parameter eine Liste mit allen zu ladenden Templates sowie einer Callback Methode.

Umsetzung in der Arbeit

In der vorliegenden Arbeit wurde die Methode `hashChangeEvent` (→ 3.1.5) so erweitert, dass sie den HTTP `Accept` Header des Requests auf den mime-type `application/json` setzt, um die Daten vom Server abzurufen.

Desweiteren wurde die Methode `handleResponse` angepasst: sie empfängt nun anstelle von HTML-Seiten, die lediglich im Browser angezeigt werden müssten, JSON-Daten. In diesen ist auch die URL zum Template zu finden, die an den Templatehandler übergeben wird.

Der Templatehandler[6] wurde so implementiert, dass er dem Framework die beiden Methoden `init` und `load` bereitstellt.

Während der Initialisierungsphase ruft das Framework die `init` Methode auf und erstellt so eine Instanz des Templatehandler, die dann für die Laufzeit des Offline Fallback bereitsteht. Hier übergibt das Framework auch die Callback-Methode `templatesLoaded`.

Die `load` Methode wird vom Framework aufgerufen, wenn Templates geladen werden sollen. In einem Parameter bekommt sie eine Liste mit allen zu ladenden Templates übergeben und erstellt für jedes Template einen eigenen XHR. Das Problem der Asynchronität löst der Templatehandler mit Hilfe eines Zählers: für jedes geladene Template wird der Zähler erhöht. Sobald alle Templates geladen sind, wird die Callback-Methode des Framework aufgerufen.

3.3.2. Daten lokal rendern

Wie erläutert (→ 3.2.2), sollte die Templatesprache so gewählt sein, dass es eine JavaScript Bibliothek gibt, die aus den vorliegenden Daten und Templates das HTML erstellen kann. Sobald dem Framework also Daten und Templates vorliegen, kann das HTML erstellt und im Browser angezeigt werden.

Abbildung 3.5 vergleicht den bisherigen mit dem neuen Ablauf. Bisher (oben) hat der Server fertige HTML-Seiten geschickt, die dann im Browser nur angezeigt wurden. Mit der Möglichkeit, das HTML lokal zu erstellen (unten), ruft man jetzt nur noch die Daten und die Templates vom Server ab.

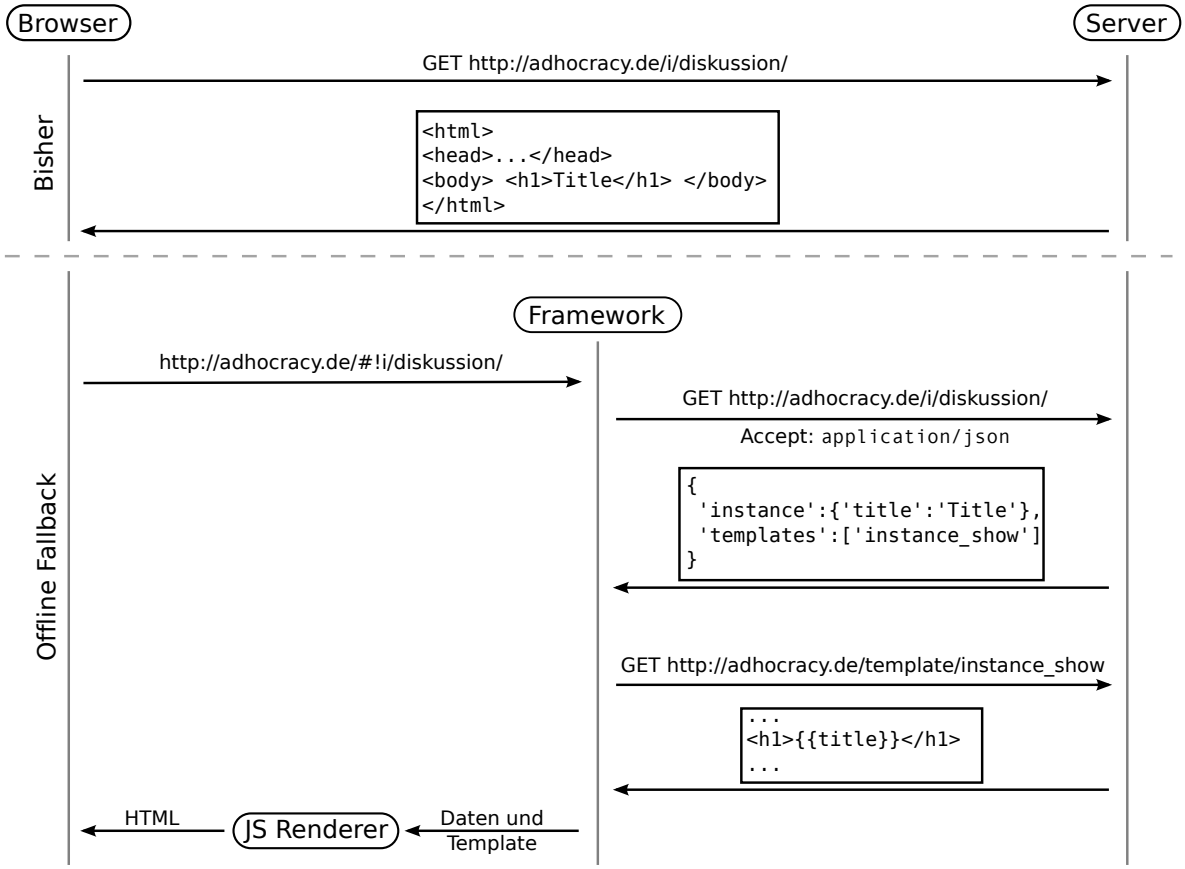
Es wäre wenig effizient und würde die offline Nutzung unmöglich machen, wenn alle Templates immer vom Server geladen werden müssten. Daher werden sie als explizite Einträge in das Cache Manifest geschrieben, damit sie auch lokal gespeichert werden.

Umsetzung in der Arbeit

In dieser Arbeit wird Mustache als Templatesprache verwendet (→ 3.2.2). Demnach ist **mustache.js**[mjs] die verwendete JavaScript Bibliothek, um das HTML lokal zu erstellen. Sobald der Templatehandler die Methode `templatesLoaded` aufruft, liegen dem Framework alle Templates und Daten vor.

[6] Bei Adhocracy ist der Templatehandler eine eigene Datei (`/js/template_handler.js`) und wird zusätzlich zum Framework ausgeliefert.

Abbildung 3.5.: Daten lokal rendern



Die Methode erstellt nun das HTML, indem sie die externe Methode `Mustache.render` aufruft und ihr die JSON-Daten und alle Templates übergibt. `Mustache.render` liefert anschließend das fertige HTML zurück, das dann im Browser angezeigt werden kann.

3.3.3. Daten lokal speichern

Um die Webapplikation offline nutzen zu können, reicht es nicht aus, die Daten nur lokal zu rendern, es ist ebenso notwendig, sie auch lokal zu speichern. Dazu bräuchte man einen persistenten Speicher, der auch offline zugreifbar ist. Wie in Abschnitt 2.2 erläutert, gibt es mit dem Browsercache und Cookies derzeit allerdings keine Möglichkeit, um dies zu erreichen. Daher sollen im Folgenden drei Mechanismen vorgestellt werden, die es ermöglichen, Daten lokal zu speichern.

WebSQL

WebSQL[web10b] ist der Versuch, SQLite[sql] basierte Datenbanken in die Browserumgebung zu integrieren. Es soll ein JavaScript Interface bereitstellen, über das Daten via SQL abgefragt werden können. Das Interface bietet Methoden an, um Datenbanken und Transaktionen zu erstellen und SQL-Abfragen auszuführen. Über den genauen Speicher, den man als Entwickler zur Verfügung gestellt bekommt, ist nicht viel bekannt: in der Spezifikation heißt es nur, „dass Browser die Größe des Gesamtspeichers für Datenbanken begrenzen sollen“[web10b]. In der Regel liegt die maximale Größe für Datenbanken aber bei 5 MB.

Der große Vorteil bei SQL basiertem Speicher ist, dass man die Mächtigkeit der SQL Sprache erlangt und auf einer abstrakteren Ebene mit den Daten umgehen kann, ohne sich Gedanken darüber machen zu müssen, wie sie tatsächlich abgespeichert werden.

WebSQL wird allerdings vermutlich keine Zukunft haben, da das W3C die aktive Standardisierung gestoppt hat. Seitens des W3C heißt es, dass „die Spezifikation einen Stillstand erreicht hat“[web10b], da „alle Browserhersteller dieselbe Implementierung, SQLite, gewählt haben“[web10b]. Um eine Standardisierung voran zu treiben, „werden aber unabhängige Implementierungen benötigt“[web10b]. Desweiteren hatte WebSQL im Entstehungszeitraum der vorliegenden Arbeit mit Safari 3.1+, Chrome 10+ und Opera 10.5+[wsq12] eine sehr geringe Browserunterstützung.

Web Storage

Mit dem Web Storage[web11] wurde ein Featureset eingeführt, dass sich – anders als WebSQL – sehr schnell durchgesetzt hat und aktuell von Chrome 10+, Firefox 10+, Safari 4+, Opera 10.5+ und IE 10+[wst12] unterstützt wird. Der Speicher ist mit 5 MB genauso groß wie derjenige von WebSQL. Der große Unterschied zu WebSQL besteht allerdings darin, dass es sich bei Web Storage um einen reinen key/ value-Speicher handelt, wobei die Values nur vom Typ String sein dürfen. Dies ist auch der große Nachteil von Web Storage, da man sich Gedanken machen muss, *wie* die Daten am besten abgespeichert werden. Im Vergleich zu WebSQL ist die Arbeit mit den Daten demnach etwas komplizierter.

Der Web Storage teilt sich auf in den **sessionStorage** und den **localStorage**. Sie unterscheiden sich deutlich in der Dauer ihres Speicherns: sessionStorage ist, wie der Name schon sagt, nur für die aktuelle Browsersession gedacht und wird üblicherweise gelöscht, sobald der Browser geschlossen wird. Der localStorage dagegen wird nur auf ausdrücklichen Wunsch des Benutzers hin gelöscht. Wenn sie neugestartet werden, stellen einige Browser die Inhalte der sessionStorage wieder her. Da

man sich darauf aber nicht verlassen kann, empfiehlt sich die Verwendung von localStorage, wenn die Daten langfristig lokal gespeichert bleiben sollen.

Filesystem API

Die Filesystem API[fsa12a] ermöglicht es Webapplikationen, Dateien auf dem lokalen System des Benutzers zu erstellen und zu verwalten. Dabei kann kein beliebiger Speicherort gewählt werden. Man bekommt vielmehr vom Browser einen Ort zugewiesen, woraus quasi ein eigenes Dateisystem entsteht. Dieses ist vom echten lokalen Dateisystem getrennt, sodass eine Art Sandkasten entsteht, in dem die Webapplikation ihre Dateien schreiben und lesen kann.

Der große Vorteil der Filesystem API ist, dass man dem Browser mitteilen kann, wieviel Speicherplatz man für die Applikation benötigt. Theoretisch gibt es dabei keine Größenbeschränkung. Derzeit wird das Thema Speicherplatz lediglich in den Sicherheitsüberlegungen erwähnt. Dort heißt es, dass durch eine „Speicherplatzbeschränkung die Gefahr eines Denial-of-Service Angriffs, durch das Vollschreiben der Festplatte, abgemildert wird“[fsa12b].

Die Filesystem API ist die neueste der hier vorgestellten Technologien – darin liegt auch ihr großer Nachteil, denn aktuell wird sie lediglich von Google Chrome ab Version 20 implementiert.

Nach der Entscheidung für einen lokalen Speicher, ist zu überlegen, wie die Daten zu speichern sind: verwendet man WebSQL, während auf dem Server ebenfalls eine SQL Datenbank läuft, kann die Struktur eins zu eins in den Browser kopiert werden. Verwendet man hingegen Web Storage, gibt es diesen Vorteil nicht. Stattdessen muss man sich eine Speicherstruktur überlegen, die auf die Webapplikation zugeschnitten ist. Dabei muss berücksichtigt werden, ob öfter lesend oder schreibend auf die Daten zugegriffen wird – also ob das Lesen oder Schreiben schnell gehen soll.

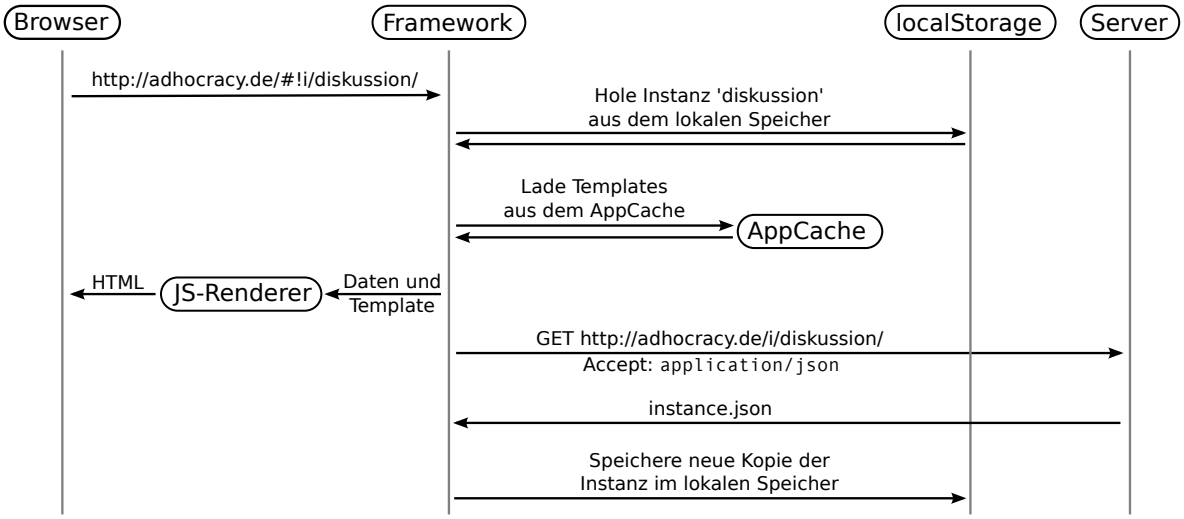
Sobald die Daten lokal gespeichert werden, kann das Framework in der Art erweitert werden, dass es diese auch nutzt: klickt ein Benutzer also auf einen Link, wartet das Framework nicht auf die neuen Daten vom Server, sondern benutzt die lokal Vorhandenen und lädt eine neue Kopie im Hintergrund. Dadurch fühlt sich das generelle Surfverhalten für den Benutzer schneller an als vorher.

Damit das Framework lokal nach den richtigen Daten suchen kann, muss es zunächst den Request verstehen: es muss um einen *URL Parser* erweitert werden, der die URL der Anfrage, ähnlich wie der Parser auf dem Server, in Controller, Action und Parameter[7] einteilt. Anhand dieser Informationen kann das Framework prüfen, ob es die Daten bereits lokal vorliegen hat. Abbildung 3.6 verdeutlicht,

[7] Im Client läuft natürlich keine MVC Applikation. Die Bezeichnungen sollen nur die Verwandheit zum Server zeigen.

wie das Framework versucht, die Daten aus dem lokalen Speicher zu laden und parallel eine neue Kopie von ihnen anfordert.

Abbildung 3.6.: Daten aus dem localStorage



Umsetzung in der Arbeit

Aufgrund der größten Browserunterstützung wurde für diese Arbeit der localStorage verwendet. Es gab viele Überlegungen, wie die Instanzen am besten lokal abgespeichert werden können. Eine erste Idee war, alle Instanzen in einem Request vom Server abzurufen und lokal in einem Array im localStorage abzulegen, sobald der Benutzer sich anmeldet. Das hätte den Vorteil, dass die ganze Applikation mit allen Daten direkt offline verfügbar ist.

Es wurde aber schnell klar, dass diese Vorgehensweise nicht sinnvoll ist, da die Datenmenge für einen Request zu groß ist, vor allem für Benutzer mit einer langsamen bzw. instabilen Internetverbindung, etwa auf mobilen Geräten. Sollte die Verbindung während der Übertragung abbrechen, hätte dies zu Folge, dass die Daten des unvollständigen Downloads unbrauchbar wären. Man sollte demnach nur Instanzen anfordern, die der Benutzer aktiv ausgewählt hat. Desweiteren ist es empfehlenswert, pro Instanz einen Eintrag im localStorage anzulegen.

Anschließend musste geklärt werden, wie die Instanz selbst – mit ihren eigenen Informationen, einer Liste mit allen Vorschlägen und einer Liste aller Kommentare pro Vorschlag – am besten zu speichern ist. Als problematisch bzw. umständlich stellte es sich heraus, dass man bei der Suche nach einem bestimmten Vorschlag die gesamte Liste durchgehen musste.

Ein Anzatz war es daher, die Vorschläge und Kommentare noch einmal seperat von den Instanzen zu speichern. Beim Empfang einer Instanz werden also alle Vorschläge und Kommentare extrahiert und als seperate Objekte gespeichert, was das Suchen und die Arbeit mit den Daten zwar erleichtert, gleichzeitig aber zur Folge hat, dass die Daten getrennt im Speicher liegen. Damit auf der Instanzseite auch alle Vorschläge angezeigt werden, benötigt Mustache die Daten der kompletten Instanz inklusive aller Vorschläge. Also müssten die Daten jedes Mal wieder zusammengebaut werden.

Aus diesen Überlegungen resultierte der zweite Ansatz, die Vorschläge und Kommentare nicht aus der Instanz zu extrahieren, sondern sie zu kopieren, sodass die originale Instanz immer noch vollständig im Speicher liegen bleibt. Dieses Vorgehen hat aber den Nachteil, dass die kompletten Daten zweimal gespeichert sind, was die Verwaltung und Pflege erschwert.

Letztendlich wurde pro Instanz ein Eintrag im localStorage angelegt, während die Vorschläge und Kommentare *nicht* seperat gespeichert wurden. Das bedeutet zwar einen Mehraufwand beim Suchen nach Vorschlägen oder Kommentaren, verringert den Verwaltungsaufwand aber erheblich.

Nachdem das Thema der Inszanzensicherung geklärt war, wurden Methoden geschrieben, um Vorschläge und Kommentare speichern und laden zu können. Desweiteren wurde ein URL Parser implementiert, der eine URL in Controller, Action und Parameter übersetzen kann und zum etwa die URL `http://adhocracy.de/#!/i/diskussion/proposal/mein_vorschlag` in den Controller `proposal` und die Action `show` umwandelt. Als Parameter werden die Instanz-URL (`/i/diskussion/`) und der Key des Vorschlags (`mein_vorschlag`) gesetzt. Anhand dieser Informationen kann das Framework also prüfen, ob es die Instanz *diskussion* lokal vorliegen hat und dann daraufhin den Vorschlag *mein_vorschlag* anzeigen.

Um die lokalen Daten nun auch zu verwenden, wurde die Methode `hashChangeEvent` (→ 3.1.5) in der Art erweitert, dass sie die Methode `renderLocalData` aufruft, bevor sie den Request an den Sever weiterleitet.

Die Methode `renderLocalData` ruft dann zunächst den URL-Parser auf und lädt die lokalen Daten passend zu den Informationen, die er bereitstellt. Anschließend lädt sie die zugehörigen Templates mit Hilfe des Templatehandler. Das Erstellen und Anzeigen der HTML-Seite erfolt dann wie zuvor erläutert (→ 3.3.2).

3.4. Synchronisation

Das Offline Fallback hat sich zum jetzigen Zeitpunkt der vorliegenden Arbeit so weit entwickelt, dass Ressourcen (Templates, CSS, Bilder), Code (JavaScript) und Daten lokal vorhanden sind und auch die

HTML-Ausgaben lokal erstellt werden. Die Webapplikation kann jetzt also offline verwendet werden, allerdings mit der Einschränkung, dass der Zugriff auf die Daten bis jetzt nur lesend erfolgen kann.

Auf das Ändern der lokalen Daten wird später in Abschnitt 3.4.3 noch genauer eingegangen. Zunächst soll es darum gehen, die Synchronisation in einer Richtung, nämlich vom Server zum Client, zu verbessern. Zuvor sei bemerkt, dass das Thema Synchronisation hier nur sehr rudimentär behandelt wurde, da es nicht Hauptthema dieser Arbeit war und ein viel zu weites Themengebiet ist, um es im hier beschränkten Rahmen ausführlich zu behandeln. Deutlich ausführlichere Untersuchungen bieten [esm79] und [sfs04].

3.4.1. Online oder Offline

Wichtig ist zunächst, dass das Offline Fallback eine Möglichkeit hat, zu entscheiden, ob der Benutzer online oder offline ist und ob der Server zu erreichen ist. Letzteres wird vom Framework bereits implizit durch die Verwendung von XMLHttpRequests (→ 3.1.4) geleistet. Ist der Server unerreichbar, schlägt der Request fehl und das Framework kann etwa ein Flag setzen und periodisch prüfen, ob der Server wieder erreichbar ist.

Um zu entscheiden, ob der Benutzer überhaupt online ist, wurde in HTML5 ein Flag eingeführt. Das **`window.navigator.onLine` Flag** zeigt an, ob eine der Netzwerkkarten mit einem Netzwerk verbunden ist. Das bedeutet zwar noch nicht, dass der Benutzer auch eine Verbindung zum Internet hat, dient aber immerhin als Anhaltspunkt, um den offline Fall zu prüfen: ist keine der Netzwerkkarten verbunden, kann der Benutzer auch den Server nicht erreichen. In diesem Fall braucht es das Framework gar nicht erst versuchen, seine Requests an der Server zu schicken und kann stattdessen nur mit den lokal vorliegenden Daten arbeiten. Das Framework kann zusätzlich die `window.online` und `window.offline` Events registrieren, um über den Status der Netzwerkkarten informiert zu werden.

Umsetzung in der Arbeit

In dieser Arbeit wurden beide oben erläuterten Mechanismem implementiert. Sobald ein Request fehl schlägt, wird die globale Variable `_status` auf `false` gesetzt, was anzeigt, dass der Server nicht erreichbar ist, und verhindert, dass weitere Requests an den Server geschickt werden. Gleichzeitig wird ein Timer gestartet, der periodisch prüft, ob der Server wieder erreichbar ist und im Erfolgsfall die `_status` Variable auf `true` setzt.

Neben der `_status` Variable wird das `window.navigator.onLine` Flag eingesetzt, um zu erkennen, ob der Benutzer generell online ist. Desweiteren werden während der Initialisierungsphase die beiden Events `window.online` und `window.offline` registriert, um über Änderung des Benutzerstatus informiert zu werden.

Sobald der Benutzer von einem Offline Zustand in einen Online Zustand wechselt, werden alle lokalen Daten zur Aktualisierung neu vom Server abgerufen.

3.4.2. Datenabruf optimieren

Derzeit schickt der Server bei jedem Request eine neue Version der Daten an den Browser, auch wenn es keine Veränderungen gegeben hat. Das verbraucht unnötig Bandbreite und belastet den Server, der die Daten jedes Mal aufbereiten muss. Daher soll es im nächsten Schritt darum gehen, wie entscheiden werden kann, ob das Versenden wirklich notwendig ist.

Checksumme

Checksummen sind weit verbreitet und werden immer dort eingesetzt, wo Daten schnell miteinander verglichen werden sollen. Dazu wird ein Hash, üblicherweise ein MD5 Hash, von diesen Daten berechnet.

Im Bezug auf das Offline Fallback berechnet der Server also die Checksumme eines Datensatzes und schickt beides zusammen an den Client. Ruft dieser zu einem späteren Zeitpunkt eine neue Version des Datensatzes ab, schickt er die Checksumme im Request mit – zum Beispiel im HTTP `If-None-Match` Header[FGM+99b]. Der Server berechnet nun auch wieder eine Checksumme und vergleicht sie mit der vom Client Gesendeten. Nur wenn sie sich unterscheiden, schickt der Server eine neue Version des Datensatzes.

Im Fall gleicher Daten spart man Traffic ein, da jetzt nur noch Kontrollnachrichten ausgetauscht werden. Allerdings hat diese Methode einen entscheidenden Nachteil, wenn es sich um hierarchische Daten handelt, wie zum Beispiel eine Instanz mit allen zugehörigen Vorschlägen und Kommentaren: da für die Berechnung der Checksumme jede Stufe der Hierarchie durchlaufen und diese Berechnung bei jedem Request durchgeführt werden muss, kann diese Methode einem erheblichen Mehraufwand für den Server verursachen.

Wenn auf die Daten der Webapplikation also öfter lesend als schreibend zugegriffen wird, ist es sinnvoller, die Checksumme bei den schreibenden Zugriffen zu berechnen und dann mit in der Datenbank abzuspeichern.

Zeitstempel

Einen alternativen Ansatz, um die Aktualität der Daten zu prüfen, bieten Zeitstempel: jedes Mal, wenn die Daten bearbeitet werden, aktualisiert sich auch der Zeitstempel. Wie bei den Checksummen schickt der Server den aktuellen Zeitstempel eines Datensatzes mit. Ruft der Client die Daten dann erneut ab, kann der Server anhand des mitgeschickten Zeitstempels entscheiden, ob er eine neue Version der Daten schicken muss.

Auch hier ergibt sich bei hierarchisch aufgebauten Daten eine Besonderheit: damit der Server nicht bei jedem Request alle Hierarchiestufen durchlaufen und den aktuellsten Zeitstempel suchen muss, ändert der Server bei Bearbeitungen eines Datensatzes den Zeitstempel des übergeordneten Datensatzes. Wenn zum Beispiel an einem Kommentar etwas geändert wird, muss auch der Zeitstempel des Vorschlags und der Instanz aktualisiert werden.

Zeitstempel haben gegenüber Checksummen den Nachteil, dass identische Daten irrtümlich als neu erkennt werden, wenn an den Daten etwas geändert und später rückgängig gemacht worden ist. Im Fall der Checksumme würde man erkennen, dass die Daten nachwievor die selben sind. Werden Zeitstempel verwendet, sieht der Server hingegen nur, dass sich die Daten aktualisiert haben, und schickt sie an den Client.

Versionierung

Eine dritte Möglichkeit wäre die Verwendung einer vollständigen Versionierung der Daten. Das bedeutet, der Client bekommt zunächst eine initiale Version der Daten und ab diesem Zeitpunkt nur noch die Änderungen. Dies ist das beste System, da es den wenigesten Datenverkehr versursacht und reversibel ist. Bei Checksummen und Zeitstempeln hingegen kann man zwar entscheiden, ob sich etwas geändert hat, dafür sind die Änderungen nicht umzukehren, sobald sie einmal akzeptiert wurden: alte Zustände können nicht wieder hergestellt werden. Darüber hinaus kann man bei der Verwendung einer Versionierung auch stets nachvollziehen, was genau sich verändert hat.

Auf der anderen Seite muss das Versionierungssystem tief in der Webapplikation verankert sein. Es im Nachhinein einzubauen ist nur schwer machbar, da die gesamte Daten- und Datenbankstruktur darauf abgestimmt sein muss.

Unabhängig davon, welche der drei Methoden man wählt, schickt der Client immernoch bei jeder Interaktion des Benutzers einen Request an den Server. Trotz der Reduktion der übertragenen Datenmenge werden immernoch viele unnötige Requests an den Server geschickt. Um das zu ändern, kann man das Framework so erweitern, dass es Polling einsetzt und nur noch periodisch nach neuen Daten fragt. Das Framework arbeitet demnach nur noch auf lokalen Daten und ruft im Hintergrund persiodisch neue Versionen der Daten ab.

Umsetzung in der Arbeit

In dieser Arbeit wurden Zeitstempel verwendet, um auf dem Server zu entscheiden, ob die Daten an den Client geschickt werden müssen. Dabei wurde je ein Zeitstempel pro Instanz, Vorschlag und Kommentar angelegt, der anzeigt, wann sich an diesen Daten direkt etwas geändert hat. Zusätzlich erhielt die Instanz noch einen Zeitstempel – das Maximum der einzelnen Zeitstempel –, der bezeichnet, wann innerhalb der Instanz die letzte Änderungen stattgefunden hat.

Alle relevanten Controller wurden so erweitert, dass sie anhand dieses Zeitstempels entscheiden, ob die Daten geschickt werden sollen. Listing 3.9 zeigt einen Ausschnitt aus dem Instanz-Controller und verdeutlicht, auf welche Weise der Server entscheidet, ob seine Daten aktueller sind als die des Clients. Es wird geprüft, ob der Client einen Zeitstempel mitgeschickt hat und ob dieser größer oder gleich dem aktuellen Zeitstempel der Instanz ist. Nur wenn die Instanz einen jüngeren Zeitstempel aufweist, werden neue Daten geschickt.

Listing 3.9: Instanz-Controller: Aktualität der Daten prüfen

```
1  def show(self, id, format='html'):
2  (...)
3      if request.headers.has_key('if-modified-since'):
4          client_timestamp =
5              h.datetime_from_string(request.headers.get('if-modified-since'))
6
7          if client_timestamp >= c.page_instance.timestamp:
8              return render_real_json({ 'info': 'instance up-to-date' })
9
10     c.mustache_values = c.page_instance.to_mustache_dict(with_proposals=True,
11                                                          with_comments=True)
12     return render_real_json(c.mustache_values)
13 (...)
```

Um Requests an den Server einzusparen, wurde das Framework so erweitert, dass es die Daten nur noch periodisch alle fünf Minuten aktualisiert. Neben dem Polling werden nur dann noch neue Daten

abgerufen, wenn das Framework initialisiert wird oder wenn der Benutzer vom Offline- in den Online-Zustand wechselt.

3.4.3. Änderungen an lokalen Daten

Sobald das Framework neue Daten vom Server erhält, werden die alten, lokalen Daten überschrieben. Solange der Benutzer nur lesend auf die Daten zugreifen kann, ist dies auch nicht weiter schlimm. Das Design des Offline Fallback sieht aber auch Änderungen an den lokalen Daten vor. Ändern bedeutet dabei, Einträge erstellen und bearbeiten zu können, also die typischen Aufgaben des Servers in den Client zu portieren und dem Offline Fallback damit eine Menge Logik und Wissen über die Daten zu vermitteln.

Desweiteren muss das Framework alle relevanten Formulare manipulieren, ähnlich wie schon vorher die Links für die lokalen Ressourcen angepasst wurden (→ 3.1.4). Schickt ein Benutzer ein Formular ab, zum Beispiel zum Erstellen eines Kommentars, muss das Framework diesen Schritt erkennen, den Request an den Server unterdrücken und ihn lokal verarbeiten.

Umsetzung in der Arbeit

In dieser Arbeit wurden Methoden implementiert, die das Erstellen und Bearbeiten von Vorschlägen und Kommentaren ermöglichen. Dabei ist der einfachere Fall das Bearbeiten von Daten, da dabei nur einzelne Teile, wie zum Beispiel die Beschreibung des Vorschlags, in einem bestehenden Datensatz angepasst werden.

Einen neuen Datensatz zu erstellen ist schon deutlich komplexer, da zusätzlich zum Vorschlag oder Kommentar noch ein Bewertungsobjekt erstellt werden muss. Ein weiteres großes Problem ist die ID-Vergabe: sie ist nicht deterministisch und kann daher nur vom Server übernommen werden. Deshalb wird lokal im Framework nur eine vorläufige ID erstellt, die dann durch die Korrekte vom Server ersetzt wird (→ 3.4.4).

3.4.4. Änderungen zurücksynchronisieren

Nun kann der Benutzer die Applikation zwar komplett lokal nutzen, allerdings werden lokal vorgenommen Änderungen beim nächsten Update der Daten vom Server überschrieben. Daher soll es nun darum gehen, dem Server diese Änderungen mitzuteilen, damit sie in den Datenbestand der Webapplikation einfließen.

Wenn der Benutzer eine Verbindung zum Server hat, ist es am sinnvollsten, diesem die Änderungen sofort mitzuteilen. Dazu kann das Framework den zuvor unterdrückten Request verwenden und an den Server schicken. Der Server erkennt dann anhand des HTTP `Accept` Headers, dass der Request vom Offline Fallback kommt (→ 3.2.3) und kann mit einer einfachen `success` oder `error` Nachricht antworten und muss keine HTML-Seite ausliefern.

Ist der Benutzer offline oder hat keine Verbindung zum Server, muss das Framework sich merken, dass eine Änderung an den Daten vorgenommen wurde. Dazu kann es den Request in eine *Ausgans-Queue* speichern. Sobald dann wieder eine Verbindung besteht, ist es wichtig, zunächst diese Queue abzuarbeiten und alle Requests an den Server zu schicken, bevor neue Daten geladen werden.

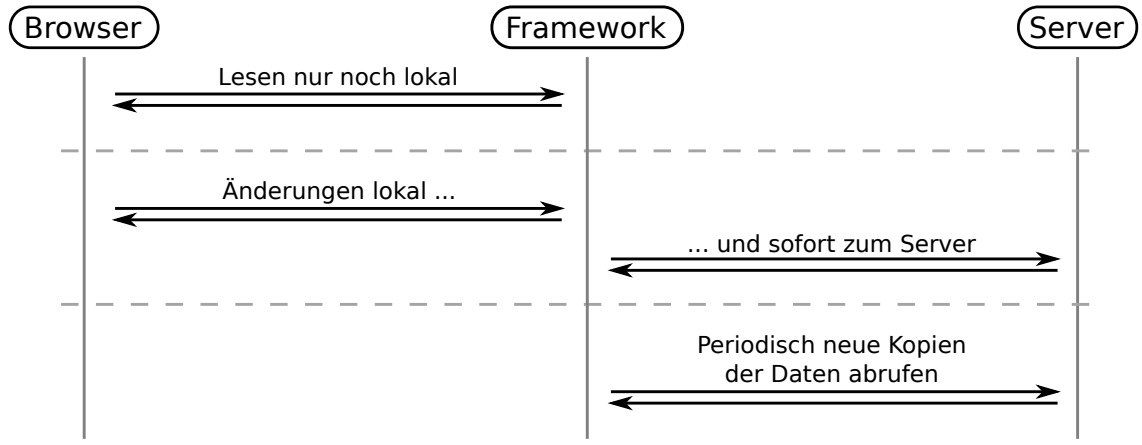
Umsetzung in der Arbeit

In dieser Arbeit wurde die Synchronisation, wie zuvor erläutert, implementiert. Dazu prüft das Framework vor dem Senden eines Requests die beiden Flags `_status` und `window.navigator.onLine`. Ist eine von beiden `false`, wird der Request in die Queue geschrieben. Besteht wieder eine Verbindung, werden diese ausstehenden Requests an den Server geschickt und erst wenn alle abgearbeitet wurden, werden die lokalen Daten aktualisiert.

Wie zuvor erläutert, kann lokal nur temporäre IDs an die Daten vergeben werden: wenn also ein Request zum Erstellen eines Kommentars oder Vorschlags an den Server geschickt wird, antwortet dieser nicht nur mit einem `success` oder `error`, sondern schickt auch die richtige ID an den Client, der sie dann in die lokalen Daten einpflegt.

Abbildung 3.7 zeigt einen vereinfachten Ablauf der Synchronisation beim Offline Fallback von Adhocracy.

Abbildung 3.7.: Synchronisation bei Adhocracy



Kapitel 4.

Evaluation

Neben dem Vorteil, die Webapplikation offline verwenden können, scheint sie zudem auch schneller zu sein, selbst wenn eine Verbindung zum Server besteht. Um dies zu überprüfen, werden im Folgenden mehrere durchgeführte Tests vorgestellt. Da es zu aufwändig ist, diese manuell vorzunehmen, wurde nach einem Testframework gesucht, das automatisierte Tests von Webseiten aus Sicht einer Benutzers erlaubt. Eine der Hauptkriterien an ein solches Testframework ist, dass es die direkte Interaktion mit der Webapplikation ermöglicht, zum Beispiel Links anklicken oder Eingaben in Textfeldern.

4.1. Testframework

Selenium

Da sich Selenium bereits im Vorfeld dieser Arbeit bei Adhocracy bewährt hat, wurde es auch für die Tests des Offline Fallback verwendet. Selenium bietet die Möglichkeit, Webbrowser zu automatisieren – also definierte Aktionen auszuführen, beispielsweise „Klick auf einen Link“. Dazu müssen zunächst Treiber heruntergeladen werden, die als Schnittstelle zwischen dem Desktopbrowser und Selenium dienen und unter anderem für Chrome[cdr] und Firefox[ffd] verfügbar sind.

Anschließend können Tests geschrieben werden, in denen die Selenium Bibliothek verwendet wird, um Befehle an den Desktopbrowser zu schicken. Die Bibliothek gibt es für viele Programmiersprachen, unter anderem für JavaScript, PHP, Ruby und Python – in dieser Arbeit wurde letzteres verwendet.

Der Vorteil von Selenium ist, dass es das Schreiben von high-level Befehlen in der Art von „Schreibe ‚Foo‘ in das Textfeld mit der ID X und schicke das dazugehörige Formular ab“ ermöglicht und sich

somit sehr gut für komplexe, automatisierte Tests eignet, die den zuvor aufgezählten Anforderungen entsprechen (→ 4).

4.2. Setup

Die Tests liefen alle lokal auf einem MacBook mit 2,3 GHz i5 Dualcore Prozessor, 8 GB DDR3 Ram und 128 GB SSD. Der eigentliche Adhocracy Server lief in einer virtuellen Maschine mit 2 Prozessorkernen, 2GB RAM und 8 GB SSD.

Aufgrund dieses Setups gilt für alle Tests folgendes: die Datenübertragungsrate ist nahezu unendlich und die Latenz der Requests beträgt fast Null. Daher wurden einige Tests einmal ohne und einmal mit einer künstlichen Latenz von 200 ms durchgeführt. Die künstliche Latenz wurde auf Betriebssystemebene mit Hilfe von `ipfw`[1] für alle Requests an den Port 4280, auf dem auch Adhocracy läuft, erzeugt.

4.3. Tests

4.3.1. Initialisieren der Webapplikation

Im ersten Test wurde untersucht, wie lange es dauert, bis die Webapplikation initialisiert ist und der Benutzer anfangen kann, sie zu verwenden.

Normale Webapplikationen, also auch die bisherige Version von Adhocracy, sind im Wesentlichen nutzbar, sobald das ganze HTML, CSS und alle Bilder geladen und angezeigt werden. Beim Offline Fallback ist dies anders, da es initial noch „installiert“ werden muss. Das bedeutet, Appcache und Framework müssen initialisiert werden: alle Ressourcen laden und speichern, relevante Links umschreiben und localStorage initialisieren.

Um zu testen, wann eine Seite komplett fertig geladen und initialisiert ist, können verschiedene JavaScript Events abgefragt werden. **DOM-Ready** etwa zeigt an, dass eine HTML-Seite komplett geparst wurde. Zu diesem Zeitpunkt kann das HTML zwar angezeigt werden, würde aber unter Umständen nicht wie gewollt aussehen, da eventuell noch nicht alle verlinkten Ressourcen (CSS, Bilder, ...) geladen sind. Sobald diese fertig geladen sind, und das HTML gerendert wird, löst der Browser das

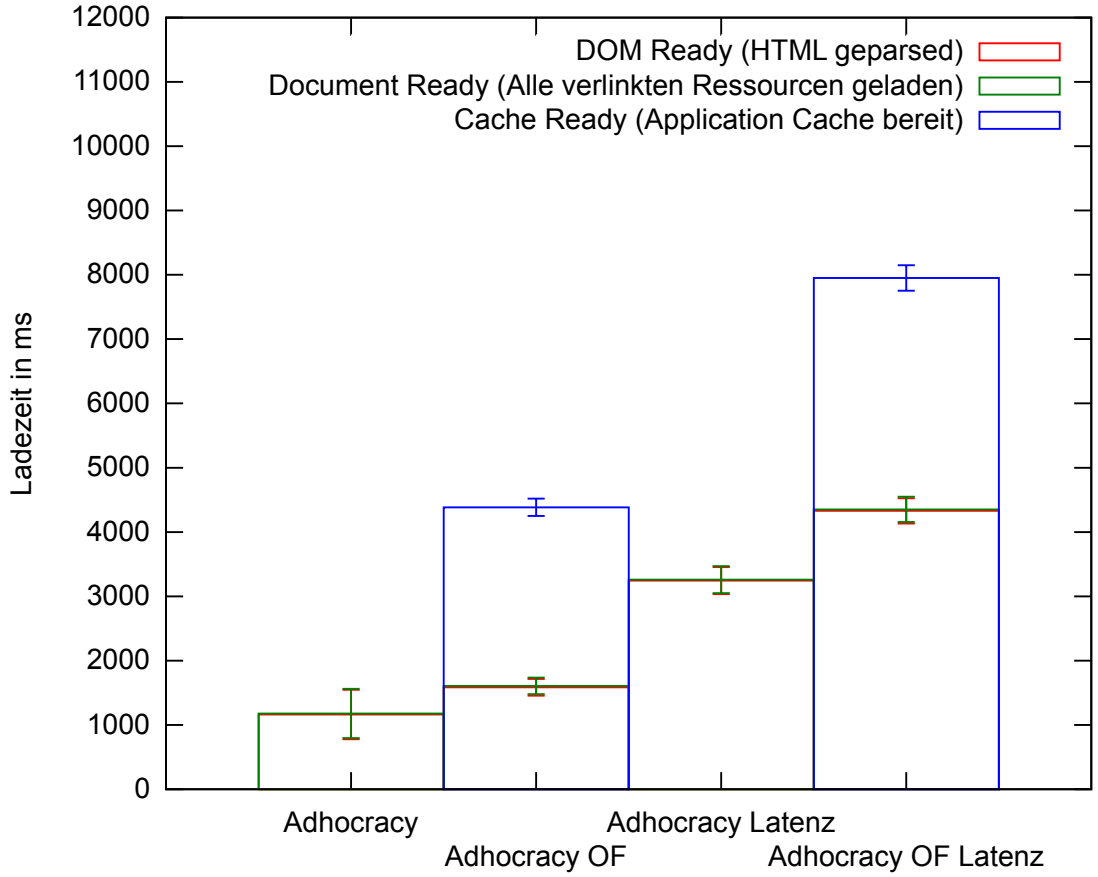
[1] Künstliche Latenz von 200ms: `sudo ipfw pipe 1 config delay 200ms && sudo ipfw add pipe 1 dst-port 4280`

Document-Ready Event aus. Für das Offline Fallback ist zusätzlich noch das **Cache-Ready** Event des Applicationcache wichtig, welches anzeigt, dass alle Ressourcen lokal gespeichert wurden und der Cache einsatzbereit ist.

Diese JavaScript Events werden allerdings nur im Browser ausgelöst und können in Selenium nicht unmittelbar erkannt werden. Daher wird hier ein kleiner Umweg gegangen, indem ein Stück JavaScript geschrieben wurde, das auf die Events reagiert, jeweils ein `div` Element mit der ID des Events erstellt und in den Browser einbettet, zum Beispiel `<div id="docready"></div>`, sobald das Document-Ready Event ausgelöst wird. In Selenium kann nun die Zeit gemessen werden, die bis zum Erscheinen dieser Elemente vergangen ist.

Abbildung 4.1 zeigt die Ergebnisse dieses Tests: auf der X-Achse sind die verschiedenen Versionen von Adhocracy aufgetragen, auf der Y-Achse die absolut benötigte Zeit für die einzelnen Events. Bei allen Varianten ist zu erkennen, dass das DOM-Ready und das Document-Ready Event nahezu gleich sind. Das hängt damit zusammen, dass auf der Startseite nicht viele Ressourcen (CSS, Bilder usw.) eingebunden werden.

Abbildung 4.1.: Dauer bis die Webapplikation initialisiert ist (10 Durchgänge)



Es fällt auf, dass das Initialisieren des Offline Fallback insgesamt deutlich länger braucht, was hauptsächlich auf das Cache-Ready Event zurückzuführen ist – was aber normal ist, da alle Ressourcen geladen und gecacht werden müssen. Allerdings erkennt man auch, dass das DOM-Ready Event ca. 500 ms später gefeuert wird.

Das Offline Fallback muss zwar zusätzlich zum normalen JavaScript Code noch das Framework laden, da aber die Übertragungsrate beliebig groß ist, kann dieser Unterschied nicht darin begründet sein. Ein weiterer Test ergab, dass nicht das JavaScirpt Framework, sondern das Cache Manifest – genauer gesagt sein Anfordern – zu der deutlich längeren Wartezeit führt.

Bei Adhocracy ist das Cache Manifest, wie erläutert, keine Datei, sondern wird beim jedem Seitenaufruf erstellt (→ 3.1.2). Darin liegt der Grund für die längere Wartezeit, bis das DOM-Ready Event ausgelöst wird – das Erstellen und Ausliefern des reinen Manifests benötigt etwa 400 ms.

Wartezeit verkürzen

Das Initialisieren des Application Cache wird vom Browser gesteuert, weshalb man keinen Einfluss darauf hat. Die halbe Sekunde, bis das DOM-Ready Event gefeuert wird, kann jedoch verkürzt werden.

Eine Möglichkeit dazu ist, das Erstellen des Cache Manifests zu verbessern. Die im Rahmen des Prototypen entwickelte Methode bei Adhocracy weist einige Optimierungsmöglichkeiten auf. Zum Beispiel könnte das Erstellen der URL der JavaScript Ressource[2] verbessert werden, damit nicht alle JavaScripte durchlaufen werden müssen.

Desweiteren kann das Initialisieren des kompletten Offline Fallback, ähnlich wie bei der Verwendung des Application Cache (→ 3.1.6), in ein IFrame verlagert werden. Dies hätte den Vorteil, dass man initial nur noch ein Stück JavaScript-Code ausliefern müsste, der das IFrame erstellt und die Initialisierung des Offline Fallback in Gang setzt. Das eigentliche Initialisieren des Framworks würde dann im Hintergrund ablaufen, sodass der Benutzer nicht mehr darauf warten müsste.

4.3.2. Laden einer Instanzseite

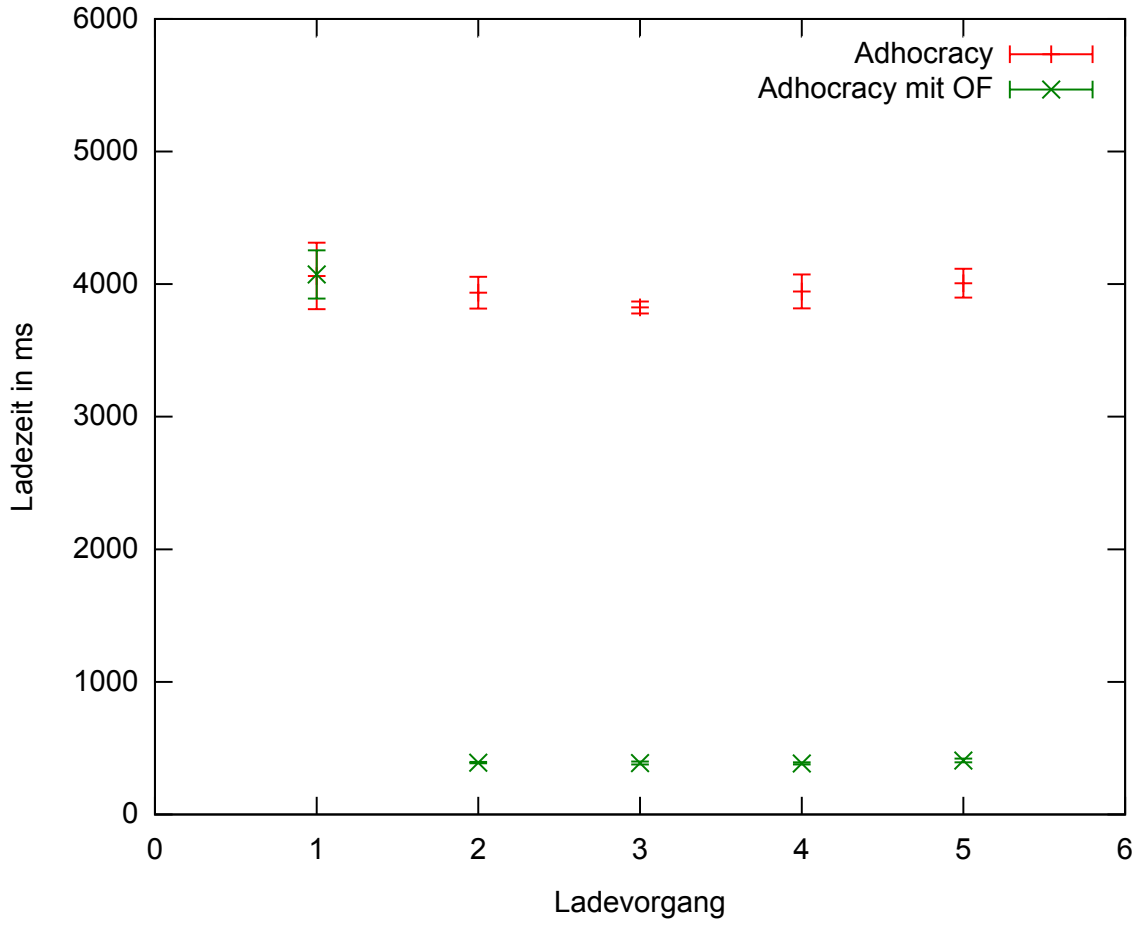
Der zweite Test soll zeigen, wie lange es dauert, eine Instanzseite mit unterschiedlich vielen Vorschlägen zu laden. Der Test simuliert zum einen das Neuladen der Seite mit F5 und zum anderen das

[2] Bei Adhocracy werden die JavaScripte nicht getrennt voneinander, sondern kombiniert in einer großen Ressource ausgeliefert. Die URL dieser großen Ressource wird zu Caching-Zwecken mit dem MD5 Hash des gesamten JavaScripts versehen.

Navigieren auf einer Instanzseite über das Menü. Gemessen wird bei diesem Test die Zeit vom Eingeben der URL bzw. Klicken des Links bis zu dem Zeitpunkt, an dem die Seite vollständig angezeigt wird.

Es wurden Instanzen mit 25, 50, 100, 150 und 200 Vorschlägen jeweils zehnmal neugeladen. Abbildung 4.2 zeigt einen solchen Durchlauf für eine Instanz mit 100 Vorschlägen. Beim normalen Adhocracy ist die durchschnittliche Ladezeit mit etwa 4 Sekunden, wie ertwartet, sehr gleichmäßig. Beim Offline Fallback fällt allerdings eine Besonderheit auf: alle Ladevorgänge sind mit etwa 500 ms deutlich schneller – nur der Erste benötigt genauso lange wie beim normalen Adhocracy. Dies kann damit erklärt werden, dass die Daten noch nicht lokal vorhanden sind und das Framework sie im ersten Request vom Server laden und lokal speichern muss. In allen folgenden Aufrufen werden dann diese lokalen Daten für die Anzeige verwendet.

Abbildung 4.2.: Ladezeiten bei einer Instanz mit 100 Vorschlägen (10 Durchgänge)

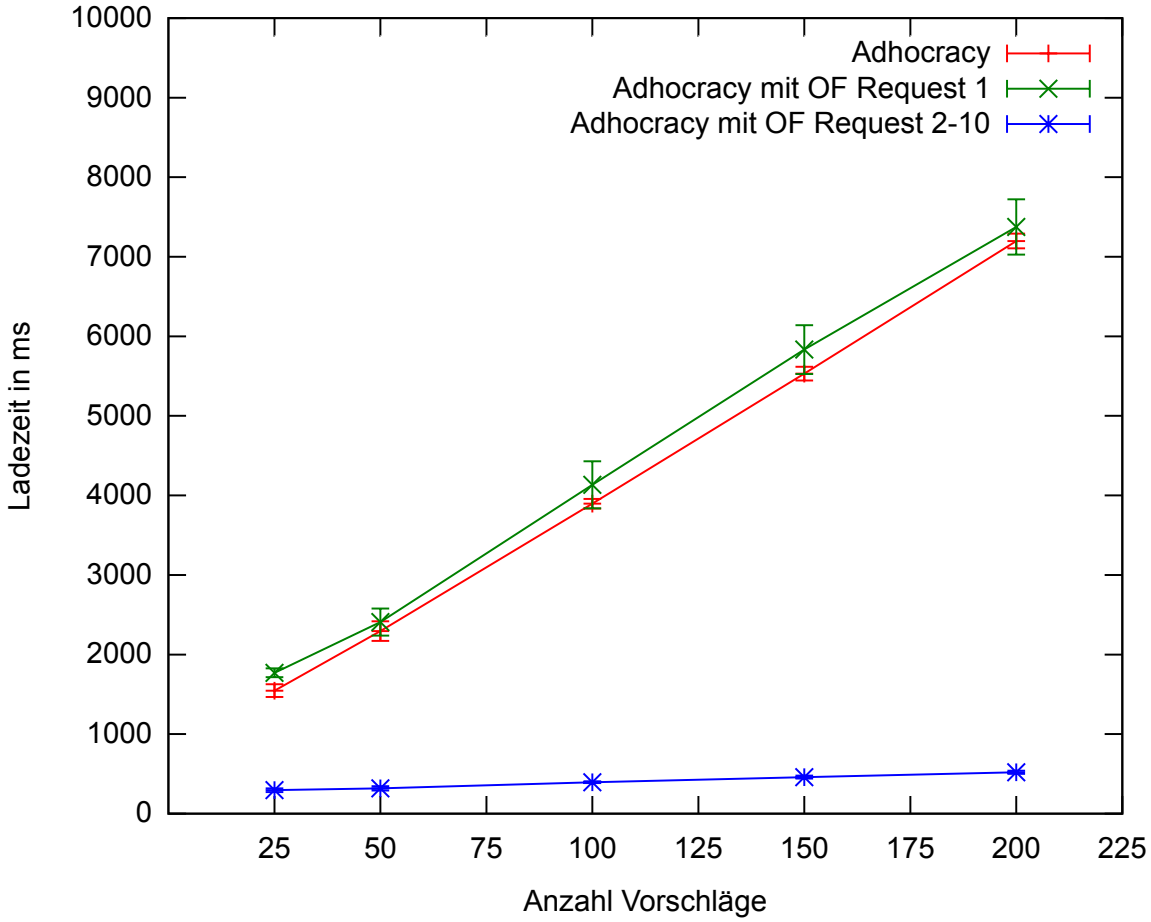


Abbilung 4.3 zeigt die Ladezeit einer Instanzseite in Abhängigkeit von der Anzahl der Vorschläge an. Dabei wurden die oben erwähnten langen, ersten Requests(→ 4.3.2) des Offline Fallback seperat auf-

getragen, da sich anderenfalls eine deutlich abweichende Ladezeiten-Verteilung ergeben hätte. Beim normalen Adhocracy und den ersten Requests des Offline Fallback fällt auf, dass die Ladezeit linear zur Anzahl der Vorschläge steigt und somit in O(n) liegt. Die Folgerequests steigen zwar auch linear, verlaufen aber deutlich flacher.

Es fällt desweiteren auf, dass die Ladezeit des Offline Fallback – bis auf die Instanz mit 100 Vorschlägen – immer größer als die Ladezeit des HTML beim normalen Adhocracy ist. Dies hat im Wesentlichen zwei Gründe: die Datengröße (→ 4.3.3) und die Zeit, die der Server benötigt, um die JSON Darstellung bzw. das HTML zu erzeugen.

Abbildung 4.3.: Ladezeiten einer Instanz bei verschieden vielen Vorschlägen (10 Durchgänge)



Für den Test wurde der Offline Speicher des Browsers nach jedem Durchlauf gelöscht, weshalb diese langen Requests in jedem Durchgang auftreten. Verwendet ein Benutzer die Webapplikation mit dem Offline Fallback, wird er seinen Offline Speicher nicht andauernd löschen – damit tritt dieser initiale, lange Request nur beim ersten Mal auf, wenn die Webapplikation gestartet wird. Sobald die Instanz

dann einmal lokal vorliegt, wird sie im Hintergrund synchroniert und der Benutzer bekommt von der langen Ladezeit nichts mehr mit.

4.3.3. Größe einer Instanz

Wie in der Arbeit erläutert, speichert das Offline Fallback alle Ressourcen und Daten für die Offline-Verwendung lokal im Browser ab. Dieser lokale Speicher ist bei vielen Browsern aber auf 5 MB begrenzt. An dieser Stelle drängt sich natürlich die Frage auf, wie lange man das Offline Fallback verwenden kann, bis diese Grenze erreicht ist. Die Ressourcen (Bilder, CSS, Templates und JavaScript) für das Offline Fallback nehmen aktuell ca. 500 KB des lokalen Speichers ein. Meta- und Statusinformationen verbrauchen davon ca. 200 KB, bleiben also etwa 4,3 MB für die Daten, sprich die Instanzen in JSON.

Die Größe einer Instanz hängt natürlich von vielen Faktoren ab, unter anderem von der Länge ihres Beschreibungstextes sowie der Anzahl und Länge ihrer Vorschläge und Kommentare. In einem weiteren Test wurde nun die Größe einer Instanz in Abhängigkeit von der Anzahl ihrer Vorschläge untersucht.

Beim normalen Adhocracy wird die Größe der HTML-Seite gemessen, die vom Server kommt – beim Offline Fallback die Größe der JSON-Repräsentation der Instanz. Abbildung 4.4 stellt die Größen der Instanzen mit steigender Vorschlagsanzahl. Man erkennt, dass die Größe der JSON-Daten immer höher ist als die der HTML-Seiten beim normalen Adhocracy und, dass sie zudem schneller ansteigt.

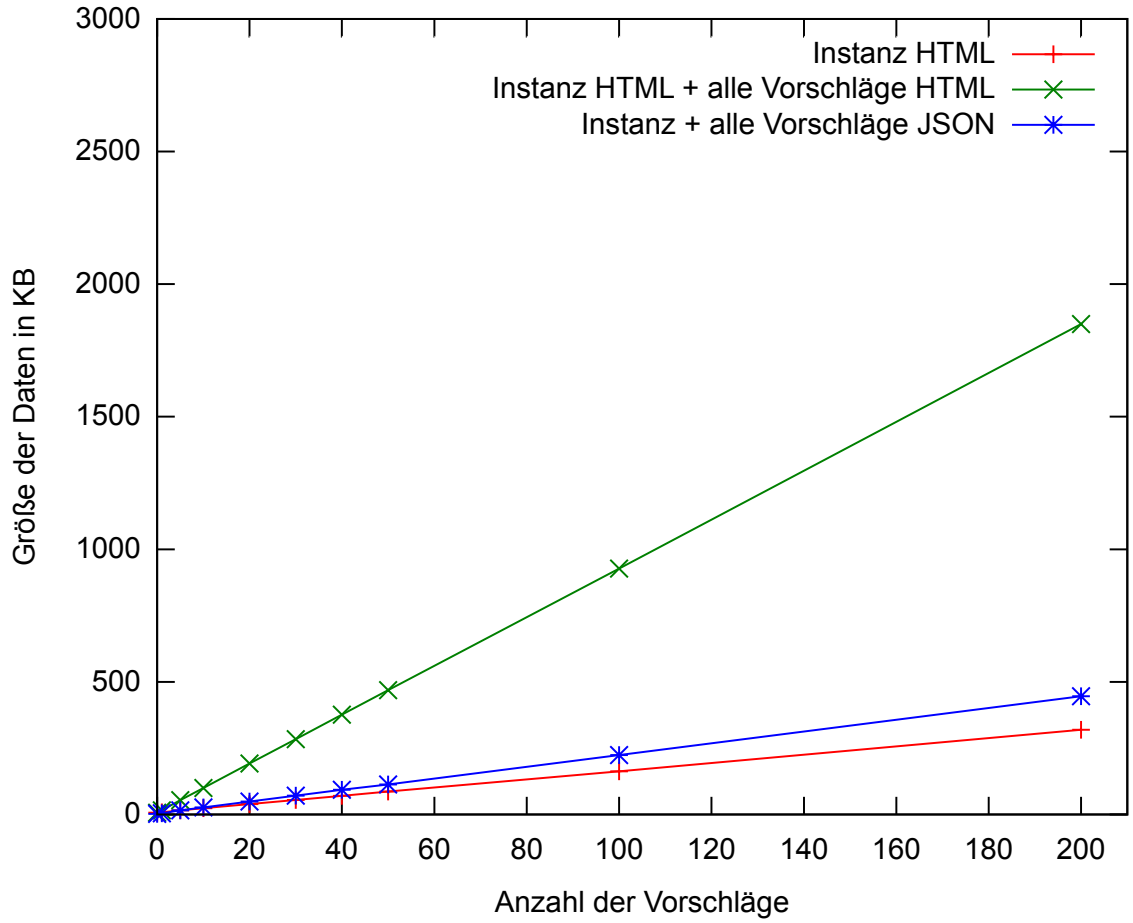
Dieser Unterschied hängt damit zusammen, dass in den JSON-Daten die Informationen der gesamten Instanz – also auch alle Vorschläge und Kommentare – enthalten sind. Beim normalen Adhocracy ist dies hingegen nicht der Fall: hier kann man anhand der vorliegenden Instanz HTML-Seite nicht die Vorschläge darstellen, sondern muss dazu wieder ein neues HTML Dokument vom Server abrufen.

Um also die Größenwerte zu vergleichen, muss zur Größe der Instanz HTML-Seite noch die Größe aller Vorschlag HTML-Seiten addiert werden. In diesem Test sind alle Vorschlagseiten mit je 7,48 KB gleich groß. Wie in Abbildung 4.4 zu sehen ist, müssen beim normalen Adhocracy demnach deutlich mehr Daten übertragen werden, wenn sich der Benutzer alle Vorschläge ansieht.

4.3.4. Erstellen eines Vorschlags

Der nächste Test widmet sich einer sehr häufigen Aktion bei Adhocracy, dem Erstellen eines Vorschlags. Konkret wurden dabei 50 Vorschläge erstellt und die Zeiten zwischen dem Abschicken des

Abbildung 4.4.: Vergleich der Datengrößen



Formulars und dem vollständigen Anzeigen des neu erstellten Vorschlags gemessen.

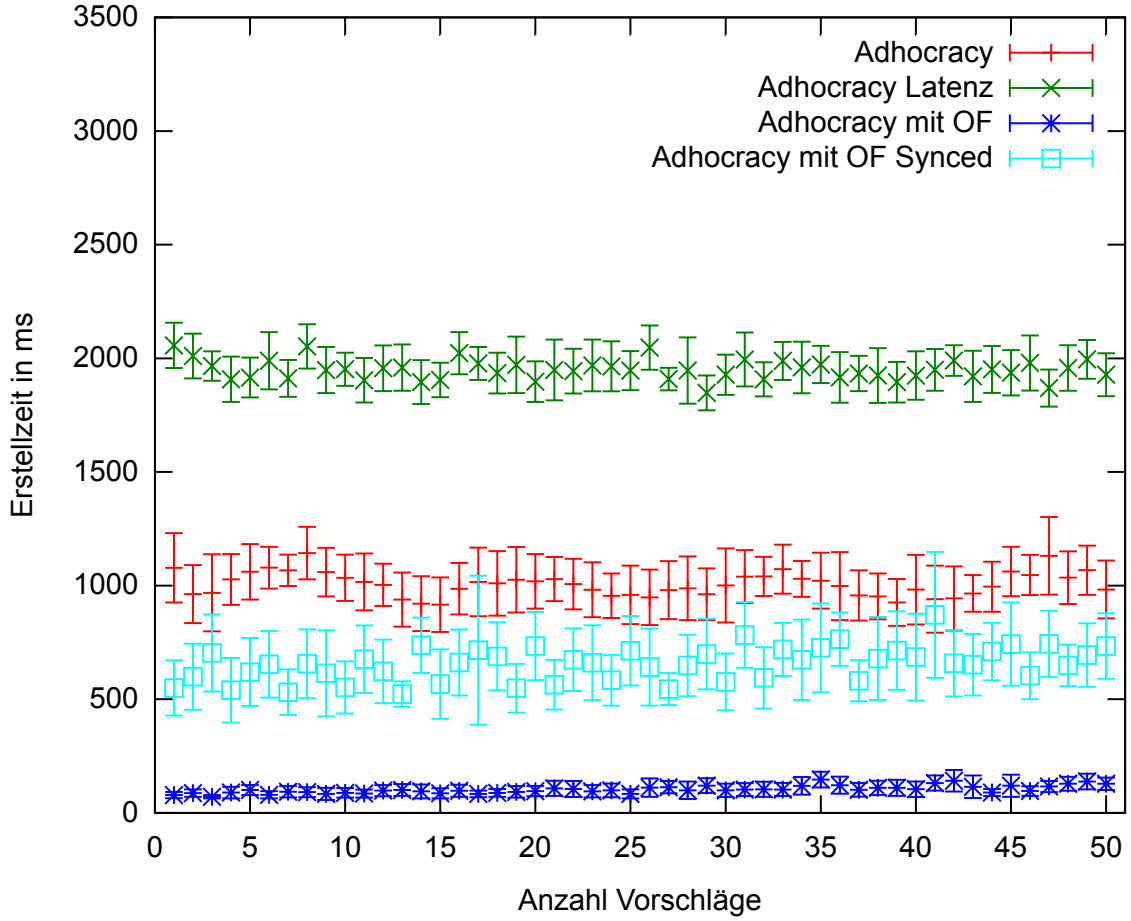
Auch bei diesem Test ergibt sich erneut eine Besonderheit für das Offline Fallback, denn hier wird der Vorschlag zunächst lokal erstellt und anschließend mit dem Server synchronisiert (→ 3.4.4). Dies läuft im Hintergrund ab, sodass der Benutzer davon nichts mitbekommt und auch nicht warten muss, bis der Vorgang abgeschlossen ist.

Es ergeben sich demnach zwei Szenarien, die für das Offline Fallback getestet wurden: einmal die Zeit, die das rein lokale Erstellen des Vorschlags benötigt, und zum anderen die Zeit, die für das lokale Erstellen in Verbindung mit der Synchronisation zum Server benötigt wird.

Abbildung 4.5 zeigt die Ergebnisse des Tests. Die serverseitig benötigte Zeit bei Adhocracy beträgt konstant etwa eine Sekunde. Das Offline Fallback liegt in diesem Test erwartungsgemäß deutlich darunter. Selbst wenn man im Test wartet, bis die Synchronisation zum Server abgeschlossen ist, liegt

das Offline Fallback in fast allen Fällen unter der Zeit von Adhocracy. Der Grund dafür ist, dass der Server im Fall der Synchronisation mit dem Offline Fallback kein HTML mehr erzeugen muss, sondern über JSON mit dem Client kommuniziert (→ 3.4.4).

Abbildung 4.5.: Dauer zum Erstellen eines Vorschlags (10 Durchgänge)



4.3.5. Generelles Surfverhalten

Der letzte Test simuliert das normale Surfverhalten von Benutzern, indem eine definierte Folge von Aktionen ausgeführt und die Zeit zwischen diesen gemessen wurde. Konkret wurde die folgende Kette von Aktionen mehrmals durchgeführt:

1. Webapplikation über die Startseite aufrufen.

2. Instanzseite `/i/meta/` aufrufen.

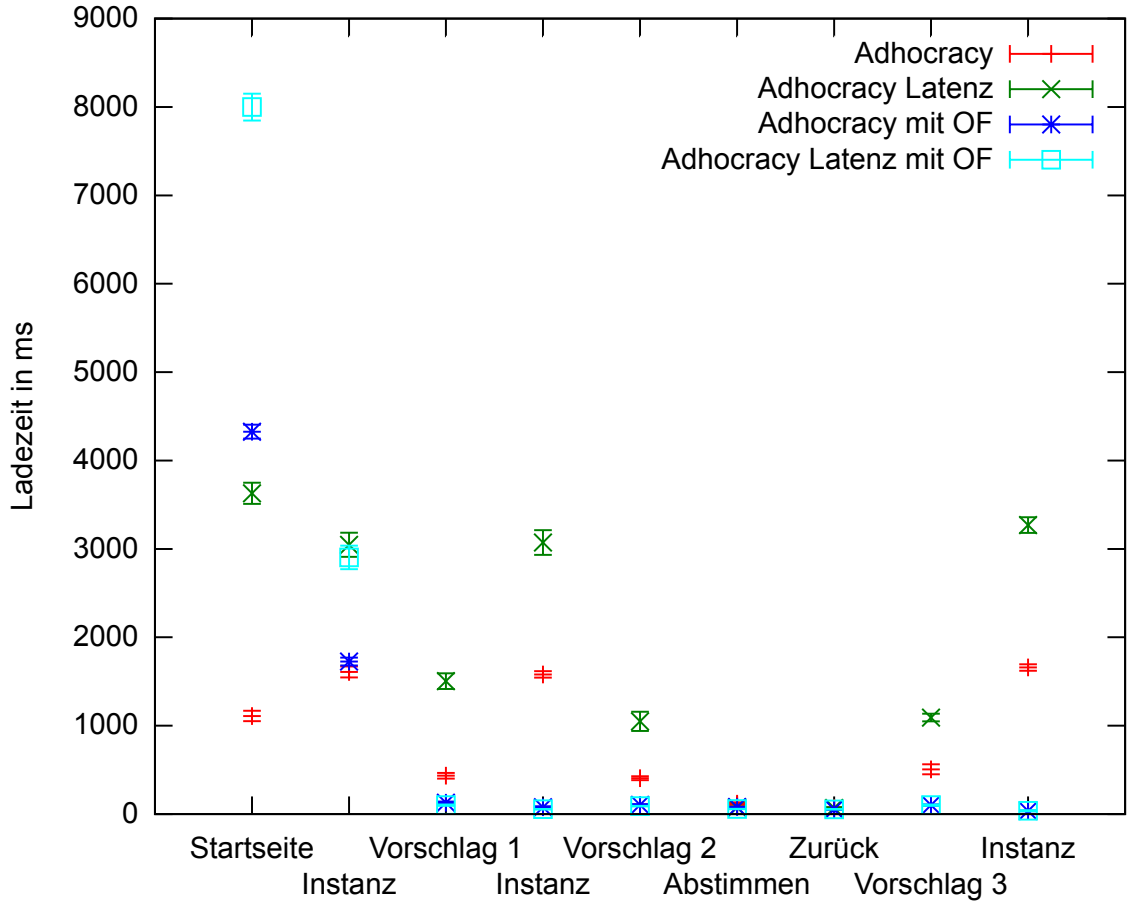
3. Ersten Vorschlag auswählen.
4. Instanzseite `/i/meta/` erneut über das Menü öffnen.
5. Zweiten Vorschlag anzeigen.
6. Vorschlag 2 auf- bzw. abwerten.
7. Über den Browser-Back-Button zurückkehren zur Instanz.
8. Dritten Vorschlag aufrufen.
9. Instanzseite `/i/meta/` über das Menü auswählen.

Die Ergebnisse sind in Abbildung 4.6 zu sehen. Der erste Schritt – Startseite aufrufen – dauert beim Offline Fallback deutlich länger. Dies liegt daran, dass hier die Zeit gemessen wurde, bis der Application Cache fertig geladen ist (→ 4.3.1).

Der zweite Schritt – das erste Aufrufen der Instanz – benötigt bei den beiden Versionen jeweils mit und ohne Latenz gleich lang. Dies lässt sich mit dem Test „Laden einer Instanzseite“ (→ 4.3.2) erklären.

Die Schritte Abstimmen (Schritt 6) und Zurück (Schritt 7) sind bei allen getesteten Varianten gleich schnell erfolgt. Das hängt beim 6. Schritt damit zusammen, dass das Bewerten auch beim normalen Adhocracy asynchron im Hintergrund passiert. Das Benutzen des Browser-Back Buttons bei Schritt 7 erzeugt keinen neuen Request, sondern lädt die HTML-Seite aus dem Browsercache. Generell lässt sich aber erkennen, dass das Offline Fallback schneller ist, sobald alle benötigten Daten und Ressourcen lokal vorliegen.

Abbildung 4.6.: Ladezeiten bei generellem Surfverhalten (10 Durchgänge)



Kapitel 5.

Fazit und Ausblick

5.1. Fazit

In dieser Arbeit wurde ein Offline Fallback für eine bestehende Webapplikation, konkret für die Diskussionsplattform Adhocracy, entwickelt, damit die Hauptfunktionen der Anwendung auch ohne Verbindung zum Server möglich sind. Dabei war wichtig, dass die offline Version der Webapplikation weiterhin im Browser läuft und der Benutzer keine extra Software installieren muss.

Um das Ziel dieser Arbeit zu erreichen, waren Änderungen am Server sowie die Entwicklung eines clientseitigen JavaScript Frameworks notwendig. Desweiteren musste sichergestellt werden, dass alle statischen Ressourcen (CSS, Bilder usw.) und alle Daten lokal vorliegen, damit der Browser auch offline auf sie zugreifen kann.

Um alle statischen Ressourcen lokal vorliegen zu haben, wurde der HTML5 Application Cache eingesetzt. Einschränkende Eigenschaften des Appcache machten die Verwendung von Hashbang URLs notwendig, um ihn dennoch bei Adhocracy einsetzen zu können.

Der Server wurde dahin gehend umgebaut, dass er nicht nur fertige HTML-Seiten, sondern auch reine Daten und Templates seperat liefert. Desweiteren kann der Server nun zwischen Anfragen von Benutzern des Offline Fallback und Benutzern der bisherigen Version der Webanwendung unterscheiden und dementsprechend Daten oder HTML liefern.

Das JavaScript Framework wurde so entwickelt, dass es Teile der Server-Logik in den Browser portiert: es speichert nun alle Daten lokal und erzeugt das HTML. Desweiteren synchronisiert es die Daten mit dem Server und sorgt so dafür, dass dem Nutzer stets die aktuellen Daten vorliegen.

Zusätzlich zur Offline-Verwendbarkeit der Webapplikation wurde aufgezeigt, dass die Verwendung des Offline Fallback auch einen Performance Gewinn für die allgemeine Nutzung mit sich bringt. Die

dazu durchgeführten Tests wiesen nach, dass nach einer vergleichweise längeren Initialisierungsphase das generelle Surfverhalten sowie das Erstellen von Vorschlägen und Kommentaren schneller war.

5.2. Ausblick

5.2.1. JavaScript Framework

Das im Rahmen der Arbeit entwickelte Framework für Adhocracy ist noch ein sehr einfaches, weshalb bei zukünftigen Betrachtungen des Themas folgende Punkte verbessert werden könnten:

Markdown

Markdown[mkd] ist ein einfaches Schriftsatzsystem, beispielsweise mit Regeln für Überschriften und Listen. In Adhocracy wird es eingesetzt, damit die Benutzer Vorschlags- und Kommentartexte strukturieren und das Aussehen anpassen können. Bisher übersetzt der Server Markdown in HTML, wenn er eine HTML-Seite erstellt.

Beim Offline Fallback wird das HTML im Browser erstellt, wobei Markdown derzeit noch nicht unterstützt wird. Es existieren aber JavaScript Renderer für Markdown, sodass es recht schnell eingebunden werden könnte.

Synchronisation

Wie beschrieben, ist die Synchronisation beim Prototypen sehr einfach gehalten. Das Framework pollt den Server alle fünf Minuten und bekommt vom ihm eine neue Kopie der gesamten Instanz, wenn sie einen aktuelleren Zeitstempel hat.

Langfristig wäre es aber sinnvoll, von Polling auf Server Push umzustellen, damit die Kommunikation nur noch dann stattfindet, wenn es tatsächlich neue Daten gibt. Um Server Push zu erreichen, gibt es zum Beispiel Websockets. Damit baut der Server eine direkte TCP-Verbindung zum Client bzw. dem JavaScript Framework auf und kann ihm anschließend neue Daten per Push schicken. Das Problem dabei ist, dass das Konzept von Websockets von der Serverapplikation unterstützt werden muss – Adhocracy baut aber auf einer Pylons Version auf, in der dies nicht ohne weiteres möglich ist.

Desweiteren wäre auf lange Sicht die Verwendung eines vollständig versionierten Systems sinnvoll, damit neben der Anzahl der Requests auch die Datenmenge auf ein Minimum reduziert werden kann.

URL Schema

Der in dieser Arbeit entwickelte Prototyp verwendet Hashbang URLs, um implizite Master-Entries zu verhindern. Wie erläutert, bringt das aber viele Nachteile mit sich. Daher sollte das Framework, sobald es eine breit unterstützte Intercept Methode gibt, darauf umgestellt werden. Auf diese Weise werden die URLs zwischen Benutzern des Offline Fallback und der normalen Version wieder austauschbar und zum anderen könnte der On-Page-Anker wieder sinngemäß eingesetzt werden.

Initialisierung

Wie in der Evaluation aufgezeigt, benötigt man beim Offline Fallback deutlich länger, bis die Webseite für die Verwendung bereit ist. Deshalb sollten die an jener Stelle vorgeschlagenen Verbesserungsmethoden umgesetzt werden, um so die Wartezeit zu verkürzen. Das bedeutet zum einen, die Verlegung der kompletten Initialisierung des Offline Fallback in ein IFrame und zum anderen, ein effizienteres Abrufen des Cache Manifests. Desweiteren könnte man sich überlegen, das Cache Manifest doch als Datei auf dem Server abzulegen – dann ist es zwar nicht so flexibel, könnte aber die Wartezeit verkürzen.

5.2.2. Integration in Adhocracy 2

In dieser Arbeit wurde mit der ersten Version von Adhocracy gearbeitet, der zu diesem Zeitpunkt aktuellsten Version. Mittlerweile ist aber Adhocracy 2 erschienen, sodass überprüft werden müsste, ob und mit welchem Aufwand sich die hier dargestellten Serveränderungen auf Adhocracy 2 übertragen lassen.

An der Grundstruktur der Serversoftware hat sich nicht viel verändert, sodass die Änderungen an Model und Controller wahrscheinlich eins zu eins übernommen werden können. Allerdings wurde bei Adhocracy 2 Theming eingebaut, was vom Framwork dieser Arbeit nicht unterstützt wird.

Theming

Mit Hilfe von Theming können Struktur und Layout von HTML-Seiten umgebaut werden, ohne, dass man dazu die eigentlichen Templates umschreiben muss – Adhocracy 2 verwendet dafür Diazo[dia11]. Dazu werden zunächst die Mako Templates gerendert. Anschließend kann die Struktur der fertigen HTML-Seite mit Hilfe von Diazo-Regeln bearbeitet werden.

Die Schwierigkeit im Bezug auf das Offline Fallback ist Folgende. Da beim Offline Fallback das HTML lokal erstellt wird, müsste das Layout der fertigen HTML-Seite auch hier anchließend mit Hilfe von Diazo-Regeln angepasst werden können. Dazu benötigt man eine Diazo-Implementierung in JavaScript, die es aber nicht gibt.

Möchte man im Offline Fallback also Theming verwenden, müsste entweder Diazo für JavaScript implementiert oder eine Möglichkeit gefunden werden es vor dem HTML-Rendering durchzuführen, sodass das Ergebnis des Themings ein Mustache Template ist. Die fertige HTML-Seite könnte dann wie bisher auch lokal auf dem Client erstellt werden.

Anhang A.

Weitere Abbildungen

Listing A.1: Aktuelles Cache Manifest bei Adhocracy

```
1  CACHE MANIFEST
2  # v0.16
3  NETWORK:
4  *
5  CACHE:
6  /locales/de-DE/translation.json
7  /template/instance_show
8  /template/proposal_show
9  /template/proposal_new
10 /template/proposal_edit
11 /template/proposal_tile_overview
12 /template/comment_tile_show
13 /template/account_bar
14 /style/style.css?v=97ebc331b4b111e79049dba2d3ceea61
15 /js/adhocracy-all_de.js?v=bc60aec6e75a00f0de765f734ed02d51
16 /img/arrows/arrow_down_active.png
17 /img/arrows/arrow_down_inactive.png
18 /img/arrows/arrow_up_active.png
19 /img/arrows/arrow_up_inactive.png
20 /img/unicoop_logo.png
21 /img/uni_duesseldorf_logo.gif
22 /img/sidebar_background.png
23 /img/icons/vote_against.png
24 /img/menu_bg.png
25 /img/loading.gif
26 /img/tile_bg.png
```

Listing A.2: Ausschnitt einer Instanz im JSON Format

```
{
    "_templates": {
        "show": "instance_show"
    },
    "instance": {
        "id": 2
        "num_proposals": 1,
        "creator": "admin",
        "label": "Testdiskussion",
        "instance_url": "/i/testdiskussion/",
        ...
        "proposals": [
            {
                "id": 7,
                "creator": "admin",
                "votes_for": 0,
                "votes_against": 0,
                "votes_id": 4,
                "label": "Ein toller Vorschlag",
                "comment_count": 1,
                ...
                "comments": [
                    {
                        "creator": "admin",
                        "url": "/i/testdiskussion/comment/2",
                        "text": "Toller Kommentar\r\n",
                        "votes_for": 1,
                        "topic": 8,
                        "votes_class": "upvoted",
                        "votes_id": 5,
                        "vote_against": 0,
                        ...
                    }
                ]
            }
        ]
    }
}
```

Literaturverzeichnis

[adh] *Adhocracy.de*. https://adhocracy.de/_pages/about/uber-adhocracy, .

[app12a] *Application cache: Disk space*. : *Application cache: Disk space*. http://www.w3.org/TR/html5/offline.html#disk-space, 2012

[app12b] *Application caches*. : *Application caches*. http://www.w3.org/TR/html5/offline.html#appcache, 2012

[app12c] *Can I Use: Appcache*. http://caniuse.com/#search=appcache, 8 2012.

[Bar11] BARTH, A.: *HTTP State Management Mechanism*. RFC 6265 (Proposed Standard). http://www.ietf.org/rfc/rfc6265.txt. Version: April 2011 (Request for Comments).

[cdr] *chromedriver - WebDriver for Google Chrome*. http://code.google.com/p/chromedriver/, .

[css10] *Cascading Style Sheets (CSS) Snapshot 2010*. http://www.w3.org/TR/CSS/, 2010.

[ctr] *Cut the rope*. http://www.cuttherope.ie, .

[dac] *Add an API to make appcache support caching specific URLs dynamically*. https://www.w3.org/Bugs/Public/show_bug.cgi?id=17974, .

[dia11] *Diazo*. http://docs.diazo.org/en/latest/index.html, 2011.

[dom12] *DOM4*. http://www.w3.org/TR/dom/, 2012.

[esm79] *Evaluating synchronization mechanisms*. 1979 . 24–32 S.

[fcb] *Facebook*. http://facebook.com, .

[ffd] *FirefoxDriver*. http://code.google.com/p/selenium/wiki/FirefoxDriver, .

[FGM+99a] FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). http://www.ietf.org/rfc/rfc2616.txt. Version: Juni 1999 (Request for Comments). Updated by RFCs 2817, 5785, 6266

[FGM+99b] FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). http://tools.ietf.org/html/rfc2616#section-14.26. Version: Juni 1999 (Request for Comments). Updated by RFCs 2817, 5785, 6266

[fsa12a] *Filesystem API*. http://www.w3.org/TR/file-system-api/, 2012.

[fsa12b] *Filesystem API: Security Considerations*. http://www.w3.org/TR/file-system-api/#security-considerations, 2012.

[gof] *Using Gmail, Calendar and Docs without an Internet connection*. http://gmailblog.blogspot.de/2011/08/using-gmail-calendar-and-docs-without.html, .

[gop] *Google+*. https://plus.google.com/?hl=de, .

[his12] *The History interface*. : *The History interface*. http://www.w3.org/TR/html5/history.html#the-history-interface, 2012

[htm12a] *HTML: The Markup Language*. http://www.w3.org/TR/html-markup/, 2012.

[htm12b] *HTML5*. http://www.w3.org/TR/html5/, 2012.

[ifr] *iframe – nested browsing context*. http://dev.w3.org/html5/spec/browsers.html#nested-browsing-context, .

[jso] *JSON*. http://www.json.org, .

[lbf12] *Facebook: Like Box*. http://developers.facebook.com/docs/reference/plugins/like-box/, 2012.

[mak] *Mako 0.7.1 Documentation*. : *Mako 0.7.1 Documentation*. http://docs.makotemplates.org/en/latest/index.html

[mjs] *Mustache.js*. : *Mustache.js*. https://github.com/janl/mustache.js

[mkd] *Markdown*. http://daringfireball.net/projects/markdown/,.

[mus] *Mustache Manual*. : *Mustache Manual*. http://mustache.github.com/mustache.1.html

[mvc] *Model-View-Controller*. http://openbook.galileocomputing.de/oo/oo_06_moduleundarchitektur_001.htm#Rxxob06moduleundarchitektur001040015ff1f012130,.

[nex] *Nexus von Google*. http://www.google.de/nexus/#/,.

[owa12] *Offline Web applications*. : *Offline Web applications*. http://www.w3.org/TR/html5/offline.html#offline, 2012

[pin] *Pinterest*. http://pinterest.com,.

[pyl] *The Pylons Project Documentation*. : *The Pylons Project Documentation*. http://docs.pylonsproject.org/en/latest/index.html

[Res00] RESCORLA, E.: *HTTP Over TLS*. RFC 2818 (Informational). http://www.ietf.org/rfc/rfc2818.txt. Version: Mai 2000 (Request for Comments). Updated by RFC 5785

[sfs04] *A synchronization framework for personal mobile servers*. 2004 . 208–212 S.

[skp] *Sketchpad*. http://mudcu.be/sketchpad/,.

[spw12] *SPWR: A Framework to Enable Web Applications Work Offline in Challenged Network Environments*. Bd. *5*. 2012 . 227–236 S.

[sql] *SQLite*. http://www.sqlite.org/about.html,.

[twt] *Twitter*. http://twitter.com,.

[url01] *URIs, URLs, and URNs: Clarifications and Recommendations 1.0*. http://www.w3.org/TR/uri-clarification/, 2001.

[web10a] *Google is eating Microsoft's lunch, one tasty bite at a time.* http://blog.rescuetime.com/2010/06/17/google-is-eating-microsofts-lunch-one-tasty-bite-at-a-time/, 2010.

[web10b] *Web SQL Database*. http://www.w3.org/TR/webdatabase/, 2010.

[web11] *Web Storage*. http://www.w3.org/TR/webstorage/, 2011.

[wsq12] *Can I Use: Web SQL Database*. http://caniuse.com/#search=websql, 8 2012.

[wst12] *Can I Use: Web Storage*. http://caniuse.com/#search=webstorage, 8 2012.

[xhr] *XMLHttpRequests*. http://www.w3.org/TR/XMLHttpRequest/,.

[xml] *XML*. http://www.w3.org/XML/,.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 19.September 2012 Tim van Cleef

Hier kommt die CD rein