Studiengangsspezifisches Deckblatt

Das studiengangsspezifische Deckblatt ist in der Materialiensammlung enthalten.
Das Deckblatt kann separat ausgefüllt und ausgedruckt werden.

Kurzfassung

TODO

English Title of the Thesis

Abstract

TODO

Ehrenwörtliche Erklärung

Hiermit erkläre ich, **Philipp Reinking**, geboren am **21. Januar 1989 in Marl**, ehrenwörtlich, dass ich meine Diplom-/Bachelor-/Masterarbeit mit dem Titel:

„*Entwicklung einer Social Extranet Plattform als hybride Applikation für mobile Endgeräte mit Online und Offline Synchronisierung*“

selbstständig und ohne fremde Hilfe angefertigt und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe.

Die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen habe ich innerhalb der Arbeit gekennzeichnet.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Zweibrücken, 14. April 2014

Sperrvermerk

Die vorliegende Bachelorarbeit enthält vertrauliche Daten und Informationen der **Pixelpark AG**. Veröffentlichungen oder Vervielfältigungen - auch nur auszugsweise - sind ohne ausdrückliche schriftliche Genehmigung des Unternehmens nicht gestattet.

Die Arbeit ist nur den Korrektoren sowie erforderlichenfalls den Mitgliedern des Prüfungsausschusses zugänglich zu machen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Soziale Netzwerke haben sich in den letzten Jahren zu einem alltäglichen Instrument der Kommunikation und Informationsbeschaffung entwickelt. Menschen können mithilfe solcher Netzwerke sehr einfach miteinander in Kontakt treten, sich vernetzen und organisieren.
Auch das Nachrichtenwesen befindet sich zur Zeit im Wandel, denn Informationen verbreiten sich auf Twitter oder Facebook meist schneller, als über klassische Medien wie TV, Radio oder Print. Die oftmals als "viral" bezeichnete Ausbreitung von Beiträgen ist ein Phänomen, das man sehr häufig in sozialen Netzwerken beobachten kann und von dem viele Personen, Organisationen und Unternehmen zu profitieren versuchen, besonders wenn es um die Reichweite von Informationen und Kommunikation geht.

Ein soziales Netzwerk, in welchem sich Menschen miteinander vernetzen können, ob als Extranet einer Organisation oder als öffentliches Netzwerk, wird heutzutage von einer Vielzahl von Faktoren bestimmt und beeinflusst.

Ein vor allem entscheidender Faktor in einem Netzwerk ist die Orientierung und Ausrichtung an einem bestimmten Thema oder Themenbereich. Und immer häufiger entstehen auf solche sehr beschränkte Themenbereiche spezialisierte Plattformen, die unter Umständen zwar weniger Personengruppen ansprechen können, aber in der grundlegenden Funktionsweise das Ziel dieses Netzwerks besser unterstützen. (Beispiele: dribbble.com, stackoverflow.com, github.com).

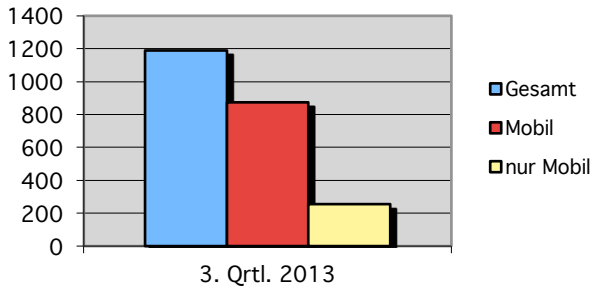

Abbildung 1 - Facebook Nutzerzahlen 2013

Auch die mobile Internetnutzung muss als weiterer wichtiger Faktor genannt werden, dem immer größere Bedeutung zugesprochen wird. Facebook liefert beispielsweise interessante Statistiken dazu (siehe Abbildung 1). So haben ca. 73% der monatlich aktiven Nutzer das Netzwerk auch von einem mobilen Endgerät genutzt. Zusätzlich nutzten ca. 21% der Nutzer das Netzwerk ausschließlich über ein mobiles Gerät. Ein Grund für die ausschließliche mobile Nutzung kann das Fehlen der entsprechenden Infrastruktur für einen festen Internetanschluss sein, insbesondere in Entwicklungsländern.

Zusätzlich sind Sicherheit, Datenschutz und Privatsphäre gerade zur heutigen Zeit, nicht zuletzt wegen der NSA-Affäre und dem Whistleblower Edward Snowden, immer wieder Bestandteil von Diskussionen wenn es um die Übertragung von Kommunikation und Informationen über das Internet geht und ebenfalls ein Faktor für moderne soziale Netze. OpenSource Software gewinnt so natürlich immer mehr an Bedeutung, da offener Quellcode von jedem Menschen eingesehen werden kann und die Wahrscheinlichkeit reduziert, dass eine Applikation seine Nutzer aktiv ausspioniert.

1.2 Aufgabenstellung im Detail

Die rasant fortschreitende Entwicklung im Bereich der Webtechnologien, welche auch immer stärker für die Verwendung auf mobilen Endgeräten wie Smartphones oder Tablets ausgelegt sind, ermöglicht webbasierten Applikationen eine größere Leistungsfähigkeit. Nicht nur durch schnellere Hardware, sondern auch durch bessere Webbrowser und deren JavaScript Interpreter hat ein Webentwickler heutzutage größere Chancen auf verschiedensten Endgeräten, ob Mobil oder Desktop, die gleichen technischen Voraussetzungen vorzufinden und so Applikationen mit einer einheitlichen Codebasis zu schreiben, die auf allen Geräten ohne größere Unterschiede funktionieren.

Diese Arbeit widmet sich dem Ziel, unter Beachtung der zuvor genannten Faktoren, eine experimentelle Basisplattform für sogenannte Social Extranets zu entwickeln. Es soll untersucht werden ob die erarbeitete Lösung für einen Einsatz unter realen Bedingungen geeignet ist und welche Anforderungen erfüllt werden müssen um einen sinnvollen Einsatz der Plattform, auch im Hinblick auf die Erweiterbarkeit und Weiterentwicklung durch andere Entwickler, zu gewährleisten.

Im Kern entstehen so zwei Anforderungen die im Laufe dieser Arbeit bewältigt werden sollen. Die Applikation soll zum einen mit mobilen Geräten benutzbar sein und zum anderen auch noch ohne aktive Internetverbindung (eingeschränkt) funktionieren.

Als wichtigste Komponente für dieses Vorhaben steht die Datenbank im Mittelpunkt. Im Allgemeinen hat man heutzutage meist noch sehr begrenzte Möglichkeiten um in einem Webbrowser große Mengen an Daten ohne den Einsatz externer Datenbanksoftware direkt im Browser zu speichern. Um die Applikation grundsätzlich unabhängig von externen Einflüssen und darüber hinaus auch offline funktionsfähig zu machen, muss die Datenbank also innerhalb der Laufzeitumgebung (PhoneGap/Webview) bereitgestellt werden.

Die Offline Funktionalität stellt eine weitere Herausforderung da, denn es muss dafür Sorge getragen werden, dass Änderungen welche auf einem Gerät ohne Internetverbindung getätigt wurden, auch auf andere Geräte übertragen werden können, sobald dieses Gerät wieder eine Internetverbindung hat und im Umkehrschluss müssen Änderungen anderer Geräte auch empfangen werden. Das Netzwerk muss also in der Lage sein Datenbanken in alle Richtungen zu replizieren.

1.3 Umfeld

Diese Arbeit entstand im Innovation Lab der Pixelpark AG am Standort Köln. Das Innovation Lab ist eine Forschungsabteilung die aktiv mit neuen und teilweise experimentellen Technologien verschiedenste Projekte umsetzt. Als Teilnehmer des "Future Internet" Programms im Bereich "FIcontent2" der Europäischen Union übernimmt Pixelpark die Verantwortung zur Erstellung eines sogenannten "Social Network Enablers".

Das Social Network Enabler Projekt ist im Bereich der Smart City Services angesiedelt und steht als offene Plattform für Entwickler zur Verfügung, um darauf aufbauend eigene Ideen umzusetzen.

Die in dieser Arbeit erreichten Ergebnisse werden als Kernkomponente in den Social Network Enabler eingebunden.

1.4 Planung

2 Theoretische Grundlagen

2.1 NoSQL

NoSQL ist im allgemeinen ein Begriff für Datenbanksysteme, die nicht dem Prinzip von relationalen Datenbanken folgen. Daten werden nicht relational gespeichert und es wird auch kein SQL als Abfragesprache verwendet.

In der Regel werden bei relationalen Datenbanken sogenannte Transaktionen verwendet, welche bei Zugriff auf einen Datensatz, diesen Datensatz für andere Transaktionen sperren und dabei dem ACID-Prinzip folgen. NoSQL setzt dagegen auf eine andere Vorgehensweise, welche auch als BASE bekannt ist und von Eric Brewer definiert wurde:

- **Basic availability**: Jede Anfrage erhält garantiert eine Antwort
- **Soft state**: Daten können sich auch ohne spezifische Eingaben verändern
- **Eventual consistency**: Die Datenbank kann zeitweise inkonsistent sein, kehrt dann aber zu einem konsistenten Status zurück

In einem verteilten Datenbanksystem definiert Eric Brewer 3 Eigenschaften, welche für verschiedene Anwendungsfälle wichtig sein können: 'Consistency', 'Availability' und 'Partition tolerance'. Das sogenannte CAP-Theorem besagt, das lediglich nur zwei dieser drei Eigenschaften in einem Datenbanksystem garantiert werden können. Im Falle von NoSQL sind Verfügbarkeit und Partitionstoleranz die herausstechenden Merkmale.

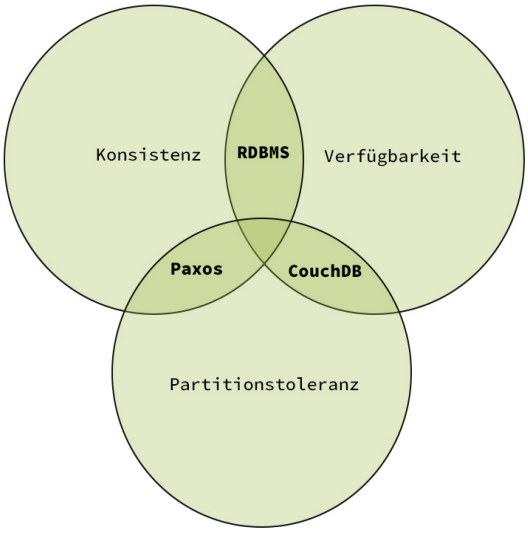

Abbildung 2 - CAP-Theorem

Letztendlich sind die Vorteile von NoSQL Datenbanken vor allem hohe Skalierbarkeit und Geschwindigkeit. Des weiteren ist auch die oft schemalose Repräsentation und die damit verbundene Flexibilität von Datensätzen ein Grund für Entwickler auf NoSQL Datenbanken zu setzen, da hier Entwicklungszeit eingespart wird.

2.2 IndexedDB

Die IndexedDB ist eine HTML5 Spezifikation und befindet sich im Entwicklungsprozess des W3C in der 'Last Call Working Draft' Phase. Die Implementierung in den aktuellen Browsern ist relativ konsistent umgesetzt, d.h. es gibt keine merkbaren Unterschiede. Lediglich Apple mit Safari und iOS und Opera Mini bieten aktuell keine Unterstützung für die IndexedDB-Spezifikation.

IndexedDB macht es möglich große Datenmengen im Browser des Nutzers zu speichern. Die dort abgelegten Daten können mithilfe einer API durchsucht werden.

Im Gegensatz zu Cookies oder Webstorage besteht für die IndexedDB theoretisch kein Limit was die Größe des verfügbaren Datenspeichers betrifft, diese kann aber durch den Browser begrenzt sein. Jede Applikation die notwendigerweise große Datenmengen übertragen muss, kann davon profitieren diese Daten dauerhaft in dem Browser des Nutzers zu speichern. Ebenso hat der Webstorage keinen Suchmechanismus, sondern nur die Möglichkeit über einen *Key* auf einen *Value* zuzugreifen. Die IndexedDB kommt dem Begriff einer Datenbank also sehr viel näher, als der einfache Webstorage.

2.3 AngularJS

AngularJS ist ein MVC (Model-View-Controller) JavaScript Framework zum Erstellen von interaktiven, dynamischen Single Page Applications.

Bis zum Jahr 2008 war JavaScript eine vergleichsweise langsame Skriptsprache. Durch ständiges Weiterentwickeln der JavaScript Engines, konnte das Ausführen von JavaScript Code in den letzten Jahren jedoch deutlich beschleunigt werden. Durch diesen enormen Performance-Zuwachs in der JavaScript Technologie, konnte man in Erwägung ziehen JavaScript für sogenannte Rich Internet Applications zu verwenden.

Ein Problem das viele JavaScript Frameworks versuchen zu beheben, ist die Tatsache das HTML eine statische Auszeichnungssprache ist. HTML ist nicht dazu konzipiert worden dynamisch zu sein oder dem Nutzer reichhaltige Interaktionsmöglichkeiten zu bieten. Das moderne Web verlangt jedoch nach Lösungen, sowohl im Desktopbereich auch als im mobilen Sektor, Webapplikationen von Grund auf interaktiv zu gestalten.

Eine Methode um den Nutzer mit einer Webapplikation interagieren zu lassen, ist der herkömmliche Weg über einen Webserver, welcher Nutzereingaben verarbeitet und darauf in der Regel eine Antwort an den Nutzer in Form einer HTML Seite ausgibt. Hier entstehen jedoch schon einige Nachteile. Für jede Interaktion des Nutzers muss der Server mit einbezogen werden, was nicht nur Traffic verursacht und vor allem

schlecht für mobile Geräte ist, sondern je nach Auslastung des Servers auch noch längere Ladezeiten verursachen kann.

Eine andere Methode zur Umsetzung interaktiver Webapplikationen ist die Manipulation von HTML und CSS über JavaScript. Als bekanntestes Beispiel ist hier jQuery zu nennen, eine JavaScript Bibliothek die unter anderem eine einfache API zur Manipulation des DOM bietet.

Jedoch ist jQuery keine vollständige Lösung um interaktive Anwendungen zu erstellen, sondern bietet nur eine Schnittstelle zum DOM. AngularJS und vergleichbare Frameworks setzen hier an und bieten zum Teil sehr gut durchdachte Konzepte um Rich Internet Applications effizient zu erstellen.

Eine Fähigkeit von AngularJS ist das 2-way-data-binding, welches den Zugriff auf Models direkt aus der View heraus ermöglicht. Veränderungen am Model können dann durch das Framework erkannt werden und das DOM wird aktualisiert.

```
<body ng-app ng-init="name = 'World'">
    Say hello to: <input type="text" ng-model="name">
    <h1>Hello, {{name}}!</h1>
</body>
```

Abbildung 3 - AngularJS 2-way-data-binding

Eine Besonderheit von AngularJS ist die Fähigkeit HTML um eigene Tags anreichern zu können. Durch die Implementierung sogenannter Direktiven werden Funktionen nicht nur vom restlichen Quellcode abgekapselt und dadurch wiederverwendbar, sondern bekommen in der endgültigen View eine semantische Bedeutung und vereinfachen somit die Lesbarkeit für Entwickler. In Abbildung 4 wird die von AngularJS mitgelieferte Direktive ng-repeat verwendet um über ein Array zu iterieren und dessen Inhalte auszugeben.

```
<ul ng-controller="WorldCtrl">
    <li ng-repeat="country in countries">
        {{country.name}} has population of {{country.population}}
    </li>
    <hr>
    World's population: {{population}} millions
</ul>

<script>
    var WorldCtrl = function ($scope) {
        $scope.population = 7000;
        $scope.countries = [
            {name: 'France', population: 63.1},
            {name: 'United Kingdom', population: 61.8},
        ];
    };
</script>
```

Abbildung 4 - AngularJS ngRepeat Direktive

2.4 Phonegap

Phonegap ist ein OpenSource Framework um die Entwicklung von hybriden Applikationen für Smartphones mittels HTML5, CSS und JavaScript zu ermöglichen.

Eine erste wesentliche Fähigkeit von Phonegap ist das Bereitstellen installierbarer Applikationen für eine Vielzahl von Betriebssystemen, darunter iOS, Android, WindowsPhone, Blackberry, WebOS, Bada und Symbian. Dabei entsteht jede Applikation aus der gleichen Codebasis, bestehend aus HTML, CSS und JavaScript. Um auf allen Betriebssystemen denselben Code ausführen zu können, wird die Applikation in einer Webview ausgeführt.

Eine native Applikation auf dem Handy ist effektiver, wenn man auf gerätespezifische Funktionen, wie zum Beispiel die Kamera, zugreifen kann. Hier stellt Phonegap eine API zur Verfügung die mit JavaScript angesprochen werden kann.

Der Build-Prozess einer Applikation erfordert zusätzlich das SDK des entsprechenden Betriebssystems. Zum Beispiel muss für die Android Kompilierung das Android SDK installiert sein und für die Kompilierung zu einer iOS App wird xCode benötigt.

3 Datenbankreplikation mit CouchDB

Die Entscheidung für ein passendes Datenbanksystem ist für das Ergebnis dieser Arbeit ein sehr wichtiger Faktor. Es gibt zahlreiche Hersteller und damit verbunden eine sehr große Anzahl von entsprechenden Lösungen. Im Folgenden werden die Anforderungen an die Datenbank definiert und die Auswahl für die in dieser Arbeit verwendeten Lösung begründet.

3.1 Analyse der Anforderungen

Die Anforderungen an die Datenbank werden aus einem geplanten Testszenario hergeleitet, welches im Laufe dieser Arbeit an einer Schule mit ca. 15 Testpersonen durchgeführt wird:

"Eine Smartphone Applikation soll für maximal 15 Person ermöglichen, Statusmeldungen in Form von Text über ein Netzwerk in Echtzeit auszutauschen. Das Experiment wird an einem Ort mit mangelhaftem Mobilfunk durchgeführt, sodass Verbindungsabbrüche zum Netzwerk auftreten können."

Aus diesem Szenario ergeben sich folgende Einschränkungen:

- das Netzwerk hat maximal 15 Teilnehmer
- Teilnehmer können Statusmeldungen in Form von Text verschicken
- Statusmeldungen werden nicht editiert
- die Netzwerkverbindung kann unterbrochen werden

Die folgenden drei Anwendungsfälle verdeutlichen das gewünschte Verhalten der finalen Applikation:

3.1.1 Use Case: Initialer Start

Vorbedingungen:
App startet zum ersten Mal.

Beschreibung:
1. Es wird ein Ladesymbol angezeigt.
2. Alle relevanten Daten aus dem Netzwerk werden zur Offline-Nutzung auf das Gerät geladen und gespeichert.
3. Die Statusmeldungen werden chronologisch angezeigt.

3.1.2 Use Case: Statusmeldung absenden

Vorbedingungen:
App ist gestartet; Initialer Start durchgeführt.

Beschreibung:
1. Der Teilnehmer gibt seine Statusmeldung ein und drückt auf Senden.
2. Der Teilnehmer ist online, die Statusmeldung wird lokal gespeichert und über das Netzwerk synchronisiert.

Alternativ:
2b. Der Teilnehmer ist offline, die Statusmeldung wird nur lokal gespeichert.

Nachbedingung:
Die Statusmeldung ist garantiert auf dem Absendergerät gespeichert.

3.1.3 Use Case: Statusmeldungen synchronisieren

Vorbedingungen:
Statusmeldungen wurden nur lokal gespeichert; Während der Offline-Nutzung wurden im Netzwerk neue Statusmeldungen versendet.

Beschreibung:
1. Das Teilnehmergerät verbindet sich mit dem Internet.
2. Zuvor nur lokal gespeicherte Statusmeldungen werden über das Netzwerk synchronisiert.
3. Neue Statusmeldungen aus dem Netzwerk werden auf dem Gerät gespeichert und angezeigt.

Nachbedingung:
Alle neuen Statusmeldungen wurden über das Netzwerk übertragen.

3.2 Übertragung der Anforderungen auf die Datenbank

Auf Ebene der Datenbank lassen sich 3 funktionale Anforderungen identifizieren, die benötigt werden um das zuvor beschriebene Szenario erfolgreich bestehen zu können.

A: Kompatibilität zur Laufzeitumgebung PhoneGap (Webview):
Diese Anforderung ergibt sich vorwiegend aus der Wahl von PhoneGap als Zielplattform. Die Datenbank muss mit JavaScript kompatibel sein und über eine entsprechende Schnittstelle verfügen, die auch auf mobilen Geräten verwendet werden kann.

B: Daten können offline gespeichert werden:
Die Datenbank kann Daten auch offline speichern und ist damit unabhängig von einer aktiven Internetverbindung.

C: Synchronisierung zwischen beliebig vielen Geräten:
Um Statusmeldungen auf alle Geräte verteilen zu können, muss die Datenbank in der Lage sein, diese Daten zu synchronisieren.

3.3 Auswahl der Datenbank

Die wohl am schwersten zu bewältigende Anforderung ist durch die Wahl von PhoneGap als Zielplattform entstanden. Die Tatsache, dass die Anwendung in einer Webview laufen wird, verringert die Auswahlmöglichkeiten einer passenden Datenbank.

3.3.1 Lokaler Datenspeicher in PhoneGap

Für HTML5 sind bisher 3 Datenspeicher entwickelt worden:

- Webstorage kann als Ersatz für herkömmliche Cookies betrachtet werden und funktioniert nach dem Key-Value-Prinzip, ist aber eher für kleine Datenmengen ausgelegt.
- WebSQL ist eine Sqlite-Implementation für den Browser, wird jedoch nicht weiterentwickelt und soll daher auch nicht näher betrachtet werden.
- Die IndexedDB ist eine für größere Datenmengen ausgelegte Datenbankschnittstelle zum Speichern von komplex strukturierten Objekten, die durch *Keys* indexiert werden.

Objekte werden in der IndexedDB mit einem *Key* verknüpft. Es gibt keine Query-Language wie bei relationalen Datenbanken üblich, sondern es wird mithilfe eines Cursors über die *Keys* iteriert um Einträge zu finden.
Die API wird über JavaScript angesprochen und Objekte werden in der JSON Notation gespeichert, sind somit also direkt in JavaScript verwendbar. Die IndexedDB wird auch zur Klasse der Dokumentenorientierten Datenbanken gezählt.

```
var request = indexedDB.open("library");

request.onupgradeneeded = function() {
    // The database did not previously exist, so create object stores and
indexes.
    var db = request.result;
    var store = db.createObjectStore("books", {keyPath: "isbn"});
    var titleIndex = store.createIndex("by_title", "title", {unique: true});
    var authorIndex = store.createIndex("by_author", "author");

    // Populate with initial data.
    store.put({title: "Quarry Memories", author: "Fred", isbn: 123456});
    store.put({title: "Water Buffaloes", author: "Fred", isbn: 234567});
    store.put({title: "Bedrock Nights", author: "Barney", isbn: 345678});
};

request.onsuccess = function() {
    db = request.result;
};
```

Abbildung 5 - IndexedDB API

3.3.2 Synchronisierung

Die lokal gespeicherten Daten in der IndexedDB mit anderen Geräten im Netzwerk zu synchronisieren, ist eine weitere Anforderung, die erfüllt werden muss.

Datenbankreplikationen werden von den meisten modernen Datenbanken unterstützt. Es muss hier aber zwischen zwei Arten von Replikation unterschieden werden. Die meisten Datenbanken müssen früher oder später skaliert werden, um wachsende Schreib- und Lesezugriffe bewältigen zu können. Dafür wird überwiegend die Master-Slave Replikation eingesetzt. Eine im Slave-Modus angebundene Datenbank hat damit die Möglichkeit die Daten der Master-Datenbank zu lesen, besitzt jedoch keine Schreibrechte auf die Master-Datenbank.

Für diese Arbeit ist eine andere Art von Replikation relevant, da die Skalierung bei maximal 15 Teilnehmern keine Rolle spielt. Jedes Gerät im Netzwerk soll Daten auch lokal speichern und später zur zentralen Datenbank replizieren können. Alle Endgeräte besitzen somit eine Master-Datenbank und synchronisieren die Daten zum Server, auf dem sich ebenfalls eine Master-Datenbank befindet. Somit kann jede Datenbank unabhängig von den anderen im Netzwerk befindlichen Datenbanken funktionieren.

Datensätze liegen grundsätzlich als Objekte in der JSON-Notation vor, weshalb eine dazu kompatible Datenbank vorzuziehen ist.

MongoDB und CouchDB sind jeweils Dokumentenorientierte und mit JavaScript kompatible Datenbanken. In Abbildung 5 werden alle relevanten Spezifikationen der Datenbanken gegenübergestellt.

Tabelle 1 - Vergleich MongoDB und CouchDB

	MongoDB	**CouchDB**
Format	BSON	JSON
Versionierung	Nein	Ja
Map/Reduce	JavaScript + andere	JavaScript
Replikation	Master-Slave	Master-Slave, Master-Master
Protokoll	TCP/IP	HTTP/REST API

Die CouchDB übertrumpft im Anwendungsfall dieser Arbeit die MongoDB, da die Unterstützung für JSON-Objekte, Versionierung und Master-Master Replikation gegeben ist.

3.3.3 Gemeinsame API für IndexedDB und CouchDB

Es gibt nicht viele Datenbanken, die eine direkte Kompatibilität zur IndexedDB herstellen. Die PouchDB ist ein im Juni 2010 entstandenes OpenSource Projekt, dass es sich zur Aufgabe gemacht hat, die Funktionsweise der CouchDB direkt in den Browser zu übertragen. Das Projekt befindet sich fortlaufend in Entwicklung, und wurde am 2. Januar 2014 in der Version 1.1.0 veröffentlicht.

Mit der PouchDB ist es möglich, die Replikation zwischen IndexedDB, welche von PouchDB als Speicher verwendet wird, und CouchDB durchzuführen.

3.3.4 IndexedDB, PouchDB und CouchDB

Mit der IndexedDB kommt eine in modernen Browsern verfügbare Datenbank zum Einsatz, welche die Anforderungen A und B erfüllt. Zusätzlich wird die CouchDB in Kombination mit der PouchDB verwendet, um Anforderung C, die Synchronisierung zwischen den Datenbanken, zu erfüllen.

3.4 CouchDB im Detail

Es ist wichtig zu verstehen wie die CouchDB im Detail funktioniert, um einen effizienten Mechanismus zu erarbeiten, der die Daten im Netzwerk zwischen den einzelnen Geräten synchronisieren kann.

Daten werden grundsätzlich als JSON enkodierte Strings gespeichert. Die JSON-Notation ist eine in JavaScript verwendete Schreibweise um Objekte darzustellen. Jedoch wird diese Notation unabhängig von JavaScript, auch von anderen Programmiersprachen verwendet und so hat sich JSON, in Konkurrenz zu XML, zu einem sehr beliebten Format für die Übertragung von Daten innerhalb von Webapplikation entwickelt. Der Vorteil bei Verwendung von JSON als Format in der Applikation dieser Arbeit liegt auf der Hand, denn in AngularJS können die aus der CouchDB gelesenen Daten ohne größeren Aufwand, sprich Umwandlung oder Dekodierung, direkt verwertet werden.

Dokumente in der CouchDB folgen keinem festen Schema. Als Entwickler hat man die Freiheit jederzeit Objekte mit beliebiger Struktur und verschiedenen Datentypen zu speichern. Andere Dokumente werden dadurch nicht beeinflusst.

```
{
    _id: '123',
    _rev: '1-123',
    city: 'Cologne'
}
```

Abbildung 6 - JSON-Objekt

Jedoch gibt es zwei Felder die jedem Dokument zugeordnet werden müssen. Zum einen muss das Feld _id einen innerhalb der Datenbank einzigartigen Wert haben und wird verwendet um das Dokument eindeutig zu identifizieren. Der zweite Wert ist das Feld _rev, welches die Revisionsnummer speichert. Revisionen entstehen genau dann, wenn ein Dokument verändert wird und in der Datenbank unter derselben _id gespeichert werden soll. So lassen sich Änderungen bis zum Erstellen von Dokumenten zurückverfolgen.

Die Abfrage von Daten erfolgt über in der CouchDB erstellte Views und unterscheidet sich damit sehr stark von Datenbankabfragen in relationalen Datenbanksystemen. Jede View enthält eine *Map* Funktion, die auf jedes Dokument einzeln angewendet wird, um einen *key* zu erzeugen.

In der *Map* Funktion kann jeder Wert des aktuellen Dokumentes ausgelesen werden und dazu verwendet werden einen *key* für dieses Dokument zu generieren. Die CouchDB nutzt diese *keys* zur Sortierung in Spalten und macht es so möglich durch Angabe eines bestimmten Bereichs auch große Datenmengen effizient zu durchsuchen und das entsprechende Ergebnis auszugeben.

Das Interessante daran ist, dass die *keys* nicht nur einzelne Werte, sondern auch Arrays sein können. Zueinander in Relation stehende Dokumente können damit in der View hintereinander indexiert werden.

```
{
    _id: 'abc',
    city: 'Cologne'
},
{
    _id: 'def',
    city: 'Zweibruecken'
},
{
    _id: '123',
    place: {
        name: 'Pixelpark AG Cologne',
        related: 'abc'
    }
}
```

Abbildung 7 - JSON Objekte mit Relation

In diesem Beispiel wird gezeigt, wie mithilfe einer Map-Funktion *keys* erstellt werden, die der CouchDB eine sortierte Indexierung und Ausgabe ermöglichen.

```
function(doc){
    if(doc.city){
        emit([doc._id, 0], doc.city);
    } else {
        emit([doc.place.related, 1], doc.place.name);
    }
}
```

Abbildung 8 - Map Funktion

```
{
    key: ['abc', 0],
    value: Cologne },
{
    key: ['abc', 1],
    value: 'Pixelpark AG Cologne'
},
{
    key: ['def', 0],
    value: 'Zweibruecken'
}
```

Abbildung 9 - Ergebnis der Map Funktion

Die Replikation von Datenbanken innerhalb eines CouchDB Clusters erfolgt schrittweise. Das bedeutet dass die Datenbanken keine ständige Verbindung zueinander benötigen, wie es bei vielen relationalen Datenbanksystemen der Fall ist. Sobald eine Datenbank einen Replikationsprozess beendet hat ist sie auch eigenständig funktionsfähig.

Die Applikation entscheidet selbst, wann eine Replikation sinnvoll ist und kann diese durch die CouchDB API auslösen. Durch Angabe von Quelle und Ziel wird bestimmt in welche Richtung repliziert wird.

Natürlich kann es passieren dass Datenbanken während einer Replikation Dokumente beinhalten die zueinander in Konflikt stehen. In diesem Fall besitzt die CouchDB einen Algorithmus, der entscheidet welche der beiden Versionen als aktuell gespeichert werden soll. Das Verlierer-Dokument wird aber keinesfalls verworfen, sondern wird unter einer anderen Revision gespeichert. So ist es auch möglich Applikationen zu bauen, die Konflikte nachträglich überprüfen können und gegebenenfalls selbst entscheiden welches Dokument behalten werden soll.

Um Dokumente auf allen Servern zu löschen, benutzt die CouchDB ein *deleted* Feld. Ein Dokument das gelöscht wurde kann keine neuen Revisionen mehr erhalten und wird ebenfalls bei einer Replikation übertragen, sodass auch in allen anderen Datenbanken dieses Dokument als gelöscht markiert wird.

5 Methodik der Daten-Synchronisierung

Die Applikation benötigt eine Methodik, welche den Replikationsprozess zwischen der lokalen Datenbank und der CouchDB auslösen kann. Dazu ist es zum einen wichtig die mitgelieferten Funktionen der PouchDB zu kennen, zum anderen muss aber auch darauf geachtet werden, dass die erdachte Methodik performant ist.

Es ist ebenfalls von Vorteil die Synchronisierung so unabhängig wie möglich von der restlichen Applikationslogik zu gestalten. Im Detail bedeutet dies, dass die gesamte Logik zum Schreiben, Lesen und Synchronisieren von Daten als Modul für AngularJS programmiert wird.

5.1 Planung

Die erarbeitete Strategie trennt den Replikationsprozess in 2 Komponenten, das replizieren von der lokalen Datenbank zum Server und umgekehrt. Abbildung 10 zeigt das zugehörige Aktivitätsdiagramm für diese beiden Komponenten.

Abbildung 10 - Aktivitätsdiagramm Synchronisierung vom Server

Die Synchronisierung vom Server zur lokalen Datenbank beinhaltet zwei Schritte und geht davon aus das eine aktive Verbindung zum Netzwerk besteht (ONLINE). Apps die zum ersten Mal gestartet werden haben natürlich noch keine lokale Kopie der Datenbank des Netzwerks. Um den Nutzer aber nicht zu lange auf aktuelle Datensätze warten zu lassen, denn diese würden normalerweise als letztes geladen werden, wird der Initiale Replikationsprozess (bei App-Start) mit einem Filter versehen, welcher nur Dokumente vom Server synchronisiert, die in den letzten 24 Stunden gespeichert wurden. Die Zeitspanne wurde für das geplante Szenario auf 24 Stunden festgelegt, kann für andere Anwendungsfälle jedoch unpassend sein und sollte dann dementsprechend geändert werden.

Nachdem die Initiale Synchronisierung abgeschlossen ist, wird ein weiterer Replikationsprozess gestartet. Diesmal ohne Filter, jedoch als andauernder Prozess, welcher jede Änderung auf dem Server sofort registriert und die entsprechenden Daten auf die lokale Datenbank überträgt.

Durch dieses Vorgehen werden 2 Ziele erreicht, zum einen ein performanter Start für die Applikation, denn es werden zuerst die aktuellsten Daten geladen und zum anderen eine dauerhafte Synchronisierung aller Änderungen auf dem Server.

Abbildung 11 - Aktivitätsdiagramm Synchronisierung auf den Server zeigt das Aktivitätsdiagramm zum Replizieren der Daten auf den Server.

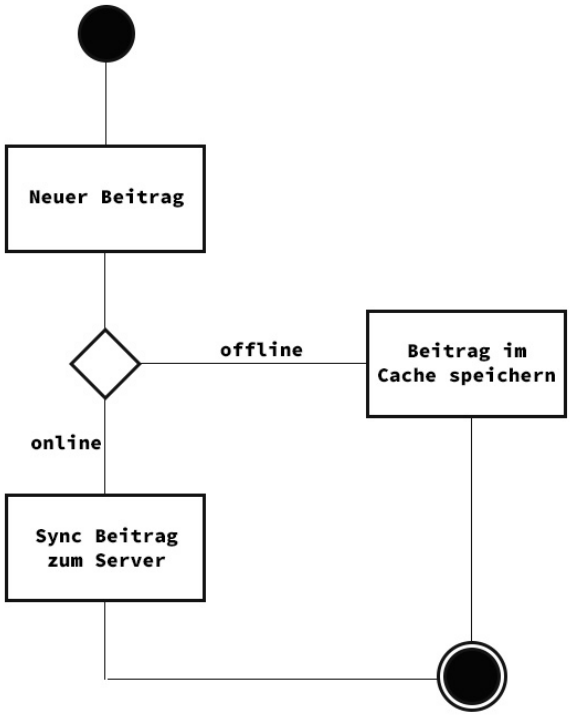

Abbildung 11 - Aktivitätsdiagramm Synchronisierung auf den Server

Die zweite Komponente ist für das Hochladen der Daten auf den Server verantwortlich. Hier macht es wenig Sinn einen dauerhaften Replikationsprozess zu starten, weil neue Daten auf dem lokalen Gerät nur dann entstehen wenn der Nutzer aktiv einen Beitrag hinzufügt. Entgegen dem Vorgehen beim Herunterladen aktueller Daten, wird das Hochladen gezielt beim Speichern neuer Daten ausgelöst.

Nach diesem Vorgehen entsteht für die Offline-Funktionalität ein Hindernis, denn es gibt keine Methode die pauschal alle Daten in der lokalen Datenbank zum Server hoch lädt. Neue Beiträge, die gespeichert werden während das Gerät *Offline* ist, würden so nicht zum Server repliziert werden. Die Lösung hier ist beim Speichern eines neuen Datensatzes direkt zwischen *Online* und *Offline* zu unterscheiden.

Ist das Gerät *Online*, so kann sofort die Methode zum Replizieren auf den Server ausgelöst werden.

Ist das Gerät jedoch *Offline*, wird eine Referenz auf das Dokument zwischengespeichert. Dieser Cache-Mechanismus erlaubt es so alle Änderungen nachträglich zum Server zu replizieren, nämlich genau dann wenn das Gerät wieder eine Netzwerkverbindung erlangt.

Der Nutzen dieses Vorgehens ist eine effektivere Verwendung der Netzwerkapazitäten des Endgerätes. Durch den gezielten Einsatz der Replikation nur auf geänderte bzw. neue Datensätze wird vermieden das alle anderen gespeicherten Daten in der lokalen Datenbank nochmals mit denen der Datenbank auf dem Server abgeglichen werden muss.

5.2 PouchDB API

Zur Umsetzung der Daten-Synchronisierung wird die API der PouchDB verwendet. Die dazu benötigten Methoden werden im folgenden Abschnitt erläutert.

```
var db = new PouchDB('dbname');
var remote = 'http://couchdb.simple-url.com:5984/dbname';
```

Abbildung 12 - Neues PouchDB Objekt erstellen

Die Methode zum Erstellen einer Datenbank betrifft die Replikation zwar nicht direkt, jedoch dient es dem besseren Verständnis. Zur Initialisierung der Datenbank wird ein neues PouchDB-Objekt erstellt. Dabei ist es egal ob diese Datenbank schon besteht oder nicht. Beim erstmaligen Aufruf dieses Objekts wird in der IndexedDB die Struktur für die lokale Datenbank erstellt.

Das zurückgegebene PouchDB-Objekt wird in einer Variablen gespeichert und bietet danach Zugriff auf die API der PouchDB.

```
db.post(object, [Options]);
```

Abbildung 13 - Neues Dokument speichern

Um Dokumente in der lokalen Datenbank zu speichern, wird die Funktion *PouchDB.post* benutzt. Der erste Parameter enthält das zu speichernde Objekt im JSON-Format. Im zweiten Parameter kann ein Objekt mit Optionen übergeben werden. Da fast alle Methoden der PouchDB API asynchrone Funktionsaufrufe sind, werden *Promises* zur Abwicklung der Rückgabewerte verwendet.

Im Abbildung 8 wird ein Dokument gespeichert und der Rückgabewert in der Konsole ausgegeben.

```
db.post({name: 'Hallo FH Zweibruecken'})
    // then wird aufgerufen wenn post fertig ist
    .then(function(response){
        console.log(response);
});
```

Abbildung 14 - Hinzufügen eines Dokuments mit PouchDB

Die Funktion, die letztendlich die Synchronisierung durchführt ist *PouchDB.replicate*. Bei dieser Methode wird im ersten Parameter die URL der entfernten Datenbank übergeben. Der zweite Parameter kann ein Objekt mit Optionen enthalten. Die für diese Arbeit relevanten Optionen sollen im Folgenden erklärt werden.

```
db.replicate.from(remote, [Options]);
db.replicate.to(remote, [Options]);
```

Abbildung 15 - Methoden zur Replikation

```
db.replicate.from(remote,{
    filter: filterFunction,
    doc_ids: [],
    complete: completeFunction,
    continuous: false
});
```

Abbildung 16 - Replikation mit Optionen

filter:
Ruft eine Funktion auf um Dokumente selektiv zu replizieren. Hiermit kann erreicht werden, dass nur Daten geladen werden die einen Zeitstempel aufweisen der nicht älter als 24 Stunden ist. Alternativ kann aber auch nur nach Dokumenten gesucht werden die einen bestimmten Wert enthalten.

doc_ids:
Repliziert nur Dokumente mit den angegebenen *ids*.

complete:
Wenn der Replikationsprozess fertig ist, wird diese Funktion aufgerufen.

continuous (seit PouchDB 2.0 *live*):
Diese Option legt fest ob ein Prozess nur einmalig durchlaufen wird und danach beendet wird oder ob der Prozess fortlaufen auf Änderungen wartet.

6 Modellierungen

Dieses Kapitel behandelt die visuelle und technische Umsetzung der Social Extranet Plattform, im weiteren Verlauf auch als *BAnet* bezeichnet. Ziel ist es, eine um Komponenten erweiterbare, flexible und auf mobile Geräte optimierte Applikation zu entwerfen.

6.1 Model View Controller

Mit AngularJS wird ein Framework eingesetzt, welches das Konzept des MVC Entwurfmusters unterstützt. Der Vorteil dabei ist, dass man mit diesem Muster eine sehr klare Trennung der verschiedenen technischen Bestandteile einer Applikation erreichen kann.

Views sind in AngularJS das, was der Benutzer der Applikation am Ende auf seinem Endgerät sehen soll. Eine solche View ist in der Regel eine einfache HTML-Datei, welche AngularJS-Expressions enthalten kann. AngularJS-Expressions sind eine von mehreren Möglichkeiten, um den Inhalt von Models auszugeben.

Controller stellen die Funktionen für eine View zur Verfügung. In einer View ist ganz genau definiert auf welchen Controller zugegriffen werden soll, weshalb auch nur die Funktionen des entsprechenden Controllers aufrufbar sind. In AngularJS hat man außerdem die Option, bestimmten Teilen des HTML-Codes eigene Controller zuzuweisen.

Models speichern die Daten der Applikation. Die View und der Controller haben Zugriff auf Models und können die enthaltenen Werte manipulieren. Es ist üblich die Geschäftslogik, also das Verwalten, Validieren und Manipulieren der Daten, innerhalb einer eigenen Model-Datei auszulagern.

6.2 Ordnerstrukturen und Entwicklungsumgebung

PhoneGap benötigt eine bestimmte Ordnerstruktur um den Build-Prozess für ein Betriebssystem durchführen zu können. Es empfiehlt sich daher ein PhoneGap Projekt mithilfe des Befehls *„phonegap create ~/path/to/project“* zu erstellen. Daraus resultierend erhält man die in ABBILDUNG gezeigte Ordnerstruktur, welche natürlich noch an die eigenen Bedürfnisse angepasst werden kann.

```
Phonegap Project
├── merges
├── platforms
├── plugins
└── www
    ├── config.xml
    ├── css
    ├── img
    ├── index.html
    ├── js
    ├── res
    └── spec
```

Abbildung 17 - Ordnerstruktur des PhoneGap Projekts

Die gesamte Applikationslogik wird unter dem Ordner www abgelegt. Zusätzlich befindet sich dort die Datei config.xml, eine Konfigurationsdatei in der beispielsweise Name und Versionsnummer der Applikation festgelegt wird, aber auch welche Berechtigungen auf Geräten mit Android benötigt oder welche Plugins eingebunden werden. PhoneGap stellt solche Plugins zur Verfügung und ermöglicht dadurch auf gerätespezifische Funktionen wie Kamera oder GPS zuzugreifen.

Zusätzlich zu dem PhoneGap Projektordner, wird ein weiterer getrennter Ordner zum aktiven Entwickeln der Applikation erstellt. Unter Zuhilfenahme der Tools Yeoman, Grunt und Bower wird das AngularJS Projekt verwaltet.
Der Grund für den Einsatz dieser Tools ist, dass viele Prozesse während der Entwicklung einer modernen Webapplikation automatisiert werden sollten um effizient daran arbeiten zu können.
Als Beispiel soll hier der der Einsatz von Sass in Kombination mit Compass als CSS-Precompiler dienen. Wird eine Sass Datei geändert, kann Grunt automatisch den Compass Compiler ausführen und die CSS Datei damit aktualisieren. Ebenso ist es möglich den JavaScript Code mit einem sogenannten Linter zu validieren, der Warnungen ausgibt, sobald Fehler in der JavaScript Syntax gefunden wurden.

Diese Prozesse sind vollständig auf die eigenen Bedürfnisse einer Anwendung konfigurierbar und bieten damit größtmögliche Flexibilität beim Entwickeln von Applikationen.

Die fertige Applikation wird mit `grunt serve` in einer optimierten Fassung mit minimiertem CSS, JavaScript und HTML generiert.

6.3 Komponenten

Die Applikation *BAnet* wird in ihrer finalen Version 3 Komponenten enthalten, die verschiedene Aufgaben erfüllen und damit die Funktionalität der Synchronisierung demonstrieren.

Der Mechanismus zur Speicherung und Synchronisierung der Daten erfolgt in einem von der Applikation unabhängigen Modul. Durch das Abkapseln der Datenbanklogik

als Modul wird es möglich, dieses auch ohne weiteres in anderen Projekten zu verwenden. Im weiteren Verlauf wird dieses Modul als *ppSync* bezeichnet.

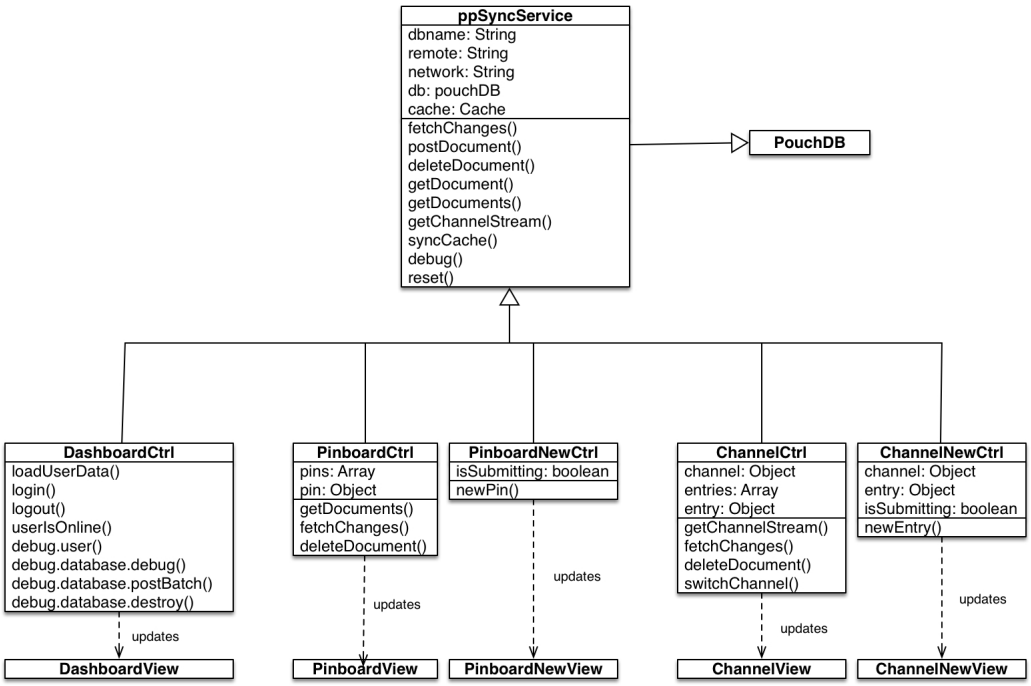

Abbildung 18 - Klassendiagramm BAnet

In Abbildung 18 - Klassendiagramm BAnet wird das Zusammenspiel der einzelnen Schichten in *BAnet* deutlich.

6.4 ppSync Service

AngularJS bietet neben den Controllern eine weitere Möglichkeit um die Logik für eine Applikation modular zu gliedern, die Services. Ein Service ist ein besonderes AngularJS Modul, das nach einer von drei Vorlagen implementiert werden kann, und über den AngularJS Dependency Injector für Controller oder andere Services verfügbar gemacht wird.

Dependency Injection (DI) ist eine sehr mächtige Funktion von AngularJS, denn durch den Einsatz von Services und DI erreicht man eine weitere Kapselung von wiederverwendbaren Code, der durch einfaches Deklarieren von Abhängigkeiten (Dependicies) in Controllern und Services, genau dort verfügbar gemacht wird, wo er auch gebraucht wird, ohne den globalen JavaScript Bereich zu belasten.

Für *ppSync* ist ein Service der optimalste Weg, die Datenbank- und Replikationslogik in *BAnet* zu implementieren. Zusätzlich wird der Umstand genutzt, das Services in AngularJS als Singleton erzeugt werden, also im globalen Bereich nur einmal existiert. Damit ist garantiert, dass jeder Controller mit dem gleichen Datenbank-Objekt interagieren kann und es nur einen Synchronisierungsprozess gibt.

6.5 Visuelle Umsetzung

Die App setzt sich aus 3 wesentlichen Bereichen zusammen:

- Die Navigation im Kopfbereich
- Der Hauptbereich für Inhalte der aktiven Komponente
- Der Fußbereich für optionale Inhalte

Es ist sinnvoll, diese 3 Bereiche vertikal anzuordnen um so jedem Bereich die Möglichkeit zu geben sich auf die volle verfügbare Breite des Anzeigeräts zu erstrecken.

Abbildung 19 - Wireframe Layout

Der Kopfbereich beinhaltet die Navigation und optional ein Logo der Applikation. Hier soll jeder Komponente ein Link zugeordnet werden, welcher bei Bedarf auch ein weiteres Dropdown Menü öffnen kann.

Der Hauptbereich ist ausschließlich für die Inhalte der Komponente reserviert. Das Layout wird von der Komponente bestimmt und kann sich dementsprechend von anderen Seiten unterscheiden. Für diese Arbeit setzt jede Komponente auf eine ebenfalls vertikale Anordnung der Elemente, um so die maximale verfügbare Breite auszunutzen.

Der Fußbereich ist ein optionaler Bereich, der beispielsweise mit ergänzenden Inhalten gefüllt werden kann.

[GRAFIK]

Mit Hilfe des Frontend Frameworks Twitter Bootstrap (Version 3.1.1) kann das geplante Layout responsiv umgesetzt werden. Die verfügbare Bildschirmbreite wird dabei in *BAnet* auf kleineren Geräten immer voll ausgenutzt, auf größeren Geräten wie z.B. einem Desktop Computer wird diese jedoch auf 700px begrenzt um die Elemente nicht zu sehr zu strecken.

Als Einstieg in die Applikation ist die erste Komponente, das Dashboard, gedacht. Das Dashboard wird standardmäßig geladen und beinhaltet das Loginformular wenn der Benutzer nicht angemeldet ist oder, falls eine Anmeldung erfolgt ist, Informationen zur Applikation mit Debugfunktionen. Diese Debugfunktionen sind während der Entwicklung der App sehr wichtig, da es beispielsweise oft notwendig ist die lokale Datenbank zu löschen.

```
<div class="panel panel-primary" ng-hide="userIsOnline()">
    <div class="panel-heading">
        <h3 class="panel-title">Login to BAnet</h3>
    </div>
    <div class="panel-body">
        <form ng-submit="login()">
            <div class="form-group">
                <label for="userName">Username</label>
                <input type="text" class="form-control" id="userName"
placeholder="Username" ng-model="user.name" required>
            </div>
            <div class="form-group">
                <label for="userMail">E-Mail</label>
                <input type="text" class="form-control" id="userMail"
placeholder="E-Mail" ng-model="user.email" required>
            </div>
            <button type="submit" class="btn btn-primary" ng-
class="{disabled: !user.name || !user.email}">
                Login {{user.name}}
                <span ng-show="user.email">@</span>
                {{user.email}}
            </button>
        </form>
    </div>
</div>
```

Abbildung 20 - View Template für Login Formular

In Abbildung 20 - View Template für Login Formular ist der HTML-Code des finalen Login-Formulars zu sehen. In diesem Beispiel werden schon einige AngularJS Funktionen benutzt, welche hier nochmals im Detail erklärt werden sollen.

Der Einsatz von Direktiven ist in AngularJS eine einfache Möglichkeit um vorhandene Elemente zu manipulieren oder funktional zu erweitern. AngularJS stellt eine Vielzahl von einsatzfähigen Direktiven, erkennbar an dem Prefix *ng-*, bereit, es lassen sich aber auch eigene Direktiven erstellen.

ng-hide ist eine Direktive, welche Elemente ausblendet wenn der übergebene Parameter als *wahr* ausgewertet wird. Das Login Formular nutzt diese Direktive um abzufragen ob der Nutzer schon angemeldet ist, und setzt für den div Container den CSS Stil `display: none!important` ein. Diese Direktive gibt es auch in der Variante *ng-show*, bei der ein Element nur dann gezeigt wird wenn der Parameter als *wahr* ausgewertet wird.

Das Formular selbst wird mit der Direktive *ng-submit* belegt. Wenn das Formular, durch ein Klick auf den Submit-Button oder durch Drücken der Enter-Taste, abgesendet wird, so wird die hier als Parameter hinterlegte Funktion aufgerufen.

Die 2 Input-Elemente werden mit der *ng-model* Direktive ausgestattet und bewirkt, dass die Inhalte der Input Elemente direkt in die hier deklarierten Models gespeichert werden. Der Controller erhält dann natürlich Zugriff auf diese Models und kann diese verarbeiten.

Als letzte hier verwendetete Direktive kommt *ng-class* zum Einsatz, welche ähnlich wie *ng-show* und *ng-hide* funktioniert, jedoch hier im Falle einer als wahr ausgewerteten Bedingung, die mitübergebene CSS-Klasse gesetzt wird. Der Submit-Button wird im Login-Formular mithilfe von *ng-class* erst dann aktiviert, wenn beide Input-Felder ausgefüllt wurden.

6.6 Technische Umsetzung ppSyncService

6.6.1 AngularJS Modul Aufbau

Der Aufbau des ppSyncService Moduls als Factory ist in ABBILDUNG zu sehen. Der Rückgabewert dieses Services ist ein Objekt mit Funktionen, die benötigt werden um mit der Datenbank zu interagieren. Funktionen und Variablen die nicht ausserhalb dieses Moduls zugänglich sein sollen, werden innerhalb der anonymen Funktion der Factory erstellt werden.

```
var ppSync = angular.module('ppSync', ['ng']);
ppSync.factory('ppSyncService', function($q, $window) {

    // Code zur Initialisierung der Klasse
    // oder private Funktionen

    return {
        // Objekt mit den bereitgestellten Funktionen
    }
});
```

Abbildung 21 - ppSyncService Struktur

6.6.2 Initialisierung (private)

Hier erfolgt die Initialsierung der von ppSyncService benötigten Variablen. Die Variable *dbname* enthält den Namen der Datenbank, welche zur Verwendung mit der PouchDB auf dem lokalen Gerät verwendet werden soll. Die Serveradresse eines CouchDB Servers befindet sich in der Variablen *remote*. Der lokale Datenbankname wird an die Serveradresse angehängt um die Datenbankbezeichnung auf Server und lokalen Gerät konsistent zu halten.

Das wohl wichtigste Objekt des Services, wird in der Variablen *db* gespeichert. Hier wird ein PouchDB Objekt mit dem vorher definierten Datenbanknamen erstellt. Die

Option Option *auto_compation: true* sorgt dafür das PouchDB die lokale Datenbank automatisch säubert.

```
var dbname = 'banet_default';
// var remote = 'http://127.0.0.1:5984/' + dbname;
var remote = 'http://philreinking.de:5984/' + dbname;
// Create PouchDB object
var db = new PouchDB(dbname, {auto_compaction: true});
```

Abbildung 22 - ppSyncService Initialisierung

6.6.3 syncFromRemote (private)

Die Funktion syncFromRemote repliziert die Daten des Servers in die lokale Datenbank. Dazu werden innerhalb von syncFromRemote zwei weitere Funktionen deklariert, um den Prozess wie in der Planung durchführen zu können.

Dabei ist *syncFilter* die Filterfunktion, welche in den Optionen der initialen Replikation übergeben wird, um nur die Beiträge der letzten 24 Stunden zu laden. In *syncFilter* wird ein Unix-Timestamp *timeBarrier* erstellt, der genau 24 Stunden alt ist. Danach wird überprüft ob der Timestamp des Dokuments größer ist, als der zuvor Berechnete Wert und gibt in diesem Fall *true* zurück, was den Replikationsprozess dazu veranlasst das Dokument zu replizieren.

Die *continuousSync* Funktion wird in der Option *complete* übergeben und gestartet wenn *PouchDB.replicate* beendet ist. In*continuousSync* findet ebenfalls ein Aufruf von *PouchDB.replicate* statt, jedoch ohne die zuvor verwendeten Optionen filter und complete, aber mit *continuous: true*.

```
var syncFromRemote = function() {
    // Fiterfunktion 24 Stunden
    var syncFilter = function(doc) {
        var timeBarrier = Date.now() - (86400 * 1000);
        return doc.created > timeBarrier ? true : false;
    };
    // Funktion zur dauerhaften Synchronisierung
    var continuousSync = function() {
        setTimeout(function() {
            PouchDB.replicate(remote, dbname, {continuous: true});
        }, 1000);
    };
    // Initiale Replikation mit Filter
    PouchDB.replicate(remote, dbname, {
        continuous: false,
        filter: syncFilter,
        complete: continuousSync
    });
}; syncFromRemote();
```

Abbildung 23 - ppSyncService Funktion syncFromRemote

6.6.5 monitorNetwork (private)

Um festzustellen wann die Applikation Online oder Offline ist, wird für ppSyncService eine Variable erstellt die den aktuellen Status beinhaltet und über die Funktion monitorNetwork aktualisiert wird.

Es werden innerhalb dieser Funktion EventListener für die Events *offline* und *online* registriert, die beim Eintreten dieser Events die Variable network mit dem entsprechenden Status versehen.

Beim Event *online* werden die Funktionen syncCache und syncFromRemote aufgerufen um den Replikationsprozess neuzustarten, welche ohne aktive Verbindung zuvor beendet wurde.

```
var network = 'online';
var monitorNetwork = function() {
    // Prüfe den Verbindungsstatus zum Netzwerk
    if ('onLine' in navigator) {
        $window.addEventListener('offline', function() {
            network = 'offline';
        });
        $window.addEventListener('online', function() {
            network = 'online';
            syncCache(); // Push Cache
            syncFromRemote(); // Restart Replication
        });
    }
}; monitorNetwork();
```

Abbildung 24 - ppSyncService Funktion monitorNetwork

6.6.6 Cache (private)

Der Cache ist als Objekt mit eigenen Methoden angelegt, um die Verwaltung des Caches so einfach wie möglich zu machen. Das Attribut docs ist ein Array, in welches die Ids der Dokumente abgelegt werden können, die zu einem späteren Zeitpunkt repliziert werden sollen.

Die Methode addDoc erwartet eine Id als Parameter und speichert diese in das Array. Um den Cache dauerhaft zu speichern, wird dieses Array im LocalStorage abgelegt.
Mit getDocs wird das Array aus dem LocalStorage zurückgegeben und reset löscht den Cache.

```
function Cache() {
    this.docs = [];
}
Cache.prototype = {
    addDoc: function(docId) {
        if (JSON.parse(localStorage.getItem('cache')) !== null) {
            this.docs = JSON.parse(localStorage.getItem('cache'));
        }
        this.docs.push(docId);
        localStorage.setItem('cache', JSON.stringify(this.docs));
    },
    getDocs: function(){
        return JSON.parse(localStorage.getItem('cache'));
    },
    reset: function() {
        this.docs = [];
        localStorage.setItem('cache', JSON.stringify(this.docs));
    }
};
var cache = new Cache();
```

Abbildung 25 - ppSyncService Objekt Cache

Desweiteren gibt es noch die Funktion syncCache, welche einen Replikationsprozess startet und nur die Dokumente zum Server synchronisiert, die sich im Cache befinden. Am Ende dieses Prozesses wird überprüft ob alle Dokumente hochgeladen wurden und der Cache geleert.

```
var syncCache = function() {
    db.replicate.to(remote, {
        doc_ids: cache.getDocs(),
        complete: function(err, response) {
            if (response.docs_read === cache.getDocs().length) {
                cache.reset();
            }
        }
    });
};
```

Abbildung 26 - ppSyncService Funktion syncCache

6.6.7 fetchChanges (public)

syncFromRemote repliziert dauerhaft neue Daten vom Server in die lokale Datenbank. Um diese neuen Daten zu verwerten, wird die Funktion fetchChanges zur Verfügung gestellt. Über *PouchDB.changes()* wird ein Prozess gestartet der bei jeder neuen Änderung in der lokalen Datenbank die *onChange* Funktion auslöst.

Der Rückgabewert von fetchChanges ist ein mithilfe des AngularJS Services $q erstellter Promise. Über *deferred.notify(change)*wird die registrierte Änderung über den Promise an den Funktionsaufruf weitergeleitet und kann dort weiterverarbeitet werden.

```
fetchChanges: function() {
    var deferred = $q.defer();
    db.changes({
        continuous: true,
        since: 'latest', // Nur neue Änderungen
        include_docs: true, // Die Inhalte der Dokumente einbinden
        complete: function(err, changes) {
            deferred.resolve(changes);
            deferred.reject(err);
        },
        onChange: function(change) {
            deferred.notify(change);
        }
    });
    return deferred.promise;
}
```

Abbildung 27 - ppSyncService Funktion fetchChanges

6.6.9 postDocument (public)

Mit postDocument kann ein Dokument in der Datenbank gespeichert werden. Die Funktion erwartet ein JSON-Objekt als Parameter.

Mit *PouchDB.post(obj)* wird das Dokument zunächst in die lokale Datenbank gespeichert. In der Callback Funktion *then* wird dann der aktuelle Status der Netzwerkverbindung überprüft. Im Falle, dass das Gerät online ist, wird das neu gespeicherte Dokument zum Server repliziert. Ist das Gerät jedoch offline, muss die Id des neuen Dokuments über *cache.addDoc()* in den Cache gespeichert werden.

Der Rückgabewert von postDocument ist ein Promise welche bei Erfolg die id des neuen Dokuments enthält.

```
postDocument: function(obj) {
    var deferred = $q.defer();
    db.post(obj).then(function(response) {
        if (network === 'online') {
            db.replicate.to(remote, {
                doc_ids: [response.id],
                complete: function() {
                    deferred.resolve(response.id);
                }
            });
        } else {
            deferred.resolve(response.id);
            cache.addDoc(response.id);
        }
    }).catch (function(error) {
         deferred.reject(error);
    });
    return deferred.promise;
}
```

Abbildung 28 - ppSyncService Funktion postDocument

6.6.10 deleteDocument (public)

Mit deleteDocument kann ein Dokument aus der Datenbank gelöscht werden. Damit diese Löschung auch auf anderen Geräten registriert wird, kann das Dokument nur als *deleted* markiert werden, um dann auf den Server synchronisiert zu werden.

Als Parameter wird das zu löschende Dokument mit *id* und *rev* erwartet und optional ein boolescher Wert ob die Änderung im Cache gespeichert oder direkt synchronisiert werden soll.

Das Löschen wird mit *PouchDB.remove()* durchgeführt. Wie beim Erstellen eines Dokuments erfolgt in der Callback Funktion eine Abfrage, ob die Änderungen direkt auf den Server repliziert werden kann, oder aber ob das Dokument im Cache gespeichert werden muss.

```
deleteDocument: function(doc, push) {
    var deferred = $q.defer();
    push = typeof push !== 'undefined' ? push : true;
    db.remove(doc, {}, function(err, response) {
        if (push) {
            db.replicate.to(remote, {
                doc_ids: [response.id],
                complete: function() {
                    deferred.resolve(response.id);
                }
            });
        } else {
            cache.addDoc(response.id);
            deferred.resolve(response.id);
        }
    });
    return deferred.promise;
}
```

Abbildung 29 - ppSyncService Funktion deleteDocument

6.6.11 getDocument (public)

Mit dieser Funktion wird ein einzelnes Dokument aus der Datenbank geladen. Als einziger Parameter muss die entsprechende Id des Dokuments übergeben werden.

Der Rückgabewert ist ein Promise mit dem vollständigen Dokument als Inhalt.

```
getDocument: function(docId) {
    var deferred = $q.defer();
    db.get(docId).then(function(doc) {
        deferred.resolve(doc);
    });
    return deferred.promise;
}
```

Abbildung 30 - ppSyncService Funktion getDocument

6.6.12 getDocuments (public)

Um alle Dokumente aus der Datenbank zu laden, wird die Funktion getDocuments benötigt. Als Parameter kann ein Array mit den Dokumententypen übergeben werden, die ausgelesen werden sollen. Die Reihenfolge der Werte in dem Array bestimmt auch die Reihenfolge in welcher die Dokumente mithilfe der Map-Funktion angeordnet werden.

documentTypes = ['POST', 'COMMENT'] würde zur Folge haben das Dokumente vom Typ *POST* einen key in der Form [doc.id, 0] und Dokumente vom Typ 'COMMENT' einen key in der Form [posting.id, 1] zugewiesen bekommen.
Für das Auslesen wird *PouchDB.query()* verwendet. Die Optionen in *queryOptions* bewirken, dass die Dokumente mit Inhalten und in absteigender Reihenfolge geladen werden.

Der Rückgabewert ist ein Promise mit einem Array der ausgelesen Dokumente.

```
getDocuments: function(documentTypes) {
    var deferred = $q.defer();

    documentTypes = typeof documentTypes !== 'undefined' ? documentTypes :
'uncategorized';

    var queryOptions = {
        descending: true,
        include_docs: true,
    };

    db.query(function(doc, emit) {
        if (documentTypes === 'uncategorized') {
            emit([doc, 0], doc.created);
        } else {
        // Jeder Typ aus dem Array wird mit dem aktuellen Dokument
verglichen
            for (var i = 0; i < documentTypes.length; i++) {
                 if (doc.type === documentTypes[i]) {
                    if (doc.type === 'POST') {
                       emit([doc._id, i], doc.created);
                    } else {
                       emit([doc.posting, i], doc.created);
                    }
                    break;
                 }
            }
        }
    }, queryOptions, function(error, response) {
        deferred.resolve(response.rows);
    });

    return deferred.promise;
}
```

Abbildung 31 - ppSyncService Funktion getDocuments

6.6.13 getChannelStream (public)

Die Funktion getChannelStream ist eine alternative Implementierung zu getDocuments. Zum einen werden nur Dokumente geladen, die dem Wert im Parameter *channel* entsprechen. Zusätzlich ist es über numberOfPosts möglich, die Anzahl der Dokumente zu begrenzen und mit end kann ein key bestimmt werden ab dem Dokumente geladen werden. Damit ist es möglich eine Paginierung zu implementieren.

In den queryOptions befinden sich die zwei zusätzliche Optionen zur Eingrenzung der Dokumente.

```
getChannelStream: function(channel, numberOfPosts, end) {
    var deferred = $q.defer();

    numberOfPosts = typeof numberOfPosts !== 'undefined' ? numberOfPosts :
50;
    end = typeof end !== 'undefined' ? end : [Date.now(), 0];
    channel = typeof channel !== 'undefined' ? channel : 'uncategorized';

    var queryOptions = {
        descending: true,
        limit: numberOfPosts,
        include_docs: true,
        endkey: end
    };

    db.query(function(doc, emit) {
        if (doc.channel === channel) {
            emit([doc.created, 0], doc.created);
        }
    }, queryOptions, function(error, response) {
        deferred.resolve(response.rows);
    });
    return deferred.promise;
}
```

Abbildung 32 - ppSyncService Funktion getChannelStream

6.6.14 syncCache (public)

Mit syncCache kann der Mechanismus zum Synchronisieren des Caches manuell ausgelöst werden.

```
syncCache: function() {
    syncCache();
}
```

Abbildung 33 - ppSyncService Funktion syncCache

6.6.15 debug (public)

Die Funktion debug gibt mithilfe von *PouchDB.info()* aktuelle Informationen über die lokale Datenbank zurück.

```
debug: function() {
    var deferred = $q.defer();
    db.info().then(function(response) {
        deferred.resolve(response);   }).catch (function(error) {
            console.log(error);
        });
    return deferred.promise;
}
```

Abbildung 34 - ppSyncService Funktion debug

6.6.16 reset (public)

Zum Löschen der lokalen Datenbank kann die Funktion reset aufgerufen werden. Die Datenbank wird zerstört und ein neues PouchDB Objekt erstellt.

```
reset: function() {
    PouchDB.destroy(dbname);
    db = new PouchDB(dbname);
}
```

Abbildung 35 - ppSyncService Funktion reset

6.7 Technische Umsetzung Dashboard

Controller werden mit dem in der Applikation genutzen AngularJS Modul *banetApp* erstellt. Dieses Modul besitzt die Funktion*controller()*, welche genau 2 Parameter erwartet. Der erste Parameter ist der Name des Controllers, welcher im Gültigkeitsbereich von *banetApp* eindeutig sein muss. Der zweite Parameter ist eine anonyme Funktion, die den Code für den Controller enthält. Zusätzlich können in dieser anonymen Funktion AngularJS Services übergeben werden, die dem Controller dann zur Verfügung stehen.

Das Scope-Objekt ist eines der wichtigsten Referenzen, denn darüber erhält der Controller Zugriff auf den Gültigkeitsbereich der zu steuernden View. Models und Funktionen die in der View verfügbar sind, werden in *$scope* abgelegt.

```
angular.module('banetApp')
    .controller('DashboardCtrl',function($scope, ppSyncService, dummyContent,
simpleUser) {
        // Controller Code
        $scope.hello = 'Hello World!';
    };
```

Abbildung 36 - AngularJS Controller Struktur

Der Dashboard Controller enthält insgesamt 4 Funktionen, welche zur Verwaltung des Benutzers benötigt werden. *simpleUser* ist dabei ein einfacher Service, der die Login Daten des Benutzers ohne Verifizierung im LocalStorage speichert. Möchte man eine richtige Authentifizierung erreichen, muss man den Service simpleUser ersetzen.

In der Funktion *loadUserData()* werden die Benutzerdaten ausgelesen und in der Variablen *$scope.user* gespeichert. Die View kann dann über *user* auf diese Daten zugreifen.

login() speichert die Daten, die sich aktuell in der Variablen *$scope.user* befinden und loggt den Benutzer damit ein.

logout() löscht die Login Daten des Benutzers.

userIsOnline() gibt einen booleschen Wert zurück, ob der Benutzer eingeloggt ist oder nicht.

```
$scope.loadUserData = function() {
    if ($scope.userIsOnline()) {
        $scope.user = simpleUser.getUserData();
    }
};
$scope.login = function() {
    // Set the online flag to true
    $scope.user.online = true;
    simpleUser.login($scope.user);
};
$scope.logout = function() {
    $scope.user = simpleUser.logout();
};
$scope.userIsOnline = function() {
    return simpleUser.isOnline();
};
```

Abbildung 37 - Dashboard Controller Funktionen

In Abbildung 38 wird ersichtlich wie mithilfe von AngularJS Expression die im Controller zur Verfügung gestellten Benutzerdaten ausgegeben werden können. Alles was innerhalb einer *{{ expression }}* steht, wird von AngularJS als JavaScript interpretiert und durch das ausgewertete Ergebnis dieser Expression ersetzt.

Mit *ng-show="userIsOnline()"* wird sichergestellt, dass nur eingeloggte Benutzer diese Informationen angezeigt bekommen.

```
<div ng-show="userIsOnline()">
    <p>Your are logged in as <strong>{{user.name}}</strong> with the id
<strong>{{user.id}}</strong>.</p>
</div>
```

Abbildung 38 - Darstellung von Benutzerdaten in der Dashboard View

Das Pinboard ist nach dem Dashboard die zweite Komponente und demonstriert die Funktionsweise der Datenbank und Synchronisierung.

Der Code des Controllers beinhaltet lediglich 2 Funktionsaufrufe und stellt eine Funktion zum Löschen der Beiträge bereit. Dadurch das die Logik für die Datenbank komplett im Service ppSyncService ausgelagert wurde, ist der Controller somit sehr übersichtlich.

Mit *ppSyncService.getDocuments()* werden alle Dokumente vom Typ *POST* angefragt. Der Rückgabewert dieses Aufrufs wird über die Promise Implementierung abgefangen und dann in *$scope.pins* gespeichert. Die weitere Verarbeitung kann dann in der View erfolgen.

Änderungen an der Datenbank werden mit *ppSyncService.fetchChanges()* registriert und über den Notify Promise abgefangen. Es muss dann an dieser Stelle noch überprüft werden ob das neue Dokument nicht gelöscht wurde und vom Typ *POST* ist, bevor es in das Array *$scope.pins* geschoben wird.

Um ein Dokument zu löschen, wird die Funktion *deleteDocument()* deklariert. Diese Funktion benutzt*ppSyncService.deleteDocument()* und erwartet das zu löschene Dokument als Parameter.

```
angular.module('banetApp')
    .controller('PinboardCtrl', function($scope, ppSyncService) {
        ppSyncService.getDocuments(['POST']).then(function(response){
            $scope.pins = response;
        });
        ppSyncService.fetchChanges().then(function(response) {
            console.log(response);
        }, function(error) {
            console.log(error);
        }, function(change) {
            if ($scope.pins) {
                if (!change.deleted && change.doc.type === 'POST') {
                  return $scope.pins.unshift(change);
                }
            }
        });
        $scope.deleteDocument = function(pin) {
            ppSyncService.deleteDocument(pin.doc);
        };
    });
```

Abbildung 39 - Pinboard Controller

In der View werden Direktiven benutzt um die im Array *$scope.pins* gespeicherten Dokumente auszugeben.

Die Aufgabe der View ist es die Daten aus dem Array *$scope.pins* optisch ansprechend auszugeben. AngularJS bietet zur Ausgabe von Arrays die sehr nützliche Direktive *ngRepeat*, welche für jedes Element in einem Array die HTML-Vorlage einmal ausgibt. Jedes Element erhält dabei seinen eigenen Scope und kann so die anderen Elemente nicht beeinflussen.

Zusätzlich kommt in *ngRepeat* der Filter *orderBy* zum Einsatz, der es ermöglicht das Array nach dem timestamp zu sortieren.

Da der Benutzer bei der Anmeldung eine E-Mail Adresse angeben kann, wird über den Internetservice Gravatar versucht ein Avatar für den Benutzer zu finden. Die Direktive *gravatar-image* enthält das Attribut *email*, in dem über *pin.doc.user.email* die Email des Autoren ausgelesen wird. Die Umwandlung zu einem gültigen HTML Image Tag geschieht innerhalb der Direktive.

```
<div class="pin ng-cloak" ng-repeat="pin in pins |
orderBy:'doc.created':true" ng-hide="pin.deleted">
    <div class="media">
        <a class="pull-left" href="#/pinboard/show/{{pin.id}}">
            <gravatar-image email="pin.doc.user.email" />
        </a>
        <div class="media-body">
            <h5 class="media-heading">
                <span class="glyphicon glyphicon-user"></span>
                {{pin.doc.user.name}}
                <small>{{pin.doc.created | date:'dd. MMMM yyyy @
HH:mm:ss'}}</small>
                <button type="button" class="btn btn-xs btn-link pull-right"
ng-click="deleteDocument(pin); pin.deleted = true">
                    delete
                </button>
            </h5>
            <p class="lead">{{pin.doc.msg}}</p>
        </div>
    </div>
</div>
```

Abbildung 40 - Template zur Ausgabe der Pinboard Beiträge

Zum Erstellen eines neuen Beitrags wird einen separate Seite mit dem Controller *PinboardNewCtrl* aufgerufen. Das Attribut isSubmitting wird dazu verwendet um der View mitzuteilen ob ein Beitrag gerade in die Datenbank gespeichert wird.

Die Funktion zum Speichern eines Beitrags *$scope.newPin()* erweitert das zu speichernde Objekt *pin* um einige wichtige Parameter wie den aktuellen Zeitstempel, Beitragstyp, und Benutzerdaten.

Wenn das Speichern des Dokuments erfolgreich war, erfolgt eine Umleitung zurück auf die Pinboard View.

```
angular.module('banetApp')
    .controller('PinboardNewCtrl', function($scope, $location, ppSyncService,
simpleUser) {
        $scope.isSubmitting = false;
        $scope.newPin = function() {
            $scope.isSubmitting = true;
            $scope.pin.created = Date.now();
            $scope.pin.user = simpleUser.getUserData();
            $scope.pin.type = 'POST';
            ppSyncService.postDocument($scope.pin).then(function() {
                $location.path('pinboard');
            });
        };
    });
```

Abbildung 41 - Pinboard New Controller

In der View erhält das Formular die Direktive *ngSubmit* um die Funktion *newPin()* auszulösen. Der Textarea wird mit *ngModel* die Variable zugewiesen, die von der *newPin()* Funktion ausgelesen wird.

```
<form ng-submit="newPin()">
    <div class="form-group">
        <label for="PinContentId">Content</label>
        <textarea class="form-control" id="PinContentId" rows="3" ng-
model="pin.msg" ng-disabled="isSubmitting" required></textarea>
    </div>
    <button type="submit" class="btn btn-primary" ng-
disabled="isSubmitting">Submit <i ng-show="isSubmitting" class="fa fa-spinner
fa-spin"></i></button>
</form>
```

Abbildung 42 - Formular zum Speichern eines neuen Beitrags

Der Channel ist eine zusätzliche Komponente, die man als eine Art Erweiterung der Pinboard Komponente betrachten kann.

Der Benutzer hat die Möglichkeit einen beliebigen String in ein Input-Feld einzugeben um dadurch in einen Channel zu wechseln. Beiträge die in einem Channel erstellt werden, sind dann auch nur in diesem Channel sichtbar.

Zur Verwaltung des aktiven Channels wird das Objekt *channel* verwendet, welches den String mit Kleinbuchstaben und als lesbaren String mit großen Anfangsbuchstaben enthält.

Beiträge des aktiven Channels werden über den Funktionsaufruf *ppSyncService.getChannelStream()* geladen und in *$scope.entries* gespeichert.
Änderungen werden wie schon im Pinboard Controller über *ppSyncService.fetchChanges()* registriert.

Das Wechseln zu einem Channel erfolgt über die in der Applikation definierte Route für Channels */channel/:name*. *:name* ist hier ein Platzhalter für den Namen des Channels und wird dann über das Objekt *$routeParams* ausgelesen.

```
$scope.channel = {
    name: $routeParams.name.toLowerCase(),
    humanizedName: Helpertools.humanize($routeParams.name.toLowerCase())
};
$scope.switchChannel = function() {
    $location.path('channel/' + $scope.newChannel.toLowerCase());
};
```

Abbildung 43 - Auszug aus dem Channel Controller

```
<form ng-submit="switchChannel()">
    <div class="input-group input-group-lg">
        <span class="input-group-addon">#</span>
        <input type="text" class="form-control" ng-model="newChannel"
placeholder="Switch Channel">
        <span class="input-group-btn">
            <button class="btn btn-default" type="submit">Submit</button>
        </span>
    </div>
</form>
```

Abbildung 44 - Formular zum Ändern des aktiven Channels

46

8 Realisierung

Abbildungsverzeichnis