Hochschule Furtwangen University
Aktuelle Entwicklungen im Bereich Online Medien

Die Verwendung von MongoDB im Kontext einer Nachrichten-Aggregations-Anwendung

Moritz Göckel, MIB 5, 247243

Inhaltsverzeichnis

1. Zielsetzung und Thema

Diese Arbeit entstand im Rahmen der Vorlesung „Aktuelle Entwicklungen im Online-Bereich“ bei Herrn Prof. Eisenbiegler. Aufgabe ist die Evaluation einer Technologie, welche im Online-Bereich eingesetzt wird, anhand eines konkreten Anwendungsbeispiels.

Es wurden mehrere Technologien von Herrn Prof. Eisenbiegler vorgeschlagen, so unter anderem auch die Evaluation von NoSQL-Datenbanken. Ich entschied mich für dieses Thema.

Ich wählte die konkrete NoSQL-Datenbank MongoDB, da ich mit dieser Datenbank schon etwas Vorerfahrung hatte. Ein weiterer Grund, der für MongoDB spricht, ist der Fakt, dass MongoDB eine der bekanntesten und meist eingesetzten NoSQL-Datenbanken ist. Somit schien mir MongoDB ein guter Vertreter aller NoSQL-Datenbanken zu sein.

Bei der Beispielanwendung entschied ich mich für eine News-Aggregator Anwendung. Es sprachen drei Gründe für die Umsetzung dieser Anwendung:

- Ich habe schon einmal eine ähnliche Anwendung mit dem gleichen Ziel programmiert, war aber mit dem Ergebnis nicht zufrieden. Vor allem die Performance ließ damals zu wünschen übrig, darunter litt auch die Usability.
- Eine solche Anwendung ist extrem datenlastig. Daten sind letztendlich der Kern dieser Anwendung. Außerdem lässt sich sagen, dass extrem viele Daten anfallen und verarbeitet werden müssen. Deshalb ist diese Art von Anwendung sinnvoll für die Evaluierung einer Datenbanktechnologie.
- Einer der wichtigsten Gründe warum ich diese Anwendung entwickeln wollte, war ganz einfach, dass ich mir so eine Anwendung wünschen würde. Ich wollte, dass eine vergleichbare Anwendung existiert, konnte aber bei meinen Recherchen keine finden. Deshalb wollte ich sie entwickeln.

Auch wenn ich sowohl mit MongoDB schon gearbeitet habe als auch eine ähnliche Anwendung schon gebaut habe, konnte ich so gut wie keinen Code wiederverwenden. Das liegt vor allem daran, dass ich dieses Projekt wesentlich sauberer und besser umsetzen wollte.

Um diese „Sauberkeit“ des Codes und eine hohe Qualität der Anwendung zu erzielen, entwickelte ich eine neue und sehr viel komplexere Architektur für die Anwendung und verwendete andere Technologien wie beim ersten Mal. Deshalb ließ sich so gut wie kein Code wiederverwenden.

2. NoSQL im Unterschied zu SQL

MongoDB ist eine NoSQL-Datenbank, was für „Not Only SQL“ steht. Der entscheidende Unterschied zwischen NoSQL- und SQL-Datenbanken ist die Datenbankstruktur. Bei SQL-Datenbanken ist die genaue Struktur jeder Tabelle festgelegt. Jede Spalte hat einen eindeutigen Datentyp und klare Vorgaben was darin gespeichert werden kann. Diese

festgelegte Struktur haben die NoSQL-Datenbanken nicht. Es wird im Voraus keine Struktur der Tabelle angelegt, sondern man kann von Anfang an Daten mit einer beliebigen Struktur in die Tabelle schreiben. So können auch Daten mit verschiedenster Struktur in die gleiche Tabelle geschrieben werden.

Inserts und Selects

Wegen dieser Strukturlosigkeit muss dafür beim Auslesen der Daten auf mehr geachtet werden, da man niemals wissen kann, was für eine Struktur die gerade gelesenen Daten haben. Auf der anderen Seite darf man alles einfügen, was immer man möchte. Ganz im Gegenteil zu SQL-Datenbanken.

Integrität und Relationen

Da NoSQL-Datenbanken so lax mit ihrer Struktur umgehen, können sie keine Integrität der Daten garantieren. So lassen sich Relationen zwar manuell programmieren, die Datenbank wird diese aber nie kontrollieren. Man kann also über IDs von einem Objekt zum anderen zeigen lassen und dann über den Programmcode erst das eine und dann das andere Objekt auslesen. In SQL-Datenbanken könnte man für diesen Vorgang einfach Joins verwenden, in NoSQL-Datenbanken muss man dies manuell machen. Auch beim Löschen eines dieser Objekte muss man die Verlinkungen selbst managen.

Kurz: NoSQL-Datenbanken können weder für die Integrität der Daten garantieren, noch beherrschen sie Relationen. Diese können aber selbst programmiert werden.

Atomizität und Transaktionen

NoSQL-Datenbanken achten auf die Atomizität der Transaktionen nur im Bezug auf ein einzelnes Dokument. Transaktionen die mehrere Dokumente betreffen sind nicht möglich.

Index und Unique

Wie ganz normale SQL-Datenbanken können auch NoSQL-Datenbanken ihre Objekte redundant abspeichern, um spätere Suchen zu beschleunigen. Dies wird sowohl in SQL- als auch in NoSQL-Datenbanken via Indizes erreicht.

Einen Unterschied gibt es hier allerdings doch: In SQL-Datenbanken werden die Indizes von Anfang an festgelegt, und jedes Objekt muss einen solchen Index-Wert beinhalten. NoSQL-Datenbanken fügen im Nachhinein auf Befehl einen Index ein, und Objekte ohne einen entsprechenden Wert werden einfach nicht beachtet.

Unique-Values sind ebenfalls möglich. Diese werden dann wie Indizes im Nachhinein pro Collection festgelgt.

3. NoSQL-Datenbanken

Es gibt viele verschiedene Arten von NoSQL-Datenbanken. Jede dieser Arten von NoSQL-Datenbanken hat wieder mehrere Vertreter von tatsächlichen Datenbank-Softwares:

Merkmal		Beispiele
Dokumentenorientierte Datenbanken		Apache Jackrabbit, BaseX, CouchDB, eXist, IBM Notes, MongoDB, OrientDB
Graphdatenbanken	Generisch	Neo4j, OrientDB, InfoGrid, HyperGraphDB, Core Data, DEX
	RDF-Zentriert	AllegroGraph, 4store, andere
Verteilte ACID-Datenbanken		MySQL Cluster
Key-Value-Datenbanken	Festplattenspeicher	Chordless, Google BigTable, GT.M, InterSystems Caché
	Caches im RAM	Membase, memcached, Redis, Aerospike
	Eventually-consistente Speicher	Amazon Dynamo, Project Voldemort, Riak
	Sortierte Key-Value-Speicher	Berkeley DB, Memcachedb
Multivalue-Datenbanken		OpenQM, Rocket U2
Objektdatenbanken		Db4o, ZODB
Spaltenorientierte Datenbanken		Apache Cassandra, Google BigTable, HBase, SimpleDB

(Quelle: Wikipedia - https://de.wikipedia.org/wiki/NoSQL)

- Dokumentenorientierte Datenbanken sind besonders gut für komplexe Daten-Strukturen und viele Daten. Die NoSQL-Datenbank, welche ich evaluiere, also MongoDB, gehört zu dieser Kategorie.
- Graphendatenbanken sind besonders gut geeignet für die Darstellung von Beziehungen von Objekten.
- Key-Value-Datenbanken sind besonders geeignet für die Speicherung von simplen Daten-Strukturen und können mit ihrer Geschwindigkeit punkten.
- Spaltenorientierte Datenbanken, wie Facebooks Cassandra, sind für das schnelle Zusammenrechnen der Datensätze optimiert.
- Und es gibt noch mehr Arten...

NoSQL-Datenbanken werden seit ein paar Jahren immer beliebter und sind aus vielen Anwendungen nicht mehr weg zu denken. Sie werden meistens da verwendet, wo viele Daten anfallen und Performance benötigt wird.

Unter den ersten zehn Plätzen des Database-Rankings finden sich sieben SQL- und drei NoSQL-Datenbanken. Auf Platz vier steht als beliebteste NoSQL-Datenbank die dokumentenorientierte MongoDB, auf Platz sieben steht Cassandra, welche von Facebook verwendet wird, und auf Platz zehn steht Redis, eine Key-Value-Datenbank.

305 systems in ranking, June 2016

Rank Jun 2016	Rank May 2016	Rank Jun 2015	DBMS	Database Model	Score Jun 2016	Score May 2016	Score Jun 2015
1.	1.	1.	Oracle	Relational DBMS	1449.25	-12.78	-17.11
2.	2.	2.	MySQL	Relational DBMS	1370.13	-1.69	+91.78
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1165.81	+22.99	+47.76
4.	4.	↑ 5.	MongoDB	Document store	314.62	-5.60	+35.57
5.	5.	↓ 4.	PostgreSQL	Relational DBMS	306.60	-1.01	+25.70
6.	6.	6.	DB2	Relational DBMS	188.57	+2.61	-10.12
7.	7.	↑ 8.	Cassandra	Wide column store	131.12	-3.38	+22.21
8.	8.	↓ 7.	Microsoft Access	Relational DBMS	126.22	-5.35	-20.27
9.	↑ 10.	9.	SQLite	Relational DBMS	106.78	-0.48	-1.19
10.	↓ 9.	10.	Redis	Key-value store	104.49	-3.74	+9.00
11.	11.	↑ 14.	Elasticsearch	Search engine	87.41	+1.10	+17.32
12.	12.	↑ 13.	Teradata	Relational DBMS	73.84	+0.09	+0.75
13.	13.	↓ 11.	SAP Adaptive Server	Relational DBMS	71.68	+0.21	-16.49
14.	14.	↓ 12.	Solr	Search engine	64.07	-1.55	-17.18
15.	15.	15.	HBase	Wide column store	52.99	+1.15	-8.71

(Quelle: http://db-engines.com/en/ranking)

Allgemein werden NoSQL-Datenbanken meist mit ihrer starken Performance, ihrer großen Geschwindigkeit beworben. Diese Aussagen sind aber mit Vorsicht zu genießen, da bei gleichem Einsatz NoSQL-Datenbanken nicht per se schneller sind als SQL-Datenbanken. Es lassen sich jedoch große Geschwindigkeitsvorteile gegenüber SQL-Datenbanken erreichen wenn man zwei Vorteile nutzt:

- NoSQL-Datenbanken lassen sich wunderbar parallelisieren. Soll heißen, mehrere Server betreiben eine Datenbank und teilen sich die Rechenlast. So lassen sich extrem viele Anfragen schnell beantworten. Dies wäre bei den komplexen Datenstrukturen und der Transaktionssicherheit von SQL-Datenbanken nicht so leicht umzusetzen.
- Joins sind in NoSQL-Datenbanken ja nicht möglich, lassen sich aber durch die Freiheit in der Datenstruktur auch umgehen. Die in einer SQL-Datenbank durch Joins verknüpften Daten lassen sich in NoSQL-Datenbanken einfach direkt in das gleiche Dokument, an die gleiche Stelle speichern. Dadurch ist die Abfrage um ein Vielfaches einfacher und schneller.

Diese beiden Optimierungen in der Performance sind nur durch den laxen Umgang mit der Datenstruktur und durch den Ausschluss von Datenintegrität und Transaktionssicherheit zu erreichen. Wir zahlen also einen hohen Preis für die Geschwindigkeit.

Es lässt sich also sagen, dass NoSQL-Datenbanken nicht besser sind als SQL-Datenbanken und diese auch niemals ersetzen würden. Sie sind einfach für einen anderen Zweck entwickelt und bilden damit keine direkte Konkurrenz zu SQL-Datenbanken.

4. MongoDB

MongoDB ist eine NoSQL-Datenbank auf Basis einer Dokumenten-Struktur. Die Verwendung von MongoDB ist kostenlos, und die Software kann man unter https://www.mongodb.com herunterladen.

MongoDB ist der Marktführer unter den NoSQL-Datenbanken, soll heißen, es ist die meist verwendete. Die Datenbank wird Open Source entwickelt und der Name kommt von Humongous, also „Extrem groß“.

Die Hauptmerkmale von MongoDB sind seine Skalierbarkeit und sein Dokumentenmodell mit dynamischem Schema. Damit wollen die Macher eine schnellere Softwareentwicklung ermöglichen und eine Möglichkeit zur Speicherung von extrem vielen oder großen Daten anbieten. Da MongoDB kostenlos ist, finanzieren sich die Entwickler durch Consulting und Schulungen zum Thema MongoDB.

Firmen, die MongoDB verwenden, sind unter anderem die Amerikanische Regierung, die Britische Regierung, BuzzFeed, Ebay, Adobe, McAfee, eHarmony, CitiGroup und LinkedIn. Der Finanzdienstleister MetLife beschreibt auf der Website von MongoDB, wie MongoDB half einen riesige Datenbank für Kunden anzulegen:

> “MetLife built a single view of 100M+ customers across 70 different systems in just 90 days using MongoDB. It had been trying for 8 years to build the same application with a relational database. Customer service just got a lot better.“

5. Bson

MonogoDB basiert auf einer grundlegenden Dokumentenstruktur. Als Basis wird Bson, die binär codierte Version von Json verwendet. Jeder Datensatz, den man einspeichern möchte, muss also im Bson-Format vorliegen, und jeder gelesene Datensatz wird als Bson ausgeliefert.

Die größte Einheit in MongoDB ist die Datenbank. Jeder Datenbankserver kann mehrere Datenbanken halten. Eine Datenbank enthält wiederum mehrere Collections, so etwas wie Tabellen in SQL-Datenbanken. Diese Tabellen wiederum beinhalten Dokumente im Bson-Format. Diese Bson-Dokumente sind die kleinste Einheit in MongoDB. Alle Befehle wie SELECT, REMOVE oder UPDATE beziehen sich immer auf die Dokumente. Ein Zugriff auf eines dieser Dokumente ist unteilbar. Es wird immer das ganze Dokument bei einem Insert eingefügt oder bei einem Select ausgeliefert.

Der Aufbau dieser Dokumente ist eine Key-Value-Struktur. Dabei sind die Keys immer Strings, und die Values können eine der verschiedenen MongoDB-internen Datentypen sein. Jedes Dokument besitzt zwangsweise eine ID mit dem MongoDB-eigenen Typ ObjectID. Diese ID wird beim INSERT automatisch zugewiesen.

Die MongoDB Datentypen sind unter anderem:

- Double
- String
- Object
- Array
- Binary data
- ObjectId
- Boolean
- Date
- Null
- Regular Expression
- Timestamp
- ...

Ein wichtiger Unterschied zwischen den MongoDB-Datentypen und den gängigen SQL-Datentypen sind die Typen ObjectID, Array und Object.

ObjectID: Die ObjectID in MongoDB wird automatisch vom Datenbankserver beim Insert generiert. Diese ID ist eine Kombination aus Integern und Chars und ist eindeutig für jedes Objekt in der Datenbank.

Beispiel:
```
{ "_id" : ObjectId("54ee1f2d33335326d70987df") }
```

Array: In MongoDB lassen sich ganz einfach Arrays von wiederum jedem beliebigen Datentyp abspeichern. Das ist in SQL-Datenbanken nicht so einfach möglich. In NoSQL-Datenbanken müsste man eine weitere Tabelle zur Speicherung von Array-Werten erstellen, und müsste bei der Abfrage beide Tabellen durchsuchen. Durch dieses Feature in MongoDB lassen sich ganz einfach, schnell, und vor allem performant 1:n-Beziehungen beschreiben. Diese Beziehungen können sowohl zwischen Objekten und simplen Datentypen (ein Buch hat mehrere Titel) oder zwischen Objekten und anderen Objekten (ein Autor hat mehrere Bücher, welche über die jeweilige ObjectID angesprochen werden) bestehen.

Beispiele:
```
{ "Titel" : [ "Lorem", "Ipsum", "Dolor", "Sit" ] }

{ "Buecher" : [ ObjectId("67ed1y...7df"), ObjectId("523sd...g4f"), ObjectId("gsd34...af3") ] }
```

Object: Ein Objekt ist eine Sammlung von Key-Value-Paaren. Damit ist auch ein Bson-Dokument ein Objekt. Spannend ist, dass ein Key-Value-Paar wiederum ein Objekt als Value haben kann. Damit lassen sich also ganze Objekte in andere Objekte einbinden. Liest man das eine Objekt aus, so bekommt man das andere auch direkt geliefert. Dies ließe sich in SQL-Datenbanken über mehrere Tabellen und einen JOIN bewerkstelligen, wäre aber um ein Vielfaches langsamer.

Beispiele:
```
{ "Buch" : { "Titel" : [ "Lorem", "Ipsum", "Dolor", "Sit" ] }}

{ "Buecher" : [
        { "Titel" : [ "Lorem", "Ipsum", "Dolor", "Sit" ] },
        { "Titel" : [ "Lorem", "Ipsum", "Dolor", "Sit" ] },
        { "Titel" : [ "Lorem", "Ipsum", "Dolor", "Sit" ] }
] }
```

Hier ein umfangreicheres Beispiel für ein komplexes Bson-Dokument. Es enthält eine ObjectId, ein Namens-Objekt, ein String-Array und ein Objekt-Array.

Beispiel:

```
{
   '_id' : ObjectId("67ed1y...7df"),
   'name' : { 'first' : 'John', 'last' : 'Backus' },
   'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
   'awards' : [
      {
         'award' : 'W.W. McDowell Award',
         'year' : 1967,
         'by' : 'IEEE Computer Society'
      }, {
         'award' : 'Draper Prize',
         'year' : 1993,
         'by' : 'National Academy of Engineering'
      }
   ]
}
```

(Quelle: MongoDB.com)

6. Einrichten von MongoDB

MongoDB ist kostenlos zum Download erhältlich unter www.mongodb.com/download-center. Dort erhält man einige Dateien zur Administration der Datenbank und am wichtigsten: Den Datenbankserver selbst. Zum Start des Servers muss man die Datei mongod.exe in der Commandozeile ausführen. Als Parameter lohnt es sich, den Pfad der Datenbankdatei zu definieren. Der Befehl, um den Server zu starten, könnte beispielsweise so aussehen:

```
C://.../mongod.exe –dbpath C://.../datenbank/
```

Nun läuft die Datenbank, und man kann sich zu ihr verbinden. Ich habe mich meist auf zweierlei Weisen mit dem Server verbunden: Einerseits mit dem Admininterface Robomongo, auf welches ich im Kapitel „Administrieren von MongoDB“ eingehe, und andererseits mit meiner selbst programmierten Anwendung.

Um von einer Anwendung aus auf den Datenbankserver zugreifen zu können, ist es notwendig den Treiber für die entsprechende Programmiersprache herunterzuladen. Es gibt offizielle Treiber für die folgenden Programmiersprachen:

- C
- C++
- C#
- Java
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

(Quelle: https://docs.mongodb.com/ecosystem/drivers/)

Da ich meine Anwendung in ASP.net, also in C# schreiben wollte, habe ich mich mit dem C#- Treiber beschäftigt.

Der C#-Treiber enthält einige DLLs, welche man einfach in das Projekt einbinden kann. Zum einen ist da die MongoDB.Bson, welche all die Bson-Funktionalitäten beinhaltet. Und zum anderen ist da die eigentliche MongoDB.Driver, welche all die Datenbank-Funktionalität beinhaltet.

Diese Dlls müssen in das Projekt eingebunden und dann via Using-Statement nutzbar gemacht werden. Danach lässt sich der MongoDB-Datenbank-Server nutzen, sobald er gestartet ist.

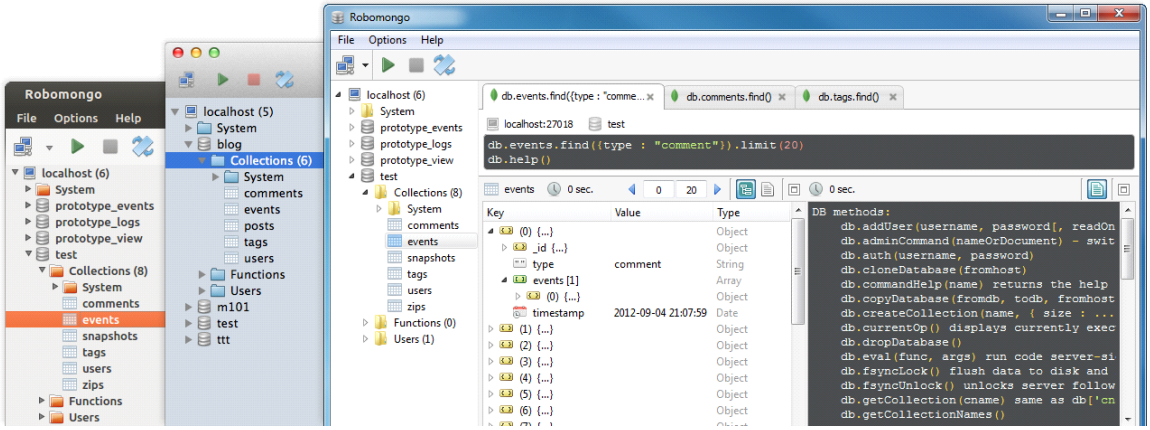
7. Administieren von MongoDB

Eine Administration auf Datenebene von MongoDB ist wegen der dynamischen Datenstruktur nicht unbedingt notwendig, da man keinerlei Struktur zu Beginn des Projektes festlegen muss. Diese Struktur kann durch INSERTS durch die Anwendung vollständig dynamisch festgelegt werden. Die Tabellen, also Collections, und die Datenbank werden ebenfalls durch die INSERTS direkt erstellt. Eine dedizierte Erstellung dieser Strukturen ist weder notwendig, noch möglich.

Das Einzige, das man vielleicht an der Datenstruktur festlegen möchte, sind die Indizes und die Unique-Felder. Außerdem kann es sein, dass man Daten überprüfen oder manuell einfügen möchte.

Vor ein paar Jahren gab es noch keine kostenlose graphische Oberfläche für die Administration von MongoDB unter Windows. Doch nun hat sich ein Projekt namens Robomongo durch Crowdfunding finanziert.

Robomongo läuft unter Windows, Mac und Linux und bietet eine einfache Oberfläche zur Darstellung und Bearbeitung von MongoDB-Datenbanken. Es lassen sich Json-Queries per Console abfeuern, und man kann einige Befehle auch über die graphische Oberfläche verwenden. Man kann das Tool unter https://robomongo.org/ herunterladen.



Die Oberfläche von Robomongo unter den verschiedenen Betriebsystemen (Quelle: https://robomongo.org/)

8. MongoDB C#-Treiber

Um MongoDB von C# aus zu nutzen, ist es notwendig die Treiber-DLLs in das Projekt einzubinden. Dazu gehören die MongoDB.Bson und die MongoDB.Driver.

Allgemein kann man sagen, dass der MongoDB-Treiber sehr ähnlich funktioniert wie die Datenbank-Treiber von SQL-Datenbanken. Es gibt SELECT-, INSERT-, REMOVE- und UPDATE-Queries, und diese sind von der Struktur her sehr ähnlich wie die SQL-Queries.

Server starten

```
Process serverProcess = new Process();
serverProcess.StartInfo.FileName = "...\\mongo\\mongod.exe\";
serverProcess.StartInfo.Arguments = "--dbpath [PFAD]";

serverProcess.Start();
```

In diesem Code wird ein neuer Prozess für den Datenbank-Server erstellt, welcher die mongodb.exe als ausführbare Datei nutzt. Als Startparameter wird der –dbpath gesetzt, um den Pfad zu der Datenbankdatei festzulegen. Am Ende wird der eigentliche Prozess gestartet.

Datenbank erzeugen

```
MongoClient client = new MongoClient();
MongoServer server = client.GetServer();
MongoDatabase database = server.GetDatabase(„testdb");
```

Hier wird erst der Client erstellt und dann darüber der Server abgefragt. Aus diesem Server wird dann die eigentliche Datenbank geholt. Zu beachten ist, dass die Datenbank an diesem Punkt nicht existieren muss. Man kann sie in jedem Fall abfragen und sie wird erstellt, sobald man seinen ersten INSERT-Befehl ausführt. Das MongoDatabase-Objekt kann von nun an für die weiteren Operationen verwendet werden.

Collection verwenden

```
var collection = database.GetCollection("numbers");
if(collection.Exists() == false)
        //Existiert noch nicht, kann aber verwendet werden
```

In diesem Stück Code wird eine Collection, also das Equivalent einer Tabelle, in der MonogDB abgefragt. Genau wie bei der Datenbank wird hier nicht beachtet, ob die Collection bereits besteht. Sie wird erstellt, sobald ein INSERT ausgeführt wird.

Wir können jedoch trotzdem bei Bedarf überprüfen, ob die Collection bereits existiert, um darauf zu reagieren. Erstellen können wir die Collection aber nur über einen INSERT-Befehl.

Insert

```
BsonDocument document = new BsonDocument();
document.Add(new BsonElement("name", new BsonString("Hans")));
document.Add(new BsonElement("nummer", new BsonInt64(30)));
document.Add(new BsonElement("datum", new BsonDateTime(new DateTime(1993, 7, 17))));
```

Dieser Code erstellt ein Bson-Dokument und füllt es mit den Key-Value-Paaren. Die Key-Value-Paare eines Bson-Dokuments sind immer vom Typ BsonElement und beinhalten einen String Key und einen Bson-Datentyp als Value.

```
List<string> stringListe = new List<string>();
stringListe.Add("eins");
stringListe.Add("zwei");
stringListe.Add("drei");

document.Add(new BsonElement("eineListe", new BsonArray(stringListe)));
```

Ein Array einzufügen in MongoDB ist sehr einfach. Man erstellt einfach eine Liste mit einem von MongoDB-unterstützten Datentyp und erstellt daraus ein BsonArray. Für die Liste lassen sich sämtliche Klassen verwenden, welche das Interface IEnumerable implementieren.

```
collection.Insert(document);
```

Der eigentliche Vorgang des Inserts ist extrem einfach: Wir fügen das erstellte Dokumenten-Objekt vom Typ BsonDocument über den Befehl Insert in die Collection ein.

Select

```
var documents = collection.Find(Query.LT("nummer", 40))
        .SetSortOrder(SortBy.Descending("nummer"))
        .SetLimit(10);

List<BsonDocument> list = documents.ToList<BsonDocument>();
```

Ein Select-Befehl funktioniert im C#-Treiber ähnlich wie beim SQL-Treiber. Wir suchen in der Collection mit dem Befehl Find() und erstellen dafür eine Filter-Query. Diese Queries bekommen wir über die statische Query-Klasse. Die Funktion LT bedeutet in diesem Fall „Less then“, der String ist das zu vergleichende Feld, und die Zahl ist der Vergleichswert. Auf diesen Find-Befehl wenden wir dann noch eine Sort-Order an, welche wir über die ebenfalls statische Klasse „SortyBy“ bekommen. In diesem Fall sortieren wir absteigend nach dem Feld „nummer“. Am Ende wird noch ein Limit von zehn Elementen gesetzt.

Nach diesem Befehl hält der Datenbankserver alle gefundenen Dokumente vor, und gibt uns einen MongoDB-Pointer auf das Ergebnis. Wir können dann entweder nach und nach durch das Ergebnis iterieren, um Arbeitsspeicher zu sparen, oder wir laden das gesamte Ergebnis in eine Liste. Ich habe mich bei dem Beispiel für zweiteres entschieden.

Der Datentyp in den spitzen Klammern, ist der Datentyp, in den das Resultat geparst wird. Da alle Daten in einer MongoDB immer im BsonDocument-Format gespeichert werden, wird dieser Cast also immer funktionieren.

Update

```
collection.Update(Query.GT("nummer", 40), Update.Set("nummer", 40));
```

Der Update-Befehl funktioniert sehr ähnlich wie der Select-Befehl, mit dem einzigen Unterschied, dass wir noch ein Update-Query angeben müssen. In diesem Fall suchen wir alle Dokumente in collection, bei denen das Feld nummer einen Wert größer als 40 besitzt und setzen diese Werte auf genau 40.

```
collection.Update(Query.GT("nummer", 40), Update.Inc("nummer", 1));
```

Dieser Update-Befehl ist wie der vorhergehende, außer dass wir einen logischen Operator verwenden. Wir addieren auf den Wert in dem Feld nummer Eins hinzu.

Remove

```
collection.Remove(Query.EQ("nummer", 30));
```

Der Remove-Befehl ist genau so aufgebaut wie der Select-Befehl. Sie unterscheiden sich nur im Ergebnis. In diesem Fall werden alle Dokumente gelöscht, bei denen der Wert in dem Feld nummer genau gleich 30 ist.

Object to Bson

Durch die MongoDB.Bson DLL lassen sich komplexe Objekte, welche durch serialisierbare Klassen definiert werden, direkt in eine MongoDB-Collection speichern.

```
[Serializable]
class TestClass
{
    BsonObjectId _id;
    public int nummer;
    public string text;
    public TestClass subObj = null;

    public TestClass(string text, int nummer)
    {
      this.nummer = nummer;
      this.text = text;
    }
}
```

```
TestClass tc = new TestClass("hallo", 3);
```

Dies ist die Klasse, deren Objekte wir direkt in die Datenbank speichern wollen. Wichtig ist, dass wir sie als Serializable gekennzeichnet haben, und dass sie ein Feld für die ID mit dem Typ BsonObjectId enthält. Außerdem definieren wir ein paar mehr Attribute, und geben uns sogar die Möglichkeit, die Objekte mit dem Feld subObj rekursiv zu speichern. Am Ende erstellen wir ein Objekt von der Klasse namens tc.

Nun gibt es zwei Möglichkeiten dieses Objekt in der Datenbank zu speichern:

```
collection.Insert(tc.ToBsonDocument());
```

Wir konvertieren das Objekt mit einer vorgefertigten Methode namens ToBsonDocument, welche wir für jedes Objekt verwenden können, in ein BsonDocument und fügen es dann in die Datenbank.

```
collection.Insert<TestClass>(tc);
```

Oder wir geben der Insert-Methode die Information über den Typ unseres Objektes mit und speichern unser Objekt direkt ab.

Bson to Object

Um ein Objekt einer bestimmten Klasse direkt aus der Datenbank auslesen zu können, müssen wir genau über die Struktur in der Datenbank Bescheid wissen. Dabei hilft uns MongoDB nicht weiter, dies müssen wir selbst bewerkstelligen.

Sind wir jedoch in der Lage, genau zu wissen, was für eine Struktur wir bei einem Select auslesen werden, so können wir die Daten direkt in ein passendes Objekt speichern.

```
TestClass tc = collection.FindOneAs<TestClass>(Query.EQ("nummer", 3));
```

Dies ist ein einfacher SELECT-Befehl, welcher nur ein Ergebnis erwartet. Wir suchen ein Dokument in der Collection, dessen Wert des Feldes nummer genau drei ist. Dieses Dokument muss genau der Struktur der TestClass entsprechen, ansonsten wird ein Fehler geworfen. Am Ende werden die Daten direkt in ein neues Objekt des Types TestClass gespeichert, welches wir nun verwenden können.

```
List<TestClass> tcList = collection.FindAs<TestClass>(Query.EQ("nummer", 3))
.ToList<TestClass>();
```

Dieses Stück Code macht genau das gleiche wie der letzte Select-Befehl, mit dem Unterschied, dass wir mehrere Ergebnisse entgegen nehmen können und deshalb in einer Liste abspeichern können.

9. Fazit zum MongoDB-Treiber

Der MongoDB-Treiber ist einfach zu verwenden und extrem konfortabel. Die meisten Befehle sind genau so aufgebaut wie SQL-Befehle. Wer SQL versteht, wird sich hier sehr schnell zurecht finden. Die Ähnlichkeit zwischen MongoDB und SQL ist nicht weiter verwunderlich, schließlich steht NoSQL ja für „Not Only SQL“, es gibt also große Gemeinsamkeiten.

Besonders praktisch sind die Features des MongoDB-Treibers, welche ich in den Abschnitten Object to Bson und Bson to Object beschreibe. Auf diese Weise lassen sich Objekte extrem einfach und komfortabel in einer Datenbank persistieren und auch wieder auslesen. Trotz all des Komforts muss man hier auch nicht auf die mächtigen SQL-ähnlichen Befehle für die Datenbank verzichten.

Will man die gesamte Flexibilität des NoSQL-Ansatzes in MongoDB nutzen, so wird man nicht umhinkommen, sich auch mit den BsonDocuments zu beschäftigen. Doch wenn man eine stets konstante Datenstruktur verwenden möchte, so kann man MongoDB durch diese Features mit extrem kleinem Aufwand verwenden, um seine Objekte zu persistieren.

10. Fazit zu MongoDB

Vorteile von MongoDB

MongoDB ist extrem flexibel. Es gibt keine festgelegte Struktur, und man kann gefühlt speichern was man will: Von rekursiven Datenstrukturen zu riesigen Listen und Binär-Files.

Natürlich entsteht trotzdem eine gewisse Struktur, diese wird aber nicht von der Datenbank festgelegt oder durchgesetzt. Stattdessen legt die Anwendung die Struktur mit ihren INSERTs fest. Der Anwendungsentwickler ist also ebenfalls für die Struktur der Datenbank verantwortlich, und kann sich diese so zurechtlegen wie es am besten für die Anwendung ist. Die Anwendung bestimmt also vollständig die Datenbank.

Die Datenbank muss nicht dediziert eingerichtet werden. Es ist im MongoDB-Server gar nicht möglich eine Datenbank einzurichten oder eine Struktur zu definieren. Die Entwickler sparen sich also einen ganzen Arbeitsschritt und fangen direkt mit der Anwendung an. Das Festlegen der Struktur geschieht dann nach und nach bei der Entwicklung der Anwendung. Dies passt besonders gut zur agilen Softwareentwicklung.

Bei einer Datenstruktur, welche bei SQL-Datenbanken eine Vielzahl an Joins benötigen würde, bietet MongoDB die Möglichkeit diese kompakt zu speichern. Anstatt von einem Objekt auf das andere über eine ID zu verweisen, wie in einer herkömlichen SQL-Datenbank üblich, lassen sich in MongoDB einfach untergeordnete Objekte in ein anderes Objekt einbetten. Liest man das Oberobjekt aus, so hat man ohne Mehraufwand Zugriff auf die Unterobjekte. Bei SQL-Datenbanken wären dafür Joins, und damit ein wesentlich größerer Rechenaufwand nötig. Bei der richtigen Datenstruktur kann MongoDB bei den Abfragen also um ein vielfaches schneller sein als eine SQL-Datenbank.

MongoDB ist dank seiner simplen Datenstruktur sehr gut skalierbar. Da keine Transaktionssicherheit besteht, kann eine MongoDB-Instanz einfach auf mehreren Servern

gleichzeitig laufen, um damit die Verarbeitungsgeschwindigkeit zu steigern. Dies ist einer der wichtigsten Vorteile gegenüber SQL-Datenbanken.

Die Datenstrukur einer MongoDB kann während des Betriebs einer Anwendung ganz einfach von der Anwendung selbst geändert werden. Beispielsweise fügt die Anwendung einfach Datensätze mit der neuen Struktur in die Datenbank ein. Die Collection enthält dann also verschiedene Datensätze mit verschiedenen Versionen und Datenstrukturen. Dies ist bei herkömlichen SQL-Datenbanken nicht so einfach möglich. Eine explizite Migration ist bei MongoDB ebenfalls denk- und umsetzbar, aber eben nicht unbedingt nötig.

Ein Vorteil von MongoDB gegenüber den herkömmlichen SQL-Datenbanken ist die Fähigkeit, ganz einfach Arrays zu speichern. Dies ist ein eher komplexer und rechenaufwendiger Prozess in SQL-Datenbanken. Dadurch lassen sich in MongoDB ganz einfach 1:n- Beziehungen von Objekt zu simplen Datentypen oder von Objekt zu anderen Objekten realisieren. Dieses Feature ist einfacher und performanter in MongoDB als in SQL-Datenbanken. Dies ist eines meiner Lieblingsfeatures von MongoDB.

Nachteile von MongoDB

Eine MongoDB hat keine garantierte Struktur, das bedeutet, Fehler seitens der Anwendung werden einfach so übernommen. MongoDB bietet keinerlei Sicherheit, die Anwendung ist für alle Daten verantwortlich, MongoDB übernimmt da keine Verantwortung. Dadurch kann man sagen, dass der Anwendungsentwickler zwei Aufgaben übernimmt. Er ist sowohl für Logik, als auch für die Daten verantwortlich. Es bestehen keinerlei Kontrollmechanismen für ihn.

Sollte die Anwendung fehlerhaft sein, so wird die Datenbank das dank der Strukturlosigkeit einfach akzeptieren. Fehler sind schwer zu finden und werden in die Daten übernommen. Mit MongoDB kann es schnell passieren, dass die Daten über einen längeren Zeitraum falsch eingespeichert werden und es niemand merkt.

In MongoDB müssen die Relationen von Hand programmiert werden. MongoDB bietet keinen Schutz vor verwaisten Relationen. Beim Löschen eines Objektes müssen möglicherweise sämtliche Objekte bearbeitet werden, die irgendwie in einer Beziehung zu diesem Objekt stehen. MongoDB hilft einem bei dieser Aufgabe nicht. All diese Operationen müssen selbst implementiert werden und sind dadurch auch fehleranfälliger.

MongoDB sorgt nicht für die Integrität der Daten. Für die Integrität der Daten übernimmt die Anwendung die vollständige Verantwortung.

In MongoDB gibt es keine Transaktionssicherheit. Ein Dokument, das bearbeitet wird, ist in diesem Moment gesperrt. Eine Aktion, die über mehrere Dokument hinweg geschieht, ist jedoch nicht atomar. So kann es zu den verschiedensten Fehlern durch Reentrants kommen.

Allgemein kann man sagen, dass MongoDB weniger sicher und weniger zuverlässig ist, im Vergleich zu den herkömlichen SQL-Datenbanken.

Fazit

MongoDB ist eine gute Alternative zu den klassischen SQL-Datenbanken. Es kommt jedoch immer auf den jeweiligen Fall an.

MongoDB ist sehr geeignet, wenn SQL-Datenbanken viele Joins brauchen würden. Hier kann man einen großen Performance-Vorteil erwarten. Ausserdem lässt sich die Performance durch die gute Skalierbarkeit noch weiter verbessern. Allgemein kann man sagen, das MongoDB immer dann sinnvoll ist, wenn Geschwindigkeit und Performance wichtiger sind als Sicherheit und Datenintegrität.

SQL-Datenbanken sind immer dann zu bevorzugen, wenn die Sicherheit und Datenintegrität wichtiger ist als die Geschwindigkeit und Performance. SQL-Datenbanken übernehmen durch ihre feste Datenstruktur und ihre internen Relationen zwischen den Objekten eine große Verantwortung für die Integrität der Daten. Wenn die Anwendung Fehler macht, dann können diese oft von der Datenbank erkannt werden. Dazu kommt, dass die Integrität der Daten durch Modifikationen nicht verletzt wird. Dies wird garantiert durch die Transaktionsicherheit, die bei den SQL-Datenbanken gewährleistet wird.

Die meisten größeren Anwendungen brauchen oftmals beides: Performance und Sicherheit. Das ist durchaus möglich. Zum Beispel verwendet man für die Account-Daten eine SQL-Datenbank, um maximale Sicherheit zu haben, und für den Content verwendet man MongoDB, um bessere Performance zu erzielen. Dann hat man das Beste aus beiden Welten.

11. Beispielanwendung

Problemstellung

Meine Anwendung sollte folgende Aufgabe erfüllen: „Wie erfahre ich welche Themen in der deutschen Presselandschaft in diesem Moment aktuell sind?“. Außerdem wollte ich eine Möglichkeit bieten, zu erfahren, wie sich die Popularität eines Themas über die Zeit in den Medien entwickelt. Man sollte einfach mehr über ein bestimmtes Thema erfahren können.

Die Anwendung sollte für jeden erreichbar sein, und die Features sollten möglichst leicht, intuitiv und schnell verwendbar sein. Kurz: Mir war auch die Usability und Performance wichtig.

Lösung

Um diese Aufgabe lösen zu können, muss die Anwendung Daten über die Presselandschaft jetzt und in der Vergangenheit haben. Um dies zu bewerkstelligen, habe ich mich entschieden, automatisch jede Stunde die RSS-Feeds aller großen deutschen Zeitungen abzufragen, und alle Artikel zu speichern. Dies sind die Daten über die Gegenwart. Um Daten über die Vergangenheit zu bekommen, speichere ich die Daten einfach und baue mir so eine immer größer werdende Datenbank auf.

Diese Daten über die deutsche Presselandschaft wertet die Anwendung dann regelmäßig aus, um zu erfahren welche Themen gerade aktuell sind. Dies geschieht über eine Auswertung der verwendeten Schlagworte in den Überschriften.

Die Anwendung zerlegt also alle aktuellen Überschriften in Schlagworte und überprüft, wie oft welches Schlagwort vorkommt. Je häufiger ein Schlagwort vorkommt, desto wichtiger ist das Thema im Moment.

Über die Zeit läßt sich so ein Graph aufbauen, wie oft welches Wort an welchem Tag verwendet wurde. Man sieht, ob das Thema immer wichtiger wird, langsam an Popularität verliert, oder sich in Wellen bewegt.

Außerdem lassen sich die fünfzig wichtigsten Worte in den letzten 24 Stunden anzeigen, man weiß also, was gerade aktuell ist. Interessiert sich jemand für ein bestimmtes Thema, so kann er es auswählen, und bekommt die neuesten Artikel der verschiedenen Zeitungen dazu zu sehen.

Warum dieses Thema?

Ich wählte dieses Thema, da ich eine solche Anwendung nutzen will. Ich wusste aus Erfahrung, dass dies für mich umsetzbar sein wird.

Außerdem passt diese Anwendung gut zu dem Thema NoSQL und Datenbanken, da extrem viele Daten anfallen und diese aufwändig ausgewertet werden müssen.

Ausgangslage

Ich hatte eine ähnliche Anwendung mit dem gleichen Ziel bereits schon einmal umgesetzt. Damals war es aber eher ein schnell und unsauber entwickelter Prototyp. Ich hatte damals große Probleme mit der Performance, die Anwendung war einfach zu langsam. Außerdem war ich mit der Wartbarkeit und der Oberfläche der Anwendung nicht zufrieden.

Ich habe also bereits Erfahrung mit der Problemstellung, und habe mir schon einmal ein grobes Konzept für die Lösung überlegt. Ich wusste recht genau was zu tun war. Obwohl ich damals einen Prototyp entwickelt hatte, konnte ich fast keinen Code wiederverwenden. Dies lag daran, dass ich eine andere Architektur verwende, und andere Technologien zum Einsatz kommen.

Durch diese grundlegenden Änderungen, konnte ich keinen Code wiederverwenden, versprach mir aber ein wesentlich besseres Ergebnis. Meine Ziele waren eine gute Wartbarkeit, starke Performance, also kurze Ladezeiten und eine saubere Architektur.

12. Systemarchitektur

Um ein möglichst wartbares und effizientes System zu erstellen, entwickelte ich eine Systemarchitektur aus mehreren sehr spezialisierten Komponenten. Jede Komponente sollte möglichst ein gut abgrenzbares Aufgabengebiet haben und auf der dafür am besten geeigneten Technologie aufgebaut sein. Die Systemarchitektur besteht aus vier großen Komponenten:

- Aggregator
- Datenbank
- Rest-API
- Frontend

Aggregator:

Das Aggregations-Modul hat die Aufgabe des Downloads der Daten von den großen deutschen Zeitungen und die Verarbeitung dieser Daten. Ganz konkret werden die Artikel über die RSS-Feeds heruntergeladen und dann in ein einheitliches Format gebracht. Danach werden die Überschriften in Schlagworte aufgespalten.

Die Artikel und die Schlagworte werden nun nach dem Download und dem Processing in die Datenbank geschrieben.

Der Aggregator wird jede Stunde aktiv und ist vollständig unabhänig vom Frontend. Seine Aufgabe ist es nur, die Daten immer aktuell zu halten. Hier wird durch das Processing der Daten und den Download die meiste Rechenzeit des Servers verbraucht.

Datenbank:

Die Datenbank ist eine MongoDB. Hier werden alle Daten aus der Anwendung gespeichert. Dazu gehören unter anderem alle Artikel, die heruntergeladen werden. Außerdem werden die Schlagworte mit Datum und Anzahl gespeichert. Ebenfalls nötig ist eine Collection mit allen Datenquellen mit den jeweiligen URLs der RSS-Feeds. Eine weitere Tabelle beinhaltet die Schlagworte, die keinen Nutzen für den User haben. Diese werden beim Processing im Aggregator herausgefiltert. So zum Beispiel „Der“, „Die“ oder „Das“.

Rest-API:

Die Rest-API ist das Modul, welches die Daten aus der Datenbank bei Bedarf wieder herausliest. Wenn der Aggregator also sozusagen die schreibende Komponente ist, dann ist die Rest-API die lesende.

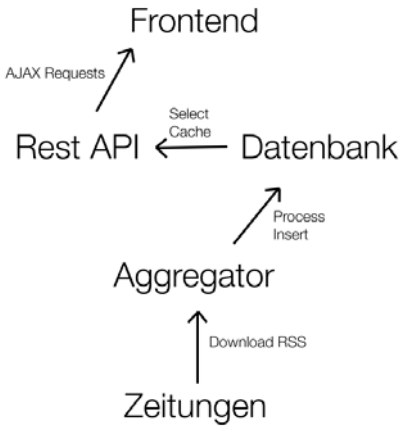
Dieses Modul ist eine Schnittstelle nach dem Rest-Standard zwischen den Daten und dem Frontend. Die Daten werden online abrufbar im Json-Format bereitgestellt.

Da die Rest-API direkt mit dem Frontend interagiert, ist es entscheidend, dass sie schnell ist. Die Daten müssen möglichst schnell zur Verfügung stehen, und die nötigen Rechenoperationen sollten möglichst schon vorher durchgeführt worden sein. Deshalb greift auch hier ein starkes Caching der Daten. Die Rest-API sollte die meisten Daten möglichst schon bereit halten, bevor sie überhaupt abgerufen werden, um dann schnell reagieren zu können.

Frontend:

Das Frontend ist die Stelle, an der der User mit dem System interagiert. Hier kommt es vor allem auf das Design und auf Geschwindigkeit an. Am Frontend sind Ladezeiten tabu.

Um eine gute Usability zu gewährleisten kann das Frontend in Echtzeit auf den Benutzer reagieren und besorgt sich die benötigten Daten direkt von der Rest-API. Diese Abfragen laufen alle im Hintergrund, damit die Userinteraktion niemals ins Stocken gerät.



Plan der Systemarchitektur mit all den verwendeten abkapselbaren Modulen und deren groben Aufgaben.

13. Technologien

Ich wählte die verschiedenen Technologien für die verschiedenen Module mit der Absicht, für jedes Modul die Technologie zu nutzen, die am effizientesten für die jeweilige Aufgabe funktionieren würde. Dies ist mir auch geglückt, mit einer Ausnahme.

Für die Wahl der Technologie teile ich mein System in drei verschiedene Ebenen auf:

1. Backend:

Das Backend ist alles, was ich programmiere und nicht direkt mit dem Nutzer in Kontakt kommt. Dazu zählt das Aggregationsmodul, welches den Data-Download und das Data-Processing durchführt. Außerdem zählt dazu die Rest-API, welche nie vom Nutzer direkt abgerufen wird, sondern nur vom Frontend. Hier werden die Daten von der Datenbank ausgelesen und verfügbar gemacht.

Ich habe für das Backend ASP.net als Technologie gewählt. Der Grund dafür ist, dass das Backend die ganze Arbeit mit der Datenbank übernimmt. Hier liegt also die Schnittstelle zwischen MongoDB und meinem System.

Um diese Schnittstelle nur einmal programmieren zu müssen, habe ich das gesamte Backend, also Aggregator und Rest-API, in einer Programmiersprache geschrieben und daraus eine Software gemacht. Ich programmierte das Backend mit C#.

Außerdem ist ASP.net spezialisiert auf Websites und Webservices. Dadurch war es sehr einfach über ASP.net eine Rest-API aufzubauen. Das Konvertieren von Objekten in C# zu Json-Strings ist hier extrem einfach. Auch für das Routing der API gibt es in ASP.net schon fertige Lösungen.

2. Datenbank:

Die Datenbank ist natürlich MongoDB. Dies habe ich gewählt, weil ich mir durch MongoDB einen Performance-Vorteil gegenüber SQL-Datenbanken erhoffte, und weil ich so ganz einfach komplexe Datenstrukturen speichern kann. Der Zugriff auf die Datenbank läuft vollständig über C# und eine Klasse, welche ich für meine Anwendung in C# implementierte. Diese Klasse wird nur von ASP.net aus verwendet.

3. Frontend:

Das Frontend sollte interaktiv und benutzerfreundlich sein, außerdem wollte ich, dass es sehr gut zugänglich ist. Dadurch war klar, dass es eine Website sein würde. Wegen der erforderlichen Interaktivität entschied ich mich für eine Logik vollständig in Javascript, mit Jquery als Erweiterung. Das Design definierte ich mit HTML und CSS. Um die Website responsive zu machen, verwendete ich das Grid-System von Twitters Bootstrap. Die Daten für das Frontend werden vollständig asynchron über Ajax im Hintergrund von der Rest-API heruntergeladen.

14. Datenstruktur

Der Kern und der eigentliche Nutzen der Anwendung sind ihre Daten. Diese Daten werden vom Aggregations-Modul heruntergeladen und verarbeitet. Nach der Verarbeitung werden

sie in die Datenbank gespeichert. Spätestens hier werden die Daten in das Bson-Format konvertiert. In diesem Kapitel gehe ich darauf ein, wie ich die Daten organisiere, und wie sie in der Datenbank abgespeichert werden:

Artikel:

Die wichtigste Date der Anwendung ist die Liste der Artikel. Diese wird in einer Collection auf der MongoDB gehalten, und es werden jede Stunde die neuen Artikel hinzugefügt. Um die Struktur dieser Artikel-Daten zu erläutern, hier ein Beispiel:


```
{
   "_id" : ObjectId("573c83ae78514dcd10a92b1a"),
   "headline" : "Netzneutralität: Niederländisches Parlament verbietet Zero Rating",
   "sourceid" : "HeiseDE",
   "summery" : "Die &amp;quot;Tweede [...] preislich bevorzugen.",
   "published" : 1463582820,
   "downloaded" : 1463583661.43959,
   "url" : "http://www.heise.de/da[...]beitrag.atom",
   "id" : "Netzneutralität: Niede[...]etet Zero Rating_1582761805_HeiseDE"
}
```


Ganz typisch für MongoDB fängt das Dokument mit der ID an. Standardgemäß heißt dabei das Feld „_id“, und der Typ ist eine ObjectId. Weiter geht es mit der Headline, also der Überschrift des Artikels. Die SourceID ist ein Verweis auf eine andere Collection mit den Quellen für Artikel. In diesem Fall ist es die Website mit dem Namen HeiseDE. Zu den sogenannten Sources siehe den Abschnitt „Quellen“. Summery ist eine optionale, von der Website bereit gestellte Zusammenfassung des Artikels, in Realität ist es aber meist eher ein Teaser-Text. Published ist ein UnixDateTime in Form eines Integers, welches die Sekunden seit 1970 zählt. Ich habe dieses Format für die Zeit gewählt, um besonders einfach Daten vergleichen zu können. Dabei ist zu beachten, dass diese Zeit von der Website bereit gestellt wird und möglicherweise nicht korrekt ist. Besser ist der Downloaded-Wert. Dieser ist ein von meiner Anwendung hinzugefügter Wert, welcher angibt, wann das Aggregations-Modul den entsprechenden Artikel entdeckt und heruntergeladen hat. Die Anwendung nutzt stets diesen Wert, da dieser weit zuverlässiger ist als der Published-Wert. Die URL ist ein Link zu dem eigentlichen Artikel im Internet, und die ID ist eine von mir generierte Unique-ID des Artikels, eine Art Hash.

Common Words:

Die Collection mit dem Namen Common-Words wurde von mir manuell befüllt und hält alle Worte, welche ich für den Nutzer als unwichtig einstufte. Dies sind Worte, die kein Thema sind und deshalb von der Statistik nicht beachtet werden sollen.


```
{
  "_id" : ObjectId("570a73a689e1fe3290a7eff0"),
  "word" : "bewirken"
}
```


Diese Dokumente enthalten nur eine ID und ein Feld namens Word, welches einen String hält. Befüllt habe ich diese Tabelle mit den meist verwendeten Worten in der deutschen Sprache, da diese am allgemeinsten sind und deshalb keinen Mehrwert bieten. Eine Liste solcher meist verwendeten Worte findet man im Internet.

Quellen:

Die Collection mit den sogenannten Sources ist eine manuell eingepflegte Liste von Zeitungen und Websites mit RSS-Feed. Vor allem sind die Websites der großen deutschen Zeitungen und Zeitschriften vertreten. Dazu kommen ein paar Websites wie der Webauftritt der Tagesschau und ein paar Fernsehsender wie NTV. Diese Liste wird immer durchgegangen, wenn wieder Daten heruntergeladen werden sollen. Die URLs sind die Ziele für den Downloader des Aggregations-Moduls.


```
{
  "_id" : ObjectId("570a73a689e1fe3290a7efac"),
  "name" : "Hamburger Abendblatt",
  "typ" : "Zeitung",
  "country" : "DE",
  "id" : "HamburgerAbendblattDE",
  "urls" : [
    "http://www.abendblatt.de/?service=Rss",
    "http://www.abendblatt.de/hamburg/?service=Rss",
    "http://www.abendblatt.de/sport/?service=Rss",
    "http://www.abendblatt.de/region/norddeutschland/?service=Rss",
    "http://www.abendblatt.de/vermischtes/?service=Rss",
    "http://www.abendblatt.de/kultur-live/kino/?service=Rss",
    "http://www.abendblatt.de/kultur-live/tv-und-medien/?service=Rss",
    http://www.abendblatt.de/wirtschaft/karriere/?service=Rss
  ],
  "error" : 9877
}
```


Ein Quellen-Dokument besteht aus der üblichen ID, dem Namen, einem Typ, dem Land, einer Unique-ID und einer Liste von URLs für die RSS-Feeds der Seite. Außerdem wird gespeichert, wie oft es einen Fehler beim Download gab, also wie oft die Seite nicht reagiert hat.

Worte:

Die Collection mit den sogenannten Words speichert das Vorkommen der einzelnen Schlagworte pro 24 Stunden. Diese Daten sind theoretisch redundant, da man diese Information aus der Artikel-Collection extrahieren könnte. In Realität braucht das aber extrem viel Rechenaufwand, weshalb ich diese Daten immer beim Anfallen neuer Daten direkt mitführe und dann redundant speichere. Dadurch kann ich die Ergebnisse wesentlich schneller ausliefern.


```
{
  "_id" : ObjectId("573f81d878514d8b301457d4"),
  "date" : 1463779800.50259,
  "word" : "schneckentempo",
  "count" : 1
}
```


Ein Dokument für ein Wort beinhaltet die obligatorische ID und ein Datum, für wann dieser Wert gilt. Dies gilt immer für einen Zeitraum von 24 Stunden, eine höhere Auflösung speichere ich nicht. Natürlich braucht es zu dem Datum noch das eigentliche Wort, welches unter dem Feld word gespeichert wird, und die Anzahl, wie oft dieses Wort in diesen 24 Stunden in den Artikeln vorkam.

In diesem Fall wurde das Wort „Schneckentempo“ am Fri, 20 May 2016 innerhalb 24 Stunden genau ein Mal in einer der Überschriften verwendet. Es war also kein großes Thema an diesem Datum. Ein Wort wie Fußball-WM oder Brexit kommt teilweise auf Werte von über 500 Mal in 24 Stunden.

15. Performance und Cache

Das Aggregations-Modul kommt niemals in Konakt mit dem Nutzer, und es ist deshalb nicht relevant, wie schnell es seine Aufgaben erledigt. Das Frontend hat keine aufwendigen Rechenoperationen zu erledigen und bedarf deshalb ebenfalls keiner Optimierung bezüglich der Performance. Die beiden Performance-kritischen Module sind die Datenbank, welche die Rest-API mit den Daten versorgt und die Rest-API selbst. Um diese beiden Module zu entlasten, habe ich zwei Strategien verfolgt:

- Caching
- Redundante Daten

Caching

Die meisten Daten welche die Rest-API ausliefern muss, stehen schon im Voraus fest. So zum Beispiel die Top 50-Themen für die letzten 24 Stunden. Auch die Artikel für jeweils diese Themen stehen vorher schon fest. Da ich die Daten alle 60 Minuten aktualisiere, ist auch das Cache-Invalidating trivial. Die Software baut den Cache einfach jede Stunde nach der Aktualisierung der Daten neu auf.

Konkret werden die Daten jede Stunde neu aus der Datenbank ausgelesen und in eine schnell abrufbare Form gebracht. Die Daten werden dann in den Arbeitsspeicher gelegt und verwahrt. Sobald dann eine Anfrage von der Rest-API eingeht, wird nachgesehen, ob die Daten schon im Cache vorhanden sind, und dann direkt aus dem Arbeitsspeicher zurück gegeben. Dies geht wesentlich scheller als die Abfrage aus der Datenbank.

Redundanzen

Normalerweise versucht man so wenige Redundanzen in den Daten, also in der Datenbank, zu haben, um möglichst wenig Speicherplatz zu verbrauchen. In diesem Fall habe ich mich aber dafür entschieden absichtlich Daten redundant zu speichern, um das Abfragen der Daten zu beschleunigen.

Ansich lassen sich alle Daten aus der Artikel-Collection generieren. Doch da das Aufspalten und Zählen der Schlagworte und das Zuordnen zu dem jeweiligen Datum eine sehr rechenaufwendige Prozedur ist, habe ich diese Daten ebenfalls abgespeichert. Die Collection „Words“ ist also redundant und lässt sich vollständig aus der Collection „Articles“ generieren. Dadurch habe ich das „Processing“ der Daten zu dem Zeitpunkt der Abfrage schon erledigt und greife nur noch auf das Ergebnis zu. Dies macht den Abfrage schneller, benötigt jedoch mehr Speicherplatz.

Ergebnis

Durch diese beiden Strategien erreiche ich eine Beschleunigung des Abfrage-Prozesses für die API an zwei Punkten. Einerseits ist es für das API-Modul sehr viel schneller, die Daten aus der Datenbank abzufragen, da die Ergebnise des „Processings“ schon vorliegen. Andererseits werden diese Ergebnise dann meistens nicht einmal von der Datenbank geholt, sondern befinden sich schon direkt im Arbeitsspeichers des Servers.

16. Cron, Hangfire und Quartz.net

Da ich das Aggregations-Modul zusammen mit der Rest-API in C# auf ASP.net umsetze, hatte ich Schwierigkeiten bei der Umsetzung der Funktionalität des Aggregations-Moduls. Genauer gesagt ist es schwierig in ASP.net sich regelmäßig wiederholende Hintergrundprozesse zu erstellen.

ASP.net ist für Webanwendungen ausgelegt, die auf Anfragen reagieren. Ein autonomer, im Hintergrund laufender Prozess scheint nicht vorgesehen zu sein. Doch ich brauche genau so einen Prozess, um jede Stunde neue Daten herunterzuladen, zu verarbeiten und in die Datenbank zu schreiben.

Diese Anforderung ist sehr vergleichbar mit einem einfachen Cron-Job. Ich wollte jedoch keine Cron-Jobs verwenden, da ich das Aggregations-Modul und die Rest-API in einer einheitlichen Technologie umsetzen wollte, damit sich beide Module die Datenbank-Anbindung teilen können und ich diese nur einmal umsetzen muss.

Also recherchierte ich nach C#-Bibliotheken um diese Cron-Funktionalität in ASP.net zu bekommen. Ich fand zwei beliebte Bibliotheken: Hangfire und Quartz

Hangfire

Hangfire ist eine kostenlose Bibliothek für ASP.net. Man kann es herunterladen unter http://hangfire.io/. Hangfire bietet genau die Funktionalität, welche ich mir erwartet habe. Man kann sich wiederholende Aufgaben erstellen, die dann im Hintergrund abgehandelt werden. Praktischerweise hat Hangfire auch noch ein Webinterface, wo man nachsehen kann, welche Hintergrundprozesse laufen und wie lange diese dauern.

Ein Problem an Hangfire und damit der Grund, warum ich es letztendlich nicht verwendet habe, ist, dass Hangfire eine MySQL-Datenbank benötigt, um die Hintergrundprozesse zu verwalten. Ich habe mich dagegen entschieden, da ich auf meinem Server neben meinem MongoDB-Server keine MySQL-Datenbank laufen lassen wollte.

Quartz.net

Quartz.net ist eine Bibliothek, die eigentlich für Java entwickelt wurde, und nun auf C# portiert wurde. Man kann die Bibliothek kostenlos von http://www.quartz-scheduler.net/ herunterladen.

Quartz.net bietet all die Funktionalität, die ich brauche: Man kann Hintergrundprozesse regelmäßig ausführen lassen. Im Vergleich zu Hangfire gibt es kein Webinterface, und es wird auch keine MySQL-Datenbank benötigt.

Leider ist Quartz.net sehr unzuverlässig auf ASP.net, da hier wohl teilweise Threads gestoppt werden, und sogar die ganze Anwendung bei zu wenigen Anfragen pausiert wird. Als Lösung musste ich einige Anpassungen an den Konfigurationen des IIS-Webservers vornehmen. Doch selbst wenn die Anwendung eigentlich nicht pausiert werden sollte und immer aktiv bleibt, kann es zu Aussetzern von Quartz.net unter ASP.net kommen. Ich entschied mich dann irgendwann für eine schnelle und unsaubere Lösung: Ich Pinge meine Website regelmäßig an, um die Anwendung aktiv zu halten. Dann funktioniert es.

17. Klassenarchitektur des Backends

Für das Backend habe ich insgesamt zwanzig Klassen programmiert. Dies umfasst die Module für die Aggregation und für die Rest-API.

Aggregator

Die NewsAggregatorScheduler-Klasse ist für das wiederholte Aufrufen des Aggregations-Hintergrund-Prozesses zuständig. Hier werden eigentlich alle Aggregations-Methoden aufgerufen. Dazu verwendet es die Klassen ArticleProcessor und DataDownloader. Wie der Name schon sagt, sind diese beiden Klassen für den Download und die anschließenden Verarbeitungsroutinen zuständig.

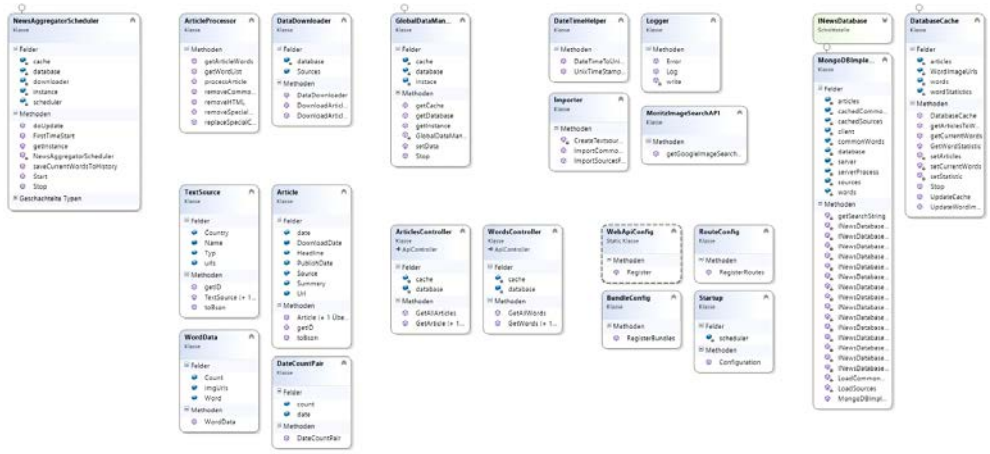
Diese Klassen verwenden wiederum das sogenannte Model. Das bedeutet alle Klassen, die für die Modelierung der Daten zuständig sind. Dazu gehören die Klassen Article, TextSource, WordData und DateCountPair. Diese Klassen definieren auch das Format, wie die Daten in der Datenbank liegen.

Wenn der Download und das Processing beendet sind, werden die Daten über die MongoDBImplementation-Klasse in die MongoDB geschrieben. Diese implementiert die Schnittstelle InewsDatabase. Durch dieses Vorgehen kann man theoretisch die Datenbank austauschen ohne den Rest des Codes anrühren zu müssen.

Rest-API

Auf der anderen Seite der Anwendung steht die Rest-API. Diese wird bedient durch die Klassen ArticklesController und WordsController. Über diese Klassen wird die eigentliche Web-API definiert. Diese Klassen verwenden ebenfalls die Klasse MongoDBImplementation, um auf die Datenbank zuzugreifen. Dies war mir sehr wichtig, damit ich diese Klasse für die beiden Module verwenden kann, und nicht zwei Mal implementieren muss.

Außer der Datenbank verwendet die API natürlich auch die Klasse DatabaseCache für die Cachingfunktionalität. Dazu kommen noch ein paar Hilfs- und Konfigurations-Klassen.



(Sämtliche Klassen des Backends)

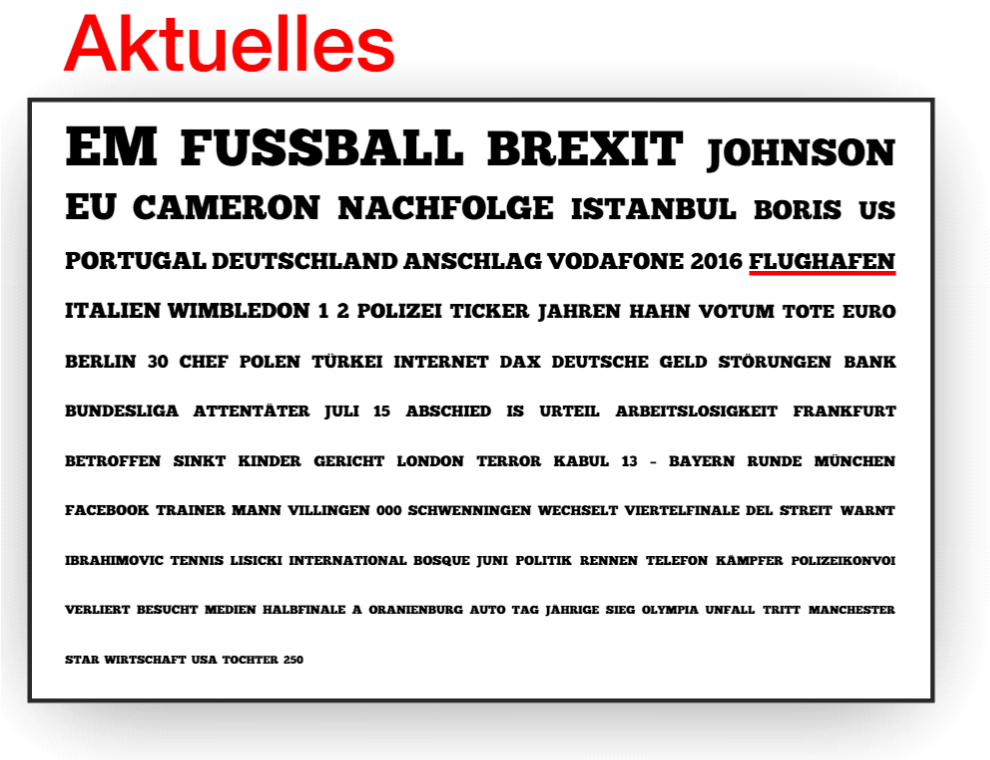
18. Frontend

Meine Anforderungen an das Frontend waren eine gute Usability, ein schickes Design und schnelle Ladezeiten. Um die Anwendung verfügbar machen zu können, war es klar, dass es eine Website werden sollte.

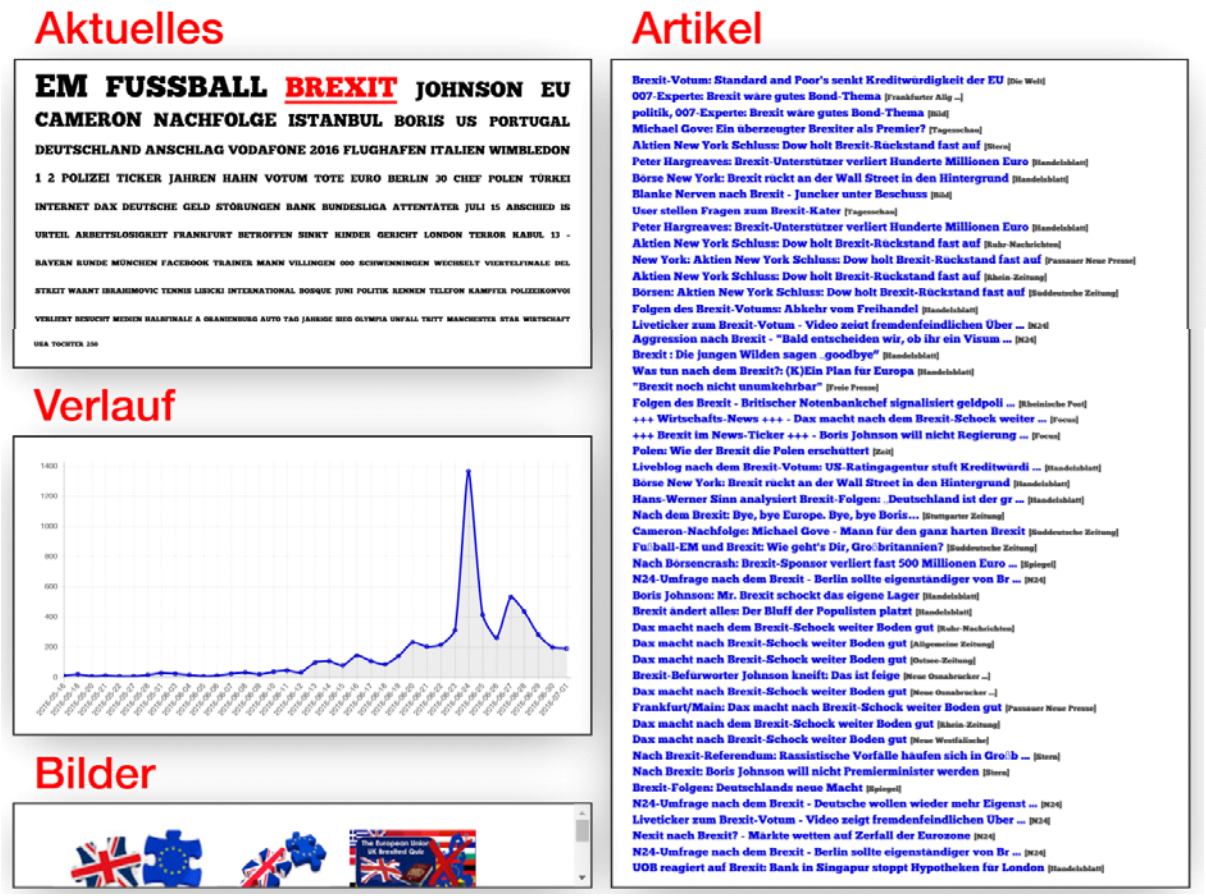
Die Gestaltung der Website setzte ich mit HTML und CSS um. Damit das Layout responsive wird, und es auf so vielen Geräten und Bildschirmgrößen wie möglich gut aussieht, habe ich Twitters Bootstrap und dessen Grid-System verwendet.

Die Interaktivität der Website habe ich vollständig mit Javascript umgesetzt, damit alle Funktionen in Echtzeit und gegebenenfalls im Hintergrund ablaufen können. So werden zu Beginn keinerlei Daten mit der Website ausgeliefert. Beim Abruf der Seite erhält man tatsächlich nur die HTML-, CSS- und Javascript-Dateien.

Die Daten werden dann im Nachhinein mit Javascript über die Rest-API abgerufen und dargestellt. Dies geschieht vollständig asynchron. Für die Logik in Javascript habe ich JQuery verwendet. Außerdem muss ein Chart dargestellt werden, dafür habe ich die Javascript-Bibliothek Chart.Js benutzt.



Wie Oben zu sehen wird dem Benutzer erst einmal eine Visualisierung der in den letzten 24 Stunden aktuellen Themen gezeigt. Je größer das Wort, desto populärer ist es gerade in den großen Zeitungen.



Klickt man auf eines der Worte, so werden die entsprechenden Daten dynamisch nachgeladen und man bekommt drei zusätzliche Fenster zu sehen: Die neuesten Artikel zu dem Thema, der Verlauf der Polularität des Wortes über die Zeit und ein paar Bilder zu dem Thema.

Die Bilder sind die ersten Bilder, die bei Google zu diesem Begriff auftauchen. Dies ist eine Zusatzfunktion welche erst nicht geplant war, die ich dann aber doch noch umgesetzt habe.

Vom Design her entschied ich mich für ein möglichst klares und sauberes Aussehen. Ich verwende die Farben Rot, Schwarz und Blau, und ich verwende leichte Schatten um den Hintergrund vom Vordergrund abzugrenzen.

19. Fazit Anwendung

Ich bin mit der Anwendung zufrieden, sie ist Modular aufgebaut und deshalb gut wartbar, sie ist zuverlässig und skalierbar.

Ich habe eine Anwendung aus verschiedenen Modulen aufgebaut, dabei jeweils eine passende Technologie gewählt und diese miteinander verzahnt. Dieser Prozess des Konzipieren einer Architektur und der Wahl von Technologie, also die Technische „Gestaltung“ der Anwendung, war für mich sehr lehrreich.

Der Aspekt der Performance war ein sehr ergiebiges Übungsgebiet, da dieser Bereich bei einer guten Anwendung für den Benutzer immer unsichtbar ist und gleichzeitig eine große technische Herausforderung darstellt. Es hat sich gelohnt mich damit zu beschäftigen. Siehe dazu „Kapitel 15. Performance und Cache“.

Es gibt einige Features, welche ich noch gerne umsetzen würde, welche durch die gute Architektur auch technisch relativ leicht realisierbar sind. So liese sich die Anwendung recht einfach internationalisieren. Alles was zu tun wäre ist weitere Quellen aus dem Ausland hinzuszufügen und die API und das Frontend anzupassen.

Außerdem würde es sich anbieten eine Suchfunktion sowohl für Themen, als auch für Artikel hinzuzufügen. Die Umsetzung dieser Features ist jedoch etwas komplexer, wenn man sie performant halten möchte. Auch andere Statistische Auswertungen der Daten wären denkbar.

Eine weitere wichtige Funktion der Anwendung, neben der Kernfunktionalität, war ja die Evaluation der Technologie MonogDB. Was das angeht bin ich ebenfalls zufrieden, da sich die Anwendung auch im Nachhinein für eine Evaluation von MongoDB als geeignet herausstellte. Ich habe MongoDB durch diese Anwendung sehr umfangreich und vor allem praxisnah ausprobiert. Ich kenne nun die Vor- und Nachteile von MongoDB und weiß wie und wann ich die Datenbank am besten verwende. Siehe dazu „Kapitel 10. Fazit MongoDB“.