

Fakultät für  
Informatik  
und Mathematik



ARCHITEKTUR  
FÜR SKALIERBARE WEBANWENDUNGEN

**Bachelor-Thesis**

zur Erlangung des akademischen Grades

***Bachelor of Science (B.Sc.)***

im Studiengang Wirtschaftsinformatik

an der

Hochschule für angewandte Wissenschaften München  
Fakultät für Informatik und Mathematik

BETREUER: Prof. Dr. Oliver Braun  
VORGELEGT VON: Vladislav Faerman  
MATRIKELNUMMER: 02929612  
BEARBEITUNGSZEIT: 3 Monate  
EINGEREICHT AM: 24. April 2017

# **Eidesstattliche Erklärung**

Hiermit versichere ich, dass ich die vorliegende Arbeit 'Architektur für skalierbare Webanwendungen' selbstständig und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt worden und auch nicht veröffentlicht worden.

München, 24. April 2017

---

Vladislav Faerman

# Inhalt

<b>Eidesstattliche Erklärung</b>	<b>I</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation und Ziel der Arbeit . . . . .	1
<b>2 Skalierbarkeit und Wartbarkeit</b>	<b>3</b>
2.0.1 Skalierbarkeit . . . . .	3
2.0.2 Vertikale Skalierbarkeit . . . . .	3
2.0.3 Horizontale Skalierbarkeit . . . . .	4
2.0.4 ACID-Prinzip . . . . .	4
2.0.5 Das CAP-Theorem . . . . .	5
2.0.6 BASE . . . . .	8
2.1 Wartbarkeit . . . . .	8
2.1.1 Dependency Injection (DI) . . . . .	10
2.1.1.1 Inversion of Control (IoC) . . . . .	11
2.1.1.2 Dependency Injection und Mock-Objekte . . . . .	11
2.1.2 MVC-Pattern . . . . .	12
2.1.2.1 Workflow . . . . .	12
2.1.2.2 Beobachter Muster . . . . .	13
2.1.2.2.1 Idee . . . . .	14
<b>3 Architektur</b>	<b>15</b>
3.1 Datenhaltungsschicht (Databases) . . . . .	17
3.1.1 NoSQL-Datenbanken . . . . .	17
3.1.1.1 Kategorien von NoSQL-Systemen . . . . .	18
3.2 MongoDB . . . . .	19
3.2.1 Datensätze in Form von Dokumenten . . . . .	20

3.2.2	Die Architektur	21
3.2.3	Server/Client starten	21
3.2.4	CRUD = IFUR-Operationen	22
3.2.4.1	Create/Insert	23
3.2.4.2	Read/Find	23
3.2.4.3	Update/Update	23
3.2.4.4	Delete/Remove	24
3.2.5	Indizes	24
3.2.6	Aggregation	25
3.2.6.1	Aggregation Framework	25
3.2.7	Horizontale Skalierung (Sharding)	26
3.2.7.1	Fragmentierung des Datenbestands nach <i>Shard-Keys</i>	28
3.2.8	Replikation (Replication)	29
3.2.8.1	Eine Replikationsgruppe erzeugen	31
3.2.9	MongoDB mit Java	31
3.3	Apache Cassandra	31
3.3.1	Allgemein	32
3.3.2	Architektur	32
3.3.3	Datenmodell	33
3.4	Logikschicht (Backend)	34
3.4.1	Rest Server	34
3.4.2	Spring MVC	34
3.5	Präsentationsschicht (Frontend)	34
3.6	Frameworks	34
<b>4</b>	<b>Implementierung</b>	<b>35</b>
4.1	Fachliche Spezifikation	35
4.2	3-Schichten-Architektur	35
4.2.1	Präsentationsschicht (Frontend)	35
4.2.2	Logikschicht (Backend)	35
4.2.3	Datenhaltungsschicht (Datenbank)	35
<b>5</b>	<b>Prototyp: Testen auf Cluster (optional)</b>	<b>36</b>
<b>Zusammenfassung</b>		<b>37</b>

**Literaturverzeichnis**

**A**

**Abbildungen**

**B**

**Tabellen**

**C**

# 1 Einleitung

Internetnutzer erwarten heutzutage von den Webanwendungen, dass diese kurze Ladezeiten, flüssige selbsterklärende Bedienung und ständige Verfügbarkeit aufweisen. Viele Webanwendungen sind nicht in der Lage, mit rasant steigender Anzahl von Anfragen und großen Datenmengen effizient umzugehen.

Dies ist für erfolgreiche Projekte, die rapide populär werden und ein exponentielles Anwenderwachstum erleben, ein ernsthaftes Problem. Um von dem Projekterfolg profitieren zu können und diesen auszubauen, ist es überlebenswichtig, den Wachstumsanforderungen gerecht zu werden. Die Hardware stellt gegenwärtig kein großes Problem mehr dar: Die Cloud-Services wie z. B. *Amazon* oder *Microsoft Azure* ermöglichen den Zugang zu den fast unbegrenzten Hardwareressourcen. Die Herausforderung für Entwickler besteht darin, die Webanwendung so zu bauen, dass diese von dem Hardwareangebot Gebrauch machen kann. Die Wartungs- und Erweiterungsfähigkeit sind zwei weiteren wichtigen Punkte, die für den Erfolg unabdingbar sind. Um die Internetnutzer beizubehalten, sollen die neuen Features schnell implementiert und ausgerollt werden können, auch wenn das Projekt größer wird.

## 1.1 Motivation und Ziel der Arbeit

Das Ziel dieser Arbeit ist, eine solche Architektur für Webanwendungen vorzustellen, die die Entwicklung von skalierbaren, wartungs- und erweiterungsfähigen Webanwendungen ermöglicht. Es wird des Weiteren ein Software- und Frameworkstack vorgeschlagen, der diese Architektur abdeckt. Die vorgeschlagenen Software/Frameworks sind unter freien Lizzenzen verfügbar.

Um die Realisierbarkeit und das Zusammenspiel aller Komponenten zu untersuchen, wird ein Prototyp implementiert und eigene Erfahrungen aus dem Entwicklungsprozess berichtet. Für den Prototyp wird der webbasierte Foto-Verwaltungs-Service gewählt. Diese Anwendung zeichnet sich dadurch aus, dass jeder Internetnutzer eigene Daten unabhängig von den anderen Nutzern verwalten kann, was bei vielen Webanwendungen der Fall ist. Andererseits (besse

## 2 Skalierbarkeit und Wartbarkeit

In diesem Kapitel wird der Begriff *Skalierbarkeit* erklärt sowie die notwendigen Kompromisse erläutert, die bei der Entwicklung einer verteilten skalierbaren Anwendung eingegangen werden müssen. Das **CAP**-Theorem beschreibt die Grenzen eines verteilten Systems und **BASE** fasst größtmögliche Anforderungen an ein verteiltes System zusammen. Danach werden die *Best Practices* und *Patterns* beschrieben, deren Einhaltung die Entwicklung einer modulären Webanwendung mit austauschbaren Komponenten ermöglicht.

### 2.0.1 Skalierbarkeit

Der Begriff *Skalierbarkeit* beschreibt die Fähigkeit eines Systems, das bei wachsenden Anforderungen, entweder die Leistung der vorhandenen Ressourcen verbessert oder zusätzlich die neuen Ressourcen hinzufügt.

Der Begriff *Skalierbarkeit* beschreibt die Fähigkeit eines Systems, aufgrund der wachsenden Anforde

Das System, bei dem die neuen Ressourcen hinzugefügt werden, nennt man *verteilte Systeme*.

Bei der Skalierung sind zwei Arten zu unterscheiden, eine *vertikale* und eine *horizontale Skalierung*, die demnächst näher erläutert wird.

### 2.0.2 Vertikale Skalierbarkeit

Die *vertikale Skalierbarkeit (scale-up)* strebt eine qualitative Steigerung der Leistungsfähigkeit an, bei der die bereits eingesetzten Ressourcen, beispielsweise durch die Speichererweiterung oder CPU-Steigerung, verbessert werden.

Die vertikale Skalierbarkeit hat den Vorteil, dass die Daten nicht verteilt werden müssen. Die Nebenläufigkeit kann mit *Threads* realisiert werden. Jedoch hat die vertikale Skalierbarkeit ihre Grenzen - ein Rechner kann nicht endlos vergrößert werden.

### 2.0.3 Horizontale Skalierbarkeit

Im Gegensatz zur vertikalen Skalierung verteilt die *horizontale Skalierbarkeit (scale-out)* die Daten auf verschiedenen Knoten im großen Cluster, wobei die quantitative Steigerung der Leistungsfähigkeit angestrebt wird. Somit können mehrere weniger leistungsfähige, nicht so teure Rechner eingesetzt werden. Ein verteiltes System kann viel mehr als ein vertikales Skalieren - Erweiterung eines Clusters um weitere Rechner ist sehr einfach und Clusters können auch sehr groß werden. Die horizontale Skalierbarkeit ist günstiger - je leistungsfähiger ist der Rechner, desto teurer ist seine Erweiterung. Allerdings unterscheidet sich die Entwicklung eines verteilten Systems von den klassischen Anwendungen, die auf einer Maschine laufen, da die Daten in dem Cluster verteilt sind. Die *Trade-offs* einer verteilten Anwendung wurden bereits in der CAP-Theorem (**Kap. 2.0.5**) formalisiert.

### 2.0.4 ACID-Prinzip

Des Weiteren sind sinnvolle Regeln zum effektiven und effizienten Umgang mit Transaktionen unvermeidbar. Solche Regeln sind in einem **ACID-Prinzip** definiert.

**ACID** steht für **A**tomicity (Atomarität), **C**onsistency (Konsistenz), **I**solation (Isolation) und **D**urability (Dauerhaftigkeit) und beschreibt somit die Eigenschaften eines Datenbankmanagementsystems zur Sicherung der Datenkonsistenz bei Transaktionen.

- **A**tomicity (Atomarität): Die *Atomarität* einer Transaktion bedeutet, dass sie entweder ganz oder gar nicht ausgeführt wird. Falls eine Transaktion abgebrochen wird, werden alle im Laufe der Transaktion schon durchgeführte Änderungen rückgängig gemacht, um Konflikte mit der Ausführung neuer Transaktionen zu vermeiden.

- **Consistency** (Konsistenz): Die *Konsistenz* besagt, dass vor und auch nach dem Ablauf einer Transaktion die Integrität und Plausibilität der Datenbestände gewährleistet werden. Die Integrität der Datenbank ist es möglich, beispielsweise mit Integritätsbedingungen<sup>1</sup> zu gewährleisten.
- **Isolation** (Isolation): Die *Isolation* dient zu Kapselung von Transaktionen, um unerwünschte Nebenwirkungen vermeiden zu können. Die Transaktionen müssen unabhängig voneinander ablaufen.
- **Durability** (Dauerhaftigkeit): Die *Dauerhaftigkeit* gewährleistet nach einer erfolgreichen Transaktion die Persistenz aller Datenänderungen. Im Falle eines Systemfehlers oder Neustarts müssen die Daten nichtsdestotrotz zur Verfügung stehen, dass sie in einer Datenbank dauerhaft gesichert sein müssen.

### 2.0.5 Das CAP-Theorem

Im Jahr 2000 präsentierte Eric A. Brewer das **CAP**-Theorem - ein Ergebnis seiner Forschungen zu verteilten Systemen. Das Ergebnis zeigte, dass bei den verteilten Systemen alle drei folgenden Anforderungen wie **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz) gleichzeitig nicht zu erfüllen sind.

---

<sup>1</sup>Unter Integritätsbedingungen (Zusicherungen, Assertions) sind Bedingungen zu verstehen, die die Korrektheit der gespeicherten Daten sichern. Diese werden in SQL zum Beispiel mithilfe von CONSTRAINTS formuliert. Folgende CONSTRAINTS sind möglich: NULL, NOT NULL, PRIMARY KEY, FOREIGN KEY etc.

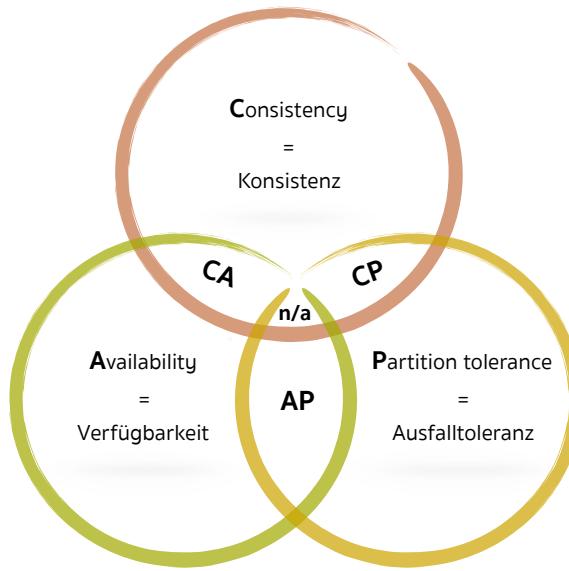


Abb. 2.1: Anforderungen an verteilte Systeme gemäß dem **CAP**-Theorem

Das Akronym **CAP** steht für die englischsprachigen Begriffe **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz). Diese sind mögliche Anforderungen an eine verteilte Anwendung.

- **Consistency** (Konsistenz): Diese Anforderung ist erfüllt, wenn nach Abschluss einer atomaren<sup>2</sup> Transaktion (oder Interaktion mit dem System) nicht nur die manipulierenden Datensätze, sondern auch alle replizierenden Knoten in einem großen Cluster über die gleichen Daten verfügen. Wenn ein Wert auf einem Knoten geändert wird und die Interaktion mit dem System abgeschlossen wird, muss der aktualisierte Wert von anderen Knoten zurückgeliefert werden können. Dies hat zur Folge, dass ein System erst dann die Interaktion abschließen darf, wenn sichergestellt ist, dass die Änderungen auf alle Datenkopien angewendet wurden. Für die verteilten Systeme, die Daten replizieren, resultiert es in langen Antwortzeiten für die Schreiboperationen.
- **Availability** (Hochverfügbarkeit): Die *Hochverfügbarkeit* ist eine weitere Anforderung, die besagt, dass immer alle gesendeten Anfragen durch User an das System mit einer

<sup>2</sup>Eine atomare Transaktion bedeutet, dass sie entweder ganz oder gar nicht ausgeführt wird. Falls eine atomare Transaktion abgebrochen wird, werden alle im Laufe der Transaktion bereits durchgeführte Änderungen rückgängig gemacht.

akzeptablen Reaktionszeit beantwortet werden müssen.

- Partition Tolerance (Partitionstoleranz): Die *Partitions- oder Ausfalltoleranz* bedeutet, dass der Ausfall eines Knoten bzw. eines Servers aus einem Cluster das verteilte System nicht beeinträchtigt und es weiterhin fehlerfrei funktioniert. Falls einzelne Knoten in so einem System ausfallen, wird deren Ausfall mit den verbleibenden Knoten aus dem Cluster kompensiert, um die Funktionsfähigkeit des Gesamtsystems aufrecht zu halten.

Die graphische Darstellung für das Brewer's **CAP**-Theorem ist aus der Abbildung 2.1 zu entnehmen. Wie die Abbildung 2.1 erkennen lässt, können in einem verteilten System gleichzeitig und vollständig nur zwei von drei Anforderungen **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit), **Partition Tolerance** (Partitionstoleranz) erfüllt werden. Konkret aus der Praxis bedeutet das, dass es für eine hohe Verfügbarkeit und Partitions- oder Ausfalltoleranz notwendig ist, die Anforderungen an die Konsistenz zu lockern [1, S. 31].

Die Anforderungen in Paaren klassifizieren gemäß dem **CAP**-Theorem bestimmte Datenbanktechnologien. Für jede Webanwendung muss daher individuell entschieden werden, ob sie als ein **CA-**, **CP-** oder **AP**-System zu realisieren ist.

- **CA** (Consistency und Availability): Die klassischen relationalen Datenbankmanagementsysteme (RDBMS) wie Oracle, DB2 etc. fallen in **CA**-Kategorie, die vor allem auf **Consistency** (Konsistenz) und **Availability** (Hochverfügbarkeit) aller Knoten in einem Cluster hinzielen. Hierbei werden die Daten nach dem **ACID**-Prinzip verwaltet. Die relationalen Datenbanken sind für Ein-Server-Hardware konzipiert und vertikal skalierbar. Das bedeutet, dass solche Systeme mit hochverfügbaren Servern betrieben werden und **Partition Tolerance** (Partitionstoleranz) nicht unbedingt in Frage kommt.
- **CP** (Consistency und Partition tolerance): Ein gutes Beispiel für die Webanwendungen, die zu der **CP**-Kategorie zuzuordnen sind, sind Banking-Anwendungen. Für solche Anwendungen ist es wichtig, dass die Transaktionen zuverlässig durchgeführt werden und der mögliche Ausfall eines Knotens verschmerzt werden kann.
- **AP** (Availability und Partition tolerance): Für die Anwendungen, die in die **AP**-Kategorie fallen, rückt die Anforderung **Consistency** (Konsistenz) in den Hintergrund.

Beispiele für solche Anwendungen sind die Social-Media-Sites wie Twitter oder Facebook, da die Hauptidee der Anwendung nicht verfällt, wenn zum gleichen Zeitpunkt die replizierten Knoten nicht die gleiche Datenstruktur aufweisen.

## 2.0.6 **BASE**

**BASE** steht für **B**asically **A**vailable, **S**oft **S**tate, **E**ventually **C**onsistent und beschreibt den Gegenteil zu den strengen **ACID**-Kriterien (**Kap. 2.0.4**). **BASE** ist wie **CAP**-Theorem (**Kap. 2.0.5**) auch für verteilte Datenbanksysteme formuliert, für die die *Konsistenz* nicht mehr im Vordergrund steht, sondern die *Verfügbarkeit* eines Systems. Bei solchen Systemen, die nach dem **BASE**-Prinzip gestaltet sind, ist eher wichtig, dass für alle Clients das System ständig verfügbar ist. Die Clients müssen nicht unbedingt zu dem gleichen Zeitpunkt die gleichen Daten sehen.

## 2.1 **Wartbarkeit**

Um die Wartungs- und Erweiterungsfähigkeiten von der Software zu gewährleisten, sollen die grundlegenden Designprinzipien eingehalten werden. **SOLID**<sup>3</sup>, beschrieben von Martin Fowler, steht für fünf Prinzipien des objektorientierten Designs. Die richtige Anwendung dieser Prinzipien bringt die Struktur in das Softwareprodukt. Die Software wird mit der Einhaltung von **SOLID** Regeln entwickelt, bestehend aus vielen kleinen Modulen. Jedes Modul hat eine klare Funktion und die Interaktion zwischen den Modulen erfolgt über explizit definierte Schnittstellen. Im Einzelnen beschreibt **SOLID** folgende Regeln:

- *Single responsibility principle* - Eine Klasse (oder Modul) soll nur eine bestimmte Funktion abdecken und eine Funktion soll von einer Klasse implementiert werden. Martin Fowler betont, dass für die Änderung einer Klasse nur ein Grund geben kann. Im Kontext der Prototypanwendung könnte das Backend Teil beispielsweise zwei Funktionen haben, die Bearbeitung von Frontend-Anfragen und Verwaltung der Daten in der Datenbank. Es wäre schlechtes Design, wenn eine Klasse beide

---

<sup>3</sup>SOLID, [https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf), zugegriffen am 03.02.2017

diese Funktionalitäten implementieren würde. Es gäbe dann mehrere Gründe für die Änderung dieser Klasse - z. B. Änderung in Kommunikation mit Frontend oder in der Datenhaltung.

- *Open/closed principle* - Die Klassen/Module sollen für die Erweiterung offen sein, die bestehenden Klassen sollen aber nicht geändert werden. Die Idee dahinter ist, dass wenn die neuen Funktionalitäten eingeführt werden, darf der bestehende Code nur minimal geändert werden. Zum Beispiel die Daten der Foto-Verwaltungs-Anwendung wurden bisher in der relationalen Datenbank gehalten und zusätzlich soll die Datenhaltung in der NoSQL Datenbank implementiert werden. Die NoSQL Datenhaltung soll in eigenem Modul implementiert werden und darf die Codebasis für die Interaktion mit relationaler Datenbank nicht ändern.
- *Liskov substitution principle* - Die Subklassen dürfen das Verhalten der Elternklassen nicht ändern. Der Code, der auf bestehenden Funktionen der Elternklassen aufgebaut ist, muss auch mit Subklassen fehlerfrei funktionieren.

Wenn z. B. die Foto-Verwaltungs-Anwendung erweitert wird und z. B. die Videos verwaltet werden müssen, wäre es falsch die Klasse ‘*Video*‘ aus der Klasse ‘*Foto*‘ abzuleiten, weil Videos andere Eigenschaften als Bilder haben.

- *Interface-segregation principle* - Die Interfaces sollen so klein wie möglich sein und nur einzelne Funktionen abdecken.
- *Dependency inversion principle* - Die Abhängigkeiten zwischen Modulen sollen über Abstraktionen (*Interfaces*) gekoppelt werden. Ein Modul soll keine direkte Abhängigkeit zu den anderen Modulen haben, die Abhängigkeiten werden zu den Interfaces definiert. Demzufolge können die Module die Interfaces implementieren und ohne großen Aufwand ausgetauscht werden. Die implementierende Module sehen nur die Interfaces, die zu implementieren sind und wissen nicht bei welchen Modulen diese eingesetzt werden. Wir nehmen Backend der Foto-Verwaltungs-Anwendung nochmal als Beispiel und betrachten die Interaktion zwischen dem *Service* Modul, der mit Frontend interagiert und dem Modul der Daten in der relationalen Datenbank verwaltet. Anstatt eine Referenz zu diesem spezifischen Datenbankmodul zu deklarieren, wird eine Referenz zu einem Interface deklariert, der die Datenhaltungskomponente

beschreibt. Aus der Sicht von dem *Service* Modul gibt es eine Menge von notwendigen Operationen, um die Daten zu speichern oder abzurufen. Diese Operationen sind ähnlich für verschiedene Arten von Datenhaltung und deswegen werden die in einem *Interface* zusammengefasst. Der *Service* Modul deklariert eine Abhängigkeit zu diesem Interface. Dieser Interface kann von verschiedenen Dateihaltungskomponenten implementiert werden.

### 2.1.1 Dependency Injection (DI)

*Dependency Injection* ist der nächste Schritt nach *Dependency Inversion*. Der *Dependency Injection Pattern* basiert auf dem *Inversion of Control Konzept*. Das bedeutet, dass die verwendeten Klassen sich nicht mehr selbst um Ablauf und Abhängigkeiten kümmern, sondern diese werden an eine externe Komponente ausgelagert. Die Klasse gibt sozusagen die Kontrolle ab und lässt sich von außen steuern. Konkret bedeutet es, dass die Klassen nur die Abhängigkeiten zu den Interfaces deklarieren müssen und die konkrete Implementierung wird zur Laufzeit eingefügt (*injected*). Die genauere Zusammensetzung einer Applikation wird deklarativ definiert, es können auch verschiedene Konfigurationen existieren.

\_\_\_\_\_ ist der untere Teil unbrauchbar????? \_\_\_\_\_

Ein weiteres Ziel, dass in dieser Abschlussarbeit verfolgt wird, ist es unter anderem, eine Architektur nicht nur für eine skalierbare, sondern auch für eine wartbare Webanwendung aufzustellen. Für die ordentliche Wartbarkeit der Anwendung sind die Grundlagen für das *Dependency Injection (DI)* Pattern<sup>4</sup> unvermeidbar. Das Prinzip *Dependency Injection (DI)* wird bei vielen Frameworks wie zum Beispiel Google Guice<sup>5</sup>, Dagger<sup>6</sup> etc. umgesetzt. Im Kapitel ‘Implementierung‘ (**Kap. 4**) wird das Prinzip *Dependency Injection (DI)* sowohl im Präsentationsschicht durch JavaScript-Framework **Angular 2** als auch im Logiksschicht durch **Spring Framework** veranschaulicht.

Der Einsatz des *Dependency Injection (DI)* Pattern ermöglicht den Entwicklern, der Arbeitsaufwand für die Entwicklung großer Anwendungen stark zu reduzieren. Bei sei-

<sup>4</sup>Inversion of Control Containers and the Dependency Injection pattern, <https://martinfowler.com/articles/injection.html>, zugegriffen am 03.02.2017

<sup>5</sup>Google Guice, <https://github.com/google/guice>, zugegriffen am 03.02.2017

<sup>6</sup>Dagger, <http://square.github.io/dagger/>, zugegriffen am 03.02.2017

nem Einsatz wird eine **lose Kopplung** der Anwendungskomponenten erreicht, die dem Entwickler die Konzentration auf die Entwicklung einzelner Komponenten unabhängig voneinander ermöglicht. Die Unabhängigkeit der Programmteile erleichtert dem Entwickler nicht nur die Anwendungskomponenten unabhängig voneinander zu entwickeln, sondern auch diese leichter zu testen.

### 2.1.1.1 Inversion of Control (IoC)

Eine wichtige Idee hinter dem *Dependency Injection Pattern* entspringt dem *Inversion of Control* Konzept. Das bedeutet, dass die verwendeten Klassen sich nicht mehr selbst um Ablauf und Abhängigkeiten kümmern, sondern dass dies eines der Frameworks, wie z. B. **Spring (Kap. ??)**, übernimmt. Die Klasse gibt sozusagen die Kontrolle ab und lässt sich von außen steuern. Dadurch wird eine **lose Kopplung** von Abhängigkeiten erreicht. Das ist der zentrale Vorteil der *DI*.

### 2.1.1.2 Dependency Injection und Mock-Objekte

Enge Kopplung von Abhängigkeiten erschwert nicht nur die Erweiterung oder Änderung von Codebasis, sondern auch die Testbarkeit. Zum Beispiel mit UnitTests sollen die Komponenten unabhängig von den anderen Komponenten getestet werden. Dafür werden die Abhängigkeiten zu anderen Komponenten simuliert. Falls die zu testende Methode die Daten aus der Datenbank für Ihre Berechnung benötigt, kann die Datenbankverbindung simuliert und die Testdaten zurückgegeben werden. Solche Klassen, die die Abhängigkeiten simulieren nennt man **Mock-Klassen**.

Die lose Kopplung des *DI* Patterns ermöglicht die Verwendung von Mock-Objekten. Die entsprechenden Abhängigkeiten werden zur Laufzeit von Tests von dem entsprechenden Framework injiziert, somit sind seitens des Entwicklers keine weiteren Schritte notwendig.

---

---

## 2.1.2 MVC-Pattern

**MVC**<sup>7</sup> ist ein Prinzip der modernen Programmierung und ist nach wie vor das wichtigste und verbreitetste Muster für die Architektur von objektorientierten Anwendungen. Heutzutage ist Model-View-Controller ein bekanntes Modell, das in vielen Programmiersprachen für die Architektur von Frameworks und Anwendungen verwendet wird.

Das Ziel des **MVC**-Musters ist Geschäftslogik einer Anwendung von der Benutzerschnittstelle abzutrennen, so dass Entwickler einen Bereich bequem verändern kann und der Rest der Anwendung wird dadurch nicht beeinflusst. Es soll ein flexibler Programmierentwurf geben, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit und Austauschbarkeit einzelner Komponenten ermöglicht.

### 2.1.2.1 Workflow

Der Workflow-Prozess (**Abb. 2.2**) stellt eine vollständige Beschreibung aller Aktivitäten, der **MVC**-Pattern voraussetzt. Die Abbildung 2.2 ist grober Workflow des **MVC**-Pattern anhand einer Beispielinteraktion und ihrer Ergebnisse präsentiert.

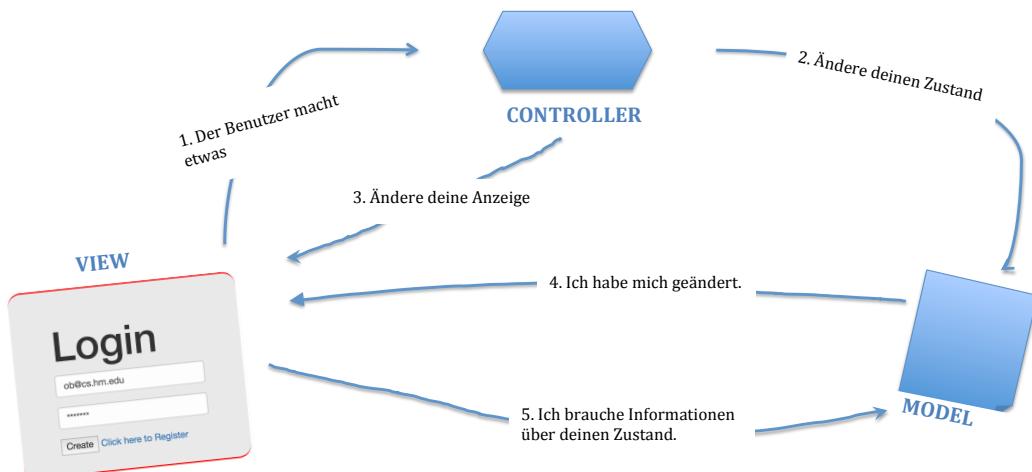


Abb. 2.2: Workflow zum MVC-Konzept

<sup>7</sup>Objektorientierte Programmierung, [http://openbook.rheinwerk-verlag.de/oop/oop\\_kapitel\\_08\\_002.htm](http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_08_002.htm), zugegriffen am 20.01.2017

**Beschreibung des Workflow-Prozesses:****1. Der Benutzer interagiert mit dem View**

Der Benutzer führt irgendeine Aktion an dem View aus. Dadurch teilt der View dem Controller mit, was zu tun ist. Erst dann ist die Aufgabe des Controllers, entsprechende Steuerungsmaßnahmen zu ergreifen.

**2. Der Controller fordert das Model auf, seinen Zustand zu ändern**

Nach der Ausführung irgendeiner Aktion an dem View durch den Benutzer, nimmt der Controller die Aktion an und interpretiert sie. Bei der Interpretation stellt der Controller heraus, was gemacht werden muss und wie das Model aufgrund dieser Aktion beeinflusst werden kann.

**3. Der Controller kann auch den View auffordern, seinen Zustand zu ändern**

Der Controller kann bei der Ausführung einer Aktion auch den View auffordern, sich zu ändern. Zum Beispiel, beim Klick auf einen Button durch den Benutzer kann der gerade eingeblendete View ausgeblendet und ein anderer View eingeblendet werden.

**4. Das Model informiert den View über seine Zustandsänderung**

Dem Model selbst sind Views und Controller nicht bekannt bzw. diese sind an dem Model nicht festprogrammiert. Aber das Model kann diejenige, die sich beim Model registriert haben, über seine Zustandsänderungen informieren, (**Kap. 2.1.2.2**).

**5. Der View erfragt den Zustand des Models**

Das Model stellt weitere Methoden zur Verfügung, über die der aktuelle Zustand des Models erfragt werden kann. Jeder View kann sich somit durch den Aufruf dieser Methoden über den Zustand des Models informieren.

Um die Benachrichtigung über Modelsänderungen an Views oder auch an Controller zu ermöglichen, nutzt MVC das sogenannte Beobachter Muster (**Kap. 2.1.2.2**).

**2.1.2.2 Beobachter Muster**

Beobachter Muster (engl. Observer-Pattern) ist eines der am meisten genutzten und bekanntesten Pattern. In diesem Muster teilt die Komponente Model allen Interessenten proaktiv mit, dass ihr Zustand geändert wurde.

Würde man ohne das **Observer-Pattern** eine solche Beobachtung implementieren, so müssten die Interessenten die Komponente Model regelmäßig abfragen, ob ihr Zustand geändert wurde.

**2.1.2.2.1 Idee** Beim **Observer-Pattern** gibt es eine Komponente (Observable), deren Zustand sich ändern kann und andere Komponenten (Observers), die über Zustandsänderung informiert werden sollten. Das **Observer-Pattern** sieht vor, dass die Observers sich beim Observable registrieren und bei einer Zustandsänderung informiert Observable alle registrierte Objekte.

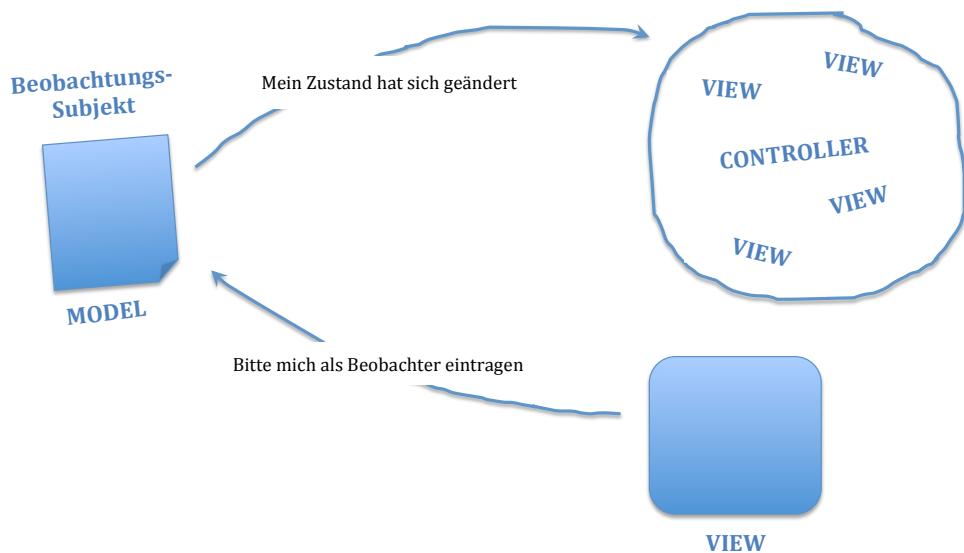


Abb. 2.3: Observer Pattern

### Beschreibung des Observer-Pattern Prinzips:

Die Abbildung 2.3 zeigt, wie **Observer-Pattern** im **MVC** verwendet wird. Wenn ein View bei einer Zustandsänderung des Models informiert werden möchte, registriert er sich beim Model. Der View wird somit in die Liste hinzugefügt, in der sich schon andere Observers befinden können. Im Fall einer Zustandsänderung läuft dann das Model die Liste durch und informiert somit alle, die sich als Beobachter eingetragen haben.

## 3 Architektur

Der webbasierte Foto-Verwaltungs-Service wird auf Basis des **MVC**-Pattern entwickelt, welcher aus drei Komponenten (Model, View und Controller) besteht. Der Backend-Teil enthält das Model. Zwei weitere Komponente, View und Controller befinden sich im Frontend-Teil der Webanwendung. View übernimmt die Verantwortung für die Gestaltung der Webseite und Controller wird in JavaScript implementiert, der die Client-Anfragen entsprechend bearbeitet (**Abb. 3.1**).

Der Zugriff auf Funktionen auf der Serverseite erfolgt über *REpresentational State Transfer* oder kurz **REST**. Durch die Verwendung von REST erfolgen die Web-Client-Anfragen auf Basis des **HTTP**<sup>1</sup>-Protokolls. Der Web-Client sendet **HTTP**-Request und erwartet **HTTP**-Response, in dem Fall des implementierten Prototyps im **JSON**<sup>2</sup>-Format.

Die Skalierung der Daten findet auf der Datenbankebene statt. Im Fall des implementierten Prototyps übernimmt die Verantwortung für die Skalierbarkeit **MongoDB**, eine dokumentenorientierte *NoSQL*-Datenbank.

Die rechtzeitige Synchronisation von Daten wird nicht in Kauf genommen.

Der Zustand (= *Session*) der Web-Client-Anfragen wird im Frontend-Teil implementiert. Der Web-Server enthält keine Information über Session, da dieser *stateless* arbeitet. *Stateless* bedeutet, dass die Web-Clients mit jeder Anfrage einen neuen Verbindungsauflaufbau und erneute Datenbeschaffung erfordern und diese voneinander unabhängig sind. Gleich nach einer Web-Server Response wird die Verbindung geschlossen und der Zustand bleibt nicht gespeichert. Somit ist bei einer *stateless*-Webanwendung Sessions nicht möglich.

---

<sup>1</sup>**HTTP** ist ein zustandloses Protokoll zur Übertragung der Daten zwischen Web-Client (= Browser Anwendung) und Web-Server. Jede neue Anfrage, die vom Web-Client erfolgt, erfordert einen neuen Verbindungsauflaufbau und erneute Datenbeschaffung.

<sup>2</sup>JSON (JavaScript Object Notation), <http://www.json.org>

Falls nach dem Anwendungskontext eine Zustandsspeicherung gefordert wird, wäre die Umsetzung in einer *stateful*-Webanwendung möglich. Bei einer *stateful*-Webanwendung werden Web-Client Anfragen zu einer gemeinsamen logischen Sitzung zusammengefasst, um alle Nutzerinteraktionen hinweg in einem einzigen Kontext auszuführen. Dies wird mithilfe von *Session-Cookies*<sup>3</sup> realisiert.

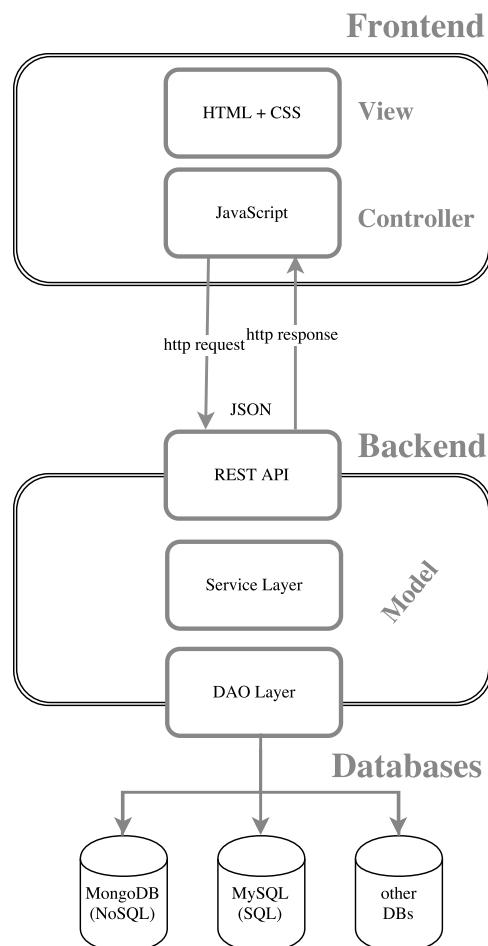


Abb. 3.1: Architektur-Prototyp

<sup>3</sup>Session-Cookie,  
[https://help.sap.com/erp2005\\_ehp\\_04/helpdata/de/cc/d6eef928f711d5991f00508b6b8b11/content.htm](https://help.sap.com/erp2005_ehp_04/helpdata/de/cc/d6eef928f711d5991f00508b6b8b11/content.htm), zugegriffen am 20.01.2017

## 3.1 Datenhaltungsschicht (Databases)

Im Vergleich zu den relationalen Datenbanken, die sich als eine strukturierte Sammlung von Tabellen (den Relationen) vorstellen, in welchen Datensätze abgespeichert sind, eignen sich *NoSQL*-Datenbanken zur unstrukturierter Daten, die einen nicht-relationalen Ansatz verfolgen.

### 3.1.1 NoSQL-Datenbanken

Der Begriff NoSQL steht nicht für 'kein SQL', sondern für 'nicht nur SQL' (Not only SQL). Das Ziel von NoSQL ist, relationale Datenbanken sinnvoll zu ergänzen, wo sie Defizite aufzeigen. Entstanden ist dieses Konzept in erster Linie als Antwort zur Unflexibilität, sowie zur relativ schwierigen Skalierbarkeit von klassischen Datenbanksystemen, bei denen die Daten nach einem stark strukturierten Modell gespeichert werden müssen.<sup>4</sup> Dokument-datenbanken gruppieren die Daten in einem strukturierten Dokument, typischerweise in einer *JSON*-Datenstruktur. **MongoDB** ist eine von vielen NoSQL-Datenbanken, die auch diesen Ansatz verfolgt und bietet darauf aufbauend eine reichhaltige Abfragesprache und *Indexe* auf einzelne Datenfelder. Die Möglichkeiten der *Replikation* und des *Shardings* zur stufenlosen und unkomplizierten Skalierung der Daten und Zugriffe macht **MongoDB** auch für stark frequentierte Websites äußerst interessant. ([2], Kapitel 14, Seite 435)

Beispiele für NoSQL-Datenbanken:

- CouchDB
- MongoDB
- Redis
- Google BigTable
- Amazon Dynamo
- Apache Cassandra
- Hbase (ApacheHadoop)
- Twitter Gizzard
- weitere...

Jede NoSQL-Datenbank verfolgt seine eigene Ziele.

<sup>4</sup>MySQL vs. MongoDB: <http://www.computerwoche.de/a/datenbanksysteme-fuer-web-anwendungen-im-vergleich,2496589>, zugegriffen am 3. Januar 2016

### 3.1.1.1 Kategorien von NoSQL-Systemen

- Eine Key-Value-Datenbank (*Key-Value Store*) ist eine Datenbank, in der die Daten in Form von Schlüssel-Werte-Paaren abgespeichert werden. Der Schlüssel verweist dabei auf einen eindeutigen (meist in Binär- oder Zeichenketten-Format vorliegenden) Wert<sup>5</sup>. Value kann oft beliebiger Datentyp wie Arrays, Dokumente, Objekte, Bytes etc. sein.
- In einer spaltenorientierten Datenbank (*Column Store*), wie der Name vermuten lässt, werden die Datensätze spalten- statt zeilenweise abgespeichert. Durch die spaltenorientierte Abspeicherung der Daten wird der Lesezugriff stark beschleunigt, da keine unnötigen Informationen mehr gelesen werden, stattdessen nur diejenigen, die wirklich benötigt wurden. Dadurch wird der Schreibprozess aber erschwert, falls die schreibenden Daten aus mehreren Spalten bestehen werden, auf die entsprechend zugegriffen werden muss. Der Schreibprozess wird sich in diesem Fall etwas verlangsamen.
- Eine Graphen-Datenbank (*Graph database*) ist die weitere Kategorie aus der NoSQL Gruppe, in der die Daten anhand eines Graphen dargestellt und abgespeichert werden.

Die Graphen bestehen grundsätzlich aus Knoten (*Node*) und Kanten (*Edge*). Dabei stellen die Kanten die Verbindungen zwischen den einzelnen Knoten dar.

- Eine Datenbank, in der die Daten in Form von Dokumenten abgespeichert werden, ist als eine dokumentenorientierte Datenbank (*Document Store*) zu definieren. In diesem Zusammenhang ist ein Dokument als eine Zusammenstellung bestimmter Daten zu verstehen, das mit einem eindeutigen Identifikator angesprochen werden kann. Da die Daten in der dokumentenorientierten Datenbank nicht in Form von Tabellen, sondern in Form von Dokumenten abgespeichert werden, ergibt sich daraus keinen Strukturzwang.

---

<sup>5</sup>NoSQL: Key-Value-Datenbank Redis im Überblick: <https://www.heise.de/developer/artikel/NoSQL-Key-Value-Datenbank-Redis-im-Ueberblick-1233843.html>, zugegriffen am 17. Januar 2017

Möchte man ein bestimmtes Dokument erweitern, so kann man es einfach tun, da eine dokumentenorientierte Datenbank strukturfrei ist. Weitere Datenformate sind beispielsweise YAML<sup>6</sup> (angelehnt an XML) oder XML<sup>7</sup> selbst.

Bei dem Auswahl einer Datenbank für Foto-Verwaltungs-Service fiel die Entscheidung auf eine der NoSQL-Datenbank **MongoDB**. Laut DB-Ranking<sup>8</sup> ist die **MongoDB** einer der populärsten Datenbanken aus der NoSQL-Gruppe und passt ideal für Webprojekte mit *Big Data*.

## 3.2 MongoDB

**MongoDB** ist eine schemalose, dokumentenorientierte Open-Source-Datenbank. Der Name stammt von dem englischen Begriff *humONGous*, ins Deutsche als *gigantisch* oder *riesig* übersetzen lässt. Die genannte NoSQL-Datenbank macht mit seinem effizienten dokumentenorientierten Ansatz, einfacher Skalierbarkeit und hoher Flexibilität dem bewährten MySQL<sup>9</sup>-System zunehmend Konkurrenz.<sup>10</sup>

Relational	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Join	Embedding and Linking
Primary key	<i>_id</i> -Field (default)

Tab. 3.1: Konzepte im Vergleich

**MongoDB** präsentiert sich als eine quelloffene, dokumentenorientierte NoSQL-Datenbank mit den folgenden Konzepten wie *Ausfallsicherheit* und *horizontale Skalierung*, deren Bedeutung im Einzelnen in folgenden Unterkapiteln zu erläutern sind. Die Unterschiede zu

<sup>6</sup>YAML: <http://www.yaml.org/start.html>

<sup>7</sup>XML: <https://www.xml.com/>

<sup>8</sup>DB-Ranking: <http://db-engines.com/de/ranking>, zugegriffen am 13. März 2017

<sup>9</sup>MySQL: <https://www.mysql.com>

<sup>10</sup>MySQL vs. MongoDB: <http://www.computerwoche.de/a/datenbanksysteme-fuer-web-anwendungen-im-vergleich,2496589>, zugegriffen am 19. Januar 2017

den Konzepten der relationalen und nicht-relationalen Datenbanken konkret von **MongoDB** stellt die Tabelle 3.1 dar.

### 3.2.1 Datensätze in Form von Dokumenten

Die Datensätze werden in der NoSQL-Datenbank **MongoDB** in Dokumente gespeichert. **MongoDB** verwendet für die Dokumentenspeicherung und den Datenaustausch das sogenannte **BSON**<sup>11</sup>-Format, das eine binäre Darstellung von **JSON**-ähnlichen Dokumenten bietet. Nachfolgend sind alle für **BSON** definierten Datentypen aufgelistet:

- Double
- String
- Object
- Array
- Binary Data
- Undefined
- Object Id
- Boolean
- Date
- Null
- Regular Expression
- JavaScript
- Symbol
- JavaScript(with scope)
- 32-Bit Integer
- Timestamp
- 64-Bit Integer
- Min Key
- Max Key

Der Grund für die große Anzahl an Datentypen ist ein wesentliches Ziel der Entwickler von **BSON**: *Effizienz*.

Die Dokumente selbst werden von **MongoDB** in sogenannten Kollektionen (*Collections*) gespeichert, die grob mit den Tabellen einer relationalen Datenbank vergleichbar sind. Ein Zugriff auf Daten mehrerer Kollektionen (*Collections*) ist nicht möglich, wie es aus dem *Joins*-Konzept relationaler Datenbank bekannt ist. Die *CRUD*-Operationen sind auf Ebene der *Collection* durchzuführen (**Abs. 3.2.4**). Wie es schon aus dem Abs. ?? bekannt ist, kann jedes Dokument eine beliebige Anzahl an Feldern besitzen, unabhängig voneinander.

---

<sup>11</sup>BSON: <http://www.bjson.org>

Zudem dürfen Dokumente auch innerhalb eines Dokuments gespeichert werden<sup>12</sup>. Die Speicherung von Daten in Form von Dokumenten bietet den Vorteil, das sowohl strukturierte, als auch semi-strukturierte und polymorphe Daten gespeichert werden können. Dokumente, die jedoch das gleiche oder ein ähnliches Format haben, sollten zu einer Kollektion (*Collection*) zusammengefasst werden<sup>13</sup>.

### 3.2.2 Die Architektur

Zu den wichtigsten Eigenschaften, die für einen Einsatz von **MongoDB** sprechen, gehören:

- Verfügbarkeit: Auch bei Ausfall einer Datenbankinstanz soll die Applikation weiterhin verfügbar bleiben. (**Abs.. 3.2.8**)
- Skalierbarkeit: Mit Sharding (**Abs. 3.2.7**), einem Verfahren zur horizontalen Skalierung, kann der effiziente Umgang mit großen Datenmengen erreicht werden.<sup>14</sup>

### 3.2.3 Server/Client starten

Zum Starten des Server-Prozesses muss im Terminal der folgende Befehl ausgeführt werden:

Listing 3.1: Server-Prozess starten

```
vlfa:~ vlfa$ mongod
```

Mit dem Befehl (**List. 3.2**)

<sup>12</sup>Einführung in MongoDB: <https://www.iks-gmbh.com/assets/downloads/Einfuehrung-in-MongoDB-iks.pdf>, zugegriffen am 19. Januar 2017

<sup>13</sup>MongoDB: <http://www.moretechnology.de/mongodb-eine-dokumentenorientierte-datenbank/>, zugegriffen am 21. Januar 2017

<sup>14</sup>MongoDB Eigenschaften: <https://entwickler.de/online/datenbanken/mongodb-erfolgreich-ein-dokumentenorientiertes-datenbanksystem-einfuehren-115079.html>, zugegriffen am 12. Dezember 2016

**Listing 3.2: Client-Prozess starten**

```
vlfa:~ vlfa$ mongo
```

wird ein Client-Prozess gestartet, falls die *mongod*-Instanz aktiv ist. Nachdem das Client-Prozess gestartet ist, kann der Client an der Datenbank berühmte *CRUD*-Operationen (**Kap. 3.2.4**) ausführen.

Ohne irgendeine Konfiguration vornehmen zu müssen, verwendet **MongoDB**-Server von Anfang an als Default den TCP-IP-Port 27017 für eingehende Verbindungen.

**MongoDB** unterstützt auch eine HTML-basierte Administrationsoberfläche. Falls sie benötigt wird, ist es möglich, im Terminal mit dem folgenden Befehl die HTML-basierte Administrationsoberfläche zu starten:

**Listing 3.3: HTML-basierte Administrationsoberfläche starten**

```
vlfa:~ vlfa$ mongod --httpinterface --rest
```

und dementsprechend sie im beliebigen Browser der folgenden URL aufzurufen:

**Listing 3.4: HTML-basierte Administrationsoberfläche aufrufen**

```
http://localhost:28017/
```

Auf der Seite sind alle relevante Informationen zu Replikationsgruppen, Skalierung, verbundene Clients etc. zu finden.

### 3.2.4 CRUD = IFUR-Operationen

Die **CRUD**-Operationen aus SQL heißen in **MongoDB** *Insert*, *Find*, *Update* und *Remove*. Bei **MongoDB** ist es nicht einmal notwendig, eine Datenbank oder eine Collection zu definieren, bevor etwas gespeichert wird. Datenbanken und Collections werden zur Laufzeit beim ersten Einfügen eines Dokuments von MongoDB erzeugt.

### 3.2.4.1 Create/Insert

Für die Speicherung von neuen Dokumenten in der Datenbank bietet **MongoDB** drei Funktionen (**Listing 3.5**) an, welche als Parameter ein einzelnes oder eine Reihe von Dokumenten in einem Array annehmen.

#### Listing 3.5: Dokument(e) speichern

```
> db.<collection>.insert(<...>)
> db.<collection>.insertMany(<...>)
> db.<collection>.insertOne(<...>)
```

**MongoDB** generiert für jedes neues Dokument eine ID mit dem Feldnamen `_id`, falls keine konkrete Dokument-ID angegeben ist.

### 3.2.4.2 Read/Find

Zum Durchsuchen nach Dokumenten mit bestimmten Eigenschaften bietet **MongoDB** drei weitere Funktionen (**Listing 3.6**) an. Das Ergebnis beim Aufruf einer der folgenden Funktionen ist ein Cursor, der auf alle passenden Dokumente zeigt.

#### Listing 3.6: Dokument(e) finden

```
> db.<collection>.find(<...>)
> db.<collection>.findOne(<...>)
> db.<collection>.findOneAndDelete(<...>)
```

### 3.2.4.3 Update/Update

Die drei nächsten Funktionen (**Listing 3.7**) ermöglichen, anhand des Inhalts bestimmte Dokumente zu filtern und in diesen Änderungen durchzuführen. Die Änderungen umfassen das Hinzufügen, Entfernen oder Umbenennen von Feldern.

**Listing 3.7:** Dokument(e) aktualisieren

```
> db.<collection>.update(<...>)
> db.<collection>.updateMany(<...>)
> db.<collection>.updateOne(<...>)
```

**3.2.4.4 Delete/Remove**

Das Löschen von ganzen Dokumenten erfolgt in **MongoDB** anhand der folgenden Funktion (**Listing 3.8**), indem beim Aufruf entsprechende Informationen für die Dokumente angegeben werden.

**Listing 3.8:** Dokument(e) löschen

```
> db.<collection>.remove(<...>)
```

**3.2.5 Indizes**

Um die Laufzeit von Datenbankabfragen zu optimieren bzw. zu beschleunigen, können Indexe verwendet werden. Indexe in MongoDB werden als *B-Tree*-Datenstrukturen<sup>15</sup> verwaltet.

Neben dem obligatorischen Primär-Index auf dem Feld `_id` ist es möglich, beliebige Sekundärindexe anzulegen. Insgesamt erlaubt **MongoDB** pro Collection auf einzelnen Feldern oder einer Gruppe von Feldern bis zu 64 Indexe zu definieren.

Auf der Kommandozeile ist es möglich, das Administrationswerkzeug *Mongo Shell* zu verwenden, um mit einer Collection aus einer Datenbank verbinden zu können. Indexe anzulegen, ermöglicht **MongoDB** mit dem folgenden Befehl (**Listing 3.9**).

**Listing 3.9:** Index auf ein Feld anlegen

```
> db.<collection>.createIndex( {<feld>: 1} )
```

<sup>15</sup>B-Tree-Datenstrukturen: <https://de.wikipedia.org/wiki/B-Baum>, zugegriffen am 10. Februar 2017

### 3.2.6 Aggregation

**MongoDB** bietet eine Menge von Aggregationsoperationen an, die die Datensätze wunschmäßig verarbeiten und die berechneten Ergebnisse zurückliefern. Die Aggregationsoperationen gruppieren Werte aus mehreren Dokumenten zusammen. Des Weiteren ist es möglich, eine Vielzahl von Operationen auf den gruppierten Daten auszuführen, um ein einziges Ergebnis zurückzuliefern. **MongoDB** bietet drei Möglichkeiten, Datenaggregation durchzuführen. Diese sind

- Aggregation Framework
- Map/Reduce und
- Single purpose aggregation.<sup>16</sup>

In dieser Arbeit wird auf nur eine der drei Möglichkeiten eingegangen und nicht nur allgemein. Das ist Aggregation Framework. Durch den Ansatz an dem Prototyp (**Kap. ??**) wird das Aggregation Framework näher kennengelernt.

#### 3.2.6.1 Aggregation Framework

Analog zu *GROUP BY* in SQL hat **MongoDB** sein eigenes Konzept entwickelt, das im eigenen Aggregation Framework modelliert ist. Welche Möglichkeiten bietet **MongoDB**'s Aggregation Framework an, wird im Kapitel ?? mit Anwendungsfällen detailliert erläutert. Die einzelnen Aggregationsoperationen mit entsprechender Beschreibung sind aus der Tabelle 3.2 zu entnehmen. Die Datenaggregation durch das Aggregation Framework ist natürlich nicht nur auf *Shell*-Ebene, sondern auch in vielen gängigen Programmiersprachen, wie zum Beispiel Java, C++, C#, PHP, Python etc. durch bereitgestellte **MongoDB**'s Treiber<sup>17</sup> möglich.

---

<sup>16</sup>Aggregation: <https://docs.mongodb.com/manual/aggregation/>, zugegriffen am 1. März 2017

<sup>17</sup>MongoDB Drivers: <https://docs.mongodb.com/ecosystem/drivers/>, zugegriffen am 18. Januar 2017

MongoDB	SQL	Beschreibung
\$match	WHERE, HAVING	Der <code>\$match</code> -Operator funktioniert nach dem gleichen Prinzip wie <code>db.&lt;collection&gt;.find()</code> .
\$group	GROUP BY	Der <code>\$group</code> -Operator gruppiert berechnete Ergebnisse nach bestimmten Feldern.
\$skip	-	Der <code>\$skip</code> -Operator ermöglicht, eine bestimmte Anzahl an Dokumenten zu überspringen.
\$limit	-	Der <code>\$limit</code> -Operator formuliert eine konkrete Anzahl an zurücklieferenden Dokumenten.
\$sort	ORDER BY	Der <code>\$sort</code> -Operator sortiert Dokumente.
\$project	SELECT	Der <code>\$project</code> -Operator ermöglicht, die Form der berechneten Ergebnisse zu manipulieren, bzw. das zurücklieferende Ergebnis wunschmäßig zu formen-
\$unwind	-	Der <code>\$unwind</code> -Operator dekonstruiert ein Array-Feld aus einem Dokument, falls so ein Array-Feld existiert-

Tab. 3.2: Aggregationsoperationen

### 3.2.7 Horizontale Skalierung (Sharding)

Um eine kostengünstige Lösung für eine Steigerung der Leistung von Systemen zu ermöglichen, ermöglicht das Datenbanksystem von **MongoDB** eine horizontale Skalierung, die allgemein im Teilabschnitt 2.0.1 schon diskutiert ist. Die horizontale Verteilung der Daten erfolgt bei **MongoDB** auf Ebene der *Collections* nach *Sharding-Keys*, siehe Abbildung 3.2. Die *Sharding-Keys* (**Abs. 3.2.7.1**) dienen dazu, um später Zugriffe auf einzelne Dokumente zu ermöglichen.

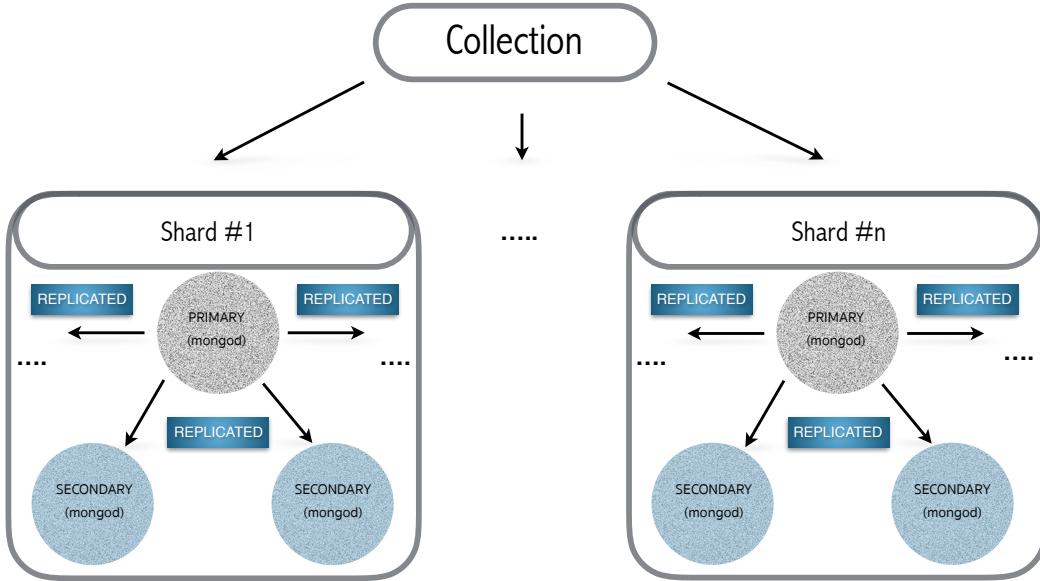


Abb. 3.2: Ein Beispiel für Verteilung einer *Collection* auf mehreren *Shards*

Die Aufteilung der *Collections* erfolgt in Blocks, auch als *Chunks* genannt. Ein *Chunk* ist ein Teil einer bestimmten *Collection*. Gespeichert werden *Chunks* auf Servern, die in diesem Zusammenhang als *Shards* bezeichnet werden. Was es unter *Shards* zu verstehen ist, erläutert der Teilabschnitt 3.2.8 zur Replikation.

Um die Aufteilung der *Collections* in *Chunks* auf *Shards* realisieren zu können, verwendet **MongoDB** folgende Komponenten:

- *shards*: Die *Shards* enthalten letztendlich die Daten. In einer *Shard* ist es möglich, Replikationsgruppen zu verwenden, näher dazu im Teilabschnitt 3.2.8.
- *mongos*: Der *mongos* gilt als ein *RoutingService*, der die Anfragen der Anwendungsschicht entgegennimmt und diese an eine entsprechende *Shard* weiterleitet, die die nötigen Daten enthält.
- *config servers*: Die Konfigurationsserver speichern die Metadaten für einen Sharded-Cluster. Im Fall einer Schreiboperation entscheiden die Konfigurationsserver, in welchen Chunk auf welchem Shard das entsprechende Dokument eingefügt wird. Bei der Leseoperation geben die Konfigurationsserver den Auskunft darüber, welcher

Shard die gewünschten Daten enthält. Bei den Konfigurationsservern handeln es um eine *mongod*-Instanzen.

Die folgende Abbildung 3.3 veranschaulicht die Interaktion von den gerade genannten Komponenten innerhalb eines Sharded-Cluster:

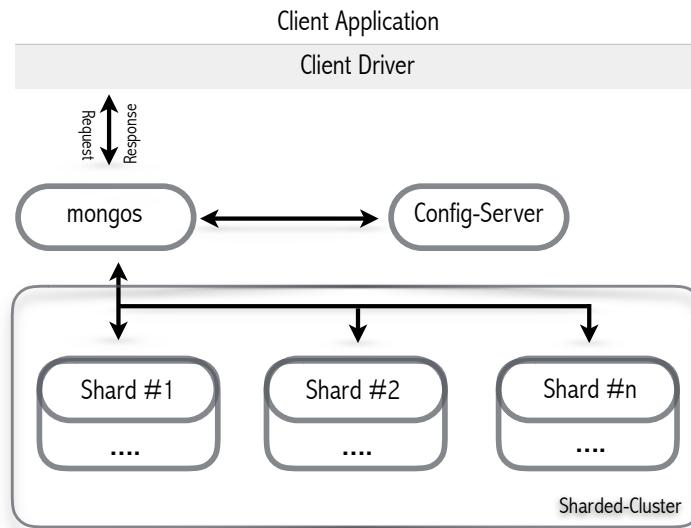


Abb. 3.3: Horizontale Skalierung (*Sharding*)

Das Ziel des Ganzen ist die horizontale Skalierbarkeit an Datenmengen, um die Performance des Datenbanksystems zu steigern.

### 3.2.7.1 Fragmentierung des Datenbestands nach *Shard-Keys*

Die Fragmentierung des Datenbestands erfolgt auf Ebene der *Collections* nach *Shard-Keys*. In jeder *Collection* muss ein Schlüssel als sogenannter *Sharding-Key* definiert sein, der entsprechend in jedem Dokument derselben *Collection* existiert. *Sharding-Key* kann entweder aus einem einzigen indexierten Feld oder einem zusammengesetzten Index bestehen. Die Dokumente werden dann nach *Shard-Key* alphabetisch oder nummerisch sortiert und anschließend in  $n$ -Blocks gleicher Größe eingeteilt. **MongoDB** garantiert die gleichmäßige Verteilung der Blocks an *Shards*.

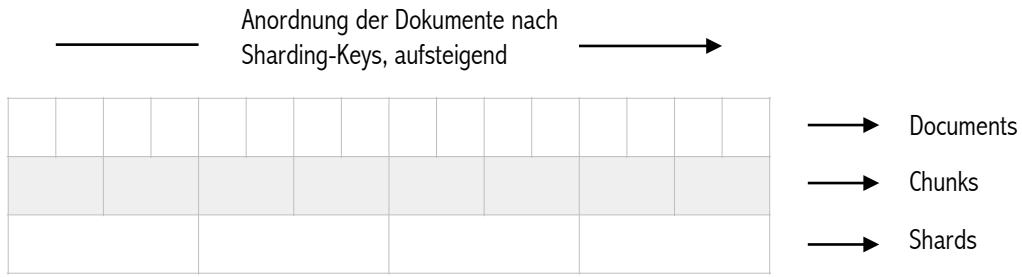


Abb. 3.4: Anordnung der Dokumente in Blocks (=Chunks) unter Verwendung des *Shard-Keys*. Mehrere Blocks bilden dementsprechend eine *Shard*.

Bei der *Shard-Keys* Konfiguration müssen folgende *Constraints*<sup>18</sup> berücksichtigt werden:

- *Shard-Keys* sind unabänderlich
- *Shard-Keys* verfügen über eine hohe Kardinalität
- *Shard-Keys* sind eindeutig
- *Shard-Keys* existiert dann in jedem Dokument
- *Shard-Keys* ist bis zum 512 bytes limitiert
- *Shard-Keys* ist es nicht möglich, als Multi-Key zu bilden

### 3.2.8 Replikation (Replication)

Manchmal kann es dazu kommen, dass ein Server ausfällt und die Schreib- und Lesezugriffe dadurch auf eine kurze Zeit nicht mehr möglich sind. Um Schreib- und Lesezugriffe auch im Fall eines Serverausfallen ständig ermöglichen zu können, hat **MongoDB** einen Replikationsmechanismus entwickelt. Der Replikationsmechanismus dient zur Replikation bzw. zum Spiegeln der Daten auf mehreren Servern und funktioniert nach einem *Master-n-Slaves-Prinzip*.

<sup>18</sup>Introduction to Sharding: [https://www.mongodb.com/presentations/back-to-basics-4-introduction-to-sharding?p=589c5c940aca4c55041426f1&utm\\_campaign=Int\\_WB\\_Back%20to%20Basics%20Series%20%28English%29\\_01\\_17\\_EMEA%20-%20Follow%20Up%204&utm\\_medium=email&utm\\_source=Eloqua](https://www.mongodb.com/presentations/back-to-basics-4-introduction-to-sharding?p=589c5c940aca4c55041426f1&utm_campaign=Int_WB_Back%20to%20Basics%20Series%20%28English%29_01_17_EMEA%20-%20Follow%20Up%204&utm_medium=email&utm_source=Eloqua), zugegriffen am 2. Februar 2017

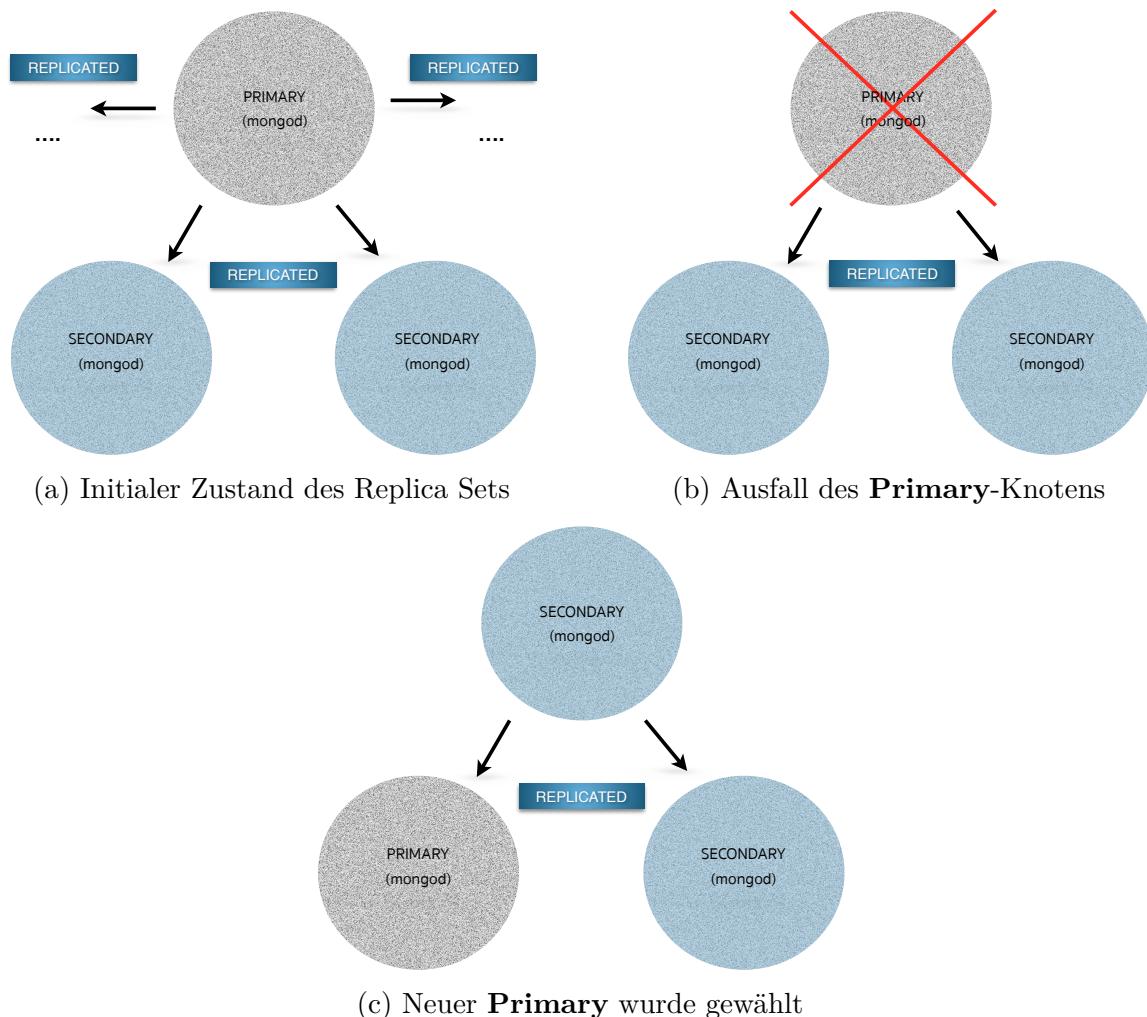


Abb. 3.5: Szenario für eine Replikationsgruppe mit drei Servern in einer *Shard*

Ein *Master*, auch ein *Primary* genannt, besitzt Schreib- und Leserechte. Dieser repliziert die Daten auf *n-Slaves*, die auch als *Secondaries* bezeichnet werden. Ein *Primary* mit *n-Secondaries* bilden gemeinsam eine *Shard*. Eine *Shard* kann aus mind. einem Server bestehen. Falls eine *Shard* aus mehreren Servern besteht, so kann **MongoDB** die Server in Replikationsgruppe (*Replica set*) anordnen, damit bei Ausfall eines Servers die Verfügbarkeit der Datenbank trotzdem gewährleistet ist. Mit Replikationsgruppen will **MongoDB** die Ausfallsicherheit sicherstellen. Die Abbildung 3.5 veranschaulicht ein Szenario für eine Replikationsgruppe mit drei Knoten. Jeder Knoten aus der Gruppe ist als einen eigenen Server vorzustellen. Das *Master-n-Slaves-Prinzip* besagt, dass in einer Replikationsgruppe nur ein Master und n-Slaves existieren können, um eine strenge Konsistenz gewährleisten

zu können.

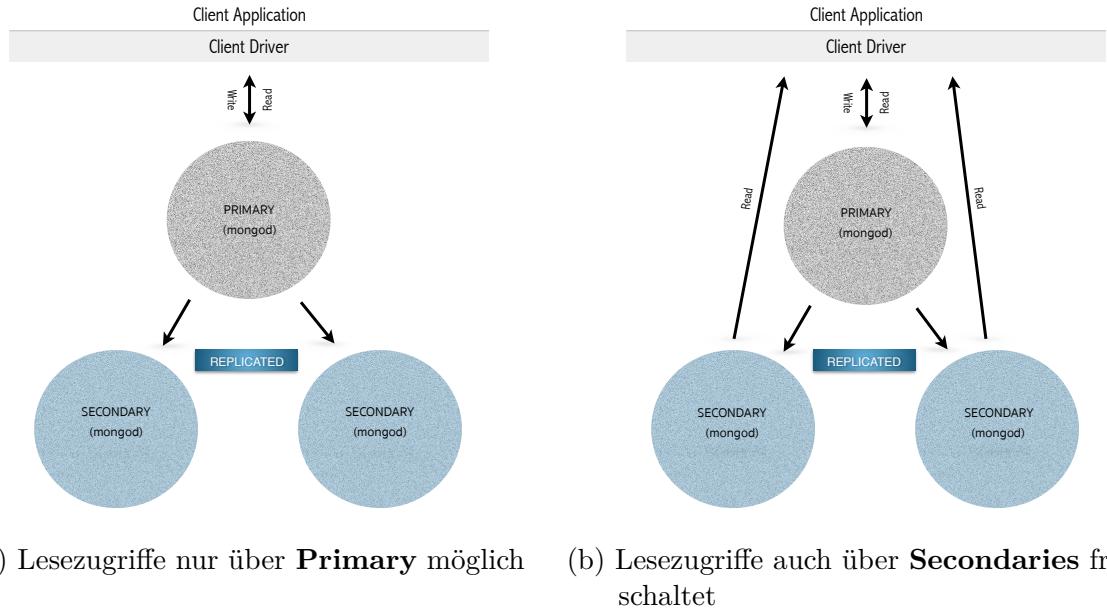


Abb. 3.6: Freischaltung der Lesezugriffe

Im Gegenteil zu dem Primary sind bei Secondaries die Schreib- und Leserechte von Anfang an nicht möglich. Falls der Kontext der Anwendung verlangt, können nur die Leserechte entsprechend freigeschaltet werden.

### 3.2.8.1 Eine Replikationsgruppe erzeugen

In diesem Teilabschnitt wird eine Replikationsgruppe mit insgesamt drei Servern erzeugt. Jeder Schritt der Konfiguration wird in diesem Teilabschnitt nachgespielt.

### 3.2.9 MongoDB mit Java

## 3.3 Apache Cassandra

In diesem Kapitel wird ein weiterer wichtiger Vertreter der NoSQL-Datenbanken vorgestellt, nämlich **Apache Cassandra**. Wieso ist **Apache Cassandra** eigentlich? **Apache**

**Cassandra** ist laut DB-Ranking<sup>19</sup> die beliebteste spaltenorientierte Datenbank und nach **MongoDB** die beliebteste NoSQL-Datenbank.

### 3.3.1 Allgemein

**Apache Cassandra** war ursprünglich eine proprietäre Datenbank von Facebook und wurde 2008 als Open-Source-Datenbank veröffentlicht. Konzipiert ist **Apache Cassandra** als skalierbares, ausfallsicheres System für den Umgang mit großen Datenmengen auf verteilten Systemen (Clustern) und im Gegensatz zu **MongoDB** (C++) in Java geschrieben.

Zunächst werden die Konzepte von **Apache Cassandra** allgemein diskutiert und dann die Unterschiede zu der schon besprochenen NoSQL-Datenbank **MongoDB** gezeigt.

Im Vergleich zu der **MongoDB**-Datenbank, die in Replikation nach dem Master-Slave-Prinzip funktioniert, verfolgt **Apache Cassandra** komplett anderes Prinzip. Dieses Prinzip wird in der Architektur der **Apache Cassandra** erläutert.

### 3.3.2 Architektur

- **Apache Cassandra** ist nach peer-to-peer<sup>20</sup> verteiltes System aufgebaut. Ein peer-to-peer verteiltes System beschreibt ein Cluster, bestehend aus mehreren gleichberechtigten Knoten.
- Jeder Knoten ist in so einem Cluster dezentral, unabhängig und akzeptiert sowohl Schreib- als auch Leseoperationen.
- Falls irgendeiner Knoten aus dem definierten Cluster ausfällt, werden die Schreib- und Leseanforderungen von anderen verfügbaren Knoten bedient.

**Apache Cassandra** stellt die Verfügbarkeit und Partitionstoleranz über die Konsistenz.

---

<sup>19</sup>DB-Ranking: <http://db-engines.com/de/ranking>, zugegriffen am 13. März 2017

<sup>20</sup>Peer-to-Peer-Netze (P2P) sind Netze, bei denen alle Knoten im Netz dezentral sind.

### 3.3.3 Datenmodell

Die Hauptbestandteile des Datenmodells von **Apache Cassandra** veranschaulicht die Abbildung 3.7.

- Cluster
- Keyspace
- Keys
- Columns
- Column Family sowie
- Super Columns

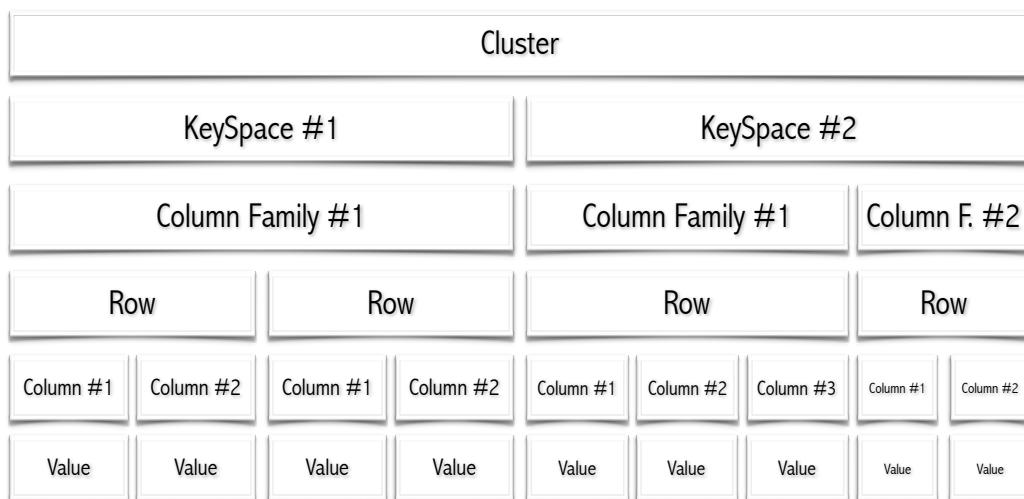


Abb. 3.7: Datenmodell

**Apache Cassandra** definiert einen Datenbankserver als ein Cluster, auf dem mehrere Datenbanken (Keyspaces) angelegt werden. Eine Spaltenfamilie (Column Family) entspricht einer Tabelle und enthält Zeilen (Rows), welche mit einer eindeutigen Id zu identifizieren sind. In Zeilen (Rows) werden die Datensätze gespeichert, wobei jede Zeile bis zu 2 Milliarden Spalten (Columns) enthalten kann. Die Spalten (Columns) dagegen enthalten jeweils ein Paar „Schlüssel-Wert“ (Key-Value-Paar). Um bei Leseoperationen einen effizienteren

Zugriff erreichen zu können, müssen die Spalten (Columns) Super Columns definieren, indem mehrere Spalten zusammengesetzt werden.

## 3.4 Logikschicht (Backend)

### 3.4.1 Rest Server

### 3.4.2 Spring MVC

## 3.5 Präsentationsschicht (Frontend)

AngularJS 2

## 3.6 Frameworks

Allgemeine Architektur Überblick Datenschicht Nosql

Frameworks

# 4 Implementierung

## 4.1 Fachliche Spezifikation

## 4.2 3-Schichten-Architektur

### 4.2.1 Präsentationsschicht (Frontend)

Vorhanden ist, dass Frontend die Liste von Photos, dargestellt als List von byteArrays (jeweils ein byteArray stellt ein Photo dar) bekommt und diese im Frontend anzeigt. Besser ist, nur die LIste von Photos-Ids zu bekommen und Session in Frontend einschalten. Der User erhält dann nur einen Teil von Photos, falls gewünscht den weiteren Teil etc.

### 4.2.2 Logikschicht (Backend)

### 4.2.3 Datenhaltungsschicht (Datenbank)

# **5 Prototyp: Testen auf Cluster (optional)**

## **5.1**

# Zusammenfassung

blabla

# Literaturverzeichnis

- [1] S. Edlich. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. 2., aktualisierte und erw. Aufl. München: Hanser, 2011.
- [2] A. Hollosi. *Von Geodaten bis NoSQL: leistungsstarke PHP-Anwendungen: Aktuelle Techniken und Methoden für Fortgeschrittene*. München: Hanser, 2012.
- [3] O. Kurowski. *CouchDB mit PHP*. Frankfurt am Main: Entwickler.press, 2012.

# Abbildungen

2.1	CAP-Theorem	6
2.2	Workflow zum MVC-Konzept	12
2.3	Observer Pattern	14
3.1	Architektur-Prototyp	16
3.2	Ein Beispiel für Verteilung einer <i>Collection</i> auf mehreren <i>Shards</i>	27
3.3	Horizontale Skalierung ( <i>Sharding</i> )	28
3.4	Anordnung der Dokumente in Blocks (= <i>Chunks</i> ) unter Verwendung des <i>Shard-Keys</i> . Mehrere Blocks bilden dementsprechend eine <i>Shard</i> .	29
3.5	Szenario für eine Replikationsgruppe mit drei Servern in einer <i>Shard</i>	30
3.6	Freischaltung der Lesezugriffe	31
3.7	Datenmodell	33

# Tabellen

3.1 Konzepte im Vergleich . . . . .	19
3.2 Aggregationsoperationen . . . . .	26

# Quelltextverzeichnis

3.1	Server-Prozess starten	21
3.2	Client-Prozess starten	22
3.3	HTML-basierte Administrationsoberfläche starten	22
3.4	HTML-basierte Administrationsoberfläche aufrufen	22
3.5	Dokument(e) speichern	23
3.6	Dokument(e) finden	23
3.7	Dokument(e) aktualisieren	24
3.8	Dokument(e) löschen	24
3.9	Index auf ein Feld anlegen	24