

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

Waldstr. 7, 82024 Taufkirchen

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

Eidesstattliche Erklärung	I
1 Einleitung	1
1.1 Motivation und Ziel der Arbeit	1
2 Skalierbarkeit und Wartbarkeit	3
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	7
2.1 Wartbarkeit	7
2.1.1 Dependency Injection (DI)	8
2.1.1.1 Ziel	8
2.1.2 MVC-Pattern	8
2.1.2.1 Ziel	9
2.1.2.2 Workflow	9
2.1.2.3 Beobachter Muster	10
2.1.2.3.1 Idee	11
3 Architektur	12
3.1 Überblick	12
3.2 Datenschicht	12
3.2.1 NoSQL	12
3.2.1.1 Kategorien von NoSQL-Systemen	12
3.3 Frameworks	12

4 Implementierung	13
4.1 Fachliche Spezifikation	13
4.2 3-Schichten-Architektur	13
4.2.1 Präsentationsschicht (Frontend)	13
4.2.2 Logikschicht (Backend)	13
4.2.3 Datenhaltungsschicht (Datenbank)	13
5 Prototyp: Testen auf Cluster (optional)	14
Zusammenfassung	15
6 Datenhaltungsschicht	16
6.1 Allgemein	16
6.1.1 ACID-Prinzip	16
6.1.2 Skalierung	17
6.1.2.1 Vertikale Skalierung	18
6.1.2.2 Horizontale Skalierung	18
6.2 NoSQL-Datenbanken	19
6.2.1 Was ist NoSQL?	19
6.2.2 Das CAP-Theorem	20
6.2.3 BASE	22
6.2.4 Arten von NoSQL-Datenbanken	23
6.2.4.1 Key-Value-Datenbanken	23
6.2.4.2 Spaltenorientierte Datenbanken	24
6.2.4.3 Graphen-Datenbanken	24
6.2.4.4 Dokumentenorientierte Datenbanken	25
6.3 MongoDB	26
6.3.1 Datensätze in Form von Dokumenten	27
6.3.2 Die Architektur	28
6.3.3 Server/Client starten	28
6.3.4 CRUD = IFUR-Operationen	30
6.3.4.1 Create/Insert	30
6.3.4.2 Read/Find	30
6.3.4.3 Update/Update	31
6.3.4.4 Delete/Remove	31

6.3.5	Indizes	31
6.3.6	Aggregation	32
6.3.6.1	Aggregation Framework	32
6.3.7	Horizontale Skalierung (Sharding)	33
6.3.7.1	Fragmentierung des Datenbestands nach <i>Shard-Keys</i>	35
6.3.8	Replikation (Replication)	36
6.3.8.1	Eine Replikationsgruppe erzeugen	38
6.3.9	MongoDB mit Java	38
6.4	Apache Cassandra	38
6.4.1	Allgemein	39
6.4.2	Architektur	39
6.4.3	Datenmodell	40

Literaturverzeichnis**A****Abbildungen****B****Tabellen****C**

1 Einleitung

Heutzutage erwarten die Internetnutzer kurze Ladezeiten, flüssige Bedienung und ständige Verfügbarkeit von den Web-Applikationen, die zur Verfügung stehen. Dabei sind viele davon nicht in der Lage, mit rasant steigender Anzahl von Anfragen und großen Datenmengen effizient umzugehen.

Es ist ein ernsthaftes Problem für erfolgreiche Projekte, die rasant populär werden und ein exponentielles Anwenderwachstum erleben. Um von dem Projekterfolg zu profitieren und den auszubauen, es ist überlebenswichtig diesen Wachstumsanforderungen gerecht zu werden. Die Hardware ist heutzutage kein großes Problem mehr, die Cloud-Services wie *Amazon* oder *Microsoft Azure* ermöglichen den Zugang zu den fast unbegrenzten Hardwareressourcen. Die Herausforderung besteht darin, die Web-Applikation so zu bauen, dass die von diesem Hardwareangebot Gebrauch machen kann. Die Wartung- und Erweiterungsfähigkeit sind zwei weitere wichtige Punkte, die für den Erfolg unabdingbar sind. Um die Internetnutzer zu halten, 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 Abschlussarbeit ist eine Architektur vorzuschlagen, die die Entwicklung der skalierbaren, wartungs- und erweiterungsfähigen Web-Applikationen ermöglicht. Es wird weiterhin 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 eine Prototyp implementiert und eigene Erfahrungen aus dem Entwicklungsprozess berichtet. Für den Prototyp wurde der webbasierte Foto-Verwaltungs-Service gewählt. Diese

Applikation zeichnet sich dadurch aus, dass jeder Internetnutzer eigene Daten unabhängig von den anderen verwalten kann, was bei vielen Webanwendungen der Fall ist. Andererseits sollen nicht nur triviale Textdaten verwaltet werden, sondern auch mehrere Dateien.

2 Skalierbarkeit und Wartbarkeit

In diesem Kapitel wird der Begriff Skalierbarkeit erklärt und die Kompromisse erläutert, die bei der Entwicklung einer verteilten skalierbaren Applikation eingegangen werden müssen. Die **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 Applikation mit austauschbaren Komponenten ermöglicht.

2.0.1 Skalierbarkeit

Der Begriff *Skalierbarkeit* beschreibt die Fähigkeit eines Systems, aufgrund der wachsenden Anforderungen, entweder die Leistung der vorhandenen Ressourcen zu verbessern oder zusätzlich die neuen Ressourcen hinzufügen. Das System, bei dem die neuen Ressourcen hinzugefügt werden, nennt man *verteilte Systeme*.

Bei der Skalierung sind zwei Arten zu unterscheiden, eine *vertikale* und *horizontale Skalierung*, die es zunächst näher zu erläutern gilt.

2.0.2 Vertikale Skalierbarkeit

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

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

Die horizontale Skalierbarkeit (*scale-out*), im Gegensatz zur vertikalen Skalierung verteilt die Daten auf verschiedenen Knoten im großen Cluster, wobei die quantitative Steigerung der Leistungsfähigkeit angestrebt wird. Somit können mehrere weniger leistungsfähigere, 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. Horizontale Skalierbarkeit ist auch günstiger - je leistungsfähiger ist der Rechner, desto teurer ist seine Erweiterung. Allerdings, da die Daten in dem Cluster verteilt sind, die Entwicklung eines verteilten Systems unterscheidet sich von den klassischen Applikationen, die auf einer Maschine laufen. Die *Trade-offs* einer verteilten Applikation wurden 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.

- **Atomicity (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 durchgeföhrte 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 Integritätsbedingungen¹ zu gewährleisten.

¹Unter 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.

- 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.

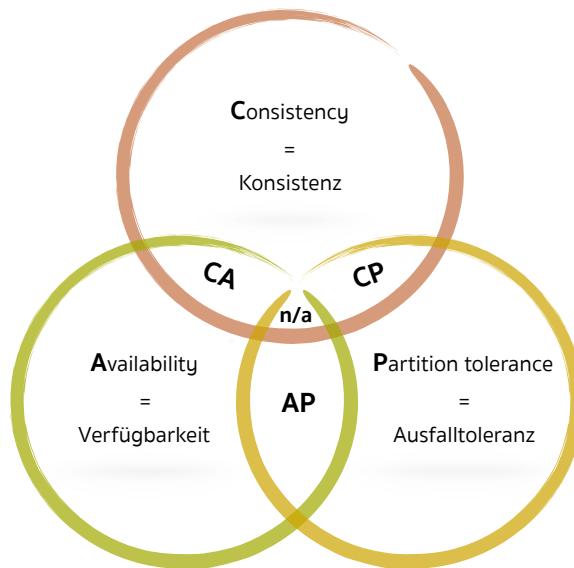


Abb. 2.1: Anforderungen an verteilte Systeme gemäß dem **CAP**-Theorem

Das Akronym **CAP** steht für die englischsprachigen Begriffe **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz) und be-

schreiben die Anforderungen für die Skalierung an verteilte Systeme, die es zunächst näher zu erläutern gilt.

- **Consistency** (Konsistenz): Die *Konsistenz* beschreibt einen konsistenten Zustand an Daten in einem verteilten System. Bedeutet, dass nach Abschluss einer Transaktion nicht nur die manipulierenden Datensätze, sondern auch alle replizierenden Knoten in einem großen Cluster über die gleichen Daten verfügen. Falls ein Wert auf einem Knoten durch eine Transaktion per Schreiboperation geändert wird, muss der aktualisierte Wert auf Anfrage mit der Leseoperation von anderen Knoten zurückgeliefert werden können. Die Transaktion selbst ist eine atomare² Einheit in der Datenbank.
- **Availability** (Hochverfügbarkeit): Die *Hochverfügbarkeit* ist die weitere Anforderung, die besagt, dass immer alle gesendeten Anfragen durch User ans System beantwortet werden müssen und mit einer akzeptablen Reaktionszeit.
- **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 fehlerfrei weiter funktioniert. Falls einzelne Knoten in so einem System ausfallen, wird deren Ausfall von 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 6.2 zu entnehmen. Wie die Abbildung 6.2 erkennen lässt, können in einem verteilten System gleichzeitig und vollständig nur zwei dieser drei Anforderungen **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit), **Partition Tolerance** (Partitionstoleranz) erfüllt sein. 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 Applikation muss daher individuell entschieden werden, ob sie als ein **CA**-, **CP**- oder **AP**-System zu realisieren ist.

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

- **CA** (Consistency und Availability): Die klassischen relationalen Datenbankmanagementsysteme (RDBMS) wie Oracle, DB2 etc. fallen in **CA**-Kategorie, die vor allem **Consistency** (Konsistenz) und **Availability** (Hochverfügbarkeit) aller Knoten in einem Cluster hinzielt. 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 Applikationen, die zu der **CP**-Kategorie zu ordnen sind, sind Banking-Applikationen. Für solche Applikationen ist es wichtig, dass die Transaktionen zuverlässig durchgeführt werden und der mögliche Ausfall eines Knotens sichergestellt wird.
- **AP** (Availability und Partition tolerance): Für die Applikationen, die in die **AP**-Kategorie fallen, rückt die Anforderung **Consistency** (Konsistenz) in den Hintergrund. Beispiele für solche Applikationen sind die Social-Media-Sites wie Twitter oder Facebook, da die Hauptidee der Applikation dadurch nicht verfällt, wenn zum gleichen Zeitpunkt die replizierten Knoten nicht über die gleiche Datenstruktur verfügen.

2.0.6 BASE

BASE steht für **B**asically **A**vailable, **S**oft State, **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

blablabla

2.1.1 Dependency Injection (DI)

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 Web-Applikation aufzustellen. Für die Wartbarkeit der Applikation sind die Grundlagen für das *Dependency Injection (DI)* Pattern unvermeidbar. Das Prinzip *Dependency Injection (DI)* wird bei vielen Frameworks wie zum Beispiel Google Guice³, Dagger⁴ etc. umgesetzt. Im Kapitel der Implementierung (**Kap. 4**) wird die Umsetzung der losen Kopplung sowohl im Präsentationsschicht durch **AngularJS 2**, als auch im Logikschicht durch **Spring Framework** veranschaulicht.

2.1.1.1 Ziel

Der Einsatz des *Dependency Injection (DI)* Pattern ermöglicht den Entwicklern, der Arbeitsaufwand für die Entwicklung großer Applikationen stark zu reduzieren. Bei seinem Einsatz wird eine lose Kopplung der Applikationskomponenten erreicht, die dem Entwickler die Konzentration auf die Entwicklung einzelner Komponenten unabhängig voneinander ermöglicht. Die Unabhängigkeit der Programmteile erleichtert dem Entwickler nicht nur die Applikationskomponente unabhängig voneinander zu entwickeln, sondern auch diese leichter zu testen.

2.1.2 MVC-Pattern

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

³Google Guice, <https://github.com/google/guice>, zugegriffen am 03.02.2017

⁴Dagger, <http://square.github.io/dagger/>, zugegriffen am 03.02.2017

2.1.2.1 Ziel

Das Ziel des **MVC**-Musters ist Geschäftslogik einer Applikation von der Benutzerschnittstelle abzutrennen, so dass Entwickler einen Bereich bequem verändern kann und der Rest der Applikation 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.

2.1.2.2 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.

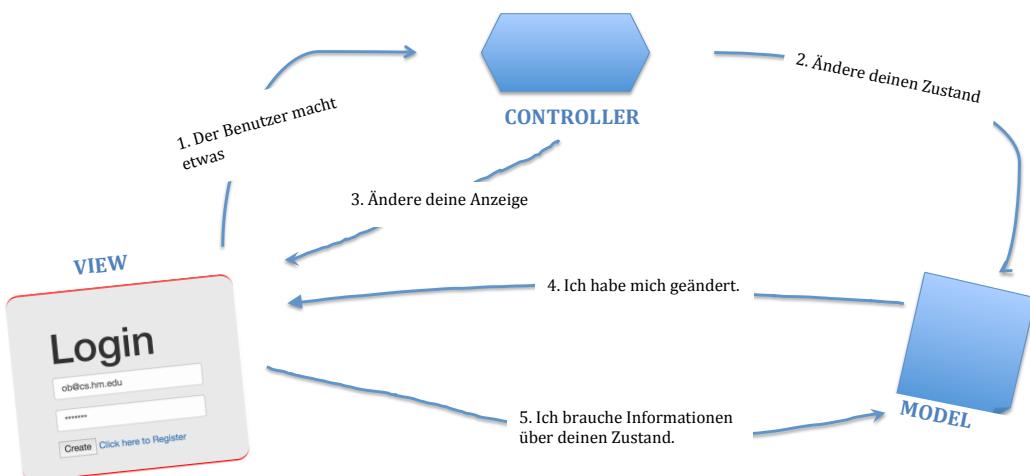


Abb. 2.2: Workflow zum MVC-Konzept

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. 2.1.2.3**).

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. 2.1.2.3**).

2.1.2.3 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.

2.1.2.3.1 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.

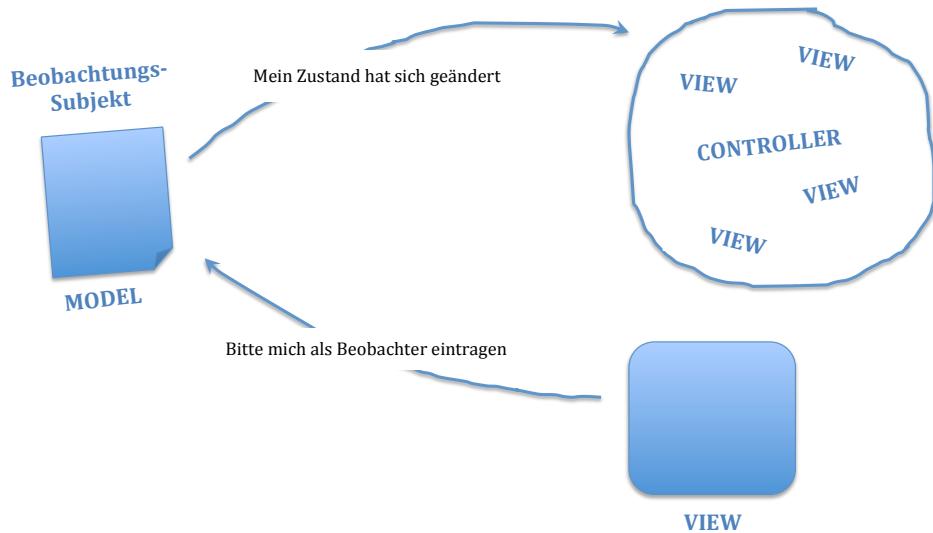


Abb. 2.3: Observer Pattern

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

Graph mit allen Tools, Architektur in Graph zeigen
Beschreibung alles Frameworks, die in Prototyp verwendet werden
DI in Backend und DI in Frontend, reference auf Kapitel Theorie->DI

3.1 Überblick

3.2 Datenschicht

3.2.1 NoSQL

3.2.1.1 Kategorien von NoSQL-Systemen

3.3 Frameworks

Allgemeine Architektur Überblick Datenschicht
Nosql

Frameworks

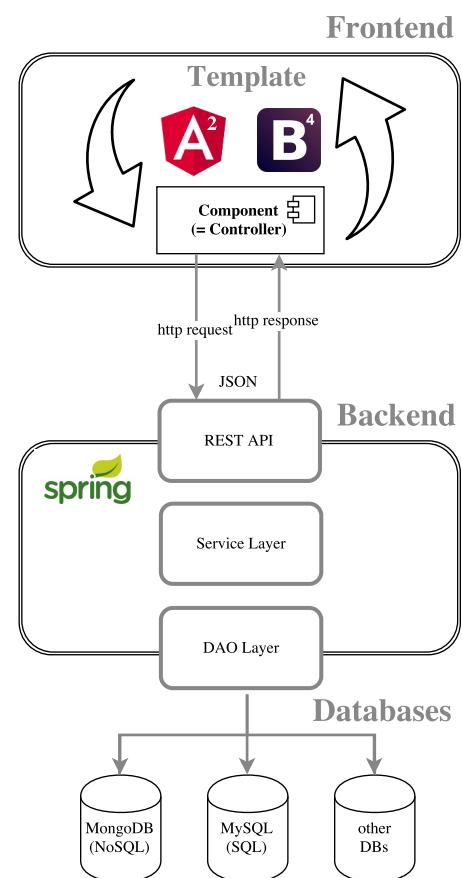


Abb. 3.1: Architektur-Prototyp

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

6 Datenhaltungsschicht

Jede Anwendung verarbeitet heutzutage sehr große Datenmengen. blabla

6.1 Allgemein

blablabla

6.1.1 ACID-Prinzip

Fehlt noch warum ACID, wozu ACID im allgemeinen....., NUR FÜR RELATIONALE DATENBAN

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.

- **Atomicity (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.
- **Consistency (Konsistenz):** Die *Konsistenz* besagt, dass vor und auch nach dem Ablauf einer Transaktion die Integrität und Plausibilität der Datenbestände ge-

währleistet werden. Die Integrität der Datenbank ist es möglich, beispielsweise mit Integritätsbedingungen¹ zu gewährleisten.

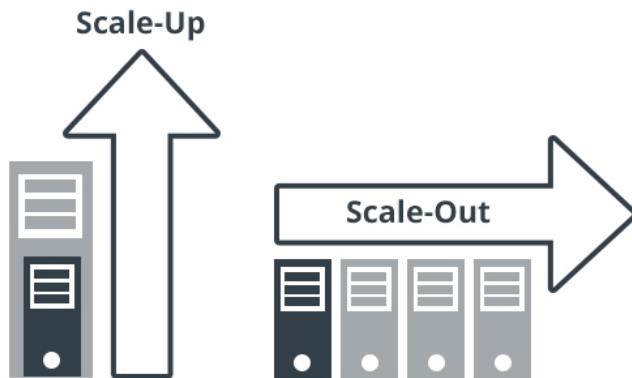
- **Isolation** (Isolation): Die *Isolation* dient zu Kapselung von Transaktionen, um unerwünschte Nebenwirkungen vermeiden zu können. Die Transaktionen müssen unabhängig voneinander ablaufen.
- **Durability** (Dauerhaftigkeit): Die *Dauerhaftigkeit* gewährleistet nach einer erfolgreichen Transaktion 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.

6.1.2 Skalierung

Der Begriff *Skalierung* beschreibt die Fähigkeit eines Systems, aufgrund der wachsenden Anforderungen, entweder die Leistung der vorhandenen Ressourcen zu verbessern oder zusätzlich die Neuen hinzufügen. In der Regel sind die verteilten Systeme besser skalierbar, da mit solchen System die Nebenläufigkeit von Prozessen realisiert wird und somit die Leistungsfähigkeit der verwendeten Ressourcen steigt. Bei der Skalierung sind zwei Arten zu unterscheiden, eine vertikale und horizontale Skalierung, die es zunächst näher zu erläutern gilt.

Ergänzung: GRAPH für vertikale und horizontale zeichnen und hinzufügen, schaue Beispiel im img-Ordner.....!

¹Unter 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. 6.1: Skalierung²

6.1.2.1 Vertikale Skalierung

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

6.1.2.2 Horizontale Skalierung

Die horizontale Skalierung (*scale-out*), im Gegensatz zur vertikalen Skalierung verteilt die Daten auf verschiedenen Knoten im großen Cluster, wobei die quantitative Steigerung der Leistungsfähigkeit angestrebt wird. Somit können mehrere weniger leistungsfähigere, nicht so teure Rechner eingesetzt werden. Dadurch ist es möglich, sehr große horizontale Skalierbarkeit zu gewährleisten, da die Knoten somit nicht so stark, im Vergleich zur vertikalen Skalierung überlastet sind.

²Skalierung: <https://magazin.kapilendo.de/den-supergau-verhindern-so-bereiten-sie-ihre-website-auf-einen-besucheransturm-vor/>, zugegriffen am 15. Januar 2017

6.2 NoSQL-Datenbanken

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.

6.2.1 Was ist NoSQL?

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.³ Dokumentdatenbanken gruppieren die Daten in einem strukturierten Dokument, typischerweise in einer JSON-Datenstruktur. Auch **MongoDB**, siehe dazu Abschnitt 6.3 verfolgt diesen Ansatz 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...

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

Jede NoSQL-Datenbanke verfolgt seine Ziele und welche sie genau verfolgen, beschreibt der Abschnitt 6.2.4, in dem verschiedene Kategorien von NoSQL-Datenbanken vorgestellt werden.

6.2.2 Das CAP-Theorem

Im Jahr 2000 hielt Brewer⁴ die Keynote auf dem ACM Symposium on Principles of Distributed Computing (PODC)⁵, einer Konferenz über die Grundlagen der Datenverarbeitung in verteilten Systemen⁶ (Principles of Distributed Computing). In seiner Keynote stellte Brewer sein **CAP**-Theorem vor, ein Ergebnis seiner Forschungen zu verteilten Systemen an der University of California [3, S. 13]. Brewer's Theorem wurde im Jahr 2002 von Seth Gilbert und Nancy Lynch formal bewiesen.

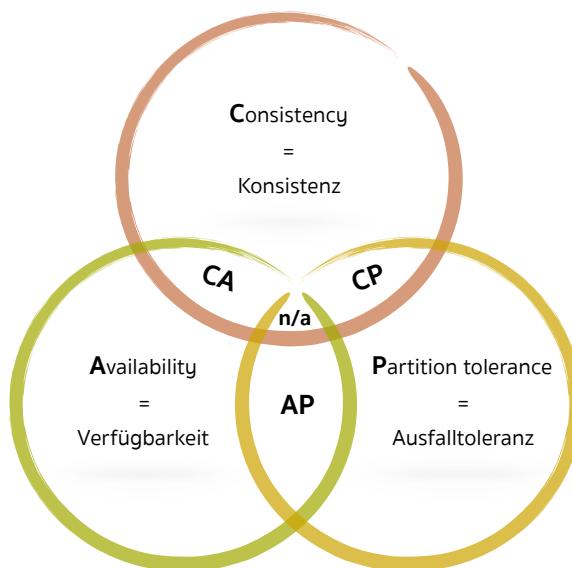


Abb. 6.2: Anforderungen an verteilte Systeme gemäß dem **CAP**-Theorem

Das Akronym **CAP** steht für die englischsprachigen Begriffe **Consistency** (Konsistenz),

⁴Eric A. Brewer ist ein Informatik-Professor an der University of California, Berkeley und einer der Erfinder der Suchmaschine Inktomi

⁵PODC2000: <http://www.podc.org/podc2000/>, zugegriffen am 02.01.2017

⁶In einem verteilten System im Bereich Datenverarbeitung werden gespeicherte Daten mehrfach über mindestens zwei verschiedene Server repliziert und miteinander synchronisiert, um die Verfügbarkeit der Daten zu erhöhen und die Zugriffszeiten der User zu verringern.

Availability (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz) und beschreiben die Anforderungen für die Skalierung an verteilte Systeme, die es zunächst näher zu erläutern gilt.

Was besagt eigentlich dieses **CAP**-Theorem? Das **CAP**-Theorem besagt, dass die verteilten Systeme, die mit großen Datenmengen zu tun haben, gleichzeitig die folgenden Anforderungen wie **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz) nicht erfüllen können.

- **Consistency** (Konsistenz): Die *Konsistenz* der Daten in einem verteilten System wird im Prinzip genauso geschätzt wie im Abschnitt 2.0.4 die schon besprochene **Atomicity** (Atomarität)-Eigenschaft. Bedeutet, dass alle replizierenden Knoten aus einem großen Cluster über die gleichen Daten verfügen. Falls ein Wert auf einem Knoten durch eine Transaktion per Schreiboperation geändert wird, muss der aktualisierte Wert auf Anfrage mit der Leseoperation von anderen Knoten zurückgeliefert werden können. Die Transaktion selbst ist eine atomare⁷ Einheit in der Datenbank.
- **Availability** (Hochverfügbarkeit): Die *Hochverfügbarkeit* ist die weitere Anforderung, die besagt, dass immer alle gesendeten Anfragen durch User ans System beantwortet werden müssen und mit einer akzeptablen Reaktionszeit.
- **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 fehlerfrei weiter funktioniert. Falls einzelne Knoten in so einem System ausfallen, wird deren Ausfall von 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 6.2 zu entnehmen. Wie die Abbildung 6.2 erkennen lässt, können in einem verteilten System gleichzeitig und vollständig nur zwei dieser drei Anforderungen **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit), **Partition Tolerance** (Partitionstoleranz) erfüllt sein. Konkret aus der Praxis bedeutet das, dass es für eine hohe Verfügbarkeit und Partitions-

⁷Eine atomare Transaktion bedeutet, dass sie entweder ganz oder gar nicht ausgeführt wird. Falls eine atomare Transaktion abgebrochen wird, werden alle im Laufe der Transaktion schon durchgeführte Änderungen rückgängig gemacht.

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 Anwendung 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 **Consistency** (Konsistenz) und **Availability** (Hochverfügbarkeit) aller Knoten in einem Cluster hinzielt. 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 Anwendungen, die zu der **CP**-Kategorie zu ordnen sind, sind Banking-Anwendungen. Für solche Anwendungen ist es wichtig, dass die Transaktionen zuverlässig durchgeführt werden und der mögliche Ausfall eines Knotens sichergestellt wird.
- **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 Twitter⁸ oder Facebook⁹, da die Hauptidee der Anwendung dadurch nicht verfällt, wenn zum gleichen Zeitpunkt die replizierten Knoten nicht über die gleiche Datenstruktur verfügen.

6.2.3 **BASE**

BASE steht für **Basically Available**, **Soft State**, **Eventually Consistent** und beschreibt den Gegenteil zu den strengen **ACID**-Kriterien aus dem Teilabschnitt 2.0.4. **BASE** ist wie **CAP**-Theorem (Teilabschnitt 2.0.5) auch für verteilte Datenbanksysteme formuliert,

⁸Twitter: <https://twitter.com/>

⁹Facebook: <https://www.facebook.com/>

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.

6.2.4 Arten von NoSQL-Datenbanken

Eigene Graphen als Abbildungen verwenden, diese sind nur als Beispiel drin.....! Der folgende Abschnitt stellt insgesamt vier Kategorien von NoSQL-Datenbanken dar (Abb. 6.3).

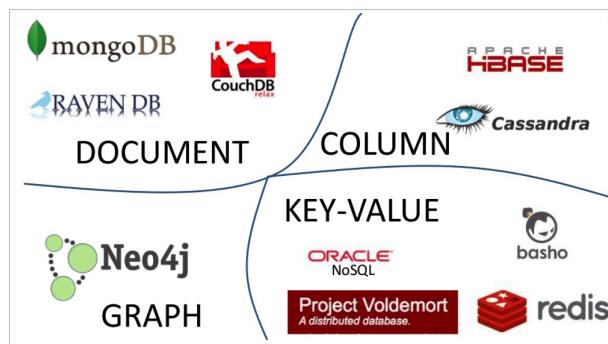


Abb. 6.3: NoSQL-Datenbanken, verteilt in vier Gruppen¹⁰

Die **MongoDB** wird im Abschnitt 6.3 detailliert beschrieben. BlaBlaBla

6.2.4.1 Key-Value-Datenbanken

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) Wert¹¹. Value

¹⁰NoSQL-Datenbanken: <http://bigdata-blog.com/key-value-database>, zugegriffen am 17. Januar 2017

¹¹NoSQL: 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

kann oft beliebiger Datentyp wie Arrays, Dokumente, Objekte, Bytes etc. sein. Siehe dazu die Abbildung 6.4 mit einem Beispiel:

Example: Key-Value Store	
Key	Value
Mahesh	{"Mathematics, Science, History, Geography"}
Uma	{"English, Hindi, French, German"}
Paul	{"Computers, Programming"}
Abraham	{"Geology, Metallurgy, Material Science"}

Abb. 6.4: Key-Value-Datenbank als Beispiel¹²

6.2.4.2 Spaltenorientierte Datenbanken

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.

Abbildung==?????

6.2.4.3 Graphen-Datenbanken

Eine Graphen-Datenbank (*Graph database*) ist weitere Kategorie aus der NoSQL Gruppe, in der die Daten anhand eines Graphen dargestellt und abgespeichert werden.

¹²NoSQL: A Silver Bullet for handling Big Data?: <https://www.linkedin.com/pulse/nosql-silver-bullet-handling-big-data-shashank-dhaneshwar>, zugegriffen am 17. Januar 2017

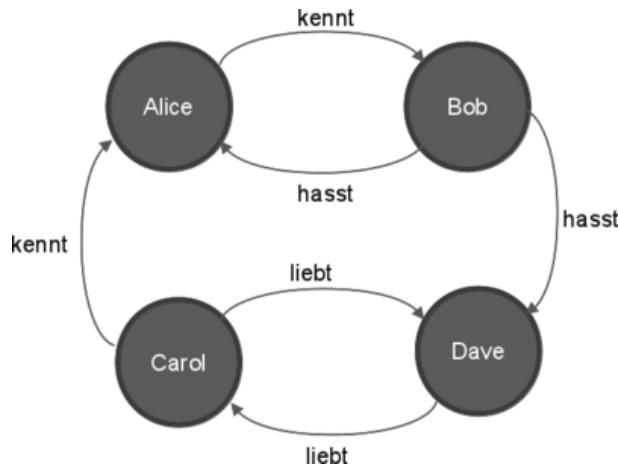


Abb. 6.5: Beispiel für die Darstellung der Daten in einer Graphen-Datenbank¹³

Wie der Abbildung 6.5 zu entnehmen ist, bestehen Graphen grundsätzlich aus Knoten (*Node*) und Kanten (*Edge*). Dabei stellen die Kanten die Verbindungen zwischen den einzelnen Knoten dar.

6.2.4.4 Dokumentenorientierte Datenbanken

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. Als Beispiel ist aus der Abbildung 6.6 zwei Dokumente im **JSON**-Format zu entnehmen, die sich voneinander gänzlich unterscheiden.

¹³Beispiel für die Darstellung der Daten in einer Graphen-Datenbank: <https://de.wikipedia.org/wiki/Graphdatenbank>, zugegriffen am 17. Januar 2017

```
{
  "city" : "FISHERS ISLAND",
  "loc" : [
    -72.017834,
    41.263934
  ],
  "pop" : 329,
  "state" : "NY",
  "_id" : "06390"
}

{
  "_id" : ObjectId("50b1aa983b3d0043b51b2c52"),
  "name" : "Nexus 7",
  "category" : "Tablets",
  "manufacturer" : "Google",
  "price" : 199
}
```

Abb. 6.6: Zwei Dokumente im JSON Format

Möchte man ein bestimmtes Dokument erweitern, so kann man es einfach tun, da eine dokumentenorientierte Datenbank strukturfrei ist. Weitere Datenformate sind beispielsweise YAML¹⁴ (angelehnt an XML) oder XML¹⁵ selbst.

6.3 MongoDB

MongoDB ist eine schemalose, dokumentenorientierte Open-Source-Datenbank und gehört somit zu einer der im Teilabschnitt 6.2.4 besprochenen NoSQL-Datenbankarten. 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 MySQL¹⁶-System zunehmend Konkurrenz.¹⁷

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

¹⁴YAML: <http://www.yaml.org/start.html>

¹⁵XML: <https://www.xml.com/>

¹⁶MySQL: <https://www.mysql.com>

¹⁷MySQL vs. MongoDB: <http://www.computerwoche.de/a/datenbanksysteme-fuer-web-anwendungen-im-vergleich,2496589>, zugegriffen am 19. Januar 2017

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

Tab. 6.1: Konzepte im Vergleich

den Konzepten der relationalen und nicht-relationalen Datenbanken konkret von **MongoDB** stellt die Tabelle 6.1 dar.

6.3.1 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**¹⁸-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.

¹⁸BSON: <http://www.bjson.org>

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. 6.3.4**). Wie es schon aus dem Abs. 6.2.4 bekannt ist, kann jedes Dokument eine beliebige Anzahl an Feldern besitzen, unabhängig voneinander.

Zudem dürfen Dokumente auch innerhalb eines Dokuments gespeichert werden¹⁹. 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 werden²⁰.

6.3.2 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.. 6.3.8**)
- Skalierbarkeit: Mit Sharding (**Abs. 6.3.7**), einem Verfahren zur horizontalen Skalierung, kann der effiziente Umgang mit großen Datenmengen erreicht werden.²¹

6.3.3 Server/Client starten

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

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

²⁰MongoDB: <http://www.moretechnology.de/mongodb-eine-dokumentenorientierte-datenbank/>, zugegriffen am 21. Januar 2017

²¹MongoDB Eigenschaften: <https://entwickler.de/online/datenbanken/mongodb-erfolgreich-ein-dokumentenorientiertes-datenbanksystem-einfuehren-115079.html>, zugegriffen am 12. Dezember 2016

Listing 6.1: Server-Prozess starten

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

Mit dem Befehl (**List. 6.2**)

Listing 6.2: 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. 6.3.4**) 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 6.3: HTML-basierte Administrationsoberfläche starten

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

und dementsprechend sie im beliebigen Browser der folgenden URL aufzurufen:

Listing 6.4: HTML-basierte Administrationsoberfläche aufrufen

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

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

6.3.4 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.

6.3.4.1 Create/Insert

Für die Speicherung von neuen Dokumenten in der Datenbank bietet **MongoDB** drei Funktionen (**Listing 6.5**) an, welche als Parameter ein einzelnes oder eine Reihe von Dokumenten in einem Array annehmen.

Listing 6.5: Dokument(e) speichern

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

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

6.3.4.2 Read/Find

Zum Durchsuchen nach Dokumenten mit bestimmten Eigenschaften bietet **MongoDB** drei weitere Funktionen (**Listing 6.6**) an. Das Ergebnis beim Aufruf einer der folgenden Funktionen ist ein Cursor, der auf alle passenden Dokumente zeigt.

Listing 6.6: Dokument(e) finden

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

6.3.4.3 Update/Update

Die drei nächsten Funktionen (**Listing 6.7**) ermöglichen, anhand des Inhalts bestimmte Dokumente zu filtern und in diesen Änderungen durchzuführen. Die Änderungen umfassen das Hinzufügen, Entfernen oder Umbenennen von Feldern.

Listing 6.7: Dokument(e) aktualisieren

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

6.3.4.4 Delete/Remove

Das Löschen von ganzen Dokumenten erfolgt in **MongoDB** anhand der folgenden Funktion (**Listing 6.8**), indem beim Aufruf entsprechende Informationen für die Dokumente angegeben werden.

Listing 6.8: Dokument(e) löschen

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

6.3.5 Indizes

Um die Laufzeit von Datenbankabfragen zu optimieren bzw. zu beschleunigen, können Indexe verwendet werden. Indexe in MongoDB werden als *B-Tree*-Datenstrukturen²² 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.

²²B-Tree-Datenstrukturen: <https://de.wikipedia.org/wiki/B-Baum>, zugegriffen am 10. Februar 2017

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 6.9**).

Listing 6.9: Index auf ein Feld anlegen

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

6.3.6 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.²³

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.

6.3.6.1 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 6.2 zu entnehmen. Die Datenaggregation durch das Aggregation Framework ist

²³Aggregation: <https://docs.mongodb.com/manual/aggregation/>, zugegriffen am 1. März 2017

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 Treiber²⁴ möglich.

MongoDB	SQL	Beschreibung
\$match	WHERE, HAVING	Der <code>\$match</code> -Operator funktioniert nach dem gleichen Prinzip wie <code>db.<collection>.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. 6.2: Aggregationsoperationen

6.3.7 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 6.7. Die *Sharding-Keys* (**Abs. 6.3.7.1**) dienen dazu, um später Zugriffe auf einzelne Dokumente zu ermöglichen.

²⁴MongoDB Drivers: <https://docs.mongodb.com/ecosystem/drivers/>, zugegriffen am 18. Januar 2017

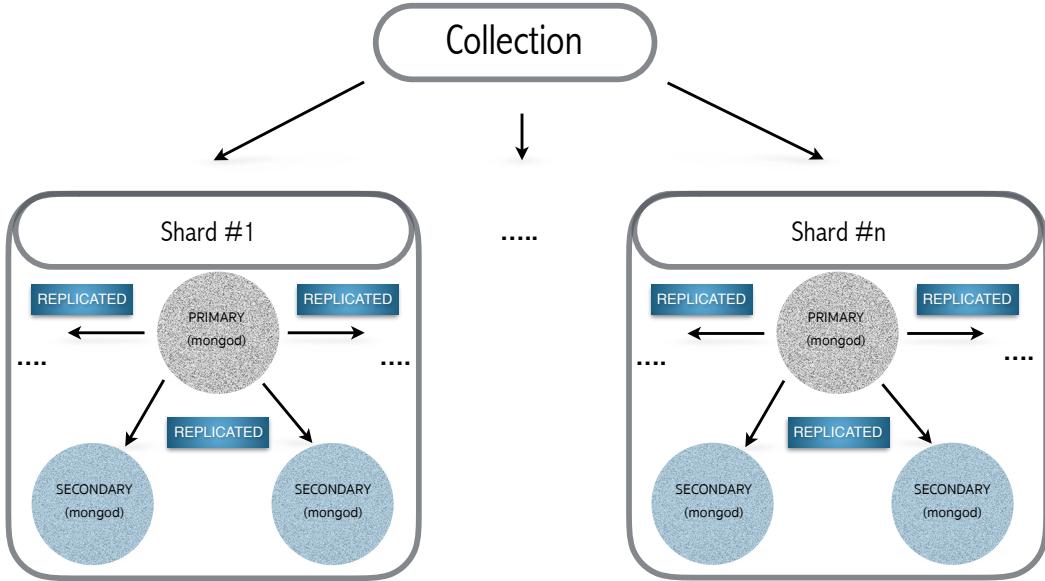


Abb. 6.7: Ein Beispiel für Verteilung einer *Collection* auf mehreren *Shards*

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 6.3.8 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 6.3.8.
- *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 6.8 veranschaulicht die Interaktion von den gerade genannten Komponenten innerhalb eines Sharded-Cluster:

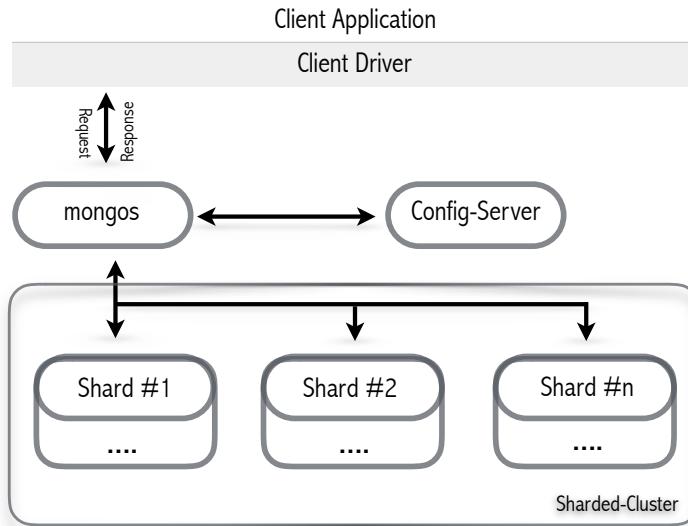


Abb. 6.8: Horizontale Skalierung (*Sharding*)

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

6.3.7.1 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*.

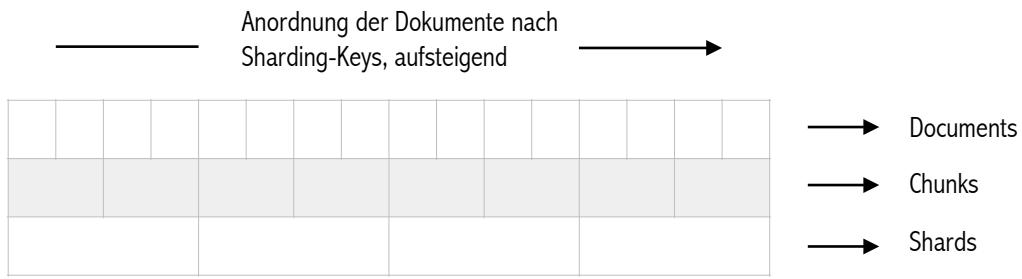


Abb. 6.9: Anordnung der Dokumente in Blocks (=Chunks) unter Verwendung des *Shard-Keys*. Mehrere Blocks bilden dementsprechend eine *Shard*.

Bei der *Shard-Keys* Konfiguration müssen folgende *Constraints*²⁵ 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

6.3.8 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*.

²⁵Introduction 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, zugegriffen am 2. Februar 2017

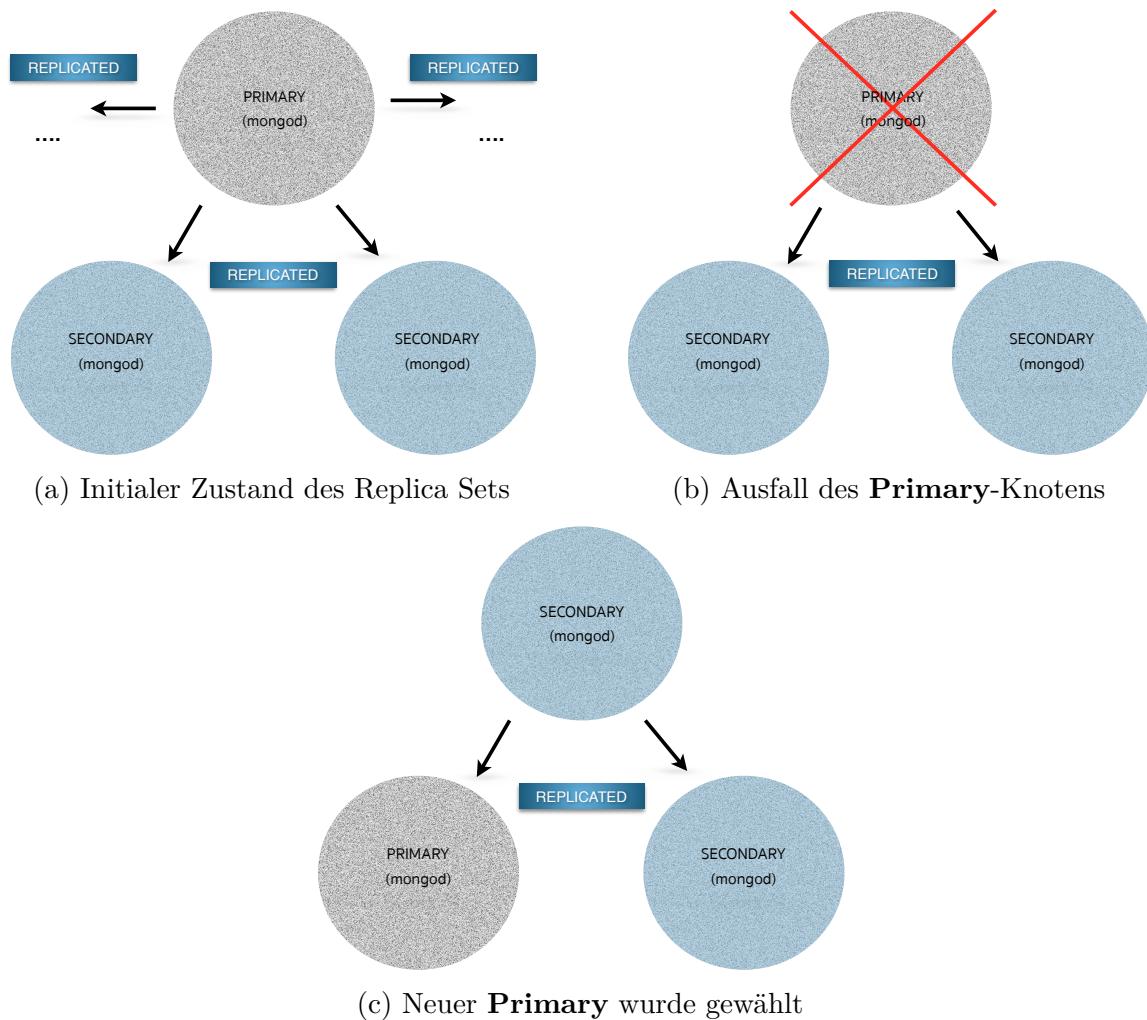


Abb. 6.10: Szenario für eine Replikationsgruppe mit drei Servern in einer *Shard*

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 6.10 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.

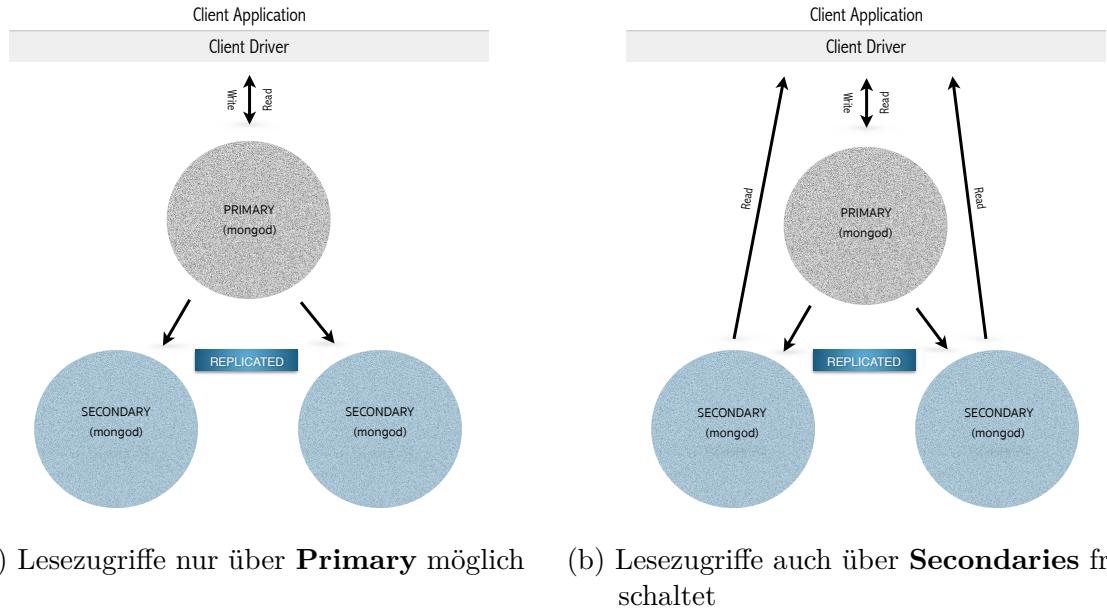


Abb. 6.11: Freischaltung der Lesezugriffe

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.

6.3.8.1 Eine Replikationsgruppe erzeugen

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

6.3.9 MongoDB mit Java

6.4 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-Ranking²⁶ die beliebteste spaltenorientierte Datenbank und nach **MongoDB** die beliebteste NoSQL-Datenbank.

6.4.1 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.

6.4.2 Architektur

- **Apache Cassandra** ist nach peer-to-peer²⁷ 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.

²⁶DB-Ranking: <http://db-engines.com/de/ranking>, zugegriffen am 13. März 2017

²⁷Peer-to-Peer-Netze (P2P) sind Netze, bei denen alle Knoten im Netz dezentral sind.

6.4.3 Datenmodell

Die Hauptbestandteile des Datenmodells von **Apache Cassandra** veranschaulicht die Abbildung 6.12.

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

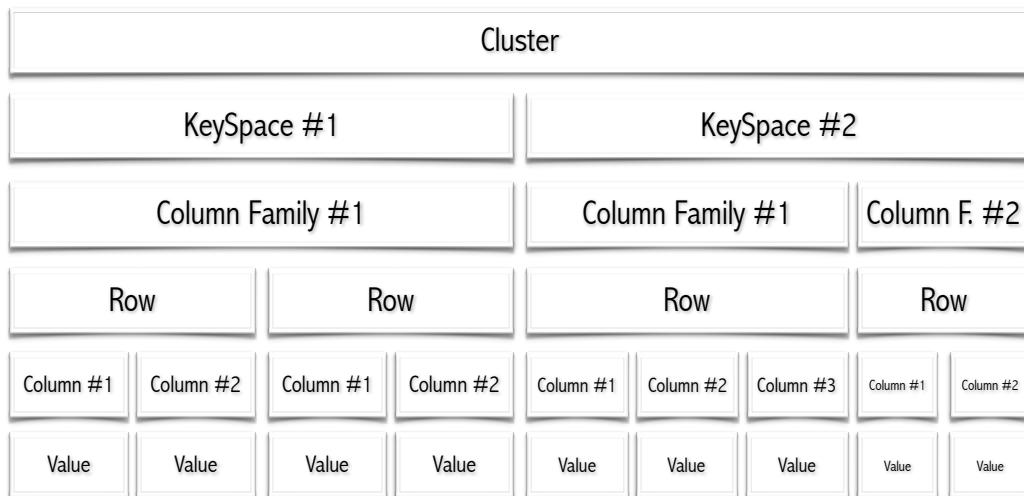


Abb. 6.12: Datenmodell

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.

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	5
2.2	Workflow zum MVC-Konzept	9
2.3	Observer Pattern	11
3.1	Architektur-Prototyp	12
6.1	Skalierung	18
6.2	CAP-Theorem	20
6.3	NoSQL-Datenbanken	23
6.4	NoSQL: A Silver Bullet for handling Big Data?	24
6.5	Beispiel für die Darstellung der Daten in einer Graphen-Datenbank	25
6.6	Zwei Dokumente im JSON Format	26
6.7	Ein Beispiel für Verteilung einer <i>Collection</i> auf mehreren <i>Shards</i>	34
6.8	Horizontale Skalierung (<i>Sharding</i>)	35
6.9	Anordnung der Dokumente in Blocks (= <i>Chunks</i>) unter Verwendung des <i>Shard-Keys</i> . Mehrere Blocks bilden dementsprechend eine <i>Shard</i> .	36
6.10	Szenario für eine Replikationsgruppe mit drei Servern in einer <i>Shard</i>	37
6.11	Freischaltung der Lesezugriffe	38
6.12	Datenmodell	40

Tabellen

6.1 Konzepte im Vergleich	27
6.2 Aggregationsoperationen	33

Quelltextverzeichnis

6.1	Server-Prozess starten	29
6.2	Client-Prozess starten	29
6.3	HTML-basierte Administrationsoberfläche starten	29
6.4	HTML-basierte Administrationsoberfläche aufrufen	29
6.5	Dokument(e) speichern	30
6.6	Dokument(e) finden	30
6.7	Dokument(e) aktualisieren	31
6.8	Dokument(e) löschen	31
6.9	Index auf ein Feld anlegen	32