

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 Theorie</b>	<b>3</b>
2.1 Skalierbarkeit und Wartbarkeit . . . . .	3
2.1.1 Skalierbarkeit . . . . .	3
2.1.1.1 Vertikale Skalierbarkeit . . . . .	3
2.1.1.2 Horizontale Skalierbarkeit . . . . .	4
2.1.1.3 ACID-Prinzip . . . . .	4
2.1.1.4 Das CAP-Theorem . . . . .	5
2.1.1.5 Das BASE-Prinzip . . . . .	8
2.1.2 Wartbarkeit . . . . .	8
2.1.2.1 Die SOLID-Prinzipien . . . . .	9
2.1.2.2 Dependency Injection (DI) . . . . .	10
2.1.2.3 Dependency Injection und Mock-Objekte . . . . .	11
2.1.2.4 MVC-Pattern . . . . .	11
2.1.2.4.1 Workflow . . . . .	12
2.1.2.4.2 Beobachter Muster . . . . .	13
2.2 Relevante Technologien . . . . .	14
2.2.1 NoSQL-Datenbanken . . . . .	14
2.2.1.1 MongoDB . . . . .	16
2.2.1.1.1 Datensätze in Form von Dokumenten . . . . .	17
2.2.1.1.2 CRUD = IFUR-Operationen . . . . .	17
2.2.1.1.3 Indizes . . . . .	18

2.2.1.1.4	Aggregation . . . . .	19
2.2.1.1.5	Aggregation Framework . . . . .	19
2.2.1.1.6	Replikation (Replication) . . . . .	20
2.2.1.1.7	Horizontale Skalierung (Sharding) . . . . .	22
2.2.1.1.8	Fragmentierung nach <i>Shard-Keys</i> . . . . .	24
2.2.1.1.9	GridFS . . . . .	25
2.2.1.1.10	Treiber für MongoDB . . . . .	26
2.2.1.2	Apache Cassandra . . . . .	26
2.2.1.2.1	Architektur . . . . .	26
2.2.1.2.2	Datenmodell . . . . .	27
2.2.2	Spring Framework . . . . .	28
2.2.2.1	Module des Spring Frameworks . . . . .	28
2.2.2.2	Konfiguration mit Maven . . . . .	29
2.2.3	<b>REpresentational State Transfer</b> (REST) . . . . .	30
2.2.4	AngularJS 2 - JavaScript Framework . . . . .	31
<b>3</b>	<b>Architektur</b>	<b>32</b>
3.1	Konzept . . . . .	32
3.1.1	Backend . . . . .	33
3.1.2	Datenbank . . . . .	34
3.2	<b>Umsetzung</b> . . . . .	34
<b>Fazit</b>		<b>35</b>
<b>Literaturverzeichnis</b>		<b>A</b>
<b>Onlinequellen</b>		<b>B</b>
<b>Abbildungen</b>		<b>D</b>
<b>Tabellen</b>		<b>E</b>

# 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. Andernfalls sollen nicht nur triviale Textdaten verwaltet werden, sondern auch mehrere Dateien.

# 2 Theorie

## 2.1 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.

### 2.1.1 Skalierbarkeit

Der Begriff *Skalierbarkeit* [1] beschreibt die Fähigkeit eines Systems, aufgrund der wachsenden Anforderungen, entweder die Leistung der vorhandenen Ressourcen zu verbessern oder zusätzlich die neuen Ressourcen hinzufügen.

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

#### 2.1.1.1 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.

### 2.1.1.2 Horizontale Skalierbarkeit

Im Gegensatz zur vertikalen Skalierung wird bei der *horizontalen Skalierbarkeit (scale-out)* die Last auf zusätzliche Rechner verteilt. Die Anwendung wird auf einem verteilten System (Cluster) ausgeführt. In dem verteilten System können mehrere weniger leistungsfähige, nicht so teure Rechner eingesetzt werden. Die Daten müssen auf die Knoten im Cluster verteilt werden und bei manchen Anwendungen ist die Synchronisation der Zwischenergebnissen notwendig. Dabei entstehen zusätzliche Kommunikationskosten, weil die Kommunikation per Netzwerk viel teurer ist als Lesen vom Arbeitsspeicher. Deswegen wäre es falsch, die zusammengesetzte Leistung eines Clusters mit Leistung eines leistungsstarken Rechners zu vergleichen. Jedoch kommt es immer auf die Anwendung an, wie viel davon überhaupt parallelisierbar und wie viel Synchronisation notwendig ist. Ein Cluster kann sehr groß werden und wenn auch jeder zusätzlicher Core in dem Cluster nicht so viel bringt im Vergleich zu dem, was ein zusätzlicher Core in einer Applikation auf einem Rechner bringen würde, dadurch dass es im Cluster viel Ressourcen verfügbar sind, können die Applikationen besser skalieren.

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 formalisiert.

### 2.1.1.3 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 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.

#### 2.1.1.4 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.

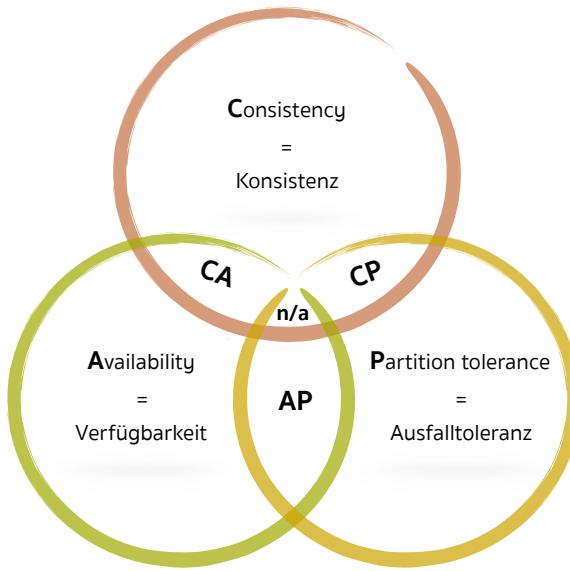


Abb. 2.1: Anforderungen an verteilte Systeme gemäß dem **CAP**-Theorem

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 [2, 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.

#### 2.1.1.5 Das BASE-Prinzip

**BASE** steht für **B**asically **A**vailable, **S**oft **S**tate, **E**ventually **C**onsistent und beschreibt eine Alternative zu den strengen **ACID**-Kriterien. Bei den Systemen, die nach dem **BASE**-Prinzip gestaltet sind, wird bewusst in Kauf genommen, dass die Daten nach Schreiboperationen eine absehbare Zeit inkonsistent sein können.

Bei solchen Systemen ist viel mehr die permanente Verfügbarkeit des Systems für die Clients wichtig, als die Möglichkeit, die gleichen Daten zu dem sofortigen Zeitpunkt zu sehen. Nach Ablauf einer gewissen Zeit muss das System den inkonsistenten Zustand der Daten wieder in einen konsistenten Zustand bringen, so dass alle Clients die gleichen Daten sehen können [3].

- **Basically Available** bedeutet, dass ein System immer verfügbar ist und eine Antwort immer zurückkommt. Die gelieferten Daten müssen nicht konsistent sein. Es können inzwischen Updates geben, die zwar den Datenstand geändert haben, allerdings mit dem Knoten, der geantwortet hat, noch nicht synchronisiert wurden.
- **Soft State** kennzeichnet, dass der Datenzustand nicht unbedingt alle Updates abbildet, die bisher eingegangen sind. Es können Updates geben, die noch nicht überall angewendet wurden.
- **Eventually Consistent** besagt, dass sich das System in einem konsistenten Zustand befindet, wenn alle eingegangenen Update-Operationen auf den Daten überall angewendet wurden und keine neuen Daten zwischenzeitlich eingegangen sind.[4]

#### 2.1.2 Wartbarkeit

Um die Wartungs- und Erweiterungsfähigkeiten von der Software zu gewährleisten, sollen die grundlegenden Designprinzipien eingehalten werden.

### 2.1.2.1 Die SOLID-Prinzipien

Das von Martin Fowler geprägtes Akronym **SOLID**[5] steht für fünf Prinzipien des objektorientierten Designs. Bei korrekter Anwendung dieser Prinzipien erfolgt eine höhere Wartbarkeit am Softwareprodukt. Die Software, die mit der Einhaltung von **SOLID** Regeln entwickelt wird, besteht aus vielen kleinen Modulen. Jedes Modul übernimmt bei der Entwicklung eine klare Funktion und die Interaktion zwischen diesen Modulen erfolgt über die explizit definierten 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 [6] nach, kann es für die Änderung einer Klasse nur einen Grund geben. Im Kontext der Prototypanwendung könnte der Backend-Teil beispielsweise zwei Funktionen haben, die Bearbeitung von Frontend-Anfragen und Verwaltung der Daten in der Datenbank. Demnach wäre das Design schlecht, wenn eine Klasse beide diese Funktionalitäten implementieren würde. In diesem Fall gäbe es mehrere Gründe für die Änderung dieser Klasse - z. B. eine 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 jedoch nicht geändert werden. Die Idee, die dahinter steht, kann folgendermaßen zusammengefasst werden. Sofern die neuen Funktionalitäten eingeführt werden, darf der bestehende Code nur minimal geändert werden. Beispielsweise wurden die Daten der Foto-Verwaltungs-Anwendung bisher in der relationalen Datenbank abgelegt und zusätzlich soll die Datenhaltung in der NoSQL-Datenbank implementiert werden. Die NoSQL-Datenhaltung soll in eigenem Modul implementiert werden, dadurch darf die Codebasis für die Interaktion mit relationaler Datenbank nicht geändert werden.
- **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 sollen, 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 eine direkte Abhängigkeit zu den anderen Modulen vermeiden, die Abhängigkeiten werden zu den Interfaces definiert. Demzufolge können die Module die Interfaces implementieren und ohne großen Aufwand ausgetauscht werden.

Für die Module sind nur die Interfaces sichtbar, die zu implementieren sind. Es können keine Annahmen getroffen werden, welche Teile der Anwendung diese Module verwenden werden. Sie sollen überall einsetzbar sein, wo sie ihre Funktion, die in dem Interface definiert ist, erfüllen können. Als Beispiel wird erneut Backend der Foto-Verwaltungs-Anwendung genommen und die Interaktion zwischen dem *Service*-Modul betrachtet, das mit Frontend interagiert und dem Modul, das Daten in der relationalen Datenbank verwaltet. Anstatt eine Referenz zu diesem spezifischen Datenbankmodul zu deklarieren, wird eine Referenz zu einem Interface deklariert, das die Datenhaltungskomponente beschreibt. Aus der Sicht des *Service*-Moduls gibt es eine Menge von notwendigen Operationen zum Speichern oder zum Abrufen der Daten. Diese Operationen sind unabhängig von der Art der Daten ähnlich aufgebaut und deswegen werden sie in einem *Interface* zusammengefasst. Das *Service*-Modul deklariert eine Abhängigkeit zu diesem *Interface*. Dieses *Interface* kann von verschiedenen Dateihaltungskomponenten implementiert werden.

### 2.1.2.2 Dependency Injection (DI)

*Dependency Injection* [7] ist der nächste Schritt nach *Dependency Inversion*. Das *Dependency Injection Pattern* basiert auf dem *Inversion of Control Konzept* [8]. Das bedeutet, dass sich die verwendeten Klassen nicht mehr selbst um Ablauf und Abhängigkeiten kümmern, sondern diese werden an eine externe Komponente ausgelagert. Die Klasse gibt vermeintlich 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 zur Laufzeit eingefügt (injected) wird.

Die genauere Zusammensetzung einer Anwendung wird deklarativ definiert, sodass verschiedene Konfigurationen existieren können.

Die Einhaltung des *Dependency Inversion* Prinzips zusammen mit der Anwendung des *Dependency Injection (DI)* Patterns ermöglicht den Entwicklern, den Arbeitsaufwand für die Entwicklung großer Anwendungen stark zu reduzieren. Es 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.

### 2.1.2.3 Dependency Injection und Mock-Objekte

Enge Kopplung von Abhängigkeiten erschwert nicht nur die Erweiterung oder Änderung von Codebasis, sondern auch die Testbarkeit. Beispielsweise sollten die Komponenten unabhängig von den anderen Komponenten mit *UnitTests* getestet werden. Dafür werden die Abhängigkeiten zu anderen Komponenten simuliert. Falls die zu testende Methode die Daten aus der Datenbank für die 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, sodass seitens des Entwicklers keine weiteren Schritte notwendig sind.

### 2.1.2.4 MVC-Pattern

**MVC**[9] ist ein Prinzip der modernen Programmierung und ist nach wie vor das wichtigste und verbreitetste Muster für die Architektur von GUI-Anwendungen. Solche Architektur kommt fast bei allen GUI-Frameworks zum Einsatz.

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

**2.1.2.4.1 Workflow** Der Workflow-Prozess (**Abb. 2.2**) stellt eine vollständige Beschreibung aller Aktivitäten, der den Einsatz des **MVC**-Patterns voraussetzt. Die Abbildung 2.2 stellt einen groben Workflow des **MVC**-Patterns anhand einer Beispielinteraktion und ihres Ergebnisses dar.

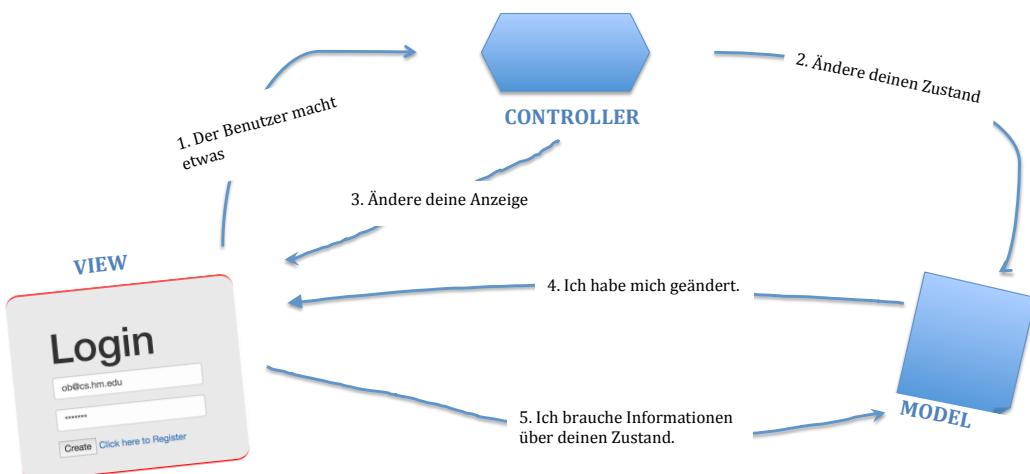


Abb. 2.2: Workflow zum MVC-Konzept

#### Beschreibung des Workflow-Prozesses:

##### 1. Der Benutzer interagiert mit der View

Der Benutzer führt eine Aktion an der View aus. Dadurch teilt die 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 einer Aktion an der View durch den Benutzer, nimmt der Controller diese Aktion an und interpretiert sie. Bei der Interpretation stellt der Controller fest, was gemacht werden muss und wie das Model aufgrund dieser Aktion beeinflusst werden kann.

### 3. Der Controller kann auch die View auffordern, ihren Zustand zu ändern

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

### 4. Das Model informiert die 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 diejenigen, die sich beim Model registriert haben, über seine Zustandsänderungen informieren.

### 5. Die View erfragt den Zustand des Models

Das Model stellt weitere Methoden zur Verfügung, über die der aktuelle Zustand des Models erfragt werden kann. Jede 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 realisieren, nutzt MVC das sogenannte Beobachter Muster.

**2.1.2.4.2 Beobachter Muster** Beobachter Muster (engl. Observer-Pattern) ist eines der am meisten genutzten und bekanntesten Patterns. 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.

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 sich die Observers beim Observable registrieren und bei einer Zustandsänderung alle registrierte Objekte informiert.

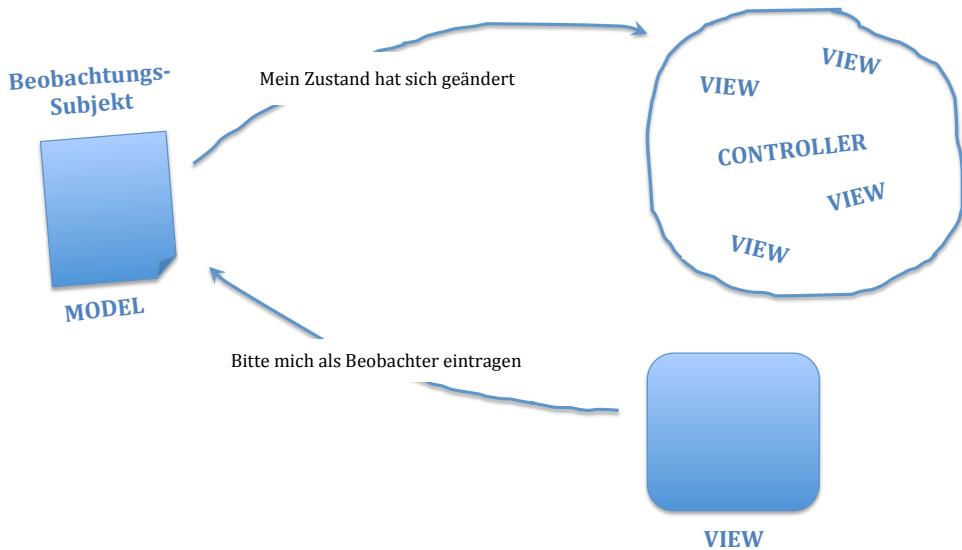


Abb. 2.3: Observer Pattern

### Beschreibung des Observer-Pattern Prinzips:

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

## 2.2 Relevante Technologien

Dieser Abschnitt widmet sich den aktuellen Technologien, die für den Prototyp relevant sind.

### 2.2.1 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.

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. [10] Dokumentdatenbanken gruppieren die Daten in einem strukturierten Dokument, typischerweise in einer *JSON*-Datenstruktur [11]. **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. ([12], 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 eigenen Ziele und ist kategorisiert. Die möglichen Kategorien werden demnächst näher erläutert.

- 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 [13]. 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.

Möchte man ein bestimmtes Dokument erweitern, so kann man es einfach tun, da eine dokumentenorientierte Datenbank strukturfrei ist. Weitere Datenformate sind beispielsweise YAML [14] (angelehnt an XML) oder XML [15] selbst.

### 2.2.1.1 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.

**MongoDB** präsentiert sich als eine quelloffene, dokumentenorientierte NoSQL-Datenbank mit den folgenden Konzepten wie *Ausfallsicherheit* und *horizontale Skalierung* (**Kap. 2.1.1.4**).

**2.2.1.1.1 Datensätze in Form von Dokumenten** Die Daten speichert **MongoDB** in Form von Dokumenten im **BJSON**-Format [16]. Die Dokumente selbst werden in sogenannten Kollektionen (*Collections*) gespeichert, die grob mit den Tabellen einer relationalen Datenbank vergleichbar sind. Ein Zugriff auf Daten mehrerer Kollektionen (*Collections*), wie es aus dem *Joins*-Konzept relationaler Datenbank bekannt ist, ist nicht möglich. Die **CRUD**-Operationen sind auf Ebene der *Collection* durchzuführen. Jedes Dokument ist an keine vordefinierte Struktur gebunden und kann eine beliebige Anzahl an Feldern besitzen. Die Dokumente aus einer *Collection* sind komplett unabhängig voneinander.

Die Speicherung von Daten in Form von Dokumenten bietet den Vorteil, dass 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 [17].

**2.2.1.1.2 CRUD = IFUR-Operationen** Die **CRUD**-Operationen aus SQL heißen in **MongoDB** **Insert**, **Find**, **Update** und **Remove**. Bei **MongoDB** wird eine Datenbank bzw. eine Collection erst zur Laufzeit und beim ersten Einfügen des ersten Dokuments von **MongoDB** selbst erzeugt. Die gesonderte Erstellung einer Datenbank oder einer Collection kann somit erspart werden.

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

Listing 2.1: 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.

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

Listing 2.2: Dokument(e) finden

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

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

Listing 2.3: Dokument(e) aktualisieren

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

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

Listing 2.4: Dokument(e) löschen

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

**2.2.1.3 Indizes** Um die Laufzeit von Datenbankabfragen zu optimieren bzw. zu beschleunigen, können Indexe verwendet werden. Indexe in MongoDB werden als *B-Tree*-Datenstrukturen [18] 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 2.5: Index auf ein Feld anlegen

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

**2.2.1.1.4 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.[19]

**2.2.1.1.5 Aggregation Framework** Analog zu *GROUP BY* in SQL hat **MongoDB** sein eigenes Konzept entwickelt, das im eigenen Aggregation Framework modelliert ist. Die einzelnen Aggregationsoperationen mit entsprechender Beschreibung sind aus der Tabelle 2.1 zu entnehmen.

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. 2.1: Aggregationsoperationen

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

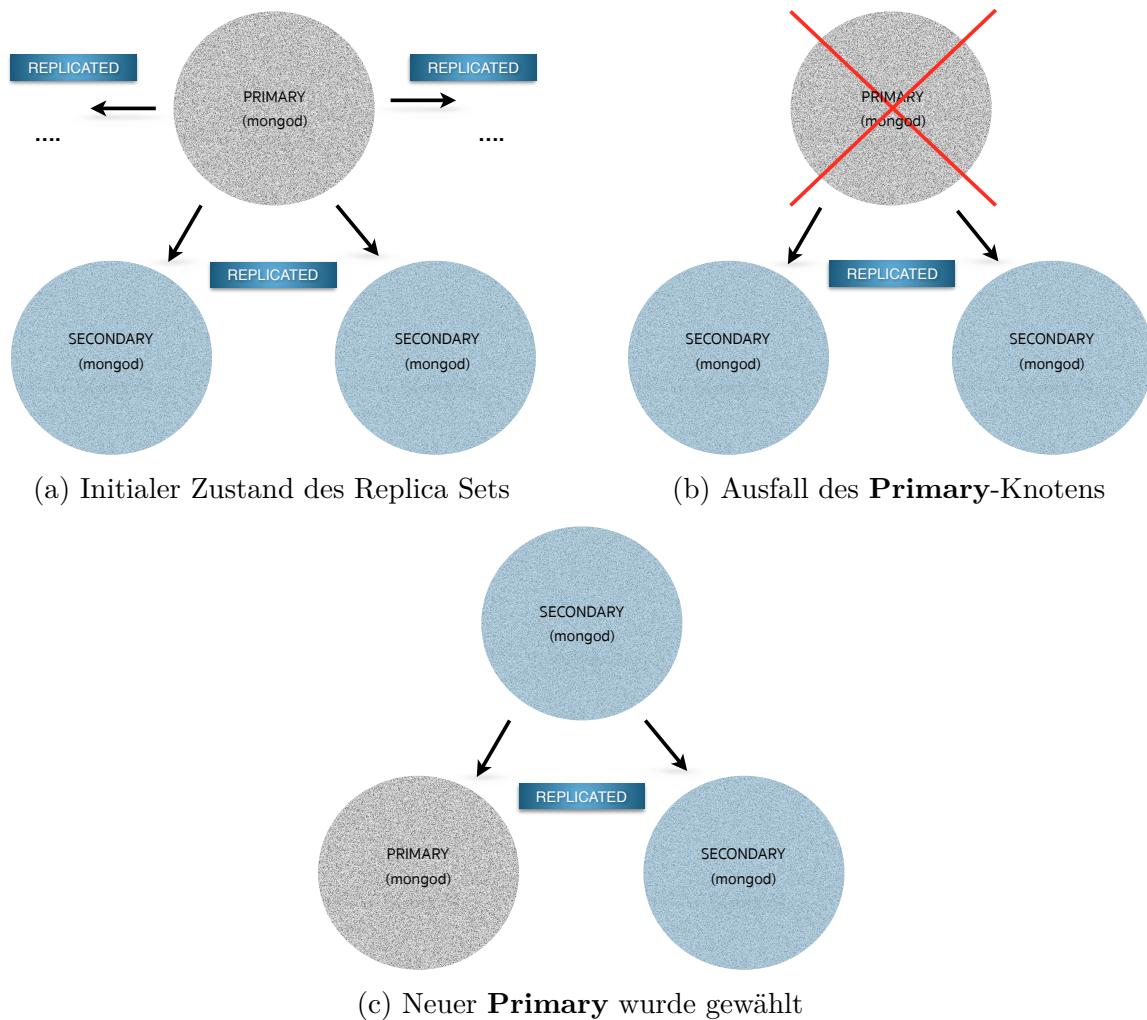
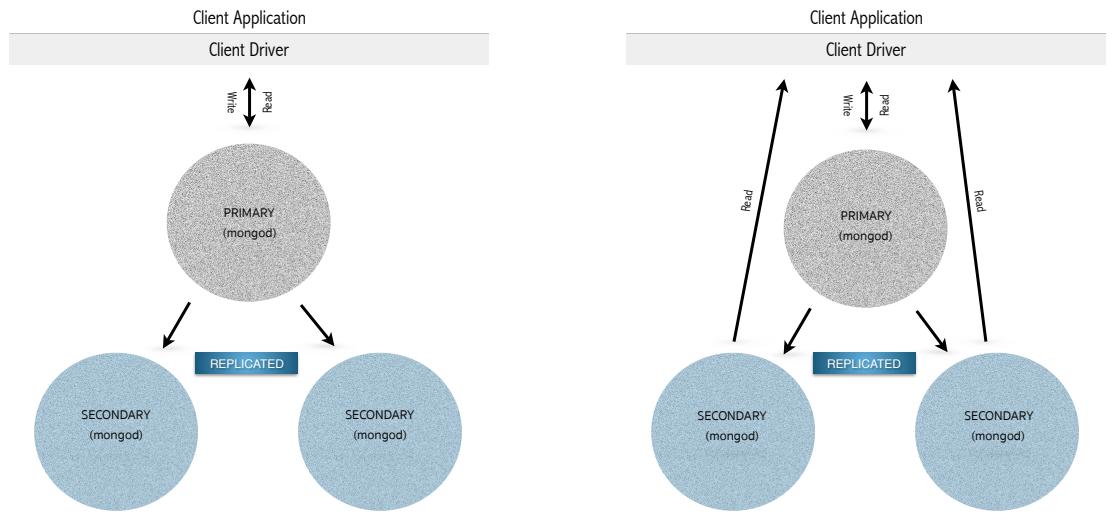


Abb. 2.4: Szenario für eine Replikationsgruppe mit drei Servern in einer *Shard*

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 2.4 veranschaulicht ein Szenario für eine Replikationsgruppe mit drei Knoten. Jeder Knoten aus der Gruppe ist als einen eigenen Server vorzustellen.



(a) Lesezugriffe nur über **Primary** möglich      (b) Lesezugriffe auch über **Secondaries** freigeschaltet

Abb. 2.5: 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 auch bei Secondaries entsprechend freigeschaltet werden. Bei der Freischaltung der Leserrechte durch Secondaries muss jedoch in Kauf genommen werden, dass das Lesen durch Secondaries den konsistenten Zustand an Daten jedoch nicht garantiert. Der Grund dafür ist, dass die Schreiboperationen nur über den Master erfolgen und die Replikation der Daten etwas Zeit nimmt.

**2.2.1.7 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 horizontale Verteilung der Daten erfolgt bei **MongoDB** auf Ebene der *Collections* nach *Sharding-Keys*. Die *Sharding-Keys* dienen dazu, um später Zugriffe auf einzelne Dokumente zu ermöglichen, die auf verschiedenen Servern abgelegt sind.

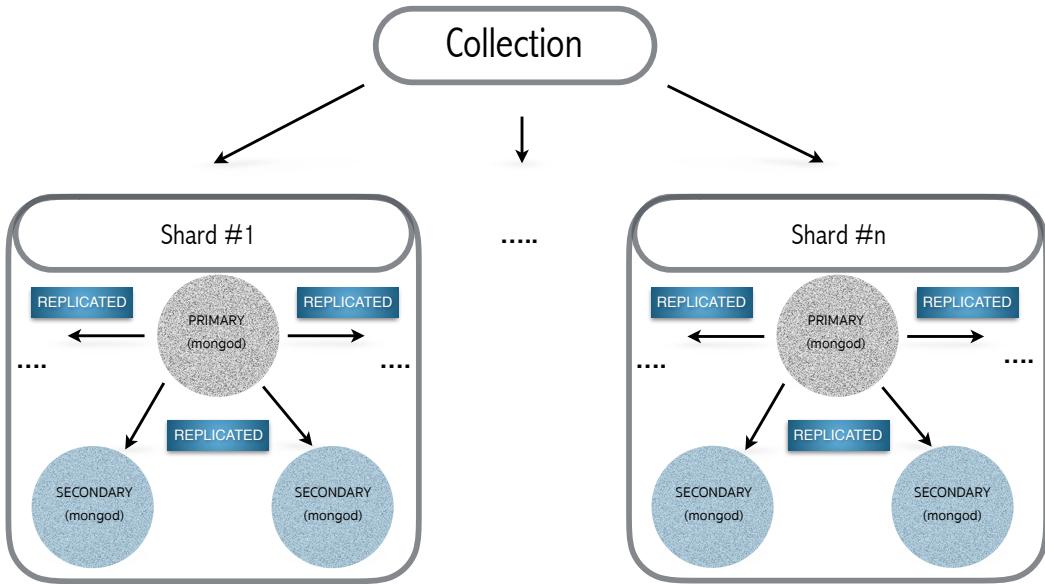


Abb. 2.6: Ein Beispiel für Verteilung einer *Collection* auf mehreren *Shards*

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.

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.
- *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 handelt es um eine *mongod*-Instanzen.

Die Abbildung 2.7 veranschaulicht die Interaktion von den gerade genannten Komponenten innerhalb eines Sharded-Cluster:

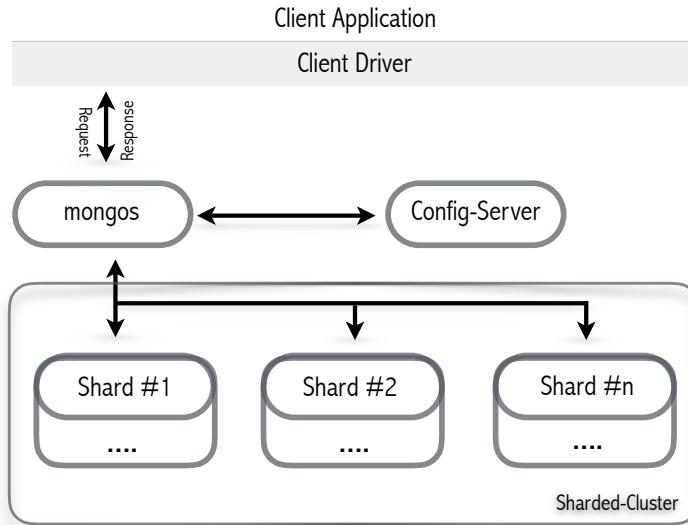


Abb. 2.7: Horizontale Skalierung (*Sharding*)

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

**2.2.1.1.8 Fragmentierung nach *Shard-Keys*** Die Fragmentierung der Daten 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*.

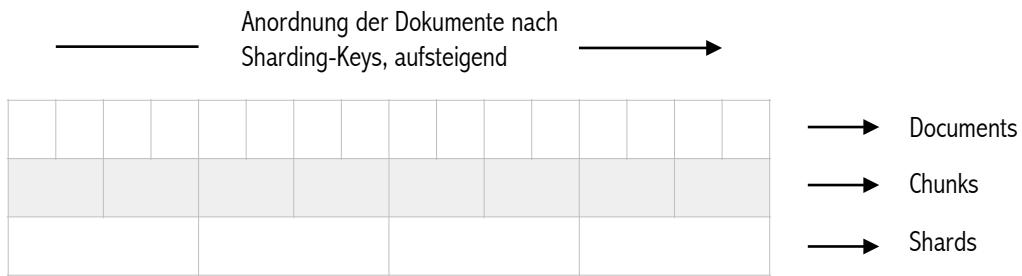


Abb. 2.8: Anordnung der Dokumente in Blöcke (=Chunks) unter Verwendung des *Shard-Keys*. Mehrere Blöcke bilden dementsprechend eine *Shard*.

Bei der *Shard-Keys* Konfiguration müssen folgende *Constraints* [20] 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

**2.2.1.1.9 GridFS** Für die Daten, die eine Größe in Höhe von 16MB überschreiten, stellt **MongoDB** ein Dateisystem namens **GridFS** zur Verfügung. Abgelegt werden solche Daten in zwei spezielle *Collections*:

- *fs.files* - die *fs.files*-Collection enthält die Metainformationen zu den einzelnen Dokumenten. Listing 2.1 veranschaulicht ein Beispiel dazu.
- *fs.chunks* - in *fs.chunks*-Collection werden die eigentlichen Daten gespeichert.

Listing 2.1: Beispiel für ein Dokument in *fs.files-Collection*

```
{  
    "_id" : ObjectId("58f5f0056ce5d24528e152c3"),  
    "filename" : "Bicycle.jpeg",  
    "aliases" : null,  
    "chunkSize" : NumberLong(261120),  
    "uploadDate" : ISODate("2017-04-18T10:52:53.974Z"),  
    "length" : NumberLong(68090),  
    "contentType" : "image/jpeg",  
    "md5" : "85a866edfc999c0163e7bbc7cd5269b"  
}
```

**2.2.1.1.10 Treiber für MongoDB** Alle mögliche Operationen, die **MongoDB** zur Verfügung stellt, sind 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 [21] möglich.

### 2.2.1.2 Apache Cassandra

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

**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.

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

#### 2.2.1.2.1 Architektur

- **Apache Cassandra** ist nach peer-to-peer<sup>3</sup> verteiltes System aufgebaut. Ein peer-

<sup>3</sup>Peer-to-Peer-Netze (P2P) sind Netze, bei denen alle Knoten im Netz dezentral sind.

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.

**2.2.1.2.2 Datenmodell** Die Hauptbestandteile des Datenmodells von **Apache Cassandra** veranschaulicht die Abbildung 2.9.

- Cluster
- Keyspace
- Column Family
- Row
- Column sowie
- Value

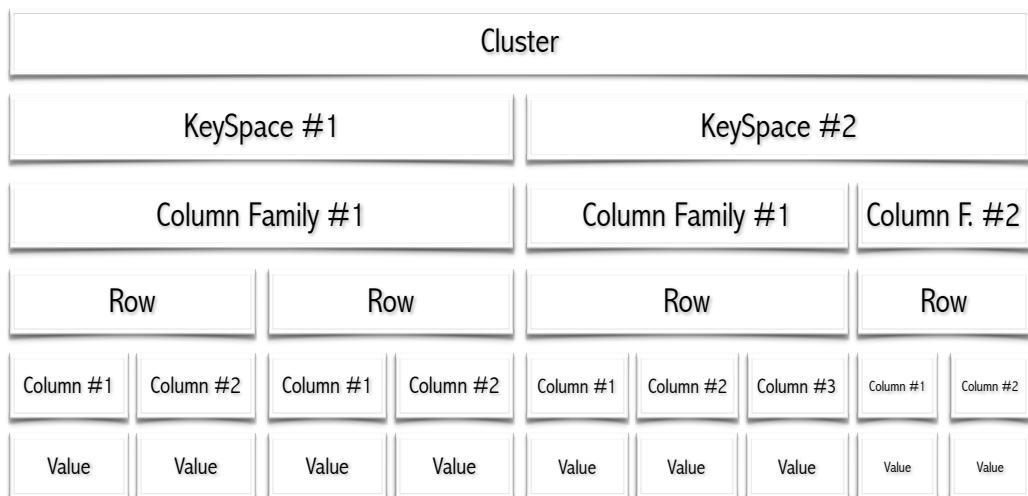


Abb. 2.9: Datenmodell

**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.

### 2.2.2 Spring Framework

Das Spring Framework ist ein Open Source Java Framework, welches einfache Java Objekte, sogenannte Plain-Old-Java-Objects (POJO), als **Spring-Beans** verwaltet. Dabei stellt das Spring Framework unter Anderem einen *Inversion of Control Container* zu Verfügung, der per *Dependency Injection* abhängige Spring-Beans miteinander verknüpft und konfiguriert.

Die wesentlichen Funktionen des Spring Frameworks sind:

- Plain-Old-Java-Objects basierendes Programmiermodell.
- Verknüpfungen durch DI über den *Inversion-of-Control Container*.

#### 2.2.2.1 Module des Spring Frameworks

Spring Framework besteht aus Modulen, die den Entwicklern unabhängig zur Verfügung stehen. Die folgenden drei Module werden für die Umsetzung der Architektur vorgeschlagen.

- Spring Core Container - Das Basismodul, das den *Inversion of Control Container* [22] und die Funktionen für die *Dependency Injection* [23] bereitstellt. Die Anwendungsobjekte existieren in einer Spring-basierten Anwendung in einem Spring Core Container (**Abb. 2.10**). Der Spring Core Container erstellt Objekte, verschaltet und konfiguriert sie. Weiterhin übernimmt der Spring Core Container die Kontrolle

über den Lebenszyklus der Objekte. In der Spring Terminologie heißen die von einem Spring-Container verwalteten Objekte **Spring-Beans** oder einfach **Beans**.

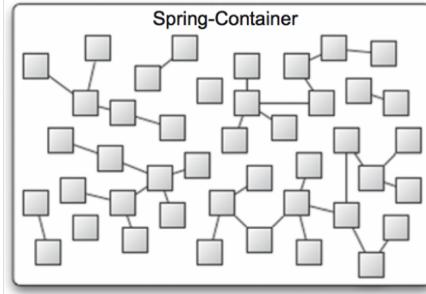


Abb. 2.10: Spring Core Container für Beans

Seit Erscheinen der Springs-Version 1.0 im Jahr 2004 müsste man den Container mit XML konfigurieren. Ausgehend von Springs-Version 2.5 im Jahr 2007 wurde der Aufwand der XML-Konfiguration in einer Spring-basierten Anwendung durch die Kombination mit @-Annotationen extrem reduziert. Mit der Springs-Version 3.0 ist es endlich möglich geworden, den Spring-Container vollständig ohne XML-Konfiguration zu erzeugen. Diese Konfiguration ohne XML-Konfiguration nennt man **Java-basierte Konfiguration**.

- Spring Web - Das MVC-Framework mit der Möglichkeit REST-Webanwendungen umsetzen zu können. Dazu wird das MVC-Pattern implementiert. Zusätzlich werden allgemeine Web-Technologien wie HTTP zu Verfügung gestellt.
- Spring Boot - Mit diesem Module können Self-Contained Anwendungen erstellt werden. Diese Art von Anwendungen werden in einem einzelnen JAR/WAR ausgeliefert, abhängig von Konfiguration. Zusätzlich werden die Abhängigkeiten aller benötigten Bibliotheken durch Spring Boot verwaltet. [24]

### 2.2.2.2 Konfiguration mit Maven

Um Spring Framework in einer Java Anwendung verwenden zu können, müssen die entsprechenden Bibliotheken dem Java Projekt hinzugefügt werden. Listing 2.2 zeigt ein Beispiel für eine Maven-Konfigurationsdatei [25], in der benötigte Bibliotheken für Spring deklariert sind.

Listing 2.2: Konfiguration mit pom.xml

```
<!-- Spring Core Container Modul -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
</dependency>

<!-- Spring Web Modul -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
</dependency>

<!-- Spring Boot Starter Modul -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Maven wird verwendet, um die Abhängigkeiten zu den externen Bibliotheken zu verwalten. Die benötigten Bibliotheken werden von User in der Maven- Konfigurationsdatei `pom.xml` als Abhängigkeiten (engl. `dependencies`) deklariert und das Tool lädt die Bibliotheken automatisch herunter, speichert die in einem lokalen Repository und fügt sie dann dem Projekt hinzu.

### 2.2.3 *REpresentational State Transfer* (REST)

**REST** ist ein Designkonzept für die Web Services. Die Daten werden in der Form von Ressourceneinheiten ausgetauscht. Jede Ressource ist eindeutig und mit einer *URI* identifizierbar.

Der Zugriff auf Ressourcen erfolgt auf Basis des *HTTP*<sup>4</sup>-Protokolls. Um Ressourcen lesen, aktualisieren, löschen oder anlegen zu können, werden die gängigen *HTTP*-Methoden

---

<sup>4</sup>*HTTP* ist ein zustandloses Protokoll zur Übertragung der Daten zwischen Web-Clients (=Browser Anwendungen) und Web-Servers. Jede neue Anfrage, die von einem Web-Client erfolgt, erfordert einen neuen Verbindungsaufbau und erneute Datenbeschaffung.

gebraucht. Diese sind:

- *GET* - ist ein lesender Zugriff auf Daten.
- *PUT* - aktualisiert bestehende Daten.
- *DELETE* - löscht vorhandene Daten.
- *POST* - legt neue Daten an.

Die Rückmeldung des Web-Servers erfolgt im *JSON*-Format in Kombination mit einem Code, der zusätzlich den Status der Rückmeldung mitteilt. **REST** ist eine beliebte Technologie, wenn es um den reinen Datenaustausch zwischen Web-Clients und Web-Servers geht.

#### 2.2.4 AngularJS 2 - JavaScript Framework

Angular 2 ist ein JavaScript basiertes Frontend Framework, das die Entwicklung von so genannten *Single-Page*-Anwendungen ermöglicht. In einer *Single-Page*-Anwendung wird die *HTML*-Seite mit dem ganzen Inhalt nur einmal geladen und die Teile davon werden dynamisch nachgeladen oder updated, z. B. als Ergebnis einer Nutzerinteraktion. Weder das Neuladen der Seite noch die Weiterleitung zu den anderen Seiten ist notwendig. Die ganzen Informationen werden auf einer Seite dargestellt.

Bei der Entwicklung mit dem AngularJS Framework werden die *HTML*-Seiten mit speziellen Tag Attributten erweitert. Diese Tag Attribute sind mit JavaScript Variablen assoziiert. Falls diese Variablen geändert werden, werden die entsprechenden Teile der Seite updated. Die entsprechenden JavaScript Variablen können z. B. infolge einer WebService-Anfrage geändert werden.

Angular 2 ist komplett in TypeScript entwickelt worden. TypeScript ist eine JavaScript-Erweiterung von Microsoft, die JavaScript um statisches Typsystem und objektorientierten Programmierstil über die Nutzung von Interfaces, Klassen, Module etc. erweitert. [26]

# 3 Architektur

## 3.1 Konzept

Die vorgeschlagene Architektur ist auf der Abbildung 3.1 zu sehen.

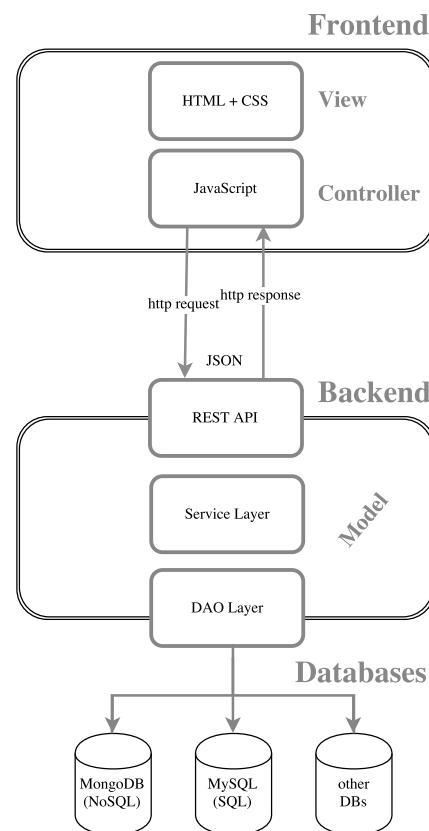


Abb. 3.1: Architektur-Prototyp

Die Auswahl dieser Architektur hat zum Ziel, ein möglichst gutes Skalierungsverhalten der Anwendung zu erreichen. Die für die Performance zwei kritischen Komponenten - Backend-Server und die Datenbank - werden auf mehreren Rechnern verteilt. Der entscheidende Faktor für das Skalierungsverhalten beider Komponenten ist, wann und wie viel Synchronisation notwendig ist.

In dieser Architektur wird das Ziel verfolgt, bei steigender Anzahl der Internetnutzer, sowohl für die Anfragen, die Leseoperationen in der Datenbank benötigen, als auch für die Anfragen, bei denen die Daten gespeichert werden müssen, die Antwortzeit konstant bleibt.

### 3.1.1 Backend

Der Backend stellt seine Funktionalität als die Menge von Webservices mit *REST*-Interface zu Verfügung.

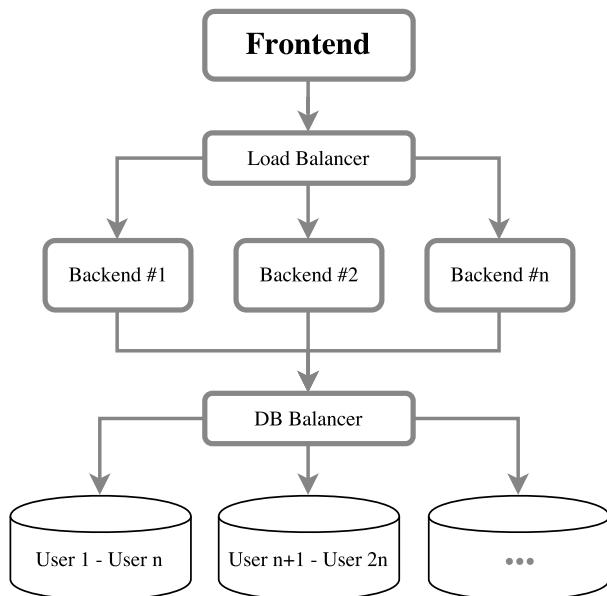


Abb. 3.2: Überblick zur vorgeschlagenen Architektur

Um ein gutes Skalierungsverhalten zu erreichen, wird jede Anfrage unabhängig von den anderen Anfragen bearbeitet. In dem Backend werden keine Informationen zwischenge-

speichert, alles was gespeichert werden muss, wird in die Datenbank geschrieben. Solche Komponente nennt man *stateless*, weil sie keinen Zustand speichert. Somit ist keine Synchronisation zwischen den Maschinen des Backends notwendig, sogar die Anfragen innerhalb einer User-Session können von verschiedenen Maschinen bearbeitet werden. Ein *Load Balancer* kann die Anfragen auf verschiedenen Maschinen in Backend verteilen ohne Rücksicht darauf nehmen zu müssen, welcher Maschine die jeweilige Anfrage entstammt.

Die Webservices sind unabhängig voneinander und können auch auf verschiedenen Maschinen ausgeführt werden. Diese Architektur ist auch als *Microservice* Architektur bekannt.

### 3.1.2 Datenbank

Das Skalierungskonzept für die Datenbank basiert auf der Annahme, dass die Web-Anwendung auf *Multi-User* Betrieb ausgelegt ist und jeder User seine Daten unabhängig von den anderen Usern verwaltet. Daher ist es möglich, die Daten so aufzuteilen, dass sich alle Daten eines bestimmten Users nur in einem Teil der Daten wiederfinden. Dann ist es möglich, jeden Teil der Daten getrennt von allen anderen Teilen, z. B. auf eigener Maschine, zu speichern. Wenn der Backend-Server eine Anfrage bearbeitet, sind die Daten nur eines Nutzers betroffen. Damit fragt der Backend-Teil nur die Komponente ab, die entsprechenden Teil der Daten verwaltet. Es ist notwendig, diese Komponente zu identifizieren. Jedoch ist es eine billige *Mapping*-Abfrage, z. B. von dem UserID zu der IP-Adresse der Maschine, die die Daten verwaltet. Während des *Live*-Betriebs ist auch keine Synchronisation erforderlich. Die Synchronisation ist nur dann relevant, wenn die Daten neu aufgeteilt werden müssen.

## 3.2 Umsetzung

# **Fazit**

Die wesentliche Erkenntnis der folgenden Arbeit ist das Verständnis, dass die Entwicklung von skalierbaren, wartungs- und erweiterungsfähigen Webanwendungen Vieles voraussetzt. Unter Anderem sind es die Konzepte, Kenntnisse mehrerer Programmiersprachen, Frameworks blabla

# Literaturverzeichnis

- [1] S. Springer. *Node.js: Das umfassende Handbuch. Serverseitige Webapplikationen mit JavaScript entwickeln.* Rheinwerk Computing, 2016, S. 560.
- [2] S. Edlich. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* 2., aktualisierte und erw. Aufl. München: Hanser, 2011.
- [12] A. Hollosi. *Von Geodaten bis NoSQL: leistungsstarke PHP-Anwendungen: Aktuelle Techniken und Methoden für Fortgeschrittene.* München: Hanser, 2012.

# Onlinequellen

- [3] *BASE*. URL: <https://blog.codecentric.de/2011/08/grundlagen-cloud-computing-cap-theorem/> (besucht am 05.02.2017).
- [4] *ACID vs. BASE*. URL: <http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/> (besucht am 14.02.2017).
- [5] *SOLID*. URL: [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) (besucht am 03.02.2017).
- [6] *Martin Fowler: Inversion of Control*. URL: <http://martinfowler.com/bliki/InversionOfControl.html> (besucht am 12.01.2017).
- [7] *DI Einfuehrung*. URL: <http://www.itwissen.info/definition/lexikon/Dependency-Injection-dependency-injection-DI.html> (besucht am 15.01.2017).
- [8] *Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern*. URL: <http://martinfowler.com/articles/injection.html> (besucht am 14.01.2017).
- [9] *Objektorientierte Programmierung*. URL: [http://openbook.rheinwerk-verlag.de/oop/oop\\_kapitel\\_08\\_002.htm](http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_08_002.htm) (besucht am 20.01.2017).
- [10] *MySQL vs. MongoDB*. URL: <http://www.computerwoche.de/a/datenbanksysteme-fuer-web-anwendungen-im-vergleich,2496589> (besucht am 03.01.2017).
- [11] *JSON*. URL: <http://www.json.org>.
- [13] *NoSQL: Key-Value-Datenbank Redis im Überblick*. URL: <https://www.heise.de/developer/artikel/NoSQL-Key-Value-Datenbank-Redis-im-Ueberblick-1233843.html> (besucht am 17.01.2017).
- [14] *YAML*. URL: <http://www.yaml.org/start.html>.

- [15] *XML*. URL: <https://www.xml.com/>.
- [16] *JSON*. URL: <http://www.bjson.org>.
- [17] *MongoDB*. URL: <http://www.moretechnology.de/mongodb-eine-dokumentenorientierte-datenbank/> (besucht am 21.01.2017).
- [18] *B - Trees*. URL: [http://btechsmartclass.com/DS/U5\\_T3.html](http://btechsmartclass.com/DS/U5_T3.html) (besucht am 21.01.2017).
- [19] *Aggregation*. URL: <https://docs.mongodb.com/manual/aggregation/> (besucht am 27.02.2017).
- [20] *Introduction to Sharding*. URL: <https://docs.mongodb.com/manual/sharding/> (besucht am 02.02.2017).
- [21] *MongoDB Drivers*. URL: <https://docs.mongodb.com/ecosystem/drivers/> (besucht am 18.02.2017).
- [22] *DI in Spring*. URL: <https://entwickler.de/online/web/mit-dependency-injection-klassenabhaengigkeiten-kontrollieren-134784.html> (besucht am 15.01.2017).
- [23] *DI in Spring*. URL: <https://springframework.guru/dependency-injection-example-using-spring/> (besucht am 21.01.2017).
- [24] *Das Spring Framework*. URL: <https://www.frank-rahn.de/einfuehrung-spring-framework/%5C#toggle-id-1> (besucht am 18.02.2017).
- [25] *Apache Maven*. URL: <https://maven.apache.org> (besucht am 21.02.2017).
- [26] *TypeScript Documentation*. URL: <http://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-2.html> (besucht am 03.03.2017).

# Abbildungen

2.1	CAP-Theorem	6
2.2	Workflow zum MVC-Konzept	12
2.3	Observer Pattern	14
2.4	Szenario für eine Replikationsgruppe mit drei Servern in einer <i>Shard</i>	21
2.5	Freischaltung der Lesezugriffe	22
2.6	Ein Beispiel für Verteilung einer <i>Collection</i> auf mehreren <i>Shards</i>	23
2.7	Horizontale Skalierung ( <i>Sharding</i> )	24
2.8	Anordnung der Dokumente in Blocks (= <i>Chunks</i> ) unter Verwendung des <i>Shard-Keys</i> . Mehrere Blocks bilden dementsprechend eine <i>Shard</i> .	25
2.9	Datenmodell	27
2.10	Spring Core Container für Beans	29
3.1	Architektur-Prototyp	32
3.2	Überblick zur vorgeschlagenen Architektur	33

# Tabellen

2.1 Aggregationsoperationen . . . . .	20
---------------------------------------	----

# Quelltextverzeichnis

2.1	Dokument(e) speichern . . . . .	17
2.2	Dokument(e) finden . . . . .	18
2.3	Dokument(e) aktualisieren . . . . .	18
2.4	Dokument(e) löschen . . . . .	18
2.5	Index auf ein Feld anlegen . . . . .	19
2.1	Beispiel für ein Dokument in <i>fs.files-Collection</i> . . . . .	26
2.2	Konfiguration mit <code>pom.xml</code> . . . . .	30