Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander Ponomarenko

Entwurf und Realisierung einer verteilten NoSQL-Anwendung

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Ponomarenko

Entwurf und Realisierung einer verteilten NoSQL-Anwendung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachter : Prof. Dr. sc. pol. Wolfgang Gerken

Abgegeben am 22. August 2011

Alexander Ponomarenko

Thema der Bachelorarbeit

Entwurf und Realisierung einer verteilten NoSQL-Anwendung

Stichworte

NoSQL, BASE, Eventually Consistent, CAP-Theorem, Map/Reduce, CouchDB

Kurzzusammenfassung

In dieser Arbeit wurde untersucht, ob sich eine typische NoSQL-Datenbank für die Realisierung eines typischen Anwendungsfalls eignet. Es wurde ein Prototyp einer Ausschreibungs- und Bewerbungsverwaltung sowie eine Stellenbörse entwickelt und anschließend bewertet. Durch Replikationsmechanismen wird die Ausschreibungs- und Bewerbungsverwaltung auf die Endgeräte der Benutzer synchronisiert, sodass diese offline arbeiten können. Die Stellenbörse bietet Bewerbern die Möglichkeit, sich auf eine Ausschreibung online zu bewerben. Im Anschluss an die Realisierung wurde eine Bewertung des Prototypen vorgenommen.

Alexander Ponomarenko

Title of the paper

Design and implementation of a distributed NoSQL application

Keywords

NoSQL, BASE, Eventually Consistent, CAP-Theorem, Map/Reduce, CouchDB

Abstract

In this paper it was investigated whether a typical NoSQL database ist suitable for a typical use case. A prototype of an application for the administration for jobs and applications has been developed and evaluated. Through replication mechanisms of the application, users can work offline as well. It also has been developed a prototype of an application for vacancies where applicants may apply online. Finally, an evaluation of the prototype was made.

Inhaltsverzeichnis

1. Einführung

1.1. Motivation

Die Grundlagen relationaler Datenbanken stammen aus den 1970er Jahren. Jedoch waren die Anforderungen und die Hardware-Voraussetzungen an ein Datenbankmanagementsystem andere, als sie es heute sind.

Unterschiedliche Hardware-Voraussetzungen In den 1970ern beherrschten Großrechner (Mainframes) und kleine Workstations den Markt. Diese waren meist bei großen Unternehmen, staatlichen Einrichtungen und Universitäten im Einsatz. [32, S. 115f] Hingegen sind heutzutage verschiedene kleine Geräte wie z.B. PCs, Laptops, Mobiltelefone, etc. mit einer um ein Vielfaches höheren Rechenleistung ausgestattet als die der damaligen Großrechner. [23, S. 21] Durch Kopplung mehrerer Computer (Cluster-Computing, Grid-Computing) kann man sowohl die Rechengeschwindigkeit erhöhen, als auch die Verfügbarkeit und die Ausfallsicherheit steigern. [21, S. 24ff] Prozessoren und Hauptspeicher sind deutlich schneller und die Festplattenkapazitäten sind deutlich größer geworden. Es ist heutzutage möglich, nahezu alles zu speichern, während früher die Datenmengen möglichst gering zu halten waren. [30]

Unterschiedliche Märkte für DBMS Zur Entstehungszeit relationaler DBMS gab es praktisch nur einen Markt, den Markt der Geschäftsdatenverarbeitung. [30] Jedoch sind z.B. mit dem Web 2.0 neue Anforderungen entstanden. Sehr große Datenmengen müssen verarbeitet und durch die Agilität von Web 2.0-Projekten die Strukturen der anfallenden Daten oft verändert werden. Das ist mit relationalen Datenbanken schwer möglich, da diese 1. nicht so leicht zu skalieren sind und 2. sie zu starke Schemarestriktionen aufweisen. [16, S. 1ff]

1. Mit Standard-Hardware kann man relationale Datenbanken nicht ohne weiteres horizontal skalieren. Es lassen sich nicht einfach beliebig weitere Server hinzufügen. Relationale Datenbanken sind stattdessen eher darauf ausgelegt, dass der Server, auf dem die Datenbank läuft, mit mehr Leistung ausgestattet wird, also vertikal skaliert wird. Dies hat jedoch seine Grenzen im Vergleich zum Hinzufügen beliebig vieler weiterer Server.

2. Schemaerweiterungen in relationalen Datenbanken können sehr lange dauern und so das gesamte System für eine gewisse Zeit lahmlegen. Dies ist jedoch bei Web 2.0-Projekten nicht hinnehmbar.

Die Hersteller relationaler Datenbanken haben versucht, ihre Systeme den neuen Anforderungen anzupassen, um mit der Anforderungsentwicklung Schritt halten zu können. Jedoch haben diese Erweiterungen nicht dazu geführt, dass weiterhin praktisch ausschließlich ihre Produkte benutzt werden. So entstehen immer mehr unabhängige Datenbanken, die sich der neuen Anforderungen annehmen. [29]

Mit Ansätzen wie Map/Reduce [13] und BigTable von Google [11] sowie Dynamo von Amazon [14] entstanden Vorreiter für eine neue Bewegung, die alternative, nicht-relationale Datenbanken bereitstellt. Im Mai 2009 tauchte der Begriff *NoSQL* für diese alternativen Datenbanken auf. Diese werden in einer Reihe einschlägiger Projekte betrieben, unter anderem bei Yahoo, Amazon, Google, Facebook, MySpace, LinkedIn, etc. [16, S. 1]

Wie man sieht, gibt es ernstzunehmende Alternativen zu relationalen Datenbanken.

1.2. Zielsetzung

In dieser Arbeit wird die NoSQL-Bewegung untersucht. Dabei soll zunächst geklärt werden, was als ein NoSQL-DBMS angesehen wird und weshalb NoSQL-DBMS überhaupt entwickelt wurden. Es ist nicht Ziel dieser Arbeit, einen umfassenden Vergleich von NoSQL-Datenbanken aufzustellen. Stattdessen soll die Einsetzbarkeit einer charakteristischen NoSQL-Datenbank anhand eines typischen Anwendungsfalls untersucht werden. Mit einer dokumentenorientierten Datenbank soll überprüft werden, ob damit funktionsfähige Software in Form einer verteilten Anwendung umgesetzt werden kann. Um dies zu überprüfen, wird ein Prototyp einer Ausschreibungs- und Bewerbungsverwaltung erstellt und anschließend bewertet.

1.3. Aufbau der Arbeit

Zunächst werden in Kapitel 2 die Grundlagen dieser Arbeit geklärt. Dabei wird zunächst auf relationale Datenbanken eingegangen, um anschließend den Vergleich zu NoSQL-Datenbanken zu ziehen. Außerdem werden die Grundlagen der NoSQL-Bewegung erläutert. Zum Schluss des Kapitels werden aktuelle NoSQL-Systeme vorgestellt.

In Kapitel 3 wird ein Anwendungsszenario der zu erstellenden Software beschrieben und die funktionalen und nichtfunktionalen Anforderungen an das System definiert.

Das Kapitel 4 befasst sich mit dem Entwurf des zu erstellenden Systems. Es werden die eingesetzten Technologien sowie die Architektur des Systems und die Konfliktbehandlung beschrieben. Zum Schluss wird auf das Testen der Software eingegangen.

In Kapitel 5 wird gezeigt, wie der entwickelte Prototyp des Systems umgesetzt wurde. Dabei werden die einzelnen Komponenten beschrieben.

Kapitel 6 enthält eine kritische Bewertung des entwickelten Prototyps. Dabei wird auf die Unterschiede zwischen Analyse, Entwurf und Realisierung eingegangen, sowie eine Bewertung darüber vorgenommen, welche Teile des Systems sich für die Praxis eignen könnten und welche Teile besser anders gelöst werden sollten.

Im letzten Kapitel 7 wird die Arbeit zusammengefasst und es wird ein Ausblick auf eine mögliche Weiterentwicklung des Systems gegeben.

2. Grundlagen

In diesem Kapitel wird ein Überblick über die Grundlagen dieser Arbeit gegeben. Dabei wird zunächst auf die relationalen Datenbanken eingegangen, um anschließend einen Überblick über NoSQL-Datenbanken zu geben. Es werden die Grundlagen der NoSQL-Datenbanken beschrieben und zum Schluss aktuelle NoSQL-Systeme vorgestellt.

2.1. Relationale Datenbanken

Relationale Datenbanken beruhen auf dem von Ted Codd in seiner 1970 veröffentlichen Arbeit *A Relational Model of Data for Large Shared Data Banks* entwickelten relationalen Modell. Das Modell fand aufgrund seiner Einfachheit und mathematischen Grundlagen sofort Aufmerksamkeit. Es basiert auf dem Konzept einer mathematischen Relation als grundlegenden Baustein. Es hat seine theoretische Basis in der Mengentheorie und Prädikatenlogik. [17, S. 133]

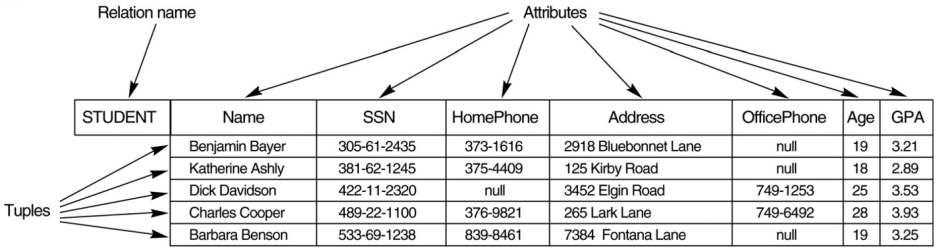
2.1.1. Relationales Modell

Das relationale Modell repräsentiert die Datenbank als Sammlung von Relationen. Jede Relation ähnelt einer Tabelle oder in gewissem Sinn einer »flachen« Datei mit Datensätzen.

Stellt man sich eine Relation als Tabelle mit Werten vor, so repräsentiert jede Zeile der Tabelle einen Datensatz. Beim relationalen Modell stellt jede Zeile einer Tabelle eine Tatsache dar, die normalerweise einer Entität oder eine Beziehung der realen Welt entspricht. Dabei werden die Namen der Tabelle und der Spalten dazu benutzt, um die Bedeutung der Werte in jeder Zeile zu interpretieren. Beispielsweise kann eine Tabelle *Student* heißen, da jede Zeile Fakten über eine bestimmte Studentenentität darstellt. Die Spaltennamen würden die Eigenschaften eines Studenten, wie beispielsweise Name, Anschrift, Alter, etc. repräsentieren. Alle Werte einer Spalte haben denselben Datentyp.

Formal werden im relationalen Modell Zeilen als *Tupel* und Spaltenüberschriften als *Attribute* bezeichnet. Die Tabelle selbst wird als *Relation* bezeichnet. Der Datentyp, der die Wertetypen beschreibt und in jeder Spalte erscheinen kann, wird als *Wertebereich* bezeichnet. Abbildung 2.1 zeigt eine Beispielrelation. Die Tupel einer Relation unterliegen keiner bestimm-

ten Ordnung. Es existiert zwar eine physische Ordnung, da die Tupel physisch gespeichert werden müssen. In der logischen Sicht jedoch sind die Tupel nicht geordnet. [17, S. 134f]



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

Abbildung 2.1.: Beispiel für eine Relation *Student* nach [17]

2.1.2. Einschränkungen

Im relationalen Modell können Einschränkungen getroffen werden. Es lassen sich Einschränkungen für Wertebereiche definieren. Der Wert eines Attributs muss dann einem definierten Wertebereich entsprechen. Die Wertebereiche in relationalen Datenbanken enthalten typischerweise Zahlen, reele Zahlen, Zeichenketten mit fester und variabler Länge, Datum-, Zeit- und Zeitstempeltypen, usw.

Eine Relation wird als Tupelmenge definiert. Alle Elemente einer Menge sind unterschiedlich. Demnach müssen auch alle Tupel einer Relation unterschiedlich sein. Also dürfen nie zwei Tupel die gleiche Wertekombination für alle ihre Atribute aufweisen. Normalerweise gibt es immer Teilmengen von Attributen eines Relationsschemas mit der Eigenschaft, dass keine zwei Tupel die gleiche Wertekombination für diese Attribute besitzen dürfen. Beispielsweise wäre dies für die in Abbildung 2.1 dargestellte Relation das eindeutige Attribut *SSN*, da keine zwei Studenten denselben SSN-Wert besitzen. Das Attribut SSN könnte als *Primärschlüssel* definiert werden, um einzelne Tupel voneinander zu unterscheiden.

Die *Entitätsintegritätseinschränkung* besagt, dass kein Primärschlüsselwert *NULL* sein darf. (Der Wert NULL besagt, dass das Feld keinen Eintrag hat. NULL repräsentiert also »nichts«.) Da Primärschlüssel immer eindeutig sein müssen, wären mehrere Tupel mit dem Primärschlüssel NULL nicht unterscheidbar.

Die Einschränkung der *referenziellen Integrität* wird benutzt, um die Konsistenz zwischen Tupeln unterschiedlicher Relationen zu wahren, die miteinander in Beziehung stehen. Informell besagt die referenzielle Integrität, dass ein Tupel der Relation, auf die sich ein Tupel einer anderen Replation bezieht, existieren muss. [17, S. 131ff]

2.1.3. Veränderungsoperationen

Zu den Veränderungsoperationen einer relationalen Datenbank zählen INSERT, DELETE und UPDATE. Jede dieser Operationen kann bestimmte Einschränkungen verletzen. Wenn eine Operation angewandt wird, muss nach der Ausführung der Operation der Datenbankzustand überprüft werden. Dabei wird sichergestellt, dass keine Einschränkungen verletzt wurden. [17, S. 131ff]

2.1.4. Relationale Algebra und SQL

Bei der relationalen Algebra handelt es sich um eine Menge von Operationen zur Suche auf Relationen, um Anfragen formulieren zu können. Dafür eignet sich z.B. die Abfragesprache SQL (Structured Query Language). Die Abfragesprache SQL hat sich als Standard für relationale Datenbanken durchgesetzt und hat gerade deswegen zum großen Erfolg relationaler Datenbanken beigetragen. Die Benutzer und Administratoren von Datenbankanwendungen müssen sich bei Migrationen von einem relationalen DBMS-Produkt zu einem anderen relationalen DBMS-Produkt nicht übermäßig viel Aufwand betreiben, da beide Systeme gleichen Sprachstandards folgen.

SQL ist eine umfassende Datenbanksprache. Sie bietet Anweisungen für die Datendefinition, Anfrage und Aktualisierung. Sie ist also eine *Data Definition Language* (DDL) und eine *Data Manipulation Language* (DML) in einem. Darüber hinaus können Sichten definiert, Sicherheits- und Autorisationsaspekte, Integritätseinschränkungen und Transaktionskontrollen spezifiziert werden. [17, S. 181f]

2.1.5. Transaktionen und ACID

Transaktionen bieten die Möglichkeit, logische Verarbeitungseinheiten auf einer Datenbank zu definieren. Auf große Datenbanken greifen gleichzeitig Hunderte von Benutzern zu und führen Transaktionen aus. Beispiele sind Systeme für Reservierungen, Banktransaktionen, Kreditkartenverarbeitung, Wertpapierhandel, Supermarktanwendungen und ähnliche Systeme. Hohe Verfügbarkeit und schnelle Antwortzeiten sind für solche Systeme unerlässlich. [17, S. 412]

Die wünschenswerten Eigenschaften einer Transaktion stellt ACID dar. ACID ist eine Abkürzung aus den Anfangsbuchstaben der Begriffe Atomicity, Consistency, Integrity und Durability (Atomizität, Konsistenz, Isolation und Dauerhaftigkeit). [17, S. 423] Im folgenden werden die vier Eigenschaften beschrieben:

1. *Atomizität*:
   Eine Transaktion ist eine atomare Verarbeitungseinheit. Entweder wird sie vollständig

oder überhaupt nicht ausgeführt. Für die Sicherstellung der Atomazität ist die Recoverykomponente eines DBMS zuständig. Schlägt die Ausführung einer Transaktion fehl, z.B. durch einen Systemabsturz mitten in einer Ausführung, muss die Recoverykomponente eventuelle Wirkungen auf die Datenbank rückgängig machen.

2. *Konsistenz*:
   Eine Transaktion wahrt die Konsistenz, wenn ihre vollständige Ausführung die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt. Die Konsistenz zu wahren ist meist die Aufgabe des Programmierers, der die Datenbankprogramme schreibt, oder der DBMS-Komponente, die Integritätseinschränkungen sicherstellt. Ein Datenbankprogramm sollte so geschrieben werden, dass die Datenbank, wenn sie sich in einem konsistenten Zustand befindet, sich ebenso in einem konsistenten Zustand befindet, nachdem eine Transaktion vollständig ausgeführt wurde.

3. *Isolation*:
   Eine Transaktion sollte so ausgeführt werden, als wäre sie isoliert von anderen Transaktionen. Die Ausführung einer Transaktion sollte also nicht von anderen, gleichzeitig ablaufenden Transaktionen gestört werden. Es gibt drei Stufen der Isolation. Eine Isolation der Stufe 0 liegt vor, wenn die Transaktion die *dirty reads* einer höherer Transaktionen nicht überschreibt. Eine Transaktion der Stufe 1 schließt das *Lost-Update*-Problem aus; eine der Stufe 3 schließt das Lost-Update-Problem aus und vermeidet dirty reads. Eine Transaktion mit Isolation der Stufe 3, weist zusätzlich zu den Eigenschaften der Stufe 2 wiederholbare Leseoperationen auf. Isolation der Stufe 3 wird auch als *echte Isolation* bezeichnet.

4. *Dauerhaftigkeit*:
   Die von einer bestätigten Transaktion in die Datenbank geschriebenen Änderungen müssen in der Datenbank fortbestehen. Aufgrund eines Fehlers dürfen die Änderungen nicht verloren gehen. Die Dauerhaftigkeit liegt im Verantwortungsbereich der Recoverykomponente des DBMS.

[17, S. 423f]

2.2. NoSQL

In der Einführung wurde bereits auf neue, nicht-relationale Datenbanken hingewiesen. Dabei fiel auch der Begriff *NoSQL*. Dieser bezeichnet eine Reihe nicht-relationaler Datenbanksysteme. Eine Definition, was als ein NoSQL-Datenbanksystem angesehen werden kann, findet man unter [4]:

> »Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontal scalable. The original intention has been modern web-scale databases. [...] Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge data amount, and more. So the misleading term »nosql« [...] should be seen as an alias to something like the definition above.«

Somit sollte also ein Datenbanksystem mindestens einen der Punkte erfüllen, um als NoSQL-Datenbanksystem angesehen zu werden:

- Das zugrunde liegende Datenmodell ist nicht relational.
- Das System ist verteilt.
- Es ist Open Source.
- Das System lässt sich horizontal skalieren.
- Das System ist schemafrei.
- Einfache Replikation wird vom System angeboten.
- Eine einfache API wird bereitgestellt.
- Dem System liegt ein anderes Konsistenzmodell zugrunde: Eventually Consistent und BASE, jedoch nicht ACID.

Open Source als Merkmal für ein NoSQL-Datenbanksystem zu sehen ist allerdings nicht angebracht. Denn Open Source alleine ist kein eindeutiges Merkmal für ein NoSQL-System, wie MySQL[1] oder PostgreSQL[2] beweisen. Außerdem gibt es auch NoSQL-Systeme, die nicht Open Source sind, wie BigTable von Google oder Dynamo von Amazon.

2.3. MapReduce

Da die Menge an Daten und Informationen rasant zunimmt, werden neue alternative Algorithmen, Frameworks und DBMS entwickelt. Bei der Verarbeitung großer Datenmengen spielt das MapReduce-Verfahren eine entscheidende Rolle. MapReduce wurde von Jeffrey Dean und Sanjay Ghemawa im Jahr 2004 bei Google entwickelt. [13] Es ist ein Framework, welches die Möglichkeit bietet, nebenläufige Berechnungen großer Datenmengen in Computerclustern durchzuführen. Durch die Nebenläufigkeit reduziert sich die für die Berechnung benötigte Zeit. Die folgende Darstellung, soweit nicht anders angegeben, basiert auf [16, S. 12ff] und [13].

[1] http://www.mysql.com/
[2] http://www.postgresql.org/

2.3.1. Funktionale Hintergründe

Um Prozesse parallelisieren zu können, dürfen die Ausgangsdaten nicht manipuliert werden, da sonst nicht sichergestellt werden kann, dass die parallel laufenden Prozesse dieselben Daten erhalten. Funktionale Sprachen haben die Eigenschaft, dass Daten nicht manipuliert werden, sondern bei jeder Berechnung nur auf Kopien der Daten gearbeitet wird. Es entstehen so keine Seiteneffekte, wie *Dead Locks* oder *Race Conditions*. Somit beeinflussen sich unterschiedliche Operationen, die auf denselben Daten arbeiten, nicht gegenseitig. Da keine Seiteneffekte vorhanden sind, spielt auch die Ausführungsreihenfolge der Operationen keine Rolle. Erst hierdurch wird die Parallelisierung der Operationen ermöglicht.

Die Funktionen `map()` und `reduce()` (auch als `fold()` bezeichnet) sind aus der funktionalen Programmierung bekannt. Sie werden in modifizierter Form im MapReduce-Framework nebenläufig in zwei Phasen verwendet. Die Funktion `map()` wendet eine Funktion auf alle Elemente einer Liste an und gibt eine modifizierte Liste zurück. Die Funktion `reduce()` fasst einzelne Listenpaare zusammen und reduziert diese auf einen Ausgabewert. Im MapReduce-Framework werden die beiden Funktionen in zwei Phasen hintereinander angewendet. Dabei lassen sich beide Phasen auf verschiedene Knoten im Netzwerk verteilen. Jede Phase lässt sich parallelisieren und ermöglicht so eine beschleunigte Berechnung großer Datenmengen. Parallelisierung wird bei großen Datenmengen unter Umständen schon dadurch benötigt, da die Datenmengen für einen einzelnen Knoten zu groß sind.

2.3.2. Arbeitsweise

Die grundlegende Arbeitsweise eines MapReduce-Verfahrens lässt sich gut anhand seines Datenflusses erklären (siehe Abbildung 2.2):

- Die Eingabedaten werden auf verschiedene Map-Prozesse verteilt.
- In der Map-Phase berechnen die Map-Prozesse parallel die vom Nutzer bereitgestellte Map-Funktion.
- Jede dieser Map-Instanzen legt Zwischenergebnisse in verschiedene Zwischenergebnisspeicher ab.
- Sobald alle Zwischenergebnisse berechnet sind, ist die Map-Phase beendet.
- Anschließend wird in der Reduce-Phase für jedes Zwischenergebnis genau ein Reduce-Prozess gestartet. Dieser berechnet die vom Nutzer bereitgestellte Reduce-Funktion. Die Reduce-Prozesse laufen ebenfalls parallel ab.
- Sobald alle Reduce-Prozesse beendet sind, liegen auch die Ergebnisdaten bereit. Die gesamte Durchführung des MapReduce-Verfahrens ist nun abgeschlossen.

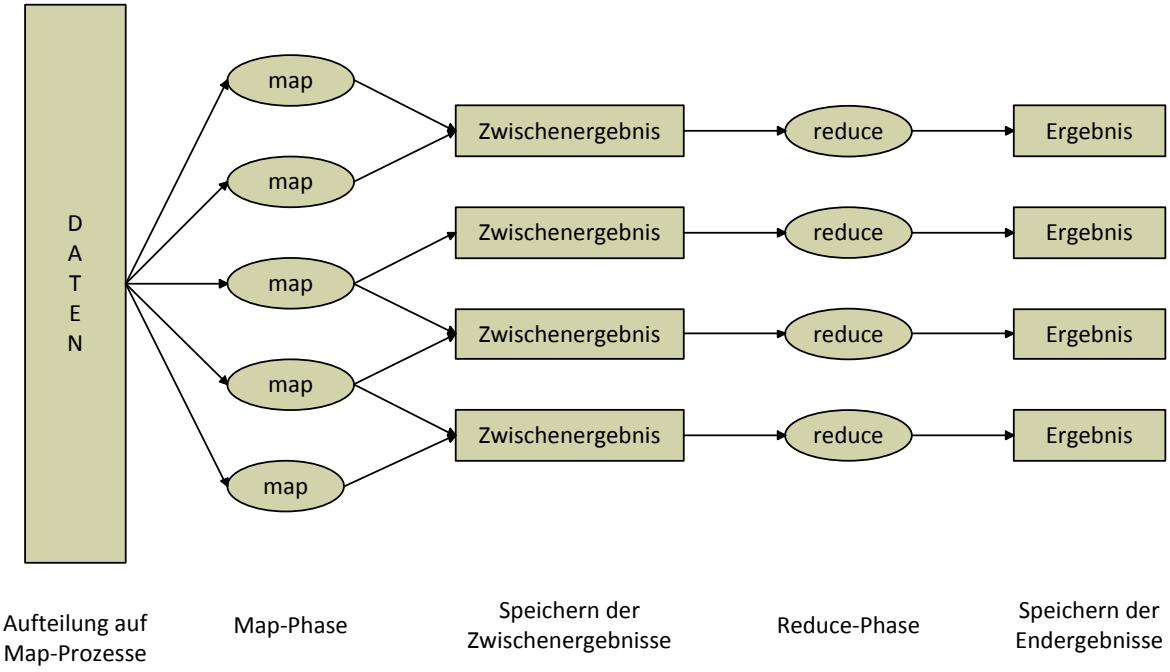


Abbildung 2.2.: Datenfluss und Phasen des MapReduce-Verfahrens

Die map- und reduce-Funktion muss vom Anwender erstellt werden. Das MapReduce-Framework stellt eine Trennung zwischen Anwendungslogik und der technischen Seite dar. Dem Anwender wird es ermöglicht, sich auf die Anwendungslogik zu konzentrieren, während sich das Framework um die technischen Details kümmert. Das Framework übernimmt folgende Aufgaben:

- Die automatische Parallelisierung und Verteilung der Prozesse.
- Die Fehlertoleranz bei Hardware- oder Softwarefehlern.
- Das I/O-Scheduling.
- Die Bereitstellung von Statusinformationen und Überwachungsmöglichkeiten.

Der Anwender muss lediglich die beiden Funktionen map() und reduce() spezifizieren, welche nachfolgend im Pseudocode angegeben sind.

```
map (in_key, in_val) -> list(out_key, intermediate_val)
reduce (out_key, list(intermediate_val)) -> list(out_val)
```

Die `map`-Funktion hat zwei Argumente, ein Key/Value-Paar: `(in_key, in_val)`. Daraus bildet die `map`-Funktion eine Liste von neuen Key/Value-Paaren: `list(out_key, intermediate_val)`. Diese sind die Zwischenergebnisse der Map-Phase. In der anschließenden Reduce-Phase werden die Werte der Zwischenergebnisse zu einem bestimmten

Schlüssel `out_key` als neue Liste kombiniert: `(out_key, list(intermediate_val)`. Diese Werte sind die Eingabewerte der reduce-Funktion, die daraus einen Satz fusionierter Ergebnisse berechnet.

2.3.3. Komponenten und Architektur

Wie in [13] beschreiben, sind viele verschiedene Implementierungen eines MapReduce-Frameworks möglich. Zum Beispiel könnte ein solches Framework für die Verarbeitung von Daten im *shared memory* optimiert sein. Ein anderes Framework könnte für eine große *NUMA*[3]-Architektur oder für einen Einsatz in großen verteilten Netzwerken implementiert sein. Googles MapReduce-Framework ist für die Verarbeitung sehr großer Datenmengen in einem Ethernet-Netzwerk mit Standardhardware entworfen. Folgende Umgebung liegt vor:

- Rechner mit Standard Dual-Core-Prozessoren x86.
- 2 bis 4 GB Arbeitsspeicher pro Rechner.
- Linux als Betriebssystem.
- Standardnetzwerkkarte mit 100 Mbit/s - 1 Gbit/s.
- Rechnercluster bestehend aus hunderten oder tausenden Rechnern.
- Speicherung der Daten auf Standard-IDE-Festplatten mittels GFS (Google File System, siehe [18]). Das GFS ist ein spezielles Dateisystem, welches benötigt wird, um große Datenmengen im Petabyte-Bereich zu verteilen.

Das von Google entwickelte MapReduce-Framework hat die in Abbildung 2.3 dargestellte Architektur. Der Datenfluss wird im Folgenden in erweiterter Darstellung genauer beschrieben:

1. Die MapReduce-Library, die im Anwendungsprogramm enthalten ist, teilt zunächst die Eingabedateien in `M` Teile auf. Diese haben typischerweise eine Größe zwischen 16 und 64 MB. Anschließend werden Kopien des Programms auf mehreren Rechnern innerhalb des Rechnerclusters gestartet.
2. Eine der Kopien des Programms hat eine besondere Rolle und wird als Master bezeichnet. Die anderen Kopien sind allesamt Worker. Es gibt `M` Map-Aufgaben und `R` Reduce-Aufgaben. Der Master weist den Workern die Aufgaben zu.
3. Ein Worker, der eine Map-Aufgabe bekommen hat, liest den zugeteilten Inhalt der aufgeteilten Eingabedatei. Er analysiert die Key/Value-Paare der Eingabedaten und verarbeitet sie in der durch den Nutzer definierten Map-Funktion. Die Zwischenergebnisse werden im Speicher als neue Key/Value-Paare gespeichert.

[3] Non-Uniform Memory Architecture

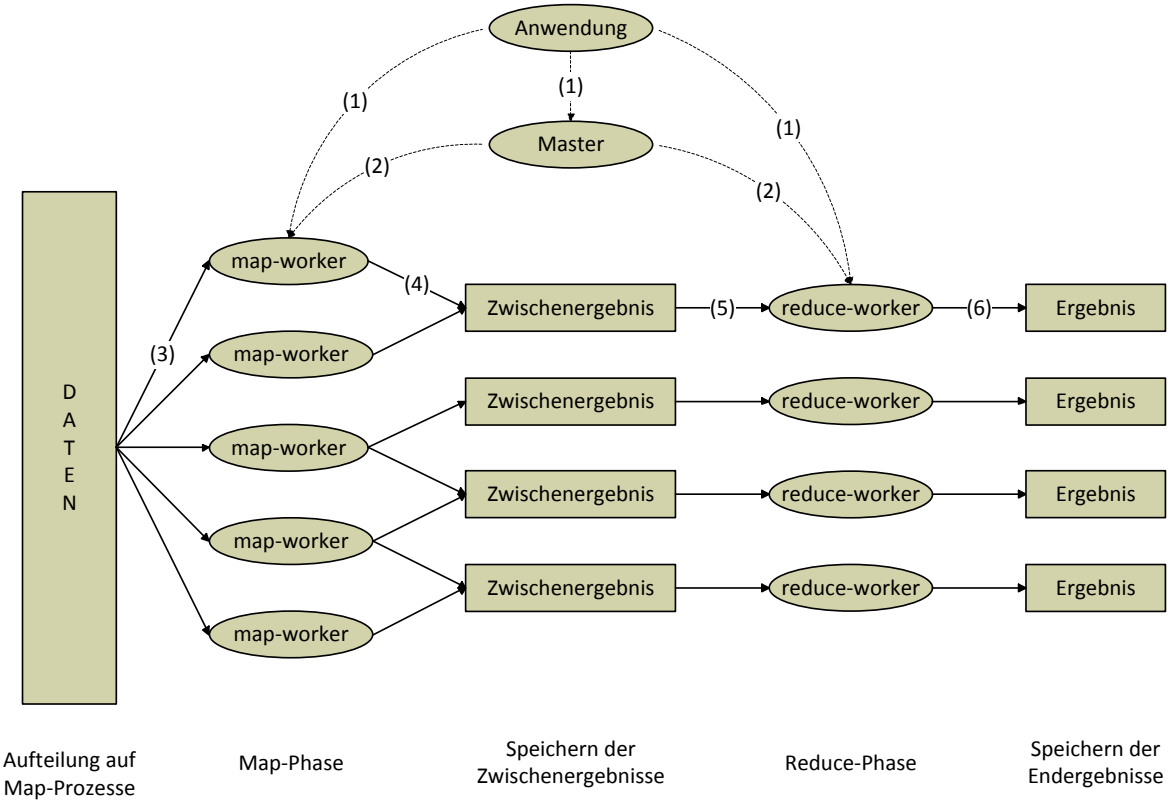


Abbildung 2.3.: Erweitertes Datenfluss- und Phasenmodell

4. Die zwischengespeicherten Key/Value-Paare werden periodisch auf die lokale Festplatte geschrieben und durch eine Partitionierungsfunktion in `R` Partitionen aufgeteilt. Die Adressen dieser Partitionen werden dem Master mitgeteilt. Dieser ist für die Weiterleitung der Adressen an die Reduce-Worker zuständig.

5. Erhält ein Reduce-Worker die Adressdaten vom Master, greift er per RPC auf die Daten zu. Sobald alle Zwischenergebnisse gelesen sind, sortiert der Reduce-Worker die Daten anhand des Schlüssels und gruppiert alle Daten mit dem gleichen Schlüssel.

6. Der Reduce-Worker iteriert über die sortierten Zwischenergebnisse. Anschließend übergibt er die Key/Value-Paare für jeden Schlüssel an die durch den Nutzer definierte Reduce-Funktion. Diese fasst die Liste von Werten der Partition zu einem Ausgabewert zusammen.

7. Sobald alle Map- und Reduce-Aufgaben beendet sind, also alle Map- und Reduce-Worker ihre Aufgaben erledigt haben, wird die Kontrolle vom Master wieder an das Anwenderprogramm zurückgegeben.

Nach erfolgreicher Bearbeitung sind die Ergebnisse in R Ausgabedateien zu finden. Diese können zusammengefasst werden oder als Eingabedateien für einen weiteren MapReduce-Durchlauf verwendet werden.

2.3.4. Beispiel und Einsatzgebiete

Ein Beispiel für die Anwendung des MapReduce-Verfahrens wäre das Zählen der Häufigkeit eines Wortes in einer großen Anzahl von Dokumenten. Dabei würde der Nutzer des MapReduce-Frameworks in etwa diesen Code schreiben:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    for each v in values:
        result = result + ParseInt(v);
    Emit(AsString(result));
```

Die `map`-Funktion durchsucht ein Dokument nach allen vorkommenden Wörtern. Sie speichert für jedes gefundene Wort das Wort selbst und eine ’1’ als Liste ab: `(w, "1")`. Die `reduce`-Funktion summiert für ein bestimmtes Wort die Zwischenergebnisse auf und gibt die Anzahl, wie oft das Wort in allen Dokumenten vorkommt, zurück.

Weitere Einsatzgebiete für ein MapReduce-Framework sind u.a.:

- *Verteiltes Suchen (Grep)*
  Die Map-Funktion liefert eine Zeile, falls der Inhalt dem gegebenen Suchmuster entspricht. Die Reduce-Funktion ist nur eine Identitätsfunktion, welche die Zwischenergebnisse zur Liste der Endergebnisse kopiert.
- *Zählen von Zugriffen auf eine URL*
  Die Map-Funktion verarbeitet die Zugriffe auf Webseiten anhand von Log-Dateien und gibt `(URL, "1")` zurück. Die Reduce-Funktion addiert alle Zugriffe für dieselbe URL und liefert so die Gesamtzugriffe pro URL: `(URL, total_count)`.
- *Erstellung von Graphen über Verlinkung von Webseiten zu einem Ziel*
  Die Map-Funktion erzeugt Key/Value-Paare in der Form `(target, source)` von jedem Link einer Ziel-URL (`target`), die auf einer Quell-URL (`source`) gefunden wurde. Die Reduce-Funktion verkettet die Liste aller Quell-URLs in Verbindung mit der Ziel-URL und liefert `(target, list(source))` zurück.

- *Ermittlung des Term-Vektors per Host*
  Ein Term-Vektor fasst die wichtigsten Worte zusammen, die in einem Dokument oder einer Gruppe von Dokumenten vorkommen. Der Vektor wird in der Form `(word, frequency)` dargestellt. Die Map-Funktion liefert für jedes zum Host gehörende Dokument `(hostname, term_vector)`. Die Reduce-Funktion prüft alle Term-Vektoren pro Dokument und fasst diese Vektoren für die Dokumente zusammen. Sie addiert diese Term-Vektoren und verwirft dabei Terme mit geringer Frequenz. Es wird ein abschließendes Paar `(word, term_vector)` geliefert.
- *Invertierter Index*
  Die Map-Funktion analysiert jedes Dokument und gibt eine Liste von Wort-Dokumentenpaaren zurück: `(word, document_id)`. Die Reduce-Funktion sortiert die Zwischenergebnisse für jedes Wort nach der Dokument-ID und fasst diese als Liste zusammen. Als Ergebnis wird ein invertierter Wortindex in der Form `(word, list(document_id))` zurückgegeben.

2.3.5. Zusammenfassung

Das MapReduce-Verfahren spielt im Bereich der NoSQL-Datenbanken eine entscheidende Rolle. Durch das Verfahren lassen sich große verteilte Datenmengen bei paralleler Ausführung effizient durchsuchen. Einige NoSQL-Datenbanken, wie CouchDB, MongoDB, Riak und HBase nutzen das Verfahren, um Abfragen auf ihre Datenbankeinträge durchzuführen.

2.4. Konsistenzmodelle

In diesem Abschnitt wird gezeigt, dass neue Konsistenzmodelle für die Anforderungen von Web 2.0-Projekten notwendig geworden sind. Das *CAP*-Theorem geht auf die Konsistenz und damit verbundene Probleme in verteilten Datenbanken ein. Anschließend wird das zu ACID (vgl. 2.1.5) alternative Konsistenzmodell *BASE* beschrieben.

2.4.1. Neue Anforderungen

Als der Internet-Boom einsetzte, haben Start-Up-Unternehmen damit begonnen, freie relationale Datenbanken wie MySQL und PostgreSQL einzusetzen. Die anfallenden Datenmengen wurden jedoch bereits bei mittelgroßen Web 2.0-Unternehmen sehr groß. Es reichte nicht mehr aus vertikal zu skalieren, also einen leistungsstärkeren Server einzusetzen. Die horizontale Skalierung, also der Einsatz mehrerer Server, wurde diskutiert.

Diese Tatsache zwang die Hersteller relationaler Datenbanken sich mit der horizontalen Skalierung auseinanderzusetzen und Lösungen bereitzustellen. Allerdings stellte sich die Konsistenz relationaler Datenbanken als keine leicht zu bewältigende Hürde dar. Es ließ sich nicht einfach realisieren, die Konsistenz der Datenbank sicherzustellen und dabei gleichzeitig horizontale Skalierung, Replikation und eine niedrige Reaktionszeit zu gewährleisten. Die Hersteller relationaler Datenbanken haben sich jedoch damit schwer getan, die Konsistenz aufzugeben.

Das bewährte Denkmuster, eine Datenbank vom Prinzip der Konsistenz her aufzubauen funktionierte mit den neuen Anforderungen nicht. [16, S. 30f]

2.4.2. CAP-Theorem

CAP steht für *Consistency* (Konsistenz), *Availability* (Verfügbarkeit) und *Partition Tolerance* (Partitionstoleranz). Eric Brewer stellte im Jahr 2000 in [1] das *CAP-Theorem* vor. Der Vortrag ist das Ergebnis seiner Forschungen zu verteilten Systemen an der University of California. Dabei zeigt Brewer auf, warum es nicht möglich ist, die drei genannten Größen in einer verteilten Datenbank vollständig zu vereinen. In einer verteilten Datenbank sind maximal zwei dieser Größen erreichbar (siehe Abbildung 2.4).

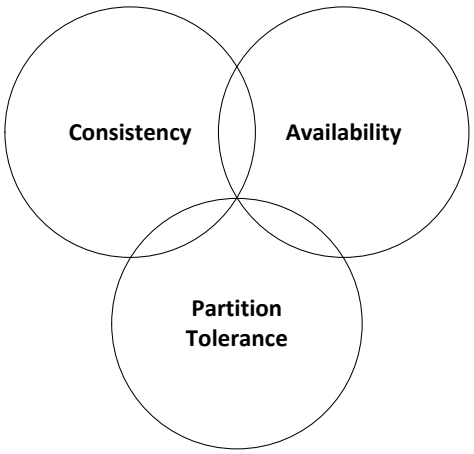


Abbildung 2.4.: CAP-Theorem und seine Schnittmengen

Ein System, das keine Partitionstoleranz unterstützt, kann Konsistenz und Verfügbarkeit bieten. Dies wird oft durch Transaktionsprotokolle gewährleistet. Dabei sind der Client und die Datenbank Teil derselben Umgebung. In größeren verteilten Systemen kommt es vor, dass Verbindungen jederzeit abbrechen können. Daher können Konsistenz und Verfügbarkeit nicht zeitgleich gewährleistet werden. Das bedeutet, dass zwei Möglichkeiten bestehen:

1. Eine schwächere Konsistenz erlaubt, dass das System hochverfügbar bleibt, trotz möglicher Ausfälle von Teilen des Netzwerks.

2. Eine starke Konsistenz bedeutet, dass das System unter bestimmten Umständen nicht verfügbar ist.

Abbildung 2.5 verdeutlicht dies.

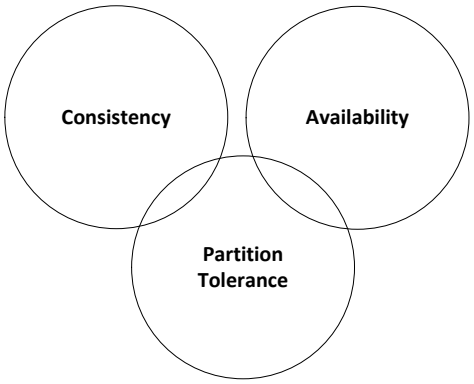


Abbildung 2.5.: CAP-Theorem und seine Schnittmengen bei verteilten Datenbanken

In beiden Fällen muss der Entwickler des Clients wissen, was ihm das verteilte System bietet. Falls das System auf starke Konsistenz ausgelegt ist, muss der Entwickler damit umgehen, dass das System eventuell nicht verfügbar ist, um zum Beispiel eine Schreiboperation auszuführen. Wenn die Schreiboperation fehlschlägt, muss der Entwickler entscheiden, was mit den zu schreibenden Daten passieren soll. Falls das System hohe Verfügbarkeit bietet, kann es zwar immer die Schreiboperation ausführen, aber es könnte der Fall eintreten, dass unter bestimmten Bedingungen ein anschließender Lesebefehl nicht die zuvor geschriebenen Daten liefert. Der Entwickler hat dann zu entscheiden, ob der Client immer Zugriff auf die zuletzt geschriebenen Daten benötigt. [31]

Beispiel zur Verdeutlichung des CAP-Theorems

Um das CAP-Theorem zu verdeutlichen, wird ein Beispiel gezeigt. Es orientiert sich dabei an einem formalen Beweis des CAP-Theorems von Gilbert und Lynch. [19] Zum besseren Verständnis wird das Beispiel jedoch vereinfacht.

Es gibt eine einfache, aus zwei Knoten K1 und K2 bestehende Datenbank. Die Knoten stellen Replikationen derselben Daten D0 dar. K1 ist für Schreiboperationen zuständig, während K2 für Leseoperationen zuständig ist. Abbildung 2.6 verdeutlicht das Szenario.

Nun wird eine Schreiboperation ausgeführt und K1 wechselt in den Zustand D1. Durch die Nachricht M wird K2 durch den Synchronisationsmechanismus der verteilten Datenbank aktualisiert (siehe Abbildung 2.7 (1)). Eine darauf folgende Leseoperation erhält von K2 den neuen Zustand D (siehe Abbildung 2.7 (2)).

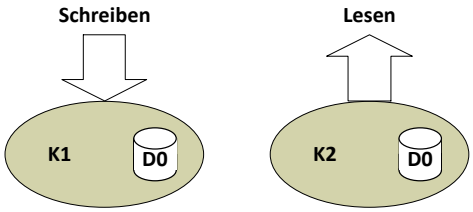


Abbildung 2.6.: CAP-Szenario

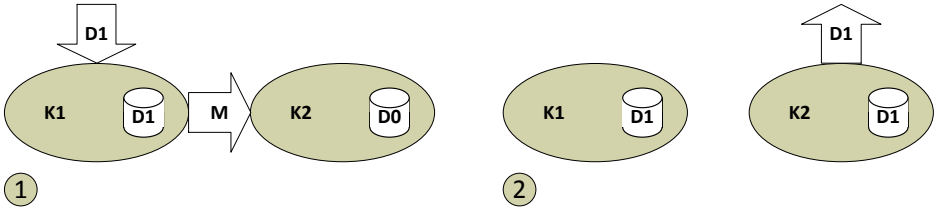


Abbildung 2.7.: CAP-Szenario: Synchronisation

Es kann aber vorkommen, dass die Kommunikation zwischen K1 und K2 durch einen Netzwerkausfall nicht mehr möglich ist. Dann ist es nicht möglich, dass K2 nach der Schreiboperation auf K1 synchronisiert wird, da die Nachricht M K2 nicht erreicht (siehe Abbildung 2.8 (1)). Falls das System ein Transaktionsprotokoll verwendet, das erst bei vollständiger Synchronisation aller Knoten die Transaktion abschließt, wären durch eine solchen Netzwerkausfall Teile der Daten auf K1 blockiert. Dies würde bei weiteren Schreiboperationen schnell zu einem Einbruch der Verfügbarkeit des Systems führen. Die Verfügbarkeit kann nur dann gewährleistet werden, wenn akzeptiert wird, dass die Daten auf K1 und K2 nicht mehr konsistent sind. K2 behielte den Stand der Daten D0. K1 würde Schreiboperationen ausführen, ohne zu gewährleisten, dass K2 auf den neuen Zustand D1 synchronisiert wird (siehe Abbildung 2.8 (2)).

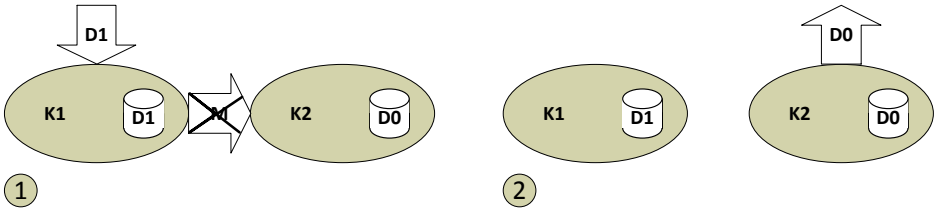


Abbildung 2.8.: CAP-Szenario: Verbindungsausfall

Wie an dem Beispiel zu sehen, ist es in der Praxis nicht möglich, Konsistenz und Verfügbarkeit in einer verteilten Datenbank gleichzeitig zu gewährleisten.

Kritik am CAP-Theorem

Das CAP-Theorem ist nicht unumstritten. Zu den bekanntesten Kritikern gehört Micheal Stonebreaker. In den Blogposts [28] und [27] erläutert er, weshalb das CAP-Theorem kritisch zu sehen ist.

Es gibt unter anderem folgende möglicher Fehler, die in einem DBMS auftreten können:

1. *Anwendungsfehler*
   Die Anwendung führt ein oder mehrere fehlerhafte Updates aus. Die Datenbank muss anschließend in einen Zustand gebracht werden, bevor die fehlerhafte Transaktion oder die fehlerhaften Transaktionen durchgeführt wurden. Die anschließenden Aktionen müssen wiederholt werden.

2. *Wiederholbare DBMS-Fehler*
   Das DBMS stürzt bei der Ausführung einer Operation auf einem Knoten ab. Würde dieselbe Transaktion auf einem replizierten Knoten ausgeführt werden, würde der replizierte Knoten ebenfalls abstürzen. Diese Fehler werden auch als *Bohrbugs* bezeichnet.

3. *Nicht wiederholbare DBMS-Fehler*
   Die Datenbank eines Knotens stürzt ab, aber die replizierten Knoten laufen weiter. Diese Fehler sind äußerst schwer oder gar nicht zu reproduzieren, da die Rahmenbedingungen hierfür extrem schwer wiederhergestellt werden können. Diese Art von Fehlern werden auch als *Heisenbugs* bezeichnet.

4. *Betriebssystemfehler*
   Das Betriebssystem eines Knotens stürzt ab.

5. *Hardwarefehler in einem lokalen Cluster*
   Beispielsweise sind dies Speicherfehler, Festplattenfehler, etc. Sie verursachen einen Stopp des Systems durch das Betriebssystem oder durch das DBMS selbst. Teilweise sind diese Fehler auch Heisenbugs.

6. *Partition in einem lokalen Cluster*
   Das LAN bricht zusammen und die Knoten können nicht mehr miteinander kommunizieren.

7. *Katastrophen*
   Der lokale Cluster wird durch eine Katastrophe (Flut, Erdbeben, etc.) ausgelöscht. Der Cluster existiert nicht mehr.

8. *Netzwerkfehler im WAN, das die Cluster verbindet*
   Das WAN bricht zusammen und die Cluster können nicht mehr miteinander kommunizieren.

Die Fehler 1 und 2 bewirken, dass eine Hochverfügbarkeit nicht gewährleistet werden kann. Außerdem ist die Konsistenz der replizierten Knoten nicht mehr gegeben. Fehler 7 kann nur dadurch abgedeckt werden, wenn eine lokale Transaktion nur dann committed wird, nachdem sicher ist, dass die Transaktion von einem anderen Cluster im WAN empfangen wurde. Teilweise wird die Latenz, die dabei entsteht, in Kauf genommen. Allerdings kann hierbei keine Eventual Consistency (siehe 2.4.3) garantiert werden, da eine Transaktion komplett

verloren gehen kann, wenn eine Katastrophe einen lokalen Cluster trifft, bevor die Transaktion erfolgreich an einen weiteren Cluster weitergeleitet werden konnte. Der Datenverlust wird also im Falle einer seltenen Katastrophe in Kauf genommen, da die Performance-Einbußen zu hoch sind, um diesen zu vermeiden.

Die Fehler 1, 2 und 7 sind Beispiele dafür, dass das CAP-Theorem ungültig ist. Jedes System muss auf diese Fehler in der Praxis vorbereitet sein.

Die lokalen Fehler 3, 4, 5 und 6 deckt das CAP-Theorem hingegen ab. Hierbei stürzt ein einzelner Knoten ab, es entsteht also eine Partition. Es gibt zahlreiche Algorithmen, die einen solchen Fall abdecken, sodass das System weiterlaufen kann. Eine Partition ist in der Praxis jedoch sehr selten. Daher wäre es besser die Ausfalltoleranz als die Konsistenz aufzugeben.

Fehler 8, eine Partition in einem WAN, ist in heutigen WANs sehr selten. Lokale Fehler und Anwendungsfehler treten häufiger auf. Die meisten WAN-Fehler sind solche, bei denen ein kleiner Teil des Systems vom Rest getrennt wird. In diesem Fall kann der Großteil des Systems weiter ausgeführt werden, während nur ein kleiner Teil geblockt wird. Man gibt also die Konsistenz dauerhaft auf, um die Verfügbarkeit einer geringen Anzahl von Knoten in einem sehr seltenen Szenario zu gewährleisten.

Wie Daniel Abadi [7] anführt, ist das CAP-Theorem zusätzlich unter den folgenden Gesichtspunkten kritisch zu sehen.

Das CAP-Theorem impliziert, dass es drei Arten von verteilten Systemen gibt:

1. *CA-Systeme*
   konsistent und verfügbar, jedoch nicht partitionstolerant
2. *CP-Systeme*
   konsistent und partitionstolerant, jedoch nicht verfügbar
3. *AP-Systeme*
   verfügbar und partitionstolerant, aber nicht konsistent

Die Definition von CP impliziert, dass ein solches System niemals erreichbar ist. Vielmehr ist jedoch gemeint, dass die Verfügbarkeit nicht garantiert werden kann, falls Partitionen existieren. In der Praxis bedeutet das also, dass die Rolle von A und C im CAP-Theorem asymmetrisch ist. Systeme, welche die Konsistenz aufgeben (AP) neigen dazu, dies dauerhaft zu tun, nicht nur falls Partitionen auftreten.

Es gibt praktisch keine Differenz zwischen CA- und CP-Systemen. Wie bereits geschildert, geben CP-Systeme die Verfügbarkeit nur bei existierenden Partitionen auf. CA-Systeme tolerieren gar keine Partitionen. Falls in der Praxis doch Partitionen auftreten, wird jedoch auch hier die Verfügbarkeit aufgegeben. Somit sind CA- und CP-Systeme im Wesentlichen identisch. Es gibt in der Praxis also nur zwei Arten von verteilten Systemen: CP-/CA- und AP-Systeme.

Außerdem liegt der Hauptfokus des CAP-Theorems auf dem Kompromiss zwischen Konsistenz und Verfügbarkeit. Daraus resultiert die Vorstellung, dass NoSQL-Systeme die Konsistenz aufgeben, um Verfügbarkeit zu gewährleisten. Dies entspricht jedoch nicht der Realität, wie Yahoos *PNUTS* [12] zeigt. PNUTS lockert die Konsistenz und gibt zusätzlich die Verfügbarkeit auf (falls die Master-Kopie eines Datenelements nicht erreichbar ist, so ist dieses Element für Updates gesperrt). Der Grund dafür, dass sowohl die Konsistenz als auch die Verfügbarkeit aufgegeben werden, ist die Verringerung der Latenz. Um replizierte Knoten über ein WAN konsistent zu halten, tauschen die Knoten Nachrichten aus. Dies steigert die Latenz einer Transaktion. Für viele Webanwendungen ist die größere Latenz nicht hinnehmbar (wie beispielsweise Anwendungen von Amazon und Yahoo). Um die Latenz zu reduzieren, müssen Replikationen asynchron durchgeführt werden. Dies reduziert die Konsistenz.

Das CAP-Theorem beinhaltet also kein vollständiges Fehlermodell. Es weist außerdem eine Asymmetrie von C, A und P auf und Latenzüberlegungen sind nicht enthalten. Diese Gründe zeigen auf, dass das CAP-Theorem nicht allein als Erklärung dafür genutzt werden kann, die Konsistenz aufzugeben, um Verfügbarkeit zu erreichen. Selbst Eric Brewer hat mittlerweile eingesehen, dass das CAP-Theorem unzulänglich ist und es ergänzt werden sollte. [10]

2.4.3. Alternatives Konsistenzmodell BASE

Um das CAP-Theorem zu lösen, wurde ein neues Modell zur Betrachtung der Konsistenz in verteilten Datenbanken herangezogen: *BASE* (Basic Available, Soft State, Eventual Consistency). BASE wird als Gegenpart zum klassischen ACID-Modell gesehen. [24]

Bei BASE dreht sich im Gegensatz zu ACID alles um Verfügbarkeit. Die Konsistenz wird dabei der Verfügbarkeit untergeordnet. Während ACID einen pessimistischen Ansatz bei der Konsistenz darstellt, stellt BASE einen optimistischen Ansatz dar. Konsistenz ist dabei kein Zustand, der nach jeder Transaktion erreicht wird, sondern ein Übergangsprozess. Die Konsistent ist also nicht dauerhaft gegeben und es liegt eine eventuelle Konsistenz vor - *Eventually Consistency*. Das System garantiert, dass falls keine weiteren Updates auf ein Datenelement getätigt werden, eventuell alle Zugreifer den aktuellen Wert des Elements erhalten. Insbesondere bei Systemen mit einer Vielzahl an replizierenden Knoten wird die Konsistenz erst nach einem Zeitfenster der Inkonsistenz erreicht. Falls keine Fehler auftreten, lässt sich die maximale Dauer des Zeitfensters durch Faktoren wie der Anzahl der replizierenden Knoten, den durchschnittlichen Reaktionszeiten und der durchschnittlichen Last des Systems bestimmen. [31]

Dabei hat man nicht die eindeutige Wahl zwischen ACID oder BASE, sondern hat eher ein Spektrum aus beiden zur Auswahl, wie Eric Brewer betont. [1] Ein Datenbanksystem liegt daher entweder näher an ACID oder näher an BASE. Bekannte relationale Datenbanken nähern sich eher ACID an, während NoSQL-Datenbanken meist näher an BASE liegen. Jedoch ist die Vielfalt der Ansätze bei NoSQL-Datenbanken entscheidend größer als die der relationalen Datenbanken.

Daher muss man jede NoSQL-Datenbank näher betrachten, um das zugrunde liegende Modell zu identifizieren. Um die Orientierung zu erleichtern, lässt sich die von Werner Vogels aufgestellte Liste von Merkmalen zur Charakterisierung verteilter Datenbanken verwenden. [31]

Angenommen, es liegt folgendes System vor:

- *Ein Datenbanksystem*
  Es sei zunächst angenommen, es handelt sich dabei um eine Black-Box. Jedoch sollte man davon ausgehen, dass es sich hierbei um ein hochskalierbares und weit verteiltes System handelt, welches Hochverfügbarkeit und Dauerhaftigkeit garantiert.
- *Prozesse A und B*
  Beide Prozesse greifen schreibend und lesend auf die Datenbank zu. Sie sind unabhängig voneinander.

Dabei gibt es folgende Fälle für eventuelle Konsistenz:

- *Causal Consistency*
  A schreibt den Wert X in die Datenbank. Anschließend liest B den Wert X und schreibt danach den Wert Y in die Datenbank. Man geht davon aus, dass B den Wert X für Vorberechnungen verwendet (auch wenn das nicht der Fall ist), die dazu führen, dass Y in die Datenbank geschrieben wird. A und B sind kausal abhängig. Causal Consistency ist also gegeben, wenn trotz der zeitlichen Nähe der von A und B ausgeführten Operationen gewährleistet ist, dass B den von A geschriebenen Wert X erhält, bevor Y geschrieben wird.
- *Read-your-write Consistency*
  Dies ist ein Spezialfall der Causal Consistency. Schreibt A einen Wert in die Datenbank, so erhält A anschließend immer den von ihm geschriebenen Wert und in keinem Fall eine ältere Version des Datenelements.
- *Session Consistency*
  Im Rahmen einer Session wird auf die Datenbank zugegriffen. Solange die Session existiert, herrscht Read-your-write Consistency. Falls die Session beendet wird (eventuell durch einen Fehler), muss eine neue Session erstellt werden. Die Read-your-write Consistency existiert dann nur noch für die innerhalb der neuen Session vorgenommenen Operationen.
- *Monotonic Read Consistency*
  Falls ein Prozess bei einer Leseoperation einen bestimmten Wert eines Datenelements erhalten hat, wird er bei darauffolgenden Leseoperationen keine ältere Version des Datenelements erhalten.

- *Monotonic Write Consistency*
  Das System garantiert, dass Schreiboperationen eines Prozesses serialisiert ausgeführt werden. Die Schreiboperationen auf der Datenbank werden also in genau derselben Reihenfolge ausgeführt, in der sie angestoßen wurden. Systeme, die keine Monotonic Write Consistency gewährleisten, sind in der Programmierung schwer zu handhaben. Falls mehrere in kurzem zeitlichen Abstand aufeinanderfolgende Schreiboperationen desselben Prozesses auf demselben Datenelement ausgeführt werden, ist nicht sichergestellt, welcher Wert anschließend in der Datenbank steht.

Diese Merkmale lassen sich miteinander kombinieren. Beispielsweise ist es möglich, *Monotonic Read Consistency* mit *Session Consistency* zu kombinieren. In der Praxis sind diese zwei Merkmale die wünschenswertesten Konsistenzeingenschaften in einem System, das Eventual Consistency aufweist, denn sie ermöglichen die Entwicklung von Anwendungen, welche die Konsistenz lockern und dafür dauerhafte Verfügbarkeit bieten.

Sogar moderne RDBMS verwenden das Modell der Eventual Consistency. Beispielsweise wird auf replizierte Knoten zugegriffen, um die Lesezugriffe zu optimieren. Die Knoten werden asynchron repliziert, sonst würde eine Transaktion zu lange dauern (es müssten erst alle Knoten repliziert werden, bis die Transaktion abgeschlossen ist). Dabei gibt es ein gewisses Zeitfenster, in dem die Daten auf den zu replizierenden Knoten nicht aktuell sind. Das ist ein typischer Fall für Eventual Consistency. [31]

Eventual Consistency ist nicht für alle Bereiche geeignet. Bei stark voneinander abhängigen Benutzerdaten, also bei voneinander abhängigen Eingaben ist es nicht hinnehmbar, dass die Benutzer veraltete Daten bekommen. Anderenfalls könnten sich die dabei entstehenden Fehler soweit fortpflanzen, bis das gesamte System fehlerhaft ist.

2.5. Multiversion Concurrency Control

Eine der Hauptaufgaben von Datenbanksystemen ist die korrekte Speicherung aller Datensätze, sowie die Sicherstellung ihrer semantischen Integrität. Insbesondere sind Strategien zur Vermeidung von Inkonsistenzen notwendig, wenn mehrere Anwender gleichzeitig auf denselben Datensatz zugreifen und diesen gegebenenfalls auch manipulieren möchten. Die folgende Darstellung beruht auf [16, S. 40ff].

Die klassische Herangehensweise, die vor allem in zentralen Datenbanksystemen und Mainframes zur Anwendung kommt, beruht auf pessimistischen Sperrverfahren. Dabei wird ein Datensatz für einen exklusiven Zugriff gesperrt, verändert und anschließend für Lese- und Schreiboperationen anderer Vorgänge wieder freigegeben.

Diese pessimistischen Sperrverfahren funktionieren problemlos und effizient, solange die zur Erzielung einer Sperre notwendigen Kommunikationskosten relativ gering sind und Sperren

nicht oft vorkommen und nicht lange aufrechterhalten werden, da sonst nebenläufige Lesevorgänge lange verzögert werden.

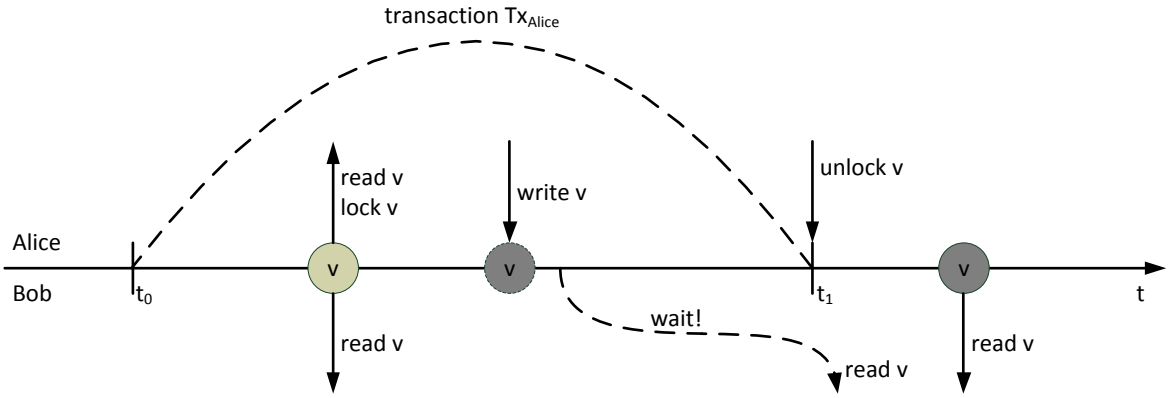


Abbildung 2.9.: Pessimistisches Konkurrenzverfahren mittels Sperren

Im Umfeld des Web 2.0 haben sich die Anwendungsszenarien von Datenbanken grundlegend geändert. Für viele Anwendungen und ihre sehr große Nutzermenge sind lange Lesesperren dabei ein immer größeres Problem. Außerdem sind die Kommunikationskosten in verteilten Datenbank-Clustern sehr teuer und ineffizient. Ein Konsens über eine Sperre ist in einem verteilten System bei Berücksichtigung von Nachrichtenverlusten allein aus mathematischer Sicht nicht zu 100 Prozent zu erzielen.

Eine Lösung für dieses Problem bietet Multiversion Concurrency Control (MVCC). Dabei wird nicht mehr versucht, den Zugriff auf einem zentralen veränderlichen Datensatz zu koordinieren. Stattdessen werden mehrere unveränderliche Datensätze in einer zeitlichen Reihenfolge organisiert (siehe Abbildung 2.10).

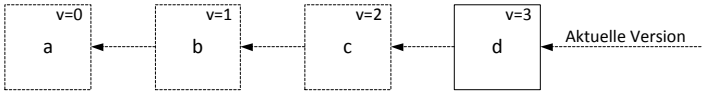


Abbildung 2.10.: Datensatz mit vier Versionen beim MVCC-Verfahren

Durch jeden Schreibvorgang wird eine neue Version des Datensatzes inklusive einer dazugehörigen eindeutigen Identifikationsnummer und einem Verweis auf die zuvor gelesene und damit indirekt veränderte Datensatzversion erzeugt. Da hierbei keine Sperren notwendig sind, werden Leseoperationen nicht mehr durch Schreibvorgänge verzögert, sondern können jederzeit durch eine frühere Version des Datensatzes beantwortet werden (siehe Abbildung 2.11).

Außerdem können auch konkurrierende parallele Schreibzugriffe einfach erkannt und behandelt werden. Beim Schreiben wird die angegebene Vorgängerversion mit der zu diesem Zeitpunkt aktuellen Version verglichen. Sollten sich die beiden Versionen widersprechen,

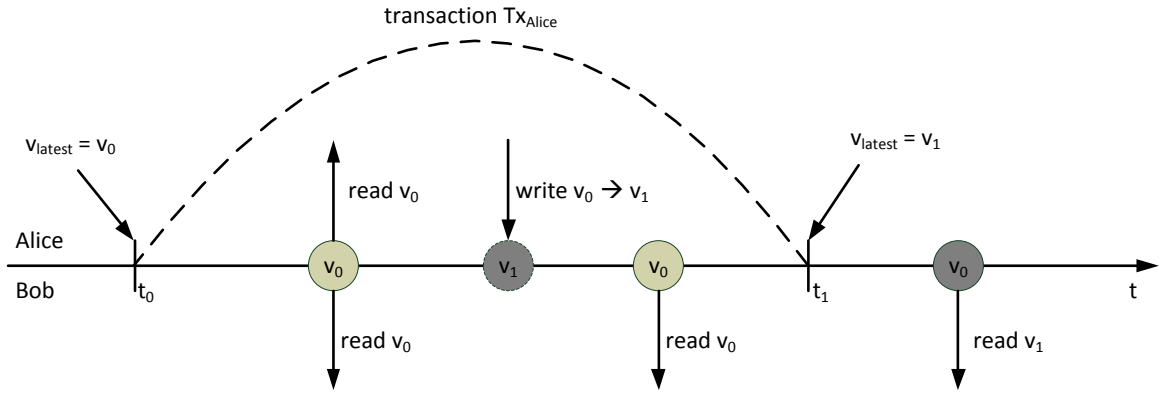


Abbildung 2.11.: Sperren verzögern keine Leseoperationen

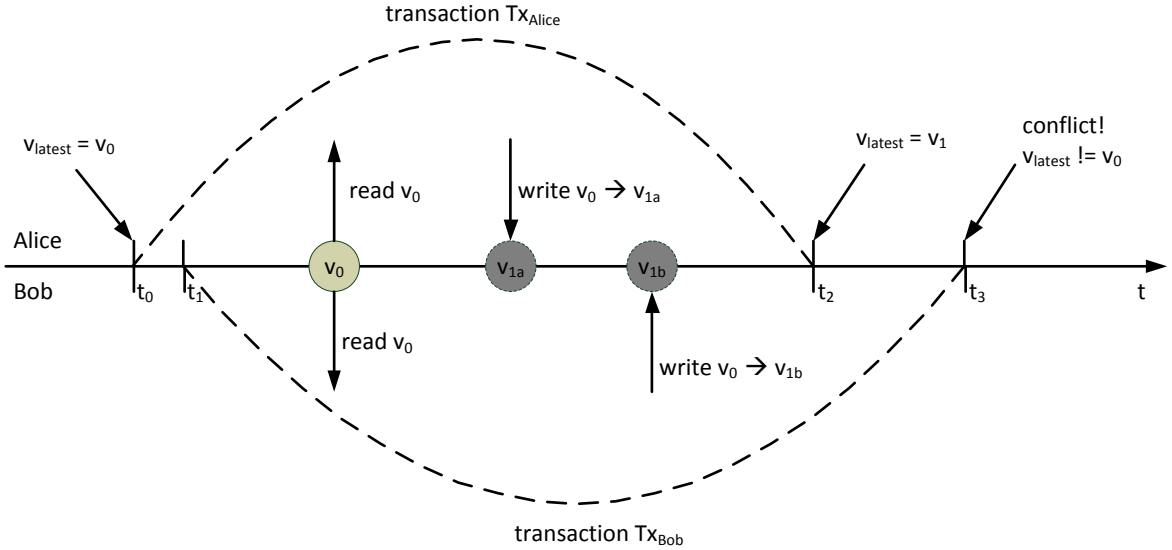


Abbildung 2.12.: Konflikterkennung beim MVCC-Verfahren

kann die gesamte Transaktion zurückgerollt und falls nötig noch einmal gestartet werden (siehe Abbildung 2.12).

Die eindeutige Identifikationsnummer des MVCC-Verfahrens ist dabei nicht festgelegt. Sie kann sich aus den unterschiedlichen zur Verfügung stehenden Informationen zusammensetzen. Beispiele für die eindeutige Identifikationsnummer sind:

- Laufende Nummer aller Transaktionen
- Startzeitpunkt der Transaktion
- Zeitpunkt des aktuellen Schreibvorgangs

Hierdurch wird mehr Speicherplatz und Rechenkapazität benötigt als für ein Verfahren mit Sperren und manipulierbaren Datensätzen. Außerdem muss die Datenbank in regelmäßigen Abständen alte und von keiner Transaktion mehr benötigte Datensatzversionen dauerhaft aus dem System entfernen. Allerdings relativieren sich die aufgezählten Nachteile recht schnell. Ein ähnliches Änderungsprotokollierungsverfahren in Form von Journalen wird von jeder Datenbank genutzt, um unerwarteten Hardwareausfällen oder Programmabstürzen vorzubeugen. Dieses Verfahren muss nur noch für das MVCC-Verfahren verwendet werden.

In der Praxis ist das MVCC-Verfahren bei vielen Datenbankanbietern zu finden. Dazu zählen unter anderem:

- Firebird;
- Oracle;
- Postgres;
- Microsoft SQL Server;
- einige MySQL-Varianten;
- Ingres;
- BerkleyDB;
- CouchDB;
- sones.

Außerdem wird das MVCC-Verfahren bei Versionsverwaltungssystemen wie Git und Mercurial, aber auch in funktionalen Programmiersprachen wie Clojure eingesetzt. Dabei gibt es unterschiedliche Implementierungen des MVCC-Verfahrens:

- Nicht transparente Implementationen verbergen die unterschiedlichen Versionen eines Datensatzes vor dem Anwender und stellen lediglich sicher, dass sich das MVCC-Verfahren analog zum Sperrverfahren verhält und somit immer ein serialisierbares Endergebnis erzeugt wird.
- Einige Implementierungen ermöglichen es, alte Versionen der Datensätze zu erkennen und zu nutzen. Hierdurch können »Snapshots« der Datensätze abgerufen werden. Darüber hinaus lassen sich so »Zeitreise«-Anfragen an die Datenbank stellen, die sich auf einen konsistenten Zustand in der Vergangenheit beziehen. Insbesondere langwierige Analysen profitieren hiervon, da diese unabhängig von aktuellen Änderungen der Datensätze ihre Berechnungen ausführen können.

- Implementationen wie beispielsweise CouchDB und Versionsverwaltungssysteme wie Git oder Mercurial lassen auch mehrere konkurrierende Schreibvorgänge zu. Dadurch können zu einem Zeitpunkt mehrere aktuelle Versionen eines Datensatzes vorhanden sein. Dies wird dann als Konflikt erkannt und unter Umständen automatisch durch einen Merge-Prozess behoben. Falls eine automatische Behandlung des Konflikts nicht möglich ist, kann der Anwender immer noch zur manuellen Lösung des vorliegenden Konflikts aufgefordert werden.

Das MVCC-Verfahren ermöglicht somit eine möglichst effiziente Parallelität beim Skalieren auf große Anwenderzahlen, CPU-Kerne, verteilte Rechnerknoten und Datenvolumina. Durch die vollständige Entkopplung der Lesezugriffe von den Schreibzugriffen, ist das Verfahren eines der wichtigsten Programmierkonzepte innerhalb des NoSQL-Umfeldes.

2.6. Aktuelle NoSQL-Systeme

Als aktuelle NoSQL-Datenbanken werden Datenbanken der vier Gruppen *Wide Column Store*, *Document Store*, *Key/Value-Datenbanken* und *Graphendatenbanken* angesehen. [4] Im folgenden werden die vier Gruppen kurz vorgestellt und ihre wichtigsten Repräsentanten genannt.

2.6.1. Spaltenorientierte Datenbanken

Bei relationalen Modellen werden die Attribute in einer Tabelle untereinander (reihenorientiert) gespeichert. Bei spaltenorientierten Datenbanken hingegen werden die Attribute in einer eigenen Tabelle hintereinander (spaltenorientiert) gespeichert. Die physische Datenorganisation ist dabei anders organisiert als bei der relationalen Speichertechnik. So würde ein relationales Datenbankschema mit den Attributen ID, Name und Geburtsjahr reihenorientiert folgendermaßen gespeichert:

```
1, Marty, 1961, 2, George, 1964, 3, Lorraine, 1961
```

In einer spaltenorientierten Datenbank würde die Daten nach ihrer Spalte gruppiert:

```
1, 2, 3, Marty, George, Lorraine, 1961, 1964, 1961
```

Letzteres Verfahren bietet Vorteile bei der Datenanalyse, Datenkompression und beim Caching der Daten. Die Aggregation der Daten ist hier jedoch einer der wichtigsten Vorteile. Daher greifen darauf OLAP- und Date-Warehouse-Umgebunden zurück.

Die Nachteile dieser Technik sind das aufwendigere Suchen und Einfügen von Daten. Außerdem ist das Schreiben und Lesen von Objektstrukturen, also von zusammengehörigen Spaltendaten, aufwendiger, da hierbei viel gesprungen und gesucht werden muss (z.B. alle Daten von Marty).

Spaltenorientierte Datenbanken gibt es bereits seit den 1990er Jahren. Einen echten Aufschwung erlebten diese allerdings erst seit 2000. Zu den leistungsfähigsten Datenbanken dieser Kategorie zählen Sybase IQ, FluidDB, C-Store und MonetDB.

Die in [4] genannten spaltenorientierten Datenbanken (Wide Column Stores) basieren im Kern auf dieser Technik, auch wenn sie etwas abweichen. Solche Datenbanken sind sehr gut skalierbar und typischerweise für sehr große Datenmengen geeignet. [16, S. 53f]

In [4] werden unter anderem folgende Datenbanken als Wide Column Stores bezeichnet:

- Hadoop / HBase;
- Cassandra;
- Hypertable;
- Cloudera;
- Amazon SimpleDB.

2.6.2. Dokumentenorientierte Datenbanken

Bei dokumentenorientierten Datenbanken werden die zu speichernden Informationen nicht in Tabellen mit festem Schema abgelegt, sondern in Dokumenten. Innerhalb eines Dokuments können beliebige Felder definiert werden. Diese bestehen aus einem Schlüssel, denen ein Wert zugeordnet wird. Der Wert kann dabei beliebig sein (eine Zahl, eine Zeichenkette, eine Liste, eine Datei, etc.). Das Format des Wertes muss vorher nicht definiert sein. Da die Dokumente komplett schemalos sind, können einem Dokument beliebige Schlüssel mit beliebigen Werten hinzugefügt werden. Innerhalb eines Dokuments muss der Schlüssel einmalig sein. In anderen Dokumenten kann der Schlüssel erneut enthalten sein, muss es aber nicht.

Dokumentenorientierte Datenbanken eignen sich besonders dann, wenn größere Mengen an Daten unbestimmter Größe zu speichern sind. Beispielsweise bei Blogs, Content Management Systemen oder Wikis. Der Vorteil liegt darin, dass alle Informationen, die zu einem Dokument gehören, innerhalb der Datenbank als ein Dokument gespeichert werden. Also beispielsweise die Meta-Daten einer Wikiseite oder die Metadaten und die von Nutzern hinterlassenen Kommentare eines Blogeintrags.

Die Abfrage erfolgt häufig mit einem MapReduce-Ansatz (siehe 2.3). Abfragen, die den Schlüssel zu einem Wert liefern, sind schwieriger zu realisieren. Sie müssen in der Regel in der Anwendung umgesetzt werden. [25]

Ein weiteres Beispiel für ein Dokument in einer dokumentenorientierten Datenbank wäre eine Visitenkarte. Das dazugehörige Dokument würde in etwa so aussehen:

```
{
   "type": "contact",
   "firstname": "Marty",
   "lastname": "McFly",
   "email": {
       "home": "marty@mcfly.com",
       "school": "marty.mcfly@hill-valley-high.com"
   },
   "phone": "555-587843"
}
```

Zu den dokumentenorientierten Datenbanken zählen nach [4] unter anderem folgende Systeme:

- CouchDB;
- MongoDB;
- Terrastore;
- ThruDB;
- OrientDB;
- RavenDB.

Dabei gehören CouchDB und MongoDB zu den verbreitetsten Vertretern dokumentenorientierter Datenbanken. [25]

2.6.3. Key/Value-Datenbanken

Key/Value-Datenbanken sind vom Prinzip her noch einfacher als dokumentenorientierte Datenbanken. Hier werden nur Schlüssel-Wert-Paare abgespeichert. Der Schlüssel muss immer einzigartig sein. Die zulässigen Formate der Werte hängen vom jeweiligen System ab. Zeichenketten werden in der Regel immer unterstützt, teilweise aber auch binäre Daten, Listen und Sets.

Die Key/Value-Datenbanken sind sehr schnell. Auf normaler Hardware sind mehr als 100000 an Schreib-/Leseoperationen möglich. Key/Value-Datenbanken haben aufgrund ihrer sehr einfachen Struktur einen sehr geringen Overhead beim Zugriff auf die Datenbank. Bei Schreiboperationen wird im günstigsten Fall nur geprüft, ob der angegebene Schlüssel bereits vorhanden ist und anschließend wird das neue Key/Value-Paar geschrieben. [25]

Key/Value-Datenbanken eignen sich besonders gut für große Datenmengen. Key/Value-Strukturen lassen sich aufgrund ihrer Unabhängigkeit zueinander viel einfacher skalieren, als dies bei relational miteinander verbundenen Daten der Fall ist. [16, S. 131]

Als Key/Value-Datenbanken werden in [4] unter anderem folgende Systeme angesehen:

- Redis;
- Chordless;
- Riak;
- MEMBASE;
- Tokyo Cabinet / Tyrant.

2.6.4. Graphendatenbanken

Mit Graphendatenbanken lassen sich Beziehungen zwischen einzelnen Objekten modellieren. Sie sind spezialisiert auf vernetzte Informationen und deren möglichst effiziente, indexfreie Traversierung. Es gibt zahlreiche Anwendungen, die sich ganz natürlich auf Graphen zurückführen lassen und somit für den Einsatz graphenorientierter Datenbanken geeignet sind. Unter anderem sind dies:

- Hyperlink-Struktur des Word Wide Web;
- Bedeutung von Seiten für Suchmaschinen (Page-Rank);
- Wer-kennt-wen in sozialen Netzwerken (Kürzeste Wege);
- Fahr-/Flugplanoptimierung (Maximaler Fluss);
- Straßenbau, Infrastrukturmaßnahmen (Minimaler Spannbaum).

Relationale Datenbanken stoßen trotz etwaiger proprietärer SQL-Erweiterungen, SQL-Design-Patterns wie z.B. dem Entity-Attribute-Value-Model, zusätzlicher Softwarekomponenten wie objektrelationale Mapper, zusätzlicher externer Query-/Result-Caches wie Memcached oder manueller Partitionierungsalgorithmen auf Probleme. Beispielsweise entstehen fehlende (relationale) Konsistenzkriterien bei partitionierten Datensätzen oder eine erschwerte Skalierbarkeit der Anwendungen, da zusätzliche Softwarekomponenten wie z.B. objektrelationale Mapper auf die Performanz einer Anwendung in hohen Lastsituationen und bei einem schnell zunehmenden Umfang der Nutzerdaten nur schwer abzuschätzen sind. Daher erfreuen sich graphenorientierte Datenbanken zunehmender Beliebtheit. [16, S. 170f]

3. Analyse

In diesem Kapitel werden die Anforderungen an die zu entwickelnde Anwendung analysiert. Diese unterteilen sich in funktionale und nichtfunktionale Anforderungen. Dabei wird zunächst anhand eines Szenarios der Einsatz der Anwendung in einem typischen Umfeld des Bewerbermanagements beschrieben.

3.1. Szenario

Es wird ein Bewerbermanagementsystem betrachtet, welches in Unternehmen eingesetzt wird, um Ausschreibungen und Bewerbungen zu verwalten.

Ein Unternehmen erstellt eine Ausschreibung. Diese enthält alle relevanten Informationen für die zu besetzende Stelle wie beispielsweise den gewünschten Eintrittstermin, die Aufgaben und Anforderungen an den Bewerber sowie die Kontaktdaten des Referenten der Ausschreibung. Anschließend wird die Ausschreibung durch den Anwendungsbenutzer (nachfolgend auch Personalbenutzer genannt) aktiviert und ist für potentielle Bewerber einsehbar.

Bewerber können sich auf die Ausschreibung über ein Online-Formular direkt bewerben. Dabei gibt ein Bewerber neben seinen Kontaktdaten auch Informationen bezüglich seiner Ausbildungs- und Berufserfahrung in das Online-Formular ein. Er hat zusätzlich die Möglichkeit, einen Lebenslauf und ein Anschreiben in Form eines PDF-Dokuments sowie ein Passfoto in Form einer JPEG-Datei hochzuladen. Nach erfolgreichem Ausfüllen des Online-Formulars sendet der Bewerber die Bewerbung ab.

Die neu eingegangenen Bewerbungen sind in der Bewerbungsübersicht der Anwendung aufgelistet und können durch den Anwendungsbenutzer eingesehen und bearbeitet werden. Falls die Netzwerkverbindung ausfällt, hat der Anwendungsbenutzer dennoch die Möglichkeit, die bis zum Ausfall der Netzwerkverbindungen eingegangenen Bewerbungen einzusehen und zu bearbeiten.

Nach Durchsicht der eingegangenen Bewerbungen für die erstellte Ausschreibung möchte das Unternehmen zwei Bewerber (Bewerber A und Bewerber B) näher kennenlernen und lädt diese zu einem Bewerbergespräch ein. Dabei setzt der Anwendungsbenutzer die Bewerbungen auf einen neuen Status »Eingeladen zum Erstgespräch«. Die übrigen Bewerbungen, die auf die Ausschreibung eingegangen sind, sind für das Unternehmen nicht interessant. Daher werden die Bewerbungen auf einen neuen Status »Abgelehnt« gesetzt.

Nachdem beide Bewerbungsgespräche gelaufen sind, entscheidet sich das Unternehmen für Bewerber B. Bewerber A wird auf den Status »Abgelehnt« gesetzt und Bewerber B auf den Status »Eingestellt«. Die Ausschreibung wird durch den Personalbenutzer anschließend manuell deaktiviert und ist nicht mehr für mögliche Bewerber einsehbar.

3.2. Anforderungsanalyse

Um einen möglichst genauen Überblick der Anforderungen an das System zu bekommen, wird eine Anforderungsanalyse durchgeführt. Hierfür wird das in Abschnitt 3.1 dargestellte Szenario herangezogen. Es wird zwischen zwei Komponenten des Systems unterschieden:

- *Komponente 1 (interne Komponente)*
  bietet alle Funktionen für authentifizierte Benutzer, um Ausschreibungen und Bewerbungen zu verwalten.
- *Komponente 2 (externe Komponente)*
  stellt eine Jobbörse dar, auf die Bewerber zugreifen und Bewerbungen abschicken.

3.2.1. Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die Funktionalität oder die Dienste, die von einem System erwartet werden. Diese Anforderungen hängen von der Funktionalität und den Eigenschaften der Anwendung, der Art des zu entwickelnden Systems und den Benutzern ab, die später mit der Anwendung arbeiten. [26]

Daher wird die Analyse der funktionalen Anforderungen mit der Untersuchung der Anwendungsfälle begonnen, um so die genauen Anforderungen an das System festzulegen.

Anwendungsfälle Personalbenutzer

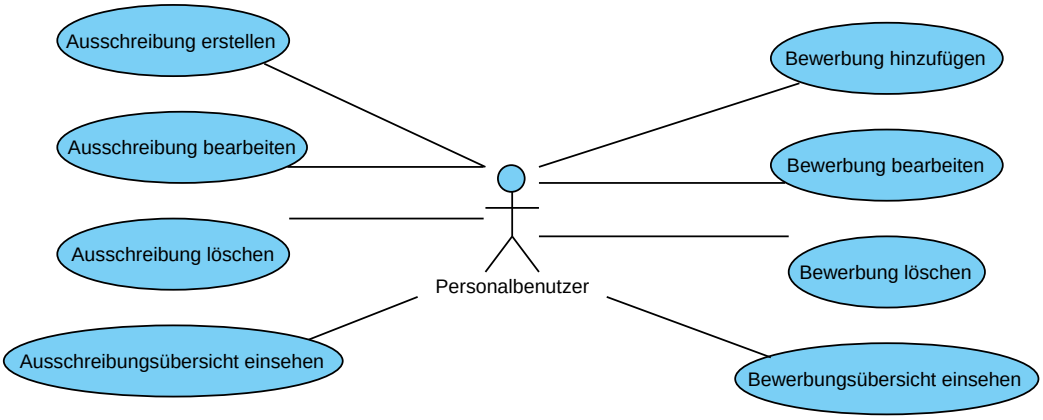


Abbildung 3.1.: Use-Case-Diagramm für die Ausschreibungs- und Bewerbungsverwaltung

Ausschreibungen verwalten Der Personalbenutzer muss eine beliebige Anzahl an Ausschreibungen erstellen können. Nach dem Erstellen einer Ausschreibung ist diese zunächst deaktiviert (**FA101**).

Die Ausschreibungen lassen sich jederzeit durch den Personalbenutzer abändern. Die einzelnen Felder der Ausschreibung lassen sich dabei ändern. Die Ausschreibung lässt sich durch den Personalbenutzer aktivieren bzw. deaktivieren (**FA102**).

Wird eine Ausschreibung nicht mehr benötigt, kann der Personalbenutzer diese löschen. Das Löschen einer Ausschreibung ist nur dann möglich, wenn sich auf ihr keine Bewerbungen befinden (**FA103**).

Es soll eine Ausschreibungsübersicht geben, in der alle Ausschreibungen in einer Liste zusammengefasst werden, um dem Personalbenutzer so einen Überblick über die im System vorhandenen Ausschreibungen zu verschaffen. Die Ausschreibungsliste soll dabei neben dem Titel der Ausschreibung auch ihren Status sowie einen Link zu allen Bewerbungen der Ausschreibung enthalten (**FA104**).

Bewerbungen verwalten Analog zu den Ausschreibungen lassen sich ebenfalls Bewerbungen durch den Personalbenutzer verwalten. Der Personalbenutzer kann eine beliebige Anzahl an Bewerbungen hinzufügen. Dabei muss jede Bewerbung genau einer Ausschreibung zugeordnet sein (**FA201**).

Existieren bereits Bewerbungen im System, lassen sich diese ebenfalls durch den Personalbenutzer abändern. Dabei können alle Felder der Bewerbung sowie ihr Status geändert

werden (**FA202**). Ebenfalls können Bewerbungen durch den Personalbenutzer gelöscht werden (**FA203**).

In der Bewerbungsübersicht werden alle im System vorhandenen Bewerbungen in Form einer Liste aufgeführt. Die Liste enthält neben dem Vor- und Nachnamen der Bewerbung ebenfalls ihren Status und den Titel der Ausschreibung, auf der sich die Bewerbung befindet. Die Liste lässt sich nach dem Titel der Ausschreibung filtern (**FA204**).

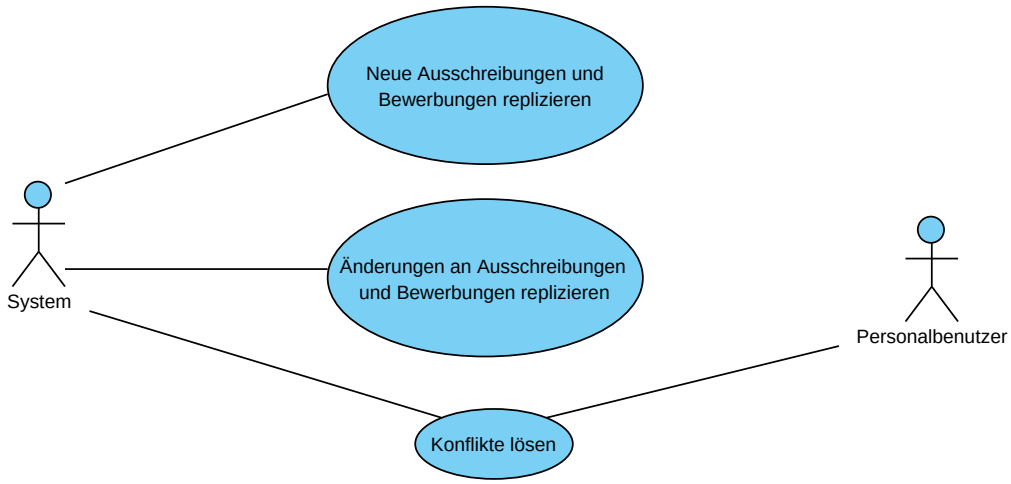


Abbildung 3.2.: Use-Case-Diagramm für die Replikation

Replikation Ist der Personalbenutzer online oder wird nach einer Offline-Phase die Verbindung wieder hergestellt, müssen von ihm gemachte Änderungen an Ausschreibungen oder Bewerbungen sofort zum Server repliziert werden (**FA301**).

Alle von anderen Benutzern getätigten Änderungen an Ausschreibungen und Bewerbungen müssen automatisch auf das Endgerät des Personalbenutzers repliziert werden, sofern oder sobald diese mit dem Server verbunden sind (**FA302**).

Bei der Replikation auftretende Konflikte sollen durch den Personalbenutzer gelöst werden können (**FA303**).

Anwendungsfälle Bewerber

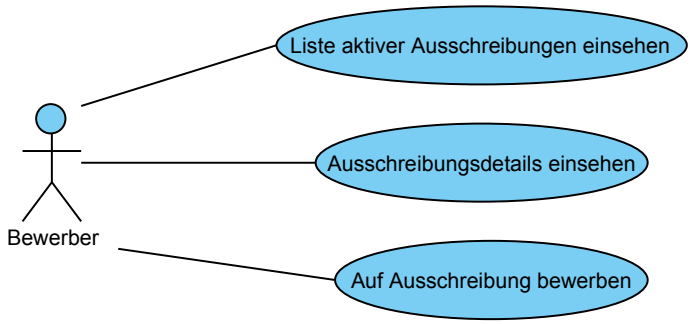


Abbildung 3.3.: Use-Case-Diagramm für den Bewerber

Aktive Ausschreibungen einsehen Der Bewerber hat die Möglichkeit, alle aktiven Ausschreibungen einzusehen. Zunächst werden dem Bewerber die aktiven Ausschreibungen in Form einer Liste angezeigt. Die Liste enthält den Titel der Ausschreibung, der gleichzeitig ein Link zu den Ausschreibungsdetails ist (**FA401**). In den Ausschreibungsdetails werden dem Bewerber die restlichen Informationen der Ausschreibung angezeigt (**FA402**).

Auf Ausschreibung bewerben Dem Bewerber steht beim Aufruf der Ausschreibungsdetails ein Link zur Verfügung, der zum Bewerbungsformular der Ausschreibung führt. Füllt der Bewerber das Bewerbungsformular aus und schickt es ab, geht seine Bewerbung ins System ein (**FA403**).

3.2.2. Nichtfunktionale Anforderungen

Während funktionale Anforderungen die speziellen Funktionen einen Systems beschreiben, beziehen sich nichtfunktionale Anforderungen eher auf das System im Ganzen. Daher kann eine nichtfunktionale Anforderung wichtiger sein als einzelne funktionale Anforderung. [26, S. 111]

> »Während das Nichteinhalten einer einzelnen funktionalen Anforderung das System schlechter werden lässt, kann das Ignorieren einer nichtfunktionalen Anforderung das gesamte System unbrauchbar machen.« [26, S. 111]

Für nichtfunktionale Anforderungen sind nach [15] folgende Ziele anzustreben:

- Vollständige Menge nichtfunktionaler Anforderungen.

- Messbare nichtfunktionale Anforderungen. Dadurch kann der Einsatz von Metriken erfolgen.
- Fokussierung des Aufwands. Dadurch wird eine Priorisierung der nichtfunktionalen Anforderungen ermöglicht.
- Konfliktfreie nichtfunktionale Anforderungen.

> »Qualitätsmodelle wie ISO9126 können eine entscheidende Unterstützung bei der effizienten Erhebung und Spezifikation einer vollständigen und konsistenten Menge von messbaren NFAs[1] bieten.« [15]

Nach [15] können Qualitätsmodelle wie ISO/IEC 9126 einen entscheidende Unterstützung bei der Erhebung und Spezifikation einer vollständigen und konsistenten Menge von messbaren nichtfunktionalen Anforderungen bieten. Daher wird in dieser Arbeit das Qualitätsmodell ISO/IEC 9126 angewendet, um so eine möglichst vollständige Menge der nichtfunktionalen Anforderungen zu ermitteln. In der Norm werden sechs Qualitätsmerkmale für Softwareprodukte spezifiziert:

- Funktionalität;
- Zuverlässigkeit;
- Benutzbarkeit;
- Effizienz;
- Wartbarkeit;
- Portabilität.

Jedes dieser Merkmale wird in Untermerkmale eingeteilt. Die Untermerkmale werden durch entsprechende interne und externe Qualitätsattribute beschrieben, die durch Metriken messbar sind. [8]

Die genannten Qualitätskriterien für Softwareprodukte und ihre Untermerkmale werden zunächst nach ihrer Wichtigkeit für die Anwendung in Tabelle 3.1 gewichtet und anschließend diskutiert und ggf. ein durch Metriken nachweisbarer Sollwert festgelegt. Nicht relevante Untermerkmale werden nicht diskutiert.

[1] Nichtfunktionale Anforderungen

Produktqualität	sehr wichtig	wichtig	normal	nicht relevant
Funktionalität				
Angemessenheit		X		
Richtigkeit	X			
Interoperabilität	X			
Ordnungsmäßigkeit				X
Sicherheit	X			
Zuverlässigkeit				
Reife		X		
Fehlertoleranz			X	
Wiederherstellbarkeit			X	
Benutzbarkeit				
Verständlichkeit	X			
Erlernbarkeit	X			
Bedienbarkeit	X			
Attraktivität			X	
Konformität				X
Effizienz				
Zeitverhalten		X		
Verbrauchsverhalten			X	
Konformität				X
Änderbarkeit				
Analysierbarkeit		X		
Modifizierbarkeit		X		
Stabilität	X			
Testbarkeit		X		
Übertragbarkeit				
Anpassungsfähigkeit	X			
Installierbarkeit		X		
Koexistenz				X
Austauschbarkeit				X
Konformität				X

Tabelle 3.1.: Gewichtung der nichtfunktionalen Anforderungen

Funktionalität

Dieses Merkmal bestimmt, wie vollständig die unter 3.2.1 geforderten Funktionen in der Anwendung enthalten sind und die definierten Anforderungen erfüllt werden.

Angemessenheit Die Funktionen der Anwendung sollen sich aus aufgabenorientierten Teilfunktionen zusammensetzen. Eine Teilfunktion wäre beispielsweise das Hinzufügen einer Ausschreibung, während die Funktion das Verwalten der Ausschreibungen ist.

Richtigkeit Die Korrektheit der durch das System ausgegebenen Daten ist als essentiell einzustufen. Beispielsweise würden falsche Bewerberdaten unter Umständen dazu führen, dass der Bewerber fälschlicherweise abgelehnt würde. Falsche Daten in der Ausschreibung könnten dazu führen, dass der potentielle Bewerber sich nicht auf die Ausschreibung bewirbt. Als Testmetrik dient der Anteil falsch ausgegebener Daten in Bezug auf die Anfragen an die Webservice-API. Der Anteil falsch gelieferter Daten des Systems soll unter 1% liegen.

Interoperabilität Eines der Hauptziele dieser Arbeit ist die Lauffähigkeit auf unterschiedlichen Geräten, auf denen ein Browser ausgeführt wird. Daher soll das System auf aktuellen Browsern lauffähig sein. Dazu zählen mindestens folgende Browser:

- Mozilla Firefox (ab Version 4);
- Opera (ab Version 11);
- Internet Explorer (ab Version 8);
- Safari (ab Version 5);
- Chrome (ab Version 10).

Die Anwendung soll auch auf mobilen Endgeräten lauffähig sein, wenn auf diesen ein Javascriptfähiger Browser installiert ist.

Sicherheit Die externe Komponente erfordert keine Authentifizierung seitens des Bewerbers. Der Bewerber greift lesend auf die Datenbank zu und schickt Bewerbungen ab. In der Datenbank befinden sich nur Ausschreibungen, die von jedem eingesehen werden dürfen. Die interne Komponente erfordert eine Authentifizierung des Benutzers am System, da hier personenbezogene Daten verarbeitet werden und der Zugriff auf Bewerberdaten nur definierten Benutzern zugänglich sein soll.

Zuverlässigkeit

Die Zuverlässigkeit beschreibt die Fähigkeit der Anwendung, Leistung unter definierten Bedingungen zu erbringen. Zur Feststellung der Zuverlässigkeit eignet sich als Messgröße beispielsweise das Verhältnis der Ausfallzeiten in Relation zum gesamten Nutzungszeitraum.

Reife Lesefehler können dazu führen, dass dem Benutzer falsche Informationen präsentiert und auf Grundlage falscher Informationen nicht gewünschte Entscheidungen getroffen werden. Zum Beispiel würde ein Bewerber fälschlicherweise abgelehnt werden oder ein Bewerber würde sich nicht auf eine Ausschreibung bewerben, wenn die Ausschreibungsinformationen fehlerhaft wären. Schreibfehler können den gesamten Datenbestand beschädigen. Daher sind sowohl Schreib- als auch Lesefehler grundsätzlich zu vermeiden.

Fehlertoleranz Fehler sind nach Möglichkeit immer abzufangen. Dies schließt Bedienfehler (etwa durch falsche Eingaben) und Softwarefehler (zum Beispiel durch einen Verbindungsabbruch) ein.

Wiederherstellbarkeit Falls die Anwendung nicht mehr auf Benutzereingaben reagiert, soll ein Neustart wieder zum korrekten Verhalten der Anwendung führen. Der Neustart sollte durch den Neustart des Browsers durch den Benutzer vollzogen sein. Die direkt vor dem Zeitpunkt des Absturzes eingegebenen Benutzerdaten sind nicht zwangsweise zu rekonstruieren. Der Benutzer muss bei einem Neustart damit rechnen, die Daten erneut einzugeben.

Benutzbarkeit

Die Benutzbarkeit betrachtet den Aufwand zur Benutzung der Anwendung. Als Benutzergruppen sind für die interne Komponente Benutzer mit entsprechenden Kenntnissen im Bereich Personalmanagement anzunehmen. Für die externe Komponente sind Benutzer anzunehmen, die einen gängigen Browser bedienen können.

Verständlichkeit Beide Komponenten sollen klar strukturiert und nicht mit Funktionen überladen sein. Anhand der Namen der einzelnen Funktionen soll der Benutzer die Aufgabe, die man mit der Funktion erledigen kann, sofort erkennen.

Erlernbarkeit Für die externe Komponente soll kein Schulungsaufwand erforderlich sein. Der Bewerber soll auf den ersten Blick erkennen können, wie er die Details einer Ausschreibung aufrufen und sich auf eine Ausschreibung bewerben kann. Für die interne Komponente soll der Schulungsaufwand möglichst gering gehalten werden. Benutzer mit Kenntnissen im Bereich Personalmanagement sollen nach einer maximal einstündigen Schulung die Software bedienen können.

Bedienbarkeit Die interne Komponente soll eine Bedienoberfläche besitzen, die nur durch das Fachwissen im Bereich des Personalmanagements durch den Benutzer ohne weitere Erklärung verstanden werden kann. Daher müssen alle Eingabemöglichkeiten eine gute Selbstbeschreibungsfähigkeit besitzen. Vor dem Löschen sämtlicher Daten muss der Benutzer grundsätzlich einen Dialog bestätigen. Dieser soll wie sämtliche Fehlermeldungen in verständlicher Sprache formuliert sein.

Attraktivität Die Anwendung soll sich durch eine klare Struktur auszeichnen und so auf den Anwender attraktiv wirken.

Effizienz

Die Effizienz wird durch das Verhältnis aus Leistung der Software und den eingesetzten Betriebsmitteln ermittelt.

Zeitverhalten Sowohl bei der internen als auch bei der externen Komponente dürfen Antworten auf Anfragen nicht länger als zwei Sekunden dauern. Nur die Anfragen nach größeren Dokumenten wie beispielsweise Bilddateien oder PDF-Dateien dürfen aufgrund der Datengröße länger als zwei Sekunden dauern. Dabei darf die Anwendung für den Benutzer jedoch nicht blockiert werden.

Verbrauchsverhalten Um mit dem Speicherplatz möglichst sparsam umzugehen, dürfen die in dem Bewerbungsformular hochgeladenen Dokumente eine Dateigröße von jeweils 1MB nicht überschreiten. Die Dateien sind daher vor dem Hochladen vom Benutzer ggf. zu verkleinern.

Änderbarkeit

Die Änderbarkeit beschreibt den Aufwand, der zur Durchführung vorgegebener Änderungen an der Software notwendig ist.

Analysierbarkeit/Modifizierbarkeit Der Aufwand, um in der Software Ansatzpunkte für Änderungen zu finden oder Änderungen durchzuführen, soll durch übersichtlich und gut strukturierten Quellcode minimiert werden.

Stabilität Die Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen soll unter 1% liegen.

Testbarkeit Zur Überprüfung der Leistungsfähigkeit der Anwendung muss ein Belastungstest ausgeführt werden. Der Belastungstest muss auf jedem System ausführbar sein, auf dem die Anwendung installiert ist. Die Liste der Anwendungsfälle dient als Grundlage zur Entwicklung der Testfälle. Für jeden Anwendungsfall muss mindestens ein Test erstellt werden.

Übertragbarkeit

Die Übertragbarkeit beschreibt den Aufwand, der notwendig ist, um den Einsatz der Anwendung in einer anderen Systemumgebung zu ermöglichen.

Anpassungsfähigkeit Es ist wichtig zwischen der internen und externen Komponente zu unterscheiden. Die externe Komponente ist für Bewerber nur auf dem Server erreichbar. Sie muss nicht beim Bewerber auf dem Endgerät installiert werden. Da Benutzer der internen Komponente die Möglichkeit haben auch offline weiterzuarbeiten, muss die interne Komponente auf möglichst vielen Betriebssystemen lauffähig sein. Die Anwendung soll auf jedem System lauffähig sein, auf dem eine Instanz von CouchDB und ein o.g. Browser installiert ist.

Installierbarkeit Die Installation der internen Komponente sollte einfach und ohne tiefgehende Kenntnisse des Gerätes erfolgen können.

3.3. Eigenes Vorgehen

Der in dieser Arbeit angewandte Softwareentwicklungsprozess lässt sich teilweise mit *Scrum* [20] beschreiben. Scrum verfolgt die drei Ziele

- Transparenz,
- Überprüfung,
- und Anpassung.

So wird der Fortschritt der Entwicklung bewertet, es werden in regelmäßigen Abständen Funktionalitäten der Software geliefert und die Anforderungen an die Software werden nach jeder Lieferung von Funktionalitäten neu bewertet und bei Bedarf ergänzt.

Die Anforderungen an die Software wurden anhand eines Szenarios definiert und nach jeder Weiterentwicklung der Software neu bewertet und überarbeitet. Das Szenario wurde ebenfalls angepasst. Die Software wurde in mehreren Schritten entwickelt, wobei das Ergebnis jeder Schritt eine neue oder überarbeitete Funktionalität lieferte.

4. Entwurf

In diesem Kapitel erfolgt die Umsetzung der Anforderungen, die in der Analyse für die Anwendung festgelegt wurden. Zunächst wird ein Überblick über die Technologien gegeben, die für die Anwendung verwendet werden sollen. Anschließend erfolgt ein Überblick über die Systemarchitektur und ihre einzelnen Komponenten. Zum Schluss des Kapitels wird auf das Testen der Anwendung eingegangen.

4.1. Eingesetzte Technologien

Als Datenspeicher für die Anwendung soll eine NoSQL-Datenbank zum Einsatz kommen. Wie bereits in Kapitel 1.2 angeführt, soll es sich dabei um eine dokumentenorientierte Datenbank handeln. In dieser Arbeit wurde CouchDB als NoSQL-Datenbank ausgewählt. Im Folgenden soll kurz auf CouchDB eingegangen und die Beweggründe für die Auswahl von CouchDB in der Zusammenfassung dargelegt werden. Soweit nicht anders angegeben, basiert die Darstellung von CouchDB auf [16, S. 102ff].

4.1.1. CouchDB

CouchDB wird seit 2005 von dem ehemaligen Senior-Entwickler von Lotus Notes, Damien Katz entwickelt. Dabei möchte CouchDB den wachsenden Anforderungen des Web 2.0 gerecht werden. CouchDB rechnet damit, dass nicht immer eine Netzwerkverbindung besteht und dass Fehler in verteilten Systemen vorkommen können. Daher steht der Name CouchDB für *Cluster of unreliable commodity hardware Data Base*. CouchDB orientiert sich an Googles BigTable. Seit November 2008 ist CouchDB ein vollwertiges Projekt der Apache Software Foundation. Entwickelt wurde CouchDB auf der Plattform Erlang/OTP[1]. Diese Plattform umfasst die Sprache, das Laufzeitsystem und eine umfangreiche Bibliothek. Da Erlang eine funktionale, nebenläufige und verteilte Programmiersprache ist, bietet sie ideale Voraussetzungen zur Entwicklung der verteilten und auf Nebenläufigkeit ausgelegten Datenbank CouchDB.

[1] http://www.erlang.org/doc/

Beschreibung

Der Zugriff auf die Daten erfolgt in CouchDB mittels einer RESTful[2] JSON[3] API. Gefilterte Abfragen auf die Datenbank werden mittels JavaScript über das Map/Reduce-Framework getätigt. Ferner unterstützt CouchDB das Replizieren der Daten auf mehrere Knoten. So lassen sich beispielsweise Lesevorgänge parallelisieren. Da CouchDB nach dem Prinzip des MVCC (siehe 2.5) arbeitet, gibt es kein Locking.

Datenmodell

Wie in jeder schemafreien, dokumentenorientierten Datenbank können in CouchDB Dokumente beliebiger Syntax abgelegt werden. Die Dokumente werden dabei in JSON-Datenstrukturen abgelegt und in B-Bäumen gespeichert. Dabei erhalten sie zur Indexierung eine Dokument-ID und eine Revisions-ID. Bei jedem Update eines Dokuments wird eine neue Revisions-ID generiert, welche später ein inkrementelles Auffinden der Änderungen ermöglicht.

View-Modell

Das View-Modell ermöglicht die Aggregation und Darstellung der Dokumente in einer Datenbank. Views werden bei Bedarf dynamisch erzeugt. Sie haben keinen Einfluss auf die zugrunde liegenden Dokumente. Von den gleichen Daten können unterschiedliche Views erstellt werden. Views werden in speziellen Dokumenten definiert, in sogenannten *Design-Dokumenten*. Views werden mittels JavaScript-Funktionen und Map/Reduce erstellt. Dabei übernimmt die JavaScript-Funktion die Rolle der Map- bzw. Reduce-Funktion. Für jedes Dokument wird die Funktion aufgerufen und die in der View geforderten Daten aggregiert.

```
1 function(doc) {
2   if(doc.type == 'account') {
3     emit(doc.name, doc.account_nr);
4   }
5 };
```

Listing 4.1: Map-Funktion eines Views

Das Listing 4.1 zeigt eine View, welche die Felder `name` und `account_nr` aller Dokumente zurückgibt, für die gilt `doc.type == 'account'`.

[2] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[3] http://www.json.org/

Zugriffskontrolle

Die Zugriffskontrolle ist in CouchDB einfach geregelt. Werden keine besonderen Einstellungen vorgenommen, ist eine Datenbank komplett frei zugänglich. Über Validierungsfunktionen lässt sich das Hinzufügen und Verändern von Dokumenten auf definierte Benutzer oder Benutzerrollen einschränken. Soll die Datenbank nicht frei zugänglich sein, lässt sich das dadurch erreichen, dass definierte *Reader* der Datenbank zugeordnet werden. Dann lässt sich die Datenbank nur nach vorherigem Login öffnen. Darüber hinaus lassen sich für jede Datenbank *Admins* festlegen. Sie können die gesamte Datenbank einsehen und weitere Datenbankbenutzer anlegen.

Authentifizierung

CouchDB verfügt aktuell über drei Authentifizierungsmodelle [22]:

- Basic Auth: Ein Authentifikationsmechanismus für HTTP, der die Benutzerinformationen im Header einer Anfrage Base64 codiert und an den Server sendet. Die Base64-Codierung ist dabei als unsicher anzusehen, da praktisch jeder, der den Netzwerkverkehr beobachtet die Zugangsdaten decodieren kann. Daher sollte Basic Auth nur in geschützten Bereichen verwendet werden (LAN, VPN, SSL).
- Secure Cookie Auth: Im Gegensatz zu Basic Auth erfolgt bei Secure Cookie Auth der Transport der Benutzerinformationen HMAC-verschlüsselt[4]. Secure Cookie Auth kann also benutzt werden, um eine Authentifizierung über eine unsichere Verbindung durchzuführen.
- OAuth[5]: Ermöglicht es Benutzern, dass eine Anwendung sich wie der Benutzer an einem Dienst anmeldet.

CouchApps

In CouchDB besteht die Möglichkeit, sogenannte Design-Dokumente anzulegen. Design-Dokumente ermöglichen die Erstellung eigenständiger Anwendungen mittels JavaScript. Diese Anwendungen, die von einer Standard-CouchDB-Instanz bereitgestellt werden, werden als CouchApps bezeichnet. Für die Lauffähigkeit von CouchApps ist also nichts weiter erforderlich als eine CouchDB-Instanz und JavaScript. Die Design-Dokumente können wie die übrigen Dokumente über den Replikationsmechanismus ausgetauscht werden. Somit ermöglicht CouchDB die Bereitstellung ganzer Webanwendungen mit dokumentenbasierter Datenhaltung.

[4] http://tools.ietf.org/html/rfc2104
[5] http://oauth.net/

Replikation

Replikation auf mehreren Knoten wird von CouchDB unterstützt. Die Daten werden inkrementell mit bidirektionaler Konflikterkennung und bidirektionalem Konfliktmanagement repliziert. Das Replizieren wird von der Anwendung ausgelöst oder erfolgt kontinuierlich. Die Replikation kann auch gefiltert erfolgen. Dabei werden dann nur die gewünschten Dokumente repliziert. Die Verteiltheit erlangt CouchDB über Replikation zwischen den einzelnen Knoten. Daher kann man CouchDB als *peer-based distributed* bezeichnen. Nach der Replikation kann jede Datenbank unabhängig von der anderen arbeiten und es entsteht so kein *Single Point of Failure* (siehe Abbildung 4.1). Alte Versionen von Dokumenten werden nicht repliziert. Eine CouchDB-Instanz geht davon aus, dass sie *offline by default* ist. Sobald wieder eine Netzwerkverbindung vorhanden ist, wird die Synchronisation in einer vorher definierten Weise durchgeführt.

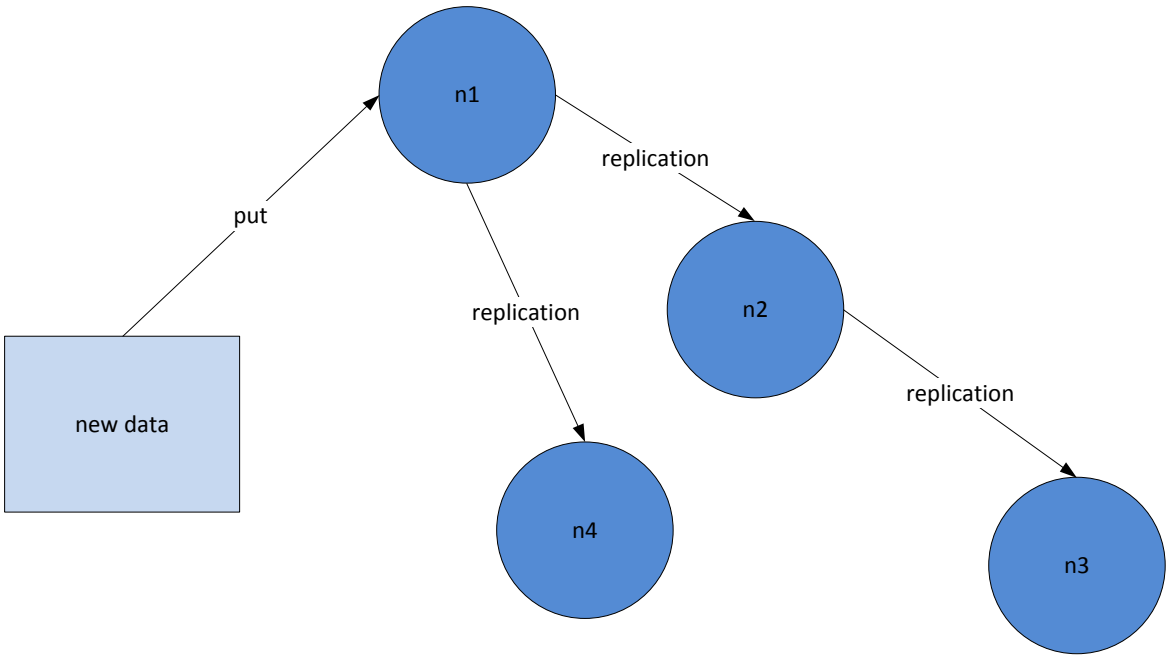


Abbildung 4.1.: Inkrementelle Replikation zwischen mehreren CouchDB-Knoten [9, S. 17]

Konflikte

Beim Replizieren können Konflikte entstehen. CouchDB sorgt mittels eines deterministischen Algorithmus dafür, dass es immer einen Datensatz gibt, der »gewinnt« und in Views zur Verfügung steht. Allerdings werden die Dokumente als Konflikt markiert, indem `doc._conflicts` gesetzt wird. Dabei wird `doc._conflicts` ein Array mit den in Konflikt zueinander stehenden Revisionen zugeordnet. Nun kann die Anwendung entscheiden, wie mit

den Konflikten umgegangen wird. Eine Möglichkeit wäre es, durch eine View alle in Konflikt stehenden Dokumente zu filtern und den Anwender anschließend die Konflikte manuell lösen zu lassen. [9, S. 153ff]

Einordnung im CAP-Theorem

Da CouchDB auf Hochverfügbarkeit und auf horizontale Skalierung ausgelegt ist, lässt sich CouchDB im CAP-Theorem wie in Abbildung 4.2 einordnen.

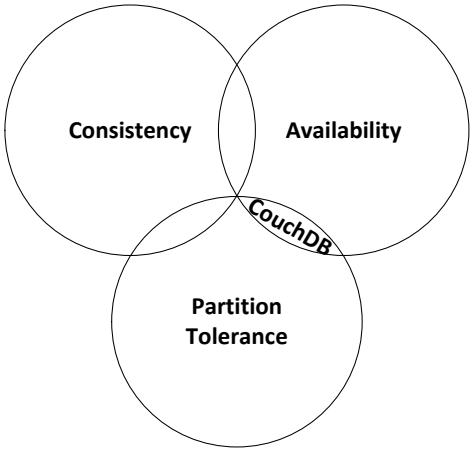


Abbildung 4.2.: Einordnung von CouchDB im CAP-Theorem [9, S. 12]

Zusammenfassung

Da CouchDB auf bewährte Webtechnologien setzt (RESTful JSON API), schemalos ist, flexible Replikationsmechanismen mit Konflikterkennung und -management bietet und durch CouchApps ganze Anwendungen bereitstellt, wird der Prototyp der Ausschreibungs- und Bewerbungsverwaltung mittels CouchDB realisiert.

4.1.2. KansoJS

KansoJS ist ein von Caolan McMahon entwickeltes Framework, mit dessen Hilfe auf einfache Art CouchApps erstellt werden können. KansoJS bietet eine CommonJS-Umgebung[6], die es erleichtert, Module einzubinden. Durch die Verwendung von `require()` hat man Zugriff auf das eingebundene Modul. Die Dateien können dabei beliebig organisiert sein, sie müssen sich nicht in definierten Ordnern befinden. KansoJS vermeidet zusätzlich Code-Fragmentierung, indem dieselbe Umgebung auf dem Server und Client läuft. Dadurch muss

[6] http://www.commonjs.org/

der Code nur an einer Stelle geschrieben werden und steht server- und clientseitig zur Verfügung. KansoJS ermöglicht es, nicht nur Ein-Seiten-AJAX-Anwendungen zu erstellen, auf die nur durch eine definierte URL zugegriffen werden kann. Stattdessen ist es möglich, dass zu jeder wohldefinierten URL eine vom Server generierte Seite gerendert wird. Somit ist KansoJS auch suchmaschinenfreundlich, was für Webanwendungen nicht unwesentlich ist. [3]

> »Kanso is a framework for creating web applications which run on CouchDB. You don't need anything else in your stack, meaning apps are easy to deploy and easy to distribute.
>
> The Kanso framework was designed for rapid development and code maintainability. Because it only requires CouchDB to host an app, you can write web apps which run on Windows, Mac, Linux, and even mobile devices!« [3]

Kanso-App-Struktur

Durch Erstellen eines Kanso-Projekts mittels `kanso create my_app` wird die in Abbildung 4.3 gezeigte Ordnerstruktur generiert:

- `lib`: CommonJS-Module, die zur Anwendung gehören
- `static`: Enthält sämtliche statische Elemente, wie jQuery[7] und CSS
- `templates`: HTML-Templates, die von der Anwendung benutzt werden

Die Ordnerstruktur ist nur ein Vorschlag von KansoJS, wie die einzelnen Dateien organisiert sein sollten. Die Organisation der Dateien lässt sich über die Datei `kanso.json` ändern. Den Inhalt von `kanso.json` nach dem Erstellen der Anwendung zeigt das Listing 4.2.

```
1 {
2   "name": "my_app",
3   "load": "lib/app",
4   "modules": "lib",
5   "templates": "templates",
6   "attachments": "static",
7   "base_template": "base.html"
8 }
```

Listing 4.2: `kanso.json` nach dem Erstellen eines KansoJS-Projekts

[7] http://jquery.com/

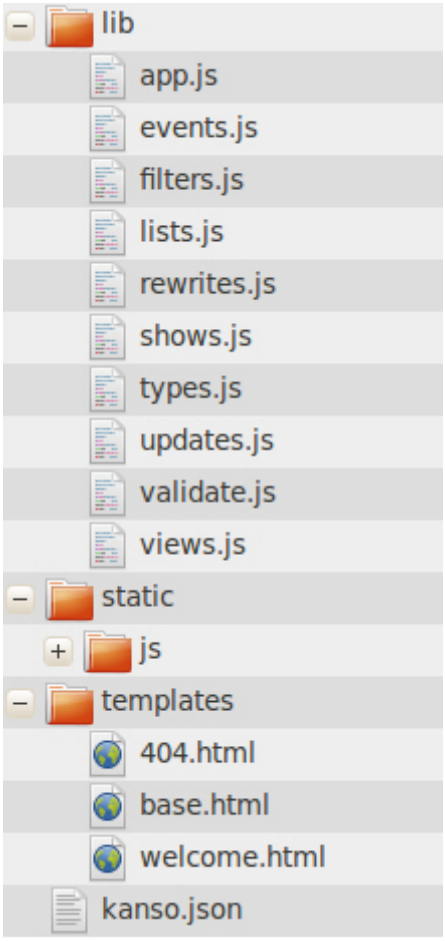


Abbildung 4.3.: Ordnerstruktur eines Kanso-Projekts

Modellierung mit KansoJS

KansoJS ermöglicht eine einfache Datenmodellierung. Typen lassen sich schnell anlegen. Diese können Widgets enthalten und Berechtigungen für das Hinzufügen, Aktualisieren und Löschen eines Dokuments des definierten Typs lassen sich schnell konfigurieren.

Listing 4.3 zeigt eine Definition eines Typs. Hier wird ein neuer Typ `blogpost` definiert. Dieser enthält die Felder `created`, `title` und `text`, wobei dem Feld `text` das Widget `textarea` zugeordnet ist. Auf diese Weise lässt sich bereits an dieser Stelle die Eingabemaske für das Feld definieren. Mit `permissions` werden Berechtigungen gesetzt. In diesem Fall dürfen Dokumente des Typs `blogpost` nur von CouchDB-Benutzern mit der Rolle `_admin` hinzugefügt, aktualisiert und gelöscht werden.

```
exports.blogpost = new Type('blogpost', {
  permissions: {
    add:    permissions.hasRole('_admin'),
    update: permissions.hasRole('_admin'),
    remove: permissions.hasRole('_admin')
  },
  fields: {
    created: fields.createdTime(),
    title: fields.string(),
    text: fields.string({
      widget: widgets.textarea({cols: 40, rows: 10})
    })
  }
});
```

Listing 4.3: Beispiel für eine Typ-Definition mit KansoJS

Templates Rendern und Formulare

Die Listings 4.4 und 4.5 zeigen, wie mit KansoJS Templates gerendert und Formulare benutzt werden.

Das Listing 4.4 zeigt das HTML-Template, das durch die show-Funktion `add_blogpost` benutzt wird. KansoJS benutzt dabei das Templatesystem Mustache[8]. In dem Template sind die Platzhalter `{form_title}` und `{form|s}` definiert, die durch die show-Funktion ersetzt werden.

Das Listing 4.5 zeigt eine show-Funktion, die aufgerufen wird, um ein Formular zum Hinzufügen eines Blogposts darzustellen. Es wird ein leeres Formular vom Typ `types.blogpost` aufgebaut, das alle Felder außer `created` des Typs enthält. Anschließend wird mit `templates.render` das Template `blogpost_form.html` mit dem aktuellen Request `req` gerendert. Dabei werden an das Template die beiden Elemente `form_title` und `form` übergeben. Mit `form.toHTML(req)` wird das HTML-Formular generiert. Das Rückgabe-Objekt `{title: 'Add new blogpost', content: content}` wird durch KansoJS verarbeitet und an das vom Server gerenderte `base_template` (siehe `kanso.json`) übergeben. Dabei entscheidet KansoJS selbst, ob dabei das gesamte Template (also sowohl das `base_template` als auch `blogpost_form.html`) serverseitig gerendert werden muss (da kein JavaScript auf dem Client läuft) oder ob nur `{title}` und `{content}` des `base_template` über jQuery aktualisiert werden müssen.

[8] http://mustache.github.com/

```
<h1>{form_title}</h1>

<form method="POST" action="">
  <table>
    {form|s}
  </table>
  <input type="submit" value="Add" />
</form>
```

Listing 4.4: HTML-Template, das durch die show-Funktion `add_blogpost` benutzt wird

```
exports.add_blogpost = function (doc, req) {
  var form = new forms.Form(types.blogpost, null, {
    exclude: ['created']
  });

  var content = templates.render('blogpost_form.html', req, {
    form_title: 'Add new blogpost',
    form: form.toHTML(req)
  });

  return {title: 'Add new blogpost', content: content};
};
```

Listing 4.5: show-Funktion, die das Template `blogpost_form.html` rendert

Zusammenfassung

KansoJS erleichtert also die Entwicklung von CouchApps. Es lassen sich auf einfache Weise Datentypen modellieren, Berechtigungen setzen, Widgets definieren und Formulare erstellen. Daher werden die Ausschreibungs- und Bewerbungsverwaltung (interne CouchDB-Komponente) und das Ausschreibungsportal (externe CouchDB-Komponente) mit Hilfe des KansoJS-Frameworks umgesetzt.

4.2. Architektur

Die Abbildung 4.4 zeigt das gesamte System mit der internen und externen CouchDB-Komponente, den Endgeräten des Personalbenutzers und des Bewerbers, der Webservice-Komponente zwischen interner und externer CouchDB-Komponente, sowie dem zwischengelagerten Kommunikationsmedium Internet. Im Folgenden werden die einzelnen Bestandteile der Architektur erläutert.

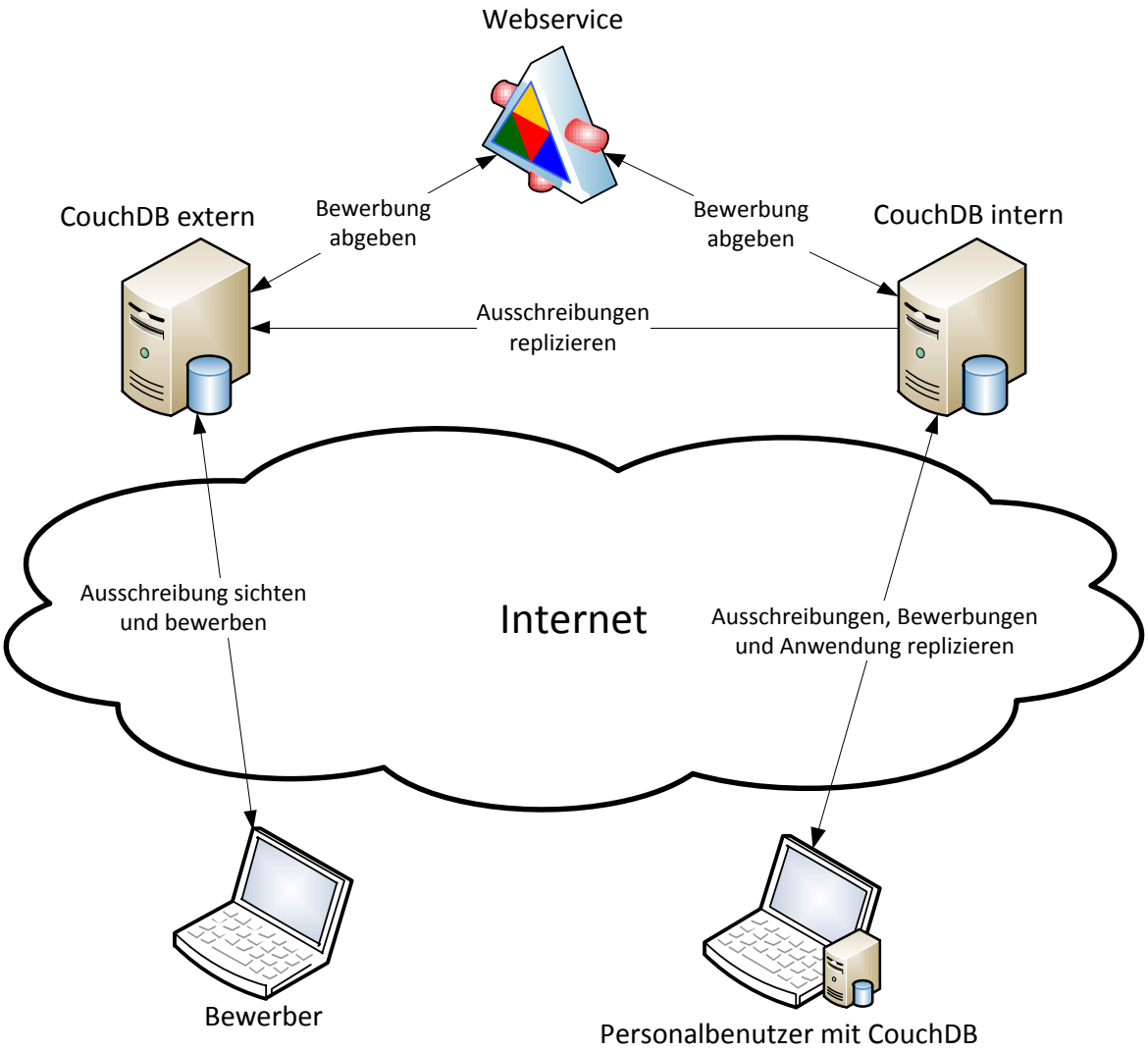


Abbildung 4.4.: Gesamtsystemübersicht

4.2.1. CouchDB intern

Die interne CouchDB-Komponente ist ein eigenständiger Server, auf dem eine CouchDB-Instanz installiert ist. Die Ausschreibungs- und Bewerbungsverwaltung (nachfolgt auch *AppMan* genannt) ist auf einer eigenen Datenbank auf der CouchDB-Instanz installiert und in Form einer Couchapp realisiert.

Die CouchApp sowie die Ausschreibungs- und Bewerbungsdaten werden über das Internet zwischen der CouchDB-Instanz des Endgeräts des Personalbenutzers und der AppMan-Datenbank auf der internen CouchDB-Komponente bidirektional repliziert. Die Replikation erfolgt dabei kontinuierlich. Somit stehen dem Personalbenutzer sämtliche Ausschreibungs-, Bewerbungsdaten und die Anwendung auch dann zur Verfügung, wenn keine Verbindung zur internen CouchDB-Komponente besteht. AppMan wird also bei Änderungen automatisch auf dem Endgerät des Personalbenutzers aktualisiert.

Auf AppMan dürfen nur autorisierte Benutzer zugreifen. Dies wird durch die Zugriffskontrolle von CouchDB geregelt. Es dürfen nur Benutzer mit der Rolle »human_resources« auf die Datenbank zugreifen. Dadurch wird gewährleistet, dass alle Daten in der Datenbank von AppMan vor unberechtigtem Zugriff geschützt sind. Dies ist wichtig, da Bewerbungen personenbezogene Daten enthalten und diese nur durch befugte Benutzer einsehbar sein sollen (vgl. 3.2.2).

4.2.2. CouchDB extern

Die externe CouchDB-Komponente ist ebenfalls wie die interne CouchDB-Komponente ein eigenständiger Server, der eine lauffähige CouchDB-Instanz enthält. Die Jobbörse ist auf einer eigenen Datenbank der CouchDB-Instanz installiert und ebenfalls in Form einer Couchapp realisiert.

Durch Replikation werden zu der Datenbank der Jobbörse die Ausschreibungen repliziert. Die Replikation wird durch die *_replicator*-Datenbank der CouchDB auf der externen CouchDB-Komponente gesteuert. Die Datenbank der Jobbörse enthält stets eine Kopie der in der Ausschreibungen, wie sie sich in der Datenbank von AppMan befinden. Dadurch ist die Aktualität der Ausschreibungen gewährleistet.

Der Zugriff auf die Datenbank der Jobbörse muss ohne Authentifizierung erfolgen können, da Bewerber auf die Couchapp haben müssen. Es ist jedoch nicht gewünscht, dass unberechtigte Benutzer schreibend auf die Datenbank zugreifen können. Durch eine Validierungsfunktion in der CouchApp der Jobbörse werden Schreibzugriffe unberechtigter Benutzer verhindert.

Für jede Ausschreibung ist es dem potentiellen Bewerber möglich, ein Bewerbungsformular auszufüllen und abzusenden. Dabei leitet die Jobbörse die Bewerbung lediglich an den

Webservice weiter, welcher das Speichern der Bewerbung in der Datenbank von AppMan übernimmt.

4.2.3. Webservice

Der Webservice befindet sich auf dem Server der externen CouchDB-Komponente. Er dient dazu, Bewerbungen bei der AppMan-Datenbank zu speichern und über Erfolg oder Misserfolg des Speicherns zu informieren. Denkbar wäre auch, dass der Webservice sich auf einem eigenständigen Server befinden würde.

Der Einsatz eines eigenständigen Webservices zum Speichern einer Bewerbung ist notwendig, da die Jobbörse als Couchapp umgesetzt ist und in einer nicht lesegeschützten Datenbank liegt. Dadurch ist auch der Programmcode der Couchapp zugänglich. Um eine Bewerbung bei der AppMan-Datenbank zu speichern, ist eine Authentifizierung notwendig. Die Couchapp der Jobbörse müsste also die dafür notwendigen Zugangsdaten enthalten. Diese wären dann offen zugänglich und die Datenbank von AppMan wäre nicht mehr geschützt vor unberechtigtem Zugriff. Der Code des Webservices ist geschützt und somit ein Ausspähen von Zugangsdaten für die AppMan-Datenbank nicht möglich.

Der Webservice soll übersichtlich gehalten werden und der Zugriff auf den Webservice soll in Anlehnung an die API von CouchDB ebenfalls RESTful sein.

4.2.4. Personalbenutzer

Auf dem Endgerät des Personalbenutzers muss eine CouchDB-Instanz installiert sein. Diese enthält eine Datenbank für AppMan. Die Daten werden bidirektional zwischen der AppMan-Datenbank des Personalbenutzers und der AppMan-Datenbank der internen CouchDB-Komponente repliziert. Die Replikation ist in der *_replicator*-Datenbank des Personalbenutzers einzustellen. Somit ist es nicht notwendig, die interne CouchDB-Komponente anzupassen, wenn weitere Personalbenutzer hinzukommen.

Die Personalbenutzer arbeiten immer auf ihrer lokalen AppMan-Datenbank. Sie können auch weiterarbeiten, ohne dass eine Verbindung zur internen CouchDB-Komponente besteht. Jedoch sind bei einer solchen Arbeitsweise Konflikte möglich. Abschnitt 4.3 behandelt die Konfliktentstehung und -lösung.

4.2.5. Bewerber

Der Bewerber geht mittels eines Browsers auf die Jobbörse und greift lesend auf die Datenbank der Jobbörse zu. Er sieht die aktiven Ausschreiben, die sich derzeit im System befinden und kann sich auf eine Ausschreibung mittels eines Bewerbungsformulars bewerben.

Um Ausschreibungen einzusehen, muss eine Verbindung zur Jobbörse bestehen. Für die Abgabe einer Bewerbung ist es erforderlich, dass zwischen der Jobbörse, dem Webservice und der AppMan-Datenbank eine Verbindung besteht. Bei Fehlern in der Verbindungskette wird der Bewerber über den Fehlschlag der Abgabe seiner Bewerbung informiert.

4.3. Konfliktbehandlung

Falls mehrere Benutzer offline an denselben Datensätzen arbeiten, kommt es zu Konflikten, sobald wieder eine Verbindung zur internen CouchDB-Komponente besteht. In diesem Abschnitt wird zunächst die Entstehung eines Konflikts beschrieben und anschließend eine Möglichkeit zur Lösung der Konflikte aufgezeigt.

4.3.1. Entstehung von Konflikten

Angenommen, es existiert eine Ausschreibung A1 (Version 1) mit der zwei Benutzer B1 und B2 offline arbeiten. Die Ausschreibung wurde bereits auf die Geräte der Benutzer repliziert, sodass die Benutzer auch offline arbeiten können. Benutzer B1 ändert die Ausschreibung und nennt diese A11. Dabei wird der Ausschreibung eine neue Version (Version 2a) auf dem lokalen Gerät des Benutzers B1 zugeordnet. Benutzer B2 ändert ebenfalls auf seinem lokalen Gerät die Ausschreibung und nennt diese A12. Die Ausschreibung erhält die Version 2b. Nun geht Benutzer B1 wieder online. Seine Änderungen werden zur internen CouchDB-Komponente repliziert. Die Ausschreibung heißt nun A11 (Version 2a). Anschließend geht auch Benutzer B2 online und seine Änderungen werden repliziert. Dabei entsteht ein Konflikt, da nicht eindeutig entschieden werden kann, welche der getätigten Änderungen mehr Gewicht haben. Für die Darstellung in Views gibt es eine »gewinnende« Version. In der CouchDB bestimmt ein deterministischer Algorithmus auf dieselbe Art und Weise die gewinnende Version [9, S. 160]. In diesem Fall wird Version 2b als »gewinnende« Version gewählt. Version 2a wird als konfligierte Version mit Version 2b verknüpft. Abbildung 4.5 zeigt die hier beschriebene Entstehung eines Konflikts.

4.3.2. Lösen von Konflikten

Ein entstandener Konflikt kann durch den Benutzer gelöst werden. Jede konfligierte Bewerbung bzw. Ausschreibung wird als Konflikt angezeigt und es wird ein Menüpunkt »Konflikte bearbeiten« eingeblendet. In der Funktion »Konflikte bearbeiten« stehen dem Benutzer neben der aktuellen Version auch alle in Konflikt stehenden Versionen des Dokuments zur Verfügung. Um den Konflikt zu lösen, muss der Benutzer die konfligierten Versionen löschen. Dabei hat er die Möglichkeit die aktuelle Version abzuändern und somit einen manuellen Abgleich der Versionen vorzunehmen.

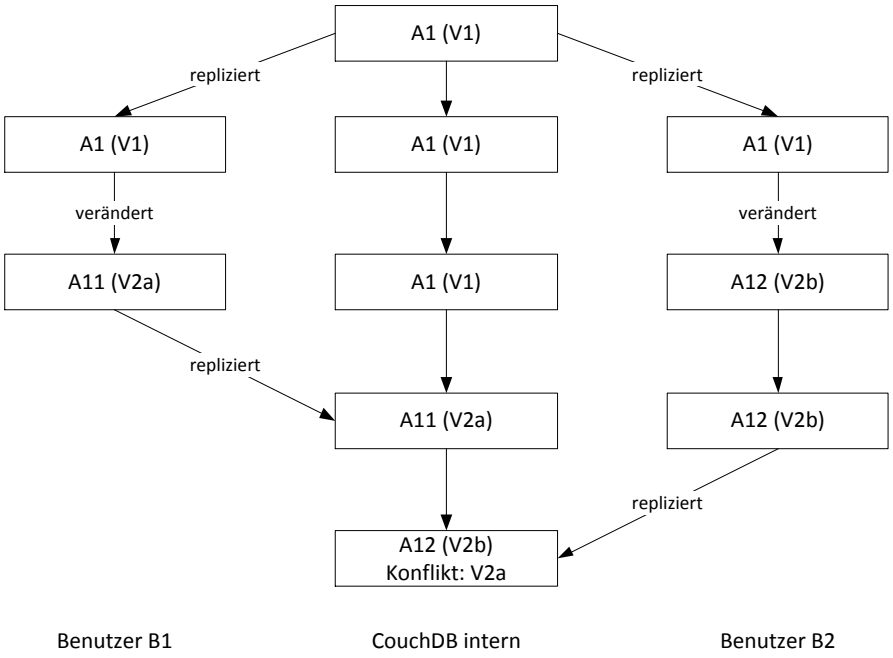


Abbildung 4.5.: Entstehung eines Konflikts

Auch bei dieser Arbeitsweise ist es möglich, dass wiederum Konflikte entstehen, denn in der Zwischenzeit könnte ein anderer Benutzer (Benutzer B) bereits den Konflikt offline gelöst haben. Benutzer A hat davon noch keine Kenntnis. Wenn der Benutzer A dann seine Variante der Konfliktlösung speichert, entsteht wiederum ein Konflikt. Solche Konflikte lassen sich nicht vermeiden und müssen erneut durch den Benutzer gelöst werden.

4.4. Test

Wie in Kapitel 3.2.2 definiert soll für jede funktionale Anforderung ein Test erstellt werden. Um die Tests der funktionalen Anforderungen zu realisieren, eignen sich *Black-Box-Tests* (siehe Abbildung 4.6). Dabei wird das zu testende System als eine Black Box betrachtet. Der Tester besitzt keine Kenntnisse von der Implementierung des Systems. Der Tester nimmt Eingaben am System vor und untersucht die entsprechenden Ausgaben. Wenn die Ausgaben nicht den Vorhersagen entsprechen, hat der Test erfolgreich ein Softwareproblem aufgedeckt. [26, S. 450]

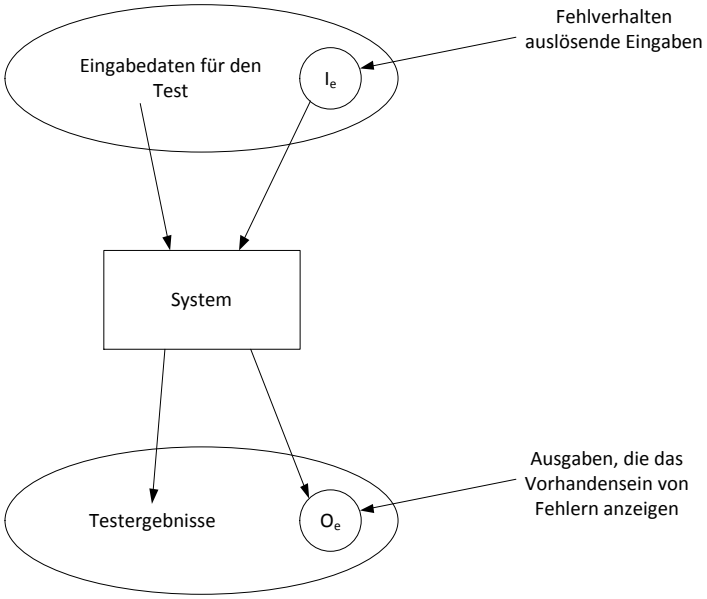


Abbildung 4.6.: Black-Box-Tests [26, S. 450]

4.4.1. Selenium

Für das Testen von Webanwendungen eignet sich das Webanwendungstestsystem Selenium[9]. Mit Hilfe von Selenium ist es möglich, Tests aufzuzeichnen und diese anschließend beliebig oft automatisiert ablaufen zu lassen. Selenium bietet drei Komponenten [5]:

1. *Selenium IDE*:
   Ein Plugin für Firefox, das Klicks, Eingaben und andere Aktionen des Benutzers aufzeichnet, um einen Test zu erstellen, der sich im Browser ausführen lässt.
2. *Selenium Remote Control (RC)*:
   Ermöglicht es, die aufgezeichneten Tests in unterschiedlichen Browsern und auf unterschiedlichen Plattformen ablaufen zu lassen. Dies kann in unterschiedlichen Programmiersprachen erfolgen (z.B. Java, .Net, Perl, PHP, Python, Ruby).
3. *Selenium Grid*:
   Erweitert Selenium RC, um die Tests auf unterschiedliche Server zu verteilen, um die Testgeschwindigkeit durch Parallelisierung zu erhöhen. Außerdem lassen sich dadurch auch Lasttests auf einem System durchführen.

Im Rahmen dieser Arbeit werden die Testfälle für die funktionalen Anforderungen mit Selenium IDE erstellt und können im Browser ausgeführt werden. Durch Selenium RC und Selenium Grid ist es anschließend möglich, die Tests automatisiert ablaufen zu lassen.

[9] http://seleniumhq.org/

Zur Zeit bietet KansoJS keine »Testgetriebene Entwicklung«. Die Diskussion um Testgetriebene Entwicklung für KansoJS ist in Gange. In Zukunft soll KansoJS laut Caolan McMahon Testgetriebene Entwicklung unterstützen. [6]

5. Realisierung

In diesem Kapitel wird beschrieben, wie der Entwurf umgesetzt wurde (vgl. Kapitel 4). Dabei werden die einzelnen Komponenten des Systems beschrieben und zum Schluss die entwickelten Testfälle vorgestellt.

5.1. Einschränkungen

In der aktuellen Version 0.0.7 unterstützt KansoJS ausschließlich CouchDB ab Version 1.1.0. Für die Entwicklung des Prototypen wird daher CouchDB in der Version 1.1.0 benutzt. Da sowohl CouchDB als auch KansoJS sich einfach auf Unix-Systemen installieren lassen, wurde für die Entwicklung des Prototypen die Linux-Distribution Ubuntu 10.04[1] als Betriebssystem verwendet. Die Entwicklung fand in virtuellen Maschinen statt.

Die externe und die interne Komponente befindet sich jeweils auf einem eigenen Server. Durch die physikalische Trennung beider Komponenten lassen sich die Zugriffe auf die jeweilige CouchDB effektiver Einschränken (vgl. 3.2.2). Die beiden Komponenten befinden sich im selben Netzwerk. Jedoch ist es ohne Weiteres möglich, die beiden Komponenten in verschiedenen Netzwerken unterzubringen.

5.2. CouchDB intern

Die CouchDB-Instanz ist so eingestellt, dass nur autorisierte Benutzer über Netzwerk Lese- und Schreibzugriffe ausüben können (`require_valid_user=true`). Die Authentifizierung wurde auf *Basic Auth* belassen, da der entwickelte Prototyp zunächst nur im LAN betrieben wird. Für den Produktiveinsatz sollte die Authentifizierung jedoch geändert werden, falls die Zugriffe auf CouchDB über eine unsichere Verbindung erfolgen sollten (vgl. 4.1.1).

[1] http://www.ubuntu.com/

5.2.1. Couchapp der Ausschreibungs- und Bewerbungsverwaltung (AppMan)

Die Couchapp der Ausschreibungs- und Bewerbungsverwaltung (nachfolgend AppMan genannt) wurde mit Hilfe des KansoJS-Frameworks (vgl. 4.1.2) umgesetzt. Der Zugriff auf die AppMan-Datenbank ist eingeschränkt, sodass als *Admins* nur Benutzer mit der Rolle *_admin* und als *Reader* Benutzer mit der Rolle *human_resources* auf die Datenbank zugreifen dürfen. Denkbar wäre auch gewesen, dass die AppMan-Datenbank auf der internen CouchDB-Komponente nur für *Admins* zugänglich ist, da auf diese Datenbank Personalbenutzer keinen direkten Zugriff haben. Allerdings ist es erforderlich, dass die AppMan-Datenbanken der einzelnen Personalbenutzer mittels bidirektionaler Replikation Lese- und Schreibzugriffe auf die AppMan-Datenbank der internen CouchDB-Komponente ausüben. Diese Zugriffe entsprechen den Zugriffen eines *Readers* von CouchDB. Daher ist es sinnvoll, eine Benutzergruppe als Reader zu definieren.

AppMan beinhaltet zwei Datentypen *job* (Ausschreibung) und *application* (Bewerbung), wobei jeder Bewerbung genau eine Ausschreibung zugeordnet ist. Dies wird durch AppMan sichergestellt, indem bei jeder Bewerbung die *_id* der Ausschreibung gespeichert wird.

Die Übersicht der Ausschreibungen und Bewerbungen beinhaltet stets die Aktionen *Anzeigen*, *Bearbeiten*, *Löschen* und falls Konflikte bestehen *Konflikte bearbeiten*. Ebenfalls stehen in der Übersicht Funktionen zum Hinzufügen von Ausschreibungen und Bewerbungen bereit.

Hinzufügen, Bearbeiten und Löschen

Das Hinzufügen einer Ausschreibung oder Bewerbung erfolgt, indem zunächst die dafür vorgesehene show-Funktion aufgerufen wird. Der Aufruf der show-Funktion erfolgt durch eine GET-Methode (GET /jobs/add). Die show-Funktion rendert das zum Hinzufügen benötigte Formular. Das Formular ergibt sich aus den definierten Datentypen. Sobald das Formular abgesendet wird, wird die passende update-Funktion durch eine POST-Methode aufgerufen (POST /jobs/add). Die update-Funktion prüft zunächst, ob das übergebene Formular valide ist. Bei fehlenden Pflichtfeldern wird das Formular mit den bereits eingegebenen Daten neu gerendert und mit entsprechenden Fehlermeldungen für die Pflichtfelder ergänzt. Sobald das Formular valide ist, wird das Dokument an die Datenbank gesendet und gespeichert. Der Benutzer bekommt als Feedback eine Erfolgs-, bzw. Fehlermeldung. Abbildung 5.1 zeigt den Ablauf des Hinzufügens einer Ausschreibung.

Analog zum Hinzufügen einer Ausschreibung oder Bewerbung erfolgt deren Bearbeitung. Das dazu benötigte Formular wird durch eine show-Funktion bereitgestellt. Die update-Funktion validiert das Formular und speichert das Dokument in der Datenbank. Für die Anzeige einer Ausschreibung oder Bewerbung wird lediglich eine show-Funktion benötigt,

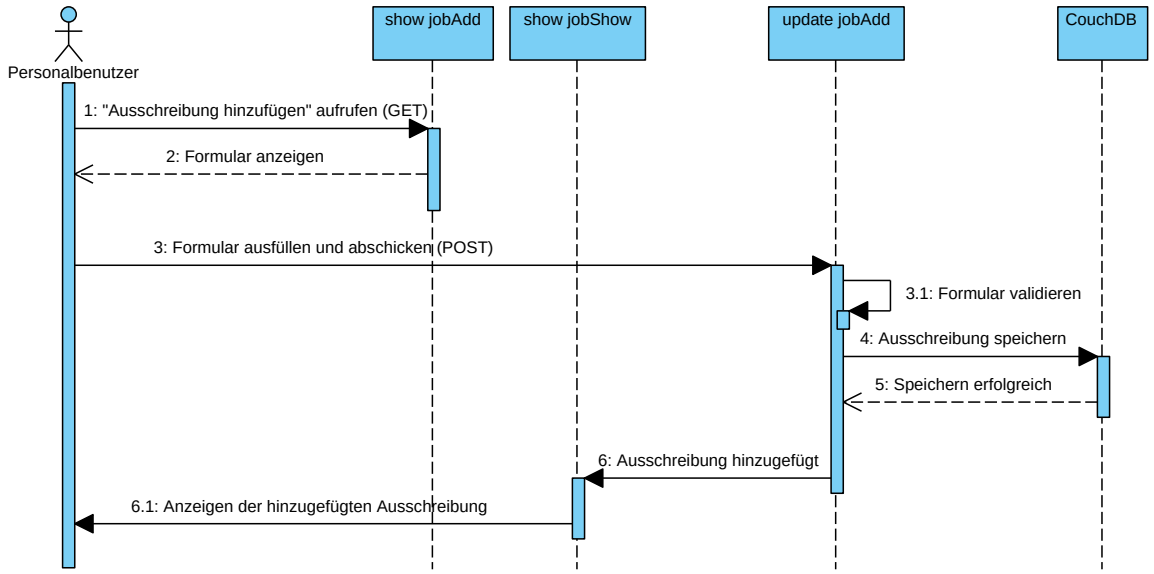


Abbildung 5.1.: Sequenzdiagramm: Hinzufügen einer Ausschreibung

die das Dokument anzeigt, ohne dass es der Benutzer bearbeiten kann. Für das Löschen einer Ausschreibung oder Bewerbung werden ebenfalls eine show- und eine update-Funktion benötigt. Die show-Funktion fragt den Benutzer, ob das Dokument wirklich gelöscht werden soll. Die update-Funktion führt eine DELETE-Operation auf der Datenbank aus und gibt dem Benutzer ein Feedback. Das Löschen von Ausschreibungen ist nur dann möglich, wenn ihr keine Bewerbungen zugeordnet sind. Außerdem ist das Löschen von Ausschreibungen und Bewerbungen nur dann möglich, wenn sie sich nicht in einem Konflikt befinden.

Konflikte bearbeiten

Wie in Abschnitt 4.3 erläutert, können im Betrieb von AppMan Konflikte entstehen, wenn beispielsweise mehrere Benutzer an demselben Datensatz offline arbeiten. Konflikte werden durch AppMan automatisch erkannt und in der Ausschreibungs- bzw. Bewerbungsübersicht sowie beim Anzeigen einer Ausschreibung bzw. Bewerbung angezeigt. Der Benutzer kann bei konfligierten Ausschreibungen oder Bewerbungen die Funktion *Konflikte bearbeiten* aufrufen (siehe Abbildung 5.2). Beim Bearbeiten der Konflikte wird dem Benutzer die aktuelle Version des Dokuments samt Revisionsnummer (vgl. 4.1.1) und die konfligierten Versionen mit Revisionsnummer angezeigt. Der Benutzer muss einen manuellen Abgleich vornehmen und dabei die richtigen Werte der aktuellen Version zuordnen. Nach dem Speichern der abgeglichenen aktuellen Version bestehen die Konflikte weiterhin. Nun kann der Benutzer aber die konfligierten Versionen löschen und der Konflikt ist anschließend gelöst.

Wenn mehrere Personalbenutzer denselben Konflikt offline bearbeiten, kommt es zu einem

erneuten Konflikt. Dieser müsste dann ebenfalls von einem Personalbenutzer gelöst werden. Dieser Fall lässt sich nicht gänzlich vermeiden.

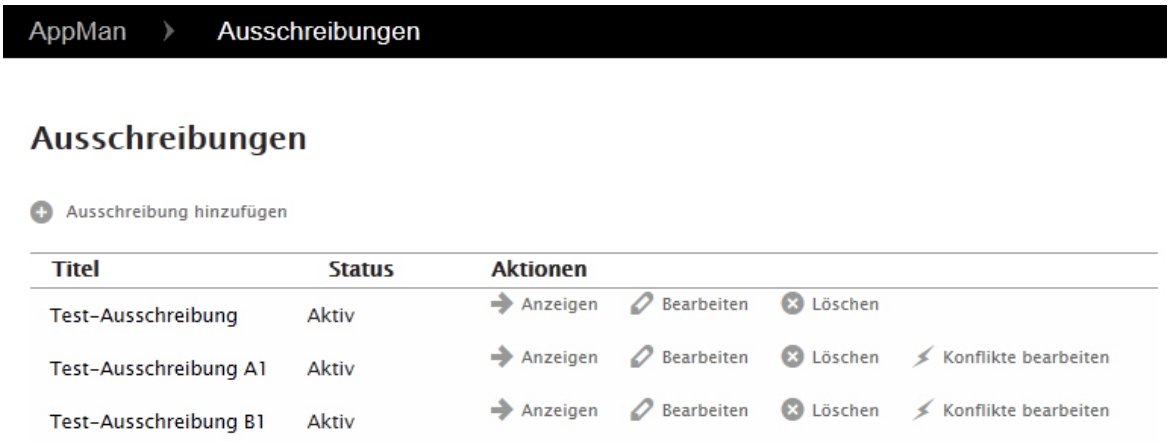


Abbildung 5.2.: Erkannte Konflikte in der Ausschreibungsübersicht

5.2.2. Bereitstellung auf lokalen Geräten der Personalbentuzer

Es muss eine CouchDB-Instanz auf dem Gerät des Personalbenutzers installiert sein, damit dieser lokal arbeiten kann. Die CouchDB-Instanz muss und sollte nicht aus dem Netzwerk erreichbar sein, damit ausschließlich Benutzer des lokalen Geräts darauf Zugriff haben. Dennoch sollte auch die CouchDB-Instanz des Personalbenutzers Administratoren haben und die *Reader* und *Admins* der AppMan-Datenbank des Personalbenutzers sollten analog zu der AppMan-Datenbank der internen Komponente eingestellt sein. Dadurch wird die Sicherheit vor unbefugten Zugriffen auf personenbezogene Daten erhöht (vgl. 3.2.2). Im Prototyp wurden die Sicherheitseinstellungen insoweit erweitert, sodass für jeden Zugriff auf die CouchDB des Personalbenutzers eine Authentifizierung notwendig ist (`require_valid_user=true`). Dies hat den Vorteil, dass beim Zugriff ohne Authentifizierung eine Login-Aufforderung vom Browser angezeigt wird und der Benutzer sich einloggen kann.

Die AppMan-Couchapp wird auf den lokalen Geräten der Personalbenutzer mittels Replikation bereitgestellt. Alle Daten zwischen der CouchDB-Instanz der internen Komponente und der des Personalbenutzers werden bidirektional repliziert, die Couchapp inbegriffen. Dem Personalbenutzer steht also immer die aktuellste Version der AppMan-Couchapp zur Verfügung (falls eine Verbindung zwischen der CouchDB-Instanz des Personalbenutzers und interner Komponente besteht). Die Listings 5.1 und 5.2 zeigen für die Replikation notwendigen Einträge der *_replicator*-Datenbank der CouchDB-Instanz des Personalbenutzers.

```
{
   "_id": "appman-192.168.0.201-localhost",
   "target": "http://admin:secret@localhost:5984/appman",
   "source": "http://admin:secret@192.168.0.201:5984/appman",
   "continuous": true
}
```

Listing 5.1: Replikationseinstellungen zwischen CouchDB-Instanz des Personalbenutzers und interner Komponente

```
{
   "_id": "appman-localhost-192.168.0.201",
   "source": "http://admin:secret@localhost:5984/appman",
   "target": "http://admin:secret@192.168.0.201:5984/appman",
   "continuous": true
}
```

Listing 5.2: Replikationseinstellungen zwischen CouchDB-Instanz der internen Komponente und der des Personalbenutzers

5.3. CouchDB extern

Die CouchDB-Instanz der externen Komponente ist so eingestellt, dass nur auf die Datenbank der Jobbörse ohne Authentifizierung zugegriffen werden kann. Dies ist so realisiert, dass keine Reader für die Datenbank definiert wurden, jedoch dürfen nur Benutzer mit der Rolle *_admin* als Admins auf die Datenbank zugreifen. Schreibzugriffe werden durch eine Validierungsfunktion unterbunden, die sicherstellt, dass nur eingeloggte Benutzer schreibend auf die Datenbank zugreifen dürfen.

Die Jobbörse enthält als Datentyp eine nahezu 1:1-Kopie des *job*-Datentyps aus der AppMan-Couchapp. Da die beiden Couchapps auf unterschiedlichen Systemen liegen, lässt sich dieses partielle Codeduplikat nicht vermeiden.

Die Jobbörse zeigt nur aktive Ausschreibungen an und stellt sicher, dass Bewerber sich nur auf aktive Ausschreibungen bewerben können. Da jedoch die Datenbank der Jobbörse für Lesezugriffe offen zugänglich ist, ist es möglich, die CouchDB-API zu benutzen und so auch inaktive Ausschreibungen einzusehen. Das Bewerben auf eine inaktive Ausschreibung wird aber von der für die Abgabe der Bewerbung zuständigen update-Funktion unterbunden.

5.3.1. Replikation von Ausschreibungen

Die Replikation für die Jobbörse wird in der *_replicator*-Datenbank der externen CouchDB-Komponente eingestellt. Da keine Reader für die Datenbank definiert wurden und Schreib-

zugriffe eine Authentifizierung erfordern, muss die Replikation so eingestellt werden, dass ein Benutzer mit der Rolle *_admin* schreibend auf die Jobbörse-Datenbank zugreift. Da nur Ausschreibungen repliziert werden sollen, ist es notwendig, dass die Replikation gefiltert abläuft. Als Filter wurde *appman/jobFilter* verwendet. Der Filter ist in der AppMan-Couchapp realisiert und filtert alle Ausschreibungen heraus. Andere Dokumente werden nicht zwischen AppMan und Jobbörse repliziert. Listing 5.3 zeigt die beschriebenen Replikationseinstellungen.

```
1 {
2    "_id": "appman-appman_portal",
3    "source": "http://admin:secret@<IP der internen CouchDB-Komponente
        >:5984/appman",
4    "target": "http://admin:secret@localhost:5984/appman_portal",
5    "filter": "appman/jobFilter",
6    "continuous": true
7 }
```

Listing 5.3: Replikationseinstellungen zwischen Jobbörse und AppMan

Kompromisse bei der Replikation der Ausschreibungen

Der Filter für die Replikation der Ausschreibungen lässt alle Ausschreibungen durch. Das bedeutet, dass auch inaktive Ausschreibungen in der Jobbörse landen, jedoch nicht von der Couchapp der Jobbörse angezeigt werden. Wäre der Filter so aufgebaut, dass nur aktive Ausschreibungen repliziert werden, würde es zu einem Fehlverhalten der Replikation kommen: Würde man den Status einer aktiven Ausschreibung auf inaktiv ändern, würde die Replikation für die nun inaktive Ausschreibung nicht angestoßen und die Ausschreibung befände sich weiterhin in der Jobbörse und hätte den Status aktiv. Daher lässt der Filter generell alle Ausschreibungen durch.

Zu einem Fehlverhalten würde es auch kommen, wenn eine aktive Ausschreibung, die sich auch in der Jobbörse befindet, direkt gelöscht würde. Auch dann würde der Replikationsmechanismus nicht greifen und die Ausschreibung würde weiterhin in der Jobbörse angezeigt. Daher muss dafür Sorge getragen werden, dass alle zu löschenden Ausschreibungen vorher auf den Status inaktiv gesetzt werden. Anschließend können sie gelöscht werden. Jedoch sind auch dann die bereits gelöschten Ausschreibungen in der Datenbank der Jobbörse vorhanden. Sie werden lediglich nicht angezeigt.

5.4. Webservice

Der Webservice spielt eine entscheidende Rolle beim Bewerben. Er steuert die Abgabe der Bewerbung, indem er die Anfragen von der Jobbörse an die interne CouchDB-Komponente

weiterleitet und über über Erfolg oder Misserfolg einer Anfrage informiert. Der Webservice hat zwei Aufgaben:

1. Anfordern einer neuen UID der internen CouchDB.
2. Speichern der Bewerbung in der internen CouchDB.

Der Webservice wurde in Java mit Jersey[2] realisiert. Jersey ist eine Open Source Referenzimplementierung von JAX-RS (Java API for RESTful Web Services). In der Version 1.8 implementiert Jersey Version 1.1 der JAX-RS-Spezifikation [2]. Der entstandene Webservice ist in einer Tomcat[3]-Umgebung lauffähig und kann folgende HTTP-Befehle verarbeiten:

- `GET /get_uid`: Durch diesen Befehl wird der Webservice dazu veranlasst, eine neue UID der CouchDB der internen Komponente zu besorgen. Bei Erfolg wird der HTTP-Status 200 (OK) mit neuer UID in JSON-Form zurückgegeben. Bei Misserfolg wird HTTP-Status 404 (Not Found) zurückgegeben
- `POST /save_app/uid`: Als JSON-String muss zusätzlich noch die Bewerbung übermittelt werden. Der Webservice nimmt die Bewerbung entgegen und speichert diese in der CouchDB unter der angegebenen *uid* ab. Bei Erfolg wird HTTP-Status 201 (Created) zurückgegeben. Falls der Versuch, die Bewerbung zu speichern fehlschlägt, wird HTTP-Status 404 zurückgegeben oder (falls möglich) der HTTP-Status der CouchDB durchgereicht.

Der Webservice ist also nur ein reiner Vermittler zwischen Jobbörse und AppMan. Er ändert die Bewerbungen, die er entgegennimmt nicht ab, sondern reicht diese nur weiter, ebenso wie er die Antworten von der CouchDB nach Möglichkeit weiterreicht.

5.4.1. Bewerben

Der Webservice ist im Kern für das Speichern einer Bewerbung verantwortlich. Die Abbildung 5.3 zeigt den Abgabeprozess einer Bewerbung. Der Bewerber lässt sich zunächst eine Liste der Ausschreibungen anzeigen. Anschließend ruft er die Ausschreibungdetails einer Bewerbung und darauf folgend die Funktion *Bewerben* auf. Dem Bewerber wird nun ein Bewerbungsformular angezeigt, welches er ausfüllt und abschickt. Nach erfolgreicher Validierung des Bewerbungsformulars wird beim Webservice eine neue UID angefordert. Der Webservice fordert wiederum eine neue UID bei der internen CouchDB an und gibt diese anschließend an die update-Funktion der Jobbörse zurück. Die update-Funktion reichert die Bewerbung um die UID an und sendet diese wiederum an den Webservice in JSON-Form. Dieser speichert die Bewerbung bei der internen CouchDB ab und sendet im Erfolgsfall eine

[2] http://jersey.java.net/
[3] http://tomcat.apache.org/

Erfolgsmeldung an die update-Funktion der Jobbörse. Die update-Funktion ruft die show-Funktion auf und dem Benutzer wird eine Erfolgsmeldung über die Abgabe der Bewerbung angezeigt.

Falls es in der Kommunikationskette zwischen der Jobbörse, dem Webservice und der internen CouchDB zu einem Fehler kommt, wird der Benutzer über das Scheitern der Bewerbungsabgabe informiert. Die eingegebenen Daten des Bewerbers bleiben bestehen, sodass er einen erneuten Versuch starten kann die Bewerbung abzugeben.

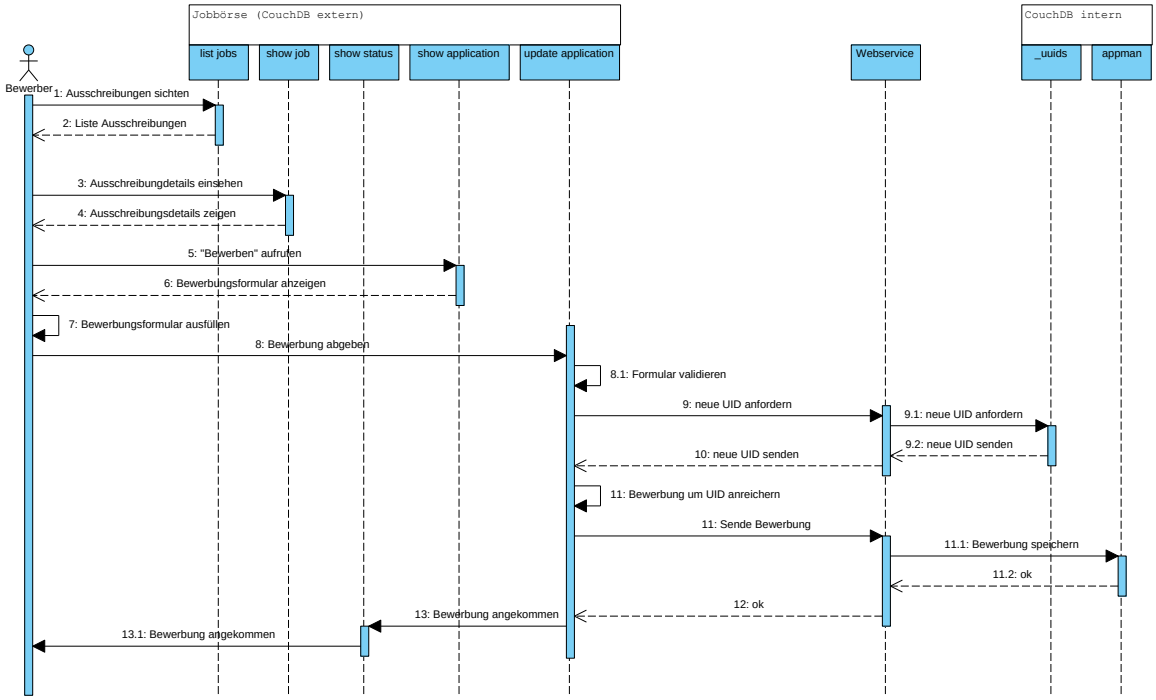


Abbildung 5.3.: Sequenzdiagramm: Bewerbern

5.5. Test

Wie in Kapitel 4.4 beschrieben, wurden die Tests mit Hilfe von Selenium entwickelt. Die im Rahmen dieser Arbeit entwickelten Tests lassen sich nur über die Selenium IDE manuell ausführen. Eine Automatisierung der Tests ist jedoch denkbar. Die Tests könnten beispielsweise in Java-Code mit Hilfe von Selenium RC konvertiert und anschließend mit Hilfe eines *Continuous Integration Servers* wie beispielsweise *Jenkins*[4] ausgeführt werden.

[4] http://jenkins-ci.org/

5.5.1. Erstellte Testfälle

Die erstellten Testfälle befinden sich im in den Unterordnern unter */tests/*. Für jede Funktionale Anforderung wurde eine Testsuite angelegt. Der Name der Testsuite ist wie folgt aufgebaut: *ts_fa<Nummer>_<Beschreibung>.html*. Jede Testsuite testet die funktionale Anforderung und löscht anschließend wieder die erstellten Testdaten.

Ausschreibungen verwalten

Die Testsuiten für die funktionalen Anforderungen für *Ausschreibungen verwalten* befinden sich im Ordner */tests/ausschreibungen_verwalten*.

- *FA101 - Ausschreibungen erstellen*:
  Der Test legt eine Ausschreibung an. Falls die Ausschreibung erfolgreich angelegt werden konnte und der Status der angelegten Ausschreibung inaktiv ist, gilt der Test als erfolgreich.
- *FA102 - Ausschreibungen bearbeiten*:
  Der Testfall legt eine Ausschreibung an und verändert diese anschließend. Es werden die Felder der Ausschreibung, sowie der Status geändert. Der Test ist dann erfolgreich, wenn die Änderungen durchgeführt werden konnten.
- *FA103 - Ausschreibungen löschen*:
  Der Test ist dann erfolgreich, wenn eine angelegte Ausschreibung sich nur dann löschen lässt, wenn ihr keine Bewerbungen zugeordnet sind und der Status der Ausschreibung inaktiv ist.
- *FA104 - Ausschreibungsübersicht*:
  Der Test legt zwei neue Ausschreibungen mit unterschiedlichen Status an. Er prüft nur darauf, ob in der Ausschreibungsübersicht die Status der Ausschreibungen angezeigt werden. Im aktuellen Prototypen von AppMan ist es mit Selenium nicht möglich zu prüfen, ob der Status zu der Ausschreibung passt. Man könnte nur zu einer genauen Tabellenreihe die x-te Spalte prüfen. Da aber die Ausschreibungen nicht immer in derselben Reihe stehen werden (wenn weitere Ausschreibungen im System vorhanden sind), lässt sich die Tabellenreihe nicht definieren. Der Test prüft anschließend, ob in den Ausschreibungsdetails der richtige Status definiert ist. Ferner ist im aktuellen Prototypen die Funktion nicht realisiert, um sich alle Bewerbungen einer bestimmten Ausschreibung anzeigen zu lassen.

Bewerbungen verwalten

Für die funktionalen Anforderungen für *Bewerbungen verwalten* befinden sich die erstellten Testsuiten im Ordner */tests/bewerbungen_verwalten*.

- *FA201 - Bewerbungen erstellen*:
  Der Test legt eine neue Ausschreibung und zwei neue Bewerbungen an, wobei die Bewerbungen der Ausschreibung zugewiesen werden. Der Test gilt als erfolgreich, wenn die Ausschreibung und die Bewerbungen erfolgreich angelegt werden konnten.
- *FA202 - Bewerbungen bearbeiten und FA203 - Bewerbungen löschen*:
  Der Test gilt dann als erfolgreich, wenn die angelegte Bewerbung abgeändert und anschließend gelöscht werden konnte.
- *FA204 - Bewerbungsübersicht*:
  Der Test legt zwei neue Bewerbungen an und prüft, ob sich diese anschließend in der Bewerbungsübersicht befinden. Falls ja, gilt der Test als bestanden. Im aktuellen Prototypen lässt sich die Bewerbungsliste noch nicht nach einer bestimmten Ausschreibung filtern.

Replikation

Für die funktionalen Anforderungen der Replikation ist es nicht ohne Weiteres möglich Testfälle mit Selenium zu entwickeln. Um zu testen, ob die Replikation zwischen dem Gerät des Personalbenutzers und der internen CouchDB-Komponente funktioniert müsste die Netzwerkverbindung des Geräts des Personalbenutzers unterbrochen und wiederhergestellt werden. Für die Überprüfung, ob Daten anderer Personalbenutzer zum Gerät des Personalbentuzers repliziert werden, müsste auf die Geräte anderer Personalbenutzer zugegriffen und dort Daten manipuliert werden. Die Tests der Replikation wurden daher manuell durchgeführt. Die Testabläufe und die Resultate werden im Folgenden kurz dargelegt:

FA301 - Ausschreibungen und Bewerbungen replizieren

1. Personalbenutzer 1 (PB1) geht offline. Zu diesem Zeitpunkt befinden sich keinerlei Ausschreibungen oder Bewerbungen im System.
2. PB1 fügt Ausschreibung *Ausschreibung-FA301* hinzu.
3. PB1 fügt eine Bewerbung *FA301 Test* hinzu.
4. PB1 geht wieder online.

5. Die erstellten Daten werden nicht sofort zur internen CouchDB-Komponente hin repliziert. Der für die Replikation zwischen der internen CouchDB-Komponente und der CouchDB des Personalbenutzers zuständige Eintrag in der *_replicator*-Datenbank läuft auf einen Fehler (`_replication_state="error"`).
6. Löschen des `_replication_state`-Feldes und speichern des Eintrags in der *_replicator*-Datenbank führt dazu, dass die Replikation erneut angestoßen wird.
7. Die erstellten Daten werden zur internen CouchDB-Komponente repliziert und sind dort verfügbar.

FA302 - Replikation der Daten anderer Personalbenutzer

1. Personalbenutzer 2 (PB2) fügt eine Ausschreibung *Ausschreibung-FA302* hinzu.
2. Die von PB2 erstellte Ausschreibung wird zur internen CouchDB-Komponente repliziert und von dort aus weiter zu den lokalen Geräten anderer Personalbenutzer.
3. Die Ausschreibung *Ausschreibung-FA302* ist auf der lokalen CouchDB des PB1 vorhanden.

Die Replikation muss für diesen Test laufen. Falls Fehler auftreten, wie im oben beschriebenen Test FA301, muss die Replikation erneut angestoßen werden.

FA303 - Konflikte Zunächst wird ein Konflikt mit einer Ausschreibung erzeugt:

1. PB1 legt eine neue Ausschreibung *Ausschreibung-FA303* an.
2. Die Ausschreibung wird zur CouchDB des PB2 repliziert.
3. PB1 und PB2 gehen offline.
4. PB1 verändert den Titel der Ausschreibung zu *Ausschreibung-FA303 PB1*.
5. PB2 verändert den Titel der Ausschreibung zu *Ausschreibung-FA303 PB2*.
6. PB1 und PB2 gehen wieder online.
7. Die Replikation auf den Geräten von PB1 und PB2 muss neu angestoßen werden.
8. Die von beiden Personalbenutzern bearbeitete Ausschreibung ist konfligiert.

Anschließend wird der Konflikt gelöst:

1. PB1 ruft die Funktion *Konflikte bearbeiten* aus den Ausschreibungsdetails der konfligierten Ausschreibung auf.
2. PB1 wählt seine Änderungen an der Ausschreibung als richtig aus und ändert den Titel der Ausschreibung zu *Ausschreibung-FA303 PB1* ab.

3. PB1 ruft erneut die Funktion *Konflikte bearbeiten* auf.
4. PB1 löscht die konfligierte Version der Ausschreibung.
5. Der Konflikt ist anschließend gelöst.

Bewerber

Für die funktionalen Anforderungen für *Bewerber* befinden sich die erstellten Testsuiten im Ordner */tests/jobboerse*.

- *FA401 und FA402 - Ausschreibung in Jobbörse einsehbar*:
  Der Test legt eine Ausschreibung lokal an. Anschließend wird überprüft, ob die angelegte Ausschreibung in der Jobbörse vorhanden ist und ob diese die korrekten Informationen enthält. Die Replikation zwischen der CouchDB des Personalbenutzers und der CouchDB der internen Komponente sowie zwischen der CouchDB der internen und der externen Komponente muss laufen, damit der Test durchgeführt werden kann. Der Test gilt als erfolgreich, wenn die angelegte Ausschreibung in der Jobbörse zu sehen ist und die korrekten Informationen enthält.
- *FA403 - Bewerben*:
  Da das Speichern der Bewerbung noch nicht korrekt funktioniert, wurde noch kein Testfall für FA403 angelegt.

6. Bewertung

In diesem Kapitel werden die Ergebnisse dieser Arbeit kritisch bewertet. Zunächst wird geprüft, ob die funktionalen und nichtfunktionalen Anforderungen erreicht werden konnten.

Eine detaillierte Bewertung der definierten funktionalen und nichtfunktionalen Anforderungen befindet sich im Anhang A.

6.1. interne Komponente

Bei der entwickelten Software handelt es sich um einen Prototypen. Daher weist dieser noch nicht die komplette Funktionalität auf. Es fehlt beispielsweise die Funktionalität zum Filtern von Bewerbungen nach einer bestimmten Ausschreibung. Die Grundfunktionen zum Verwalten von Ausschreibungen und Bewerbungen sind jedoch vorhanden, sodass die Software auch im Offline-Betrieb eingesetzt werden konnte. Es konnten mit Selenium Testfälle entwickelt werden, die fast die gesamten definierten funktionalen Anforderungen abdecken.

Die Anforderungen an die Replikation sind nicht zufriedenstellend umgesetzt worden. Nach Offline-Phasen kam es zu Fehlern bei der Replikation, sodass diese durch Verändern des zuständigen Eintrags in der *_replicator*-Datenbank manuell neu gestartet werden musste. Für einen Produktiveinsatz ist ein solches Verhalten der Software nicht geeignet. Die bei der Replikation entstandenen Konflikte von einzelnen Dokumenten konnten durch den Benutzer erfolgreich Bearbeitet und gelöst werden.

Es wurde versucht, die definierten nichtfunktionalen Anforderungen weitestgehend umzusetzen, wobei diese nicht durch eine Evaluierung nachgewiesen wurde. Die Funktionalität der Software war während der Testphase gegeben, ebenso wie die Zuverlässigkeit, Benutzbarkeit, Analysierbarkeit und Installierbarkeit. Die Effizienz wies beim Zeitverhalten keine Probleme auf, das Verbrauchsverhalten konnte nicht abschließend geklärt werden, da im Prototypen die Funktionalität zum Hochladen von Binärdaten (Lebenslauf, Passfoto) noch nicht umgesetzt wurde. Erst nach Umsetzung dieser Funktionalität kann das Verbrauchsverhalten objektiv geprüft werden.

6.2. externe Komponente und Webservice

Die entwickelte externe Komponente und der Webservice sind ebenfalls Prototypen.

Die externe Komponente weist nahezu alle definierten funktionalen Anforderungen auf. Auch für die externe Komponente wurden mit Hilfe von Selenium Testfälle entwickelt, die die funktionalen Anforderungen abdecken. Für die Funktionalität der Bewerbungsabgabe konnte kein Testfall entwickelt werden, da diese Funktionalität noch nicht zufriedenstellend umgesetzt wurde. Die Abgabe der Bewerbung führt zu einem nicht erwartungskonformen Verhalten. Der Webservice speichert die Bewerbung bei der internen Komponente ab, jedoch wird eine falsche Antwort an die externe Komponente zurückgegeben. Dem Bewerber wird eine Fehlermeldung angezeigt, obwohl die Bewerbung erfolgreich abgespeichert wurde.

Auch bei der externen Komponente sind im Bereich der Replikation Fehler aufgetreten. Falls die Verbindung zur internen Komponente abbricht, entsteht bei der Replikation der Ausschreibungen ein Fehler und die Replikation muss wie bei der internen Komponente ebenfalls manuell neu gestartet werden. Dieses Verhalten ist für einen Produktiveinsatz nicht hinnehmbar. Außerdem werden alle Ausschreibungen zur externen Komponente hin repliziert, nicht nur die aktiven Ausschreibungen.

Die nichtfunktionalen Anforderungen wurden für die externe Komponente ebenfalls weitestgehend umgesetzt. Diese wurden auch hier jedoch nicht durch eine Evaluierung nachgewiesen. Die Funktionalität konnte im Bereich Sicherheit für die externe Komponente trotz des Einsatzes eines Webservices nicht zufriedenstellend umgesetzt werden. So befinden sich in der Datenbank der Jobbörse inaktive Ausschreibungen, die nicht für die Veröffentlichung bestimmt sind, und sind durch wenige Schritte für jeden einsehbar. Dies stellt ein Sicherheitsrisiko dar, da inaktive Ausschreibungen nur für definierte Personalbenutzer einsehbar sein sollen.

7. Fazit

Im letzten Kapitel wird die Arbeit noch einmal zusammengefasst und es wird ein Ausblick auf ein mögliches weiteres Vorgehen gegeben.

7.1. Zusammenfassung

Ziel dieser Arbeit war es, die Einsetzbarkeit einer charakteristischen NoSQL-Datenbank anhand eines typischen Anwendungsfalls zu untersuchen. Durch den Entwurf und die Realisierung eines Prototypen eines Ausschreibungs- und Bewerbermanagementsystems sollte geklärt werden, ob sich eine dokumentenorientierte Datenbank für den Einsatz in der Praxis eignet. Als Datenbank wurde CouchDB verwendet.

Durch den Einsatz des KansoJS-Frameworks wurden zwei Couchapps entwickelt: AppMan und die Jobbörse. Die beiden Couchapps benötigen zur Lauffähigkeit lediglich eine CouchDB-Instanz. Wegen Sicherheitsaspekten wurde für das Einreichen der Bewerbungen aus der Jobbörse heraus noch ein Webservice mit Java Jersey implementiert, der das Speichern der Bewerbungen regelt. Die Couchapps bieten die grundlegenden Funktionen zum Verwalten von Ausschreibungen und Bewerbungen, zur Anzeige der Ausschreibungen und zum Bewerben auf eine Ausschreibung.

Der Einsatz einer Couchapp für AppMan hat sich als geeignet herausgestellt. Durch die in CouchDB integrierten Replikationsmechanismen konnte AppMan für mehrere Benutzer auf ihren Endgeräten bereitgestellt werden. Die Benutzer konnten on- und offline an Ausschreibungen und Bewerbungen arbeiten. Es kam nach Offline-Phasen jedoch zu Fehlern bei der Replikation, sodass diese manuell neu gestartet werden musste.

Der Einsatz einer Couchapp für die Jobbörse ist als ungeeignet einzustufen. Da Bewerber auf die Couchapp ohne Authentifizierung zugreifen, musste die gesamte Datenbank der Jobbörse öffentlich zugänglich gemacht werden. Die Replikation der Ausschreibungen von AppMan zur Jobbörse hin konnte nur so umgesetzt werden, dass alle Ausschreibungen repliziert werden und nicht nur die aktiven Ausschreibungen. Somit befinden sich auch nicht für die Anzeige in der Jobbörse bestimmte Ausschreibungen in der Datenbank der Jobbörse und können durch wenige Schritte eingesehen werden, da die Datenbank für Lesezugriffe offen ist. Durch den Einsatz des Webservices konnte vermieden werden, dass die notwendigen

Benutzerinformationen für die Authentifizierung an der CouchDB der internen Komponente sich nicht in der Couchapp der Jobbörse befinden und frei zugänglich sind.

7.2. Ausblick

Im nächsten Schritt sollte die externe Komponente neu entworfen und implementiert werden. Die Jobbörse sollte nicht als Couchapp umgesetzt werden und die Datenbank der Jobbörse sollte nicht offen zugänglich sein. Der Webservice würde dann nicht mehr benötigt werden, da dieser nur die Funktion des Abspeicherns der Bewerbungen hat. Diese Funktionalität kann die neu entwickelte Jobbörse selbst übernehmen. Als Datenbank der externen Komponente kann weiterhin CouchDB verwendet werden. Es wäre dann weiterhin möglich, die in CouchDB integrierten Replikationsmechanismen zu benutzen und so die Ausschreibungen zur Jobbörse zu replizieren.

Anschließend sollte die Replikation sowohl für AppMan als auch für die Jobbörse überarbeitet werden. Für AppMan sollte eine Funktionalität entwickelt werden, die entweder automatisch auf Replikationsfehler reagiert oder den Benutzer über Replikationsfehler informiert und ihn zum Neustarten der Replikation auffordert. So könnte die Replikation neu gestartet werden, sobald wieder einer Verbindung zur internen Komponente hin besteht. Für die Jobbörse sollte die Replikation bei Fehlern automatisch neu gestartet werden, sodass sichergestellt wäre, dass die in sich in der Jobbörse befindenden Ausschreibungen aktuell sind.

Im Anschluss wird es notwendig sein, die entwickelten Testfälle anzupassen. Eine Automatisierung der Tests, beispielsweise durch den Einsatz von *Jenkins* sollte umgesetzt werden.

Darüber hinaus sollten die restlichen funktionalen Anforderungen umgesetzt werden.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] *Eric Brewer's CAP-Theorem Keynote*. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, Abruf: 14.12.2010

[2] *Jersey*. http://jersey.java.net/, Abruf: 12.08.2011

[3] *Kanso CouchApps*. http://kansojs.org/, Abruf: 19.07.2011

[4] *NoSQL Databases*. http://nosql-database.org/, Abruf: 16.11.2010

[5] *Selenium*. http://seleniumhq.org/, Abruf: 11.08.2011

[6] *TTD + Kanso?* http://groups.google.com/group/kanso/browse_thread/thread/7bdd74674a643784, Abruf: 11.08.2011

[7] ABADI, Daniel: *Problems with CAP, and Yahoo's little known NoSQL system*. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html. Version: 04 2010, Abruf: 10.01.2011

[8] AL-KILIDAR, H. ; COX, K. ; KITCHENHAM, B.: The use and usefulness of the ISO/IEC 9126 quality standard. In: *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005, S. 7 pp.

[9] ANDERSON, J.C. ; LEHNARDT, J. ; SLATER, N.: *CouchDB: The Definitive Guide*. O'Reilly Media, 2009 (O'Reilly Series). – ISBN 9780596155896

[10] BREWER, Eric: *Tweet über die Notwendigkeit, das CAP-Theorem zu ergänzen*. http://twitter.com/eric_brewer/status/26819094612. Version: 07 2000, Abruf: 12.01.2011

[11] CHANG, Fay ; DEAN, Jeffrey ; GHEMAWAT, Sanjay ; HSIEH, Wilson C. ; WALLACH, Deborah A. ; BURROWS, Mike ; CHANDRA, Tushar ; FIKES, Andrew ; GRUBER, Robert E.: Bigtable: A Distributed Storage System for Structured Data. In: *ACM Trans. Comput. Syst.* 26 (2008), Nr. 2, 1–26. http://doi.acm.org/10.1145/1365815.1365816. – ISSN 0734–2071

[12] COOPER, Brian F. ; RAMAKRISHNAN, Raghu ; SRIVASTAVA, Utkarsh ; SILBERSTEIN, Adam ; BOHANNON, Philip ; JACOBSEN, Hans-Arno ; PUZ, Nick ; WEAVER, Daniel ; YERNENI, Ramana: PNUTS: Yahoo!'s hosted data serving platform. In: *Proc. VLDB Endow.* 1 (2008), August, S. 1277–1288. – ISSN 2150–8097

[13] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: simplified data processing on large clusters. In: *Commun. ACM* 51 (2008), Nr. 1, 107–113. http://doi.acm.org/10.1145/1327452.1327492. – ISSN 0001–0782

[14] DECANDIA, Giuseppe ; HASTORUN, Deniz ; JAMPANI, Madan ; KAKULAPATI, Gunavardhan ; LAKSHMAN, Avinash ; PILCHIN, Alex ; SIVASUBRAMANIAN, Swaminathan ; VOSSHALL, Peter ; VOGELS, Werner: Dynamo: amazon's highly available key-value store. In: *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–591–5, S. 205–220

[15] DÖRR, Jörg ; GEISBERGER, Eva: *Qualitätsmodelle und Nichtfunktionale Anforderungen*. http://www.gi-muc-ak-req.de/daten/GI%20Vortrag%20Qualitaetsmodelle.pdf. Version: 11 2007, Abruf: 21.04.2011

[16] EDLICH, Stefan ; FRIEDLAND, Achim ; HAMPE, Jens ; BRAUER, Benjamin: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, 2010. – ISBN 9783446423558

[17] ELMASRI, R.A. ; NAVATHE, S.B.: *Grundlagen von Datenbanksystemen*. Pearson Studium, 2009 (Pearson Studium). – ISBN 9783868940121

[18] GHEMAWAT, Sanjay ; GOBIOFF, Howard ; LEUNG, Shun-Tak: The Google file system. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), October, 29–43. http://doi.acm.org/10.1145/1165389.945450. – ISSN 0163–5980

[19] GILBERT, Seth ; LYNCH, Nancy: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *SIGACT News* 33 (2002), June, 51–59. http://doi.acm.org/10.1145/564585.564601. – ISSN 0163–5700

[20] KEN SCHWABER, Jeff S.: *The Scrum Guide*. http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%202011.pdf. Version: 07 2011, Abruf: 21.08.2011

[21] KÖRBLER, Sigrid: *Parallel Computing- Systemarchitekturen und Methoden der Programmierung*. GRIN Verlag GmbH, 2008 (Akademische Schriftenreihe). – ISBN 9783640123629

[22] LEHNARDT, Jan: *What's new in CouchDB 1.0 - Part 4: Security'n stuff: Users, Authentication, Authorisation and Permissions*. http://blog.couchbase.com/whats-new-in-couchdb-1-0-part-4-securityn-stuff. Version: 08 2010, Abruf: 02.08.2011

[23] POMBERGER, Gustav ; DOBLER, Heinz: *Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung*. Pearson Studium, 2008. – ISBN 9783827372680

[24] PRITCHETT, Dan: BASE: An Acid Alternative. In: *Queue* 6 (2008), May, 48–55. http://doi.acm.org/10.1145/1394127.1394128. – ISSN 1542–7730

[25] SCHNELLE, Jochen: *NoSQL - Jenseits der relationalen Datenbanken*. http://www.pro-linux.de/artikel/2/1455/. Version: 08 2010, Abruf: 30.01.2011

[26] SOMMERVILLE, Ian: *Software Engineering*. Pearson Studium, 2001. – ISBN 9783827370013

[27] STONEBRAKER, Michael: *Clarifications on the CAP theorem and data-related errors*. http://voltdb.com/blog/clarifications-cap-theorem-and-data-related-errors. Version: 10 2010, Abruf: 11.01.2011

[28] STONEBRAKER, Michael: *Errors in Database Systems, Eventual Consistency, and the CAP Theorem*. http://cacm.acm.org/blogs/blog-cacm/83396. Version: 04 2010, Abruf: 11.01.2011

[29] STONEBRAKER, Michael ; CETINTEMEL, Ugur: „One Size Fits All“: An Idea Whose Time Has Come and Gone. In: *Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA : IEEE Computer Society, 2005 (ICDE '05). – ISBN 0–7695–2285–8, 2–11

[30] STONEBRAKER, Michael ; MADDEN, Samuel ; ABADI, Daniel J. ; HARIZOPOULOS, Stavros ; HACHEM, Nabil ; HELLAND, Pat: The end of an architectural era: (it's time for a complete rewrite). In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007. – ISBN 978–1–59593–649–3, S. 1150–1160

[31] VOGELS, Werner: *Eventually Consistent - Revisited*. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html. Version: 12 2008, Abruf: 12.01.2011

[32] ZERDICK, Axel ; PICOT, Arnold ; SCHRAPE, Klaus: *Die Internet-Ökonomie: Strategien für die digitale Wirtschaft. European Communication Council Report*. Springer, Berlin, 2001

A. Auflistung der Umsetzung der definierten Anforderungen

A.1. Bewertung der funktionalen Anforderungen

- **FA101 - Ausschreibungen erstellen**: Im Prototypen umgesetzt und Test angelegt.
- **FA102 - Ausschreibungen bearbeiten**: Im Prototypen umgesetzt und Test angelegt.
- **FA103 - Ausschreibungen löschen**: Im Prototypen umgesetzt und Test angelegt.
- **FA104 - Ausschreibungsübersicht**: Im Prototypen teilweise umgesetzt. Test für die Teilumsetzung angelegt. Die Ausschreibungsübersicht enthält alle Ausschreibungen mit Titel und Status, jedoch keinen Link zu den Bewerbungen einer Ausschreibung. Die Funktion, die alle Bewerbungen einer Ausschreibung anzeigt, wurde noch nicht umgesetzt.
- **FA201 - Bewerbungen erstellen**: Im Prototypen umgesetzt und Test angelegt.
- **FA202 - Bewerbungen bearbeiten**: Im Prototypen umgesetzt und Test angelegt.
- **FA203 - Bewerbungen löschen**: Im Prototypen umgesetzt und Test angelegt.
- **FA204 - Bewerbungsübersicht**: Im Prototypen teilweise umgesetzt und Test für Teilumsetzung angelegt. Der Prototyp enthält noch nicht die Funktion, mit der sich alle Bewerbungen einer bestimmten Ausschreibung anzeigen lassen. Daher existiert auch noch kein Filter in der Bewerbungsübersicht, der die Sicht auf eine bestimmte Ausschreibung einschränkt.
- **FA301 - Ausschreibungen und Bewerbungen replizieren**: Im Prototypen umgesetzt, allerdings noch fehleranfällig. Die Replikation muss nach einer Offline-Phase wieder manuell angestoßen werden. CouchDB versucht während der Offline-Phase weiterhin die Replikation durchzuführen, bricht jedoch nach einer definierten Anzahl von Versuchen ab. Die Erhöhung der Anzahl der Replikationsversuche stellt keine Lösung dar. Eine Möglichkeit wäre es, den Benutzer zu informieren, sobald eine Replikation wieder möglich ist und entweder den Benutzer die Replikation manuell starten lassen oder automatisch durchführen.

- **FA302 - Replikation der Daten anderer Personalbenutzer**: Im Prototypen umgesetzt, jedoch wie bei FA301 fehlerbehaftet. Auch hier kann die Replikation nach einer Offline-Phase auf einen Fehler laufen und muss anschließend manuell neu gestartet werden. Eine mögliche Lösung wurde bereits bei FA301 beschrieben.
- **FA303 - Konflikte**: Im Prototypen umgesetzt.
- **FA401 - Ausschreibungsliste in Jobbörse**: Im Prototypen umgesetzt und Test angelegt.
- **FA402 - Ausschreibungsdetails in Jobbörse**: Im Prototypen umgesetzt und Test angelegt.
- **FA403 - Bewerben**: Im Prototypen umgesetzt. Verhält sich nicht erwartungskonform. Die Bewerbung wird bei der internen Komponente gespeichert, jedoch liefert der Webservice eine fehlerhafte Antwort, sodass dem Bewerber ein Fehler angezeigt wird.

A.2. Bewertung der nichtfunktionalen Anforderungen

Funktionalität

- **Angemessenheit**: Die Funktionen von AppMan sowie der Jobbörse sind aus aufgabenorientierten Teilfunktionen aufgebaut.
- **Richtigkeit**: Die erstellten Testfälle prüfen auf die Richtigkeit der ausgegebenen Daten. Während der Testphase des Systems wurden keine falschen Daten ausgegeben.
- **Interoperabilität**: Der entstandene Prototyp von AppMan kann auf allen Geräten ausgeführt werden, auf denen einer der definierten Browser mit Javascript installiert ist.
- **Sicherheit**: Die Sicherheit der internen Komponente ist dadurch gegeben, dass nur autorisierte Benutzer auf die Couchapp und die Datenbank zugreifen und Daten manipulieren können.
  Da die Jobbörse als Couchapp umgesetzt wurde, muss die Datenbank der Jobbörse für alle lesbar sein. Dies stellt ein Sicherheitsrisiko dar. Es dürfen sich nur Daten in der Datenbank befinden, die keine vertraulichen Informationen enthalten. Dies ist im Prototypen nicht der Fall, da sich auch inaktive Ausschreibungen in der Datenbank befinden. Hier sollte eine andere Lösung gefunden werden. Eine Möglichkeit wäre es, die Jobbörse nicht als Couchapp zu realisieren, sondern als eine Webanwendung, die nicht komplett in der Datenbank liegt. Dadurch wäre die Datenbank der Jobbörse vor unberechtigtem Zugriff geschützt.

Zuverlässigkeit

- **Reife**: Während der Testphase konnten keine Lesefehler festgestellt werden.
- **Fehlertoleranz**: Die Replikationsmechanismen des AppMan-Prototypen sind fehleranfällig. So muss nach einer Offline-Phase eines Personalbenutzers die Replikation erneut manuell gestartet werden. Bedienfehler werden von der Anwendung abgefangen. Es lassen sich beispielsweise Ausschreibungen und Bewerbungen nur dann hinzufügen, wenn alle Pflichtfelder ausgefüllt wurden. Ein kompletter Absturz der Anwendung ist während der Testphase nicht aufgetreten.
- **Wiederherstellbarkeit**: Die Wiederherstellbarkeit ist durch ein Aktualisieren der Seite im Browser gegeben.

Benutzbarkeit

- **Verständlichkeit**: Das Layout von AppMan und der Jobbörse ist klar gegliedert. Im oberen Bereich befindet sich eine Navigationsleiste, die immer anzeigt, in welcher Funktion sich der Benutzer befindet. Der Benutzer kann durch die Vor- und Zurück-Buttons des Browsers durch die Funktionen navigieren. Die Namen der Funktionen sind sprechend, sodass die dahinter steckende Aufgabe sofort erkannt werden kann.
- **Erlernbarkeit**: Für die Jobbörse ist kein Schulungsaufwand notwendig. Der Bewerber kann sich sofort zurechtfinden und zügig eine Bewerbung absenden. Für AppMan ist für Benutzer mit Kenntnissen im Bereich Personalmanagement ebenfalls kein Schulungsaufwand erforderlich. Sie können direkt Ausschreibungen und Bewerbungen verwalten.
- **Bedienbarkeit**: Sämtliche Fehlermeldungen sind im Prototypen in verständlicher Sprache formuliert. Einzige Ausnahme ist der Hinweis auf ein nicht ausgefülltes Pflichtfeld. Die Fehlermeldung »Required field« wird in englischer Sprache angezeigt, da die Fehlermeldung vom KansoJS-Framework kommt. Eine Anpassung wurde im Prototypen nicht vorgenommen, da bei fehlenden Pflichtfeldeingaben zusätzlich dem Benutzer immer die Fehlermeldung »Bitte korrigieren Sie die angezeigten Fehler« oberhalb des Eingabeformulars angezeigt wird.
  Vor dem Löschen von Daten wird der Benutzer grundsätzlich durch einen Dialog zum Bestätigen aufgefordert.
- **Attraktivität**: Das Layout der Anwendung ist klar strukturiert und schlicht gehalten.

Effizienz

- **Zeitverhalten**: Während der Testphase haben keine Anfragen länger als eine Sekunde gedauert. Für AppMan dürfte sich dieses Verhalten auch im Produktiveinsatz nicht ändern, da die Benutzer auf ihren lokalen Geräten arbeiten und keine Latenz durch das Netzwerk vorliegt. Für die Jobbörse könnte sich das Zeitverhalten ändern, da die Zugriffe dann nicht über LAN, sondern über das Internet stattfinden würden.
  Im Prototypen können zu Bewerbungen keine Anhänge wie Lebenslauf oder Passfoto hinzugefügt werden. Das Zeitverhalten bei Anfragen nach Dokumenten mit Binärdaten wäre zu überprüfen, sobald die Funktionalität zum Hochladen von Dokumenten umgesetzt ist.
- **Verbrauchsverhalten**: Die Datenbank von AppMan kann auf den Geräten der Benutzer durch Speichern alter Revisionen von Dokumenten wachsen. Standardmäßig speichert CouchDB die letzten 1000 Revisionen eines Dokuments. Dieser Wert lässt sich einstellen. Es gilt hier einen günstigen Wert zu finden. Durch einen *_compact*-Befehl lässt sich die Datenbank verkleinern, indem die alten Versionen von Dokumenten gelöscht werden.

Änderbarkeit

- **Analysierbarkeit/Modifizierbarkeit**: Der Quellcode der beiden Couchapps von AppMan und der Jobbörse, sowie der Quellcode des Webservice sind übersichtlich aufgebaut. Die Funktionen haben sprechende Bezeichnungen. Ansatzpunkte für Änderungen lassen sich schnell finden. Es sollten dafür allerdings die Konzepte von CouchDB, dem KansoJS-Framework und Java-Jersey verinnerlicht worden sein.
- **Stabilität**: Durch die klare Trennung der show-, list- und update-Funktionen lassen sich schnell die zu ändernden Stellen identifizieren, falls Modifikationen durchgeführt werden. Die übrigen Funktionen bleiben von Änderungen unberührt, sodass Seiteneffekte nicht auftreten sollten.
- **Testbarkeit**: AppMan und die Jobbörse lassen sich durch die erstellten Testfälle mit Hilfe von Selenium testen. Die Replikationsmechanismen werden derzeit nur manuell getestet. Es sollten Tests für die Replikation entwickelt und automatisch während der Programmlaufzeit auf den lokalen Geräten der Benutzer ausgeführt werden.

Übertragbarkeit

- **Anpassungsfähigkeit**: AppMan ist auf allen in Abschnitt 3.2.2 definierten Browsern mit Javascript lauffähig. Wichtig ist, dass auf dem jeweiligen Endgerät eine Instanz von CouchDB ab Version 1.1 installiert ist.
- **Installierbarkeit**: AppMan wird über Replikation auf den Endgeräten der Benutzer installiert. Es sind nur Grundkenntnisse der Replikation mittels CouchDB notwendig, um die Installation von AppMan auf den Endgeräten durchführen zu können.

B. CouchDB intern

B.1. Benutzer anlegen

Mit Hilfe von Futon (`http://localhost:5984/_utils`) wird ein Benutzer der Rolle `admin` angelegt. Für das Anlegen eines Benutzers der Rolle `human_resources` kann das Listing B.1 herangezogen werden. Dabei ist darauf zu achten, dass das Passwort SHA-verschlüsselt angelegt wird. Für die Generierung von `password_sha` und `salt` kann das in Listing B.2 dargestellte Perl-Skript verwendet werden.

```
1 {
2   "_id": "org.couchdb.user:hr_system",
3   "name": "hr_system",
4   "salt": "4166c592502c75cbf9829b628367ba94805096a4",
5   "password_sha": "e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4",
6   "type": "user",
7   "roles": ["human_resources"]
8 }
```

Listing B.1: »hr_system.json«: Datei zum Anlegen eines Benutzers mit der Rolle »human_resources«

Der Benutzer wird mit Hilfe von curl[1] angelegt:

```
curl -X PUT "http://admin:secret@localhost:5984/_users/org.couchdb.user
%3Ahr_system" -d@hr_system.json
```

B.2. Erreichbarkeit im Netzwerk

In Futon unter »Configuration« sind folgende Einstellungen vorzunehmen:
`bind_address = 0.0.0.0` (Erreichbarkeit über Netzwerk)
`require_valid_user = true` (Nur autorisierte Benutzer dürfen auf CouchDB zugreifen)

[1] http://curl.haxx.se/

B.3. AppMan Datenbankberechtigungen

Die Datenbankberechtigungen der AppMan-Datenbank sind wie folgt zu setzen:

- *Admins*:
  Names: `[]`
  Roles: `["_admin"]`
- *Reader*:
  Names: `[]`
  Roles: `["human_resources"]`

```
1  # Usage: couchdb_password [password [salt]]  or provide password on
       STDIN.
2
3  use Digest::SHA1 qw(sha1);
4
5  my $password;
6
7  if(@ARGV) {
8    $password = shift;
9  } else {
10   $password = <>;
11   chomp $password;
12 }
13
14 my $salt;
15 if(@ARGV) {
16   my $unsalted = shift;
17   $salt = pack("H*",$unsalted);
18 } else {
19   # Get some bytes from random
20   open(my $urandom, '<', '/dev/urandom') or die $!;
21   binmode($urandom);
22   my $random_bytes;
23   read($urandom,$random_bytes,16) == 16 or die $!;
24   close($urandom) or die $!;
25
26   $salt = unpack('H*',sha1($random_bytes));
27 }
28
29 print "salt = $salt\n";
30
31 my $password_sha = sha1($password,$salt);
32 print "password_sha = ".unpack('H*',$password_sha)."\n";
```

Listing B.2: Perl-Skript zur Generierung von `password_sha` und `salt`

C. Inhalt der Beiliegenden CD

Die beiliegende CD weist folgende Struktur auf:

- *pdf*:
  In diesem Ordner befindet sich die Bachelorarbeit in PDF-Form.
- *code*:
  Der Ordner enthält die Quellcodes der Prototypen von AppMan, der Jobbörse und des Webservices. In dem Ordner befindet sich eine Datei `readme.txt`, in der die zur Installation notwendigen Schritte und Voraussetzungen beschrieben sind.
  Im Unterordnet *tests* befinden sich die entwickelten Selenium-Tests (vgl. Abschnitt 5.5)

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 22. August 2011	
Ort, Datum	Unterschrift