Diplom Thesis

# Implementation of a distributed application using the document-oriented database CouchDB

## An outliner as a replicable distributed system

Lena Herrmann

Beuth Hochschule für Technik Berlin
University of Applied Sciences

Upstream-Agile GmbH

Department VI Computer Science and Media
Academic Program of Media Informatics, Software as point of main effort
Matriculation number 720742

Tutor: Prof. Dr. Stefan Edlich
Reviewer: Prof. Dr. Frank Steyer

Submitted on July 14, 2010

*«For most of mankind's history we have lived in very small communities in which we knew everybody and everybody knew us. But gradually [...] our communities became too large and disparate [...], and our technologies were unequal to the task of drawing us together. But that is changing.*
*Interactivity. Many-to-many communications. Pervasive networking. These are cumbersome new terms for elements in our lives so fundamental that, before we lost them, we didn't even know to have names for them.»*

(Douglas Adams, 1999)

# Abstract

Modern web browsers and mobile devices are capable of running complex applications that allow collaboration and data exchange between their users. Laptops and mobile phones, however, cannot be expected to keep their internet connections alive at all times. This problem can be sidestepped using data replication, which means that data are regularly synchronised and kept consistent. This thesis describes the drafting and prototypical development of a JavaScript application that uses the document-oriented database CouchDB to form distributed outliner software. Outliners can be used to record thoughts or concepts in a hierarchically structured manner. Apart from categorising the system to be developed and analysing possible approaches, the thesis will also examine the technologies used. Special focus lies on CouchDB with its built-in master-master replication and its ability to implement complex applications without the use of middleware. The final application runs locally in the browser and is therefore also usable when off-line. Conflicts are resolved when the system is synchronised, sometimes steered by user input. The thesis also evaluates the applicability of CouchDB in distributed applications with particular regard to the use case at hand.

# Table of Contents

# 1 Introduction

## 1.1 Motivation

> We're at the dawn of a new age on the web, or maybe on a horizon, or a precipice; the metaphor doesn't matter, the point is, things are changing. Specifically, as browsers become more and more powerful, as developers we're given new opportunities to rethink how we architect web applications. [Qui09]

The Internet is playing an ever more important role in our everyday lives. The share of people that frequently use the Internet in Germany is currently at 72 per cent and is still growing. Among 14 to 19-year-olds, only 3 per cent [Ini10, p. 10] never uses the Internet. Along with the rising number of Internet users, the use of web applications for co-operation, communication and data exchange becomes increasingly normal. This also means that web applications have to be able to cope with ever larger numbers of simultaneous users. The web browser is becoming an increasingly important application platform [Pau05, p. 16]. Hardly any piece of software went through such enormous evolution as the web browser [GJ05]. This gave rise to new possibilities for web applications, since they raise the bar for the user-friendliness, usability and availability.

A further trend is the growing spread of mobile devices [Ini10, p. 61]. In [Kot99, p. 7], the author predicts that the number of devices able to access the Internet will grow rapidly, thanks to the growing number of services on it, and fast developments in computer technology. Mobile devices today include mostly laptops and mobile phones. Their connectivity, however, is often less reliable than is the case with stationary devices. Long disconnected periods are common. Other barriers include high latency and limited bandwidth [Guy98].

So there is a high need for technology that can satisfy aforementioned co-operation and data exchange requirements. The technology should make it possible to implement systems that scale within a larger scope. At the same time they should vastly reduce the need for continuous connectivity. One solution for this kind of problem is data replication. [Guy98] sketches this as follows:

> Copies of data are placed at various hosts in the overall network, [...] often local to users. In the extreme, a data replica is stored on each mobile computer that desires to access that data, so all user data access is local.

The difficulty with this is that it is hard to synchronise data regularly and to keep data consistent. Consistency has to be monitored by the system responsible for replication. The goals of such

a system are high availability and control over one's own data. Such a solution can at the same time meet the high privacy standards that users expect from modern web applications [Kra09a, Kra09b]. In systems that perform replication, users can decide for themselves with whom they share their data. The usage of a correspondingly implemented system is at least temporarily independent from central servers. Possible failure of the network or its individual nodes is factored in beforehand.

This thesis describes the design and realisation of software using the CouchDB database. CouchDB allows the implementation of applications that scale „up" as well as „down": applications should support distribution over any number of nodes in order to warrant availability and performance. They should also be available on mobile devices and allow synchronisation of user data [Leh10].

## 1.2 Structure of the thesis

Initially, the central question of the thesis will be formulated (chapter 2). The goal of this thesis is to find a well-founded answer to this question. In order to do so, chapter 3 will provide a categorisation of the system to be developed and an analysis of the relevant solutions.

Chapter 4 is dedicated to the CouchDB database. The database is categorised theoretically in section 4.1. Its implementation details are explained in section 4.2. Chapter 5 presents further technologies that have influenced the application's implementation. Subsequently, the web technologies that were employed are listed and described (section 5.1), as well as cloud computing (section 5.2) and other support tools (section 5.3).

The requirements of the application are specified in chapter 6. Chapter 7 sketches the structure of the application, weighing different design alternatives against each other. The technical details of the final system are drafted in chapter 8 and supplemented with source code excerpts in the appendix. The practical part of the thesis is concluded by a manual in chapter 9. An evaluation of the results will be included in the final chapter 10.

## 1.3 Markup

In order to increase the readability of this thesis, some terms have been emphasised. Technical terminology and names of involved technologies will be printed in *italics* when they are mentioned for the first time. They will not be emphasised a second time if they re-appear later in the text. The explanations for abbreviations can be found in the abbreviation table in appendix A.1. Terms from the source code will be highlighted using the `Courier` font. Source code excerpts are printed in

`typewriter`-style. Blocks of source code have a grey background and an outline. Block quotes are indented on the left and right sides. In-line quotes are put in quotes.

# 2 Task description

In the introduction, the current (as of June 2010) changes in the circumstances and raised expectations from applications have been pointed out. More people are using ever growing systems on ever more mobile devices and are thus raising the bar for their usability and availability.

At the same time, the development of technologies that help to satisfy these requirements is making progress. In the realms of database systems, the last decade has seen the introduction of a new „movement“ called *NoSQL* [Str98]. NoSQL is an umbrella term designating a group of non-relational database systems or *key-value stores*. A common trait to these systems is that they usually do not, in contrast to relational databases, need a fixed table structure. Their strong point is their distributability, making them particularly suitable for scaling up. The advantages of traditional database systems, especially continuous consistency, are traded for better availability or partition tolerance (see section 4.1.2).

These new database systems have been developed for specific use cases. It is not the aim of this thesis to perform an exhaustive comparison of NoSQL databases. Rather, it will discuss the usability of a certain NoSQL-database system by means of a real example. Is it possible to develop functional software after a thorough analysis using the technology of choice?

The document-oriented database CouchDB [Apa09c] was already shortly introduced in the introduction. In an application that was implemented using CouchDB, it is possible to completely omit any *middleware*. *Master-master-replication* is a core functionality, making CouchDB particularly suitable for distributed operation. CouchDB applications run directly in the web browser, minimising the amount of programs necessary for using the application. This also means that the system can be used in a very high number of devices. Why this very database system was chosen will be explained in detail in the chapters 3 and 4.

As a study case for the use of CouchDB an *outliner* was chosen. Such software allows hierarchic structuring and notation of thoughts or concepts. The model used here is the program OmniOutliner [Omn10], an offline desktop program for Mac OS X. The goal is to prototypically create an outliner with similar, but limited, functionality and to analyse in this way the usability of CouchDB.

The application conceived here differs from its model on one point: it should also allow collaborative editing of documents distributed over networks, even when the user disconnects from the Internet between times. The application created for this thesis will run locally in the browser and will also be available when offline. Via the Internet, the data will be editable by several simultaneous users.

This thesis will investigate if CouchDB is suitable to develop distributed applications. In order to verify this, a prototype of the distributed outliner will be designed and developed.

# 3 Analysis

In chapter 2, the requirements for a system were roughed in. This thesis will discuss the design and implementation of this application as well as the assessment of the results. In the next chapter, the formulation of the problems to be solved by the application and its scientific classification will be examined more closely. Then, different solutions will be analysed to find out how to implement such a system most easily. Alternative solutions will be discussed.

## 3.1 Requirements of an outliner

Before the chosen solution is separated from other possibilities, the properties and requirements of the application will be established more precisely.

### 3.1.1 Definition

An outliner is described by Wikipedia as a mixture of a free-form database and a text editor [Wik10a]:

> An outliner is a computer program that allows one to organize text into discrete sections that are related in a tree structure or hierarchy. Text may be collapsed into a node, or expanded and edited. [Wik10c]

Some outliners also allow formatting entries and embedding several media. There is a multitude of implementations of such programs.

One of the most valued implementations [Mac05] is the commercial software OmniOutliner [Omn10] (screenshot see fig. A.2). This program is developed by „Omni Group“ for the Mac OS X operating system. Alongside conventional outliner properties, it also boasts some additional, special features. OmniOutliner will serve as a model in the development of the application, even though not all its features can be implemented in the prototype.

### 3.1.2 Usage examples

Much like with text editors, the usage examples of an outliner are manifold. In the next few paragraphs, we will present only a few of the possible usage scenarios. Some of those are taken from

[Wik10a], others were provided by the author herself. Some design decisions in the development of the software were made based on these scenarios.

The original goal of an outliner is defined by [Wik10a] as the card-index box that is so popular amongst humanists:

> Text of varying length (e.g. quotes) are saved after category in an indexed file and are thus available. [...] In this way, a sizable archive of important text extracts can quickly be built.

A commonly quoted usage case of outliners is in writing literary, journalistic or scientific texts. Initially, the outlines of the story can be entered in chapters and scenes. Next, the user can add details to the branches of the tree structure. Later on, parts of the text can be moved by moving the nodes in the tree around. This is a lot harder to do with a simple text editor or word processor.

Outliners can also be used to save a larger amount of short text excerpts. These may include tasks, ideas, minutes, logbooks or shopping lists.

## 3.2 Requirements of distributed systems

This section will start with a definition of distributed systems. It will be demonstrated that the application to be developed is indeed a distributed system. In the following sections, the choice for CouchDB for the implementation of such a system will be accounted for.

An often-cited definition is found in [Tan07, Chap. 1.1]:

> A distributed system is an accumulation of independent computers that appears as one coherent system to its users.

This description may refer to any number of computers, however many. For instance, in [Cou05, Chap. 1.2] and in [Ben04, p. 6], the Internet and World Wide Web itself are also termed distributed systems. There are also wider definitions, in which not only the physically distributed hardware, but also logically distributed applications are referred to as a distributed system [Ben04, p. 5]. After [Cou05, Chap. 1.1], the fundamental characteristic of a distributed system is only to communicate through *message passing*. Generally speaking:

> A distributed system is a system which operates robustly over a wide network. [And10, Chap. 2]

Accordingly, a system can be called distributed if it can be separated into multiple, autonomous components, i.e. each has to be a complete system in its own. The individual subsystems can be

very heterogenous with regard to operating systems and hardware. The connection type does not matter either. Moreover, users as well as programs that use the system should have the impression that they are dealing with one single system. The challenge in developing distributed systems is to define how the system's components will co-operate in a manner that is transparent to the user. The system's inner structure should not be visible to the user. Furthermore, it should be possible to work consistently and uniformly with the system, irrespective of time and location of access [Tan07, Chap. 1.1].

According to [Cou05, Chap. 1.1] and [Tan07, Chap. 1.1], the following obstacles should be considered when designing distributed systems:

**Concurrency:** Components of a system are executing program code concurrently when they are accessing common resources, reading from and writing to them at the same time. A distributed system therefore needs to use algorithms that are specially designed for dealing with concurrent processes.

**No global clock:** The clocks of the subsystems cannot be kept synchronised. Therefore it is not possible to solve editing conflicts with timestamps. In its stead, MVCC (Multi Version Concurrency Control) often uses *Vector clocks* [Lam78] or other unambiguous identification mechanisms such as *UUIDs (Universally Unique Identifier)* [Lea05].

**Subsystem failure:** The failure of individual components or network interruption should not hamper the operation of the entire system. The subsystems need to function independently, even when the connection between them is interrupted temporarily or for a longer period, or if one of them breaks down.

In chapter 4.1, this problem is examined in detail.

## 3.3 Requirements of groupware

According to [Ell89], groupware systems are ...

> ... computer-based systems that support two or more users engaged in a common task, and that provide an interface to a shared environment. These systems frequently require fine-granularity sharing of data and fast response times.

As with any other software project, distributed systems should be designed with not only the technical eventualities in mind. The very real demands of the target audience have to be taken into account as well. For instance, the specific questions and problems that users have with groupware should be considered, and what that means for the design, development and schooling.

CouchDB, the database used in this thesis, is often compared to Lotus Notes. CouchDB's creator, Damien Katz, has been a developer on the Notes team and has admitted having been heavily influenced by it in developing CouchDB [Scho8]. Notes resembles CouchDB to the extent that a database in Notes consists of a collection of semi-structured documents, organised by Views [Kaw88, Chap. 2]. According to [Kaw88], Lotus Notes is a communication system that allows groups to edit and share information such as texts and notes:"

> The system supports groups of people working on shared sets of documents and is intended for use in a personal computer network environment in which the database servers are „rarely connected". [Kaw88, Chap. 1]

Lotus Notes and the application being developed here are analogous in that the latter will perform at least some of Notes' tasks. Additionally, the synchronisation of documents is also achieved by database replication. Scientific insights in the usage of Lotus Notes are therefore of interest in designing the application.

Several studies have examined the success of adopting Lotus Notes for collaboration between distributed groups.

[Van96] analyses the influence of Notes on the collaboration within a major insurance company. Even though its employees were very happy with the product, there was no perceptible improvement in their collaboration efforts. The study concludes that a system has to accurately fit the target audience and that basic schooling in the new technology plays a central role.

[Kar95]'s author states that users that have never used groupware before usually approach it as they would a familiar (single-user) program:

> [The] findings suggest that when people neither understand nor appreciate the cooperative nature of groupware, it will be interpreted as an instance of some more familiar technology and used accordingly. This can result in counter-productive and uncooperative practice and low levels of use. [Kar95, p. 4]

These findings are endorsed by another research:

> The findings suggest that where people's mental models do not understand or appreciate the collaborative nature of groupware, such technologies will be interpreted and used as if they were more familiar technologies, such as personal, stand-alone software (e.g., a spreadsheet or word processing program). [Orl92, p. 1]

If the software's structure differs from the group's company culture, it is unlikely to be used in a meaningful way. Groupware should be adapted to the group's existing processes if it is to improve the workflow. If the tasks of the software include dealing with conflicts, support for the team's own conflict solving strategies is one of the keys to success [Mono1].

It still remains to be determined that software whose users are not still familiar with its central characteristics has -on one hand- to be adapted to that very audience. This is especially difficult in the concept phase, because the application prototype is developed for no clearly defined user base. The task of limiting the target group remains for a later development phase. On the other hand, the training and documentation for the users have to receive extra care. The work done here should help people get used to collaborating more closely with the aid of software.

## 3.4 Various approaches

In this section various approaches for the conceptual formulation are discussed. Advantages and disadvantages of existing solutions will be examined and a preferred alternative will be lifted out.

The problem that needs solving is: How can documents be edited by several users at the same time, even when some of them may disconnect from the Internet for a longer period of time? How can the system handle the occurring editing conflicts automatically or present them in a meaningful way for the user to be solved?

### 3.4.1 Manual data exchange

The most trivial operation is manual synchronisation. Documents are exchanged directly between individual users, e.g. per email or FTP, and edited concurrently. Users have to merge different versions by hand and the results have to be exchanged again. Details are easily lost in the process.

Some web services offer network file systems that allow automatic synchronisation of data. For example, Dropbox [Dro10] allows to synchronise whole folders and files between different computers. If a conflict occurs in the process both versions are saved as separate files. It is the user's task to resolve any conflicts.

### 3.4.2 Real-time text editors

Another approach gaining in popularity is the use of centralised collaboration systems on the Internet. Users can usually use such web applications to work on documents collectively and in real-time. Examples of such platforms are Etherpad [Fou10], Google Docs [Bel10], and more recently, Google Wave [Goo10]. The latter foregrounds the communicative aspect but also allows users to simultaneously edit longer texts. Google Docs on the other hand shares more characteristics of word processing software.

Desktop programs such as the text editor SubEthaEdit [The10] show their advantages only with an active network connection. SubEthaEdit allows users to find each other locally or over the Internet using the Bonjour protocol. They can then invite each other to work together on a document.

Both approaches presented in the last paragraphs have one thing in common. Either they can only be used with a working Internet connection, or conflict handling is only supported if all clients are connected to the server simultaneously. Otherwise, conflicting versions have to be merged manually.

Some of the programs presented here allow live tracking of other authors' keystrokes. This is not valued by everyone. In [Man09] working with Google Wave is described as „[like] talking to an overcurious mind reader". The awareness that others are watching while you think is distracting and may interfere with the work. In cases where users work longer on the same document, they may develop a feeling of ownership over the text. Under such circumstances, watching live how other people make modifications may be quite unpalatable [Edl95].

Since the application to be developed should also allow working with longer texts, we can set the live-typing feature aside.

### 3.4.3 Versioning system

In the realms of software development, the use of versioning systems such as Subversion [Apa10b] or Git [Cha10] has been widely adopted. Such systems collect changes to documents including author and time stamp and saves them in individual *commits*. These revisions can be restored at a later stage. Moreover, several changes by different users to a single file can be merged automatically by the system.

Such a system is very convenient for plain-text files. That is why such programs are mainly used in software development. Most implementations do not have an interface that is accessible for the less technically inclined. The implementation of an application with a Git back-end would be worth considering.

### 3.4.4 Databases

A database approach, especially the free-form data storage presented in chapter 2, jumps to mind. Some databases or key-value stores support master-master replication and save data on the hard disk. These generally come into question for the solution of the task at hand.

The document-oriented database CouchDB differs from the competition in that it comes with its own web server. This does not only output the available data, but also allows running JavaScript files

stored in the database. This allows the whole application to run inside the database. The resulting program is therefore automatically available on any computer running CouchDB, provided there is a browser installed. None of the other database systems examined provided such functionality.

The open, relational databases PostgreSQL [Groa] and MySQL [Cora] can be configured for master-master replication between two masters. PostgreSQL boasts a multitude of plug-ins [Grob] with which it is possible -amongst others- to set up a master-master configuration. For this same purpose, MySQL uses the MySQL cluster technique [Cord], allowing master-master replication on a shared-nothing architecture [Sto86, Chap. 1]. [Max06] describes how to implement replication for several nodes.

The key-value store Riak [Bas10] also has an web interface and saves its files distributedly. In this case, however, there is no peer-to-peer replication like in CouchDB, but rather a sort of auto-balancing to increase availability and performance in larger systems. MongoDB [10g10] has limited support for master-master replication and allows *eventual consistency* (cf. section 4.1.2), which would be useful in a distributed system. But it lacks one of CouchDB's most interesting features: to run application code directly from within the browser. Both technologies are therefore less suitable than CouchDB for the purpose of implementing the planned application.

Replication and clustering in the systems discussed above are more or less similar to the functionality offered by CouchDB Lounge. This is discussed in section 8.10. Automatic conflict highlighting are not supported by these systems either. In order to implement replication within the application logic, none of these solutions prove satisfactory.

In direct comparison, CouchDB emerges as the best solution for the issues described above, thanks to its ability to perform master-master replication, because it includes conflict resolving and a convenient consistency model, and because applications can be delivered straight from the database. In the next section, the solution chosen here will be examined more closely.

## 3.5 Description of the chosen solution

The application is developed as a local, network-ready program. The data are saved into a CouchDB database; the application logic is executed by the browser as client-side JavaScript code. The functional range of the outliner will allow at least creating and deleting outlines. Text can be entered line after line and can be indented our outdented.

The exchange of data as well as of the application is performed by master-master replication embedded in CouchDB. This means that the data can be changed by all computers that have a copy. A server runs another CouchDB instance. Replication to this server is automatically done when a user adds something to a document. Further users can download the entire application in the CouchDB installation onto their computers. If an Internet connection exists, updates to

the data are automatically replicated to the server and from there distributed to other users that happen to be on-line. The application notifies the user as soon as changes are on hand. The user can then view these changes by reloading the page.

The main task lies in dealing with editing conflicts that have been caused by simultaneous editing of data. New or changed lines have to be inserted or updated, especially when a user replicates after having been offline for a longer period. CouchDB in itself can highlight occurring conflicts. Whether conflicts are to be shown for review or to be solved automatically is a decision that has to be made in the concept phase. Since the time is limited, not all possibly occurring conflicts will be taken into consideration. Rather, some types of conflicts will be examined.

Moreover, deployment and scaling options with the CouchDB Lounge clustering framework and Amazon Elastic Compute Cloud (Amazon EC2) will be discussed. The application will be deployed as a prototype; possibilities for implementation of a highly available server will be described.

# 4 CouchDB - a database for distributed systems

Now that the desired approach has been motivated and briefly sketched, the technologies and concepts involved in the realisation of the project will be presented here.

This chapter's main focus is a detailed presentation of CouchDB, the database of choice. *Apache CouchDB („Cluster Of Unreliable Commodity Hardware Data Base")* is a document-oriented database system [Apa09c]. CouchDB has been under development as an open-source project since 2005. In November 2008, it acquired the status of an official Apache Software Foundation [Apa09d] project. CouchDB was originally written in C++, but since 2005, most of its development was done in the programming language *Erlang* [Eri10]. Erlang was designed in the late eighties for real-time systems such as telephone switchboards and consequently, it therefore excels in fault-tolerance, parallelism and stability [Len09]. The *Erlang/OTP*-system (*The Open Telecom Platform*) includes the programming language Erlang, libraries and its run-time environment.

In the first part of this chapter, the basic theoretic principles of CouchDB will be explained. The second part focuses on describing the database from an application developer's point of view.

## 4.1 Theoretical classification

In order to understand the motivation for the development of CouchDB, a brief description of the recent history of database systems and a classification of CouchDB with regard to the three-level architecture (see section 4.1.1) is in order. Afterwards, the CAP theorem and the handling of CouchDB with concurrent transactions will be presented and the „RESTfulness" of the web interface will be examined. Furthermore, the CouchDB solution will be compared to traditional relational databases. The individual aspects of CouchDB's structure, properties and features will be touched upon and elaborately described throughout the course of this chapter.

### 4.1.1 Classification of the database architecture

In 1975, the *Standards Planning and Requirements Committee (SPARC)*, of the *American National Standards Institute (ANSI)* conceived a model describing requirements of the structure of a

database system ([Häo1]). This model was called the *ANSI-SPARC architecture* or the *three-level architecture*.

For the users of a database, changes on a lower level, for instance in hardware or internal file systems, should not have any perceivable influence [Cod83, p. 377]. If a database has been built along the rules of the three-level architecture, the user's view of the database and its technical realisation are separated. Its internal data handling becomes clear to its users.

[Mato7, Chap. 1.2] divides the three levels as follows:

**External levels/user's views:** Parts of the database are available to different user groups. This allows users to enter derivative data without ever having to disclose the original data.

**Conceptual level:** This level contains the description of all of the database's data structures, i.e. the data types and links. How these data are saved is irrelevant. The conceptual level depends strongly on the database design and the choice of data model.

**Internal level:** This level contains the characteristics of the physical data handling, i.e. the distribution of the database over several computers or hard disks, or indexes for improved access speed.

This architecture can be employed irrespective whether the database model is relational, object-oriented, network-like, or anything else.

CouchDB can also be described along the ANSI-SPARC lines. The external levels correspond to the CouchDB views. The conceptual level is the representation of documents as JSON objects, i.e. the overall view of the database. The internal level is the way data is handled. In CouchDB, this is done with the aid of a *B+-Tree*. These three levels will be described in detail later in this chapter.

### 4.1.2 The CAP theorem

*CAP* stands for *Consistency, Availability and Partition Tolerance*. In modelling distributed systems, the term *partition tolerance* is of high importance [Varo9, p. 62]. It implies that subsystems should be able to continue functioning autonomously even after physical disconnection or data loss. An operation on the system should be carried out successfully even when one or some of the components are unavailable. A distributed system should meet two more requirements: *consistency* means that all components see the same data at the same time. *Availability* means that the system responds to every request, with a defined and low latency. Unless otherwise specified, the following is based on [Gilo2, p. 1-4] and [Andio, Chap. 2].

With the *CAP theorem*, professor Brewer of the University of California formulates that even though the three qualities consistency, availability and partition tolerance are expected from web

services, in reality only two of the three requirements can be met [Bre00]. Since partition tolerance is indispensable to a distributed system, the decision between consistency and availability has to be made in the design phase.

In traditional relational database systems (*RDBMS*), consistency can usually be taken for granted, since they usually meet the *ACID*-standards (*Atomicity, Consistency, Isolation, Durability*). In this context, complete consistency means that a read operation following a write operation will read the updated data. This is called *one-copy serialisability* or *strong consistency* [Mos09]. In RDBMS, this is enforced by locking mechanisms (cf. section 4.1.3). In a distributed system that distributes the data over several computers, consistency is harder to achieve. Several non-relational databases that have been developed in recent years such as Bigtable [Cha06], Hypertable [Hyp09], HBase [Apa10i], MongoDB [10g10] and MemcacheDB [Mem09] opt for absolute consistency at the expense of availability.

In contrast, other projects such as Cassandra [Apa09a], Dynamo [Vog07], Project Voldemort [Pro] and CouchDB focus on the availability. They fall back on a number of strategies to implement consistency. Thanks to a so-called *consensus algorithm* such as *Paxos* [Lam01], all components handle conflicts in the same way without negotiating [Pea80]. Another solution is to use *time clocks* that allow the sorting of data in distributed system.

CouchDB's strategy is different from most other strategies because it provides *eventual consistency* next to availability and partition tolerance. This means that *inconsistencies* may occur in a short time window between read and write access on certain data. Within this time period it is possible that outdated data are being output. Shortly afterwards all read operations will again return the result of the write operation. This means that if errors occur or latency is high, data records may temporarily be inconsistent. The data consistency is eventually achieved:

> The storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. [Vog09]

Eventual consistency is not a practical concept for every domain (e.g. in finance). If user inputs strongly build on each other, data are interdependent and users temporarily work with outdated data, errors can accumulate, compromising the consistency of the whole system. Therefore CouchDB also allows the implementation of systems with strong consistency. For many applications, however, it is of higher importance that updates can be successfully applied at any time without having to block access to the database (e.g. for a social network or the use case at hand).

### 4.1.3 Transactions and concurrency

Every database system that is set up to cater more than one user has to deal with the question of concurrency. The question to be answered here is what happens when two users try changing the

same value at the same time. „At the same time“ need not be exactly at the same point in time. An operation that encompasses reading, changing and saving data and that takes a certain while can cause a concurrency issue when another user starts a similar action in the same time span and overwrites the data the first user has changed in the meantime. The task of the database is to serialise these operations: both simultaneous operations should produce the same result as if they had been performed after each other [Var09, p. 57]. Even if the intervals are very short and conflicts seem unlikely, they have to be taken into consideration in drafting the design and architecture of the application [Vog09].

The locking mechanism that RDBMS usually employ puts a lock on the resources that are being changed. Other operations have to wait for the lock to be lifted, after which they will receive the exclusive access to the data. Locking is a very performant approach for non-distributed database systems [Var09, p. 57]. Operations need not wait only because they are concurrent. On the other hand, locking is usually accompanied by some overhead and hard to implement if the transaction's participants are distributed. There are protocols that allow locking and unlocking also in distributed systems [Ber81], but they are slow and not suitable for the application to be developed.

For concurrency control CouchDB therefore uses an implementation of *optimistic concurrency*, the so-called *Multi-Version Concurrency Control (MVCC)*:

> MVCC takes snapshots of the contents of the database, and only these snapshots are visible to a transaction. Once the transaction is complete, the modifications that were done are applied to the newest copy of the relation and the snapshot is discarded. This means that in any given time multiple different versions of the same data exists. [Par10, Chap. 2.1]

MVCC brings advantages for availability and performance, in return users may sometimes come across inconsistent data. There are several ways to implement this mechanism. One way involves time stamps or vector clocks to determine the modification times of certain transactions and therefore the validity of an update. The update can then be admitted or rejected. This is described more closely in [Lam78]. Instead of assigning vectors to the objects, CouchDB provides a UUID and a revision number. Moreover, CouchDB uses the *copy-on-write* technique that makes two processes consecutively write to a log, rather than directly accessing an entry. Felix Hupfeld describes a *log-based storage mechanism* in [Hup04, Chap. 2.2]. Also see [Joh09]:

> The basic organization of a log structured storage system is, as the name implies, a log - that is, an append-only sequence of data entries. Whenever you have new data to write, instead of finding a location for it on disk, you simply append it to the end of the log.

This very mechanism is used in CouchDB when data or document versions are saved as revisions

in the B+-tree mechanism. This will be examined more closely in section 4.2.7.

The drawback of MVCC is that the application logic has to process extra levels of complexity that would be handled quietly by an RDBMS. CouchDB has solutions for this which will be explained in section 4.2.1. In [Var09, p. 60], however, this drawback is seen as an advantage: Applications that use RDBMSs are developed in such a way that bottlenecks may arise when the throughput increases. These bottlenecks can not be corrected afterwards. MVCC urges developers to cleanly deal with possible conflict sources from the start. This lessens the risk of losing performance as access operations increase. Even under high stress, the server can be maxed out without having to wait for blocked resources. Requests are simply executed simultaneously.

### 4.1.4 Replication

In general, replication aids the synchronisation of data between the components of a distributed system. For CouchDB, this means that the contents of a database are transferred to another; Documents that exist in both databases will be reconciled. Replication can be broken down in different dimensions:

#### 4.1.4.1 *Conservative* vs. *optimistic* replication

Probably one of the most important design decisions to be made in replication is how the system treats concurrent updates: how should the system behave when several replicas want to update the same data at the same time? This question was already discussed in section 4.1.3. [Guy98, Chap. 2] mentions two approaches: With conservative or -as [Sai05] likes to call it- pessimistic replication, the consistency has to be verified before every update. Concurrent updates will be refused. For the use case at hand this strategy proves not useful, since the replicas are not permanently connected. Instead, CouchDB uses optimistic replication [Sai05, p. 43].

Optimistic replication means accepting changes to replicated records without the need for mutual agreement between the replicas and without the need for a locking mechanism. The conflicts that naturally occur as a consequence of this are the system's concern. CouchDB supports this using automatic conflict recognition and highlighting. This is discussed in section 4.2.1.

#### 4.1.4.2 *Client-server* vs. *peer-to-peer* model

According to [Guy98, Chap. 2], replication by the client-server model first notifies the server of an update. The server will then distribute the update to all clients. This makes the system simpler, but it also means that the system depends on the server never failing. Replication by the peer-to-peer model, however, allows replicas to notify about updates among themselves. This way updates can

be distributed more quickly: connectivity can be used as soon as it is available, irrespective of which components are connected. A CouchDB installation can perform bi-directional master-master replication with any other CouchDB instance. It is therefore suitable for both models. The strategy to be implemented depends on the application at hand.

#### 4.1.4.3 Notification strategies

There are two notification strategies [Guy98, Chap. 2]: immediate propagation and periodic reconciliation. Immediate propagation involves that other replicas are notified directly after an update. Periodic reconciliation means that the replicas are regularly notified about any updates. This usually happens at a convenient time. CouchDB supports both notification strategies. Since the system to be developed here also allows offline operation, it should support some sort of periodic reconciliation, since immediate propagation would produce an error if certain nodes are offline.

#### 4.1.4.4 *Eager* vs. *lazy* replication

Furthermore, Jim Gray conducts a division in *eager replication* and *lazy replication* in [Gra96, Chap. 1]. The former involves the immediate updating of all replicas. This means that they always have to be on-line. This is not very useful for the application at hand in this thesis:

> Replication in systems communicating over wide-area networks or running on mobile devices should tolerate high communication latencies. In such systems it is common to only use asynchronous replication algorithms. [Hup09, P. IV]

Accordingly, the replication of CouchDB is „lazy“: updates are being distributed asynchronously. Using CouchDB's replication algorithm, one instance of CouchDB may request the changes from another if both are connected.

### 4.1.5 HTTP interface

[KT04] recommends the use of the *REST* architecture in decentralised and independent distributed systems. REST-compliant or *RESTful* are such interfaces that can transfer data over HTTP [Fie96] when every resource can be accessed through its own URL [Fie00]. Other features of the protocol include statelessness, well-defined operations and the option to request various representations of a resource.

Not all *APIs (Application Programming Interfaces)* called RESTful (meaning that they are compatible with the REST architecture) are justly categorised as such. The classification of HTTP-based

APIs made by NORD Software Consulting [NOR] distinguishes different levels of RESTfulness. Most APIs belong to the categories „HTTP-based type I“, „HTTP-based type II“ or „REST“. APIs that belong to the first two categories violate one of the REST rules since client and server are firmly tied together by the interface design.

This also applies to CouchDB's API, even though [Apa09b] says it is RESTful. According to [NOR10] a RESTful API need not include any differentiated documentation; rather, it suffices to have a list of available media types and fields. The CouchDB API cannot be filed under the HTTP-based type II category, since it uses a generic media type that does not make the resources self-explanatory. However, since the API uses properly named method names, it can be categorised under HTTP-based type I.

[And10, Chap. 4] confirms the limited RESTfulness of certain API sections. The replication API for example takes after traditional *Remote Procedure Calls*. An exclusively loose coupling as stipulated by the REST architecture is not absolutely necessary for a database API [NOR10]. Nevertheless, the CouchDB API can also be made HTTP-based type II and REST conform with the aid of the show and list functions staged in section 4.2.5.

### 4.1.6 CouchDB versus relational database systems

The relational data model was first described scientifically in the early seventies by Edgar Codd [Cod83]. IBM and Oracle implemented the first database systems based on this model in the late seventies. Databases would still run on single mainframes that were not connected to a network. These mainframes regularly had to carry out larger operations that required lots of database logic [Leh08]. Checking the consistency of the data for every such operation did not pose any problems since the operations were carried out one after another [And10, Chap. 2]. Data were protected against loss by physical backups, replication did not emerge until later. RDBMS are optimised for such use.

#### 4.1.6.1 Replication and conflict handling

In the early eighties, relational database systems also established in other fields of application. The scenarios today are different than back then: in the realm of Internet applications servers usually have to process several individual requests simultaneously. The ratio between the complexity of requests and the number of accesses has changed heavily. Distributed systems also raise the question of how to implement replication and conflict resolution strategies.

In spite of the disadvantages, the relational database MySQL [Cora] is by far the most popular database for the implementation of web applications [Alf08, p. 18]. Replication in MySQL is structured after the principle of *log replay* [Corc]. In a master-master set-up, however, concurrent

and contradictory write operations occur on a regular basis. If the database cannot resolve inconsistencies in a pre-defined manner, database functions or application logic have to be developed to handle these conflicts [Max06]. CouchDB's replication strategy, on the other hand, is incremental. Even if the connection is interrupted during the replication process, the data remains in a consistent condition. This is exemplified in section 4.2.3. CouchDB's ability to deal with concurrent updates and arising conflicts is presented in section 4.2.1 and illustrated in section 4.2.7 by means of a description of CouchDB's implementation.

#### 4.1.6.2 No middleware

CouchDB was developed to live up to modern requirements of a database for web applications - „[it is] built *of* the web" [Kap07]. The concepts that have been implemented using CouchDB are not new. [Ass98, p. 39] describes how the first generation of static web applications advanced to the currently prevalent model: the development of database and application is done separately, interfaces such as *CGI* [W3C09] take care of the communication with the users. It is hereby necessary to use middleware in order to...

> ... make the transition between systems as easy and transparent as possible for developers, based on existing interfaces of established web servers and database systems. [Ass98, p. 24]

About the future of so-called Internet databases, [Ass98, p. 39] predicted that they will not only allow access to their data, but that they will also have to integrate the interaction with the application system. Such middleware can be omitted in a CouchDB application. Rather, an application can communicate directly with the database. This is achieved through the HTTP API, as presented in section 4.2.2. This way, stable applications can be developed with comparatively little effort.

#### 4.1.6.3 Schemalessness

Many problems that arise in designing a modern web application (cf. chapter 3 and 1.1) include unpredictable user behaviour and input by a great quantity of users with a great quantity of data [Var09, p. 36]. For instance, this encompasses searching the Internet, plotting graphs in social networks, analysing buying habits etc. These tasks are often accompanied by confusing data structures that cannot be defined and modelled accurately beforehand. According to [Bar09], such data are not easily mapped as relational data:

> RDBMSs are designed to model very highly and statically structured data which has been modeled with mathematical precision. Data and designs that do not meet these

> criteria, such as data designed for direct human consumption, lose the advantages of the relational model, and result in poorer maintainability than with less stringent models.

Document-oriented databases consist of a set of unrelated documents. All data for a document are contained in the document itself:

> In fact, there are no tables, rows, columns or relationships in a document-oriented database at all. This means that they are schema-free; no strict schema needs to be defined in advance of actually using the database. If a document needs to add a new field, it can simply include that field, without adversely affecting other documents in the database. [Len09]

CouchDB's schemalessness is indeed relevant for the implementation of the prototype but not central in designing it. However, this may change in future versions of the application, when the outliner will contain several columns containing different data types and media (cf. chapter 10).

#### 4.1.6.4 Unique identifiers

In relational databases, table rows are usually identified with a primary key whose value is often determined by an *auto-increment*-function [Len09]. Such keys, however, are only unique within the database and the table in which they were generated. If a value is added to two different databases that have to be synchronised at a later moment, this will produce a conflict [Corb]. Every CouchDB document is assigned a *UUID (Universally Unique Identifier)* on creation. Conflicts become statistically impossible this way. An overview of documents in CouchDB can be found in section 4.2.1.

#### 4.1.6.5 Views instead of joins

One of the most important differences between document-oriented and relational databases is the way data is requested from the database. Since CouchDB does not support primary or foreign keys it is not possible to retrieve associated data sets through joins. Instead, *views* can help to create a relation between any documents in the database without having to define such relations when designing the database. Views are discussed under section 4.2.6. Joins are also often made redundant by modeling data into documents.

## 4.2 Description

In this section CouchDB's features and some of its implementation details will be presented. In a CouchDB database system any number of databases containing documents can be created. The administration interface *Futon* can be reached from a web browser under the URL `http://localhost:5984/_utils`. Figure A.4 displays a screenshot of a CouchDB instance and its databases. Figure A.5 shows the contents of such a database. Database operations can either be executed programmatically or using this interface.

CouchDB allows a sophisticated implementation of access control with user administration and rights management. This will not be discussed due to limitations to this thesis. Likewise, some database functions as well as some parts of the HTTP API will be omitted. The information in the next few sections are explained in [And10], [Apa09b] and [Len09] unless otherwise stated.

### 4.2.1 Documents and conflict handling

CouchDB documents save actual data records as JSON objects (see section 5.1.3.1). A document can contain any number of fields of any size that have a name that is unique within the document. Binary data can also be attached to a document. A document is tagged with a unique ID (`_id`) that is either specified upon creation or generated as a UUID that is almost 100 per cent mathematically unique. Figure A.6 shows an example of such a document.

Another piece of meta data that the document contains is a revision number, called `_rev`. When a document is changed in any way the document is not changed; rather, the entire document is cloned into a new version that is identical except for the changes made and the new revision number. The database thus contains a complete version history of every document. Using this data storage method, CouchDB implements a lockless and optimistic document update model (see section 4.1.3).

A document is changed as follows: the client loads the document, changes it and stores it in the database along with its ID and revision number. If meanwhile a second client made and saved changes to the same document, the former client will receive an error message (HTTP status code 409: „conflict“). In order to resolve this conflict, the current version of the document needs to be reloaded and modified before a second attempt can be made at storing the data. This either fails or passes completely. Under no circumstances will an incomplete document be saved.

Deleting a document works similarly. Before saving, the most recent version of the document must be available. The actual deleting of the document is done by appending the field `_deleted=true`. Deleted documents are saved just like older versions, until the database is compacted.

Conflicts may still occur when a replicated data set is changed independently. However, CouchDB

features automatic conflict detection, simplifying conflict handling. When merging the documents that have changed in both copies they are automatically labelled as conflicting, much like version control systems would. The system adds an array called `_conflicts` containing all conflicting revisions. Using a deterministic algorithm, one of the revisions will be saved as the „winner version“. Only this version is shown in the views. The other version is saved to the document history so it can be accessed. It's the application's or the user's task to solve these conflicts; this can be done by merging, undoing or accepting the changes. It is irrelevant on which replica this is done, as long as the solved conflict is communicated to all copies using replication.

### 4.2.2 HTTP interface

The data from a CouchDB database can be read and written using an API. This API is addressable through the HTTP methods GET, POST and PUT. The data is returned as a JSON object. Since JSON and HTTP are both supported by many programming languages and libraries, almost any application can perform database operations on CouchDB.

In the process of writing this thesis, a test suite for CouchDB's JavaScript HTTP API was produced. It will be explained in section 8.8.3.

The CouchDB API has a set of functions at its disposal that can be subdivided into four categories according to [And10, Chap. 4]. For every of these categories, some applications and an example for requesting data through the command-line tool *cURL* will be given.

#### 4.2.2.1 Server API

If a simple GET request is sent to the CouchDB server's URI, it will return the CouchDB installation's version number: `curl http://localhost:5984/` returns the JSON object `{"couchdb":"Welcome","version":"0.11.0b902479"}`. Other functions may request a list of all databases or set certain configuration options. User identification is also done by the server: users can identify themselves towards the server or log out.

#### 4.2.2.2 Database API

The command `curl -X PUT http://localhost:5984/exampledb` is used to create a database. If successful, the server returns `{"ok":true}`. If not (e.g. because a database with that name already exists), an error is returned. Likewise, databases can be deleted, compacted or information about them may be requested. A new document can be created using `curl -X POST http://localhost:5984/exampledb -d '{"foo":"bar"}'`. CouchDB then returns the ID and the revision of the document

created: `{"ok":true,"id":"6651b95e15b411dbab3","rev":1-303d5e305201766b21a4274173681d6"}`. Using the PUT method rather than POST, the ID of the document may be specified.

#### 4.2.2.3 Document API

A GET request on the document's URI (`http://localhost:5984/exampledb/6651b95e15b411dbab3`) will return the document created above. The document can be updated using a PUT request, whereby the ID, the entire document with the changes and the latest revision of the document have to be included. If the revision is lacking or incorrect, the update will fail.

#### 4.2.2.4 Replication API

The command `curl -X POST http://127.0.0.1:5984/_replicate -d '{"source":"exampledb","target":"exampledb-replica"}'` starts the replication between two databases. If a bi-directional replication is desired, the command has to be invoked a second time, swapping the source and target. The parameters will be examined more closely in the next section.

### 4.2.3 Replication

In order that a replica receives note of updates to other replicas, the replication may be started with the option `continuous=true`. This mechanism is called *continuous replication*. It keeps the HTTP connection open so that every change to a document may be replicated instantly. Continuous replication has to be restarted explicitly when a network connection becomes available. After restarting a server, continuous replication will not automatically be restarted.

Only data that was changed since the last replication will be replicated. This means that the process is incremental. If the replication should fail due to node failure or network issues, the next replication will pick up where it left off. Replication may be filtered using so-called *filter functions*, so that only certain documents are replicated.

### 4.2.4 Change notifications

CouchDB databases have a sequence number that is incremented with every write access to the database. The changes between every two sequence numbers are also saved. This allows easy detection of differences between databases when the replication is continued after a pause. The differences are not only used internally for replication; they can also be used for application logic in the shape of the *changes feed*.

With the aid of a changes feed, the database may be monitored for changes. The request `http://localhost:5984/exampledb/_changes` returns a JSON object containing the current sequence number as well as a list of all changes in the database:

```
{"results": [
    {"seq":1,"id":"test","changes":[{"rev":"1-aaa8e2a031bca334f50b48b6682fb486"}]},
    {"seq":2,"id":"test2","changes":[{"rev":"1-e18422e6a82d0f2157d74b5dcf457997"}]}
  ],
"last_seq":2}
```

Different parameters may be supplied to the changes feed. `since=n`, for example, will request only those changes that were made after a certain sequence number. Using `feed=continuous`, the feed may, much like the replication, be configured to return an entry after every change to the database. The aforementioned *filter functions* make it possible to only return documents with certain changes.

### 4.2.5 Applications with CouchDB

Documents may also include program code that CouchDB can execute. Such documents are called *design documents*. Usually every application served by CouchDB has a design document assigned to it.

A design document is structured after fixed specifications. The following is a list of documents typically included in a design document:

**_id:** Contains the prefix `_design/` and the name of the design document or the application, e.g. `_design/doingnotes`.

**_rev:** Replication and conflict resolution will treat design documents just like any other document. Therefore, they contain a revision number.

**_attachments:** This field contains program code that may be executed client-side in the browser. The application logic for a CouchDB application can be contained within.

**_views:** Analogous to `list`, `show` and `filter` fields, this field contains functions with which the database contents may be returned filtered, structured and/or modified. This code is executed server-side by the database.

Figure A.7 shows a screenshot of a design document opened in Futon.

### 4.2.6 Views

Relational databases show the relations between data by saving „equal“ data into one table, and associated data in another, interlinked by primary and foreign keys. Based on these relations, dynamic requests can retrieve aggregated data sets. CouchDB chooses a contrary approach. On database level no associations are made. Links between documents may still be drawn when the data are already available. In order to so, requests of such data and their results are stored statically into the database. They have no influence on the documents in the database. These requests are saved as indexes called *views*.

Views are stored in design documents and contain JavaScript functions that build requests with the aid of *MapReduce* [Yano7]. All documents are then supplied as an argument to a *map function* that then decides if it should disclose the entire document or individual fields. A view containing a *reduce function* will use this function to aggregate the results. Listing 4.1 shows a view with which all documents are scanned for a field `kind` with the value `Outline`. If a document contains such a field, an entry containing the document ID as key and the document as value will be appended to the resulting JSON object.

```
1 function(doc) {
2   if(doc.kind == 'Outline') {
3     emit(doc._id, doc);
4   }
5 }
```

Listing 4.1: View: map function to show all outlines

If this view is stored under the name `outlines` in a design document called `designname`, it can be retrieved using an HTTP GET request from the URI `http://localhost:5984/exampledb/_design/designname/outlines`. Views are compiled when they are first requested and then stored like normal documents in a B+ tree along with their index. If further documents are added that should be contained in the view's result, they will automatically be added to the stored view when it is next requested.

Access to a view can be restricted through keys and key ranges. The request `http://localhost:5984/exampledb/design/designname/outlines?key="5"` only returns the outline with the ID `5`. The request `http://localhost:5984/exampledb/design/designname/outlines?startkey="2"&endkey="7"` returns all outlines whose ID lies between `2` and `7`. There are a number of additional parameters with which the results may be stated more precisely. The key or key ranges are mapped directly onto the database engine, making access more efficient. This will be explained in the next section.

### 4.2.7 Implementation

A CouchDB database is always in a consistent state, even if the CouchDB server were to fail in the middle of the storage process. This can be attributed to the database engine used, which will be characterised in this section. The presentation is based on [Ho,08] and [And10, Chap. G].

The data structure used is a *B+ tree*, a variation of the *B-tree*. A B+ tree is aimed at saving and quickly reproducing huge amounts of data. It guarantees access times below 10 milliseconds, even for „extremely large data sets“ [Bay08]. B+ trees grow breadth-wise; even if they contain several million entries, they usually have a single-digit depth. This is advantageous since CouchDB commits its data to hard disks where every traversal step is a time-consuming process.

A B+ tree is a completely balanced search tree, in which data are saved sorted by keys [Bay08]. A node can contain several key values. Every node refers to several child nodes. In CouchDB the actual data are exclusively saved in the leaves. In B+ trees the documents as well as the views are indexed. In doing so, for every database and every view its own B+ tree is created.

Only one simultaneous write operation per database is permitted. All write operations are serialised. Read operations can be achieved concurrently to each other and to write operations. This means that databases and views can be retrieved and refreshed simultaneously. Since CouchDB saves its data in append-only mode, the database file contains a version history of all documents. MVCC can therefore be implemented effectively.

As described in section 4.2.1, changing a document doesn't so much overwrite it as create a new revision of it. Afterwards, the nodes of the B+ tree are updated one after another, until all refer to the location of the latest version of the document. This is done from the leaf of the tree containing the document, all the way up to the root node. The latter is thus modified at the end of every write operation. If a read operation still is referring to the old root node, it will refer to an outdated but consistent snapshot of the database. Old revisions of the documents will only be deleted when the user initiates a compaction. Therefore, read operations can still complete a request, even when a new version of the document is being made.

When a B+ tree is committed to the hard disk, the changes are always appended to the end of the file, reducing the write access times to the hard disk. Furthermore, it prevents unforeseen interruptions of the saving process or power outages from corrupting the index:

> If a crash occurs while updating a view index, the incomplete index updates are simply lost and rebuilt incrementally from its previously committed state. [Apa09b]

# 5 Technical fundamentals

This chapter will present the individual web technologies that have been used in the application. Next, the fundamentals of cloud computing will be discussed since it played an important role in the deployment of the application. The final section discusses the methods and means that have played a supporting role in the application's development.

## 5.1 Web technologies

The interactive possibilities offered by the user interface of the application to be developed are few. The use of laborious view frameworks is therefore not necessary. The application can be implemented with comparatively simple technologies, frameworks and programming languages, all of which will be presented individually.

### 5.1.1 CouchApp

The application to be developed will be implemented as a so-called CouchApp [Che10]. The CouchApp project makes a set of supporting components available that make the task of developing a standalone application with CouchDB lighter. The design requires every user to install her own offline CouchDB instance on her device in which the application will run. This means that the application is also available when the user is disconnected from the Internet.

In [Kat10] Damien Katz, CouchDB's creator, describes CouchApps as follows:

> CouchDB, being an HTTP server, can host applications directly, so you can write applications and forward the HTML, CSS, and JavaScript through CouchDB. When you point your browser at it, the browser comes alive and starts the JavaScript. It becomes interactive as you query and update the server and everything. When everything is served from CouchDB, that's a CouchApp and you can replicate it around, just like the data.

A CouchApp can be deployed onto a local computer, a mobile phone, a local server or even in the cloud. In any case the application is accessed from the browser. Replication allows the updating of both data and the program itself.

After installing CouchApp (as explained in section 9.1.2), the framework for a new sample application may be generated by issuing the command `couchapp generate example-couchapp` on the command-line interface. This will create a folder `example-couchapp` with a defined directory structure (see Fig. 5.1). This directory corresponds to a CouchDB design document as described in section 4.2.5.

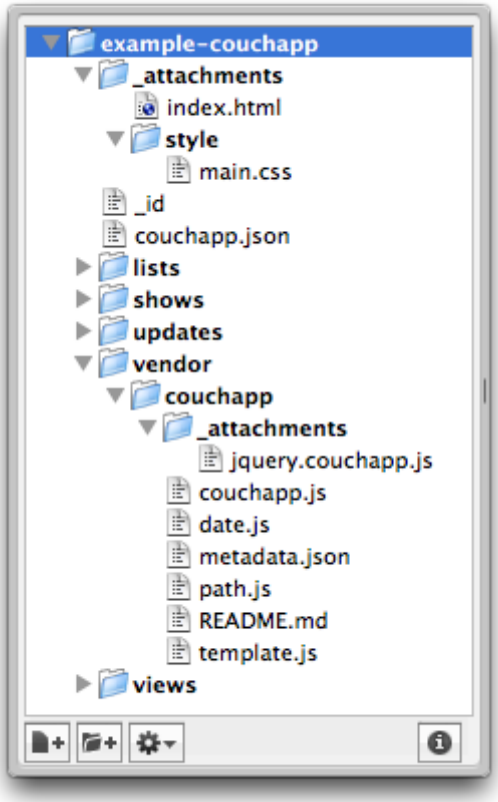


Abbildung 5.1: Generated sample CouchApp

The directory `_attachments` contains all JavaScript, HTML and CSS files that are required for the application logic and the presentation. A default HTML start page and style sheet have been automatically generated. The directories `lists`, `show`, `views` and `filters` contain CouchDB's list, show and filter functions resp. the views. The `vendor` directory is home to external libraries that are needed in the development process.

The deployment settings are saved in the file `.couchapprc`, which is formatted as follows:

```
1 {
2   "env": {
3     "default": {
4       "db": "http://user:password@localhost:5984/example-couchapp-dev"
5     },
```

```
6     "production": {
7       "db": "http:///user:password@example.com/example-couchapp"
8     }
9   }
10 }
```

Listing 5.1: Couchapp: .couchapprc

It is necessary to specify the CouchDB administrator's user name and password for the development and in some cases also the production environment. This information can be omitted if user credentials are not set. The command `couchapp push` or `couchapp push production` copies the contents of the CouchDB directory into the CouchDB instance.

The URL scheme of a CouchApp will be described in section 8.2. Section 9.1.2 will cover how to deploy the application developed in this thesis with the aid of CouchApp.

### 5.1.2 HTML5

*HTML (Hypertext Markup Language)* is the hypertext format commonly used on the world wide web. *HTML5* [Hic10a] is a specification developed by the W3C that should replace the existing HTML and XHTML standards. HTML5 is currently widely supported in recent versions of the most popular browsers, with one exception for Microsoft Internet Explorer [Smi09]. This limitation hampers the development of most web applications since they have to be compatible with browsers that are older or not standard-compliant.

Due to limitations described detailedly in section 8.5.2, the application is tailored to be used with the browser Firefox, version 3.6 or higher (see release notes [Moz10b]). This allows the application to use HTML5 functionality, even for its core workings. For example, apart from many improvements, HTML5 also allows defining custom data attributes. This technique is used in this thesis and will therefore briefly be introduced.

According to the specification [Hic10b], a custom data attribute is an attribute without namespace whose name starts with the string `data-`, having at least one lowercase character and no uppercase characters after the hyphen. Custom data attributes can store private data for the site or the application when there are no fitting attributes or elements to do so. The attributes are intended to be used by the page's own scripts, not as publicly available meta data. Every HTML element can have any number of custom data attributes.

### 5.1.3 JavaScript

JavaScript is a versatile scripting language whose affiliation with the web browser makes it „one of the most popular programming languages in the world“ [Cro08, p. 2]. The *DOM (Document Object Model)* [W3C05] allows the browser to directly access JavaScript objects in an HTML document. JavaScript is a dynamic, object-oriented programming language that follows a prototype-oriented paradigm.

JavaScript is most widely used to enhance the user experience. Websites are „enriched“ with client-sided functionality, but are also accessible without JavaScript. JavaScript can also be used on the server side as a fully fledged programming language. Unlike Java or C, JavaScript implements object-orientation with prototypes instead of classes.

The next section will discuss the JavaScript component JSON, the AJAX concept as well as the jQuery library.

#### 5.1.3.1 JSON

*JSON (JavaScript Object Notation)* is the most popular form for exchanging information in JavaScript [Che07, Chap. 2]. JSON is a subset of JavaScript [Cro06], which means that it is valid JavaScript in itself. Not all data types that exist in JavaScript can be used in JSON, only the data types `Object`, `Array`, `String`, `Number`, `Boolean` and `Null` are used. Almost all common programming languages, however, have equivalent data types. This makes JSON also suitable for exchanging data with other languages [Cro10].

Examples of JSON can be found in listings 5.5 and 8.12.

#### 5.1.3.2 AJAX

*AJAX* is an acronym for *Asynchronous JavaScript And XML*. It does not indicate a software package or framework. Jesse James Garrett, who coined the term in 2005 in [Gar05], describes AJAX as

> [...] an approach — a way of thinking about the architecture of web applications using certain technologies.

According to [Gar05], AJAX comprises several technologies from the realms of web development: the representation with XHTML and CSS, dynamic interaction through the DOM, data exchange and manipulation using XML and XSLT (or another data exchange format such as JSON), asynchronous data requests with XMLHttpRequest, and finally JavaScript to link all these components together. This set-up allows the asynchronous transfer of data between browser and server.

A web application is usually built in such a way that an action in the user interface provokes an HTTP-request from the web server. The server calculates the result with the aid of application logic and/or database requests and returns an HTML page to the client (cf. [Garo5] and figure 5.2).

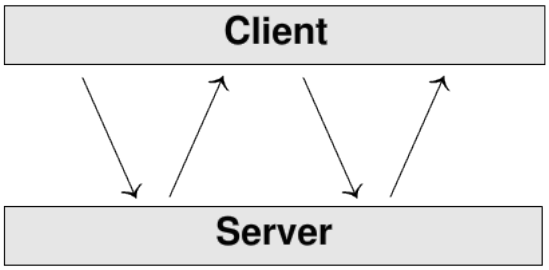


Abbildung 5.2: Synchronous interaction flow chart of a traditional web application, from [Rayo7]

The use of the technologies mentioned above allows the browser to load data from the server asynchronously and in the background, without changing the display and the behaviour of the page that is open in the browser. If the server sends an answer only those parts of the site are changed for which new data is available - there is no need to reload the page (figure 5.3). The advantage here is that the page does not have to be rendered again with every request which also means less data have to be transferred.

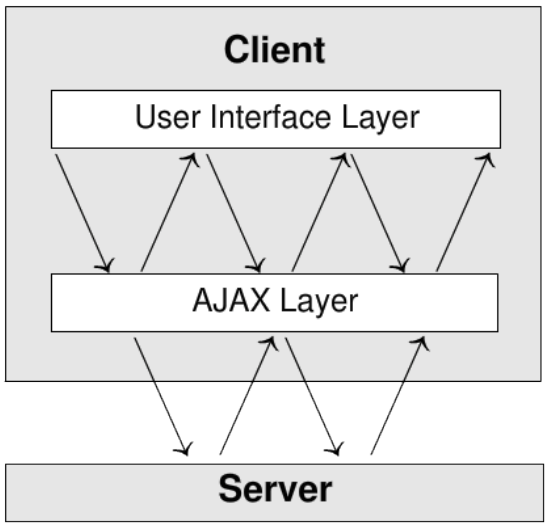


Abbildung 5.3: Asynchronous interaction flow chart of AJAX applications, from [Rayo7]

#### 5.1.3.3 jQuery

A JavaScript library is a collection of pre-defined JavaScript functions that simplify the development of web applications. The application's user interface is developed with the aid of the JavaScript library jQuery version 1.4 [jQu10]:

> jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. [jQu10]

JQuery is an abstraction of basic JavaScript functionality. In particular, it simplifies DOM traversing. At the same time it doesn't impose a structure for the application.

In a non-representative poll conducted in April 2010 by the organisation *Web Directions* among professional web developers and designers, some 78% of the inquired indicated that they used jQuery when writing JavaScript code. [All10]

### 5.1.4 Sammy.js

*Sammy.js* [Qui10a] is a routing framework that builds on jQuery. Sammy allows developers to define *routes* to which a certain behaviour can be associated. The controller part of applications can thus be implemented event-based and RESTful. The use of Sammy in the application is described in section 8.2.

#### 5.1.4.1 Routes

A Sammy route consists of the following elements:

**A path:** The anchor part in a URL, thus the part after the URL hash (#). It can be defined as a string or a regular expression in the specification of the route.

**A method:** One of the HTML methods `GET`, `POST`, `PUT` or `DELETE`.

**A callback:** A function that is called when the route is executed with a certain method.

Even when the path is the same, different callbacks can be set according to the method used. For example, the resource outline may be shown using the route „`get('#/outlines/:id')`“ and updated using „`put('#/outlines/:id')`“. Thanks to the usage of the URL anchor, client-side applications may run on one page and still react to the browser's „Back“ button.

It is Sammy's task to parse parameters from the path. In this way, IDs or slugs can be filtered from the path. Every string within the path that starts with a colon „:“ will be turned into a parameter.

Routes with the methods `POST`, `PUT` and `DELETE` are only executed by submitted HTML forms. In run-time, the `submit` method is overwritten for all forms and hooked to Sammy. When the form is submitted, a route is sought that corresponds to the form's path and the method. If such a route is found its callback function is executed.

Routes can also be hooked onto user-defined events that are then executed by the application. For example, when the Sammy application is loaded for the first time, an `init` function can be called that assigns certain behaviour to the elements on the site.

#### 5.1.4.2 Sammy plug-ins

Sammy offers the option to define custom plug-ins. A plug-in is some program code that is included just like the rest of the library but that is only used when it is first requested.

The following example, taken from [Qui10b], defines the helper function `alert()`, that overwrites the JavaScript function `alert` and replaces it by an entry from the log file:

```
1 var MyPlugin = function(app) {
2   this.helpers({
3     alert: function(message) {
4       this.log("ALERT! " + message);
5     }
6   });
7 };
```

Listing 5.2: Sammy.js: example of a plug-in

A plug-in is called using the `use()` method. This way, the plug-in function is utilised in the context of the current Sammy application. If the example plug-in is loaded the method can be used in all routes.

```
1 var app = $.sammy(function() {
2   this.use(MyPlugin);
3   this.get('#/', function() {
4     this.alert("I'm home"); //=> logs: ALERT! I'm home
5   });
6 });
```

Listing 5.3: Sammy.js: including a plug-in

### 5.1.5 Mustache.js

The rendering of HTML sites is done by the template engine *Mustache* [Leh10c]. A template engine is software that fills certain placeholders in a file with contents that are passed on to it. Mustache is

implemented in several programming languages, here the JavaScript version *Mustache.js* is used.

Using Mustache it is possible to separate code and markup [Leh10b]. The markup is done in an HTML file that contains placeholders for non-static values. The program code that calculates the resulting values or declares the variables is placed into a view. A view is a JSON object with attributes and methods that correspond to the placeholders in the template.

The following snippet exemplifies this (from [mus10]):

```
Hello {{name}},
you have just won {{gross_value}}$!
{{#taxable}}
Well, {{net_value}}$, after taxes.
{{/taxable}}
```

Listing 5.4: Mustache.js: Example of a template

```
{
  "name": "Chris",
  "gross_value": 10000,
  "net_value": 10000 - (10000 * 0.4),
  "taxable": true
}
```

Listing 5.5: Mustache.js: View passed on

The Mustache library is a JavaScript file that is loaded in run-time. After calling the method `Mustache.to_html(template, view)` the template and view are rendered:

```
Hello Chris,
you have just won 10000$!
Well, 6000.0$, after taxes.
```

Listing 5.6: Mustache.js: result

Since there is no code inside the templates that needs to be parsed, Mustache makes programming according to the *MVC (Model View Controller* architecture possible [Leh10b]. In an up-to-date comparison with seven other JavaScript templating libraries carried out by Brian Landau, Mustache.js did very well [Lan09a].

### 5.1.6 Further libraries

The interface is constructed using HTML and Cascading Style Sheets (CSS). The layout is done using the CSS framework *Blueprint* [Mon]. Blueprint produces a container with a certain width in

pixels, within which a grid-based layout can be structured. This grid is subdivided into 24 columns. The distribution of the elements on the page is done by assigning a certain `class` to them. A `div` element with the instruction `<div class="column span-16">` will automatically fill two thirds of the container. Blueprint also guarantees cross-browser compatibility; a site developed with Blueprint will show identical behaviour, irrespective of the browser type. Moreover, Blueprint contains several style sheets containing pre-defined designs that can be used in a new project.

The jQuery plug-in *jquery.autogrow* [Bado8] allows text areas that contain the outliner's lines to automatically adapt to the amount of text they contain. As text is being typed the textarea grows along in width and if necessary in height. This is achieved by calling the corresponding function on the DOM element: `$('textarea').autogrow();`

Further jQuery plug-ins being used are *jquery.md5*, *jquery.unwrap*, *jquery.scrollTo*, *jquery.color* and *date.format*.

## 5.2 Cloud computing

The deployment of the application was done using so-called *Cloud Computing*. The term in itself is a metaphor for Internet services, since they are often represented in computer network diagrams as clouds (see figure 5.4).

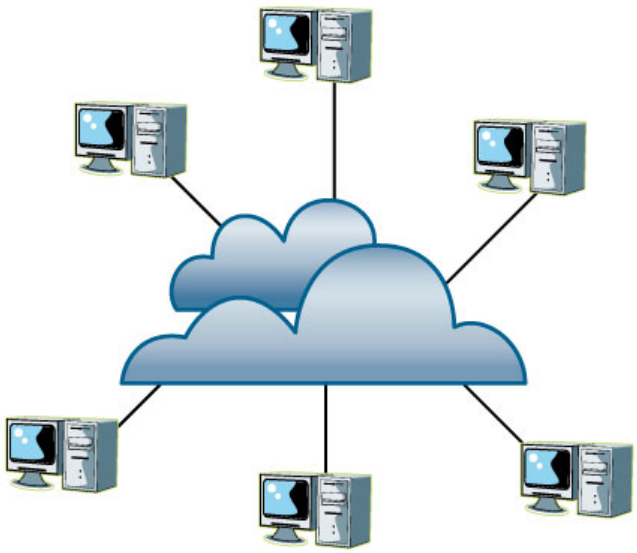

Abbildung 5.4: Cloud computing: A metaphor for Internet services [P. 09]

Cloud computing was introduced in 2006 by Google's CEO in a Talk about search engine strategies [Sch06]. The term is hard to define unambiguously. [Qia09, P. 626] defines it as „one of the vaguest terminologies in the history of technology". One reason is that cloud computing has many purposes. Moreover, the term is a marketing term used and abused by many companies. In 2009 it was one of the buzz-words in the IT industry that most often led to overblown expectations:

> The levels of hype around cloud computing in the IT industry are deafening, with every vendor expounding its cloud strategy and variations [...], compounding the hype. [Gar09]

According to the report quoted it will take less than five years for cloud computing to become „mainstream" (see the figure in section A.2.2).

Even though there exists no single definition of the term cloud computing, it is possible to establish unanimity over its basic concepts and general goals. Several definitions will be presented in the following section. After that the common styles of cloud computing will be presented. The chapter will be concluded with a section of the pros and cons of cloud computing.

### 5.2.1 Definition

First, two definitions from specialist literature will be presented:

> Cloud Computing makes IT infrastructure, platforms and applications of all kinds available for use as on-line services. [Bau10, Chap. 1.1]

> Cloud Computing is a kind of computing technique where IT services are provided by massive low-cost computing units connected by IP networks. [Qia09, P. 627]

[Qia09] mentions further central characteristics of cloud computing: virtualisation of the services offered, dynamic scheduling of resources and high scalability. If an application needs extra resources, they can instantly and effortlessly be made available. The infrastructure automatically adapts to the fluctuating or growing needs.

[Bau10] also stresses the criterion that the billing for such cloud computing services is usually done based on usage. It only charges for needed amount of resources and only those are made available. Thanks to their flexible nature, using these services may significantly lower operation costs.

The definitions mentioned above do not determine if services are hosted by distributed systems or by single, high-performance servers. This contrasts with *grid computing* which always involves distributed systems.

### 5.2.2 Styles

Cloud computing emerged around the turn of the century, when the Internet's expansion raised the bar for existing storage and computing facilities. Personal computers became ever cheaper; Internet service providers started to use them as the basic hardware platform [Qia09]. In order to put computer clusters flexibly into use, several software models were developed. That way computer resources could be abstracted, which gave rise to three major cloud computing styles. Their representation is based on [Qia09].

Amazon's cloud computing concept is based on *server virtualisation*. Since 2006 several web services are offered under the name of *Amazon Web Services (AWS)*. These make virtualised computing resources available for general use. Since AWS was initially cheaper than previous providers for on demand provision of such services, AWS pioneered the „Infrastructure as a Service (IaaS)" industry. This thesis will make use of *Elastic Compute Cloud (EC2)*, which is based on the virtualisation software *Xen*, the *Simple Storage Service (S3)* and the storage *Elastic Block Store (EBS)*. All these services are part of AWS.

Known representatives of the other two cloud computing styles include Google and Microsoft. Google offers *technique-specific sandboxes* that allow the hosting of applications that have been developed using certain technologies. Heroku, another company, offers similar services for applications written in the programming language Ruby. Microsoft's *Azure* service offers a combination of server virtualisation and technique-specific sandboxes.

Server virtualisation is regarded as the solution that is most flexible and most compatible with existing software and applications. The other approaches impose stricter limits on the choice of programming language, since every service only supports certain technologies. On the other hand, server-virtualisation means higher abstraction complexity. This approach is currently the most popular cloud computing technique to abstract services and resources.

The different styles mentioned above allow different ways of cloud computing. According to [Eym08], these are subdivided into three layers:

**Software as a Service (SaaS)** - A SaaS provider provides software as a service on the Internet. The software can be used without having to know about or deal with the infrastructure that serves as a basis for the service.

**Platform as a Service (PaaS)** - A PaaS provider makes a platform available that allows easier access to a combination of different services.

**Infrastructure as a Service (IaaS)** - IaaS providers offer hardware as an infrastructural service on which individuals and companies may run their own services.

Figure 5.5 lists target audiences and examples for providers for each of the three levels.

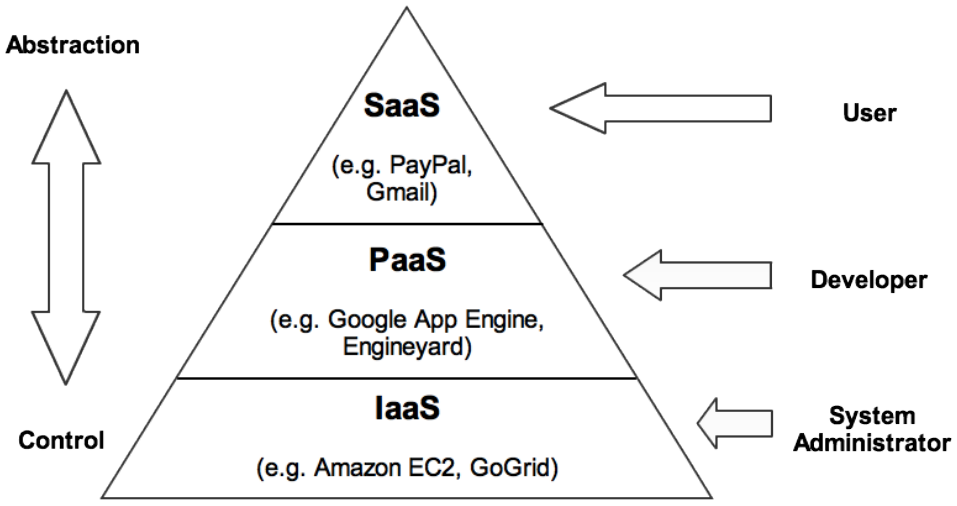


Abbildung 5.5: The three levels of cloud computing

### 5.2.3 Pros and cons

Cloud computing is described in [Qia09, p. 629] as a win-win strategy for the service provider and its users.

One of the advantages is that business requirements can be met on demand: customers can adjust at any moment the amount of resources available to tailor them to the actual needs. This saves money and energy. Resource management can be further enhanced by dynamic *resource scheduling*.

The disadvantages include the fact that traditional hosting providers usually have clearer answers to the questions of privacy and security, since here only the user knows how the system is configured. Moreover, the reliability of the services is not guaranteed in case of technical failures or power outages. In cloud computing this may have devastating consequences since such an incident may not shut off just a single service, but interrupt all services at once. Such problems can be avoided with *Service Level Agreements* that warrant the scope of services and their quality. Another noteworthy disadvantage of cloud computing is that it is usually impossible to migrate services to another provider. Until now, external interfaces have hardly been defined. Because of the advantage in competition most of those services are currently proprietary [Bau10].

In order to be able to estimate the financial gains from cloud computing, the costs of the actual use of cloud services in hours or server units have to be compared to the costs of owning a data centre or IT infrastructure. In doing so, the average load of the data centre has to be factored in. Also to be taken into consideration is the fact that a data centre's capacity is fixed, whereas cloud services have upwardly open limits [Bau10, Chap. 7.2.1].

## 5.3 Methods and tools

The previous sections described the technologies that directly contributed to the finished application; the following will deal with the tools used for the implementation. This chapter lists all technologies and methods that were used in the development but are not a part of the final product.

To start with, both agile the software development procedure and the test-driven development procedure model are presented. Afterwards, the testing frameworks that helped with the test-driven development will be described. The chapter concludes with a section on development environments.

### 5.3.1 Development procedure models

The procedure model that was chosen is the approach used by agile software development. Test-driven development can be seen as a subset of the procedures own to agile software development. This thesis does, however, rate test-driven development high, which is why an entire section will be devoted to it.

#### 5.3.1.1 Agile software development

Already in the late eighties, the realms of software development echoed growing criticism against conventional phase models and development procedures. [Hes92] lists sources of discomfort with the classical life cycle concept. The author does not attribute this discomfort to some sort of vogue; it „is based on serious experiences with the conventional models and their noticed weaknesses“ [Hes92, Chap. 2.5.1]. These include overlong time periods between specification and an executable program, and the insufficient involvement of customers and users; but most notably how strictly successive phases are out of touch with reality.

In 2001, some well-known representatives of agile software development wrote its core values down in the so-called *Agile Manifesto* [Bec01]. These values lay the foundations of the development procedure that is more precisely defined by the manifesto. When applied to the development procedure of the application developed in this thesis, the following goals can be identified:

- Frequent feedback and communication between everyone involved in the project
- Early and frequent delivery of software; this way, it is possible to verify whether the development procedure is still on the right track to achieve the project's actual goals
- The possibility to adjust the initially plans regarding requirements and procedure to the actual requirements

Building on the Manifesto's core values, [Amb] defines agile software development as follows:

> Disciplined agile software development is: an iterative and incremental (evolutionary) approach to software development; which is performed in a highly collaborative manner; by self-organizing teams within an effective governance framework; with „just enough" ceremony; that produces high quality software; in a cost effective and timely manner; which meets the changing needs of its stakeholders.

The development in this thesis is done according to the agile methods. This is understood as the established procedure of acting in an agile manner in a part or aspect of the software development [Ble08, Chap. 2.4]. Worth mentioning here are continuous code refactoring, continuous integration and test-driven development.

Code refactoring is defined in [Ham08] as *behaviour-preserving transformation*. Accordingly, refactoring is „the process of transforming source code in order to improve its internal appearance, without changing any functionality" [Ham08, p. 2]. Continuous integration means that all tests are automatically executed when new code is integrated into the project. This process guarantees continued application functionality. Test-driven development is discussed in the next section.

#### 5.3.1.2 Test-driven development

The application is developed in a test-driven manner. Test-Driven Development (TDD) is one of the most important and most widely spread practices in agile software development [Ham08, p. 2]. It was conceived to warrant the quality and serviceability of the program.

[Ham08] describes the *test-driven development cycle*. TDD always contains three recurrent steps: *test - code - refactor*.

**Test** A test for the code to be generated is written and started. Since the code does not yet exist or the desired functionality has not yet been implemented, the test will fail. It is important to proceed in small steps, only testing one aspect of the code at a time.

**Code** The code for the new feature is written in its simplest conceivable implementation. The test will now pass successfully.

**Refactor** The code is now improved by refactoring. Attention should be paid that all the tests must continue to pass successfully.

This method is also called *test-first programming*.

[Fis08] sums up the qualities of a good test. A meaningful test should be **unambiguous**, i.e. it should produce a discrete yes/no result. Furthermore, it should be **valid**: the test results have to correspond to the intention of the artifact being tested. A test is **complete** when it does not need

further input to run; a **repeatable** test is one whose result is deterministic even if the tested system does not behave deterministically. A test should be completely **isolated**, meaning that the result must not be influenced by results or side-effects of another test. In [Ham08], this anti-pattern is called *test-coupling* which should be avoided. The final trait of a good test requires that tests be able to be started **automatically**, that they finish in a **finite** amount of time, and that they may be **bundled** with other tests into a test suite.

A further development of TDD is *Behaviour-Driven Development (BDD)*. This shifts the stress from the aspect of „testing“ to the aspect of „prespecification“ [Ada07, Chap. 1]. Similar to domain-driven design, the result is here approximated from the business point of view. In this way, the language with which the problem that needs solving is described can be kept clean of technical terms. There are several BDD frameworks that allow specifications for software to be expressed in executable code:

> A waterfall „designer“ starts from an understanding of the problem and builds up some kind of model for a solution, which they then pass on to the implementers. An agile developer does exactly the same, but the language they use for the model happens to be executable source code rather than documents or UML. [Ada07, Chap. 2]

In practice, the terms TDD and BDD are often used to mean the same thing [Mil07]. In contrast to the misgivings of some developers, TDD/BDD do not usually lead to higher effort and longer development times. The earlier tests exist and the more comprehensive they are, the faster and painless the development cycle becomes. Therefore, the application at hand should be implemented using this method.

## 5.3.2 Testing frameworks

The tests that were written during the test-driven development can be divided into several layers. *Unit tests* check the functionality of individual software modules. *Integration tests*, on the other hand, verify the interaction between the system's components. Depending on the framework there might be further layers, but it suffices in the scope of this thesis to mention only these two, since they provide a very good functional test coverage when they are combined and well implemented. The following sections will present the particular frameworks.

### 5.3.2.1 JSpec

A unit test framework is a piece of software that helps writing and executing unit tests. Such frameworks provide a basis that simplifies writing tests, and functionality to perform the tests and

output results. Unit tests are developed apart from the actual application; they are not included in the final product. They use the application's objects, but exist only inside the unit test framework. This allows the isolated testing of individual objects without letting them interfere with the actual code [Ham08].

At this point, the limited scope of this thesis allows no substantial comparison of suitable unit test frameworks. An evaluation by the author can be found in [Her10]. From many respectable alternatives [Wik10b], the relatively young framework JSpec was chosen [Hol10].

JSpec is similar in functionality and syntax to the BDD framework *RSpec* [Che], with which Ruby code can be specified and tested. The JSpec syntax is a domain-specific language (DSL) which was specially designed with this purpose in mind.

*Matchers* specify a certain behaviour or the value of an object. *Assertions/expectations* verify this and compare it to a certain value. The keyword `should` can be compared to the keyword `assert` from conventional *xUnit test frameworks* that are based on Kent Beck's *SUnit framework*.

```
1 { foo : 'bar' }.should.eql { foo : 'bar' }
```

Listing 5.7: JSpec example: the matcher `eql`

Further examples may be found in the appendix A.5.8.2.

There are also matchers for JQuery functionality that allow the DOM to be tested. JSpec further supports testing asynchronous functions, and provides the option to simulate AJAX requests. Fixtures allow parts of the DOM to be supplied as HTML code.

JSpec can be installed as a Ruby Gem, meaning that tests may be run from the console and integrated into continuous integration. For this purpose it makes use of Rhino [Mozc], a Java-based JavaScript interpreter. In the console, it is possible to specify the browser that should be opened in the background; this browser will then execute the tests.

Alternatively, the JSpec library can be included in the JavaScript code. This approach involves opening a HTML file in the browser that will then run the tests. This window will also contain the results. The level of detail and the formatting of the results can be set as parameters in the HTML file. How that looks can be seen in figure 5.6.

#### 5.3.2.2 Cucumber

With TDD, tests are written before any code is, hence the tests are called black box tests [Bec07]. With these, the implementation of the program components that need testing is still unknown: only the expected functionality or rather the result are being tested. For unit tests, this is not always strictly workable, since they meticulously test the components' characteristics. Integration tests,

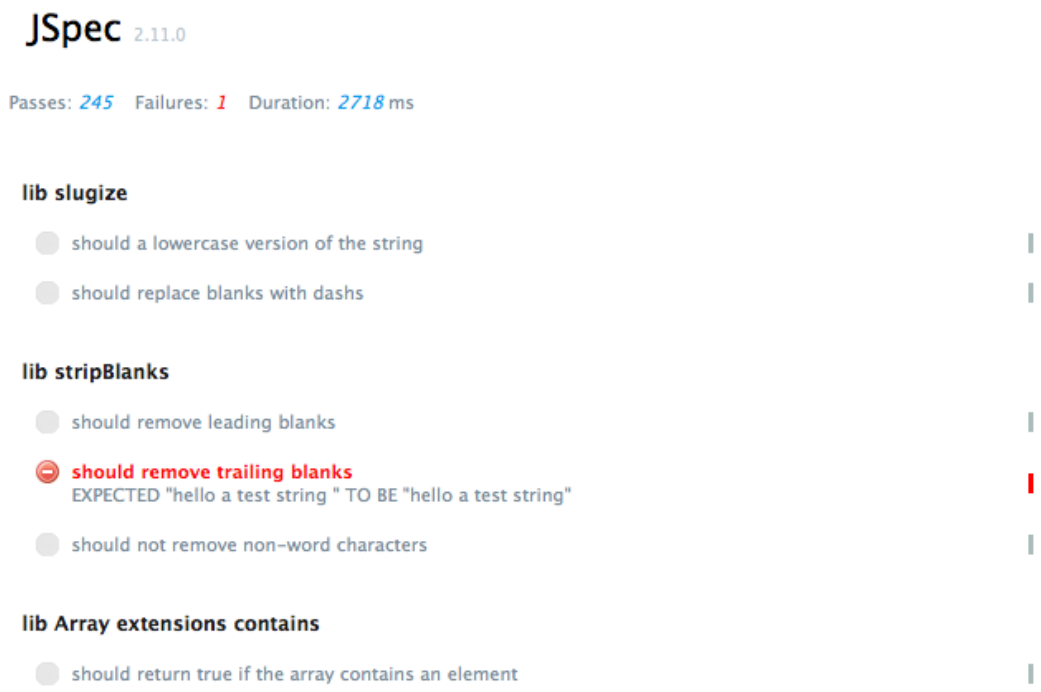


Abbildung 5.6: JSpec: A failing test

on the other hand, apply at a higher level: a component's functionality is described in a manner that is also understandable for users with no technical background. This allows the developer to think about the business logic without being troubled by implementation details.

Even if a software component has passed a unit test successfully, the quality can only be guaranteed if the component was successfully integrated into the application. The testing framework Cucumber [Hel] provides the tools to create such integration tests.

The test for larger program components that belong together (e.g. user administration, usage of the outliner) is called a *feature* in Cucumber's *Domain Specific Language (DSL)*. Every feature specifies the role of the user (*As a*), the content of the feature (*I want*), and a goodwill (*In order to*). A feature contains several *stories* that each describe the execution of a software function from the beginning to the end (e.g. user log-in, indent a line). A story contains the preconditions (*Given*), the individual steps a user makes (*When*) and the expected results (*Then*). An example from the project can be found in listing 5.8.

```
1 Feature: CRUD for outlines
2   In order to sort my notes
3   As a user
4   I want to create, list, update and delete outlines
5
6   Scenario: create an outline with note
7     When I go to the start page
```

```
      And I follow "New Outline"
      And I fill in "title" with "Songs"
      And I press "Save"
    Then I should see "Songs"
      And I should see "Here is your new outline"
      And the new note li should be blank

  Scenario: edit an outlines title
    Given an outline with the title "Songs"
      And I save
    When I go to the start page
      And I follow "Songs"
      And I follow "Change title or delete this outline"
      And I fill in "title" with "Tunes"
      And I press "Save"
    Then I should see "Title successfully changed"
      When I go to the start page
    Then I should see "Tunes"
      And I should not see "Songs"
```

Listing 5.8: A Cucumber feature with two scenarios

The meaning of individual rows, called *steps*, has to be defined in further files. Every step consists of a signal word and a regular expression for which a block of ruby code is executed. In the process, the results of the matching groups are fed to the block as a regular expression. This is exemplified in listing 5.9.

A feature can be executed from the command line (for example in figure 5.7). The successful steps are printed in green; the ones that failed are printed in red including their error messages.

```
Given /^an outline with the title "([^\"]*)"$/ do |title|
  outline = {:kind => 'Outline', :title => title}
  RestClient.put "#{host}/#{database}/#{title}", outline.to_json
end

When /I fill in "(.*)" with "(.*)"/ do |field, value|
  find_by_label_or_id(:text_field, field).set value
end
```

Listing 5.9: Cucumber step definition

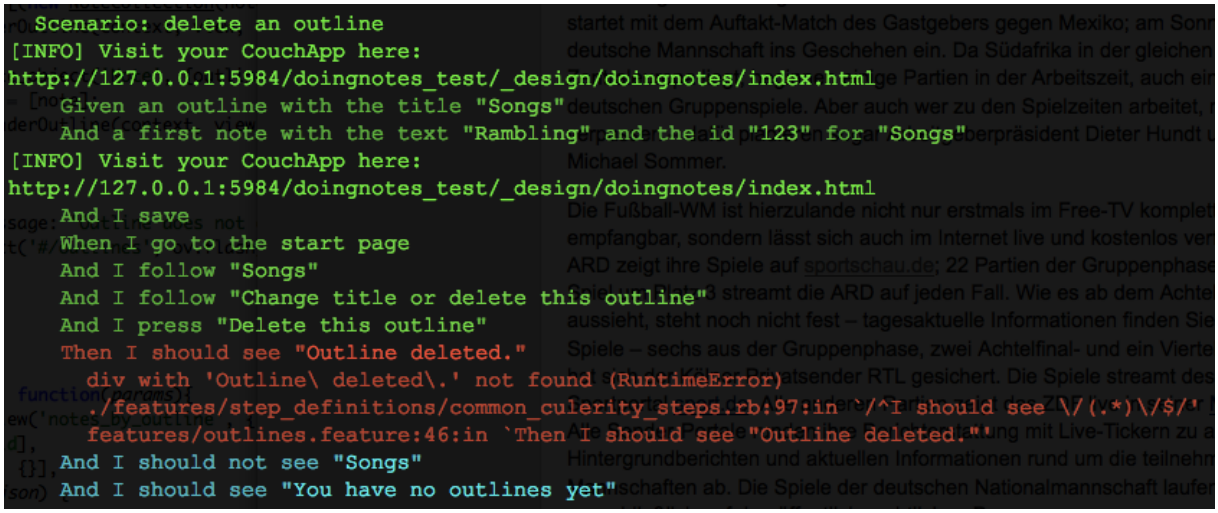


Abbildung 5.7: Cucumber: A failing test

### 5.3.3 Development environments

#### 5.3.3.1 Textmate

TextMate [Mac10] is a text editor for the operating system Mac OS X. Textmate was released in 2004 by Alan Odgaard. In August 2006, the program was awarded the „Apple Design Award for Best Developer Tool“ at Apple's „Worldwide Developers Conference“. TextMate is a very clearly-arranged editor [Ski07]. It does not boast the same functional range as Eclipse or NetBeans, but its functionality can be expanded at will thanks to its broad support for scripts and plug-ins. For the development of the task at hand, an integrated development environment wouldn't be helpful: the specific requirements for developing a CouchApp are currently not met by Eclipse.

Textmate provides syntax highlighting for all the languages used, auto-completion inside a file, project-wide searching, easy access to all files in a project, an easy-to-use interface with tabs for all open files. In addition, it is easy to adjust Textmate to particular needs. For example, a special CouchApp macro was written to update the design documents in the database, something that accelerated the development process.

#### 5.3.3.2 Firefox / Firebug

Section 8.5.2 explains why the web browser Firefox [Moz10d] in version 3.6 or higher was chosen as the target platform. Because of this, the application was developed with the aid of the firefox add-on Firebug [Moz10a].

Firebug allows developers to examine style sheets, HTML, the DOM and JavaScript on a page.

A console allows logging HTTP requests and log statements in the JavaScript code. The source code of a website can also be analysed and edited on-the-fly. This makes debugging easy. The tool is therefore extremely popular amongst web developers [IDG10]. Firebug is the sixth most downloaded add-on for Firefox [Moza]. It is used on a daily basis by two million people [Mozb].

In this project, Firebug was used for the development of the DOM, for the shaping of the front-end, and for the optimisation of the site's performance.

# 6 Requirements of the system

The following sections deal with the requirements of the system. Functional and non-functional requirements will be defined. Section A.4.1 in the appendix also contains an overview in tabular form. The functionalities that are required and desired are thereby described with the aid of use case diagrams. This chapter will provide an answer to the question of what a system has to provide for it to fulfil the requirements set in the analysis (chapter 3).

## 6.1 Functional requirements

### 6.1.1 Must-haves

Figure 6.1 depicts a use case diagram for the requirements of the outline management and the outliner that were identified as strictly necessary. Figure 6.2 depicts the requirements of replication and conflict resolution. The prototype to be developed in this thesis must meet these requirements. Only then the implementation of the task may be called successful.

#### 6.1.1.1 Outline management

The user should be able to create any number of outlines (**FA100**). The outlines should be clearly represented and their title should be changeable (**FA101**).

#### 6.1.1.2 Outliner

The outliner is to have the look & feel of a text editor with an unlimited number of lines (**FA200**). It should be possible to navigate between the lines using (combinations of) keystrokes (**FA201**). The contents of the lines should be editable (**FA202**). When the cursor leaves a line it should automatically be saved (**FA203**). When the window is closed while editing a line it should be automatically saved. Alternatively, before closing the window the user should be informed about the possible loss of data (**FA206**).

The lines have to be able to indent and outdent in order to represent the underlying hierarchy (**FA204**). Indenting or outdenting a line should immediately move any following lines that are at a deeper indentation level (**FA205**).

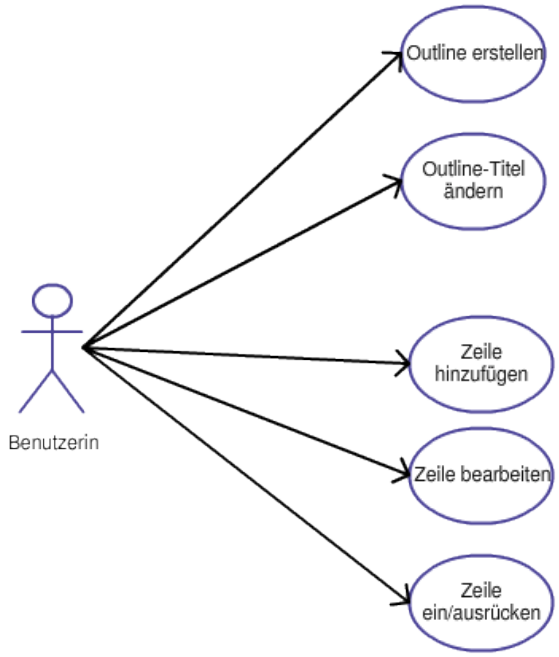


Abbildung 6.1: Use case diagram for the requirements of outline management

#### 6.1.1.3 Replication

If the user is on-line or if the connection is restored after an off-line period, the outlines (**FA300**) and changes to outlines (**FA301**) made by the user should immediately be replicated to the server.

Outlines (**FA302**) and changes to outlines (**FA303**) should automatically and immediately be replicated to the user's computer while they are connected to the server. The user must be informed about changes as soon as they arise (**FA304**) without interrupting the user's work.

If the connection is restored after an off-line period, the user should either be informed that replication is again possible, or replication should start automatically (**FA305**).

#### 6.1.1.4 Conflict handling

Of the conflicts that may arise during replication, at least one type should be resolved automatically by the system (**FA400**). At least one conlict type should be resolved manually (**FA401**).

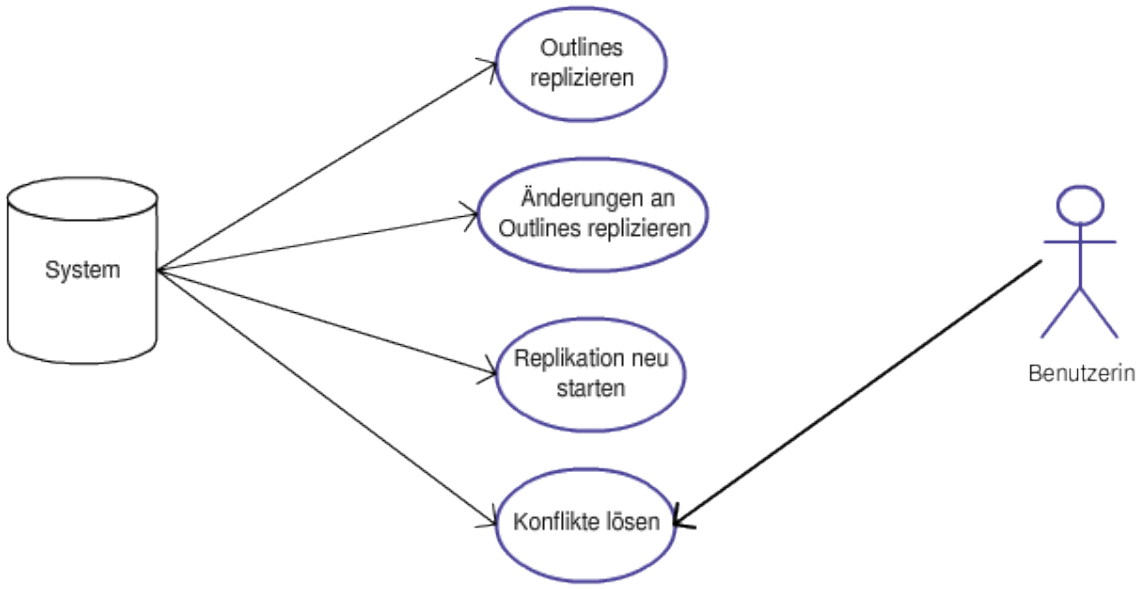


Abbildung 6.2: Use case diagram for the requirements of replication

### 6.1.2 May-haves

Not all criteria listed in this section must be implemented in the prototype. Their later implementation should, however, be taken in consideration during the design phase.

Figure 6.3 depicts a use case diagram for the requirements of outline management and the outliner that were identified as optional. Figure 6.4 illustrates the requirements of replication and conflict resolution.

#### 6.1.2.1 Outline management

It should be possible to delete outlines (**FA102**).

#### 6.1.2.2 Outliner

It should be possible to move lines up or down (**FA207**). Their size should automatically adapt to the amount of text (**FA208**). To increase clarity, it should be possible to collapse or expand the lines (**FA209**). Information about which lines have been collapsed should not be replicated but possibly locally stored (**FA210**).

The revision of a line should be saved automatically (**FA211**). The user should be able to jump between revisions (**FA212**). It should be possible to add comments to individual lines (**FA213**)

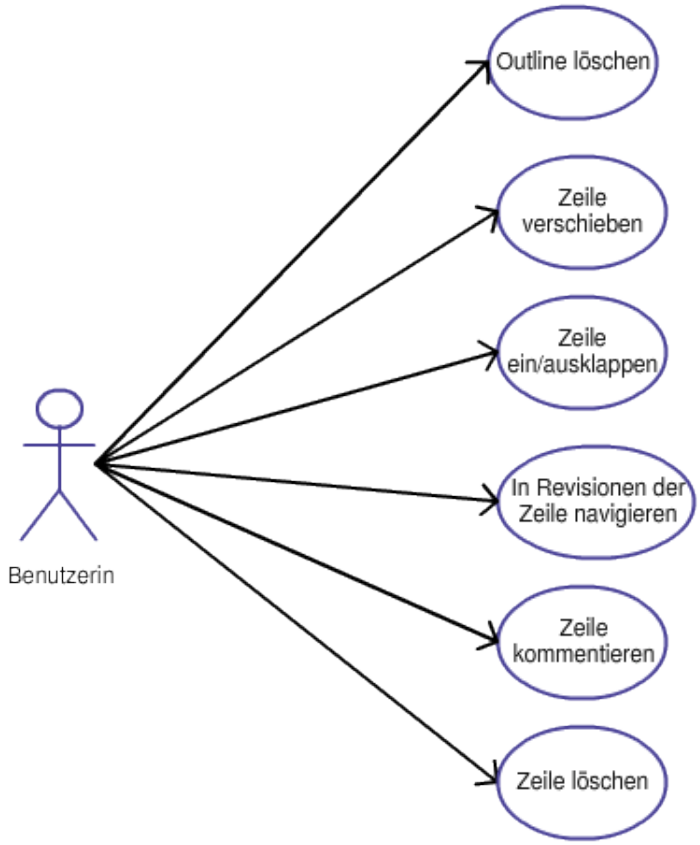


Abbildung 6.3: Use case diagram for optional requirements, outline management

and to remove the lines (**FA214**).

#### 6.1.2.3 Replication

The available outlines should be published by the system so that connected users may choose which single outlines they want to replicate (**FA306**).

The user should be notified whether a connection to the server is established (**FA307**). The interface should allow replication to be activated and deactivated (**FA308**).

#### 6.1.2.4 Conflict resolution

Combinations of multiple conflict types should be solvable by the system or the user (**FA402**). Conflicts that occur between more than two replicas should be treated correctly (**FA403**).

The system should be able to independently resolve as many conflicts as possible (**FA404**). The interface should be developed so that the user might resolve as many conflict types as possible by

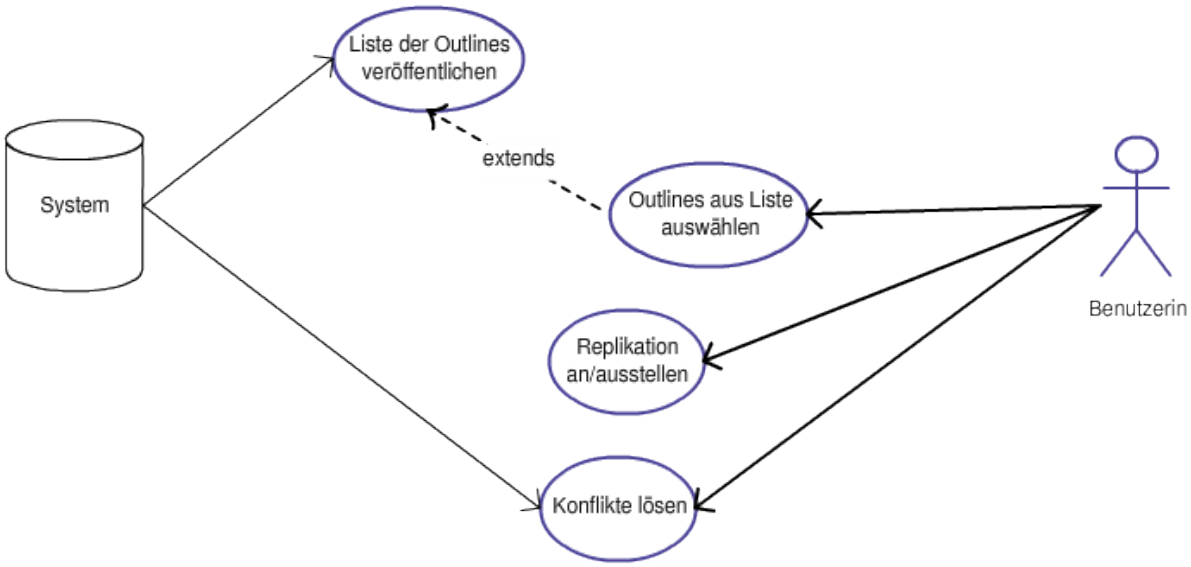


Abbildung 6.4: Use case diagram for optional requirements, replication

hand (**FA405**).

### 6.1.3 Demarcation criteria

#### 6.1.3.1 Outline management

No user or access management will be implemented for outlines.

#### 6.1.3.2 Outliner

No columns will be implemented.

#### 6.1.3.3 Replication

No peer-to-peer replication will be implemented.

#### 6.1.3.4 Conflict resolution

The system will only be optimised for use by a lesser number of users. For conflicts between more than two versions there may be no reliable conflict resolution.

## 6.2 Non-functional requirements

### 6.2.1 Usage

#### 6.2.1.1 Target audience

Users of the system should have average knowledge of how to use computers and web browsers in particular. Furthermore, they should be able to install and run a CouchDB instance on their computer. The users should have some understanding of the advantages and restrictions of using such a system.

#### 6.2.1.2 Operating conditions

The server responsible for exchanging outlines and updates should be able to run 24 hours per day and seven days a week in order for the services to remain available at all times. To ensure this, the service should be deployed using Amazon's Elastic Compute Cloud (Amazon EC2) service. Greater scalability is achieved with the aid of the clustering framework CouchDB Lounge.

The application should run on every computer capable of running CouchDB. Further information about systems supported by CouchDB can be found in section 9.1.

### 6.2.2 Environment

In order to avoid licence fees and to be able to adapt the system to varying demands, it should exclusively be implemented using open-source software.

#### 6.2.2.1 Hardware

Should the CouchDB instance that has the server role not run on Amazon EC2, but rather on a private server, this server should meet at least the following system requirements:

- Intel processor clocked at 3,0 GHz
- 5 GB of free disk space
- 1 GB RAM memory
- Ethernet connection of 100 MBit

#### 6.2.2.2 Software

The system is built on several software packages and programming languages. The versions indicated are the minimum requirements. Installation notes can be found in section 9.1.

- CouchDB 0.11.0
- Spidermonkey 1.7
- Erlang 5.6.5
- ICU 3.0
- cURL 7.18.0
- Automake 1.6.3
- Autoconf 2.59

The Rake tasks supplied for deployment and operation require Ruby version 1.8.6 or higher to run (see section 9.2.4).

Section 8.8 contains some notes regarding the test set-up.

### 6.2.3 User interface

The system requires a simple and transparent web interface. It is assumed that the user has JavaScript activated. The user interface should run without restrictions in Firefox 3.5 or newer. The optimal delay between input via the web interface and the activation of the desired function is under one second and should in no case exceed four seconds. This time span is the reaction time that people tolerate in normal conversation or telephone systems [Mil68, p. 267 & 270]. Jakob Nielsen connects this time span to interaction times in web applications [Nie93, chap. 5.5].

Figure 6.5 depicts a mock-up for the interface showing the outline overview. Figure 6.6 shows the structure of the site containing the outliner.

### 6.2.4 Quality goals

The quality of the final system is another important goal. The following quality requirements have to be respected:

- Expandability through open architecture

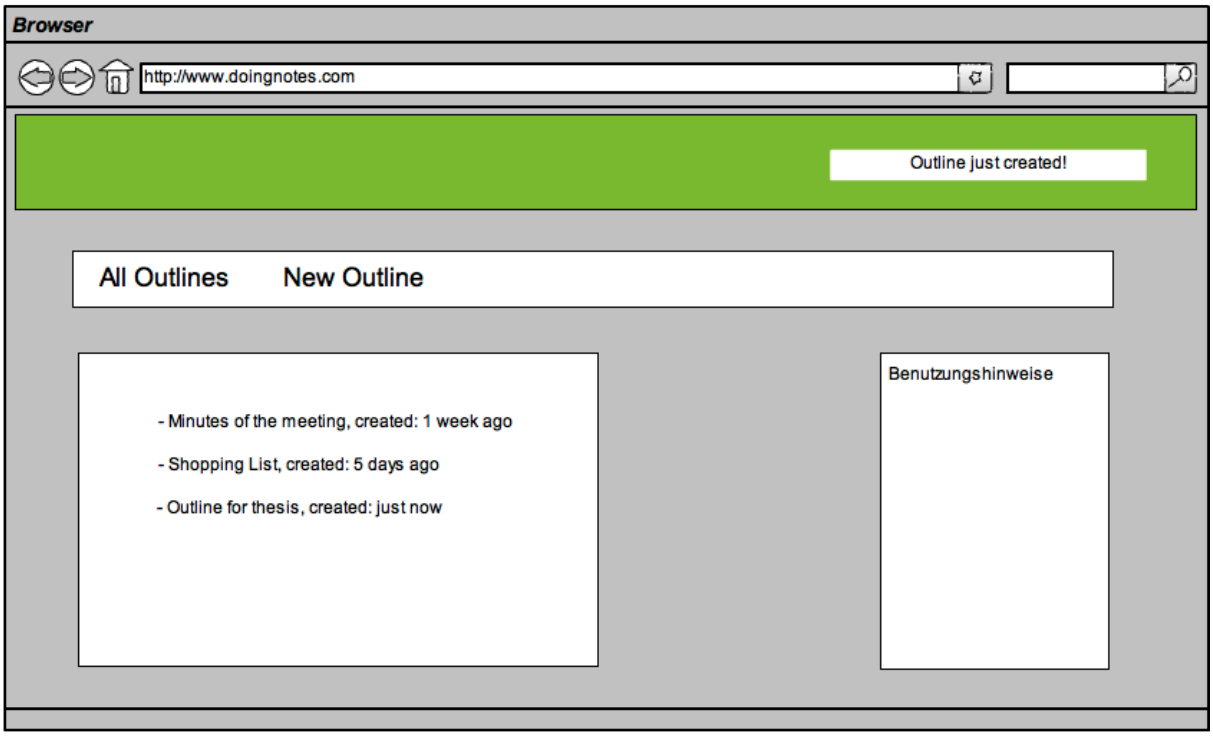


Abbildung 6.5: Layout of the web interface: outline overview

- Portability through low hardware and software requirements
- High software quality through extensive testing
- Design and programming after framework-specific standards
- Implementation according to the MVC architecture: separation of interface, application logic and data
- Highly maintainable and therefore simple and non-redundant code
- Adherence to usability and accessibility guidelines with regard to the interface
- Fast-loading application and therefore little dependency on external frameworks

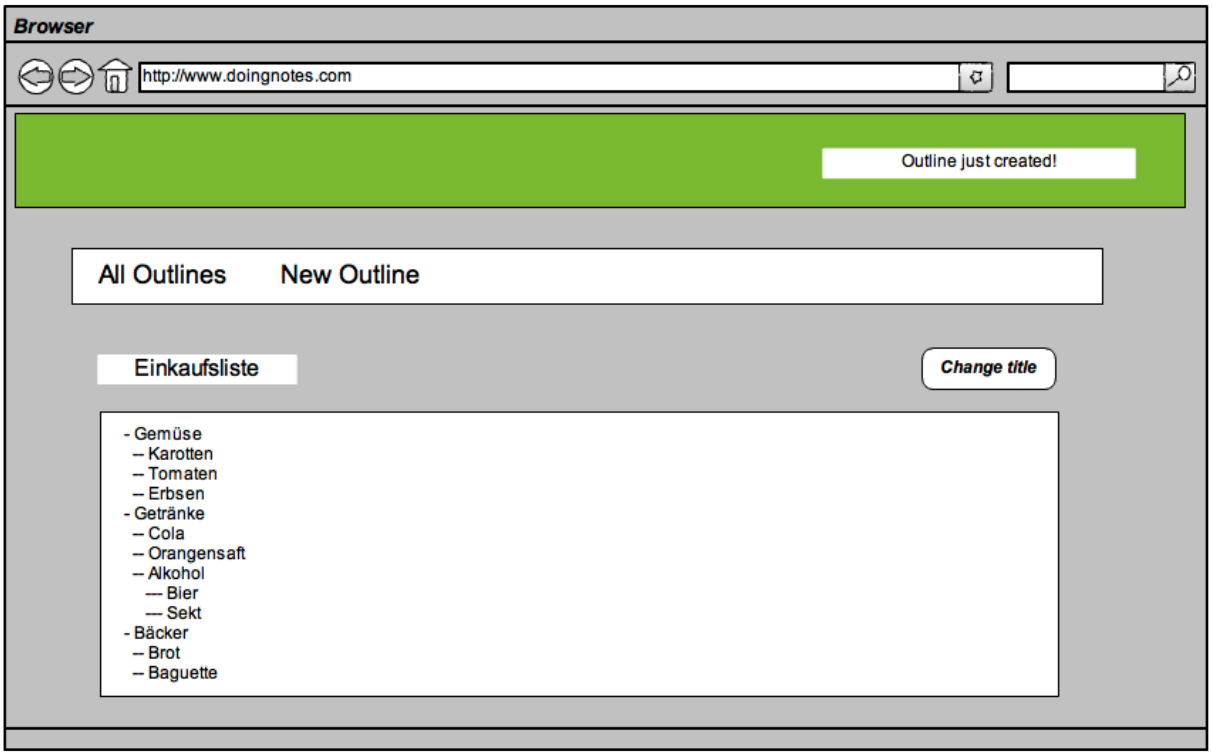


Abbildung 6.6: Structure of the web interface: single outline view

# 7 System architecture

Building on the requirements of the system described in chapter 6, this chapter will present a draft of the system architecture. Since CouchApp applications allow an architecture with no middleware, the focus lies less on the presentation of the components. Rather, the inner structure of the application will be described. The concepts for the central design problems will be discussed and every choice will be accounted for. This will convey an overview of data storage functionality, application logic and the user interface.

The code snippets in this chapter are no complete CouchDB documents. They only contain what is strictly necessary for the demonstration of certain aspects.

## 7.1 Architecture overview

Conventional web applications are built according to *client-server* architecture: the data reside in a usually relational database, the application logic is executed on the server and the results are sent to the client (see fig. 7.1). Only some parts of the application logic are sometimes executed by the browser as an add-on to enhance user experience.

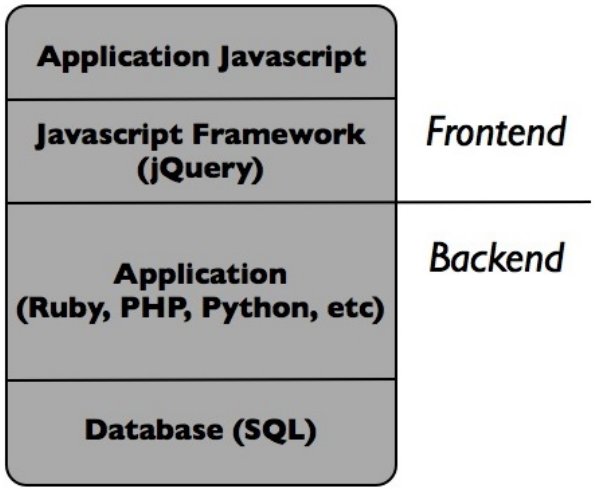


Abbildung 7.1: Architecture of a conventional web application, according to [Qui09]

If one assumes that the browser supports JavaScript and HTML5, larger parts of the application can be run locally on the user's computer. CouchDB furthermore provides its own web server. This eliminates the need to use middleware for the application logic (see fig. 7.2).

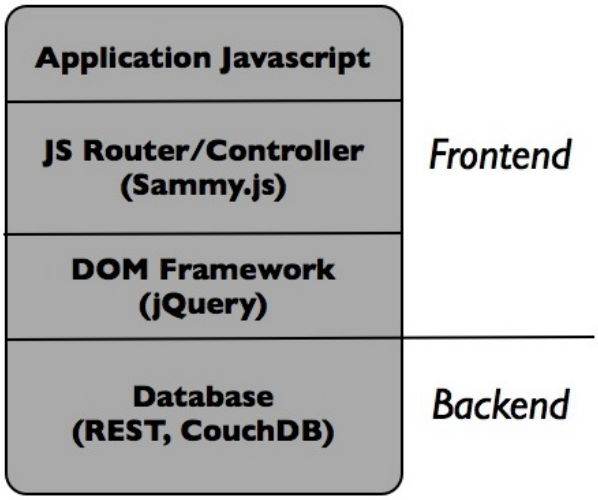


Abbildung 7.2: Architecture of a CouchApp, according to [Quio9]

If CouchDB is installed onto a local computer, the application may be run like a desktop program. The server instance's only responsibility is the synchronisation of outlines between clients (see fig. 7.3). It is sufficient to implement a single application that runs on the clients as well as the server. The application can also be used on the clients when the server is unavailable.

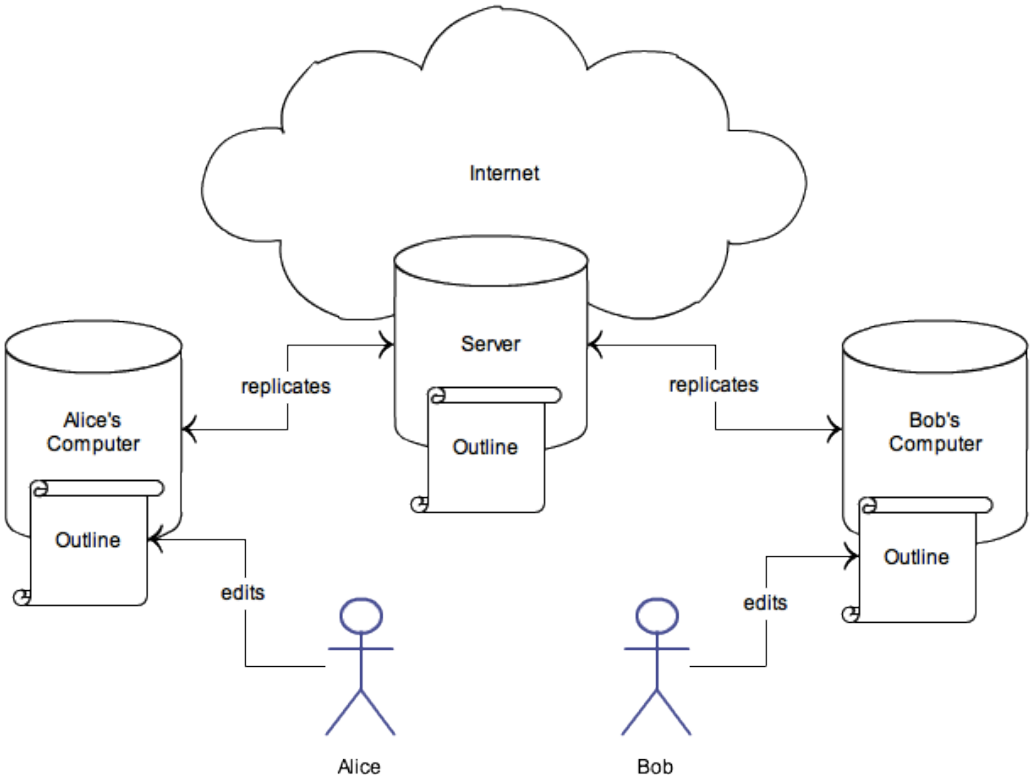


Abbildung 7.3: Project vision

## 7.2 Data structure modelling

For the storage of data in the database, there were several alternatives. The discussion of these will justify the solution chosen.

### 7.2.1 Requirements

The requirements for data structure modelling are prioritised as follows:

- It should be easy to program and maintain the application.
- Conflicts caused by simultaneous saving of data should be avoided or occur as infrequently as possible.
- Data access should be as fast as possible. Since less requests reduce access times, related data should be stored together wherever possible.

CouchDB's replication can automatically highlight conflicting documents. A workable application, however, will require some logic that allows these conflicts to be resolved. The data structure was designed with advice taken from [Cou09].

### 7.2.2 Problem

An outline is a sorted, hierarchically nested list of lines with varying indentation. A simple example is illustrated in figure 7.4. The bullets for lines that have no child nodes are circles, and triangles for lines that do. Such an outline should be implemented meeting the requirements mentioned above.

In the example a shopping list is composed with the aid of the outliner. This is indeed not the main purpose of an outliner, but it illustrates the hierarchical indentation in an intuitive way, and it allows the example to remain short yet realistic.

### 7.2.3 Storage in a JSON document

The easiest implementation is saving everything into a single JSON document (see listing 7.1). For this approach, every line has to be a JSON object in itself, so that they can be stored in their nested state.

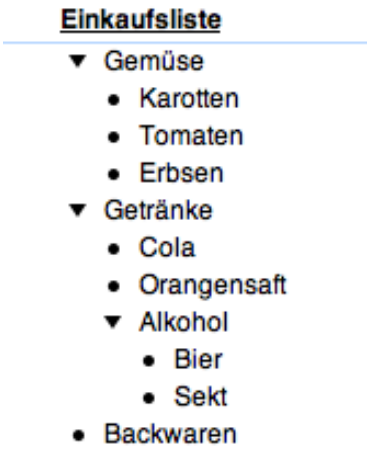


Abbildung 7.4: Simple outline

This outline will temporarily exist in two or more different versions during the application's lifetime. On the next synchronisation of both versions they should again be merged. Lines that were altered, added or deleted should also be altered, added or deleted in the other version. The user should only have to intervene when a line has two competing versions. How can this be achieved?

```
{
  "title": "Shopping list",
  "lines": [
    {"text": "Vegetables", "lines": [
      {"text": "Carrots"},
      {"text": "Tomatoes"},
      {"text": "Peas"}
    ]},
    {"text": "Drinks", "lines": [
      {"text": "Cola"},
      {"text": "OJ"},
      {"text": "Alcohol", "lines": [
        {"text": "Beer"},
        {"text": "Champagne"}
      ]},
    ]},
    {"text": "Pastries"}
  ]
}
```

Listing 7.1: Simple outline in a JSON document

The structure in listing 7.1 is already valid JSON. It could be stored as-is in a single CouchDB document. This document could be read with a single read operation. The replication of such a

document is also done in just one action. Even the implementation of application logic is relatively easy, because the indentation information is contained within the document.

However, problems may arise as soon as the document is modified. When a line is changed, added or moved, CouchDB saves a new version of the outline document with a new revision number. Without exception, conflicts occur every time the outline is replicated with another user's version after the modification took place.

After a replication with many changes in the document, too, there are only two different line sets. One set will be stored as the winning, the other as the conflicting revision. This semantic is very impractical to the user: she can only save either her own or the other version. All changes have to be repeated by hand. This voids some of the central advantages of CouchDB's replication.

### 7.2.4 System prevalence

Another option is to make use of *system prevalence* [Pato5]. This persistence technique is used in object databases such as Madeleine [Beno6]. Rather than newer versions of the outline, the database records individual operations to the original data set. This way, „change", „save", and „move" are single entries in the history of an outline. These entries are stored one after another and never changed. In doing so, no conflicts arise during their replication. The entries can be saved in an array in a single CouchDB document. If replication conflicts occur, they can be easily resolved by combining the elements of both arrays.

However, this approach would also produce problems when different versions are merged, since the order in which changes are made may influence the result. It would be necessary to develop an algorithm that determines the order of the changes in a meaningful way.

### 7.2.5 Version history storage

Rather than saving the history of the commands, it would be sensible also to save the outlines' older revisions. CouchDB deletes old revisions when the database is compacted. In order to circumvent this, revisions may be saved permanently in their own documents. The system should save a reference to the most recent version and every revision should point to its predecessor. When documents are merged after replication, both conflicting versions may be evaluated until the last common predecessor is found. Based thereon, the merging may proceed.

This is similar to the way versioning systems like Git or Subversion work, except that JSON fields are being compared instead of lines in a text file.

Following this approach, every write operation will also cause the entire outline to be saved as a new version. This means that the database quickly accumulates huge amounts of data. This

is a considerable drawback. On merging, both document versions have to be analysed by the application to establish which line was changed in which way and in which version. This process was tested with the aid of a simple prototype with limited functionality, yet quickly rejected for being too complex. Instead, a method should be developed that performs the breakdown already in the document, in order to comply to the quality requirements dictating limited complexity.

### 7.2.6 Breaking up lines in individual JSON documents

The final approach that was examined involves saving every line in its own document. Adding or deleting a line is achieved by creating or deleting the corresponding JSON document. This method does not necessarily give rise to conflicts. Changing a line will only produce conflicts when both sides change this line at the same time. Only then, user intervention is required. This way, the replication issue is relatively easily solved by the application.

This way of modelling the problem results in many smaller documents in the database, each of which contains only one line. This requires the introduction of an attribute that allows an outlines' lines to be easily recognised in the database. Every line document contains a reference to the outline it belongs to.

This approach brings about the problem that nesting and sorting information is no longer contained inside the document. The solution to this problem will be treated separately in section 7.3.

### 7.2.7 Conclusion

All four solutions have one thing in common: the history of the data has to be saved in some way. Only then it is possible to compare -on merging the replicas- the current state of the data with the state in which the versions did not yet differ. CouchDB saves different versions of documents; it makes sense to make use of this feature. The easiest and least error-prone way is therefore to save lines as individual documents. This is exemplified in listings 7.2 and 7.3.

```
{
  "_id": "1dbdcbc27b22cc7a14cd48d397000657",
  "kind": "Outline",
  "title": "Shopping list"
}
```

Listing 7.2: Outline with ID and type

```
{
   "kind": "Line",
```

```
    "text": "Vegetables",
    "outline_id": "1dbdcbc27b22cc7a14cd48d397000657"
},
{
    "kind": "Line",
    "text": "Drinks",
    "outline_id": "1dbdcbc27b22cc7a14cd48d397000657"
},
{
    "kind": "Line",
    "text": "Pastries",
    "outline_id": "1dbdcbc27b22cc7a14cd48d397000657"
}
```

Listing 7.3: Three lines with ID and type

By creating a CouchDB view with a composite key, it is possible to query the outline and all its lines in a single request. This is achieved using the widely-used *view collation* technique, which simulates the building of joins [Leno7]. For more information about the use of views, see 4.2.6. The key of this view is a JSON array composed of the outline's ID and a number. This number is 0 for documents of the `outline` type, and 1 for documents of the `line` type. Since the keys influence the collation (for the sorting order) of the lines, the first element of the resulting array will always be the outline. Only then, all associated lines follow. The document is the value of any given array element. With the definition of this result, the application can now return the outline with all its associated lines.

The view is shown in listing 7.4. It is queried using the outline ID as key parameter. This way, only this outline and the lines belonging to it are returned: `http://localhost:5984/doingnotes/_design/doingnotes/_view/notes_by_outline?key="01234567890"`. In this example, the outline's ID is „01234567890".

```
function(doc) {
  if (doc.kind == "Outline") {
    emit([doc._id, 0], doc);
  } else if (doc.kind == "Line") {
    emit([doc.outline_id, 1], doc);
  }
}
```

Listing 7.4: View to return all lines belonging to an outline

Now that an approach for data modelling has been found, there is still the problem of how best to display the sorting and indentation of the lines. This will be discussed in the following section.

## 7.3 Implementation of line sorting and indentation

An efficient way (meeting the requirements set in section 7.2.1) has to be found to store the order and level of indentation in an outline. The goal is to map the structure of the structure represented by figure 7.4. There are several plausible solutions to this problem. It should be decided which is more useful: connecting the lines (either as a concatenated list or as a tree), or storing the order within the lines. This section will discuss the advantages and disadvantages of each approach.

A solution is workable if it keeps the amount of write operations for inserting, deleting and moving a line to a minimum. Moreover, the position of any line should always be explicitly defined; lines may not claim the same place, nor may their positioning information go missing.

### 7.3.1 Indexed array

The first approach examined involves saving the lines as an unsorted list and assigning an index to each. Information about the indentation level should in this case be saved explicitly.

#### 7.3.1.1 Sorting

Listing 7.5 provides an example of the use of sort IDs.

```
{
  "text": "Vegetables",
  "sort_id": "1"
},
{
  "text": "Drinks",
  "sort_id": "2"
},
{
  "text": "Pastries",
  "sort_id": "3"
}
```

Listing 7.5: Three lines with a simple index

If an element is inserted, the element receives the `sort ID` of the following element, and each successive element is assigned a new index. This is problematic as the amount of write operations will quickly increase as the document grows in size. The most trivial solution to this problem is to assign indexes in large steps (e.g. 0, 1000, 2000). This, however, is not a scalable solution. In the worst case, multiple indexes have to be assigned very quickly when several elements are inserted in the same spot.

Another way is to use the floating point data type for the index. This solution is suggested in [And10, Chap. 24]. The new element's index value will be the mean value of the two surrounding elements' index values. In the following example of listing 7.6, a line is inserted between `Vegetables` and `Drinks`. Its index is $\frac{(0.2+0.3)}{2} = \frac{0.5}{2} = 0.25$.

```
{
  "text": "Vegetables",
  "sort_id": 0.1
},
{
  "text": "Drinks",
  "sort_id": 0.2
},
{
  "text": "Pastries",
  "sort_id": 0.3
}
```

Listing 7.6: Three lines with a float index

The advantage of this approach is that moving and inserting lines only requires a single write operation.

However, the precision of the float data type is limited. When lines are moved around often, the maximum number of figures after the decimal point is quickly reached. Multiple indexes will again be assigned when the number of lines in an outline rises, something that may be prevented by regularly setting the indexes to values with less decimals. Sadly, this is no workable solution for a distributed set-up since all users would be notified of changes to the entire outline. Additionally, this operation would lead to numerous conflicts.

[Cou09] recommends implementing the index as a string to which a character is added when a line is moved. This solves the problem of float precision, but the string length would also grow drastically, leading to the very same problem as with floating points.

The advantage of these approaches is that only a single write operation is needed for moving or inserting a line.

#### 7.3.1.2 Indentation

If lines are stored as a sorted list, they should also contain information about their level of indentation. This information can be supplied to the DOM when the outline is displayed.

There are two ways to solve this. A line either contains an attribute with information about how deeply it is indented (`"indent": 0`, `"indent": 1`, `"indent": 2`), or its indentation difference

from the previous line is indicated as a number (`"indent": 0`, `"indent": 1`, `"indent": -1`). A drawback of the latter method is that all successive lines have to be changed when a line is indented.

The downside to both is that, when indenting or outdenting a line, all successive lines with equal or deeper indentation have to be indented or outdented. For some outlines, this may severely increase the number of write operations.

### 7.3.2 Concatenated list

Alternatively, a singly-linked list may be used rather than an indexed array. A linked list is a data structure in which objects are ordered uni-dimensionally. Contrary to arrays, where the order is controlled by the array index, linked lists order their elements using pointers in every object [Coro1, Chap. 10.2].

This deals with the problem of index (re-)assignment when elements are moved or inserted. Inserting a line does indeed require one extra write operation to save the line, since the new predecessor has to point its pointer to that line. Yet this is the maximum number of writes, irrespective of the outline's complexity and length. Moving a line also takes no more than two write operations.

However, this approach means that the indentation of a line has to be implemented exactly the same way as in indexed arrays (see section 7.3.1.2), the drawbacks of which have been mentioned.

### 7.3.3 Tree structure

#### 7.3.3.1 Terminology

Arguments for mapping the data as a tree structure are the disadvantages of other methods mentioned above, especially when indenting a sorted list. Tree structures are amongst the most important data structures in information technology. According to [Knu97], they are the most important non-linear data structures in computer algorithms. It is impossible to identify their creator [Gü04, p. 89]. Tree structures have a branchlike relation between their nodes.

The terminology for the elements of trees is not standardised. [Knu97] uses the following terms: every root is the *parent node* of the subtree's roots. Directly neighbouring nodes are *siblings*, and *children* to their parents. Relations between nodes that cover multiple of the tree's levels are described using the terms *ancestor* and *descendant*.

According to the classification in [Knu97, Chap. 2.3.4], the tree structure adopted here is a *finite labeled rooted ordered tree*. Every node in such a tree needs to have a parent node; the node that

does not is called the root. Cyclical relations are not allowed; there is only one path from the root to each of the individual nodes.

#### 7.3.3.2 Implementation

[Coro1, Chap. 10.4] presents how a tree structure can be realised with the aid of pointers. The method that is best suited to deal with the problem at hand is a so-called *left child, right sibling*-representation. Every node is identified with its own ID. Every node also contains a pointer to its leftmost child that, when represented vertically, corresponds to the root above its first child. Finally, every node contains another pointer to its right sibling, i.e. the next node on the same level.

The benefit is that no more than two write operations are needed for any kind of operation. This includes the indentation or outdentation of an entire block of lines.

### 7.3.4 Conclusion

After weighing the benefits and drawbacks, it was decided to create the application using a tree structure. The unchanging number of write accesses tipped the scales.

Using a view, the outline and all associated outlines can be retrieved in one time, as was already indicated in section 7.2.7. In the context of this application this is desirable. The lines are not just loaded when needed, e.g. when expanding lines. As specified in section 6.1.1.2, the outliner should mimic a text editor that presents the entire document at a glance, and that only hides certain sections if the user so desires. The output of an outline's lines requires a recursive function to be built.

With just a slight modification, the process described in 7.3.3.2 can be of use here. Rather than a pointer to its first child, every node contains a pointer to its parent node. This means that only the node being indented or outdented needs to be changed, whereas the new parent node can remain unchanged. Figure 7.5 represents the resulting pointer structure.

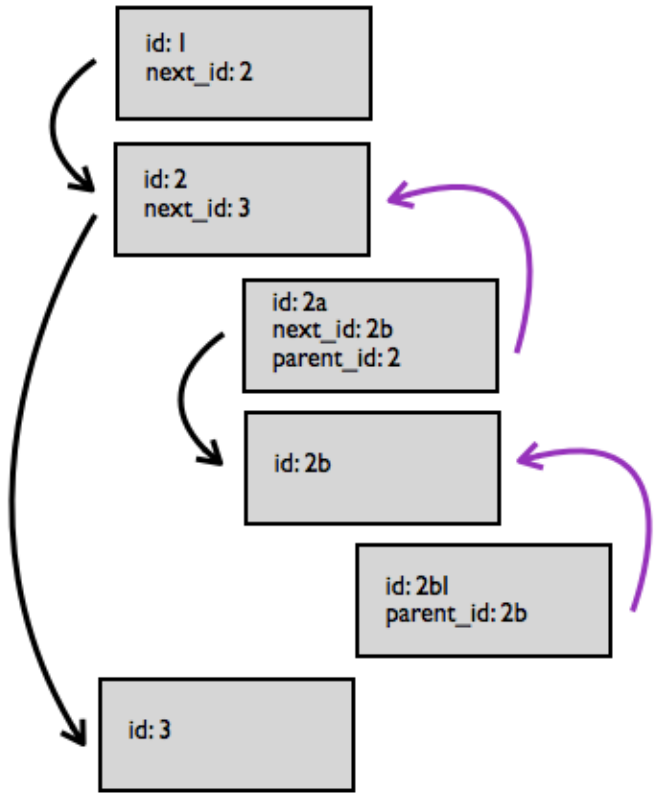


Abbildung 7.5: The pointer structure

Implemented with the pointer structure, the example outline from figure 7.4 will finally look like in listings 7.7 and 7.8.

```
{
  "_id": "01234567890",
  "kind": "Outline",
  "title": "Shopping list"
}
```

Listing 7.7: Chosen implementation of an outline

```
{
   "_id": "111",
   "kind": "Line",
   "text": "Vegetables",
   "outline_id": "01234567890",
   "next_id": "333",
   "first_note": true
},
{
```

```
    "_id": "222",
    "kind": "Line",
    "text": "Carrots",
    "outline_id": "01234567890",
    "parent_id": "111"
},
{
    "_id": "222",
    "kind": "Line",
    "text": "Drinks",
    "outline_id": "01234567890"
}
```

Listing 7.8: Chosen implementation of three lines

This solves the data modelling question. Now all that remains to be determined is how to deal with conflicts that will inevitably occur.

## 7.4 Conflict resolution

As early as 1996, Leslie Knieb developed a distributed database system that allowed distributed storing of data without a locking mechanism [Kli96]. Moreover, the system featured data merging and conflict resolution. Even if the implementation differs from the technology presented here, the requirements are much the same.

Knieb identifies three cases of data synchronisation that do *not* lead to conflicts: changes to a document, creation of a document and deletion of a document [Kli96, Chap. 3].

In CouchDB, deletion is regarded as an update to the document. The document is not actually deleted; it simply contains a `deleted=true` attribute. So „deletion“ will not be regarded as a special case, but rather treated under „update“. This reduces the number of cases in which conflicts may occur.

The following part discusses the remaining conflicts to be resolved, and how to deal with them. „Simultaneous“ below refers to the time span between two replications of two or more replicas.

### 7.4.1 Simultaneous insertion of a line

If a new line is created, the line is always conflict-free. However, if several users simultaneously insert a new line in the same place, the previous line (hereinafter simply `previous`) contains a conflict since its next-pointer is changed. This conflict is called an *append conflict*. There are two versions of `previous` that each refer to one of the two newly inserted lines.

An append conflict can be resolved by letting the system store the two new lines into the outline, sorted chronologically and in descending order. The crucial element is the time stamp in the line document. Chronological ordering is suitable for many use cases and it is meaningful to the user. However, since there is no telling whether the clocks on both clients are synchronised, it is primarily a mechanism that offers a single way to resolve this conflict on different clients. It helps avoid situations where a conflict is solved simultaneously but in different ways on two clients, which would lead the system to mistakenly re-recognise a conflict on both clients.

### 7.4.2 Simultaneous change of a line

If the text of a document is changed simultaneously by more than one user, conflicts will arise. Even if the new text is the same in both versions, the revision numbers will still differ. This conflict type is called a *write conflict*.

In order to resolve this, instead of the conflicting line, the user is presented with a form that lets her choose from the conflicting versions. She can pick one out and also edit it, i.e. the user can also manually merge both versions.

### 7.4.3 Further conflict types

A special case of change to a document is indentation. For instance, if a user indents a line, this will also become apparent in the documents of other users after the document has been replicated. If, however, a line is indented by one user and outdented by another, it remains to be decided how to present this conflict to the users. A standard solution will depend on how the outliner is actually used.

There are also hybrid types, e.g. if a line is changed and the next is indented. If a write conflict occurs while editing a line and the older version is chosen to resolve the conflict, the line's altered next pointer should be taken into consideration.

Conflicts that happen because of indentation will not be dealt with during the development of the prototype. Also, in order to make things easier, it is assumed that conflicts only occur between two versions. Although not unrealistic in everyday use, there is not enough time to cover this in this thesis. Cases in which an append and a write conflict occur in the same line will however be correctly dealt with.

### 7.4.4 Notification

The user should be notified when a conflict has occurred after replicating the outline that is currently being edited. This should happen immediately after replication without refreshing the page or interrupting the user in her work. This requirement was set in chapter „Analysis“ (section 3.4.2). Only when the user expressly decides to deal with the conflict, she has to start conflict handling explicitly. The implementation of the notification will be described detailedly in section 8.6.

## 7.5 User interface

This section will motivate the technologies used in the implementation of the user interface and their architecture.

### 7.5.1 Strategies

The user interface is to be constructed using simple HTML elements. Complex application frameworks should be omitted; only the jQuery library will be used to make development easier. The use of Adobe Flash was also rejected for the implementation of the outliner. The reason for this is that a DOM tree only uses open web standards in order to better meet expandability and accessibility requirements.

It is easy to build modified or alternative user interfaces for a standards-conform architecture. For example, it would be easy to build a screen reader that reacts to function keys and reads the contents of the lines to the user. The use of CSS makes it easy to change the application's appearance by changing style sheets. The visually impaired can change contrast and font sizes at will. A plug-in that re-formats the outline for print is also conceivable; all it needs to do is traverse the DOM and re-format the individual elements.

Using a Flash or a Sproutcore widget [Spr10] means that the outliner would be a single object, embedded in the site. The contents would thus not be accessible to other technologies. The loading times can also be held at a minimum since the jQuery libraries and the application code are relatively small in size.

### 7.5.2 Site layout

The application has three views: an overview with a list of all outlines allows the user to change to the single-outline view, which at the same time is the outliner. The third view is the outline edit

view where the outline title can be changed or the entire outline deleted.

Newsworthy status information, i.e. notifications, errors or success messages should be shown on the site. They should not be implemented as pop-ups in order not to distract the user from her work. Instead, an HTML element should be shown on the page directly after the event and faded out after a short while by a JavaScript function.

While the application waits for a response from the local database, an animated graphic (a so-called *throbber*) should indicate that the program is working. This provides the user with feedback which eliminates much of the frustration that occurs when the user has to wait a few seconds [Mil68].

### 7.5.3 Editor

The application's core piece is the outliner. It consists of a single page and does not, unlike conventional editors, contain a single text input area. The editor much rather consists of a multitude of text areas. Thus, lines can be sent as individual forms. The text area may be edited at any given time, jumping between them can be done using function keys or the mouse. When a text area loses focus, its contents will be saved immediately. The tree of DOM elements should appear as a single editor window to the user.

### 7.5.4 Interaction

When the editor is used, it does not interact with the system in the way HTML documents typically do: when the user clicks, the page is requested and loaded from the server. Instead, only some elements on the page will change in response to most interactions. This is done according to the AJAX concept. Both approaches have been more closely examined in 5.1.3.2, also see figures 5.2 and 5.3. Following the AJAX method implies that resources are displayed in different states under a single URL. This is contradictory to the REST paradigm presented in section 4.1.5, which requires every resource state to have its own address.

The REST concept is breached for a good reason here. If, for instance, a line is added, the corresponding keystroke will change something in the DOM and provoke an HTTP request. This DOM manipulation is, however, minimal and can be performed by a simple JavaScript function. Moreover, the line has to be stored into the database at once. It is crucial that changes to the DOM happen immediately, in order to ensure uninterrupted working. If the user has to wait for a server round-trip in order to see the result of her action, it would take a disproportionally long time.

# 8 System documentation

This chapter will describe the final result of the implementation of the system that was presented in the previous chapter. To start with, the application's code structure will be commented on. An overview of the entire system is presented with the aid of a class diagram and an introduction to the routing scheme and data structures. Furthermore, the modules that were developed for the realisation of the user interface, replication, conflict detection and conflict handling are presented. Afterwards the implementation of the test suite will be discussed.

Section 5.2 already introduced the cloud computing used for the deployment; this chapter's conclusion presents an illustration of how the application was deployed using cloud computing and Amazon Web Services. Finally, the clustering framework CouchDB Lounge was introduced which was used to optimise the deployment in terms of scalability.

It is not possible to comment on all the source code in this thesis. Instead, the different layers of the system will be presented. Only technically very complicated or significant algorithms or functions will be investigated. Short source code snippets are contained within the text, longer ones are kept in the appendix.

## 8.1 Structure of the project

In order to convey an overview over the code's structure, this section will explain the contents of the folders in this project. The individual classes and functions will be discussed more closely in the following sections. Their meaning was already shortly explained in the sections 4.2.5 and 5.1.1.

**_attachments:** Contains those parts of the applications that can be executed directly in the browser, as well as the start page (`index.html`).

**app:** Contains business classes and the file `application.js` in which the routing framework Sammy.js is initialised. All resources from the *helpers* and business classes are loaded here. The initialisation function `init` that is called here binds certain behaviour to window, mouse and keyboard events. The replication is also started from here.

**_controllers:** The controller's task is to define the routes introduced in 8.2 and their behaviour.

**_helpers:** Helpers contain functions that are no methods of business classes in the narrow sense. These functions are mainly in charge of certain parts of the interface's behaviour. For example, the keyboard events used in the `init` functions are defined here. Line traversal functions are also stored here.

**_lib:** Contains self-made libraries that extend JavaScript's functionality. In `resources.js` functions are abstracted by which the controllers perform read and write operations on the database.

**_models:** Contains the definition of the `Outline`, `Note` and `NoteCollection` „classes". They are *models*in the sense of *object/relational mapping*. They are described in detail in section 8.3. The functions for conflict detection, presentation and resolution are also defined here.

**_templates:** Contains HTML templates for the template engine Mustache.js. These partials are put together to construct the site.

**_views:** No CouchDB views are meant, but rather the representation of the `Outline` and `Note` models for the application's business logic and for rendering. E.g. an `OutlineView` contains an `Outline` object and prepares its data for the Mustache templates. The controllers never directly access the models, they only access the view representations. This is exemplified in the class diagram in fig. A.9.

**config:** The application's URLs for the replication service as well as the database name can be entered into the configuration file `config.js`. The subdirectory **features** also contains configuration files for the test environment.

**images:** Small graphics that are needed for the layout can be found here.

**spec:** Contains the unit tests and the unit test framework. They are described in section 8.8.1.

**style:** Contains style sheets that are self made or inherited from the Blueprint framework.

**features:** This directory contains integration tests. They are described in section 8.8.2.

**filters:** Contains filter functions with which the database can be monitored for changes and conflict status.

**Rakefile:** Macros with which the application may be put into certain conflicting situations (section 9.2.4) can be found here.

**README:** A summary of the installation manual (section 9.1) and the user's guide (section 9.2).

**vendor:** The included libraries are saved in this directory. The individual JavaScript files have to reside in a `_attachments` subdirectory so that CouchDB is able to execute them.

**views:** Contains the CouchDB views with which readily formatted data can be requested. The application needs map functions from a total of three views.

The figure in section A.5.1 shows a business class diagram that gives an overview over the core classes.

## 8.2 Routing

This section will describe the application's URL scheme. In doing so, also the URL schemes of CouchApps and applications that use the routing framework Sammy.js will become clear. These technologies were presented in section 5.1.1 and 5.1.4.

The start page can be found under the URL `http://localhost:5984/doingnotes/_design/doingnotes/index.html#/`. After the server and the port through which the application can be accessed the database name is indicated. The prefix `_design/` marks the beginning of a design document resp. the name of an application that is also called `doingnotes`. Since the `index.html` file is stored directly in the design document's `_attachments` folder, it can be accessed from within the design document. The slash after the HTML anchor forwards to a Sammy route with the path `#/`.

Further routes are defined in the controllers; they will be explained here. The route with the path `#/outlines` initialises a new `OutlinesView` that renders a list of all outlines. When `#/outlines/new` is requested, it will display a form with which a new `Outline` can be created. The route `#/outlines/edit/:id` shows a form with which the outline's title can be changed. `#/outlines/:id` shows an outline; this is where the outliner is. Further routes have `PUT`, `POST` or `DELETE` methods and are as such transparent to the user.

The author created a plug-in in which the basic database operations *create*, *read*, *update* and *delete* are abstracted. This way, recurring tasks do not have to be re-implemented in every route, but can be partially re-used for `Notes` and for `Outlines`. In listing A.1 an extract from the plug-in can be found in which, amongst others, the methods `new_object` and `load_object_view` are defined. `new_object` receives the object type (e.g. `Outline`) and a callback-function. The latter is executed after the template for the corresponding object was loaded. `load_object_view` requires the object's ID as a parameter. The document with this ID is requested from the database and a view object is created. This view object can be used to render a template, as shown in the following example, which is taken from the route `#/outlines/edit/:id`:

```
load_object_view('Outline', '123', function(outline_view){
  context.partial('app/templates/outlines/edit.mustache', outline_view, function(
      outline_view){
    context.app.swap(outline_view);
  });
});
```

Listing 8.1: Rendering of the outline editing template

## 8.3 Data structures

As already explained in section A.5.1, business classes are JavaScript functions that save attributes as local variables that correspond to the database fields. Methods are implemented by extending the function's prototype with the appropriate functionality. The following part will explain the application's data structure by looking at the structure of the CouchDB documents.

### 8.3.1 Outline

An `Outline` represents a file that can be edited in the outliner. Apart from `_id` and `_rev`, it also contains the data type `Outline` and the title (`title`). Timestamps mark the document's creation (`created_at`) and the latest change to it (`updated_at`). The latter is created only when the document's title is changed. The timestamps are generated on creation of the object using the command `new Date().toJSON()`. They are used to chronologically order the outlines in the outline overview.

```
{  "_id": "ce63ec5aaf501c567d200d89f200088a",
   "_rev": "2-00899e40fef865bb3fa294cd72860b8f",
   "created_at": "2010/07/04 12:12:52 +0000",
   "updated_at": "2010/07/04 12:28:39 +0000",
   "kind": "Outline",
   "title": "My Shopping List" }
```

Listing 8.2: An Outline document

### 8.3.2 Note

A `Note` represents a line in the outline. Similarly to the outline it contains the fields `_id`, `_rev`, the data type `Note`, `created_at` and `updated_at`. The timestamps are used for the order

of the lines inside an outline when the order has to be determined by the system after replicating (see section 8.7.1).

The contents of the lines are saved in the `text` field. `source` is used for the notification after replicating (see section 8.5). With the aid of the `outline_id` field, it is possible to determine to which outline a line belongs. The last three fields are optional: `next_id` and `parent_id` are used to render the tree structure inside an outline. This is examined more closely in section 8.4.3. `first_note` is a Boolean and as such the only field whose data type is not a string. It marks the first line in a document so it can be found more easily when traversing.

```
{
   "_id": "ce63ec5aaf501c567d200d89f2001b08",
   "_rev": "5-86d6c6ce0ad7b6b8454cbb91590e315c",
   "created_at": "2010/07/21 23:55:35 +0000",
   "updated_at": "2010/07/21 23:56:08 +0000",
   "kind": "Note",
   "text": "This is the text within one line",
   "source": "eb8abd1c45f20c0989ed79381cb4907d",
   "outline_id": "ce63ec5aaf501c567d200d89f200088a",
   "next_id": "ce63ec5aaf501c567d200d89f2002a87",
   "parent_id": "ce63ec5aaf501c567d200d89f20015ab",
   "first_note": true
}
```

Listing 8.3: A Note document

## 8.4 User interface

This section describes the implementation of the user interface. Successively, it will discuss the implementation of the outliner, the operations save, insert and indent, their effects on the DOM and the database, and the rendering of lines after an outline is re-loaded.

### 8.4.1 Implementation of the outliner

The outliner is displayed in the DOM as a `<div>` element that contains an unsorted list (see listing 8.4). The list's `<li>` elements are the outline's lines. If a line has child nodes, i.e. if there are indented lines below it, an additional `<ul>` element is inserted in the first line's `<li>`, which in turn contains further `<li>` elements. The result of such an indentation can be found in listing A.2.

```
<div id="writeboard">
  <ul id="notes">
    {{#notes}}
      <li class="edit-note" id="edit_note_{{_id}}">
        <form class="edit-note" action="#/notes/{{_id}}" method="put">
          <span class="space"> </span>
          <a class="image"> </a>
          <textarea class="expanding" id="edit_text_{{_id}}" name="text">{{text}}</
              textarea>
          <input type="submit" value="Save" style="display:none;"/>
        </form>
      </li>
    {{/notes}}
  </ul>
</div>
```

Listing 8.4: Template for the outliner in Mustache syntax

### 8.4.2 DOM modification

The initialisation function assigns certain behaviour to the text areas that represent the outliner's lines. Certain window, mouse or keyboard events will trigger this behaviour. Saving, inserting and indenting lines can be done this way.

A line should always be saved when it loses mouse focus, whether this is caused by a keystroke, the mouse or the closing of the window. The line should not be saved if its contents did not change as compared to the text stored in the database. This is realised with the aid of a custom data attribute, as described in section 5.1.2. When the user navigates into a line, the method `setDataText` is invoked for the `NoteElement`, i.e. a line's representation in the DOM. This temporarily saves the current content of the line, so that it can be compared to the text when the element loses focus again. If both values are identical, the saving process can be skipped.

If a line has the focus and the enter key is pressed, the `insertNewNode` method generates a new Note object. The callback function fills the values into the partial for the new line and inserts it into the DOM using a jQuery method. Moreover, several pointers are adjusted, as explained earlier in section 7.3.3.2: The `next_id` of the line to which the new line was attached and resp. also the `parent_ids` of any succeeding lines have to correspond to the modifications in the DOM.

A similar thing happens when lines are indented or outdented: here, too, the pointers of preceding and succeeding lines and those of any parent node and its successors have to be adjusted. In the worst case, the indenting of a line may lead to up to two further write operations. In the DOM,

the indenting is realised by wrapping the line's `<li>` in a `<ul>`, and then inserting it into its new parent node.

### 8.4.3 Rendering the tree structure

The previous section sketched what happens to the DOM and the database when the user interacts with the outliner. However, if an outline is opened or refreshed, the lines have to be rendered all at once. All of the outline's lines are retrieved at once from the database as described in section 7.2.7. In the resulting array of lines, the first line is determined; for this it is marked with a special attribute. Starting here a recursive function traverses the tree. *Traversing* means „the examination of the tree's nodes in a certain order" [Knu97, Chap. 2.3]. All nodes are systematically examined so that each node is visited exactly once. After traversing a linear copy of the nodes is available.

The function `renderNotes` receives the array that contains all of the outline's lines and a counter. The counter's initial value corresponds to the length of the array and is decremented by one with every pass. Additionally, the line that has been examined is deleted from the array. The function then verifies if the current line contains a child node or a next pointer. If so, the line is inserted into the DOM and `renderNotes` is called again. If a line is found that contains neither child node nor next pointer, the function knows that this is the final line and terminates. The function is documented in listing A.3.

Lines may be collapsed or expanded in a rendered tree. Collapsing a line's child nodes is done by simply hiding them. A triangle bullet at the beginning of the line is turned 90 degrees to indicate that some lines are collapsed. The database stores no information about collapsed lines.

## 8.5 Replication

The functions that control the replication are contained within the `ReplicationHelpers` plug-in. They will be presented in this section.

### 8.5.1 Starting replication

The functions `replicateUp` and `replicateDown` can start continuous replication to respectively from the server. Both functions are constructed in a similar way, only source and target are swapped around:

```
1 replicateUp: function(){
2   $.post(config.HOST + '/_replicate',
```

```
    '{"source":"' + config.DB + '", "target":"' + config.SERVER + '/' + config.DB + '",
        "continuous":true}',
    function(){
      Sammy.log('replicating to ', config.SERVER)
    },"json");
}
```

Listing 8.5: The `replicateUp` function

Both functions are called in the initialisation function in order to restart the replication every time the page is manually refreshed. If replication is already running, the command is simply ignored. This way it is possible to resume replication after reconnecting to the Internet. The URLs and ports of client and server are configured in the `config.js` file.

### 8.5.2 Change notification

The system requirements stipulate that the user be notified of changes brought about by replication without interrupting the user's flow of work. The page in which the outliner is embedded contains an element with the replication status message. The element remains hidden until there are changes. Clicking the link in the message reloads the page with the outline's actual version.

```
<h3 style="display:none;" id="change-warning">Replication has brought updates about. <
    a href="javascript: window.location.reload();">View them.</a></h3>
```

Listing 8.6: Change notification

In order to verify whether there are any changes, the `source` field is reset every time a line is saved. It contains a hash of the value returned by `window.location.host`. This string allows unambiguous identification of the URL and port of the system where the line was changed. Hashing prevents personal data from being stored in the database.

If an outline is shown, the function `checkForUpdates` is invoked for the outline. The changes feed is retrieved and all lines with a foreign `source` are filtered out by the filter in listing 8.7. For example, if the hash of the user's own host name equals „848c7“, this is done using the following URL: http://localhost:5984/doingnotes/_changes?filter=doingnotes/changed&source=848c7.

```
function(doc, req) {
  if(doc.kind == 'Note' && doc.source != req.query.source) {
    return true;
  }
  return false;
}
```

Listing 8.7: `changed` filter for the changes feed

The purpose of this request is to retrieve the database sequence number at the time of the latest foreign change, because changes only count as new after this point in time (see section 4.2.4). The sequence number is saved in the variable `since`. Another XMLHttpRequest is then sent to monitor the database for changes after this point in time. It is also necessary to supply the `heartbeat` parameter, which sets the interval in milliseconds for CouchDB to send line breaks. This prevents the browser from mistakenly thinking that the connection timed out. The `feed=continuous` parameter allows the HTTP connection to be kept open so that, analogous to continuous replication, changes are sent continuously.

If the sequence number is „142“, the request is sent to the following address: `http://localhost:5984/doingnotes/_changes?filter=doingnotes/changed&source=848c7&feed=continuous\&heartbeat=5000&since=142`. As soon as the returned ResponseText contains a document revision with a `changes` attribute, the document is opened and the notification introduced earlier is displayed. The function in charge for this functionality is documented in listing A.4.

Against the use of changes feeds in `continuous` mode can be said that it is currently only reliably supported by the browser Firefox. All other browsers ignore this option or react with erroneous output. However, since this is bound to change in future releases of other browsers, this limitation could be put up with.

## 8.6 Conflict detection

The database should be monitored not only for changes but also for conflicts. Again, the user should be notified immediately so conflicts may be resolved as soon as possible.

The `ConflictDetector` module - as well as the `ConflictResolver` presented in the next section - is implemented as a singleton. The `checkForNewConflicts` method uses - as described in the previous section - the changes feed in continuous mode in order to scan for changes. The filter function is used here to filter for conflicting lines. If a line is found with a `_conflicts` array and its `outline_id` corresponds to the outline currently being displayed, the first conflicting version of the line is loaded. The conflict type is identified and both competing revisions are passed on to the `ConflictResolver` or the `ConflictPresenter` for further handling.

If conflicts are not solved immediately, they should be shown again immediately after re-opening the outline. For this purpose, the `ConflictDetector` temporarily stores these conflicts.

When the outline is opened, the `checkForNewConflicts` method is invoked, which verifies if the stored conflicts belong to the currently open outline and passes them on to the `ConflictPresenter`. A hurdle that needed crossing was that changes in a document's `_conflicts` array were not identified as a change for CouchDB's changes feed. In order to solve this problem, a patch for the changes feed was written. This patch [Apa10h] is documented in section A.5.6 and in [Apa10a].

## 8.7 Conflict handling

In the application's prototype two different conflict types were dealt with: append conflicts and write conflicts. These have been defined in section 7.4. This section will discuss how these conflict types and their hybrid form are processed and resolved.

### 8.7.1 Append conflicts

The `ConflictDetector` checks a conflicting document's competing revisions to determine the kind of change that was made. If the revisions differ in `next_id`, an append-conflict is at hand in the following line. This conflict can be resolved automatically by the application. Both revisions are passed on to the `ConflictResolver`. The `solve_conflict_by_sorting` function loads the lines that follow the two conflicting revisions. These are then sorted chronologically and inserted into the tree structure. As a result of this operation they are positioned one below the other. Similarly, the pointers in the surrounding lines are adjusted to correctly reflect the new tree.

In the user interface, the lines are accordingly inserted in the right order into the DOM. A message that reads „*Replication has automatically solved updates.*“ is displayed for a few seconds. The lines that have been changed are shortly shown against a bright green background.

### 8.7.2 Write conflicts

If the `ConflictDetector` finds out that the contents of competing lines are different, the conflict is a write conflict. Both revisions are passed on to the `showWriteConflictWarning` in the `ConflictPresenter`. This function displays the conflicting lines against a bright red background and shows a notification like the one that is displayed after an automatically succeeded replication:

```
1 <h3 style="display:none;" id="conflict-warning">Replication has caused one or more
    conflicts. <a href="/doingnotes/_design/doingnotes/index.html#/outlines/{{
    outline_id}}?solve=true">Solve them.</a></h3>
```

Listing 8.8: Notification of conflicting database state

The notification contains a link to the current outline with the parameter `solve=true`. When an outline is loaded with this parameter, the `ConflictPresenter` is again invoked. The method `showConflicts` loads all conflicting lines with the aid of a CouchDB view. For every line both competing database revisions are requested. Afterwards, every conflicting line's text in the DOM is replaced by a partial, presenting the texts of both revisions so that the user may decide how to proceed. The text area is replaced by two forms, each of which contain the text of one revision. Every form contains a save button that triggers a PUT request to the route `#/notes/solve/:id`. The route's callback function instructs the `ConflictResolver` to use the `solve_conflict_by_deletion` function in order to create a new revision of the line that was chosen by the user, including any manual changes. This revision is saved and both old revisions are deleted.

If several conflicts need resolving, saving one version of a line will hide both forms, replacing them by a normal line with a text area. When the final conflict is resolved, the site is reloaded without the `solve` parameter. Also, the monitoring for conflicts will be resumed after being temporarily suspended while resolving write conflicts.

### 8.7.3 Resolving hybrid conflict types

It is possible that a line's text is changed and a line is inserted after it at the same time. In this case, the append conflict will be resolved first, according to the process described above. Afterwards, the write conflict is artificially created again by CouchDB's `bulkSave` method. If a document is stored in this way, stating the `all_or_nothing:true` parameter, CouchDB will accept changes even if this causes the database state to be conflicted. In addition to reporting the resolved append conflict, a second notification will inform the user of write conflicts that need resolving.

## 8.8 System test

The sections 5.3.1.2 and 5.3.2 discussed test-driven development as well as the testing frameworks JSpec and Cucumber. This section will examine how these technologies can be used to create a test suite.

### 8.8.1 Unit tests

The unit test framework JSpec, which was introduced in section 5.3.2.1, was used to extensively test the logic contained within business classes and helpers. The tests are distributed over multiple files according to module and functionality. The files contain tests for the following functionality:

**note_element_spec:** Traversing and automatic line storage

**inserting_note_element_spec:** Line insertion

**indenting_note_element_spec:** Line indentation

**unindenting_note_element_spec:** Line outdentation

**focusing_note_element_spec:** Focusing while navigating in the outliner

**rendering_note_element_spec:** Rendering of the tree structure when opening an outline

**outline_spec, outline_helpers_spec:** Outline sorting and rendering

**note_spec, note_collection_spec:** Locating certain lines during the outline rendering process

**resources_spec:** Abstraction of database interactions

**lib_spec:** Extensions for the string and array data types

**conflict_spec:** Display of a line for the resolution of a write conflict

The test suite is executed by loading the file `/_attachments/spec/index.html` in the browser. The file is documented in listing A.7. It starts by loading the JSpec libraries: the testing framework, the code that needs testing and any other code that is required to run the code that needs testing are loaded. Afterwards it defines the function `runSuites()`. This function calls the `exec` method on the `JSpec` object -which is made available by including the JSpec library- with every of the aforementioned files as its parameter.

Finally, the running of the tests and the output of the results are invoked. This invocation also includes information about the position of the *fixtures*. Fixtures are files that contain HTML blocks. Since the JSpec tests do not run on the database, but merely test the JavaScript functions, a certain DOM state has to be simulated this way. The fixtures are located in the `/_attachments/spec/fixtures` folder; they contain outlines in multiple states and an HTML representation of the start page. The tests use these HTML pages as the production code would the generated DOM.

The test results are output in the browser. The output may also contain information about failed tests, the number of tests and the time it took to complete them (in figure 8.1, this is just over three seconds).

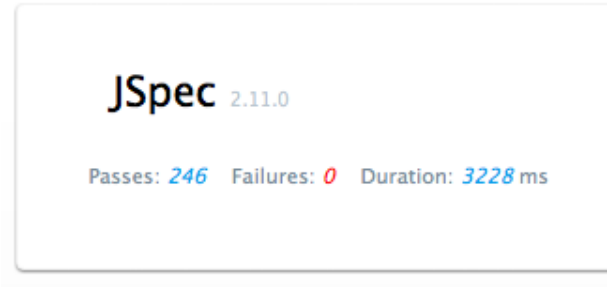


Abbildung 8.1: JSpec: Successful completion of all unit tests

### 8.8.2 Integration tests

The Cucumber framework was already introduced in section 5.3.2.2. It is usually used in combination with the *Webrat* library [Helo8]. Webrat implements a browser simulator that admittedly does not do JavaScript. Since the application was completely written in JavaScript, the integration tests had to be implemented using a set-up consisting of *HTMLunit* [Gar], *Culerity* [Lan] and *Celerity* [Bak].

HTMLUnit is a Java library that can parse HTML and execute JavaScript code. HTMLUnit is often regarded as a headless browser [Lano9b] since it has the same abilities as a browser, but lacks the user interface in which the pages are displayed. The pages that are read or executed by HTMLUnit can therefore not be seen anywhere. The test suite at hand requires HTMLUnit 2.7 or higher.

Celerity is a JRuby wrapper for HTMLUnit. It has an API for the most common browser functions, which are then executed in HTMLUnit. Culerity is a Ruby gem, which connects Cucumber to Celerity even if the code is not executed in a JRuby environment. When Culerity is used, Celerity starts a Java process to which all Celerity requests are redirected. In the Ruby environment, the results are output as if they were produced by a single Ruby process [Lano9b].

Culerity features a set of common step definitions. These are stored in the folder `/features/step_definitions/common_culerity_steps.rb`. Further step definitions can be found in the same directory.

Section 5.3.2.2 already specified some examples for features/scenarios (see section 5.8) and step definitions (see section 5.9). These were taken from the application's test suite. Further features are stored in the `/features` directory. Figure 8.2 is what it looks like when all scenarios passed successfully. The tests were completed in approximately one and a half minutes. The functions tested were: creation and deletion of an outline; changing the title of an outline; chronological ordering of the outlines in the overview; inserting, editing, indenting and outdenting a line.

Culerity does not regard it as a page change if the anchor part of the URL changes. To ensure that it does notify these kind of changes so that Sammy routes may be tested, it is necessary

```
  Scenario: delete an outline
[INFO] Visit your CouchApp here:
http://127.0.0.1:5984/doingnotes_test/_design/doingnotes/index.html
    Given an outline with the title "Songs"
    And a first note with the text "Rambling" and the id "123" for "Songs"
[INFO] Visit your CouchApp here:
http://127.0.0.1:5984/doingnotes_test/_design/doingnotes/index.html
    And I save
    When I go to the start page
    And I follow "Songs"
    And I follow "Change title or delete this outline"
    And I press "Delete this outline"
    Then I should see "Outline deleted."
    And I should not see "Songs"
    And I should see "You have no outlines yet"

9 scenarios (9 passed)
122 steps (122 passed)
0m55.681s
```

Abbildung 8.2: Cucumber: all tests pass

to explicitly execute the route that corresponds to the URL part beyond the anchor. Since this change in Sammy's functionality will deactivate the browser's „back" function, the behaviour will only be overwritten in the test environment. This is why the test environment is toggled by calling the `setTestEnv()` function in `test_environment.js` using the command `$browser.execute_script("setTestEnv();")` for every first step in a scenario. This way, integration tests may be run without impairing the application's behaviour.

### 8.8.3 Test suite for CouchDB's HTTP API

The application was developed with the aid of CouchDB's JavaScript HTTP API. This API wraps around basic database functions, which makes formulating database requests easier for developers. Requests can be made as simple JavaScript or jQuery method calls without even having to create an XMLHttpRequest. Since there were no tests available for the API and since some parts of the API were clearly faulty, a test suite was created. It was published under the Apache Licence [Apa10f]. Examples taken from the test suite and the API can be found in section A.5.8. Additionally, improvements to the API were made [Apa10e, Apa10g].

Analogous to the test suite for the application, the API tests are executed in the browser. Unlike the aforementioned test suite these tests do access the database, which is why they have to be run by the latter. This means that the files cannot simply be opened in the browser. They have to be located in the `/share/www` directory of the CouchDB installation and then compiled along

with it. The tests can then be accessed by visiting the URL http://localhost:5984/_utils/spec/run.html.

## 8.9 Deployment using Amazon Web Services

The application should not merely be able to „scale down", as stipulated in the introduction (section 1.1, it should also perform well: Even if a large number of users simultaneously synchronise their outlines to the server, this should remain possible without increase in latency. If the CouchDB instance on the server should become temporarily unavailable, the application's availability should still be guaranteed. This can be effortlessly realised using Amazon Web Services.

In the context of prototype development, it was examined how the application might be deployed using the *Amazon Elastic Compute Cloud (EC2)*. Cloud computing's technical background was already discussed in 5.2. This section will give an overview of the configuration of an application that was deployed using EC2. The overview is supported by own research and [Bau10, Chap. 4.1].

AWS is an umbrella term for all cloud computing services offered by Amazon. Amazon experiences strong seasonal variation in the demand for its services. Therefore, the majority of these sizable IT resources is not being used for most of the time. The AWS service results from a business idea that makes those free resources available for money, when they are not needed for Amazon's own products.

EC2 allows users to manage virtual servers via Web Services. Such a server is created by following a few steps. These steps are documented by means of screenshots in the appendix (section A.5.9.

A *key pair* is generated after creating an Amazon account. This key pair is used to authenticate the user against the EC2 instance (see fig. A.10). The public key is associated to the Amazon account; the private key stays on the user's computer.

Additionally, a *security group* must be defined and configured (see fig. A.11). Every EC2 instance belongs to such a security group. These groups define the security settings. Individual ports for accessing the server may be opened using the public key generated in the previous step.

The next step involves choosing an *availability zone*, i.e. a geographical region where the server should run. For larger set-ups, distributing over several zones is a way of protecting against the failure of an entire region. It also benefits the latency.

Now, the *amount of resources* required can be defined (see fig. A.12). Several packages are available, varying in processing power, memory and disk space. The packages currently range from 1,7 GB RAM / 160 GB hard disk space to 68,4 GB RAM / 1690 GB hard disk space [Amaa].

Finally, an *Amazon Machine Image (AMI)* must be chosen (see fig. A.13). An AMI is a virtual image, i.e. a snapshot of a virtual server. The different AMIs have varying operating systems and contain different software packages. It is possible to create own images for future (re-)use, and may also be published for a fee. For the testing purpose, however, it suffices to create a virtual server using a ready-made AMI. An up-to-date Ubuntu distribution was chosen, „alestic's 64bit server Ubuntu 9.04 AMI" with ID „ami-ccf615a5".

The EC2 instance is started with the aforementioned parameters. The new virtual server automatically possesses a public IP, under which it can be reached over the Internet, and a private one, which can be used to communicate with other instances. These IPs are re-leased every time the server starts. It is, however, recommendable to set-up an *Elastic IP* (see fig. A.14). This static ip address can be linked to the server, so the IP address remains the same after every restart.

The instance can normally be managed using the *AWS Management Console* (see fig. A.16). It is available under `https://console.aws.amazon.com/ec2/home`. It is also possible to manage the server using the command-line interface; if the public key is saved in `.ssh/doingnotes.pem`, it is also possible to log in using `ssh -i .ssh/doingnotes.pem root@ec2-185-73-233-128.compute-1.amazonaws.com`. As soon as the server is set up, CouchDB can be installed like on a normal Ubuntu computer (see the manual in section 9.1).

As soon as the instance is terminated, all settings and installations that have been made on it will be deleted. In order to make these settings permanent beyond the life cycle of a virtual server, it is necessary to store the instance's state externally. This is where *Amazon Elastic Block Store (EBS)* comes in (see fig. A.15). After setting up an EBS it is mounted in the EC2 instance as an external hard disk. It can then be used to store snapshots of the EC2 server.

The price for such an EC2 instance depends on the its performance and is billed by the hour. The price is made up of the amount of resources and data traffic and the amount of time the Elastic IPs and EBS were used. Under [Amab], potential users may calculate the costs of the desired package beforehand.

## 8.10 Clustering with CouchDB Lounge

This section describes how to distribute the system over several CouchDB instances without changing anything to the user experience. This is where CouchDB Lounge [Lee10] comes in. CouchDB Lounge is a proxy-based clustering and partitioning application [Lee09b]. Both concepts may help to increase a system's availability and performance.

Already in 1997, clustering was suggested as a solution to cope with the increasing needs of modern database applications [Meh97]. A computer cluster generally means a number of ...

> ... similar workstations or PCs that are interconnected with a local broadband connection. [Tan07, P. 34]

In this very case, clustering means adding redundant CouchDB servers to allow load-balancing and increase availability. Requests are routed to several parallel CouchDB instances. CouchDB also allows redundant storage of data, ensuring that multiple copies of the data are available if the hardware should fail, but this will not be discussed in detail in this thesis.

Horizontal partitioning means dividing the disk space in partitions called *shards*. The shards are distributed over multiple servers in order to increase the throughput to prevent limited hard disk performance from becoming a bottleneck. The next section will discuss this in detail.

### 8.10.1 Functionality

CouchDB has not been around for long; neither its functionality nor its use have been thoroughly documented. Hence, the following illustration is mainly inspired by [And10, Chap. 19] and [Lee09a].

Lounge consists of two core modules. A *Smartproxy* deals with CouchDB views and distributes them over the Lounge cluster's nodes. The view performance can be enhanced by increasing the number of nodes in the cluster. The Smartproxy is implemented as a daemon for *Twisted*, a „framework for writing asynchronous, event-driven networked programs in Python“ [Lef02]. A *Dumbproxy* is a module for *nginx*, a web server and reverse proxy server. The Dumbproxy is used to handle GET and PUT requests that are not intended for CouchDB views. These requests, too, are distributed over the cluster's individual nodes. The clients are still kept under the impression that their requests are handled by a single CouchDB installation. Both modules use distinct hashes, built by CouchDB Lounge using the documents' IDs. The first characters of these hexadecimal hashes determine the shard to which this document is assigned. The exact assignment is configured in a *shard map*.

Thus, CouchDB Lounge allows creating a cluster that is accessible from the outside under a single address. In figure 8.3, the cluster is depicted as a circle. Two shards are assigned to each of the eight nodes, and each of these shards is identified by a hexadecimal number. If a CouchDB document's ID hash starts with this number, Lounge will save this document in the associated shard. The process then redirects HTTP requests to the actual location of the document. This is how partitioning is implemented.

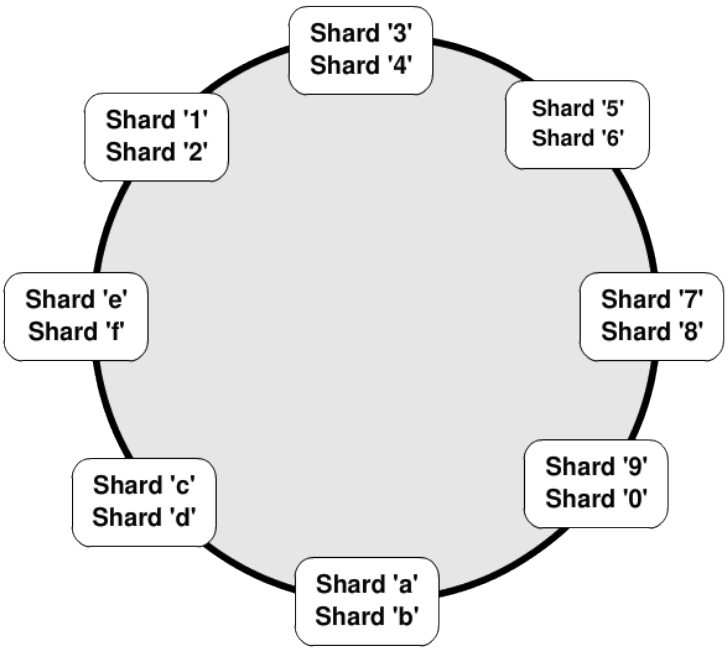


Abbildung 8.3: 16 shards, 8 nodes. From [Leh09]

> From the outside world, a CouchDB lounge cluster looks just like any other CouchDB node. [...] There's no difference from a functional perspective. [...] Its sharded nature is completely transparent. [VG10]

If the demands for the system's capacity are expected to increase during the system's life cycle, [And10] and [Kla10] recommend to aim for as many shards as possible in the beginning. The process of distributing the data over more shards than there are nodes is called „oversharding". Even if just a few nodes are enough to start with, this amount can later be increased by adding more nodes. The shards are then distributed among these nodes. This is illustrated in 8.3. The system in figure 8.3 kept its interface, but the CouchDB instances in the nodes have been replaced by Lounge configurations. If further shards are added afterwards, the entire cluster must be rebuilt.

medskip

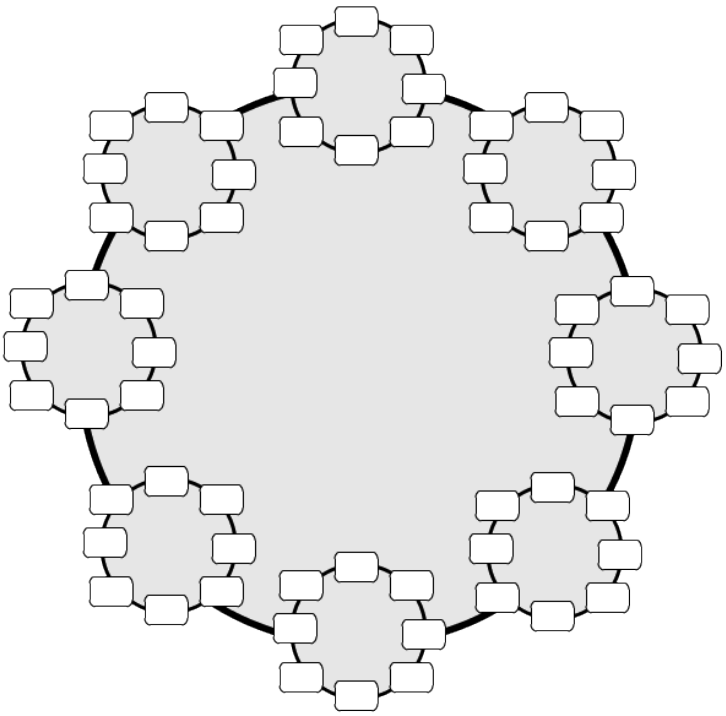

Abbildung 8.4: 16 shards, 8 nodes with 16 subshards, 8 subnodes each. From [Leh09]

Even for smaller projects, oversharding benefits operation speeds as smaller numbers of documents keep the index size low.

### 8.10.2 Configuration

[Lee10] contains CouchDB Lounge's source code. It contains instructions for the installation of Dumbproxy, Smartproxy and *Python Lounge*, a collection of required modules. The current version of CouchDB Lounge depends on CouchDB version 0.10.0. The patch that unlocks the required „design-only replication" exists only for this version. Future versions of CouchDB will come with this feature pre-installed.

[cou10a] describes how several CouchDB instances may be installed on a single computer. The central CouchDB configuration file in `/etc/couchdb/local.ini` must be copied as many times as there are instances. Listing 8.9 contains the most important parts from its copy in `local-1.ini`. The copies in `local-2.ini` etc. contain the corresponding port and log file.

```
1 [httpd]
2 port = 5984
3 bind_address = 127.0.0.1
4
```

```
[log]
file = /var/log/couchdb/couch-1.log
```

Listing 8.9: Extract from the CouchDB configuration file

Listing 8.10 shows how to start the first CouchDB instance on a Unix system (Mac OS X 10.6):

```
sudo -i -u couchdb '/usr/local/bin/couchdb -a etc/couchdb/local-1.ini -p /usr/local/
    var/run/couchdb/couchdb-1.pid -o /usr/local/var/log/couchdb/error-1.log -e /usr/
    local/var/log/couchdb/error-1.log -b'
```

Listing 8.10: Starting a CouchDB instance

If the desired number of CouchDB nodes is set up and started, the set-ups are checked for integrity by running the CouchDB test suite as described in the installation manual. If the tests pass successfully, CouchDB Lounge has to be configured before it can be used. This is done by modifying the `/var/lounge/etc/shards.conf` file, indicating the number of shards and the level of redundancy. The file contains the JSON object `nodes` which contains information about the number of CouchDB nodes. Every entry in the array contains the host name and port of an individual node. `shard_map` is an array consisting of arrays which defines the location of individual shards and where they should be replicated to. Any number of nodes and any level of redundancy may be specified.

Listing 8.11 describes two shards on two nodes. The first shard (number 0) is located on node 0, the second (number 1) on node 1. The first one is replicated to node 1, in case node 0 should fail, and vice-versa.

```
{
  "shard_map": [[0,1], [1,0]],
  "nodes": [ ["localhost", 5984], ["localhost", 5985] ]
}
```

Listing 8.11: shards.conf with two nodes and simple redundancy

Listing 8.12 defines a cluster with eight shards, distributed over four nodes with no redundancy. The nodes in both examples are located on the same computer, but they may be distributed over several systems by indicating another host name.

```
{
  "shard_map": [[0], [1], [2], [3], [0], [1], [2], [3]],
  "nodes": [ ["localhost", 5984], ["localhost", 5985], ["localhost", 5986], ["
      localhost", 5987]] ]
}
```

Listing 8.12: shards.conf with four nodes and no redundancy and simple oversharding

The installation and configuration of CouchDB has been successful if a document that was created on one node is automatically copied to all the nodes for which redundancy is defined in `shard_map`.

# 9 Application

This chapter will explain how to use the application which was developed in the context of this thesis. It has the working title *Doingnotes*. Installation and operation of all functionality are documented within.

## 9.1 Installation

The next two sections will describe the installation of CouchDB and the deployment of the application.

### 9.1.1 CouchDB

First, CouchDB has to be installed on all computers that will use Doingnotes. CouchDB is extensively discussed in section 4.2. Since earlier versions do not support continuous replication which is necessary for Doingnotes, CouchDB is needed in version 0.11.0 or higher.

CouchDB will run on most popular desktop operating systems and some established mobile platforms. Currently, the latter include Google Android ([Mil10]) and Nokia MeeGo (formerly Maemo) ([Ape09]), [Ape10]). Recently, Palm announced that the next version of its mobile Operating System WebOS would also come with a CouchDB installation ([Pro10]).

For some desktop operating systems, pre-compiled CouchDB binaries exist. All dependencies are contained within. If CouchDB is installed from the source code, these dependencies have to be resolved manually. Amongst other things, CouchDB requires Erlang [Eri10], OpenSSL [Ope09] and Spidermonkey [Moz10c] to be installed. More details can be found in section 6.2.1.

The quickest way to get CouchDB up and running on Mac OS X is by downloading *CouchDBX*: „The one-click CouchDB package for the Mac“ [Leh10d]. A binary installer is also available for Windows systems [Cou10b]. Some Linux distributions have already included CouchDB in their software repositories; e.g. in newer Ubuntu versions, CouchDB can be installed using the packaging tool *apt*.

Doingnotes will work fine using the current versions of the three aforementioned binaries, and they are by all means recommended for end users. Developers, however, may want to install CouchDB directly from the source code. In doing so, they may retrieve the latest version from the

Subversion [Apa10d] or Git [Apa10c] repositories. More detailed instructions on how to install CouchDB on different operating systems and distributions can be found in the CouchDB Wiki [Cou10c]. How CouchDB is started depends on the operating system. Again, more details can be found in the Wiki.

### 9.1.2 Deployment

The next step involves installing CouchApp [Che10]. CouchApp was introduced in section 5.1.1. CouchApp allows the application to be easily deployed into the CouchDB instance. CouchApp assumes that a recent version of Python is installed [Pyt10]: it requires Python's `easy_install` module [Fel09] in order to download and install CouchApp, also a Python package.

It is necessary to alter `.couchapprc` in the project directory to deploy the design documents (i.e. the code) into the CouchDB instance; the entry `default` must point to the running CouchDB instance (see listing 5.1). Afterwards, `couchapp push` has to be called from here in order to deploy the application into the database and make it executable.

The application can be accessed using a browser that meets the discussed in section 8.5.2. Firefox [Moz10d] 3.5 or higher is recommended.

## 9.2 Operation

### 9.2.1 Basic functionality

The application allows lines to be sorted and saved inside the outline. The page `http://localhost:5984/doingnotes/_design/doingnotes/index.html` is the main page. As all other pages, it contains a navigation bar and a content area. The main page can be reached from anywhere in the application by clicking on the „Outlines“ link. The content area lists existing outlines (see fig. 9.1). Next to the title of every outline is the time and date of creation. If there are no outlines available, the overview displays the text „You have no outlines yet“. On the right-hand side, there are some hints on how to create an outline.

In order to create a new outline, the user has to click „New Outline“. Titles must be at least three characters long and may only contain alphanumeric characters, blanks and hyphens. The title does not have to be unique as outlines are referenced by their ID. Erroneous input as well as clues about conflicts and replication status are displayed near the application's upper border and fade out after a few seconds. The signal colours of the notification indicate errors (red) or notices (green).

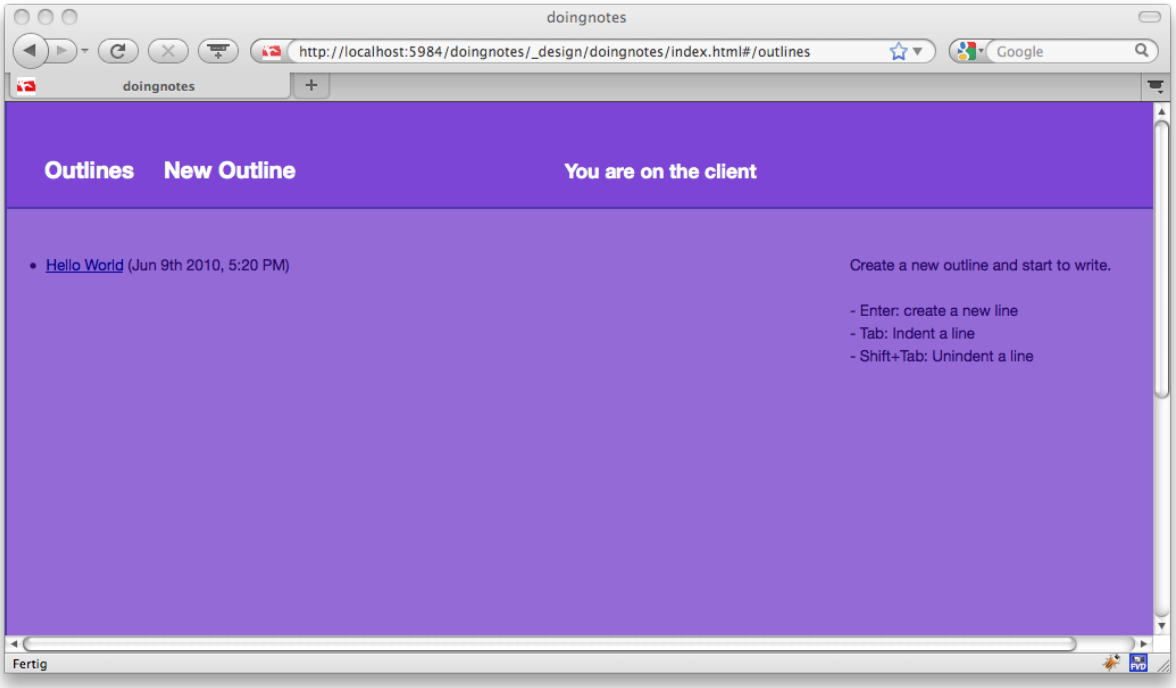


Abbildung 9.1: Screenshot: list of outlines

When an outline has been created (see fig. 9.2 it is immediately opened. Text can be entered into the first line. If the text's length exceeds that of the line, the line will automatically grow along. If **Enter** is hit, a new line will be created. An animated throbber in the right-upper corner indicates that the line is still being created.

Navigating between the lines can be done using the **arrow keys**. The **Tab** key indents a line, creating hierarchy between the entries. A line can be indented as long as there is a line with the next higher indentation level directly above it. **Shift** + **Tab** outdents a line. This keystroke will not work for lines on the first level. When manually resolving conflicts (see section 9.2.3), the **Tab** or **Shift** + **Tab** keys allow jumping back and forth between conflict fields.

„Change title or delete this outline" points to the edit outline page (see fig. 9.3). Here, the outline's title may be changed. Clicking the „Delete this outline" button will promptly delete the outline.

Now that the user has familiarised herself with the application as a single-user system, the following sections will introduce the application's multi-user features.

### 9.2.2 Replication

In order to use replication and conflict resolution, Doingnotes must run in more than one CouchDB instance; i.e. Doingnotes has to be deployed onto a server to be able to use all its

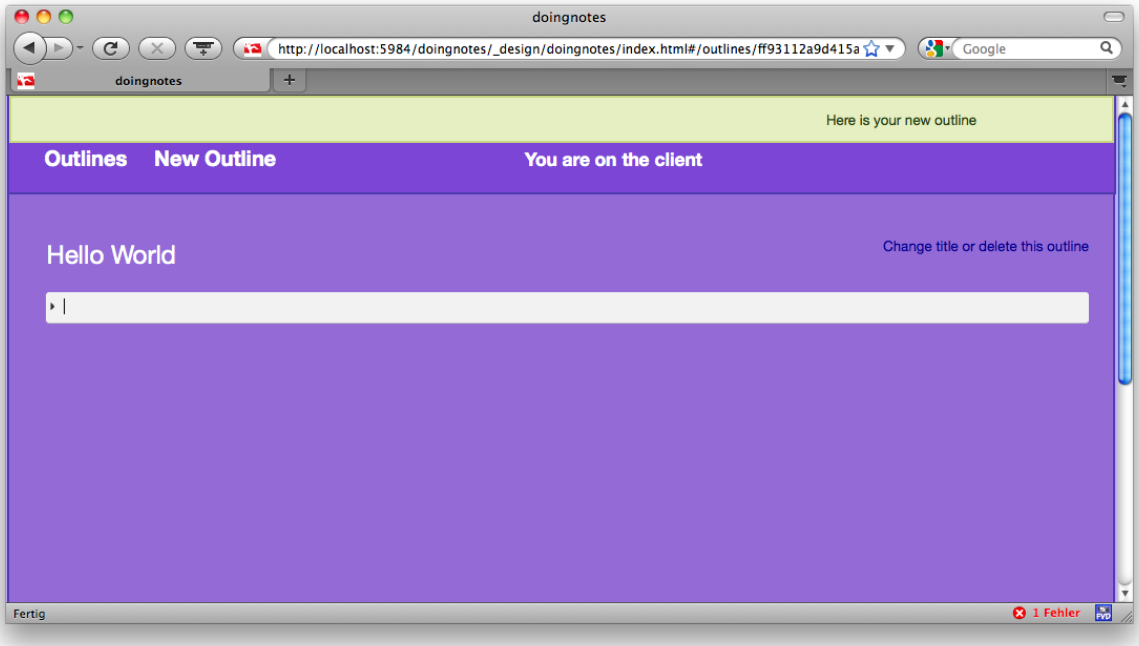


Abbildung 9.2: Screenshot: newly created outline

features. The CouchDB instance on the server will hereinafter be called the „server“; the instance on the user's computer is the „client“.

Server and client can also run on the same computer. In order to test replication, the same outline has to be opened in more than one browser window. In doing so, one can monitor how updates are automatically replicated to the server (or to other clients) and how the conflict resolution mechanism kicks in. Whether the server is on the same computer or not, its URL has to be correctly entered into the configuration file `/_attachments/app/config/config.js`.

For operation on a single computer, two CouchDB instances have to be installed, as described in section 8.10.2 (see also [cou10a]). In this set-up, the server URL in `/_attachments/app/config/config.js` must point to `http://localhost:5984`.

The CouchDB instances on ports 5984 and 5985 (client and server) are opened in two browser windows. For extra clarity, the navigation bar indicates which browser window the user is currently working in: it will either read „You are on the client“ or „You are on the server“. The same outline is now opened in each of the two windows. If something is changed in the server window, the user in the client window is notified of this: „Replication has brought updates.“ (see fig. 9.4). Clicking „View them.“ reloads the page, displaying the update (see fig. 9.5).

If the connection drops or if the user intentionally disconnects in the meantime, the replication can be restarted by simply reloading the page.

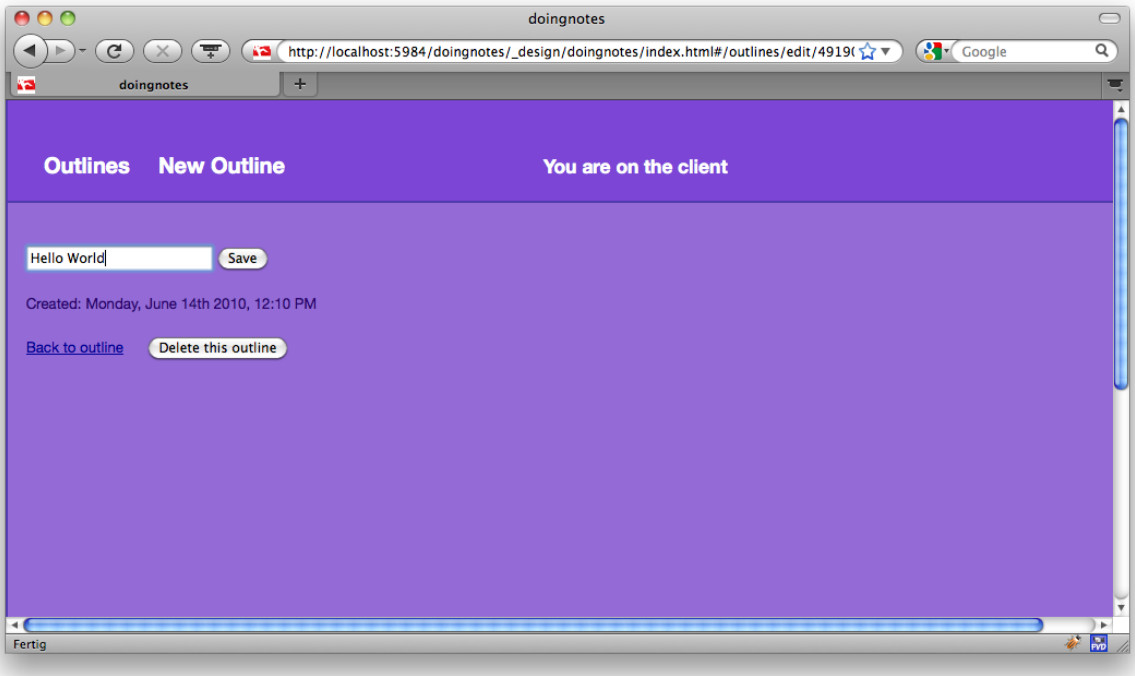


Abbildung 9.3: Screenshot: editing the outline title

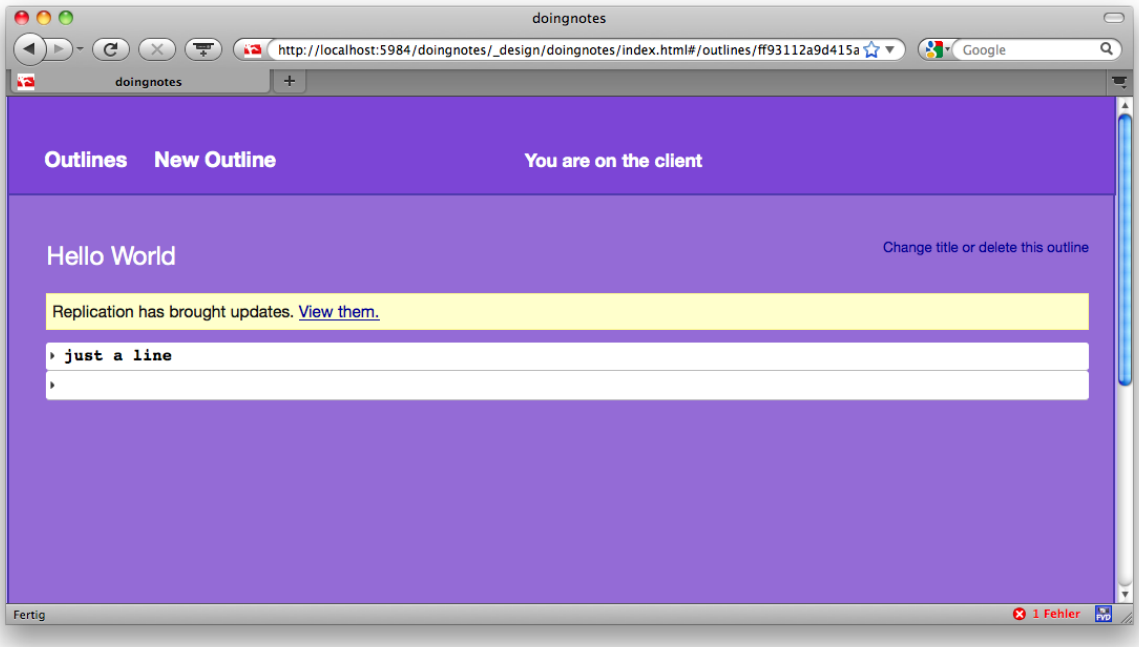


Abbildung 9.4: Screenshot: an outline with notification of replication

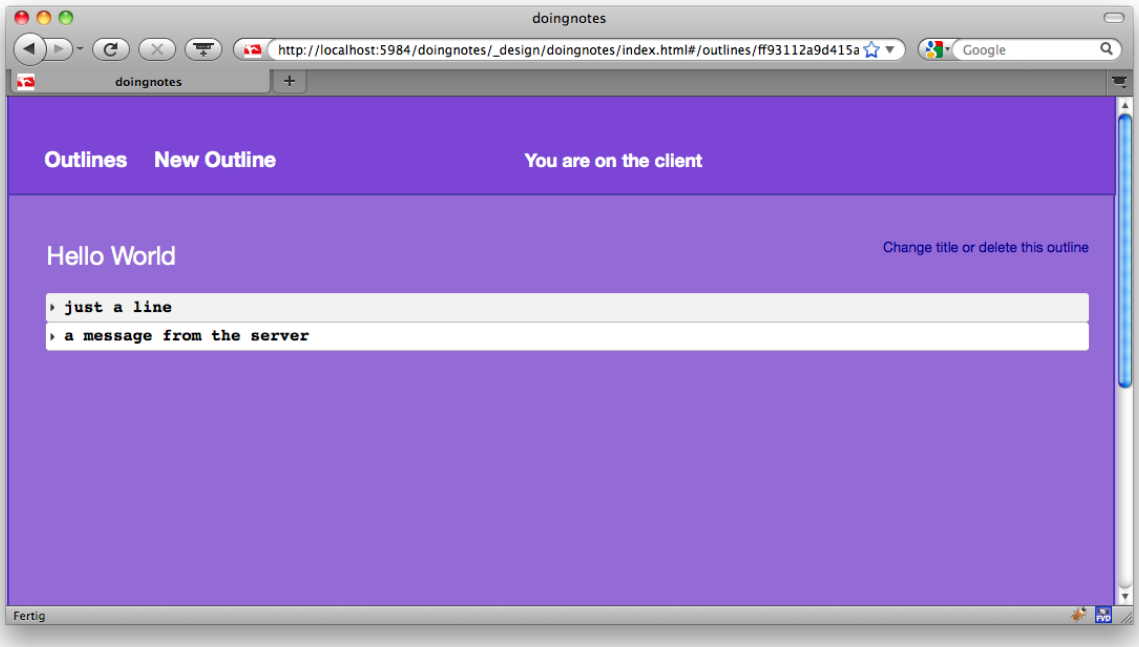


Abbildung 9.5: Screenshot: the outline after updating

### 9.2.3 Conflict handling

If replication caused no conflicts, its results are simply shown. There are two types of conflicts the application was programmed to deal with: *append conflicts* and *write conflicts* (cf. sections 7.4.1 and 7.4.2). An append conflict occurs when more than one user simultaneously add a new line in the same spot. A write conflict occurs if the contents of a line are modified by two or more users simultaneously.

#### 9.2.3.1 Creating conflicts for testing purposes

A write conflict can be provoked by following these steps one after another (client and server may be swapped around):

- stop the CouchDB server instance
- enter text in the client window
- stop the client instance
- start the server instance
- enter text in the server window
- start the client instance

To ensure that conflicts occur, the instances have to be shut down or started completely before proceeding to the next step. Only this will interrupt the automatic replication long enough for a conflict to occur.

In order to provoke an append conflict, a new line has to be added after the same line in both windows, instead of entering text. The system can also deal with both conflict types occurring in the same line.

#### 9.2.3.2 Handling

##### Append conflicts

If an update to the server causes an append conflict, it is automatically resolved by the server. A notification „Replication has automatically solved updates“ is shown and in addition, the affected lines are highlighted in green (see fig. 9.6).

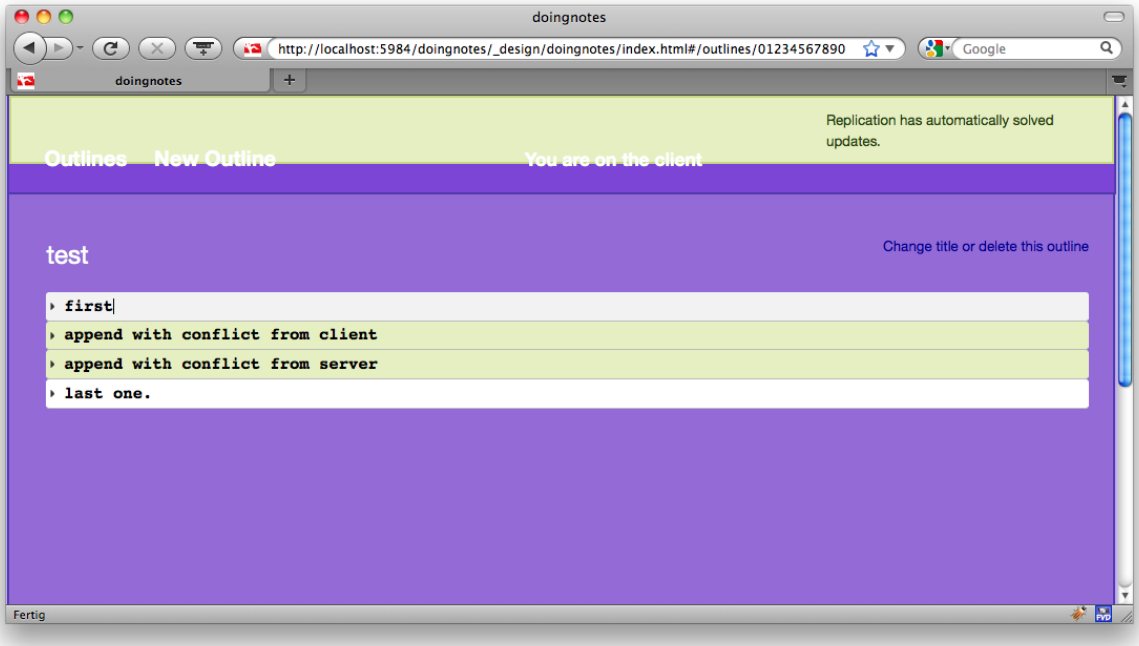


Abbildung 9.6: Screenshot: automatically resolved append conflict

The conflict is resolved by putting the line that was added first before the one that was added last. However, the order is not reliable since the clocks on two systems are not necessarily synchronised. If conflict resolution happens simultaneously on two computers (clients), each computer sorts the lines in the same way, preventing new conflicts from being produced by the conflict resolution process. More details about this process can be found in section 8.7.1.

### Write conflicts

If an update causes a write conflict to occur, the user has to decide which version shall remain in the system and which one should be rejected. The user may also manually merge the versions. A notification „Replication has caused one or more conflicts.“ is displayed. Additionally, the affected lines are coloured red (see fig. 9.7).

The write conflict must be resolved manually. The user is therefore presented with a mask which displays both versions for every conflicting line (see fig. 9.8). She may now decide which one to keep and even freely edit the line before saving. Distinct captions on the save buttons („Keep overwritten version“ and „Keep winning version“) indicate which version was elected winner by CouchDB's internal conflict resolution mechanism. More details can be found in section 8.7.2.

The conflicts are thus resolved line by line. After saving a version of a line it is conflict-free. The conflict resolution mask is hidden for every line and the winning line is shown instead. If all

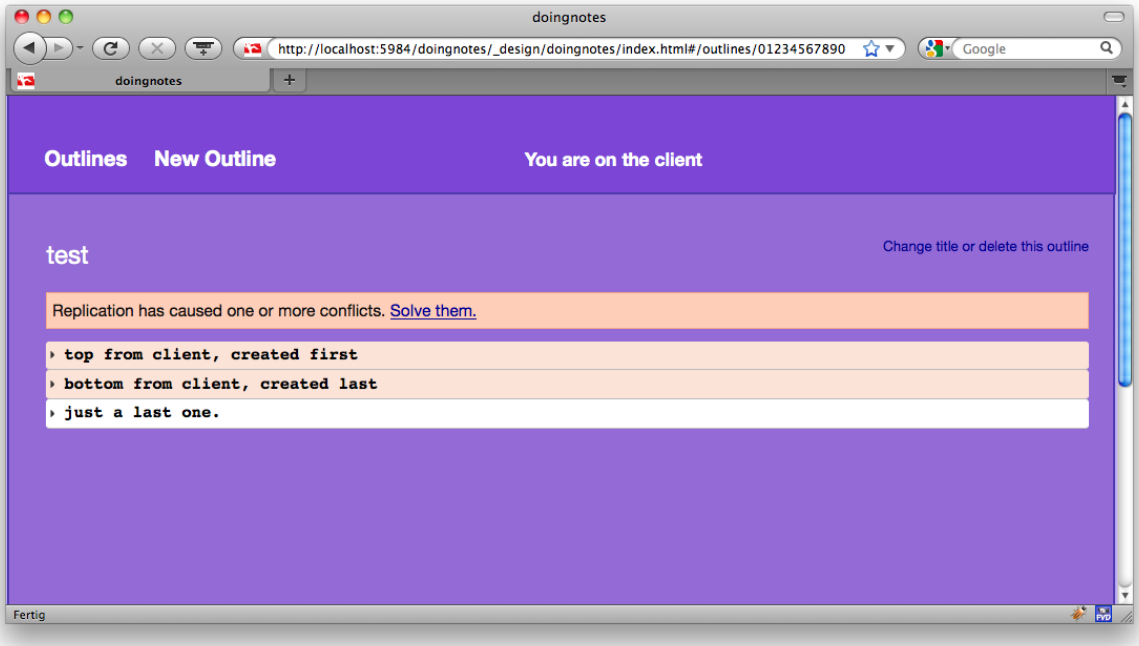


Abbildung 9.7: Screenshot: unresolved write conflict notification

conflicts have been resolved the user is notified about this, as shown in figure 9.9.

### 9.2.4 Assistance in provoking conflicts

Some macros were created to make generating conflicts easier. These put the database in a well defined state. It is not possible to manually provoke conflicts in CouchDB. Therefore, the macros simply automate the steps described in the previous section. The macros have been implemented as *Rake tasks*. Rake is a build tool written in Ruby. It is similar to the make program: commandos are executed under certain conditions and in a certain order [Wei08]. Users can create their own command batches in Ruby syntax, the so-called Rake tasks.

Prior to using Rake, Ruby [Mat10] must be installed. The Rakefile must be modified to contain the correct URLs and ports of the CouchDB instances. Client and Server are pre-set to run on localhost on ports 5984 and 5985.

The `Rakefile` file in the project's root directory contains the following Rake tasks:

**couch:recreate_host** *Resets* (Deletes and recreates) the database; The application is deployed into the CouchDB client instance

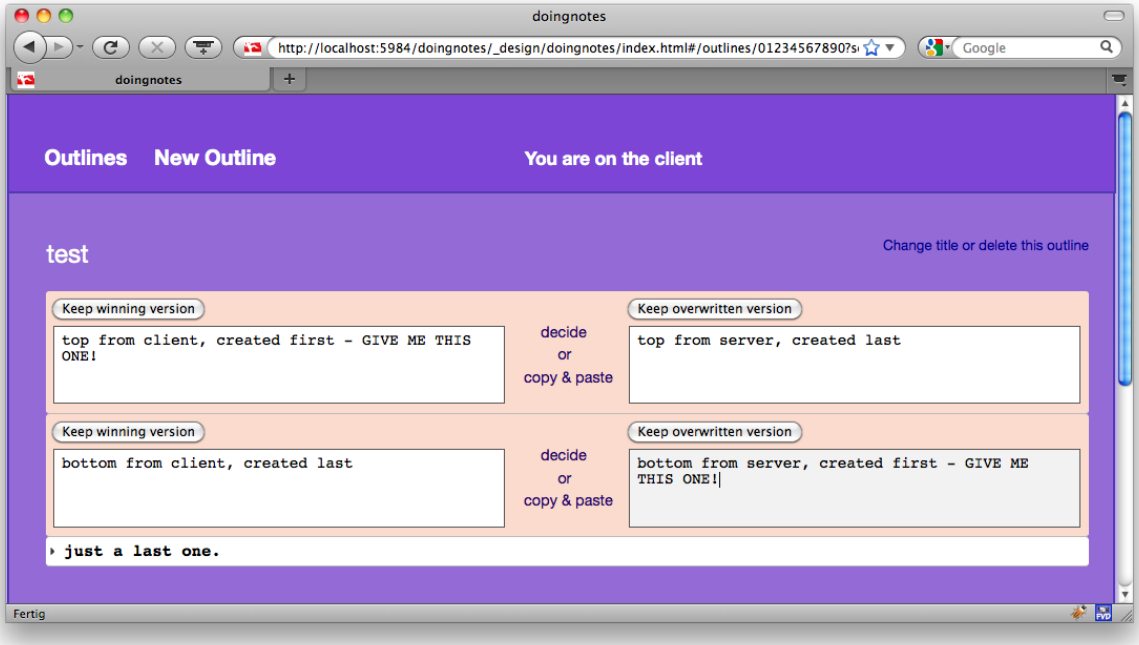


Abbildung 9.8: Screenshot: manual write conflict resolution

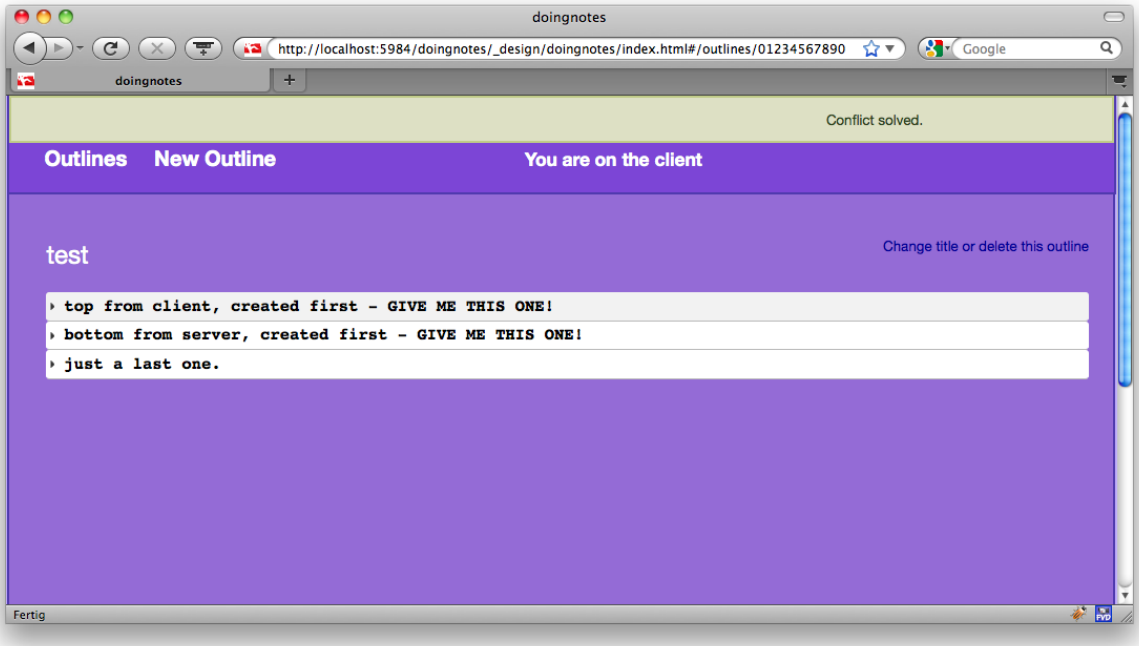


Abbildung 9.9: Screenshot: write conflict resolved

**couch:recreate_server** Resets the database; the application is deployed into the CouchDB server instance

**couch:wait** Waits two seconds before proceeding to the next step

**couch:start_host** Starts the CouchDB client instance

**couch:start_server** Starts the CouchDB server instance

**couch:stop_host** Stops the client instance

**couch:stop_server** Stops the server instance

**couch:writeconflict** Resets the database; generates an outline with a write conflict

**couch:twowriteconflicts** Resets the database; generates an outline with two write-conflicts

**couch:appendconflict** Resets the database; generates an outline with an append conflict

**couch:appendandwriteconflict** Resets the database; generates an outline with an append and a write conflict

For the tasks referring to client or server, there are also steps that execute the tasks on both instances simultaneously.

# 10 Evaluation and future prospects

In order to evaluate the result, this chapter will compare the system that was developed with the requirements set in chapter 6. The result will be evaluated with the aid of these requirements and any problems left unsolved will be discussed. Next, some future prospects for the technologies that were used will be examined. Finally, some recommendations for further development and research will be uttered.

## 10.1 Evaluation of the result

All seventeen must-haves (see section 6.1.1) have been completely implemented. Of the sixteen may-haves however (see section 6.1.2), only four could be realised. For want of time, the moving, deleting, commenting and versioning of lines in the outliner have been omitted, as well as selective replication of a single outline and the explicit enabling and disabling of replication.

In order to create a truly usable product, first of all it would be necessary to further improve conflict handling. As described in section 7.4.3, conflicts caused by simultaneous and diverging indenting and outdenting of lines will lead to errors in the outline's layout. If lines are simultaneously changed in three or more replicas this will also lead to conflicts that are not correctly dealt with. This is chiefly an interaction problem: the user interface for manual conflict resolution, as described in section 8.7.2, was built to show only two versions of a line at a time. Though unlikely to occur in everyday use, manually merging a higher number of line versions will necessitate another user interface approach.

Non-functional requirements were discussed to the greatest possible extent. The source code may be improved in simplicity and redundancy: in order to make the program truly compliant to the MVC architecture, the functions that control replication should be rewritten in object-oriented programming style. This was already done for conflict recognition, presentation and handling. The application also meets the requirements of the user interface (cf. section 6.2.3). However, due to the necessary connection establishment between front-end and database, the user interface is sometimes a little slow in its reaction to user input, interrupting the user's flow of work for a few seconds. It should be verified if this can be improved by optimising the user interface.

Notwithstanding its shortcomings, the application as described in the project definition was by all means successfully developed and realised. This thesis' most important accomplishment is that the application puts a certain paradigm of Internet use into practice: peer-to-peer communication. Compared to conventional client-server applications, the architecture conceived here allows

users to better control their data. On-line collaboration was implemented without depending on uninterrupted Internet access and ever-available servers.

## 10.2 The future of the employed technologies

The implementation was slower than imagined. In order to aptly employ the HTTP API, replication, conflict handling and monitoring the database for changes a lot of background work had to be done first. Comparatively little time was left for the actual writing of business logic. Yet, the technologies used were continuously improved and new libraries and frameworks were developed that will reduce the amount of work for similar projects in the future.

For instance, CouchDB 1.0 contains some features that may definitely make it easier to improve the application [Leh10a]. Its release is due when this thesis is finished [Sla10]. CouchDB version 1.0 will among other things allow individual document replication using a document's ID, eliminating the need to replicate the database as a whole. This makes it easier to perform selective replication of outlines. Furthermore, the support for Windows operating systems has been improved, further increasing its platform-independency. Future CouchDB releases will also natively support sharding [Lehnardt, Jan, personal conversation, 9 July 2010]. This will eliminate the need for CouchDB Lounge in the future.

Among recent developments, the framework *Evently* deserves special notice [And10]. Like Sammy, Evently allows application routing, but it was specially designed with event-based CouchDB applications in mind. Evently creates a connection between CouchDB views, the changes feed, HTML templates and any defined JavaScript callbacks, and gives a structure for the organisation of the source code. Compared to the means used in this thesis, Evently will certainly increase productivity.

## 10.3 Suggestions for further development

The outliner can be developed further without major hindrances. The use of the tree structure will make it easy to implement deleting and moving of lines by no longer displaying the line or fitting it in somewhere else in the tree. Columns in the outliner can also effortlessly be implemented by assigning several text areas to the lines.

Another field of work would be the implementation of access control and user administration. Individual outlines could be marked as private or public by users and therefore available for replication or not. Applications that use distributed data on mobile devices require higher security measures:

> Providing high availability and the ability to share data despite the weak connectivity of mobile computing raises the problem of trusting replicated data servers that may be corrupt. This is because servers must be run on portable computers, and these machines are less secure and thus less trustworthy than those traditionally used to run servers. [...] Portable machines are often left unattended in unsecured or poorly secured places, allowing attackers with physical access to modify the data and programs on such computers. [Spr97, Chap. 1]

Accordingly, if this application is to be developed for productive use, implementation of access control should be highly prioritised. Yet, the author of this thesis believes that the highest priority should be to extend the application's peer-to-peer capability. Application instances could propagate approved outlines through a web service. Protocols such as *Bonjour* make it possible to detect network services in local IP networks [App10]. Such a protocol could be used so that application instances might recognise each other inside a network and offer the possibility to replicate outlines directly with one another. In doing so, documents could be co-edited simultaneously even without an Internet connection, for instance inside an office or on conferences.

The implementation of the tasks was certainly successful. However, a considerable amount of development effort is still needed in order to make the application truly suitable for the uses listed in section 3.1.2. If several users should massively and simultaneously edit an outline -synchronising only after many changes have been made- the number and complexity of conflicts cannot yet be dealt with in a satisfying and stable manner. It is certainly possible to implement a distributed system using CouchDB and a selection of other technologies, but data merging remains the application developer's responsibility. The CouchDB development team however plans to provide built-in solutions to standard conflict resolution scenarios in the future [Lehnardt, Jan, personal conversation, 9 July 2010]. That said, merging is exceptionally difficult in the use case at hand, since the documents (i.e. the lines in an outline) are very granularly chosen and strongly linked together. For application areas where documents are not often edited simultaneously and therefore cause less conflicts, the solution may be much less complex. Conceivable cases are address books, calendars, customer data, or even services that exchange messages. [And10, Chap. 10] and [Leh10a] contain further suggestions.

# Appendix

## A.1 Abbreviations

This section contains all abbreviations that are not assumed to be commonly known, or that only occur along with their description in the text.

| Abbreviation | Description |
|---|---|
| **AJAX** | Asynchronous JavaScript And XML |
| **AMI** | Amazon Machine Image |
| **API** | Application Programming Interface |
| **AWS** | Amazon Web Services |
| **BDD** | Behaviour Driven Development |
| **CAP** | Concurrency / Availability / Partition Tolerance |
| **CGI** | Common Gateway Interface |
| **CSS** | Cascading Style Sheets |
| **DOM** | Document Object Model |
| **EBS** | Elastic Block Storage |
| **EC2** | JavaScript Object Notation |
| **IaaS** | Infrastructure as a Service |
| **JSON** | JavaScript Object Notation |
| **MVC** | Model View Controller |
| **MVCC** | Multi Version Concurrency Control |
| **PaaS** | Platform as a Service |
| **RDBMS** | Relational Database Management System |
| **REST** | Representational State Tansfer |
| **S3** | Simple Storage Service |
| **SaaS** | Software as a Service |
| **TDD** | Test Driven Development |
| **UUID** | Universally Unique Identifier |

Abbildung A.1: List of abbreviations

# A.2 Addendum to the analysis

## A.2.1 OmniOutliner

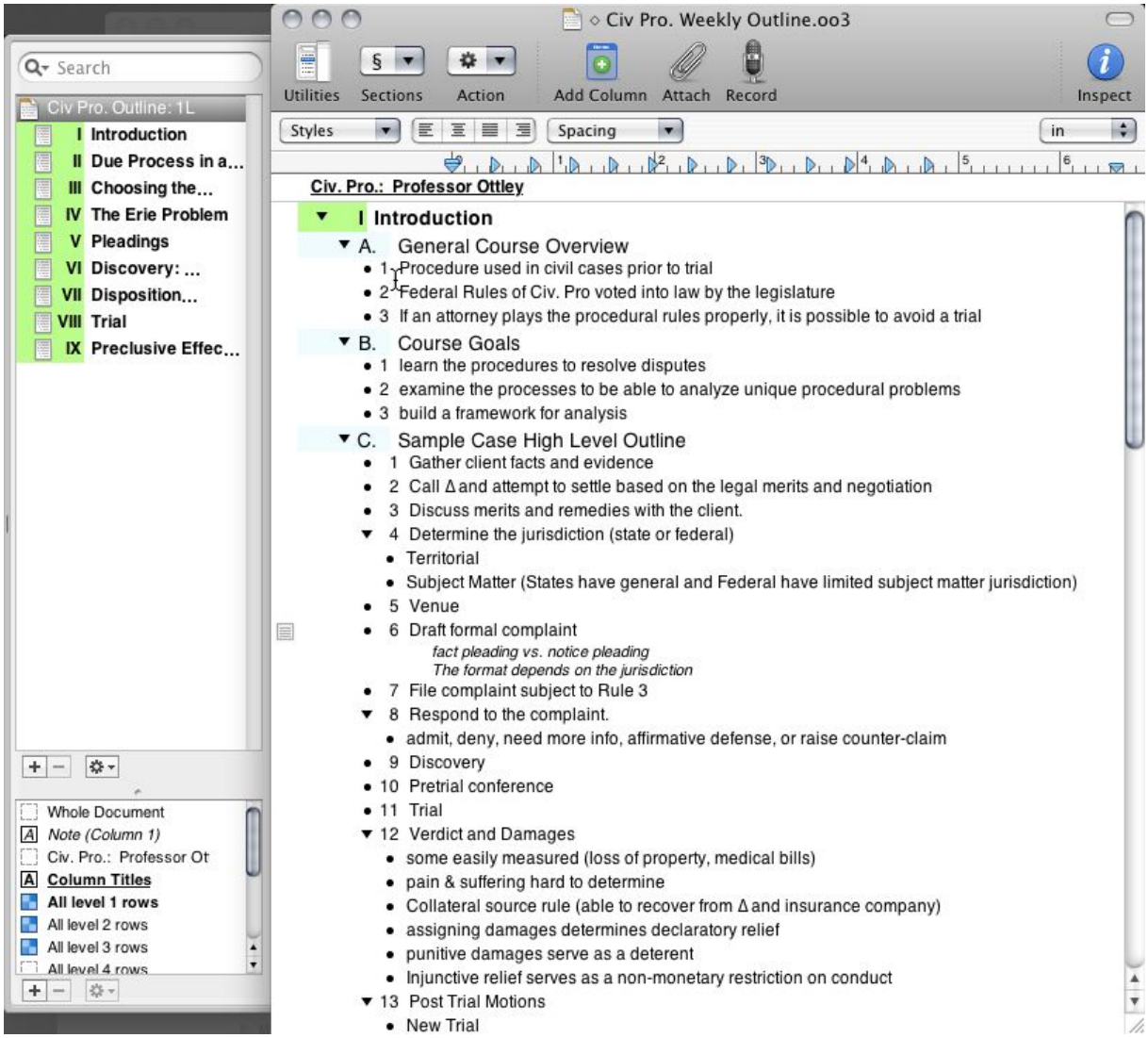


Abbildung A.2: Screenshot of OmniOutliner [Law10]

## A.2.2 Gartner's hype cycle

The hype cycle highlights the phases of public attention that new technologies go through after their introduction. The X-axis symbolises the time after introduction, the Y-axis the intensity of attention for that technology.

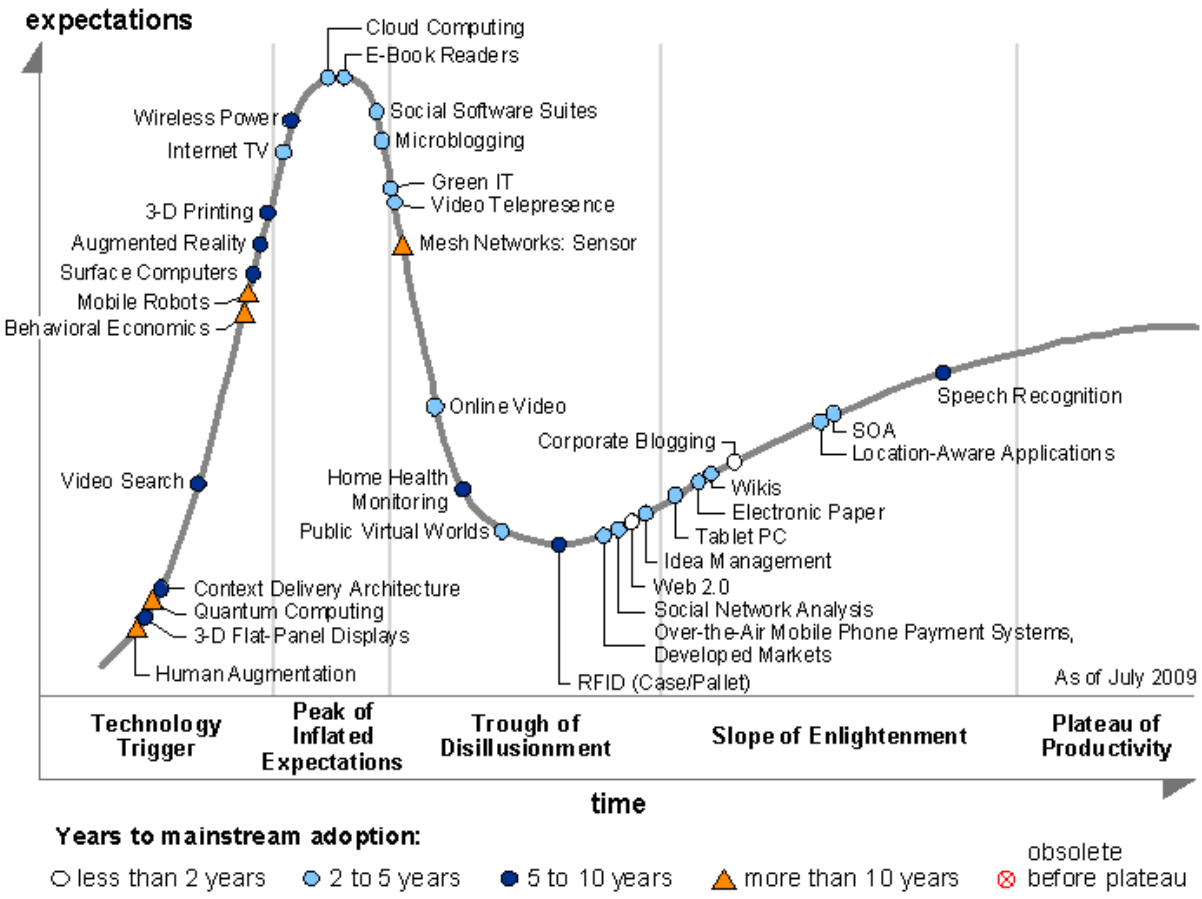


Abbildung A.3: Gartner's 2009 hype cycle, [Gar09]

## A.3 Addendum to the technical background

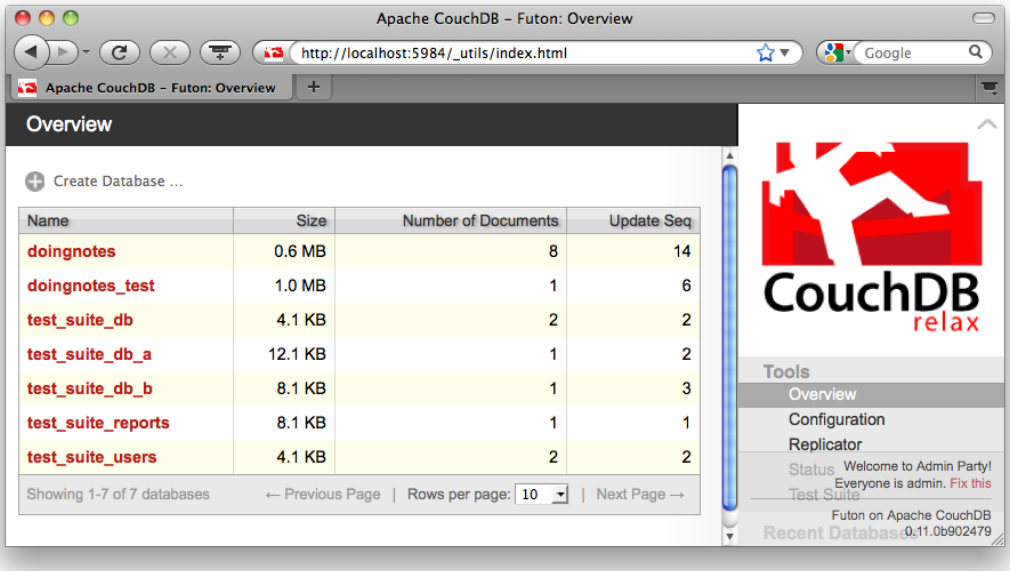


Abbildung A.4: CouchDB Futon: overview

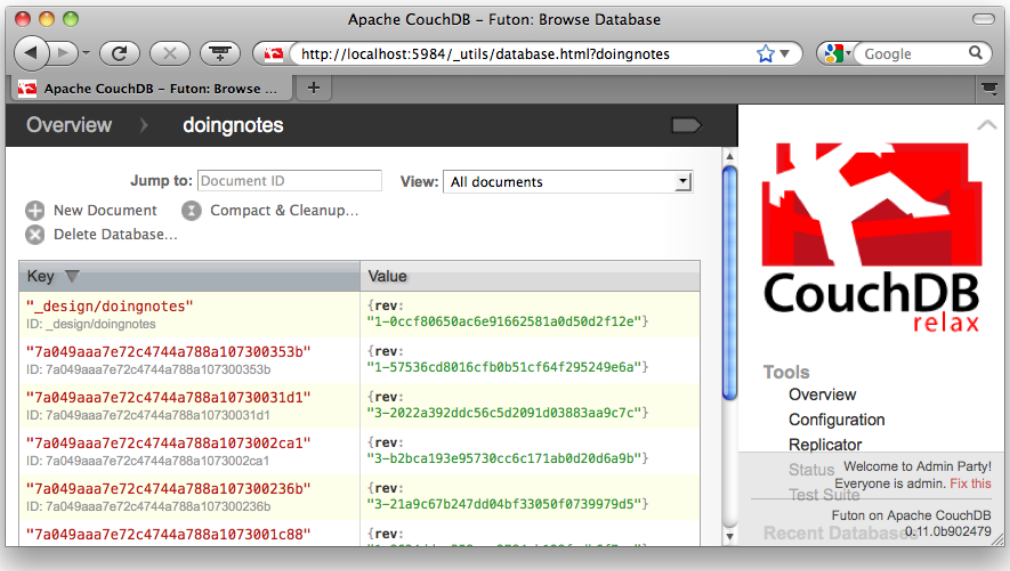


Abbildung A.5: CouchDB Futon: browse database

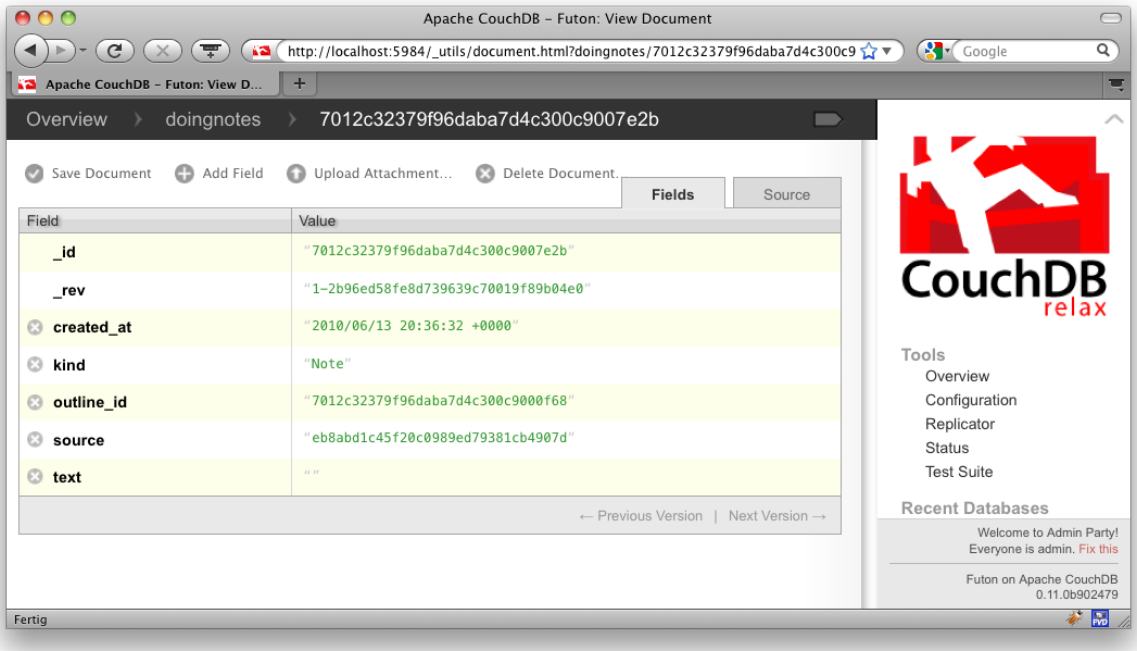


Abbildung A.6: CouchDB Futon: document

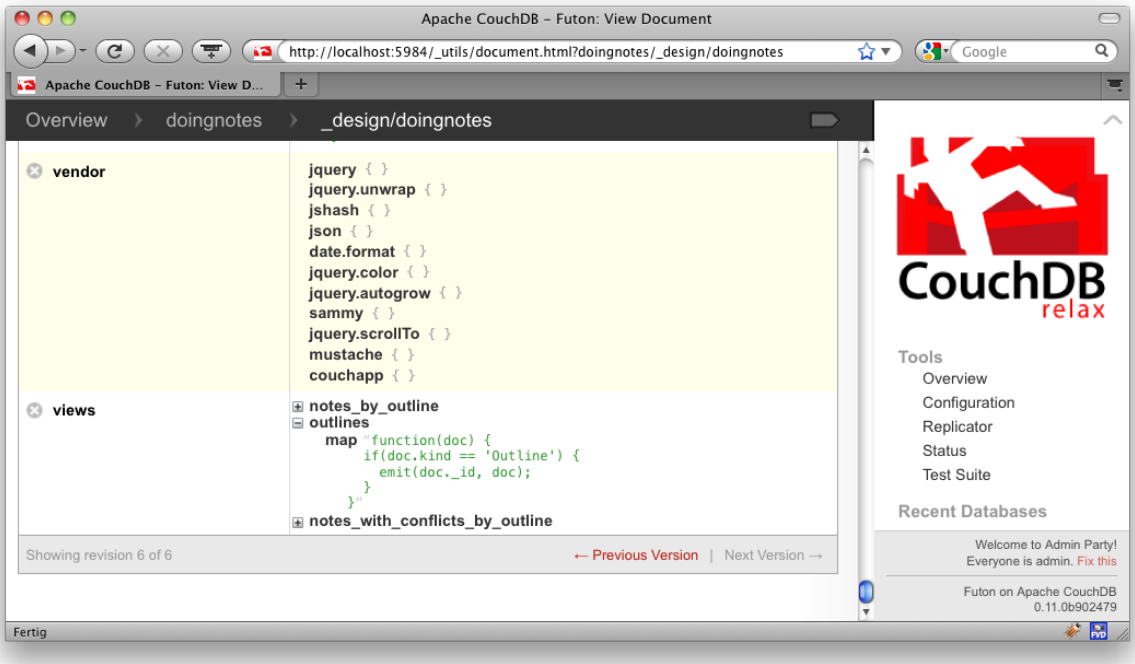


Abbildung A.7: CouchDB Futon: design document

# A.4 Addendum to the requirements of the system

## A.4.1 Functional requirements

Table A.8 contains the functional requirements that were listed in text section 6.1. The table is sorted by priority in descending order. The priority is indicated by the numbers *1* (highest priority), *2* (medium priority) and *3* (lowest priority).

| Domain | No. | Description | Priority |
|---|---|---|---|
| **Outlines** | **FA100** | Outline creation | 1 |
| | **FA101** | Edit title | 1 |
| | **FA102** | Delete | 2 |
| **Lines** | **FA200** | Adding lines | 1 |
| | **FA201** | Navigation using function keys | 1 |
| | **FA202** | Edit contents | 1 |
| | **FA203** | Save automatically when focus is lost | 1 |
| | **FA204** | Indentation and outdentation | 1 |
| | **FA205** | Indenting and outdenting entire blocks | 1 |
| | **FA206** | Warning about data loss on window close | 2 |
| | **FA207** | Moving up/down | 2 |
| | **FA208** | Adapt text area size to the length of the text | 2 |
| | **FA209** | Collapse and expand lines | 2 |
| | **FA210** | Locally save line visibility state | 2 |
| | **FA211** | Save revisions | 3 |
| | **FA212** | Exchange revisions | 3 |
| | **FA213** | Comment | 3 |
| | **FA214** | Delete | 3 |
| **Replication** | **FA300** | Replicating outlines to the server | 1 |
| | **FA301** | Replicate outline changes to the server | 1 |
| | **FA302** | Receive outlines made by others | 1 |
| | **FA303** | Receive changes to outlines made by others | 1 |
| | **FA304** | Notification when others made changes | 1 |
| | **FA305** | Resume replication or notify when Internet connection is available | 2 |
| | **FA306** | Publish chosen outlines | 3 |
| | **FA307** | Information about the connection status | 3 |
| | **FA308** | Enable or disable replication | 3 |
| **Conflicts** | **FA400** | Automatically resolve one conflict type | 1 |
| | **FA401** | Manually resolve one conflict type | 1 |
| | **FA402** | Resolve a combination of two conflict types | 2 |
| | **FA403** | Resolve conflicts between more than two replicas | 2 |
| | **FA404** | Automatically resolve multiple conflict types | 3 |
| | **FA405** | Manually resolve multiple conflict types | 3 |

Abbildung A.8: Requirements of the system

## A.5 Addendum to the system documentation

### A.5.1 Business class diagram

Figure A.9 represents an overview of the application's core business classes. Since JavaScript is not used as a class-oriented language, these aren't classes in the narrow sense. A UML class is used for a JavaScript function, local variables for its attributes. In order to implement class methods, the corresponding functions are added to the function's prototypes. An UML association means that the „class" - in fact a function - calls another „class" in one of its methods.

The diagram illustrates how business classes behave vis-á-vis each other. The leftmost controllers access the Mustache views, who in their turn retrieve their data from the models. Independently from the business classes, the `ConflictDetector` is called through input from the database. In turn, it may call the `ConflictPresenter` in order to display any conflicts, or it may call the `ConflictResolver` in order to resolve them. For this purpose, the latter needs data from a `NoteView` and the `NotesCollection`. The `ConflictResolver` can also be called by a `NotesView` in order to display write conflicts.

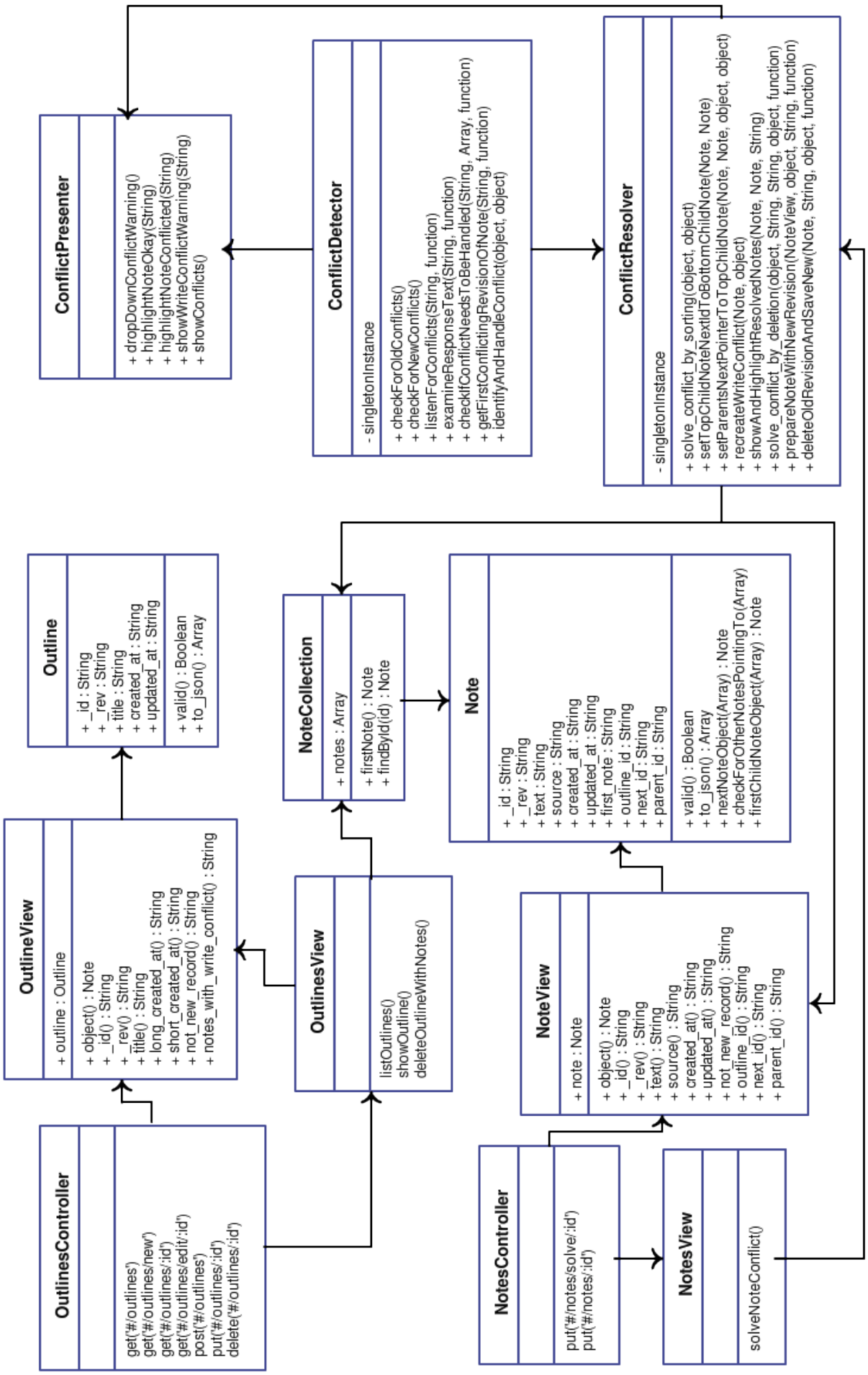


Abbildung A.9: Business class diagram

### A.5.2 Abstraction of basic database operations

```
var Resources = function(app, couchapp) {
  this.helpers({
    new_object: function(kind, callback) {
      this.partial(template_file_for(kind, 'new'), callback);
    },
    \\ [...]

    load_object_view: function(kind, id, callback){
      var context = this;
      couchapp.db.openDoc(id, {
        success: function(doc) {
          var _prototype = eval(kind);
          var view_prototype = eval(kind + 'View');
          var view = new view_prototype(new _prototype(doc));
          if(doc) {
            callback(view);
          } else {
            context.flash = {message: kind + ' with ID "' + id + '" not found.', type: '
                error'};
          }
        },
        error: function() {
          context.notFound();
        }
      });
    },

    \\ [...]
  });
};
```

Listing A.1: Extract from `/_attachments/app/lib/resources.js`

### A.5.3 Indentation of an outline's line

```
<li class="edit-note" id="edit_note_2">
  <form class="edit-note" action="#/notes/2" method="put" accept-charset="utf-8">
    <span class="space"> </span>
    <a class="image"> </a>
    <textarea class="expanding" id="edit_text_2" name="text">Some text</textarea>
    <input type="submit" value="Save" style="display:none;"/>
  </form>

```

```
  <ul class="indent">
    <li class="edit-note" id="edit_note_2a">
      <form class="edit-note" action="#/notes/2a" method="put" accept-charset="utf-8">
        <span class="space"> </span>
        <a class="image"> </a>
        <textarea class="expanding" id="edit_text_2a" name="text">More text</textarea>
        <input type="submit" value="Save" style="display:none;"/>
      </form>
    </li>
  </ul>
</li>
```

Listing A.2: Line with child nodes

### A.5.4 Rendering an outline's lines

```
renderNotes: function(context, notes, counter){
  if (notes.notes.length == 0) return;
  if (notes.notes.length == 1) {
    context.unbindSubmitOnBlurAndAutogrow();
    context.bindSubmitOnBlurAndAutogrow();
    $('#spinner').hide();
    context.i = 0;
  }
  if(typeof(context.i)=="undefined"){
    context.i = counter;
  } else {
    context.i = context.i-1;
  }
  var note_object = notes.findById(this.id());
  var child_object = note_object.firstChildNoteObject(notes.notes);
  var next_object = note_object.nextNoteObject(notes.notes);

  notes.notes = notes.notes.remove(note_object);

  if(typeof(child_object)!="undefined"){
    this.renderFollowingNote(context, child_object, function(child){
      child.renderNotes(context, notes);
    });
  }
  if(typeof(next_object)!="undefined"){
    this.renderFollowingNote(context, next_object, function(next){
      next.renderNotes(context, notes);
    });
  }
  if(typeof(next_object)=="undefined" && typeof(child_object)=="undefined"){
```

```
    context.unbindSubmitOnBlurAndAutogrow();
    context.bindSubmitOnBlurAndAutogrow();
  }
},

renderFollowingNote: function(context, note_object, callback){
  var this_note = this;
  context.partial('app/templates/notes/edit.mustache', {_id: note_object._id, text:
      note_object.text}, function(html) {
    if(typeof note_object.parent_id != "undefined"){
      $(html).appendTo(this_note.note_target.closest('li')).wrap('<ul class="indent"></
          ul>');
      callback(this_note.firstChildNote());
    } else {
      $(html).insertAfter(this_note.note_target.closest('li'));
      callback(this_note.nextNote());
    }
  });
}
```

Listing A.3: Extract from `/_attachments/app/helpers/note_element.js`

### A.5.5 Monitoring the database for changes by others

```
checkForUpdates: function(couchapp){
  var context   = this;
  var source    = context.getLocationHash();
  var url       = config.HOST + '/' + config.DB +
                  '/_changes?filter=doingnotes/changed&source=' + source;

  if(context.getOutlineId()){
    $.getJSON(url, function(json){
      var since = json.last_seq;
      var xmlhttp = new XMLHttpRequest();
      xmlhttp.onreadystatechange=function() {
        if(xmlhttp.readyState == 3){
          if(xmlhttp.responseText.match(/changes/)){
            var lines = xmlhttp.responseText.split("\n");
            if(lines[lines.length-2].length != 0){
              lines = lines.remove("");
              $.each(lines, function(i, line){
                context.parseLineAndShowChangesWarning(context, couchapp, line, lines);
              });
            }
          }
          if(xmlhttp.responseText.match(/last_seq/)){
```

```
23            Sammy.log('Timeout in checkForUpdates:', xmlhttp.responseText)
24          }
25        }
26      }
27      xmlhttp.open("GET", url+ '&feed=continuous&heartbeat=5000&since=' +since, true);
28      xmlhttp.send(null);
29    });
30  }
31 }
```

Listing A.4: `/_attachments/app/helpers/replication_helpers.js`

### A.5.6 Patch for CouchDB's changes filter

The function `make_filter_fun` creates a function that loads the corresponding document for each line in the changes feed. With the aid of the document's JSON representation, the indicated filter function can decide whether the line should be included in the returned changes feed. The `_conflicts` array is added to the JSON object in line 17, which can then be used by filter functions.

```
1  make_filter_fun(Req, Db) ->
2      Filter = couch_httpd:qs_value(Req, "filter", ""),
3      case [list_to_binary(couch_httpd:unquote(Part))
4              || Part <- string:tokens(Filter, "/")] of
5      [] ->
6          fun(DocInfos) ->
7              [{[{rev, couch_doc:rev_to_str(Rev)}]} ||
8                  #doc_info{revs=[#rev_info{rev=Rev}|_]} <- DocInfos]
9          end;
10     [DName, FName] ->
11         DesignId = <<"_design/", DName/binary>>,
12         DDoc = couch_httpd_db:couch_doc_open(Db, DesignId, nil, []),
13         #doc{body={Props}} = DDoc,
14         couch_util:get_nested_json_value({Props}, [<<"filters">>, FName]),
15         fun(DocInfos) ->
16             Docs = [Doc || {ok, Doc} <- [
17                 {ok, Doc} = couch_db:open_doc(Db, DInfo, [deleted, conflicts])
18                 || DInfo <- DocInfos]],
19             {ok, Passes} = couch_query_servers:filter_docs(Req, Db, DDoc, FName, Docs),
20             [{[{rev, couch_doc:rev_to_str(Rev)}]}
21                 || #doc_info{revs=[#rev_info{rev=Rev}|_]} <- DocInfos,
22                 Pass <- Passes, Pass == true]
23         end;
24     _Else ->
25         throw({bad_request,
```

```
            "filter parameter must be of the form `designname/filtername`"})
    end.
```

Listing A.5: Function `make_filter_fun`

For testing purposes a design document was added to the test database. This document contains a list function returning all conflicting documents.

```
var ddoc = {
  _id : "_design/changes_filter",
  "filters" : {
    "conflicted" : "function(doc, req) { return (doc._conflicts);}",
  }
}

var id = db.save({'food' : 'pizza'}).id;
db.bulkSave([{_id: id, 'food' : 'pasta'}], {all_or_nothing:true});

req = CouchDB.request("GET", "/test_suite_db/_changes?filter=changes_filter/conflicted
    ");
resp = JSON.parse(req.responseText);
T(resp.results.length == 1);
```

Listing A.6: Test for the function in listing A.6

### A.5.7 HTML file for executing the unit tests

```
<html>
  <head>
    <link rel="stylesheet" href="jspec/jspec.css" type="text/css"/>
    <script src="../../vendor/jquery/_attachments/jquery.js"></script>
    <script src="jspec/jspec.js"></script>
    <script src="jspec/jspec.jquery.js"></script>
    <script src="jspec/jspec.xhr.js"></script>
    <script src="../app/lib/lib.js"></script>
    <script src="../app/lib/resources.js"></script>
    <script src="../config/config.js"></script>
    <script src="../app/helpers/key_events.js"></script>
    <script src="../app/helpers/note_element.js"></script>
    <script src="../app/helpers/outline_helpers.js"></script>
    <script src="../app/models/note.js"></script>
    <script src="../app/models/outline.js"></script>
    <script src="../app/models/note_collection.js"></script>
    <script>
      function runSuites() {
```

```
        JSpec
        .exec('note_element_spec.js')
        .exec('inserting_note_element_spec.js')
        .exec('indenting_note_element_spec.js')
        .exec('unindenting_note_element_spec.js')
        .exec('focusing_note_element_spec.js')
        .exec('rendering_note_element_spec.js')
        .exec('outline_helpers_spec.js')
        .exec('outline_spec.js')
        .exec('note_spec.js')
        .exec('note_collection_spec.js')
        .exec('resources_spec.js')
        .exec('lib_spec.js')
        .exec('conflict_spec.js')
        .run({failuresOnly: true, fixturePath: 'fixtures'})
        .report();
      }
    </script>
  </head>
  <body class="jspec" onLoad="runSuites();">
    <div id="jspec"></div>
  </body>
</html>
```

Listing A.7: `/_attachments/spec/index.html`

## A.5.8 Test suite for CouchDB's JavaScript API

### A.5.8.1 Extract from CouchDB's JavaScript API

```
(function($) {
  \\ [...]
  $.extend($.couch, {
    \\ [...]
    db: function(name) {
      \\ [...]
      return {
        \\ [...]
        removeDoc: function(doc, options) {
          ajax({
              type: "DELETE",
              url: this.uri +
                   encodeDocId(doc._id) +
                   encodeOptions({rev: doc._rev})
            },
```

```
16            options,
17            "The document could not be deleted"
18          );
19        }
20        \\ [...]
21      };
22    }
23    \\ [...]
24  });
25 })(jQuery);
```

Listing A.8: Extract from `/share/www/script/jquery.couch.js`

### A.5.8.2 Test for listing A.8

```
1  describe 'jQuery couchdb db'
2    \\ [...]
3
4    before_each
5      db = $.couch.db('spec_db');
6      db.create();
7    end
8
9    after_each
10     db.drop();
11   end
12
13   describe 'removeDoc'
14     before_each
15       doc = {"Name" : "Louanne Katraine", "Callsign" : "Kat", "_id" : "345"};
16       saved_doc = {};
17       db.saveDoc(doc, {
18         success: function(resp){
19           saved_doc = resp;
20         },
21         error: function(status, error, reason){errorCallback(status, error, reason)}
22       });
23     end
24
25     it 'should result in a deleted document'
26       db.removeDoc({_id : "345", _rev : saved_doc.rev}, {
27         success: function(resp){
28           db.openDoc("345", {
29             error: function(status, error, reason){
30               status.should.eql 404
31               error.should.eql "not_found"
```

```
            reason.should.eql "deleted"
          },
          success: function(resp){successCallback(resp)}
        });
      },
      error: function(status, error, reason){errorCallback(status, error, reason)}
    });
  end

  it 'should return ok true, the ID and the revision of the deleted document'
    db.removeDoc({_id : "345", _rev : saved_doc.rev}, {
      success: function(resp){
        resp.ok.should.be_true
        resp.id.should.eql "345"
        resp.rev.should.be_a String
        resp.rev.length.should.be_at_least 30
      },
      error: function(status, error, reason){errorCallback(status, error, reason)}
    });
  end

  it 'should record the revision in the deleted document'
    db.removeDoc({_id : "345", _rev : saved_doc.rev}, {
      success: function(resp){
        db.openDoc("345", {
          rev: resp.rev,
          success: function(resp2){
            resp2._rev.should.eql resp.rev
            resp2._id.should.eql resp.id
            resp2._deleted.should.be_true
          },
          error: function(status, err, rsn){errorCallback(status, err, rsn)}
        });
      },
      error: function(status, error, reason){errorCallback(status, error, reason)}
    });
  end

  it 'should alert with an error message prefix'
    db.removeDoc({_id: "asdf"});
    alert_msg.should.match /The document could not be deleted/
  end
end

\\ [...]
end
```

Listing A.9: Extract from `/share/www/spec/jquery_couch_3_spec.js`

### A.5.9 Screenshots from the AWS management console

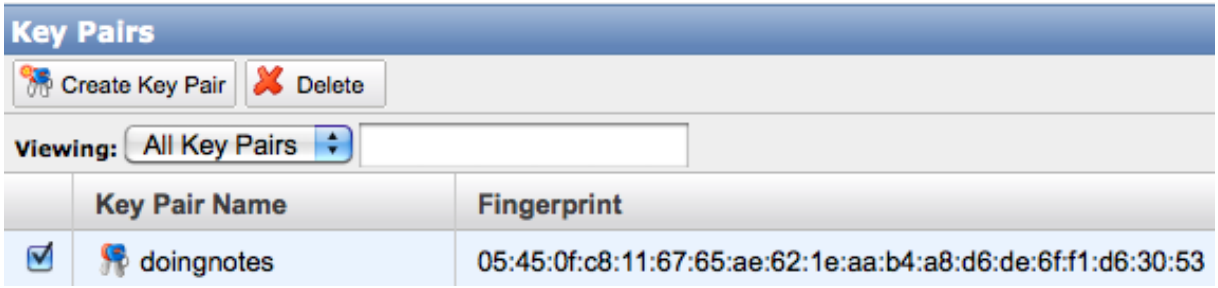


Abbildung A.10: AWS: key pair for authentification

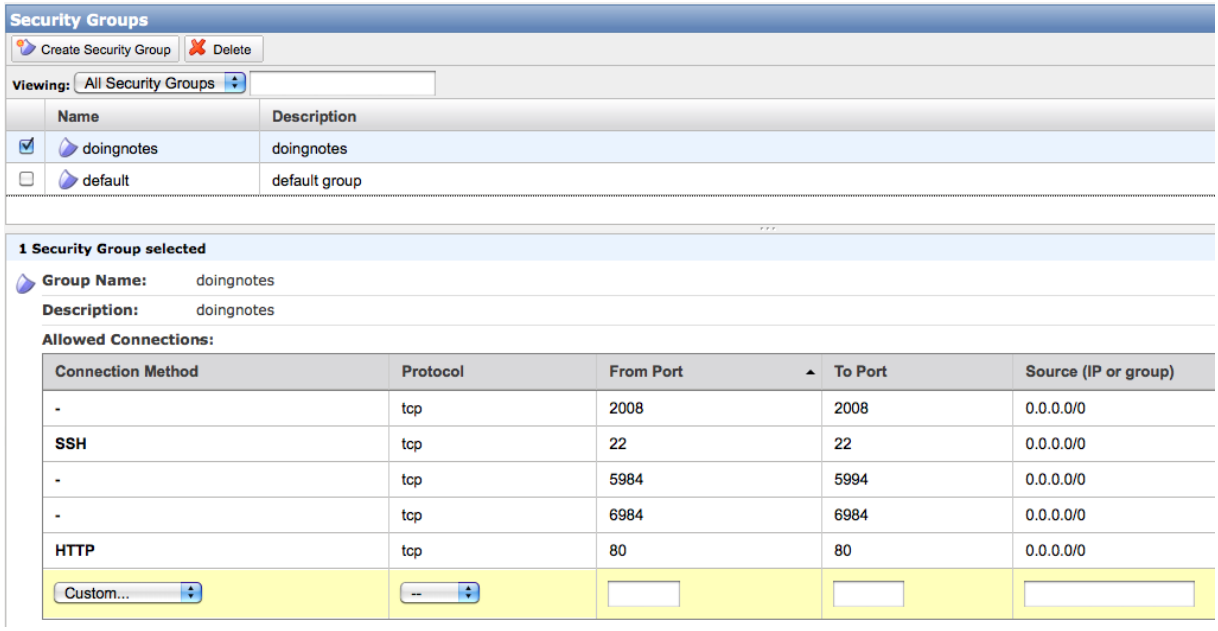


Abbildung A.11: AWS: opening ports using security groups

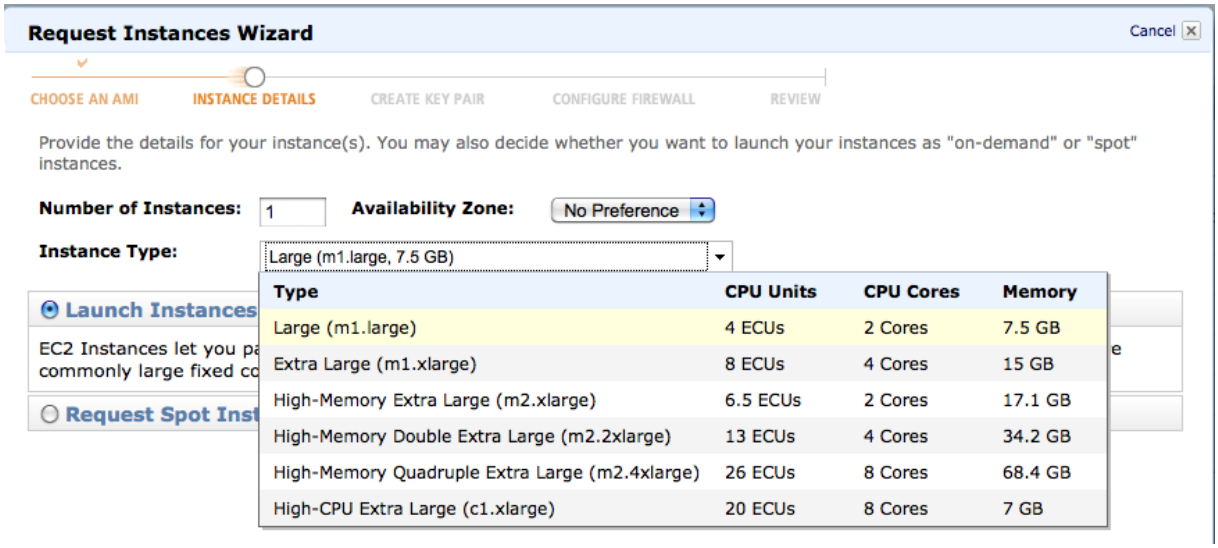


Abbildung A.12: AWS: choosing instance capacity

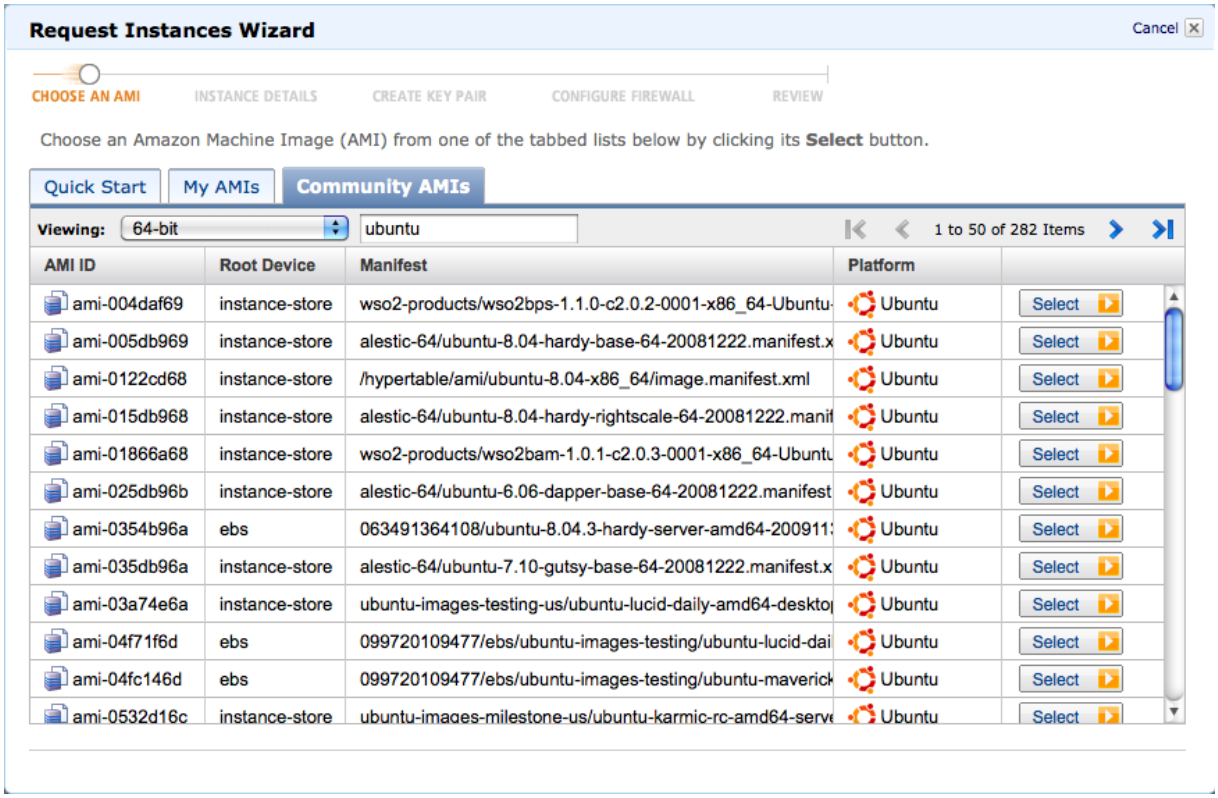


Abbildung A.13: AWS: choosing an Amazon Machine Image

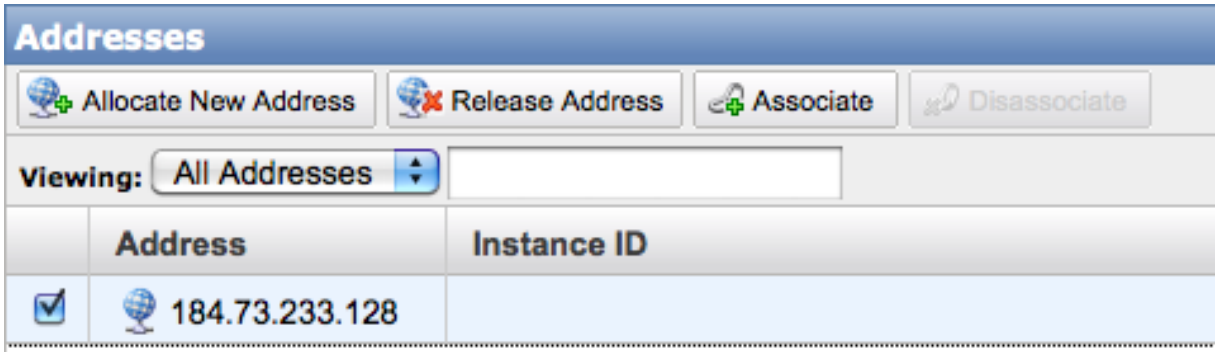


Abbildung A.14: AWS: Elastic IP set-up

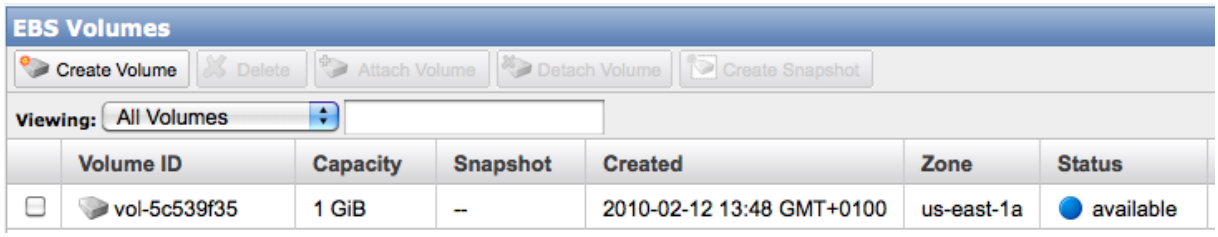


Abbildung A.15: AWS: EBS volume set-up

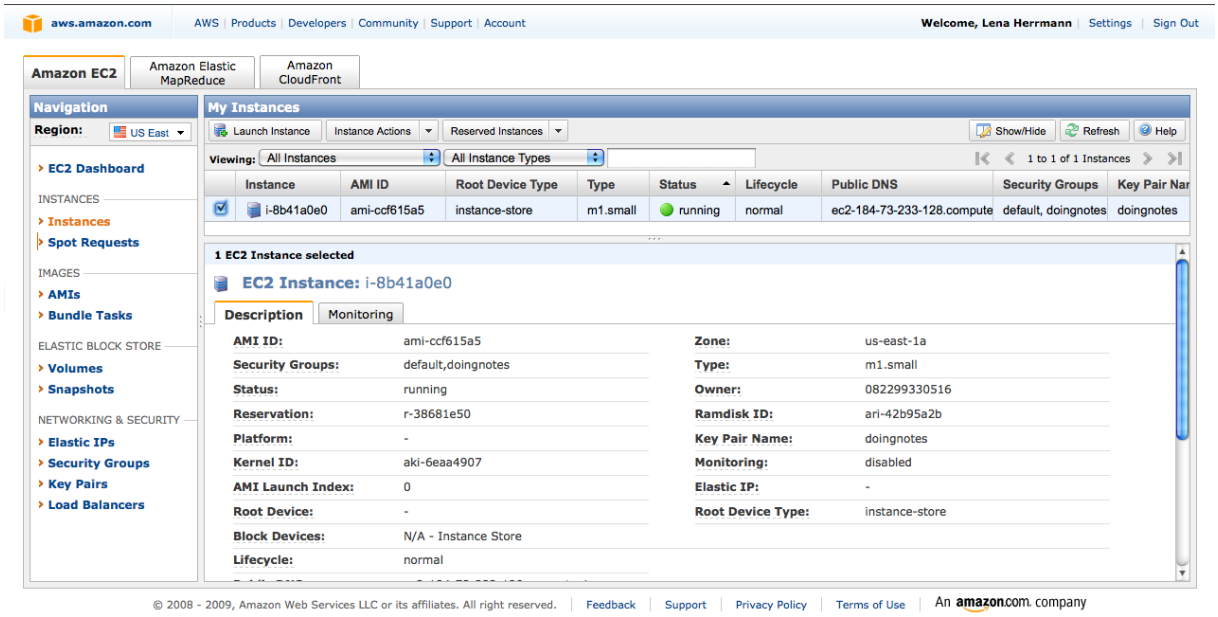


Abbildung A.16: AWS: running EC2 instance

## A.6 CD-ROM Contents

The CD-ROM supplied with this thesis contains the following information:

The file *Diplom_Lena_Herrmann.pdf* is the final version of the thesis.

The directory *Implementierung* contains the source code of the finished application.

The directory *Software* contains all software that is needed to use the application.

# Listings

# Bibliography


[Ada07] Adams, Tom: *Better Testing Through Behaviour*. Talk at Open Source Developers' Conference in Brisbane, Australia, November 2007.

[And10] Anderson, J. Chris; Lehnardt, Jan; Slater, Noah: *CouchDB: The Definitive Guide*. O'Reilly Media, Februar 2010.

[Ass98] Assfalg, Rolf; Goebels, Udo; Welter, Heinrich: *Internet Datenbanken*. Addison-Wesley, 1998.

[Bau10] Baun, Christian; Kunze, Marcel; Nimis, Jens; Tai, Stefan: *Cloud Computing. Web-basierte dynamische IT-Services*. Springer-Verlag, 2010.

[Bay08] Bayer, Rudolf: *B-tree and UB-tree*. Scholarpedia, 3(11):7742, 2008.

[Ben04] Bengel, Günther: *Verteilte Systeme. 3. Auflage*. Vieweg, 2004.

[Ber81] Bernstein, Philip A.; Goodman, Nathan: *Concurrency Control in Distributed Database Systems*. ACM Comput. Surv., 13(2):185–221, 1981.

[Ble08] Bleek, Wolf-Gideon; Wolf, Henning: *Agile Softwareentwicklung*. dpunkt.verlag, Februar 2008.

[Bre00] Brewer, Eric A.: *Towards robust distributed systems (abstract)*. *PODC*, 7, 2000.

[Cha06] Chang, Fay et.al.: *Bigtable: A Distributed Storage System for Structured Data*. OSDI'06: Seventh Symposium on Operating System Design and Implementation, 2006.

[Che07] Chess, Brian; Tsipenyuk O'Neil, Yekaterina; West, Jacob: *Javascript Hijacking*. März 2007.

[Cod83] Codd, Edgar Frank: *A relational model of data for large shared data banks*. Communications of the ACM, 26(1):64–69, 1983.

[Cor01] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: *Introduction to Algorithms*. The MIT Press, 2nd Revised edition , September 2001.

[Cou05] Coulouris, George; Dollimore, Jean; Kindberg, Tim: *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series)*. Addison Wesley, May 2005.

[Cro08] Crockford, Douglas: *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[Edl95] Edlich, Stefan: *Kooperationsmechanismen synchroner Groupwaresysteme. Entwurf, Realisierung und Einsatz eines Werkzeugsatzes für replizierte Softwarekooperation*. Technische Universität Berlin, 1995.

[Ell89] Ellis, Clarence A.; Gibbs, Simon John D.: *Concurrency control in groupware systems*. SIGMOD Rec., 18(2):399–407, 1989.

[Fie96] Fielding, R.; Frystyk, H.; Berners-Lee, T.; Gettys, J.; Mogul, Jeffrey C.: *Hypertext Transfer Protocol - HTTP/1.1*, 1996.

[Fie00] Fielding, Roy: *Architectural Styles and the Design of Network-based Software Architectures.* , University of California, 2000.

[Fis08] Fischer, Jens-Christian: *Professionelle Webentwicklung mit Ruby on Rails 2*. mitp, 2008.

[Gil02] Gilbert, Seth; Lynch, Nancy: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News, 33(2):51–59, 2002.

[GJ05] Godwin-Jones, Robert: *Ajax and Firefox: New Web Applications and Browsers*. Language Learning and Technology, 9(2):8–12, 2005.

[Gra96] Gray, Jim; Helland, Pat; O'Neil, Patrick; Shasha, Dennis: *The dangers of replication and a solution. SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 173–182, New York, NY, USA, 1996. ACM.

[Guy98] Guy, Richard; Reiher, Peter; Ratner, David; Gunter, Michial; Ma, Wilkie; Popek, Gerald: *Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication*. 1998.

[Gü04] Güting, Ralf Hartmut; Dieker, Stefan: *Datenstrukturen und Algorithmen*. Teubner Verlag, Dezember 2004.

[Ham08] Hamill, Paul: *Unit Test Frameworks*. O'Reilly Media, November 2008.

[Hes92] Hesse, Wolfgang; Merbeth, Günther; Frölich, Rainer: *Software-Entwicklung. Vorgehensmodelle, Projektführung, Produktverwaltung*. R. Oldenbourg Verlag, 1992.

[Hup04] Hupfeld, Felix: *Log-Structured Storage for Efficient Weakly-Connected Replication*. ICDCS Workshops, 458–463, März 2004.

[Hup09] Hupfeld, Felix: *Causal Weak-Consistency Replication – A Systems Approach.* , Humboldt-Universität zu Berlin, Januar 2009.

[Hä01] Härder, Theo; Rahm, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 2001.

[Ini10] Initiative D21: *(N)ONLINER Atlas 2010. Eine Topographie des digitalen Grabens durch Deutschland. Nutzung und Nichtnutzung des Internets, Strukturen und regionale Verteiltung.* Juli 2010.

[Kar95] Karsten, Helena: *"It's like everyone working around the same desk": organisational readings of Lotus Notes.* Scand. J. Inf. Syst., 7(1):3–32, 1995.

[Kaw88] Kawell,Jr., Leonard; Beckhardt, Steven; Halvorsen, Timothy; Ozzie, Raymond; Greif, Irene: *Replicated document management in a group communication system. CSCW '88: Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, 395. ACM, 1988.

[Kli96] Klieb, Leslie: *Distributed disconnected databases. SAC '96: Proceedings of the 1996 ACM symposium on Applied Computing*, 322–326. ACM, 1996.

[Knu97] Knuth, Donald: *The art of computer programming vol 1. Fundamental Algorithms, Dritte Auflage.* Addison-Wesley, 1997.

[Kot99] Kotz, D, ; Gray R.S.: *Mobile Agents and the Future of the Internet.* 7–13, August 1999.

[Kra09a] Krasnova, Hanna; Günther, Oliver; Spiekermann, Sarah; Koroleva, Ksenia: *Privacy Concerns and Identity in Social Networks (submitted).* Juli 2009.

[Kra09b] Krasnova, Hanna; Spiekermann, Sarah; Thomas, Hildebrand: *Online Social Networks: Why so we disclose (submitted).* Juli 2009.

[KT04] Khare, Rohit Richard N. Taylor: *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 428–437, Washington, DC, USA, 2004. IEEE Computer Society.

[Lam78] Lamport, Leslie: *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 21(7), Juli 1978.

[Lam01] Lamport, Leslie: *Paxos Made Simple.* ACM SIGACT News, 32(4):18–25, December 2001.

[Leh08] Lehnardt, Jan: *Introduction into CouchDB.* Talk at BBC London, August 2008.

[Leh09] Lehnardt, Jan: *Peer-to-peer Applications with CouchDB.* Talk at NoSQL Berlin, Oktober 2009.

[Leh10] Lehnardt, Jan: *Making Software for Humans - CouchDB and the Usable Peer-To-Peer Web.* Talk at Berlin Buzzwords, June 2010.

[Mat07] Matthiessen, Günter; Unterstein, Michael: *Relationale Datenbanken und Standard-SQL - Konzepte der Entwicklung und Anwendung*. Addison-Wesley, Dezember 2007.

[Meh97] Mehta, Manish; DeWitt, David J.: *Data placement in shared-nothing parallel database systems*. The VLDB Journal, 6(1):53–72, 1997.

[Mil68] Miller, Robert B.: *Response time in man-computer conversational transactions*. Proceedings of the AFIPS Fall Joint Computer Conference, 33:267–277, 1968.

[Mon01] Montoya-Weiss, Mitzi M.; Massey, Anne P.; Song, Michael: *Getting It Together: Temporal Coordination and Conflict Management in Global Virtual Teams*. The Academy of Management Journal, 44(6):1251–1262, Dezember 2001.

[Mos09] Moser, Monika: *Consistency in Distributed Key/Value Stores*. Talk at NoSQL Berlin, Oktober 2009.

[Nie93] Nielsen, Jakob: *Usability Engineering (Interactive Technologies)*. Morgan Kaufmann, November 1993.

[Orl92] Orlikowski, Wanda: *Learning from NOTES: Organizational Issues in Groupware Implementation*. 134, MIT Center for Coordination Science, August 1992.

[Par10] Parveen, Suraiya; Tanweer, Safrad: *A Study on Effect of Replication on Databases*. *Proceedings of the 4th National Conference; INDIACom-2010*, 483–490, Februar 2010.

[Pau05] Paulson, Linda Dailey: *Building Rich Web Applications with Ajax*. Computer, 38(10):14–17, 2005.

[Pea80] Pease, Marshall; Shostak, Robert; Lamport, Leslie: *Reaching Agreement in the Presence of Faults*. J. ACM, 27(2):228–234, 1980.

[Qia09] Qian, Ling; Luo, Zhiguo; Du, Yujian; Guo, Leitao: *Cloud Computing: An Overview*. *Cloud Computing - First International Conference, CloudCom 2009*, 626–631. Springer, Dezember 2009.

[Ray07] Raymond, Scott: *Ajax on Rails*. O'Reilly Media, Inc., 2007.

[Sai05] Saito, Yasushi; Shapiro, Marc: *Optimistic replication*. ACM Comput. Surv., 37(1):42–81, 2005.

[Ski07] Skiadas, C.; Kjosmoen, T.: *LATEXing with TextMate*. The PracTEX Journal, (3), 2007.

[Spr97] Spreitzer, Mike; Theimer, Marvin M.; Petersen, Karin; J, Alan; Terry, Douglas B.: *Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems*. *In Proc. of ACM/IEEE MobiCom Conf*, 234–240, 1997.

[Sto86] Stonebreaker, Michael: *The case for shared nothing*. IEEE Database Engineering Bulletin, 9(1):4–9, 1986.

[Tan07] Tanenbaum, Andrew S.; van Steen, Maarten: *Verteilte Systeme*. Pearson Studium, 2., Aufl. , 2007.

[Van96] Vandenbosch, Betty; Ginzberg, Michael J.: *Lotus notes®and collaboration: plus ça change...* J. Manage. Inf. Syst., 13(3):65–81, 1996.

[Var09] Varley, Ian Thomas: *No Relation: The Mixed Blessings of Non-Relational Databases*. , University of Texas at Austin, August 2009.

[Vog09] Vogels, Werner: *Eventually Consistent. Building reliable distributed systems at a world-wide scale demands trade-offs between consistency and availability*. Communications of the ACM, 52(1), Januar 2009.

[Yan07] Yang, Hung-chih; Dasdan, Ali; Hsiao, Ruey-Lung; Parker, D. Stott: *Map-reduce-merge: simplified relational data processing on large clusters. SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 1029–1040, New York, NY, USA, 2007. ACM.

# Internet Resources


[10g10] 10gen: *mongoDB Replication*. http://www.mongodb.org/display/DOCS/Replication, 2010. zuletzt abgerufen am 16.06.2010.

[Alf08] Alfresco Software, Inc: *The Open Source Barometer III*. hhttp://www.alfresco.com/community/barometer/, Februar 2008. zuletzt abgerufen am 26.06.2010.

[All10] Allsopp, John: *The State of Web Development 2010*. http://www.webdirections.org/sotw10/, April 2010. zuletzt abgerufen am 14.06.2010.

[Amaa] Amazon Web Services LLC: *Amazon EC2-Instanztypen*. http://aws.amazon.com/de/ec2/instance-types/. zuletzt abgerufen am 20.06.2010.

[Amab] Amazon Web Services LLC: *AWS Simple Monthly Calculator*. http://calculator.s3.amazonaws.com/calc5.html. zuletzt abgerufen am 20.06.2010.

[Amb] Ambler, Scott W.: *Disciplined Agile Software Development: Definition*. http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm. zuletzt abgerufen am 20.06.2010.

[And10] Anderson, J. Chris: *Evently Docs*. http://github.com/jchris/toast/blob/master/vendor/couchapp/docs/evently.md, 2010. zuletzt abgerufen am 07.07.2010.

[Apa09a] Apache Software Foundation: *The Apache Cassandra Project*. http://cassandra.apache.org/, 2009. zuletzt abgerufen am 06.07.2010.

[Apa09b] Apache Software Foundation: *Apache CouchDB: Technical Overview*. http://couchdb.apache.org/docs/overview.html, 2009. zuletzt abgerufen am 26.06.2010.

[Apa09c] Apache Software Foundation: *Apache CouchDB: The CouchDB Project*. http://couchdb.apache.org/, 2009. zuletzt abgerufen am 05.06.2010.

[Apa09d] Apache Software Foundation: *Graduated from incubation*. http://incubator.apache.org/projects/, 2009. zuletzt abgerufen am 24.06.2010.

[Apa10a] Apache Github Repository: *Apache CouchDB Commit: Show conflicts in changes filters. Patch by Lena Herrmann*. http://github.com/apache/couchdb/commit/5a87b852a72cc36d4d7b64271ed542ce2a1befe0, 2010. zuletzt abgerufen am 04.06.2010.

[Apa10b] Apache Software Foundation: *Apache Subversion*. http://subversion.apache.org/, 2010. zuletzt abgerufen am 16.06.2010.

[Apa10c] Apache Software Foundation: *CouchDB Github Repository*. http://github.com/apache/couchdb, 2010. zuletzt abgerufen am 06.06.2010.

[Apa10d] Apache Software Foundation: *CouchDB Subversion Repository*. https://svn.apache.org/repos/asf/couchdb/, 2010. zuletzt abgerufen am 06.06.2010.

[Apa10e] Apache Software Foundation: *JIRA-Ticket: Add Bulk Deletion to jquery.couch.js*. https://issues.apache.org/jira/browse/COUCHDB-612, 2010. zuletzt abgerufen am 04.06.2010.

[Apa10f] Apache Software Foundation: *JIRA-Ticket: Add tests for Javascript HTTP API*. https://issues.apache.org/jira/browse/COUCHDB-783, 2010. zuletzt abgerufen am 04.06.2010.

[Apa10g] Apache Software Foundation: *JIRA-Ticket: Make bulkSave actually save docs*. https://issues.apache.org/jira/browse/COUCHDB-634, 2010. zuletzt abgerufen am 04.06.2010.

[Apa10h] Apache Software Foundation: *JIRA-Ticket: Make the changes_filter use _conflicts field*. https://issues.apache.org/jira/browse/COUCHDB-630, 2010. zuletzt abgerufen am 04.06.2010.

[Apa10i] Apache Software Foundation: *Welcome to HBase*. http://hbase.apache.org/, 2010. zuletzt abgerufen am 06.07.2010.

[Ape09] Apestaart, Thomas: *Building erlang for the N900*. http://thomas.apestaart.org/log/?p=1086, Dezember 2009. zuletzt abgerufen am 10.06.2010.

[Ape10] Apestaart, Thomas: *desktopcouch on N900*. http://thomas.apestaart.org/log/?p=1106, Januar 2010. zuletzt abgerufen am 10.06.2010.

[App10] Apple Inc.: *Common QA for Bonjour*. http://developer.apple.com/mac/library/qa/qa2010/qa1690.html, Mai 2010. zuletzt abgerufen am 09.07.2010.

[Bad08] Bader, Chrys: *jQuery Plugins: Auto Growing Textareas*. http://plugins.jquery.com/project/autogrow, januar 2008. zuletzt abgerufen am 05.07.2010.

[Bak] Bakken, Jarib: *Celerity*. http://celerity.rubyforge.org/. zuletzt abgerufen am 21.06.2010.

[Bar09] Barreto, Charlton: *NoSQL: leading Cloud's "NoBah" movement?* http://charltonb.typepad.com/weblog/2009/07/

nosql-leading-clouds-nobah-movement.html, Juli 2009. zuletzt abgerufen am 09.05.2010.

[Bas10] Basho: *Riak: An Open Source Internet-Scale Data Store*. http://wiki.basho.com/display/RIAK/Riak, 2010. zuletzt abgerufen am 16.06.2010.

[Bec01] Beck, Kent; Beedle, Mike; van Bennekum Arie; Cockburn Alistair et. al.: *Manifesto for Agile Software Development*. http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm, Februar 2001. zuletzt abgerufen am 20.06.2010.

[Bec07] Beck, Kent: *Three Rivers Institute: Test-Driven Development Violates the Dichotomies of Testing*. http://www.threeriversinstitute.org/Testing%20Dichotomies%20and%20TDD.htm, Juni 2007. zuletzt abgerufen am 22.06.2010.

[Bel10] Belomestnykh, Olga: *A rebuilt, more real time Google documents*. http://googledocs.blogspot.com/2010/04/rebuilt-more-real-time-google-documents.html, April 2010. zuletzt abgerufen am 06.06.2010.

[Ben06] Bengtsson, Anders: *Madeleine*. http://madeleine.rubyforge.org/, 2006. zuletzt abgerufen am 12.06.2010.

[Cha10] Chacon, Scott: *git - the fast version control system*. http://git-scm.com/, 2010. zuletzt abgerufen am 16.06.2010.

[Che] Chelimsky, David: *RSpec*. http://rspec.info/. zuletzt abgerufen am 21.06.2010.

[Che10] Chesneau, Benoit: *Couchapp Github Repository*. http://github.com/couchapp/couchapp, 2010. zuletzt abgerufen am 05.06.2010.

[Cora] Corporation, Oracle: *MySQL*. http://www.mysql.com/?bydis_dis_index=1. zuletzt abgerufen am 22.06.2010.

[Corb] Corporation, Oracle: *MySQL 5.5 Replication and AUTO_INCREMENT*. http://dev.mysql.com/doc/refman/5.5/en/replication-features-auto-increment.html. zuletzt abgerufen am 26.06.2010.

[Corc] Corporation, Oracle: *MySQL 5.5 Replication Implementation*. http://dev.mysql.com/doc/refman/5.5/en/replication-implementation.html. zuletzt abgerufen am 26.06.2010.

[Cord] Corporation, Oracle: *MySQL Cluster Replication*. http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-replication.html. zuletzt abgerufen am 22.06.2010.

[Cou09] CouchDB Wiki: *Designing an application to work with replication*. http://wiki.apache.org/couchdb/How_to_design_for_replication, 2009. zuletzt abgerufen am 12.06.2010.

[cou10a] couchdb-lounge wiki: *Setting up two couch instances*. http://code.google.com/p/couchdb-lounge/wiki/SettingUpTwoCouchInstances, 2010. zuletzt abgerufen am 05.06.2010.

[Cou10b] CouchDB Wiki: *CouchDB Windows_binary_installer*. http://wiki.apache.org/couchdb/Windows_binary_installer, 2010. zuletzt abgerufen am 06.06.2010.

[Cou10c] CouchDB Wiki: *Installation of CouchDB and Dependencies*. http://wiki.apache.org/couchdb/Installation, 2010. zuletzt abgerufen am 05.06.2010.

[Cro06] Crockford, Douglas: *Network Working Group, Request for Comments 4627*. http://www.ietf.org/rfc/rfc4627.txt, Juli 2006. zuletzt abgerufen am 05.06.2010.

[Cro10] Crockford, Douglas: *JSON*. http://www.json.org/, 2010. zuletzt abgerufen am 05.06.2010.

[Dro10] Dropbox: *Dropbox*. https://www.dropbox.com/, 2010. zuletzt abgerufen am 16.06.2010.

[Eri10] Ericsson: *Open Source Erlang*. http://www.erlang.org/, 2010. zuletzt abgerufen am 06.06.2010.

[Eym08] Eymann, Torsten: *Cloud Computing*. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/uebergreifendes/Kontext-und-Grundlagen/Markt/Softwaremarkt/Geschaftsmodell-%28fur-Software-und-Services%29/Cloud-Computing/index.html?searchterm=Cloud+, September 2008. zuletzt abgerufen am 19.06.2010.

[Fel09] Fellowship of the packaging: *Easy Install - Distribute documentation*. http://packages.python.org/distribute/easy_install.html, 2009. zuletzt abgerufen am 06.06.2010.

[Fou10] Foundation, Etherpad: *Etherpad - Live Collaborative Text*. http://www.etherpad.org/, 2010. zuletzt abgerufen am 07.06.2010.

[Gar] Gargoyle Software: *HtmlUnit*. http://htmlunit.sourceforge.net/. zuletzt abgerufen am 21.06.2010.

[Gar05] Garrett, Jesse James: *Ajax: A New Approach to Web Applications*. http://www.adaptivepath.com/ideas/essays/archives/000385.php, Februar 2005. zuletzt abgerufen am 05.06.2010.

[Gar09] Gartner: *Gartner's 2009 Hype Cycle Special Report Evaluates Maturity of 1,650 Technologies*. http://www.gartner.com/it/page.jsp?id=1124212, August 2009. zuletzt abgerufen am 17.06.2010.

[Goo10] Google: *Google Wave*. http://wave.google.com/about.html, 2010. zuletzt abgerufen am 07.06.2010.

[Groa] Group, PostgreSQL GLobal Development: *PostgreSQL*. http://www.postgresql.org/. zuletzt abgerufen am 22.06.2010.

[Grob] Group, PostgreSQL GLobal Development: *Replication, Clustering, and Connection Pooling*. http://wiki.postgresql.org/index.php?title=Replication%2C_Clustering%2C_and_Connection_Pooling&oldid=10880. zuletzt abgerufen am 22.06.2010.

[Hel] Hellesøy, Aslak: *Cucumber. Behaviour Driven Development with elegance and joy*. http://cukes.info/. zuletzt abgerufen am 21.06.2010.

[Hel08] Helmkamp, Bryan: *Webrat - Ruby Acceptance Testing for Web applications*. http://github.com/brynary/webrat, 2008. zuletzt abgerufen am 06.07.2010.

[Her10] Herrmann, Lena: *JSpec - JavaScript Unit testing how it should be*. http://lenaherrmann.net/2010/01/04/jspec-javascript-unit-testing-how-it-should-be, Januar 2010. zuletzt abgerufen am 21.05.2010.

[Hic10a] Hickson, Ian: *HTML5 - A vocabulary and associated APIs for HTML and XHTML*. http://dev.w3.org/html5/spec/Overview.html, 2010. zuletzt abgerufen am 06.06.2010.

[Hic10b] Hickson, Ian: *HTML5 - A vocabulary and associated APIs for HTML and XHTML - Embedding custom non-visible data*. http://dev.w3.org/html5/spec/Overview.html#embedding-custom-non-visible-data, 2010. zuletzt abgerufen am 06.06.2010.

[Ho,08] Ho, Ricky: *CouchDB Implementation*. http://horicky.blogspot.com/2008/10/couchdb-implementation.html, Oktober 2008. zuletzt abgerufen am 18.06.2010.

[Hol10] Holowaychuk, TJ: *JSpec - JavaScript Testing Framework*. http://visionmedia.github.com/jspec/, 2010. zuletzt abgerufen am 06.06.2010.

[Hyp09] Hypertable: *Hypertable - An Open Source, High Performance, Scalable Database*. http://hypertable.org/, 2009. zuletzt abgerufen am 06.07.2010.

[IDG10] IDG BUSINESS MEDIA GMBH: *Firebug - ein Muss für Web-Entwickler*. http://www.computerwoche.de/software/software-infrastruktur/1934278/, April 2010. zuletzt abgerufen am 09.06.2010.

[Joh09] JOHNSON, NICK: *Damn Cool Algorithms: Log structured storage*. http://blog.notdot.net/2009/12/Damn-Cool-Algorithms-Log-structured-storage, Dezember 2009. zuletzt abgerufen am 17.05.2010.

[jQu10] JQUERY PROJECT: *jQuery: The Write Less, Do More, JavaScript Library*. http://www.jquery.org/, 2010. zuletzt abgerufen am 05.06.2010.

[Kap07] KAPLAN-MOSS, JACOB: *Of the Web*. http://jacobian.org/writing/of-the-web/, Oktober 2007. zuletzt abgerufen am 07.07.2010.

[Kat10] KATZ, DAMIEN: *Interview with Damien Katz – Apache CouchDB*. http://howsoftwareisbuilt.com/2010/06/18/interview-with-damien-katz-apache-couchdb/, Juni 2010. zuletzt abgerufen am 24.06.2010.

[Kla10] KLAMPAECKEL, TILL: *A toolchain for CouchDB Lounge*. http://till.klampaeckel.de/blog/archives/84-A-toolchain-for-CouchDB-Lounge.html, Januar 2010. zuletzt abgerufen am 18.06.2010.

[Lan] LANG, ALEXANDER: *Culerity*. http://github.com/langalex/culerity. zuletzt abgerufen am 21.06.2010.

[Lan09a] LANDAU, BRIAN: *Benchmarking Javascript Templating Libraries*. http://www.viget.com/extend/benchmarking-javascript-templating-libraries/, Dezember 2009. zuletzt abgerufen am 06.06.2010.

[Lan09b] LANG, ALEXANDER: *Culerity = Full Stack Rails Testing with Cucumber and Celerity*. http://upstre.am/2009/01/28/culerity-full-stack-rails-testing-with-cucumber-and-celerity/, Januar 2009. zuletzt abgerufen am 21.06.2010.

[Law10] LAW, PAUL: *Tools: Outlining via OmniOutliner*. http://lawpaul.wordpress.com/2010/02/20/tools-outlining-via-omnioutliner/, Februar 2010. zuletzt abgerufen am 10.06.2010.

[Lea05] LEACH, PAUL J.: *RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace*. http://tools.ietf.org/html/rfc4122, Juli 2005. zuletzt abgerufen am 07.07.2010.

[Lee09a] LEEDS, RANDALL: *couchdb-lounge wiki*. http://wiki.github.com/tilgovi/couchdb-lounge/, 2009. zuletzt abgerufen am 18.06.2010.

[Lee09b] Leeds, Randall: *Partitioning and Clustering Support for CouchDB. Proposal for Google Summer of Code 2009*. http://socghop.appspot.com/document/show/user/rleeds/couchdb_cluster, April 2009. zuletzt abgerufen am 18.06.2010.

[Lee10] Leeds, Randall: *Lounge - a proxy-based partitioning/clustering framework for CouchDB*. http://tilgovi.github.com/couchdb-lounge/, 2010. zuletzt abgerufen am 16.06.2010.

[Lef02] Lefkowitz, Glyph; Zadka, Moshe: *The Twisted Network Framework*. http://www.python.org/workshops/2002-02/papers/09/index.htm, Februar 2002. zuletzt abgerufen am 18.06.2010.

[Leh10a] Lehnardt, Jan: *Couchio: What's new in Apache CouchDB 0.11 — Part Three: New Features in Replication*. http://blog.couch.io/post/468392274/whats-new-in-apache-couchdb-0-11-part-three-new, März 2010. zuletzt abgerufen am 07.07.2010.

[Leh10b] Lehnardt, Jan: *Couchio: Mustache.js*. http://blog.couch.io/post/622014913/mustache-js, Mai 2010. zuletzt abgerufen am 06.06.2010.

[Leh10c] Lehnardt, Jan: *mustache.js Github Repository*. http://github.com/janl/mustache.js, 2010. zuletzt abgerufen am 06.06.2010.

[Leh10d] Lehnardt, Jan; Grosenbach, Geoffrey; Gretar, Jon: *CouchDBX Github Repository*. http://janl.github.com/couchdbx/, 2010. zuletzt abgerufen am 05.06.2010.

[Len07] Lenz, Christopher: *CouchDB „Joins"*. http://www.cmlenz.net/archives/2007/10/couchdb-joins, Oktober 2007. zuletzt abgerufen am 13.06.2010.

[Len09] Lennon, Joe: *Exploring CouchDB. A document-oriented database for Web applications*. http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html, März 2009. zuletzt abgerufen am 24.06.2010.

[Mac05] Macworld: *The Omni Group OmniOutliner 3 Professional Utility Software Review*. http://www.macworld.com/reviews/product/405814/review/omnioutliner_3_professional.html, Juni 2005. zuletzt abgerufen am 15.06.2010.

[Mac10] MacroMates: *textmate - the missing editor*. http://macromates.com/, 2010. zuletzt abgerufen am 06.06.2010.

[Man09] Manjoo, Farhad: *It's Just Fancy Talk - The Google Wave chatting tool is too complicated for its own good*. http://www.slate.com/id/2232311/pagenum/all/, Oktober 2009. zuletzt abgerufen am 07.05.2010.

[Mat10] Matsumoto, Yukihiro: *Ruby - A Programmer's best friend*. http://www.ruby-lang.org/de/, 2010. zuletzt abgerufen am 06.06.2010.

[Max06] Maxia, Giuseppe: *Advanced MySQL Replication Techniques*. http://onlamp.com/pub/a/onlamp/2006/04/20/advanced-mysql-replication.html, April 2006. zuletzt abgerufen am 22.06.2010.

[Mem09] MemcacheDB: *MemcacheDB: A distributed key-value storage system designed for persistent*. http://memcachedb.org/, 2009. zuletzt abgerufen am 06.07.2010.

[Mil07] Miller, Jeremy D.: *BDD, TDD, and the other Double D's*. http://codebetter.com/blogs/jeremy.miller/archive/2007/09/06/bdd-tdd-and-the-other-double-d-s.aspx, September 2007. zuletzt abgerufen am 21.06.2010.

[Mil10] Miller, Aaron: *Android CouchDB releases*. http://apage43.github.com/, 2010. zuletzt abgerufen am 10.06.2010.

[Mon] Montoya, Christian: *Blueprint: A CSS Framework*. http://www.blueprintcss.org/. zuletzt abgerufen am 04.07.2010.

[Moza] Mozilla Foundation: *Add-ons für Firefox - Beliebte Add-ons*. https://addons.mozilla.org/de/firefox/browse/type:1/cat:all?sort=popular. zuletzt abgerufen am 09.06.2010.

[Mozb] Mozilla Foundation: *Firebug Statistics*. https://addons.mozilla.org/en-US/firefox/statistics/addon/1843. zuletzt abgerufen am 09.06.2010.

[Mozc] Mozilla Foundation: *Rhino: JavaScript for Java*. http://www.mozilla.org/rhino/. zuletzt abgerufen am 22.06.2010.

[Moz10a] Mozilla Foundation: *Firebug - Web Development Evolved*. http://getfirebug.com/, 2010. zuletzt abgerufen am 06.06.2010.

[Moz10b] Mozilla Foundation: *Firefox 3.6 – Versionshinweise*. http://www.mozilla-europe.org/de/firefox/3.6/releasenotes/, Januar 2010. zuletzt abgerufen am 07.05.2010.

[Moz10c] Mozilla Foundation: *SpiderMonkey (JavaScript-C) Engine*. http://www.mozilla.org/js/spidermonkey/, 2010. zuletzt abgerufen am 06.06.2010.

[Moz10d] Mozilla Foundation: *Webbrowser Firefox*. http://getfirefox.com/, 2010. zuletzt abgerufen am 05.06.2010.

[mus10] *Mustache*. http://mustache.github.com/, 2010. zuletzt abgerufen am 06.06.2010.

[NOR] NORD Software Consulting: *Classification of HTTP-based APIs*. http://nordsc.com/ext/classification_of_http_based_apis.html. zuletzt abgerufen am 06.06.2010.

[NOR10] NORD Software Consulting: *Classifying the CouchDB API*. http://www.slate.com/id/2232311/pagenum/all/, Februar 2010. zuletzt abgerufen am 06.06.2010.

[Omn10] Omni Group: *OmniOutliner 3*. http://www.omnigroup.com/products/omnioutliner/, 2010. zuletzt abgerufen am 10.06.2010.

[Ope09] OpenSSL Project: *OpenSSL: The Open Source toolkit for SSL/TLS*. http://www.openssl.org/, 2009. zuletzt abgerufen am 06.06.2010.

[P. 09] P. T. Magazin: *Cloud Computing und die neuen Maßstäbe der Google-Ära*. http://www.pt-magazin.de/newsartikel/datum/2009/02/09/cloud-computing-und-die-neuen-massstaebe-der-google-aera/, Februar 2009. zuletzt abgerufen am 17.06.2010.

[Pat05] Paterson, Jim: *Prevalence: Transparent, Fault-Tolerant Object Persistence*. http://onjava.com/pub/a/onjava/2005/06/08/prevayler.html, Juni 2005. zuletzt abgerufen am 12.06.2010.

[Pro] Project Voldemort: *Project Voldemort - A distributed database*. http://project-voldemort.com/. zuletzt abgerufen am 06.07.2010.

[Pro10] Pronschinske, Mitchell: *CouchDB: Making it Okay to Work Offline*. http://architects.dzone.com/articles/couchdb-making-it-okay-work, Mai 2010. zuletzt abgerufen am 10.06.2010.

[Pyt10] Python Software Foundation: *Python Programming Language*. http://www.python.org/, 2010. zuletzt abgerufen am 06.06.2010.

[Qui09] Quirkey, Aaaron: *Sammy.js, CouchDB, and the new web architecture*. http://www.quirkey.com/blog/2009/09/15/sammy-js-couchdb-and-the-new-web-architecture/, September 2009. zuletzt abgerufen am 05.06.2010.

[Qui10a] Quirkey, Aaaron: *Sammy*. http://code.quirkey.com/sammy/, 2010. zuletzt abgerufen am 05.06.2010.

[Qui10b] Quirkey, Aaaron: *Sammy Documentation: Plugins*. http://code.quirkey.com/sammy/docs/plugins.html, 2010. zuletzt abgerufen am 24.06.2010.

[Sch06] Schmidt, Eric: *Conversation with Eric Schmidt hosted by Danny Sullivan*. http://www.google.com/press/podium/ses2006.html, August 2006. zuletzt abgerufen am 19.06.2010.

[Sch08] Schuster, Werner: *Damien Katz Relaxing on CouchDB*. http://www.infoq.com/interviews/CouchDB-Damien-Katz, November 2008. zuletzt abgerufen am 16.06.2010.

[Sla10] Slater, Noah: *[VOTE] Apache CouchDB 1.0.0 release, first round*. http://couchdb-development.1959287.n2.nabble.com/VOTE-Apache-CouchDB-1-0-0-release-first-round-td5267885.html#a5267885, Juli 2010. zuletzt abgerufen am 09.07.2010.

[Smi09] Smith, David: *Browser support for CSS3 and HTML5*. http://www.deepbluesky.com/blog/-/browser-support-for-css3-and-html5_72/, November 2009. zuletzt abgerufen am 06.06.2010.

[Spr10] Sprout Systems Inc. Contributors: *What is Sproutcore?* http://www.sproutcore.com/what-is-sproutcore/, 2010. zuletzt abgerufen am 15.06.2010.

[Str98] Strozzi, Carlo: *NoSQL: a non-SQL RDBMS*. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, 1998. zuletzt abgerufen am 11.06.2010.

[The10] TheCodingMonkeys: *SubEthaEdit*. http://www.subethaedit.de/index.de.html, 2010. zuletzt abgerufen am 16.06.2010.

[VG10] Van Gurp, Jilles: *CouchDB*. http://www.jillesvangurp.com/2010/01/15/couchdb/, Januar 2010. zuletzt abgerufen am 18.06.2010.

[Vog07] Vogel, Werner: *All Things Distributed: Amazon's Dynamo*. http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html, Oktober 2007. zuletzt abgerufen am 06.07.2010.

[W3C05] W3C: *Document Object Model (DOM)*. http://www.w3.org/DOM/, Januar 2005. zuletzt abgerufen am 09.07.2010.

[W3C09] W3C: *CGI: Common Gateway Interface*. http://www.w3.org/CGI/, Mai 2009. zuletzt abgerufen am 21.05.2010.

[Wei08] Weirich, Jim: *RAKE — Ruby Make*. http://rake.rubyforge.org/, 2008. zuletzt abgerufen am 09.06.2010.

[Wik10a] Wikipedia: *Gliederungseditor*. http://de.wikipedia.org/w/index.php?title=Gliederungseditor&oldid=74405760, 2010. zuletzt abgerufen am 02.06.2010.

[Wik10b] Wikipedia: *List of unit testing frameworks - Javascript*. http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#JavaScript, 2010. zuletzt abgerufen am 06.06.2010.

[Wik10c] Wikipedia: *Outliner*. http://en.wikipedia.org/w/index.php?title=Outliner&oldid=367410701, 2010. zuletzt abgerufen am 15.06.2010.

# Abbildungsverzeichnis

# Source Code Listings