Clemens Gull

Know-how ist blau.

# Web-Applikationen entwickeln mit NoSQL

> Der einfache Weg: Web-Apps mit CouchDB und CouchApp
> Erhalten Sie eine bessere Performance bei hohen Datenlasten
> Von der Datenbank bis zur fertigen Web-Applikation

**Das Buch für Datenbank-Einsteiger und Profis!**

FRANZIS

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte Daten sind im Internet über http://dnb.ddb.de abrufbar.

Alle Angaben in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. Der Verlag und der Autor sehen sich deshalb gezwungen, darauf hinzuweisen, dass sie weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernehmen können. Für die Mitteilung etwaiger Fehler sind Verlag und Autor jederzeit dankbar. Internetadressen oder Versionsnummern stellen den bei Redaktionsschluss verfügbaren Informationsstand dar. Verlag und Autor übernehmen keinerlei Verantwortung oder Haftung für Veränderungen, die sich aus nicht von ihnen zu vertretenden Umständen ergeben. Evtl. beigefügte oder zum Download angebotene Dateien und Informationen dienen ausschließlich der nicht gewerblichen Nutzung. Eine gewerbliche Nutzung ist nur mit Zustimmung des Lizenzinhabers möglich.

**© 2011 Franzis Verlag GmbH, 85540 Haar bei München**

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Das Erstellen und Verbreiten von Kopien auf Papier, auf Datenträgern oder im Internet, insbesondere als PDF, ist nur mit ausdrücklicher Genehmigung des Verlags gestattet und wird widrigenfalls strafrechtlich verfolgt.

Die meisten Produktbezeichnungen von Hard- und Software sowie Firmennamen und Firmenlogos, die in diesem Werk genannt werden, sind in der Regel gleichzeitig auch eingetragene Warenzeichen und sollten als solche betrachtet werden. Der Verlag folgt bei den Produktbezeichnungen im Wesentlichen den Schreibweisen der Hersteller.

**Lektorat:** Franz Graser, Anton Schmid
**Satz:** DTP-Satz A. Kugge, München
**art & design**: www.ideehoch2.de
**Druck:** Bercker, 47623 Kevelaer
Printed in Germany

**ISBN 978-3-645-60104-7**

# Inhaltsverzeichnis

1 **Die Theorie hinter NoSQL** ........ **17**
1.1 Die Geschichte ........ 17
1.1.1 Das Konzept NoSQL ........ 18
1.2 Arten von NoSQL-Datenbanken ........ 19
1.2.1 Dokumentenorientiert ........ 19
1.2.2 Key-Value-orientiert ........ 19
1.2.3 Spaltenorientiert ........ 20
1.2.4 Graphenorientiert ........ 20
1.2.5 Übersicht von NoSQL-Datenbanken ........ 20
1.3 Grundlagen von CouchDB ........ 22

2 **Installation** ........ **29**
2.1 Mac OS X vorbereiten ........ 29
2.1.1 Die Entwicklungsumgebung Xcode installieren ........ 29
2.1.2 MacPorts installieren ........ 30
2.1.3 Das Terminalfenster ........ 30
2.1.4 MacPorts überprüfen ........ 31
2.1.5 CouchDB installieren ........ 32
2.1.6 CouchDB automatisch starten ........ 34
2.1.7 CouchApp installieren ........ 35
2.1.8 Überprüfen der Installation von CouchApp ........ 38
2.2 Microsoft Windows vorbereiten ........ 38
2.2.1 CouchDB installieren ........ 39
2.2.2 Überprüfen der Installation von CouchDB ........ 43
2.2.3 CouchApp installieren ........ 45
2.2.4 Überprüfen der Installation von CouchApp ........ 49
2.3 Die Entwicklungsumgebung ........ 50

3 **Erste Übungen mit CouchDB** ........ **53**
3.1 Ein einfaches Programm ........ 53
3.1.1 Vorbereitung für die erste Applikation ........ 53
3.1.2 Das Grundgerüst von Hello World! ........ 54
3.1.3 Die Anzeigefunktion für Hello World! ........ 56

3.1.4 Die CouchApp veröffentlichen ... 57
3.2 Die Arbeit mit der CouchDB ... 58
3.2.1 Prüfen, ob der CouchDB-Server läuft ... 58
3.2.2 Datenbanken der CouchDB anzeigen ... 59
3.2.3 Eine neue Datenbank anlegen ... 59
3.2.4 Löschen einer Datenbank ... 61
3.3 Futon – das Webinterface von CouchDB ... 62
3.3.1 Test der CouchDB mit Futon ... 62
3.3.2 Eine neue Datenbank mit Futon anlegen ... 63
3.3.3 Ein neues Dokument in der Datenbank speichern ... 64
3.3.4 Ein Attribut hinzufügen ... 65
3.4 Eine Abfrage durchführen ... 66
3.4.1 Eine Preisdatenbank anlegen ... 66
3.4.2 Einen einfachen View erzeugen ... 68
3.4.3 Versionierung in der CouchDB ... 72
3.5 Dokumente der CouchDB ... 73
3.5.1 Datendokumente ... 73
3.5.2 Virtuelle Dokumente ... 74
3.5.3 Design-Dokumente ... 74
3.6 Arbeiten mit Views ... 75
3.6.1 Anlage der Dokumente ... 75
3.6.2 Einen View erzeugen ... 78
3.6.3 Die View-Ergebnismenge ... 80
3.6.4 Bestimmte Daten in einer View anzeigen ... 82
3.6.5 Ein Dokument anzeigen ... 83
3.6.6 Einen Bereich von Dokumenten ermitteln ... 84
3.6.7 Die Ausgabe sortieren ... 85
3.7 Dokumente vor dem Speichern überprüfen ... 86
3.7.1 Den Dokumenttyp überprüfen ... 87
3.7.2 Das Dokument löschen ... 88
3.7.3 Pflichtfelder für ein Dokument festlegen ... 89
3.7.4 Den Autor des Dokuments prüfen ... 92
3.8 Webseiten anzeigen ... 93
3.8.1 Die Show-Funktionen ... 93
3.8.2 Statisches HTML und die CouchDB ... 95
3.8.3 AJAX mit der CouchDB ... 97
3.8.4 Formatierung in der Ausgabe ... 99

**4 Die Beispielanwendung MIT – der Métro Information Tracer ...................... 101**
4.1 Daten des MIT ... 101
4.1.1 Notwendige Dokumente ... 101
4.1.2 Informationen zu den Städten ... 102
4.1.3 Informationen zu den Linien ... 102
4.1.4 Informationen zu den Stationen ... 102
4.2 Funktionen des MIT ... 102
4.2.1 Dateneingabe im MIT ... 103
4.2.2 Darstellung der Städte im MIT ... 103
4.2.3 Darstellung der U-Bahn-Linien ... 103
4.2.4 Darstellung der Stationen ... 103
4.3 Einrichten der Datenbank ... 104
4.4 Das Grundgerüst des MIT erzeugen ... 104
4.4.1 Frameworks der CouchApp ... 106
4.4.2 Anpassen der Vendor-Daten ... 107
4.4.3 Anpassen der Anwendungsinformation ... 107
4.4.4 Ein Administrator für die CouchDB ... 108
4.4.5 Die Konfigurationsdatei von CouchApp ... 109
4.4.6 Die Startseite anpassen ... 109
4.4.7 Das Benutzerinterface aufbauen ... 114
4.5 Die Dateien für das Benutzerprofil anpassen ... 120
4.5.1 Das Layout erweitern ... 120
4.5.2 Den Link für die Anmeldung verändern ... 122
4.5.3 Das Anmelde-Widget anpassen ... 122
4.5.4 Das Registrierungs-Widget anpassen ... 123
4.5.5 Die Links für angemeldete Benutzer anpassen ... 124
4.5.6 Das Widget für nicht angemeldete Benutzer anpassen ... 125
4.5.7 Das Widget für angemeldete Benutzer anpassen ... 126
4.5.8 Das Formular für das Profil ändern ... 127
4.6 Die Städte des MIT ... 128
4.6.1 Daten mit Evently anzeigen ... 128
4.6.2 Die View für die Städteauswahl ... 130
4.6.3 Das Init-Ereignis für die Städte ... 130
4.6.4 Eine Datei mit Hilfsroutinen erstellen ... 132
4.6.5 Das Benutzerinterface optisch anpassen ... 134
4.6.6 Das Menü für die Städteauswahl erstellen ... 137
4.6.7 Die Möglichkeiten von Mustache ... 138
4.6.8 Das Widget für die Auswahl der Daten in der Seitenleiste ... 139
4.6.9 Eine neue Stadt anlegen ... 141
4.6.10 Eine Prüfroutine für das Speichern der Dokumente ... 145

4.6.11 Die Daten einer Stadt bearbeiten ........ 147
4.6.12 Eine Stadt löschen ........ 152
4.7 Die Linien des MIT ........ 154
4.7.1 Die Basis für das Verwalten der Linien ........ 154
4.7.2 Das Menü für die Auswahl einer U-Bahn-Linie ........ 156
4.7.3 Das Statistik-Modul ........ 161
4.7.4 Eine neue U-Bahn-Linie anlegen ........ 164
4.7.5 Eine U-Bahn-Linie bearbeiten ........ 168
4.7.6 Eine U-Bahn-Linie löschen ........ 179
4.8 Die Stationen des MIT ........ 180
4.8.1 Das Menü für die Auswahl der Stationen ........ 182
4.8.2 Anpassungen am MIT für die Stationen ........ 185
4.8.3 Das Statistik-Modul erweitern ........ 189
4.8.4 Den Selektor für die Auswahl der Stadt anpassen ........ 192
4.8.5 Den Selektor für die Auswahl der Linie anpassen ........ 193
4.8.6 Google Maps aktivieren ........ 195
4.8.7 Eine neue U-Bahn-Station anlegen ........ 198
4.8.8 Eine U-Bahn-Station bearbeiten ........ 210
4.8.9 Eine U-Bahn-Station löschen ........ 222
4.9 Das Benutzerinterface erweitern ........ 224
4.9.1 Das Benutzerinterface anpassen ........ 224
4.9.2 Das Interface für die Darstellung der Informationen ........ 237
4.9.3 Die Google Maps zurücksetzen ........ 240
4.10 U-Bahn-Linien darstellen ........ 241
4.10.1 Anpassungen an den vorhandenen Funktionen ........ 242
4.10.2 Eine komplette U-Bahn-Linie erstellen ........ 245
4.10.3 Den kompletten Netzplan erstellen ........ 253
4.11 Der Zustand des Métro-Netzes der RATP – Screen Scraping ........ 256
4.11.1 Einschränkungen beim Screen Scraping ........ 256
4.11.2 Die CouchApp um ein Plug-in erweitern ........ 258
4.11.3 Die Selektoren für das Screen Scraping anpassen ........ 258
4.11.4 Die Webseiten der WAP-Applikation analysieren ........ 259
4.11.5 Texte bereinigen ........ 263
4.11.6 Daten mit der Scraping-Funktion extrahieren ........ 264
4.11.7 Die Daten der Scraping-Funktion speichern ........ 268
4.11.8 Die History der Scraping-Daten anzeigen ........ 274
4.12 Daten von Flickr anzeigen ........ 278
4.12.1 Eine Funktion für den Suchtext ........ 279
4.12.2 Die API von Flickr ........ 280

4.12.3 Fotos von der Anzeige ausschließen ... 281
4.12.4 Fotos von Flickr holen und anzeigen ... 284
4.12.5 Die Benutzerdaten von Flickr holen ... 288
4.13 Zusätzliche Informationen aus Wikipedia ... 294
4.14 Abschluss des Métro Information Tracer ... 296

**A Dokumente im JSON-Format ... 299**
A.1 Städte ... 299
A.1.1 München ... 299
A.1.2 Paris ... 299
A.1.3 Wien ... 300
A.2 U-Bahn-Linien (ohne Stationen) ... 300
A.2.1 München – U1 ... 300
A.2.2 München – U2 ... 300
A.2.3 Paris – M1 ... 301
A.2.4 Paris – M2 ... 301
A.2.5 Wien – U1 ... 302
A.2.6 Wien – U2 ... 302
A.3 U-Bahn Linien (mit Stationen) ... 303
A.3.1 München – U1 ... 303
A.3.2 München – U2 ... 307
A.3.3 Paris – M1 ... 313
A.3.4 Paris – M2 ... 319
A.3.5 Wien – U1 ... 326
A.3.6 Wien – U2 ... 330

**B Glossar ... 335**

**Stichwortverzeichnis ... 345**

# 1 Die Theorie hinter NoSQL

In diesem Kapitel lernen Sie die Hintergründe von NoSQL kennen. Falls Sie direkt ins kalte Wasser springen wollen, können Sie dieses Kapitel auch überspringen. Falls Sie sich aber mit Datenbanken und den dahinter stehenden Konzepten noch nicht so gut auskennen sollten, empfehle ich Ihnen, diesen Abschnitt in Ruhe durchzuarbeiten. Denn hier werde ich einige Dinge erklären, die später verwendet werden. Damit Sie aber die einzelnen Fachbegriffe schnell und einfach finden, habe ich die entsprechenden Definitionen am Ende des Buchs in einem Glossar zusammengefasst.

Nach mehr als 15 Jahren einer fast uneingeschränkten Herrschaft der SQL-basierten Datenbanken (SQL steht für Structured Query Language, also zu Deutsch »strukturierte Abfragesprache«) findet nun ein Wechsel in der Landschaft der Web-Applikationen statt. Und genau diese neuen Sichtweisen und Techniken werden Sie hier kennenlernen. Natürlich werden nicht alle SQL-Datenbanken über Nacht verschwinden, aber bei einigen großen Unternehmen vollzieht sich der Wandel dahingehend, dass in den einzelnen Nischen die entsprechenden und idealen Datenbanksysteme (auch parallel zueinander) eingesetzt werden.

## 1.1 Die Geschichte

Der Begriff NoSQL (eine Abkürzung für *not only SQL*) wurde zum ersten Mal 1998 verwendet. Carlo Strozzi[1] entwickelte zu dieser Zeit eine leichtgewichtige Open-Source-Datenbank, die bewusst keine Zugriffsmöglichkeit unter dem Standard SQL zur Verfügung stellte. Strozzi unterscheidet aber zwischen seiner NoSQL-Datenbank und der NoSQL-Bewegung. Diese verfolgt ein Konzept, das vom relationalen Datenbankmodell abstammt.

Erst mehr als zehn Jahre später wurde der Begriff *NoSQL* bekannter. Denn Johan Oskarsson verwendete ihn Anfang 2009 bei einem Treffen zum Thema strukturierte, verteilte Datenspeicher. Er bezeichnete damit Datenbanken, die nicht relational sind, auf verteilten Systemen liegen und meistens auf ACID-Eigenschaften (Englisch für *atomicity*, *consistency*, *isolation*, *durability*) verzichten.

1 *http://www.strozzi.it*

### 1.1.1 Das Konzept NoSQL

Diese Technologie hat prinzipiell ein Ziel: die Nachteile bestehender, eingeführter Datenbanksysteme auszugleichen. Die Technik soll besser skalierbar sein und eine bessere Performance bei hoher Datenlast mit vielen umfangreichen Transaktionen bieten. NoSQL-Datenbanken haben im Gegensatz zu relationalen Datenbanksystemen kein festes Schema zur Speicherung der Daten. Dies ist eine radikale Änderung zu SQL-Datenbanken, die auf eine strenge Struktur (oder ein strenges Schema) der gespeicherten Daten achten.

NoSQL-Datenbanken können folgende Eigenschaften besitzen:

- nicht-relational
- schemafrei
- horizontal skalierbar
- BASE
- einfache Replikation

#### BASE – Basically Available, Soft State und Eventual Consistent

Da RDBMS (relationale Datenbankmanagementsysteme, also konventionelle Datenbanken) sehr streng auf die Konsistenz der Daten achten, kann es hier zu Problemen mit der Performance und Verfügbarkeit kommen. Dieses Konzept wird bei NoSQL zugunsten der besseren Skalierbarkeit und auch Verfügbarkeit aufgeweicht.

Man akzeptiert die »lose Konsistenz«. Dabei wird die Datenkonsistenz von nachfolgenden Datenoperationen immer wieder neu hergestellt. Die Datenbank wechselt dadurch immer wieder zwischen einem konsistenten und inkonsistenten Zustand. Im Gegensatz dazu müssen sich SQL-Datenbanken dauerhaft in einem konsistenten Zustand befinden. Werden durch verschiedene Datenbankoperationen Duplikate im Datenbestand angelegt, so wird die Datenbank durch eine zeitversetzte Synchronisierung immer wieder in einen konsistenten Teilzustand versetzt. Wobei der Begriff *Eventual Consistent* festlegt, dass alle Clients, die die Daten nutzen, nur in einem bestimmten Zeitfenster einen konsistenten (denselben) Datenbestand sehen.

#### CAP – Consistency, Availability und Partition Tolerance

Der Informatiker Eric Brewer vermutete im Jahr 2000, dass Systeme zum verteilten Rechnen nicht alle drei genannten Eigenschaften gleichzeitig erfüllen können. Diese Überlegung war und ist besonders wichtig für das Erstellen von Applikationen mit NoSQL-Datenbanken, denn sie sind verteilte Systeme.

- *Consistency – Konsistenz*
  Alle Clients sehen zur selben Zeit dieselben Daten

- *Availability – Verfügbarkeit*
  Ein Ausfall eines Clients (be)hindert die restlichen verfügbaren Clients nicht am Weiterarbeiten.
- *Partition Tolerance – Partitionstoleranz*
  Das verteilte System arbeitet trotz zufälliger Verluste von Nachrichten fehlerfrei weiter.

Im Jahr 2002 lieferten Seth Gilbert und Nancy Lynch einen axiomatischen Beweis, dass Brewers Vermutung richtig ist und nur zwei der drei Eigenschaften gleichzeitig erfüllt werden können. Dieses Theorem wirkt sich, wie wir später sehen werden, entscheidend auf das Design von NoSQL-Datenbanken aus.

## 1.2 Arten von NoSQL-Datenbanken

Datenbanken lassen sich in verschiedene Klassifikationssysteme einordnen. Am einfachsten gehen wir dabei vor, wenn wir die Art der Datenspeicherung betrachten. Für jede der folgenden Arten gibt es typische Vertreter und auch Anwendungsgebiete. Daher muss man sich vor der Wahl der Datenbank über das Anwendungsgebiet oder auch die zu verwaltenden Daten klar werden.

### 1.2.1 Dokumentenorientiert

Diese Datenbanken speichern Textdaten von beliebiger Größe in unstrukturierter Form. Der Zugriff auf die Daten erfolgt hier über die Dokumentinhalte. Hierzu gehören *Apache CouchDB*[2], *MongoDB*[3] oder auch *Lotus Notes*[4].

### 1.2.2 Key-Value-orientiert

Hier werden definierte Schlüssel verwendet, die auf einen bestimmten Wert verweisen. Diese Werte können aus beliebigen Zeichenfolgen bestehen. Die Key-Value-Datenbanken können entweder als *In-Memory*- oder als *On-Disk*-Version implementiert werden. Die Implementierung *In-Memory* eignet sich zum Beispiel sehr gut für Cache-Speichersysteme, da sie speicherresistent ist. Dies ist zum Beispiel das System *memcached*[5]. Die *On-Disk*-Version ist für große Datenmengen vorgesehen und wird beispielsweise von

---

2 *http://couchdb.apache.org*

3 *http://www.mongodb.org*

4 *http://www-01.ibm.com/software/de/lotus*

5 *http://memcached.org*

*Google BigTable*[6] oder *Amazon SimpleDB*[7] implementiert. Eine Kombination der beiden Versionen vereint – so gut es geht – die Vorteile beider Implementierungen miteinander. Eine Implementierung ist zum Beispiel *Redis*[8] und wird aktiv von *GitHub*[9] eingesetzt.

### 1.2.3 Spaltenorientiert

Hier werden die Daten als Schlüssel-Wert-Relation, auch Key/Value oder Tupel genannt, abgelegt. Das Hauptaugenmerk liegt hier auf der Verminderung der Ein-/Ausgabe-Aktivität bei der Berechnung der Datensätze. Ein Vertreter ist beispielsweise die Datenbank *Cassandra*[10] der *Apache Foundation*[11]. Diese Art von Datenbanken ist beispielsweise bei *Facebook*[12], *Digg*[13] oder *Twitter*[14] im Einsatz.

### 1.2.4 Graphenorientiert

Diese Datenbanken spiegeln die Beziehungen der Daten zueinander wieder. Dazu werden die Daten in einer Art Baumstruktur gespeichert. Sie eignen sich ideal zur Darstellung von Beziehungen in sozialen Netzwerken. Dabei werden die Daten als einzelne Knoten und die Beziehungen als Verbindungen zwischen diesen dargestellt. Beispiele für diese Art der Datenbank sind *FlockDB*[15] (wie sie *Twitter* verwendet), *AllegroGraph*[16] oder *Neo4j*[17].

### 1.2.5 Übersicht von NoSQL-Datenbanken

In diesem Abschnitt erhalten Sie einen – nicht vollständigen – Überblick über verschiedene NoSQL-Datenbanken, ihre Funktionen, Vor- und Nachteile sowie Einsatzgebiete.

---

6 *http://labs.google.com/papers/bigtable-osdi06.pdf*

7 *http://aws.amazon.com/de/simpledb*

8 *http://redis.io*

9 *https://github.com*

10 *http://cassandra.apache.org*

11 *http://apache.org*

12 *http://www.facebook.com*

13 *http://digg.com*

14 *http://twitter.com*

15 *https://github.com/twitter/flockdb*

16 *http://www.franz.com/agraph/allegrograph*

17 *http://neo4j.org*

| | *CouchDB* | *MongoDB* | *Redis* | *Cassandra* | *Riak* | *HBase* |
|---|---|---|---|---|---|---|
| Programmiert in | Erlang | C++ | C/C++ | Java | Erlang & C | Java |
| Lizenz | Apache[18] | AGPL[19]/ Apache | BSD[20] | Apache | Apache | Apache |
| Hauptvorteil | DB-Konsistenz & einfache Benutzung | Verwendet Abfragen und Indizierung auf SQL-Basis | Geschwindigkeit | Große Tabellen | Fehlertoleranz | Riesige Tabellenstrukturen |
| Protokoll | HTTP/REST | proprietär | Ähnlich Telnet | Proprietär und Thrift | HTTP/REST | HTTP/REST und Thrift |
| Art | Dokument-orientiert | Dokument-orientiert | In-Memory | spalten-orientiert | spalten-orientiert | Key-Value-orientiert |
| Besonderheiten | bidirektionale Replikation | Master-Slave-Replikation | Master-Slave-Replikation | Trade-offs für Verteilung und Replikation | Trade-offs für Verteilung und Replikation | Entworfen wie *Google BigTable* |
| | Konflikterkennung | Abfragen sind JavaScript-Ausdrücke | Einfache Schlüsselwerte | Trade-offs anpassbar | Trade-offs anpassbar | Vordefinierte Abfragen |
| | MVCC blockieren keine Lesezugriffe | | Komplexe Datenoperationen | | Datenüberprüfung | Optimierung von Echtzeitabfragen |
| | Versionierung der Daten | Serverseitiges JavaScript | Verwendet Sets, Lists und Hashes | Abfrage nach Spalten | Datensicherheit | Performanter Thrift-Gateway |
| | serverseitige Validierung der Dokumente | | Unterstützt Transaktionen | | | |
| | Authentifizierung | Verteilte Daten (Sharding) | Ablaufdatum für Daten | Abfrage nach Indexbereichen | Volltextsuche | XML mit HTTP-Unterstützung |
| | Echtzeit-Updates | | Sortierte Sets | | | |

[18] *http://www.apache.org/licenses*

[19] *http://www.gnu.org/licenses/agpl-3.0.html*

[20] *http://www.opensource.org/licenses/bsd-license.php*

| | *CouchDB* | *MongoDB* | *Redis* | *Cassandra* | *Riak* | *HBase* |
|---|---|---|---|---|---|---|
| | Attachment-Verwaltung | Teilweise Ablage der Daten im Speicher | Überwachen von Datenänderungen | Schreibt schneller als es liest | Commits | Wahlfreier Zugriff auf Daten |
| | jQuery-Bibliothek | | | | | |
| Nachteile | Wiederholte Komprimierung notwendig | Nach einem Crash müssen die Tabellen repariert werden | Erst Version 2.0 kann auf die Disk auslagern | Schreibt schneller als es liest | Enterprise- und Open-Source-Version | Performance des wahlfreien Zugriffs ähnlich SQL |
| | | | Datenbankgröße sollte vorhersehbar sein | Übernimmt Komplexität von Java | Verteilte Daten nur mit der Enterprise-Version | |
| Einsatzgebiete | CRM | Dynamische Abfragen | Häufige Schreibzugriffe | Echtzeit-Datenanalyse | Hochverfügbarkeit | Große Tabellen mit Milliarden Datensätzen und Millionen von Datenfeldern |
| | Wenig Schreibzugriffe | Indizierung der Daten | Schnelle Änderung der Daten | Alle Systembestandteile in Java sein sollen | Schnelle Schreibzugriffe | |
| | Häufige Lesezugriffe | Große DB mit Performance | Echtzeitverarbeitung | | Seltene Lesezugriffe | Echtzeitzugriff auf die Daten |
| | Versionierung | Ähnlich SQL ohne definierte Spalten | Statistiken | Logging | Datenanalyse | Wahlfreier Datenzugriff |

In diesem Buch wird die Entscheidung nicht direkt vom Einsatzgebiet bestimmt. Aus didaktischen Gründen, und wegen dem leichteren Handling auf einem lokalen Computersystem entscheiden wir uns für den Einsatz der *CouchDB* als Vertreter der dokumentenorientierten NoSQL-Datenbanken.

## 1.3 Grundlagen von CouchDB

*Be Relaxed!* oder *Entspann Dich!* ist der Leitspruch von *Apache CouchDB*, und genau dieses Motto müssen wir verinnerlichen. Eine der Grundlagen ist die Einfachheit, mit der *CouchDB* erstellt wurde. Die Datenbank versucht sich so weit wie möglich im Hintergrund zu halten und den Administrator nicht zu »belästigen«.

Dazu gehört, dass die interne Architektur sehr fehlertolerant ist. Einzelne Fehler beziehungsweise Ausnahmen treten in einer kontrollierten Umgebung auf und werden dort auch behandelt. Wenn üblicherweise eine Ausnahme auftritt, läuft sie durch das ganze System, wo sie an der Spitze behandelt wird. Bei *CouchDB* betrifft ein Fehler nur eine einzelne Anfrage (Request), wird dort behandelt und beeinflusst den Rest des Systems nicht.

Außerdem ist *CouchDB* auf Lastwechsel ausgerichtet. Wir kennen das Problem, dass es auf einmal viele Requests bei einer Web-Applikation geben kann. *CouchDB* reagiert zwar mit einer höheren Latenzzeit darauf, aber sie vergisst keine Anfrage, sondern beantwortet alle. Ist das hohe Lastaufkommen wieder vorbei, reagiert die Datenbank wieder mit der gewohnten Geschwindigkeit.

Normalerweise wird eine Anwendung so erstellt, dass sie bereits die maximalen Hardwareanforderungen abdeckt. Einer der offensichtlichen Nachteile dieses Ansatzes sind die hohen Investitionskosten, die am Beginn des Projekts stehen. Weiters ist das Lastaufkommen oft viel zu gering, um die Hardware auszulasten und ihre Vorteile wirklich zu nutzen.

Auch ist die Programmierung viel aufwendiger, da bereits Fälle abgebildet werden müssen, die erst in der Zukunft auftreten werden. *CouchDB* verfolgt hier einen anderen Ansatz: Es besitzt eine eingebaute Skalierbarkeit. Damit diese auch funktioniert, setzt die Datenbank dem Programmierer eindeutige Grenzen und macht nicht alles möglich, was umsetzbar wäre. Dies ist zwar etwas unflexibel, und wir müssen lieb gewonnene Gewohnheiten über Bord werfen. Aber durch das Fehlen mancher Funktionen kann der Programmierer – also Sie und ich – keine Anwendung schreiben, die der Skalierung im Wege steht oder diese unmöglich macht.

#### Daten müssen modelliert werden!

Dieser Ansatz besteht seit dem Beginn der IT und ist sicherlich richtig. Aber die Art und Weise kann sich je nach Gedankenmodell und Herangehensweise unterscheiden. In der klassischen Anwendung werden Daten im ER-Modell, kurz ERM (Entity-Relationship-Modell), dargestellt. Dies ist zwar eine sehr effektive Art, Daten für Computer aufzubereiten, aber auch eine sehr schwer skalierbare und für Menschen nicht leicht zu verstehende Art.

Im klassischen Modell – nehmen wir als Beispiel eine Rechnung – stellt sich die Datenmodellierung als Referenzmodell schon sehr aufwendig dar. Und dabei sind noch gar nicht alle Möglichkeiten berücksichtigt. Werfen Sie einen Blick auf die folgende Abbildung. Sie ist schon komplex, obwohl es nur ein einfaches, schematisches ERM ist und viele Punkte nicht berücksichtigt sind.

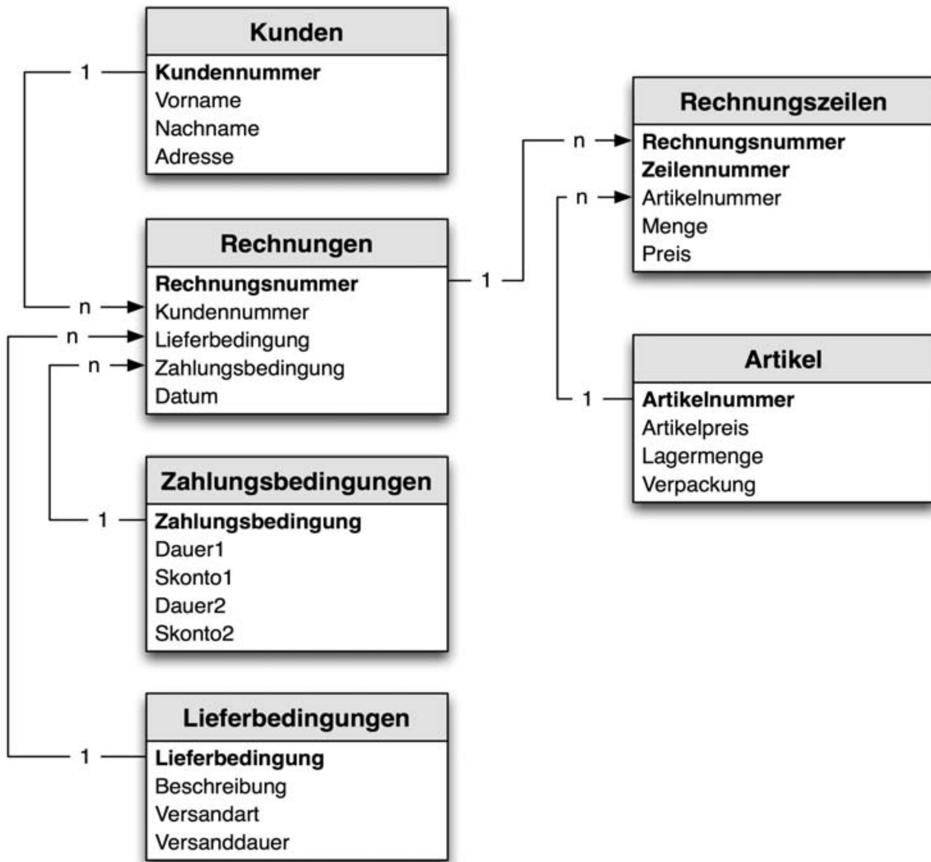


**Bild 1.1:** Daten in einem Referenzmodell

Im ER-Modell versuchen wir die Daten atomar[21], eindeutig und nach Typ definiert[22], in einer Datenbank abzulegen. Um dies zu erreichen, müssen wir alle Redundanzen vermeiden und die einzelnen Datensätze in eigene Tabellen ablegen. In unserem Beispiel sind das bereits sechs. Aber dies widerspricht unserem natürlichen Denken. Denn wir hätten gern alle Daten auf einen Blick und nicht in kleine Häppchen aufgeteilt, die wir

---

[21] Nur ein Datum pro Datenfeld. Eine Adresse – z. B. Doktorweg 13 – besteht aus zwei Daten, dem Straßennamen und der Hausnummer. In einem ERM müssten zwei Datenfelder geschaffen werden.

[22] Es wird also von Anfang an festgelegt, welcher Art die Daten sein müssen. Dies kann ein Datum, ein Text, eine Zahl oder noch etwas anderes sein. Aber hier ergeben sich bereits Probleme, denn welchen Datentyp hat beispielsweise eine Postleitzahl? In Österreich, Deutschland und der Schweiz ist es eine Zahl, aber in Großbritannien ist es eine Zeichenkette aus Buchstaben und Zahlen. Zusätzlich gibt es in Italien und Deutschland Postleitzahlen, die mit einer führenden Null geschrieben werden. Speichern wir diese jetzt als Text oder als Zahl?

später wieder zusammensuchen müssen. Außerdem hat dieses Modell noch einen weiteren Haken: Durch das schemaorientierte Modellieren müssen alle möglichen Datenfelder bereits vorhergesehen werden, späteres Hinzufügen (Skalieren) ist oft mit einem beträchtlichen Aufwand verbunden. Dieser Aufwand kann so groß sein, dass eine Anwendung komplett neu entwickelt werden muss, damit sie auf größere Systeme skaliert werden kann. Und was ist, wenn Daten nicht erfasst werden, weil sie entweder nicht vorhanden oder nicht bekannt sind? Auch dafür müssen eigene Mechanismen im Schema vorgesehen werden. Denn leere Datenfelder können eine klassische relationale Datenbank aus dem Tritt bringen.

Dies sind aber keine Gründe, relationale Datenbanken zu verteufeln und nicht mehr einzusetzen. Sie werden sehr häufig verwendet und sind auch für viele Arten der Datenverarbeitung sehr geeignet. Aber manchmal können SQL-Datenbanken die Dokumente der realen Welt nur sehr schwer abbilden. Und hier kommen dokumentenorientierte Datenbanken wie *CouchDB* ins Spiel. Diese sehen die Daten als eine Einheit und speichern und präsentieren sie auch so. Betrachten wir zum Beispiel Visitenkarten – sie kommen oft vor und wir haben alle welche auf dem Schreibtisch liegen:

**Bild 1.2:** Verschiedene Visitenkarten

Jede Visitenkarte enthält und präsentiert die für uns entscheidenden Informationen auf einen Blick. Es sind also in sich geschlossene Daten, eine der Grundlagen einer dokumentorientierten Datenbank.

Wenn wir die obigen Karten betrachten, sehen wir, dass sie alle ähnliche Daten enthalten, aber eben nicht exakt die gleichen. Denn bei Elisa Schmidt fehlt die Webseitenadresse, die Max Muster angeführt hat. Und Herr Meier hat nur eine Telefonnummer und Postanschrift, keine Mailadresse. Die fehlenden Daten werden von uns Menschen einfach ignoriert, denn wir haben das Konzept verinnerlicht. Durch einfaches Weglassen der Mailadresse gibt Herr Meier zu verstehen, dass er keine hat. Er muss nicht *E-Mail: keine vorhanden* auf die Karte schreiben.

Und hier liegt auch der grundlegende Unterschied zwischen SQL- und NoSQL-Datenbanken. Für die SQL-Variante müssen die Daten *vorher* strukturiert in ein Schema gebracht werden, sie sind daher schemaorientiert. Erst dann sind sie zu verarbeiten. Die NoSQL-Variante hingegen strukturiert die Daten erst *nach dem Speichern*, genauso wie wir Menschen. Diese Datenbanken sind schemafrei.

### Grundlegende Irrtümer!

Programmierer gehen oft von falschen Annahmen aus. Dies wurde schon vor fast 20 Jahren erkannt, wird aber heute noch oft und gern ignoriert. Die meisten Programmierer verteilter Systeme gehen von Annahmen aus, die der Hausverstand verneint. Sie wurden bekannt unter der Bezeichnung »*Fallacies of Distributed Computing*«[23].

1. Das Netzwerk ist ausfallsicher.
2. Die Latenzzeit ist gleich null.
3. Der Datendurchsatz ist unendlich.
4. Das Netzwerk ist sicher.
5. Die Netzwerktopologie wird sich nicht ändern.
6. Es gibt immer nur einen Netzwerkadministrator.
7. Die Kosten des Datentransports können mit Null angesetzt werden.
8. Das Netzwerk ist homogen.

Die *CouchDB* versucht nicht, wie andere Werkzeuge diese Punkte zu ignorieren oder zu verstecken. Im Gegensatz dazu sind im grundlegenden Konzept diese acht Punkte enthalten, und das System versucht danach zu arbeiten. Am besten sieht man das bei der Replikation der Datenbanken durch *CouchDB*.

Ein wichtiger Punkt der Skalierbarkeit ist, dass Daten an verschiedenen Orten präsent sind. Dies können verschiedene Server im selben Raum oder auch an geografisch weit auseinander liegenden Orten sein. Damit die Daten von einem Server zum anderen

---

[23] Erstmals wurden vier Trugschlüsse von William Nelson Joy und Tom Lyon veröffentlicht. 1994 wurden sie um drei weitere Punkte von Laurence Peter Deutsch erweitert, und James Gosling ergänzte etwa 1997 den achten Punkt.

kommen, müssen sie repliziert (kopiert) werden. Bei *CouchDB* passiert dies mit dem im Internet üblichen Protokoll *HTTP* und als inkrementelle Replikation. Das Protokoll ist schließlich vorhanden und funktioniert (meistens). Aber wie der erste Punkt der Irrtümer zeigt, das Netz wird ausfallen! Und das wird nach Murphys Gesetz[24] genau während der Replikation passieren. Dann ist die Datenbank bei *CouchDB* trotzdem verfügbar. Irgendwann ist eine Netzwerkverbindung wieder verfügbar, und dann beginnt das System nicht von vorne mit der gesamten Replikation, sondern setzt genau dort fort, wo die Unterbrechung stattfand.

### Die Wartezeit entscheidet!

Dies ist bei allen Anwendungen so. Sobald der Anwender warten muss, nimmt die Unzufriedenheit zu – beobachten Sie sich selbst bei der Arbeit mit lokal installierten Applikationen oder auch beim Surfen im World Wide Web. Nicht umsonst versuchen alle Webseiten, noch das letzte Quäntchen an Geschwindigkeit herauszuholen. Aber das Web hat eben keinen unendlichen Datendurchsatz und auch keine hundertprozentige Verfügbarkeit. Denken wir nur an unsere geliebten Smartphones, Tablets und anderen mobilen Geräte. Wie oft ist die Verbindung zum Anbieter nicht verfügbar? Und was ist, wenn wir im Ausland sind, wollen wir die hohen Kosten für Datenverbindungen wirklich zahlen?

Und hier ist ein anderer Ansatz der *CouchDB*: Wieso versucht man nicht, eine Skalierung nach unten vorzunehmen und eine lokale Kopie der Datenbank zu erzeugen und zu verwenden? Da die Datenbank in der Sprache *Erlang*[25] geschrieben ist, kann sie auch auf sehr einfachen Geräten mit geringer Hardwareausstattung laufen. Und warum nicht eine Datenbank auf einem Smartphone installieren, die sich – sobald eine Datenleitung verfügbar ist – automatisch mit anderen Instanzen synchronisiert? Dieser Ansatz bietet den Vorteil einer geringen Latenzzeit, denn eine lokal vorhandene Datenbank antwortet in Tausendstelsekunden. Zusätzlich wird das Problem des fehlenden Mobilfunknetzes umgangen. Denn die Daten sind jetzt offline verfügbar.

### Mit oder gegen die CouchDB arbeiten?

In der Welt der Programmierung hat sich eine Phrase eingebürgert: »Wir programmieren *gegen* eine Schnittstelle/Datenbank/API.« Und dieser Satz drückt auch die grundlegende Denkweise der Programmierung aus. Wir arbeiten gegen etwas und versuchen es in unsere Anforderung zu zwingen. Die Philosophie der *CouchDB* ist aber, *mit* der Datenbank zu arbeiten und nicht Dinge zu erzwingen. So erhalten wir automatisch einfache, skalierbare und verteilte Systeme.

---

[24] »Whatever can go wrong, will go wrong.« (»Alles, was schiefgehen kann, wird auch schiefgehen.«)

[25] Erlang wurde ursprünglich im Jahr 1987 für die Programmierung in der Telekommunikation (Vermittlungsstellen von Telefonnetzen) geschaffen. Für weitere Informationen siehe *http://erlang.org*

Bei der Arbeit mit der Datenbank müssen Sie aber einen der grundlegenden Ansätze von SQL über Bord werfen: Daten sind immer konsistent! Bei der *CouchDB* lautet der Ansatz: Die Daten sind am Ende irgendwann konsistent![26] Dies ist ein entscheidender Ansatz beim Hinzufügen von mehreren Datenbankservern zu einem bestehenden System. Denn hier müssen Entscheidungen über die Konsistenz der Daten und auch der Verfügbarkeit getroffen werden. Und hier kommt das erwähnte CAP-Theorem ins Spiel. Dabei legt die *CouchDB* das Hauptaugenmerk auf *Verfügbarkeit* und *Partitionstoleranz* und stellt die *Konsistenz* hintan.

[26] Eventual Consistency

# 3 Erste Übungen mit CouchDB

In diesem Kapitel machen Sie die ersten Gehversuche mit der *CouchDB* und auch mit der Programmierung. Es ist sicher von Vorteil, wenn Sie zuerst kleinere Programme und Übungen ausprobieren, bevor Sie sich an die große Applikation, die im folgenden Kapitel dargestellt wird, wagen.

Zusätzlich werde ich zwischen den einzelnen Übungen die theoretischen Hintergründe und auch den Ansatz für eine andere Art der Programmierung von NoSQL-Datenbanken erläutern.

## 3.1 Ein einfaches Programm

Das erste Programm in einer neu zu lernenden Programmiersprache ist immer das berühmte »Hello World!«. Wir wollen an dieser Tradition festhalten und diese Applikation als Erstes erstellen.

### 3.1.1 Vorbereitung für die erste Applikation

Wir öffnen das *Terminal* (*Mac OS X*) beziehungsweise die *Eingabeaufforderung* (*Microsoft Windows*). Normalerweise befinden wir uns in unserem Home-Verzeichnis und können hier für den Beginn auch unsere Applikation erstellen. Wir werden jetzt einige Befehle eingeben. Bestätigen Sie jede einzelne Eingabe mit der Taste `Return`.

In *Microsoft Windows* sehen wir sofort an der Eingabeaufforderung, welches Verzeichnis wir benutzen. In *Mac OS X* geben wir den Befehl `pwd` ein und erhalten den aktuellen Verzeichnisnamen als Antwort. Dieser sollte dem folgenden Bild entsprechen, mit dem Unterschied, dass Ihr Benutzername aufgeführt ist.

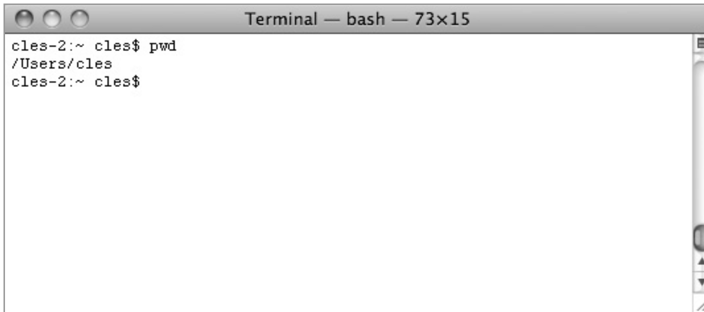


**Bild 3.1:** Ausgabe des Verzeichnisnamens

Nun legen Sie für die ersten Tests ein Verzeichnis mit dem Namen *cApps* an. In *Mac OS X* verwenden Sie dazu den Befehl `mkdir cApps` und in *Microsoft Windows* den Befehl `md cApps`. Jetzt können Sie mit dem Befehl `cd cApps` in das neu angelegte Verzeichnis wechseln.

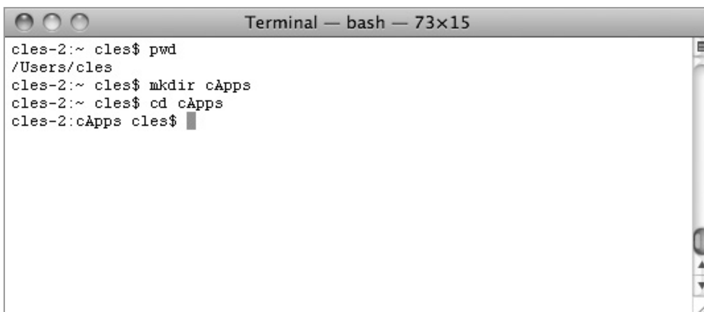


**Bild 3.2:** Verzeichnis erstellen

Damit haben wir die Vorbereitungen abgeschlossen und können die Applikation erstellen.

### 3.1.2 Das Grundgerüst von Hello World!

Die *CouchApp* stellt für das Anlegen eines Grundgerüsts einer *CouchApp* den Befehl `generate` zur Verfügung. Diesen geben Sie jetzt ein:

```
couchapp generate helloworld.
```

Nachdem Sie den Befehl mit Return bestätigt haben, wird ein Verzeichnis mit den notwendigen Dateien angelegt. Wie Sie sehen, wird zuerst die *CouchApp* aufgerufen, danach folgt der auszuführende Befehl. In unserem Fall ist dies `generate` und danach der Name der Applikation, hier ist es `helloworld`.

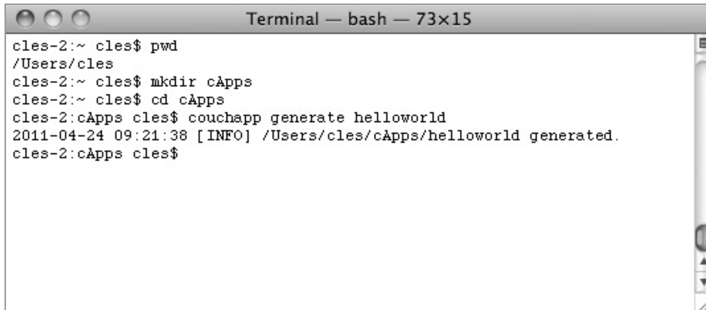


**Bild 3.3:** Generieren des Grundgerüsts für die CouchApp

Damit erzeugt die *CouchApp* ein komplettes Gerüst für eine fertige Stand-alone-Anwendung in verschiedensten Verzeichnissen.

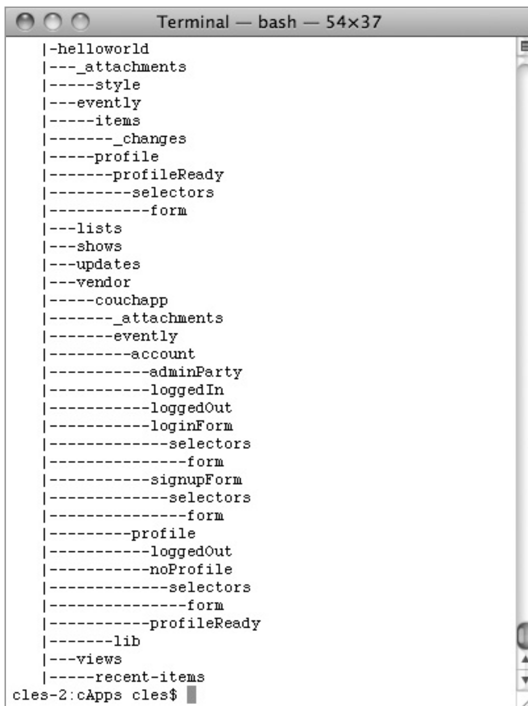


**Bild 3.4:** Verzeichnisstruktur einer CouchApp

Fürs Erste können wir diese Struktur ignorieren. Aber wir werden sie bei der kompletten Anwendung noch näher kennenlernen. Nun können Sie mit `cd helloworld` in das Verzeichnis der neuen *CouchApp* wechseln und mit dem Befehl `ls` (*Mac OS X*) beziehungsweise `dir` (*Microsoft Windows*) den Inhalt des Verzeichnisses anzeigen.

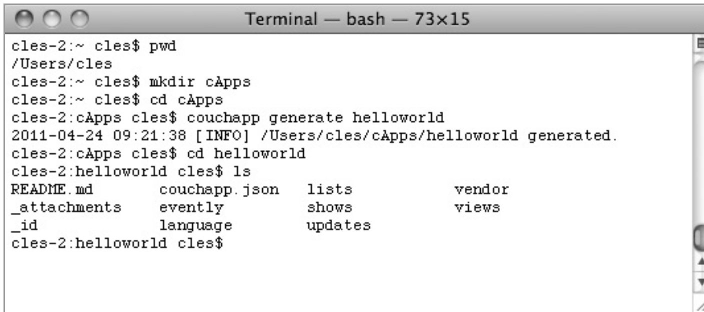


**Bild 3.5:** Inhalt des Verzeichnisses ./helloworld

### 3.1.3 Die Anzeigefunktion für Hello World!

Nun müssen wir eine Funktion zur Anzeige des Textes erstellen. Auch dafür stellt uns *CouchApp* einen Befehl zur Verfügung. Mithilfe des bereits bekannten `generate` erzeugen wir jetzt eine `show`-Funktion mit dem Namen `hello`. Wir geben als Befehl ein:

```
couchapp generate show hello
```

Wenn wir den Verzeichnisinhalt von *./helloworld* anzeigen, sehen wir, dass *CouchApp* im Unterordner *./helloworld/shows* eine Datei mit dem Namen *hello.js* erzeugt hat.

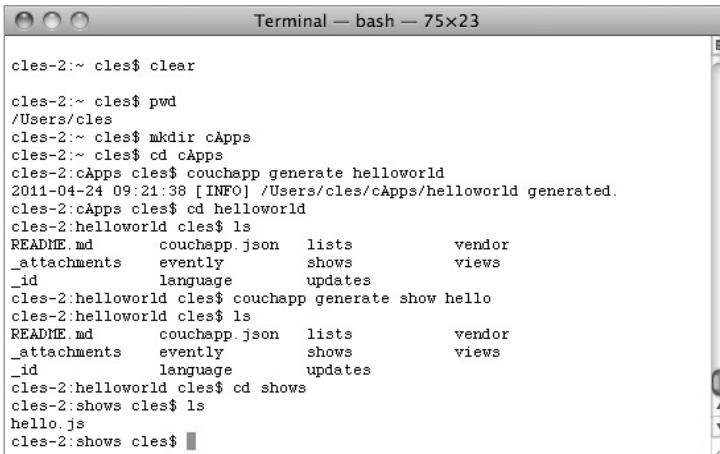


**Bild 3.6:** Inhalt des Unterordners ./helloworld/shows/

Diese Datei können wir jetzt in einem beliebigen Editor öffnen und nach unseren Wünschen ergänzen. Der Inhalt der Datei ist noch sehr einfach, denn er enthält nur das Grundgerüst der Funktion.

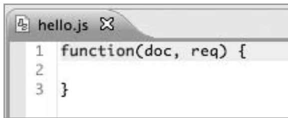


**Bild 3.7:** Grundgerüst einer Funktion

Nun fügen wir zwischen die beiden geschwungenen Klammern einen `return`-Befehl und den passenden Text ein und speichern die JavaScript-Datei.

```
function(doc, req) {
   return "Hello World!";
}
```

### 3.1.4 Die CouchApp veröffentlichen

Nachdem wir unser einfaches Programm erzeugt haben, müssen wir es in der *CouchDB* speichern. Dazu machen wir uns zuerst wieder bewusst, dass diese Art von Datenbank einerseits ein eigener Server ist und andererseits alle Arten von Dokumenten unstrukturiert speichern kann.

Nun benötigen wir die Adresse des (lokalen) Servers, in dem wir das Dokument ablegen wollen. Diese lautet, wenn alles mit den Standardvorgaben installiert wurde, *http://127.0.0.1:5984*. Wie Sie sehen, ist es eine einfache URL, die sich aus der IP-Adresse des lokalen Hosts und einer Portnummer (`5984`) zusammensetzt.

Zusätzlich benötigen wir einen Namen für die Datenbank, in der das Programm abgelegt werden soll. Für den Anfang verwenden wir aus Gründen der Einfachheit den Namen *testDB*.

Nun können wir mit dem Befehl `push` der *CouchApp* das Programm in der Datenbank speichern. Wir geben also in der Kommandozeile ein:

```
couchapp push testdb
```

Wichtig ist hier, dass Sie sich im Verzeichnis *./helloworld* befinden, sonst funktioniert es nicht richtig, da *CouchApp* die Applikation nicht findet. Jetzt sollten Sie folgende Ausgabe vor sich auf dem Bildschirm sehen.

**Bild 3.8:** Publizieren/ Speichern von Hello World

Natürlich können Sie das Resultat auch im Browser betrachten. Dazu geben Sie im Adressfeld die URL *http://127.0.0.1:5984/testdb/_design/helloworld/_show/hello* ein. Sie sehen dann – wie erwartet – die Ausgabe, wie in der folgenden Abbildung dargestellt:

**Bild 3.9:** Ausgabe des Programms im Browser

Damit haben wir unsere erste Applikation mit *CouchDB* und *CouchApp* erstellt. Nach diesem ersten Erfolg lassen wir die *CouchApp* ruhen und beschäftigen uns eingehender mit der Basis, der *CouchDB*.

## 3.2 Die Arbeit mit der CouchDB

Für unsere erste Applikation haben wir bereits verschiedene (Zusatz-)Anwendungen der *CouchDB* benutzt. Nun schauen wir uns die Arbeitsweise der Datenbank im Detail an und werden einige zusätzliche Informationen erhalten. Dazu brauchen Sie wieder ein *Terminalfenster* (*Mac OS X*) beziehungsweise die *Eingabeaufforderung* (*Microsoft Windows*).

### 3.2.1 Prüfen, ob der CouchDB-Server läuft

Als Erstes müssen Sie sich wieder als Administrator anmelden. Danach können Sie mit `curl` die Funktion des Servers überprüfen. Geben Sie dazu

```
curl http://127.0.0.1:5984
```

in der Kommandozeile ein.

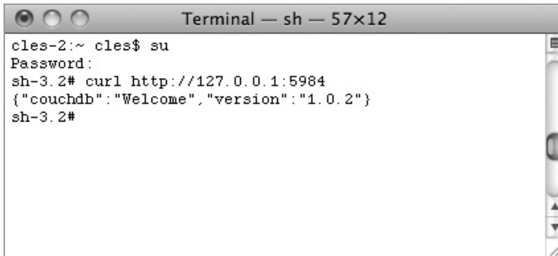


**Bild 3.10:** Den CouchDB-Server mit curl testen

Die Antwort ist an und für sich nicht sehr aufregend. Der Server begrüßt uns und gibt seine Versionsnummer bekannt, aber er läuft, und das ist uns wichtig.

### 3.2.2 Datenbanken der CouchDB anzeigen

Als Nächstes lassen Sie sich die vorhandenen Datenbanken mit der Eingabe von

```
curl http://127.0.0.1:5984/_all_dbs
```

anzeigen. Der Befehl `_all_dbs` weist den Server an, alle Datenbanken, die er gespeichert hat, aufzulisten. Gleichzeitig sehen Sie hier eine der grundlegenden Funktionsweisen. *CouchDB* ist nicht nur ein Datenbankserver, sondern auch ein Webserver und liefert alle Antworten in dem bekannten JSON-Format. Dies wird später bei der Erstellung der Anwendung sehr nützlich sein.

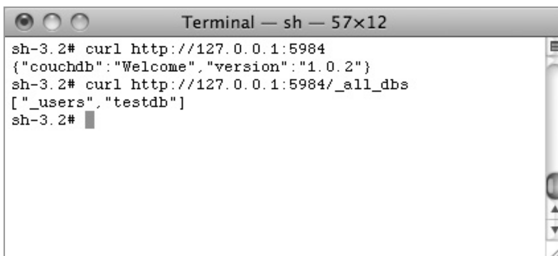


**Bild 3.11:** Vorhandene Datenbanken anzeigen

Wie Sie sehen, sind eine Standarddatenbank namens *_users* und die von Ihnen angelegte *testdb* vorhanden.

### 3.2.3 Eine neue Datenbank anlegen

Nun können Sie auch eine neue Datenbank anlegen. Bis jetzt haben wir `curl` immer mit einer GET-Anfrage verwendet. Dies ist der Standard, um Daten von einem Webserver zu erhalten. Diesmal müssen wir einen zusätzlichen Parameter hinzufügen, damit `curl` Daten zum Webserver sendet. Dies ist der Befehl PUT des HTTP-Protokolls.

Mit dem Befehl

```
curl -X PUT http://127.0.0.1:5984/kreativ
```

legen Sie eine neue Datenbank mit dem Namen *kreativ* an. Sie sehen, dass wir den Datenbanknamen nur an das Ende der URL anhängen und den Befehl absenden müssen. Schon antwortet die *CouchDB* mit einem OK.

```
Terminal — sh — 57×12
sh-3.2# curl http://127.0.0.1:5984
{"couchdb":"Welcome","version":"1.0.2"}
sh-3.2# curl http://127.0.0.1:5984/_all_dbs
["_users","testdb"]
sh-3.2# curl -X PUT http://127.0.0.1:5984/kreativ
{"ok":true}
sh-3.2#
```

**Bild 3.12:** Eine Datenbank anlegen

Nun können Sie die Liste der Datenbanken erneut abfragen und erhalten diesmal drei Datenbanken.

```
Terminal — sh — 57×12
sh-3.2# curl http://127.0.0.1:5984
{"couchdb":"Welcome","version":"1.0.2"}
sh-3.2# curl http://127.0.0.1:5984/_all_dbs
["_users","testdb"]
sh-3.2# curl -X PUT http://127.0.0.1:5984/kreativ
{"ok":true}
sh-3.2# curl http://127.0.0.1:5984/_all_dbs
["_users","kreativ","testdb"]
sh-3.2#
```

**Bild 3.13:** Liste der Datenbanken anzeigen

Wenn Sie jetzt versuchen, die Datenbank *kreativ* nochmals anzulegen, wird *CouchDB* mit einer Fehlermeldung reagieren, denn die Datenbank existiert bereits.

```
Terminal — sh — 57×12
sh-3.2# curl http://127.0.0.1:5984
{"couchdb":"Welcome","version":"1.0.2"}
sh-3.2# curl http://127.0.0.1:5984/_all_dbs
["_users","testdb"]
sh-3.2# curl -X PUT http://127.0.0.1:5984/kreativ
{"ok":true}
sh-3.2# curl http://127.0.0.1:5984/_all_dbs
["_users","kreativ","testdb"]
sh-3.2# curl -X PUT http://127.0.0.1:5984/kreativ
{"error":"file_exists","reason":"The database could not b
e created, the file already exists."}
sh-3.2#
```

**Bild 3.14:** Fehlermeldung beim Anlegen einer Datenbank

### 3.2.4 Löschen einer Datenbank

Da wir die Datenbank *kreativ* nur zur Übung angelegt haben, können Sie diese auch wieder löschen. Dazu verwenden wir wieder den Befehl `curl` und diesmal den HTTP-Befehl DELETE. Geben Sie

```
curl -X DELETE http://127.0.0.1:5984/kreativ
```

in der Kommandozeile ein, um die Datenbank zu löschen. Auch hier bestätigt *CouchDB* die erfolgreiche Ausführung des Befehls mit einem OK.

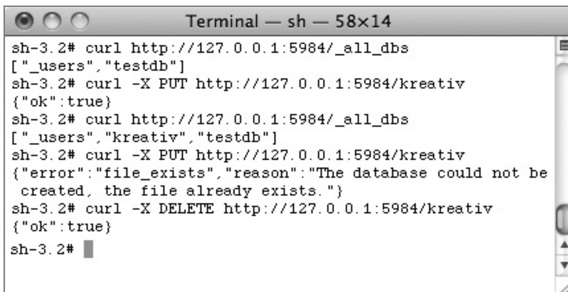


**Bild 3.15:** Löschen einer Datenbank

Sobald Sie sich die Liste der vorhandenen Datenbanken anzeigen lassen, sehen Sie, dass nur mehr zwei Datenbanken vorhanden sind.

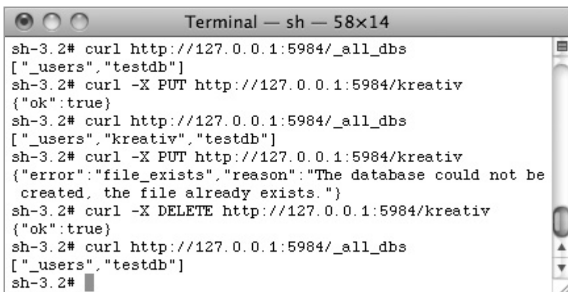


**Bild 3.16:** Liste der Datenbanken nach dem Löschen

Sie sehen, dass das Arbeiten mit der *CouchDB* eigentlich sehr einfach ist. Es setzt sich aus einfachen GET-, POST-, PUT- und DELETE-Befehlen des HTTP-Protokolls zusammen. Und die Antworten erfolgen immer in dem einfachen und im Web sehr verbreiteten JSON-Format.

## 3.3 Futon – das Webinterface von CouchDB

Aber auf Dauer ist dies sicherlich nicht sehr praktikabel. Denn eine Anwendung nur über die Befehlszeile zu erstellen, nimmt doch einige Zeit in Anspruch. Und genau hier kommt das erste Tool zur Anwendung: *Futon.*

Auch mit dem integrierten Administrationsinterface lassen sich alle diese Dinge erledigen. Am besten werfen wir einen Blick darauf. Öffnen Sie Ihren Browser und geben als URL folgendes ein:

```
http://127.0.0.1:5984/_utils
```

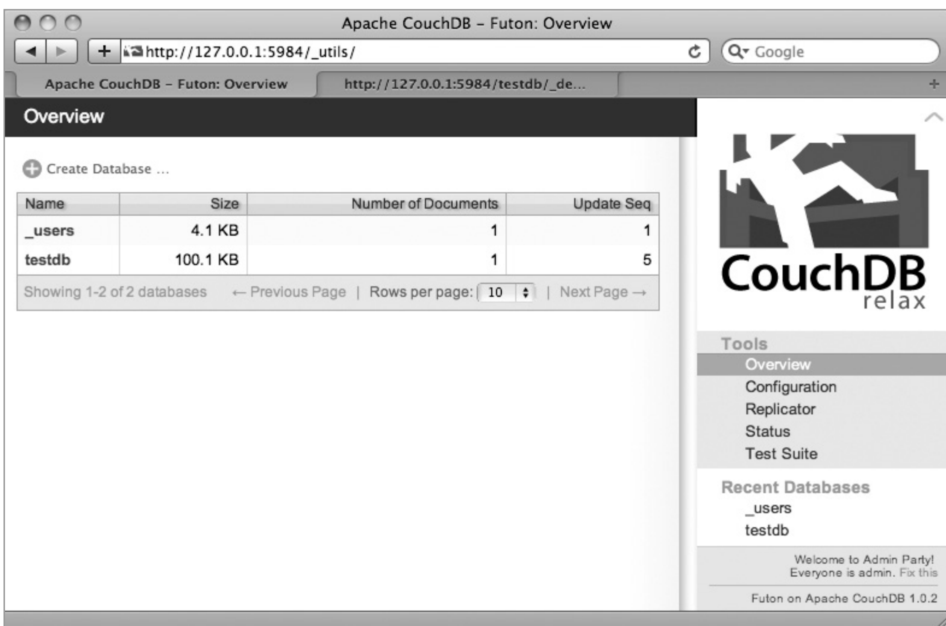


**Bild 3.17:** Übersichtsseite von Futon

### 3.3.1 Test der CouchDB mit Futon

Einer der ersten Schritte nach der Installation der *CouchDB* sind umfangreiche Tests. Dafür ist in *Futon* die TESTSUITE integriert, die rechts im Menü TOOLS aufrufbar ist.

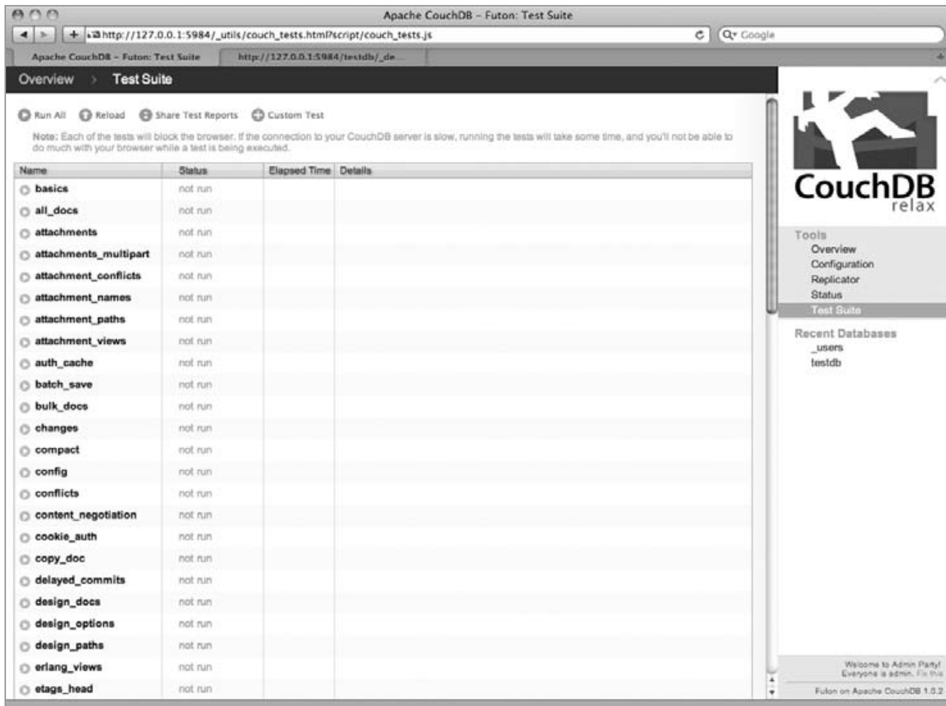


**Bild 3.18:** Testsuite von Futon

Um alle Tests durchzuführen, klicken Sie auf den Befehl RUN ALL links oben im Fenster. Während der Tests zeigt *Futon* den Fortlauf der einzelnen Schritte an. Zusätzlich werden auch alle Browser-Funktionalitäten überprüft. Dadurch sehen Sie schnell, ob eine Firewall oder eine Proxy-Konfiguration die Arbeit der *CouchDB* behindert.

### 3.3.2 Eine neue Datenbank mit Futon anlegen

Dazu finden Sie in der Übersicht (OVERVIEW) von *Futon* einen Menüpunkt CREATE DATABASE… Sobald Sie diesen anklicken, öffnet sich ein Fenster, und Sie können den Namen der neuen Datenbank eingeben. Achten Sie bitte auf den Hinweis zu den erlaubten Zeichen für einen Datenbanknamen. Insbesondere sind Großbuchstaben nicht erlaubt. Dies ist ein häufiger Fehler, wenn man schon lange programmiert und an eine bestimmte Schreibweise gewöhnt ist.

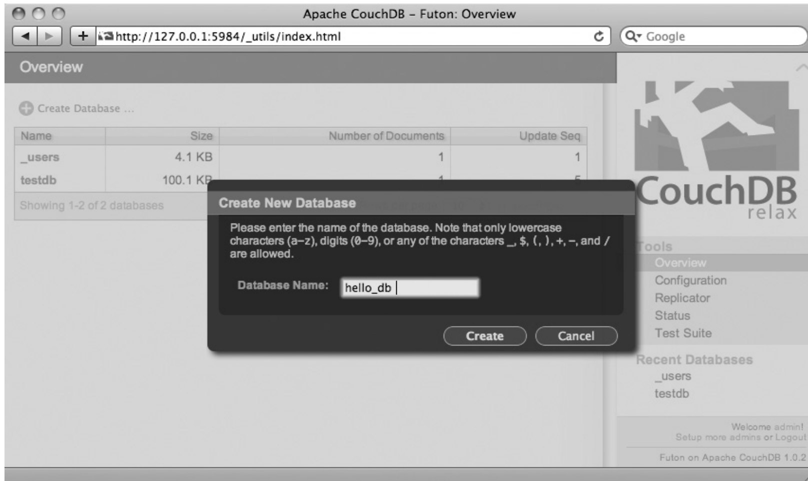


**Bild 3.19:** Datenbank mit Futon anlegen

Mit einem Klick auf CREATE wird die Datenbank erzeugt und in *Futon* angezeigt. Natürlich ist die Liste der vorhandenen Dokumente bei einer jungfräulichen Datenbank noch leer.

### 3.3.3 Ein neues Dokument in der Datenbank speichern

Dies können Sie mit dem Befehl NEW DOCUMENT im oberen Bereich von *Futon* durchführen.

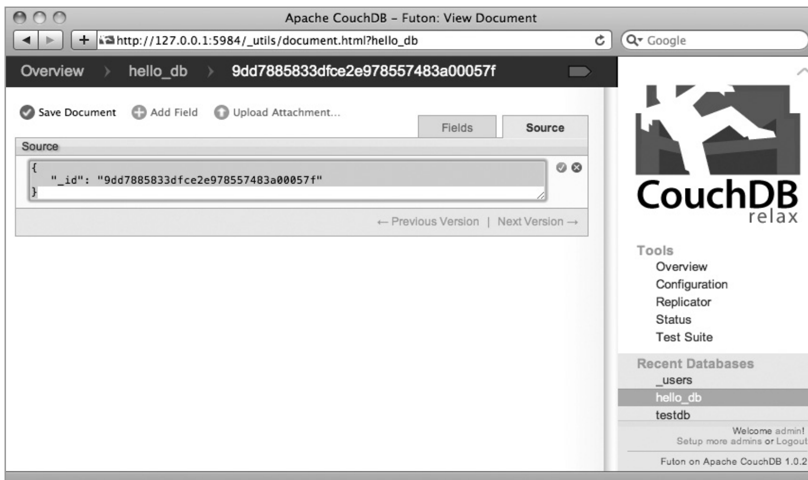


**Bild 3.20:** Neues Dokument in Futon anlegen

Das Datenfeld `_id` wurde von *Futon* bereits mit einer eindeutigen Nummer (UUID) versehen. Sie müssen daher nur noch auf den Befehl SAVE DOCUMENT klicken. Nun sehen Sie das neue Dokument, das die beiden Eigenschaften (Attribute) `_id` und `_rev` enthält.

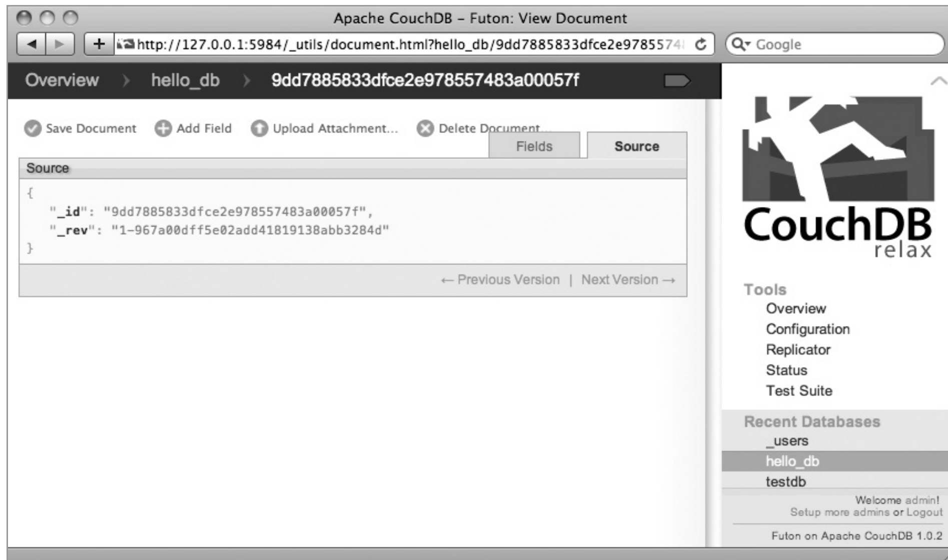


**Bild 3.21:** Ein neues Dokument in Futon

Da es sich bei *Futon* um eine typische Web-Applikation handelt, verhält sie sich auch so. Daher können Sie direkt im Browserfenster eine Eigenschaft doppelt anklicken, um sie zu verändern. Probieren Sie es mit dem Datenfeld `_id` aus, und geben Sie dem Dokument einen verständlicheren Namen.

## 3.3.4 Ein Attribut hinzufügen

Um dem Dokument eine weitere Eigenschaft hinzuzufügen, können Sie den Befehl ADD FIELD verwenden. Nun können Sie sofort den Namen der Eigenschaft (in unserem Fall `hello`) und danach den Wert festlegen (hier ist es `"world"`).

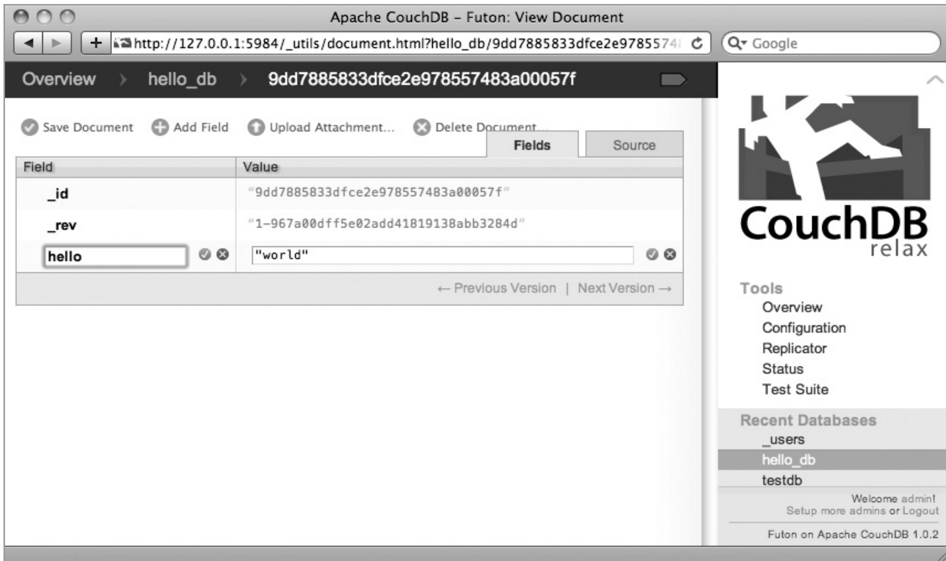


**Bild 3.22:** Ein neues Attribut hinzufügen

Um die Änderungen zu übernehmen, müssen Sie wieder den Befehl SAVE DOCUMENT anklicken. Bevor Sie dies jedoch tun, werfen Sie einen Blick auf die Eigenschaft `_rev`, und vergleichen Sie den Wert, den Sie jetzt und nach dem Speichern sehen. Ihnen wird auffallen, dass sich die Werte unterscheiden. Genau das soll auch der Fall sein, denn dies ist die interne Versionierung der *CouchDB*. Vor dem Bindestrich sehen Sie die aktuelle Versionsnummer. Sie wird bei jedem Speichervorgang automatisch um 1 erhöht.

## 3.4 Eine Abfrage durchführen

Nun sind Sie schon ein wenig mit *CouchDB* und *Futon* vertraut. Um Sie zum nächsten Schritt, der Abfrage der Datenbank, bringen zu können, benötigen wir einige Daten, die wir jetzt anlegen werden.

### 3.4.1 Eine Preisdatenbank anlegen

Verwenden wir doch die *CouchDB* dazu, um unser Fernweh ein wenig zu schüren und speichern die Preise für Flugtickets von Ihrem Heimatort nach Paris, Rom und Barcelona für je drei Fluglinien. Um dies in der *CouchDB* zu speichern, brauchen Sie das JSON-Format. Die Attribute `_id` und `_rev` und deren Werte überlassen wir der Datenbank, aber die restlichen Eigenschaften müssen Sie festlegen.

Zuerst legen Sie ein neues Dokument in der bestehenden Datenbank an. Danach fügen Sie die beiden Eigenschaften `destination` und `tickets` mit ADD FIELD hinzu. Achten Sie beim zweiten Attribut auf die Notation nach JSON.

```
{
  "_id": "9dd7885833dfce2e978557483a001575",
  "_rev": "1-967a00dff5e02add41819138abb3284d",
  "destination": "Paris",
  "tickets": {
"Air France": 133.5,
"Lufthansa":  149.7,
"Air Berlin": 139.9
  }
}
```

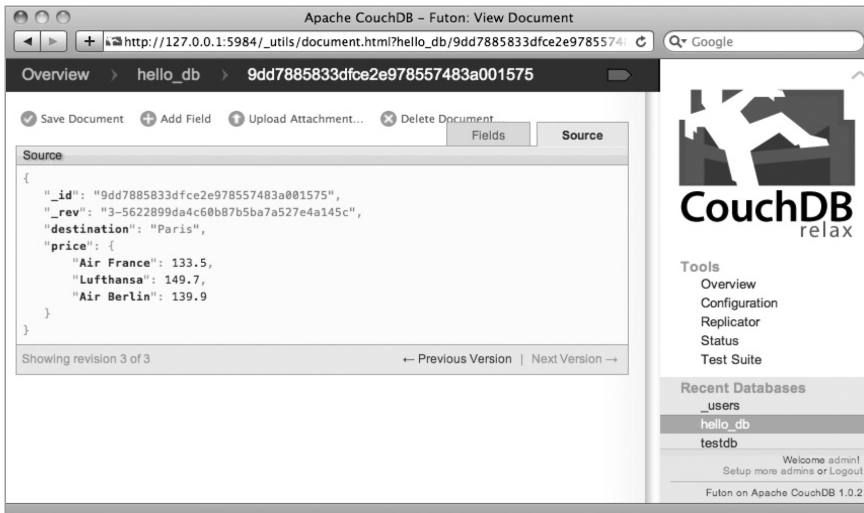


**Bild 3.23:** Das Dokument für die Ticketpreise nach Paris

Dasselbe erledigen Sie in einem neuen Dokument für Rom.

```
{
  "_id": "9dd7885833dfce2e978557483a001588",
  "_rev": "1-e34479a652eb8ee2a7a65d01d062cf43",
  "destination": "Rom",
  "tickets": {
"Air France": 123.1,
"Lufthansa":  119.7,
"Air Berlin": 99.9
  }
}
```

Und für die Destination Barcelona in einem dritten Dokument.

```
{
  "_id": "9dd7885833dfce2e978557483a0019a5",
  "destination": "Barcelona",
  "tickets": {
"Air France": 144.6,
"Lufthansa":  139.3,
"Air Berlin": 169.9
  }
}
```

Nun haben Sie in der Datenbank *hello_db* vier Dokumente gespeichert.

**Bild 3.24:** Dokumente in der Datenbank hello_db

### 3.4.2 Einen einfachen View erzeugen

Hier stoßen wir auf einen weiteren neuen Gedanke bei dokumentorientierten Datenbanken. Bei RDBMS ist es so, dass wir jederzeit – solange das Schema ordentlich aufgebaut ist – eine Abfrage gegen die Datenbank durchführen können.

Im Gegensatz dazu steht die Arbeitsweise mit *Map/Reduce*. Diese beiden Funktionen stellen eine größere Flexibilität zur Verfügung, was bei schemafreien Datenbanken eine absolute Notwendigkeit ist. Denn die Funktionen *Map* und *Reduce* können mit verschiedenen Varianten der Datenstruktur umgehen. Zusätzlich können Sie auch Indizes

für Dokumente berechnen. Dies geschieht – um die Performance zu verbessern – unabhängig und parallel.

In relationalen Datenbanken werden Abfragen zum Bestimmen der Zeilen einer Tabelle verwendet, die sich als *ResultSet* abbilden. Abfragen mit *Reduce*-Funktionen verwenden hingegen Bereiche von Indizes, die zuvor mit einer *Map*-Funktion erstellt wurden.

Eine *Map*-Funktion wird exakt einmal für ein Dokument aufgerufen und liefert normalerweise eine Zeile oder auch mehrere Zeilen mit jeweils einem Attribut/Werte-Paar zurück. Die Funktion muss in sich abgeschlossen sein und darf nur auf Daten eines Dokuments zugreifen. Dadurch ist es der *CouchDB* möglich, *Views* (ein anderer Name für eine *Map/Reduce*-Funktion) parallel und inkrementell zu erstellen. Denn jede Funktion ist von anderen Dokumenten und auch globalen Variablen komplett unabhängig.

Die Resultate der *Views* werden als Zeilen, die nach dem Schlüssel (Attribut) sortiert sind, gespeichert. Dadurch ist ein effizienter Zugriff auf die Werte möglich, auch bei sehr großen Datenmengen. Damit ist auch das Ziel einer *Map*-Funktion klar: Wir wollen einen Index erzeugen, der alle notwendigen Daten unter einem Attribut zusammenfasst.

Nehmen wir an, dass unser Fernweh sehr groß, aber unser Budget sehr klein ist. Daher benötigen wir die Destinationen nach Preis sortiert. Um diese Information zu erhalten, müssen wir einen *View* erstellen. Klicken Sie zuerst auf den Namen der Datenbank *hello_db* in der schwarzen Leiste des Browserfensters. Damit werden alle Dokumente aufgelistet und auch eine Auswahlliste mit dem Titel VIEW: am oberen Rand angezeigt. Wählen Sie aus dieser Liste den Befehl TEMPORARY VIEW… aus.

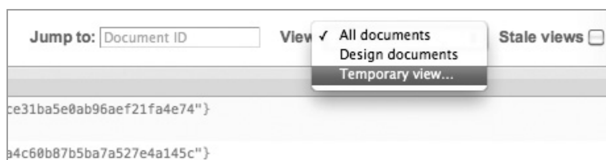


**Bild 3.25:** Eine temporäre View erstellen

Die Warnung, die Sie am unteren Rand des Browserfensters sehen, sollten Sie beherzigen. Diese Art der *Views* ist nur für Experimente und für das Ausprobieren von Abfragen gedacht. In einer Produktivumgebung sollten Sie mit permanenten *Views* arbeiten. Nun können Sie die folgende Funktion im Fenster MAP: erfassen (mit den *Reduce*-Funktionen werden wir uns erst später beschäftigen).

```
function(doc) {
  var carrier, ticket, value;
  if (doc.destination && doc.tickets) {
for (carrier in doc.tickets) {
  ticket = doc.tickets[carrier];
  value = [doc.destination, carrier];
```

```
  emit (ticket, value);
}
  }
}
```

Und sobald Sie auf RUN klicken, erhalten Sie eine Liste der Tickets.

Bild 3.26: Temporäre View mit der Liste der Flugtickets

Wenn Sie ein wenig mit JavaScript vertraut sind, ist Ihnen die Funktion, die wir erstellt haben, sicher nicht fremd. Trotzdem schauen wir sie uns genauer an.

Zuerst erzeugen wir eine Funktion für die Arbeit mit Dokumenten. In der folgenden Zeile definieren wir drei Variablen, die wir später verwenden werden. In der dritten Zeile prüfen wir die Dokumente auf ihren Inhalt. Da wir mit einer schemafreien Struktur arbeiten, können in einer Datenbank alle möglichen Daten vorkommen. In unserem Fall ist das erste Dokument – das *Hello-World*-Dokument vom Anfang dieses Kapitels –

nicht brauchbar. Alle anderen enthalten ein Attribut `destination` und `tickets`. Und die werden damit auch zum Indizieren verwendet.

Falls also ein Dokument die Anforderungen erfüllt, iterieren wir in der fünften Zeile über die `tickets` und lesen die entsprechenden Werte in der sechsten und siebten Zeile aus und übergeben sie je einer Variablen. Die achte Zeile verwendet den Befehl `emit()`, um die Werte auszugeben.

Die einzelnen Zeilen einer *View* werden immer nach dem Schlüssel sortiert, um effiziente Zugriffe zu ermöglichen. In unserem Fall ist der Schlüssel der Preis. Dies ist auch erwünscht, da uns die Destination gleichgültig war. Wir wollten nur das billigste Ticket erhalten.

Da *CouchDB* so flexibel ist, dass ein Schlüssel auch ein Array sein kann, können wir unsere *Map*-Funktion so ändern, dass zuerst nach dem Carrier und erst dann nach dem Preis sortiert wird.

```
function(doc) {
  var carrier, ticket, key;
  if (doc.destination && doc.tickets) {
for (carrier in doc.tickets) {
  ticket = doc.tickets[carrier];
  key = [carrier, ticket];
  emit (key, doc.destination);
}
  }
}
```

Nachdem die Funktion mit RUN ausgeführt wurde, sehen Sie das folgende Ergebnis.

### 4.12.1 Eine Funktion für den Suchtext

Da wir nicht nur *Flickr*, sondern auch *Wikipedia*[44] verwenden werden, ist es sinnvoll, sich Gedanken zu machen, wie von diesen Diensten Daten abgerufen werden können. Da der sogenannte Suchstring für beide Dienste ähnlich aufgebaut ist und auch noch die Daten von der Datenbank des Métro Information Tracer angeliefert werden, können wir hier eine allgemeine Funktion erstellen.

Dazu legen wir in der Datei *mit.helper.js* die Funktion `getSearch()` an.

```
/*****
 * Suchstring fuer Flickr und Wikipedia zusammenstellen
 * repWhiteSpace string
 *   fuer Leerzeichen zu verwendendes Sonderzeichen
 *   default: +
 *   URI: %20
 *   no: keine Ersetzung durchfuehren
 * return string
 *   Suchstring
 *****/
function getSearch(repWhiteSpace) {
  //Falls kein Parameter angegeben wurde, Standard setzen
  if (!repWhiteSpace || repWhiteSpace == "") {
    repWhiteSpace = "+";
  }
  //URI-encoded Whitespace
  if (repWhiteSpace == "URI") {
    repWhiteSpace = "%20";
  }
  //Je nach Stadt den passenden Suchbegriff setzen
  if ($("#city-sel").html() == "Paris") {
    search = "metro";
  } else {
    search = "ubahn";
  }
  if ($("#city-sel").html() != "") {
    //Falls eine Stadt aktiv ist,
    //diese zum Suchtext hinzufuegen
    search = search + " " + $("#city-sel").html();
  } else {
    //Falls keine Stadt ausgewaehlt ist,
    //die Funktion verlassen
    return;
  }
```

[44] *http://de.wikipedia.org*

```
  if ($("#station-sel").html() != "") {
    //Falls eine Station ausgewaehlt ist, den Namen
    //zum Suchstring hinzufuegen
    search = search + " " + $("#station-sel").html();
  }
  if (repWhiteSpace.toLowerCase() != "no") {
    //Falls WhiteSpaces ersetzt werden sollen,
    //diese nun ersetzen
    search = search.replace(/ /g, repWhiteSpace);
    search = search.replace(/\-/g, "");
    search = search.replace(/\+\+/g, repWhiteSpace);
  }
  //Den Suchstring zurueckgeben
  return search;
}
```

Diese Funktion stellt aus den einzelnen HTML-Elementen einen Suchtext zusammen und ersetzt die enthaltenen Leerzeichen. Die Ersetzung der Leerzeichen können wir durch einen Übergabeparameter steuern. Damit haben wir eine Möglichkeit geschaffen, für alle zukünftigen Dienste immer den passenden Suchstring zu erhalten.

### 4.12.2 Die API von Flickr

Bevor wir damit beginnen, diese Funktionen zu erstellen, müssen Sie einen sogenannten API-Key bei Flickr beantragen. Dieser Schlüssel wird dazu verwendet, dass Sie gegenüber dem Dienst identifiziert werden und kein Missbrauch betrieben werden kann. Der Schlüssel ist kostenlos. Falls Sie noch nicht bei *Flickr* angemeldet sind, müssen Sie als Erstes ein Konto bei *Yahoo*[45] anlegen. Danach können Sie den API-Key beantragen. Falls Sie bereits ein Konto bei *Yahoo* besitzen, können Sie es für die Anmeldung beim Fotodienst benutzen.

**▣ Lesezeichen**

http://www.flickr.com/services/apps/create/apply/
Adresse für die Erstellung des Flickr-API-Keys

Falls Sie mehr über die API von *Flickr* wissen wollen, finden Sie die gesamte Dokumentation unter der folgenden URL:

**▣ Lesezeichen**

http://www.flickr.com/services/api
Dokumentation der Flickr-API

[45] *Flickr* ist ein Dienst von *Yahoo*

Diese Funktionalität besteht insgesamt aus drei unterschiedlichen Funktionen. Diese müssen wir jetzt erstellen, bevor wir den Service testen können. Beginnen wir zuerst mit einer Hilfsfunktion.

### 4.12.3 Fotos von der Anzeige ausschließen

Da alle Bilder beim Fotodienst von den Anwendern mit Titel und Tags versehen werden, kann es – aus welchen Gründen auch immer – zu falschen Zuordnungen kommen. Daher schaffen wir eine Möglichkeit, mit der Anwender bestimmte Fotos für die Anzeige sperren können.

Dazu werden wir wieder ein neues Dokument anlegen. Die ID des Dokuments erstellen wir aus dem Schlüsselwort `restrict_` und dem Namen der Stadt. Innerhalb des Dokuments legen wir ein Datenfeld `created_at` und `type` an, wie wir es bei jedem Dokument gemacht haben. Im Datenfeld `flickr` speichern wir die IDs der gesperrten Fotos als String ab. Die IDs trennen wir mit einem Semikolon, um sie später schnell durchsuchen zu können. Natürlich könnten wir auch ein Array oder ein JSON-Objekt erstellen, aber um auch eine andere Technik kennenzulernen, verwenden wir diesmal einen einfachen String.

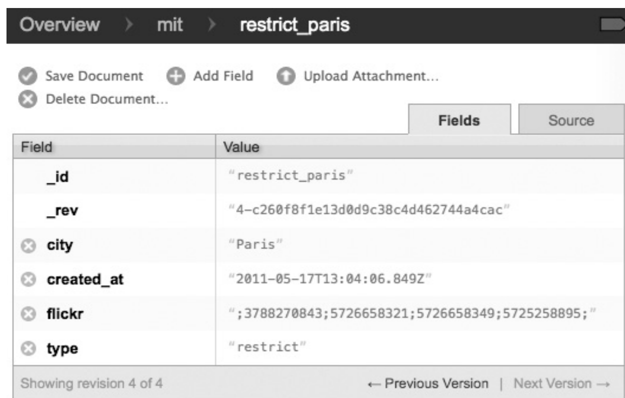


**Bild 4.68:** Dokument mit den gesperrten Foto-IDs von Paris

Dieses Dokument legen wir selbstverständlich nicht selbst an, sondern der Anwender kann bei jedem Bild auf einen Knopf klicken und das Foto zum Sperrdokument hinzufügen. Diesen Trigger werden wir erst später zuordnen, aber die Funktion `restrictFlickr()` können wir bereits jetzt in der Datei *mit.helper.js* anlegen.

```
/*****
 * Foto von Flickr fuer die Ausgabe sperren
 * elemID string
 *   ID des Knopfes der fuer die Sperre angeklickt wurde
 *****/
```

```
function restrictFlickr(elemID) {
  //ID des Fotos aus der ID des Elements extrahieren
  picID = elemID.split("-")[0];
  //Dokument-ID erstellen
  docID = "restrict_" + $("#city-sel").html().toLowerCase();
  //Das Dokument fuer die gesperrten Werte oeffnen
  mit.openDoc(docID, {
    success: function(oldDoc) {
      //Dokument vorhanden
      //Die neuen IDs dem Dokument uebergeben
      if (oldDoc.flickr.indexOf(";" + picID + ";") < 0) {
        //Nur wenn die Bild-ID noch nicht gespeichert wurde
        oldDoc.flickr = oldDoc.flickr + picID + ";";
      }
      //Das Dokument wieder speichern
      mit.saveDoc(oldDoc, {
        success: function() {
          //Code fuer erfolgreiches Speichern
          $("#error").html("<p style='color: #378C84'>" +
                           "Sperre wurde hinzugefügt" +
                           "...</p>");
          //Meldung einblenden
          $("#error").fadeIn(1500);
          //Nach 60 Sekunden die Meldung wieder ausblenden
          $("#error").delay(6000).fadeOut(1500);
        },
        error: function(status, error, reason) {
          //Code fuer einen Speicherfehler
          $("#error").html("<p>Sperre konnte nicht " +
                           "gespeichert werden!</p>" +
                           "<p>Fehler:<br/>"+ error +
                           "<br/>" + reason + "</p>");
          //Fehlermeldung einblenden
          $("#error").fadeIn(1500);
          //Nach 60 Sekunden die Fehlermeldung ausblenden
          $("#error").delay(6000).fadeOut(1500);
        }
      });
    },
    error: function() {
      //Dokument neu anlegen
      flickrPic = ";" + picID + ";";
      //Variable fuer neues Dokument erstellen
      var newDoc;
      //JSON-OBjekt fuer neues Dokument erstellen
```

```
      newDoc = {_id: docID,
                created_at: new Date(),
                type: "restrict",
                city: $("#city-sel").html(),
                flickr: flickrPic
               };
      //Das neue Dokument in der Datenbank speichern
      mit.saveDoc(newDoc, {
        success: function() {
          //Code fuer erfolgreiches Speichern
          $("#error").html("<p style='color: #378C84'>" +
                           "Sperre wurde neu angelegt" +
                           "...</p>");
          //Meldung einblenden
          $("#error").fadeIn(1500);
          //Nach 60 Sekunden die Meldung wieder ausblenden
          $("#error").delay(6000).fadeOut(1500);
        },
        error: function(status, error, reason) {
          //Code fuer einen Speicherfehler
          $("#error").html("<p>Sperre konnte nicht " +
                           "angelegt werden!</p>" +
                           "<p>Fehler:<br/>"+ error +
                           "<br/>" + reason + "</p>");
          //Fehlermeldung einblenden
          $("#error").fadeIn(1500);
          //Nach 60 Sekunden die Fehlermeldung ausblenden
            $("#error").delay(6000).fadeOut(1500);
        }
      });
    }
  });
}
```

Die Funktion selbst benutzt nur Techniken, die wir bereits kennengelernt haben, um das Dokument zu erstellen beziehungsweise zu ergänzen. Da wir später die Daten auch abfragen müssen, können wir gleich eine passende *View* im Verzeichnis *mit/views/restrict/* erstellen. Wie immer speichern wir die Funktion in der Datei *map.js* ab.

```
function(doc) {
  if (doc.type == "restrict") {
    emit([doc.city, "flickr"], doc.flickr);
  }
}
```

Diese Funktion stellt uns eine Ergebnismenge nach Stadt und dem Schlüsselwort `flickr` zur Verfügung. Als Wert wird nur der String mit den gesperrten IDs ausgegeben. Sie ist absichtlich noch sehr einfach gehalten, damit Sie diese Funktion später – falls Sie den MIT weiterentwickeln – für weitere Sperrvermerke verwenden können.

### 4.12.4 Fotos von Flickr holen und anzeigen

Die Funktion, um die Fotos mit der API von *Flickr* zu ermitteln, ist einfach gehalten. Da der Fotodienst die Antwort im JSON-Format zur Verfügung stellt, haben wir auch mit den Daten kaum Probleme, da wir sie einfach einblenden können. Beginnen wir mit der Erstellung der Funktion `getFlickr()` in der Datei *mit.helper.js*.

```
//Passende Fotos von Flickr ermitteln und anzeigen
function getFlickr() {
  //Den Suchstring zusammenstellen
  search = getSearch();
  //Falls der Suchstring leer ist, die Funktion verlassen
  if (search == "") {return;}
  //Anzahl der Bilder pro Seite
  flkPhotos = 30;
  //API-Key von Flickr
  //Hier muss der eigene Key eingesetzt werden
  var apiKey = 'EIGENEN KEY EINSETZEN';
...
```

Zuerst definieren wir einige Grundwerte für die Funktion. Wichtig ist hier, dass der Anwender bereits Daten ausgewählt hat, damit ein Suchtext zusammengestellt werden kann. Vergessen Sie bitte nicht, in der letzten Zeile Ihren eigenen API-Key einzusetzen.

```
...
  //Basis-URL der Flickr-API
  flkURL = "http://api.flickr.com/services/rest/?" +
           "method=flickr.photos.search&api_key=" +
           apiKey + "&text=";
  //Format fuer URL der Flickr-API
  urlExt = "&format=json&jsoncallback=?";
  //Falls Bilder auf der Seite angezeigt werden,
  //diese entfernen
  if ($("#metro-pics").html() != "") {
    $("#metro-pics").html("");
  }
  //Die URL fuer die Flickr-API zusammenstellen
  url = flkURL + search + "&per_page=";
...
```

Nun können wir die URL für die *Flickr*-API in einzelnen Variablen ablegen und danach das Element für die Bilder initialisieren. Dies ist notwendig, da der Anwender ja von Stadt zu Stadt oder auch von Station zu Station springen kann. Würden wir das Element nicht jedes Mal initialisieren, dann würden einerseits immer mehr Bilder angezeigt werden und andererseits wären viele falsch zugeordnete Fotos zu sehen.

```
...
  //String fuer die Sperr-IDs initialisieren
  flkLockID = "";
  //Die View fuer die Sperren direkt aufrufen
  //und auf Flickr einschraenken
  mit.view(dbName + "/restrict", {
    key: [$("#city-sel").html(), "flickr"],
    success: function(data) {
      if (data.rows[0]) {
        //Wenn Daten vorhanden sind, diese speichern
        flkLockID = data.rows[0].value;
      }
      //Anzahl der IDs ermitteln
      restCount = flkLockID.split(";").length - 2;
      //Anzahl der Fotos pro Seite um die
      //Anzahl der gesperrten erhoehen
      flkPhotos += restCount;
...
```

Jetzt rufen wir die Ergebnismenge der zuerst erzeugten *View* ab. Damit können wir die IDs der gesperrten Bilder in einer String-Variablen speichern und später die Fotos ausschließen.

```
...
      //Variable fuer die Bild-URL
      var src;
      //Bild fuer den Knopf zur Fotosperre
      imgLock = "<img src='images/icons/pic_lock." +
                "png' width='16' height='16' alt='Bild " +
                "sperren' title='Bild sperren' />";
      //Die Flickr-API abfragen
      $.getJSON(url + flkPhotos + urlExt,
        function(data) {
          //Durch alle Bilder iterieren
          $.each(data.photos.photo, function(i,item) {
            //Falls die ID des Bildes nicht in den
            //Sperr-IDs enthalten ist:
            if (flkLockID.indexOf(";"+item.id+";") < 0) {
...
```

Durch den Aufruf der *Flickr*-API erhalten wir die Anzahl der gewünschten Bilder als Objekt von dem Fotodienst. Durch dieses Array können wir iterieren und jedes Objekt verarbeiten. Dabei berücksichtigen wir die gesperrten IDs. Denn für diese ist es nicht sinnvoll, die folgenden Programmschritte durchzuführen, da sie nicht angezeigt werden.

```
...
              //Die URL zum Bild zusammenstellen
              //_m.jpg sind mittlere und
              //_s.jpg kleine Thumbnails
              src = "http://farm" + item.farm +
                    ".static.flickr.com/" +
                    item.server + "/" + item.id + "_" +
                    item.secret + "_m.jpg";
            //Ein span-Element in den Layer einfuegen
            span = $("<span></span>").appendTo("#metro-pics");
            //Die Foto-ID als Element-ID verwenden
            span.attr("id", item.id);
            //Klasse zuordnen
            span.attr("class", "flickr");
            //Das Bild in das span-Element einfuegen
            img = $("<img/>").attr("src", src).appendTo("#" + item.id);
            //Das span-Element fuer den Bildttitel innerhalb
            //des neuen span-Elements einfuegen
            span = $("<span></span>").appendTo("#" + item.id);
            //ID fuer das span-Element setzen
            span.attr("id", item.id + "-title");
            //Klasse zuordnen
            span.attr("class", "flickr-title");
            //Den Bildtitel in das Element einfuegen
            $(span).html("<strong>Titel:</strong> " +
                         item.title);
            //Das span-Element fuer den Namen des Fotografen
            //unterhalb des Bildtitels einfuegen
            span = $("<span></span>").appendTo("#" + item.id);
            //ID definieren
            span.attr("id", item.id + "-person");
            //Klasse zuordnen
            span.attr("class", "flickr-person");
            //span-Element fuer die Links zum Profil und zum
            //Fotostream des Fotografen unterhalb des
            //Namens einfuegen
            span = $("<span></span>").appendTo("#" + item.id);
            //ID definieren
            span.attr("id", item.id + "-links");
            //Klasse zuordnen
```

```
            span.attr("class", "flickr-links");
            //Das span-Element fuer den Sperrknopf unterhalb
            //der Links einfuegen
            span = $("<span>" + imgLock +
                     "</span>").appendTo("#" + item.id);
            //ID definieren
            span.attr("id", item.id + "-button");
            //Klasse zuordnen
            span.attr("class", "flickr-button");
            //Einen Trigger fuer den Knopf definieren
            span.click(function() {
              restrictFlickr(this.id);
            });
...
```

Nun integrieren wir die Bilder in je ein `span`-Element und ergänzen diese um weitere Information wie den Titel und die Elemente für Daten des Fotografen. Auch den Knopf für die Sperre eines Fotos integrieren wir und versehen ihn mit einem Trigger. Die Funktion für dieses Klick-Ereignis haben wir bereits im vorherigen Abschnitt erstellt.

```
...
            //Den Namen des Fotografen ueber
            //eine Subroutine ermitteln
            user = getFlickrOwner(apiKey,
                                  item.owner,item.id);
          }
        });
...
```

Als Letztes rufen wir die Daten des *Flickr*-Benutzers ab. Die dafür notwendige Funktion erstellen wir im nächsten Abschnitt.

```
...
        //Unterhalb der Bilder ein reset-Element einfuegen
        $("#metro-pics").append("<div class='clearfix " +
                                "sep16'> </div>");
      });
    }
  });
}
```

Der letzte Schritt ist der Einbau eines `div`-Elements mit der Klasse `clearfix`. Damit stellen wir sicher, dass das umschließende `div`-Element die richtige Höhe besitzt, obwohl alle enthaltenen Elemente mit dem Attribut `float` versehen sind.

### 4.12.5 Die Benutzerdaten von Flickr holen

Bei jedem Foto liefert *Flickr* leider nur die Basisdaten mit. Dazu gehört zwar die eindeutige Benutzerkennung, aber keine weiteren Informationen zum Benutzer. Die Daten wie Benutzer-, Nickname, Profil oder Fotostream müssen mit einem anderen API-Aufruf aus dem *Flickr App Garden* ermittelt werden.

Hier stellen sich uns aber Probleme in den Weg. Einerseits können wir die Daten bei jedem Foto direkt abrufen. Dazu müssten wir nur in der Schleife den API-Aufruf integrieren. Dies ist aber aus mehreren Gründen nicht sehr sinnvoll. Denn wenn ein Benutzer mehrere Fotos zur Verfügung stellt, werden ein und dieselben Daten mehrfach vom Métro Information Tracer abgerufen. Dies belastet die API von *Flickr* und macht unsere Applikation auch langsam. Denn 30 Fotos sind auch 30 Abrufe der Benutzerdaten.

Andererseits haben API-Aufrufe eine andere Eigenart, die wir bis jetzt ignoriert haben: Sie erfolgen immer asynchron. Damit ist gemeint, dass die Anfrage an den Server gestellt wird und das lokale Programm in der Ausführung fortfährt. Sobald die Daten vom Server übertragen wurden, wird dann der entsprechende Programmteil ausgeführt. Bei uns würde das bedeuten, dass wir die Benutzerdaten selten rechtzeitig erhalten, um sie beim Foto ausgeben zu können. Wir müssten daher synchrone Abfragen durchführen und die Applikation so lange anhalten, bis die Daten übertragen wurden. Dadurch beschleunigt sich der MIT nicht gerade.

Diese Probleme umgehen wir mit folgendem Ansatz: Einerseits tragen wir bei der Erstellung nur »Platzhalter« für die Benutzerdaten ein. Dies sind eigene `span`-Elemente, die im Moment der Darstellung noch leer sind und erst bei einer erfolgreichen Antwort der API für die Benutzerdaten gefüllt werden. Andererseits erstellen wir ein globales Array, das die bereits erhaltenen Benutzerdaten zwischenspeichert. Damit können wir prüfen, ob Benutzerdaten bereits übertragen wurden und können sie wiederverwenden. Wenn ein Anwender sich länger mit dem Métro Information Tracer beschäftigt, werden immer mehr Benutzerdaten in diesem Array gespeichert und dadurch immer weniger API-Aufrufe notwendig. Nun können wir in der Datei *mit.helper.js* die Funktion `getFlickrOwner()` anlegen.

```
//Globales, assoziatives Array fuer
//die Flickr-Benutzerdaten
var flPers = new Array;
...
```

Zuerst definieren wir ein globales Array, um die Benutzerdaten zu speichern und die API-Aufrufe zu minimieren. Wir verwenden es als assoziatives Array und definieren als Schlüssel die Benutzer-ID von *Flickr*. Damit können wir die Daten später schnell wieder abrufen.

```
...
/*****
 * Benutzerdaten von Flickr abrufen
```

```
 * apiKey string
 *   persoenlicher API-Key
 * userID string
 *   ID des Flickr-Benutzers
 * picID string
 *   eindeutige ID des Bildes des Flickr-Benutzers
 *****/
function getFlickrOwner(apiKey, userID, picID) {
  //Falls der API-Key oder die Benutzer-ID fehlt,
  //die Funktion verlassen
  if (!apiKey || !userID) {return "";}
...
```

Zuerst überprüfen wir, ob alle notwendigen Parameter übergeben wurden und verlassen die Funktion, falls etwas fehlt.

```
...
  //URL fuer Personendaten der Flickr-API
  flkOwnerURL = "http://api.flickr.com/services/rest/?" +
                "method=flickr.people.getInfo&api_key=" +
                apiKey + "&user_id=" + userID;
  //URL-Erweiterung fuer Benutzerdaten der Flickr-API
  flkOwnerURLext = "&format=json&nojsoncallback=1";
  //Falls die Daten noch nicht im Array gespeichert wurden
  if (!flPers[userID] || !flPers[userID]["nick"]) {
    //Ein neues Element im Array anlegen
    flPers[userID] = new Array;
    //Die API von Flickr abrufen
    $.getJSON(flkOwnerURL + flkOwnerURLext,
...
```

Nachdem wir die Daten für den Aufruf zusammengestellt haben, fordern wir die Benutzerdaten von *Flickr* an.

```
...
      function(data) {
        //Falls keine Daten vorhanden sind,
        //die Funktion verlassen
        if(!data) {return;}
        //Das Antwort-Objekt in einer Variablen speichern
        p = data.person;
        //Falls ein echter Name vorhanden ist,
        //diesen im Array ablegen
        if (p.realname) {
          flPers[userID]["real"] = p.realname._content;
        } else {
          flPers[userID]["real"] = "";
```

```
        }
        //Falls ein Nickname vorhanden ist,
        //diesen im Array ablegen
        if (p.username) {
          flPers[userID]["nick"] = p.username._content;
        } else {
          flPers[userID]["nick"] = "";
        }
        //Falls eine URL zum Profil vorhanden ist,
        //diese im Array ablegen
        if (p.profileurl) {
          flPers[userID]["prof"] = p.profileurl._content;
        } else {
          flPers[userID]["prof"] = "";
        }
        //Falls eine URL zum Fotostream vorhanden ist,
        //diese im Array ablegen
        if (p.photosurl) {
          flPers[userID]["pics"] = p.photosurl._content;
        } else {
          flPers[userID]["pics"] = "";
        }
...
```

Die neu erhaltenen Benutzerdaten verwenden wir dazu, um das assoziative Array mit diesen Daten zu ergänzen und uns einen weiteren API-Aufruf zu ersparen.

```
...
        if (flPers[userID]["nick"] != "") {
          //Wenn ein Nickname vorhanden ist,
          //diesen für das Bild speichern
          picTitle = flPers[userID]["nick"];
          if (flPers[userID]["real"] != "") {
            //Wenn ein echter Name vorhanden ist,
            //diesen hinzufuegen
            picTitle += " (" + flPers[userID]["real"] + ")";
          }
        } else {
          //Wenn nur ein echter Name vorhanden ist,
          //diesen für das Bild speichern
          if (flPers[userID]["real"] != "") {
            picTitle += flPers[userID]["real"];
          } else {
            picTitle += userID;
          }
        }
```

```
        //Den String ergaenzen und anzeigen
        if (picTitle != "") {
          picTitle = "<strong>Von:</strong> " + picTitle;
          $("#" + picID + "-person").html(picTitle);
        }
        //Wenn ein Profil-Link vorhanden ist,
        //diesen für das Bild speichern
        if (flPers[userID]["prof"]) {
          picLnk1 = "[<a href='" + flPers[userID]["prof"] +
                    "' target='_blank' title='Flickr-" +
                    "Profil von " + flPers[userID]["nick"] +
                    "'>Profil</a>]";
        } else {
          picLnk1 = "";
        }
        //Wenn ein Fotostream-Link vorhanden ist,
        //diesen für das Bild speichern
        if (flPers[userID]["pics"]) {
          picLnk2 = "[<a href='" + flPers[userID]["pics"] +
                    "' target='_blank' title='Flickr-" +
                    "Bilder von " + flPers[userID]["nick"] +
                    "'>Foto-Stream</a>]";
        } else {
          picLnk2 = "";
        }
        //Die Links ergaenzen und anzeigen
        picLinks = $.trim(picLnk1 + " " + picLnk2);
        if (picLinks != "") {
          picLinks = "<strong>Links:</strong> " + picLinks;
          $("#" + picID + "-links").html(picLinks);
        }
      }
    );
  } else {
...
```

Da wir die Daten von der API erhalten haben und auch das Array ergänzt wurde, können wir diese Daten im Benutzerinterface anzeigen. Wir verwenden zur Identifikation der Elemente für die Daten die ID des Fotos, die in der Variablen `picID` übergeben wurde.

```
...
    //Falls der Benutzer bereits im Array vorhanden ist,
    //die Daten zusammenstellen und ausgeben
    if (flPers[userID]["nick"]) {
      if (flPers[userID]["nick"] != "") {
```

```
        picTitle = flPers[userID]["nick"];
        if (flPers[userID]["real"] != "") {
          picTitle += " (" + flPers[userID]["real"] +
                      ")";
        }
      } else {
        if (flPers[userID]["real"] != "") {
          picTitle += flPers[userID]["real"];
        } else {
          picTitle += userID;
        }
      }
      if (picTitle != "") {
        picTitle = "<strong>Von:</strong> " + picTitle;
        $("#" + picID + "-person").html(picTitle);
      }
      if (flPers[userID]["prof"]) {
        picLnk1 = "[<a href='" + flPers[userID]["prof"] +
                  "' target='_blank' title='Flickr-" +
                  "Profil von " + flPers[userID]["nick"] +
                  "'>Profil</a>]";
      } else {
        picLnk1 = "";
      }
      if (flPers[userID]["pics"]) {
        picLnk2 = "[<a href='" + flPers[userID]["pics"] +
                  "' target='_blank' title='Flickr-" +
                  "Bilder von " + flPers[userID]["nick"] +
                  "'>Foto-Stream</a>]";
      } else {
        picLnk2 = "";
      }
      picLinks = $.trim(picLnk1 + " " + picLnk2);
      if (picLinks != "") {
        picLinks = "<strong>Links:</strong> " + picLinks;
        $("#" + picID + "-links").html(picLinks);
      }
    }
  }
}
```

Falls die Benutzerdaten bereits im Array vorhanden sind, können wir sie direkt ausgeben. Dies müssen wir ein zweites Mal durchführen. Denn auch hier gilt, dass der Abruf der Daten asynchron erfolgt. Dadurch würde die Ausgabe der Daten sofort erfolgen, ohne dass die Antwort eingetroffen ist.

Nun können wir auch die komplette Funktion für die Anzeige der *Flickr*-Fotos testen.

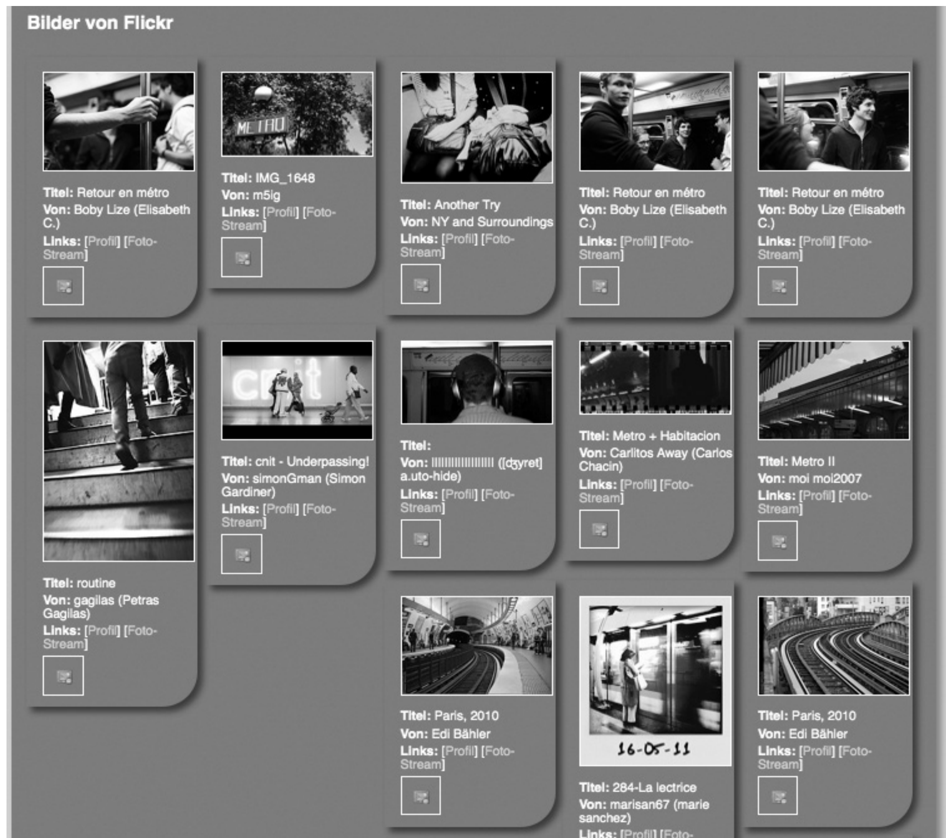


**Bild 4.69:** Der Flickr-Fotostream für Paris

Jetzt haben wir die Anzeige dieser Zusatzinformationen abgeschlossen. Es gäbe hier auch noch einige Dinge, die programmiert werden können. Allen voran wäre im Verwaltungsinterface ein Modul für die Verwaltung der gesperrten Fotos notwendig. Auch eine Bewertung der Fotos ist eine Idee, die umgesetzt werden kann. Eine weitere Möglichkeit wäre es, die Benutzerdaten in der Datenbank abzulegen, um die API-Zugriffe noch stärker zu minimieren. Mit dem *App Garden* könnten auch die Tags der Bilder abgefragt werden. Dies kann eine zusätzliche Information für die Anwender darstellen. Sie könnten sie aber auch in der Datenbank ablegen, wenn ein Anwender ein Foto anklickt. Durch diese Information könnte der Métro Information Tracer bei jedem Aufruf genauere Suchergebnisse zur Verfügung stellen. Sie sehen: Raum für Erweiterungen ist genügend vorhanden, und nur Ihre Vorstellungskraft setzt hier die Grenzen.

Clemens Gull

Web-Applikationen entwickeln mit

# NoSQL

*Ohne Datenbanken geht nichts mehr im Web! Aber nach einer Phase fast uneingeschränkter Herrschaft der SQL-basierten Datenbanken findet im Moment ein Wandel in der Landschaft der Web-Applikationen statt. Teil dieses Wandels ist die vermehrte Verwendung von NoSQL-Datenbanken. NoSQL-Datenbanken sind nicht relational, können auf verteilten Systemen liegen und verzichten meist auf ACID-Eigenschaften (atomicity, consistency, isolation, durability). Die Vorteile liegen in einer besseren Skalierbarkeit und besserer Performance bei hoher Datenlast bzw. vielen Transaktionen. Für den gelungenen Einstieg wird hier mit CouchDB der angesagteste Vertreter aus der Riege der NoSQL-Datenbanken verwendet.*

▶ **Schneller Einstieg in die Praxis**
Nach einer kurzen Einführung in die Unterschiede zwischen SQL- und NoSQL-Datenbanken geht es gleich weiter in die Praxis. Nach der lokalen Installation der NoSQL-Datenbank Apache CouchDB stellt Autor Clemens Gull die Programmierung von Web-Applikationen in einfachen Beispielen mithilfe von CouchApp vor.

▶ **Schritt für Schritt zur umfangreichen Web-Applikation**
Im Rahmen der Erstellung einer großen Beispielapplikation, des Métro Information Tracers, lernen Sie alle notwendigen Schritte zur Entwicklung einer Web-Applikation kennen: Vom Einrichten der Datenbank über die Anpassung des Benutzerinterfaces bis zur Einbindung von Daten aus Google Maps, Flickr oder Wikipedia bzw. durch Screen Scraping aus weiteren externen Seiten.

▶ **Wichtig für Einsteiger: Die Fachbegriffe für den Umgang mit Datenbanken**
In einem umfangreichen Glossar werden alle Fachbegriffe, die bei der Arbeit mit Datenbanken und ihrem Einsatz zur Webseiten-Erstellung auftauchen, ausführlich erklärt.

## Auf www.buch.cd:

- Installationsdateien für CouchDB und CouchApp
- Alle Beispielcodes des Buchs
- Erweiterter Quelltext
- Alle verwendeten Bilder und Icons
- Alle Daten und Dokumente im JSON-Format

## Aus dem Inhalt:

- Die Theorie hinter NoSQL
- Grundlagen von CouchDB
- Installation von CouchDB und CouchApp auf Ihrem System
- Die Entwicklungsumgebung
- Erste Übungen mit CouchDB
- Futon – das Webinterface der CouchDB
- Daten im JSON-Format
- Arbeiten mit Views
- Die Show-Funktionen
- Statisches HTML und die CouchDB
- AJAX mit der CouchDB
- Die Beispielanwendung MIT – der Métro Information Tracer
- Frameworks der CouchApp
- Auf Benutzeraktionen mit Evently reagieren
- Die Möglichkeiten von Mustache
- Google Maps einbinden
- Screen Scraping
- Das in die CouchApp integrierte jQuery um ein Plug-in erweitern
- Daten von Flickr anzeigen
- Zusätzliche Informationen aus Wikipedia integrieren

## Über den Autor:

Clemens Gull studierte Informationstechnologie und Systemmanagement. Er arbeitete als Programmierer und Netzwerkadministrator unter anderem für die Salzburger Sparkasse. Heute leitet er das Webdesignunternehmen Byte Brothers, darüber hinaus ist Gull als Dozent für die Fachhochschule Salzburg und andere Institute aktiv. Sein Weblog „Guru 2.0" (www.guru-20.info) zählt zu den meistgelesenen deutschsprachigen Blogs zum Thema Internetprogrammierung.

**30,– EUR** [D]
ISBN 978-3-645-60104-7

Besuchen Sie unsere Website
**www.franzis.de**