3 Das kleine MongoDB-Einmaleins

3.1 Grundlagenforschung

Früher hat mich mein Lateinlehrer immer damit erziehen wollen, dass ich doch ohne Grundlagen nicht weit kommen würde. Recht hat er, deswegen muss ich jetzt auch Bücher schreiben. Schade eigentlich, dass es damals zwar viele Mongos aber kein MongoDB-Leistungskurs in der Schule gab.

3.1.1 Was ist denn bitteschön BSON

Ein Wortspiel vorweg, BSON hat nichts mit dem gleichnamigen Tier zu tun, das aussieht wie eine haarige Kuh, sondern es ist der ältere Bruder von JSON. JSON, das als schlankes Serialisierungsformat im JavaScript-Umfeld bekannt ist, hat ein kleines Problem. Es kennt nur eine beschränkte Anzahl an Datentypen. Ziemlich aufwändig ist zum Beispiel das Übertragen von Binärdaten in JSON. Dies ist nicht so einfach möglich. Man müsste nämlich die Binärdaten vorher durch den Base64-Fleischwolf drehen, um sie übertragen zu können, denn JavaScript kennt keinen binären Datentyp von Haus aus.

Auch gibt es in JSON keinen Datentyp für einen Datumswert. Zwar könnte man ein Datum als Timestamp übertragen, aber dann geht die Information der Zeitzone verloren. Man sieht also, dass JSON zwar schlank und sehr einfach zu lernen, aber nicht allmächtig ist.

Und so wurde BSON geboren, das nun ein fester Bestandteil von MongoDB ist. Denn MongoDB speichert Dokumente intern in BSON ab, hält sie als BSON im Speicher und tauscht diese auch wieder in BSON mit dem Client Driver aus. Daher muss jeder Client Driver BSON verstehen und sprechen können. Anfangs war BSON noch fest in MongoDB integriert und auch nur für dieses wirklich nutzbar. Inzwischen hat 10gen daraus eine eigene Spezifikation gemacht, die man auch für andere Projekte nutzen kann. Zu finden ist BSON unter bsonspec.org.

Ein Beispiel für ein von BSON serialisiertes Objekt wäre:

`{"kuh":"moo","bson":"boo"} ➔`	`\000\000\000\002kuh\000\004\000\000\000 moo\000\002bson\000\004\000\000\000boo\ 000\000`

Listing 3.1: Aus JSON mach BSON

HINWEIS: Wenn man z. B. in der mongo JavaScript Shell ein Query abschickt, dass ein Dokument mit Binärdaten zurückliefern soll, werden diese Binärdaten nicht in der Shell angezeigt (geht auch gar nicht, da JavaScript kein Binary sprechen kannt), sondern durch einen Platzhalter ersetzt. Daher nicht wundern, wo die Binärdaten bleiben.

3.1.2 Eine Datenbank

Jetzt wirst du sicher gleich fragen: Wieso erklärt der mir bitteschön, was eine Datenbank ist? Ich bin doch nicht auf den Kopf gefallen. Da stimme ich dir vollkommen zu, denn sonst hättest du dich sicher nicht für MongoDB entschieden. Jedoch unterscheidet sich die Datenbank in MongoDB von der klassischen Datenbank wie man sie z. B. bei MySQL oder anderen RDBMS findet.

Nun ein klassisches Beispiel einer RDBMS-Datenbank und ihrem Inhalt:

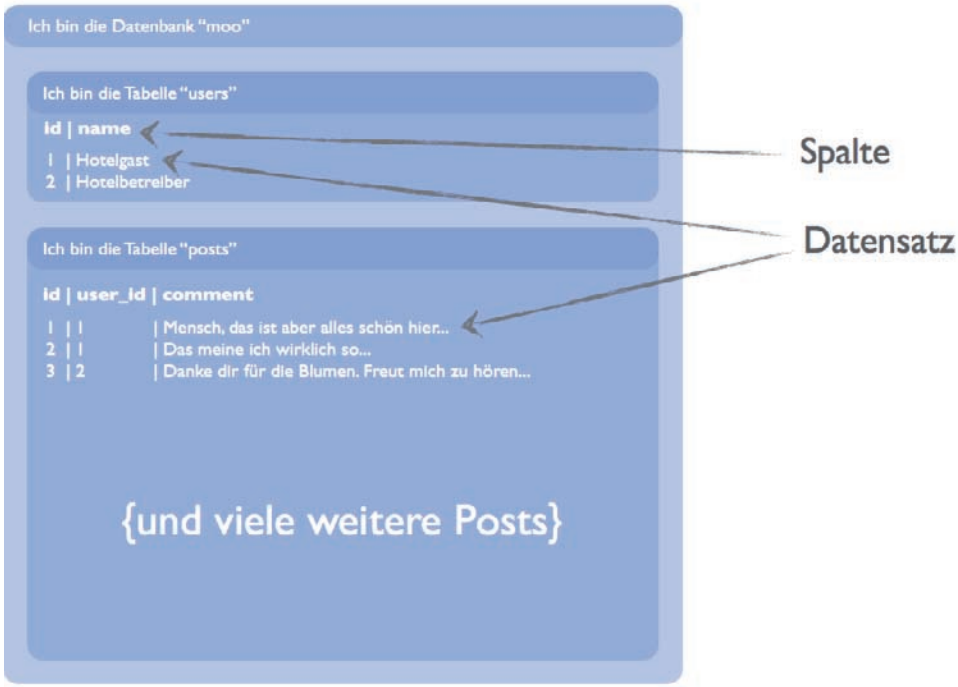

Abbildung 3.1: Klassische RDBMS-Konstruktion

In einer Datenbank gibt es mehrere Tabellen, die ein festes Schema besitzen. Diese Tabellen haben Spalten mit Inhalten. Referenzen werden über Fremdschlüssel hergestellt. In unserem Beispiel wird ein Post über den Fremdschlüssel user_id mit der Tabelle users verknüpft. Also alles kalter Kaffee und nicht wirklich spannend.

Bei MongoDB sieht das etwas anders aus. In folgender Abbildung kann man den Unterschied erkennen.

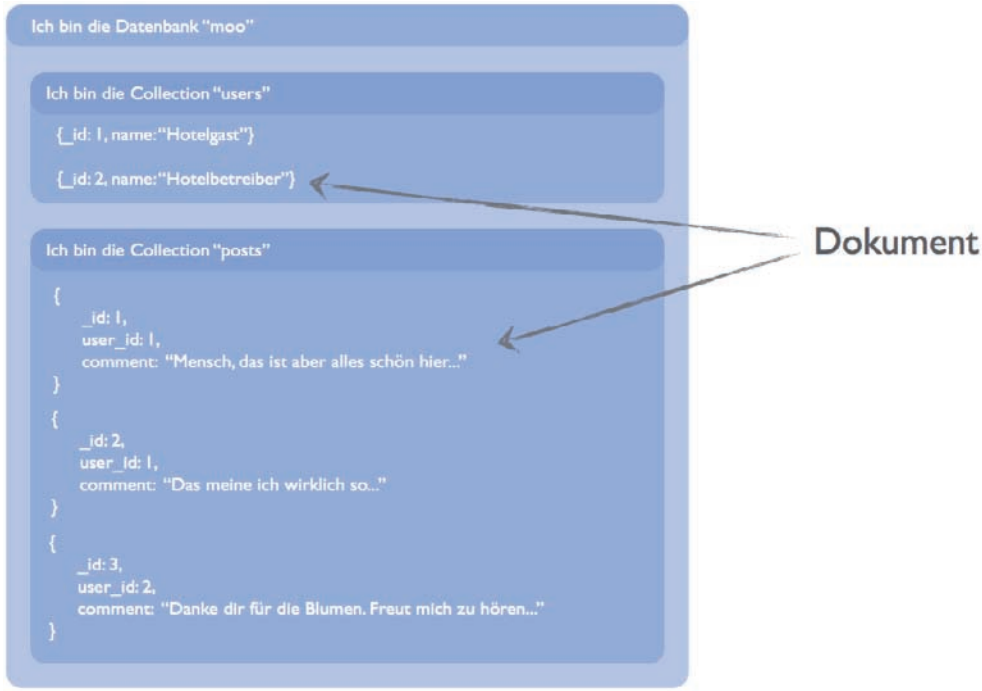

Abbildung 3.2: MongoDB-Datenbank mit Collection und Dokumenten

Wie man sieht, sieht man kaum einen Unterschied. Verdammt! Das liegt aber daran, dass die einzigen Änderungen nur die Bezeichnung „Collection" im Vergleich zu „Tabelle" und die „Dokumente" im Vergleich zu den „Datensätzen" in einem RDBMS sind. Man kann sagen, dass eine Datenbank bei MongoDB viele Collections enthält, die den Tabellen in einem RDBMS entsprechen.

Technisch gesehen, legt der mongo Daemon beim Initialisieren einer Datenbank einfach drei neue Datafiles im Datenverzeichnis an. Würde die Datenbank gisela heißen, dann würde mongod die Dateien gisela.0, gisela.1 und gisela.ns anlegen. gisela.0 und gisela.1 sind Datafiles, die die Dokumente später im BSON-Format enthalten. gisela.ns ist ein Namespace-File, in dem eine Liste der Collections, die in der Datenbank gisela leben, abgelegt wird. Je größer das ns-File ist, desto mehr Collections und Metadaten können dort angelegt werden.

HINWEIS: MongoDB legt Datenbanken automatisch an. Das bedeutet, man muss nicht explizit sagen: „Lieber MongoDB-Server, ich hätte gerne eine neue Datenbank mit dem Namen gisela. Kannst du dich mal bitte drum kümmern?“ Sondern man verwendet einfach die Datenbank gisela, indem man in der mongo JavaScript Shell use gisela eingibt oder im Client Driver die Datenbank gisela auswählt.

3.1.3 Eine Collection

Eine Datenbank enthält viele Collections. Diese muss man sich wie kleine Tonnen vorstellen, in denen die Dokumente leben. Jetzt aber bitte nicht denken, dass die Dokumente unglücklich wären, nur weil sie in einer Tonne leben. Denen geht's sehr gut da drin.

Wie schon eben bei den Datenbanken erwähnt, muss man auch eine Collection nicht explizit anlegen. Sondern sie wird bei Bedarf einfach frisch erzeugt. Collections können beliebig benannt werden (Buchstaben, Zahlen und Unterstriche mit max. 128 Zeichen Länge). Jedoch mit Ausnahme des Keywords system, das von MongoDB reserviert wird, um Metadaten bereitzustellen. Auch sollte eine Collection mit einem Buchstaben anstatt mit einer Zahl beginnen.

Ein Vorteil der flexiblen Namensgebung ist, dass man mit einem Punkt (Dot-Notation) Gruppen und Hierarchien in Collections erzeugen kann.

Ein etwas hinkendes Beispiel dafür wäre folgendes:

```
- users.females.intelligent
- users.females.beautiful
```

Hier ist die User-Collection unterteilt nach Geschlecht und Eigenschaft. Ein schlauer Entwickler würde jetzt natürlich die beiden Collections miteinander kreuzen und daraus selektieren. Technisch gesehen handelt es sich aber um zwei vollkommen verschiedene Collections, jedoch hilft hier der Name bei der Identifizierung/Gruppierung.

Würde man ein Portal entwickeln (übrigens ein ziemlich öder Job), das einen Blog und ein Forum als Modul enthält, kommt man mit den Begrifflichkeiten schnell zu einem Konflikt, denn Posts gibt es bei beiden Modulen, Forum und Blog. Daher gruppiert man am besten anhand des jeweiligen Hauptmoduls:

```
- blog.posts
- blog.comments
- blog.categories
- forum.users
- forum.topics
- forum.posts
```

Listing 3.2: Gruppierung von Collections mithilfe der Dot-Notation

3.1.4 Die Capped Collection

Auch unter den Collections gibt es Sonderlinge, die anders ticken als der Rest. Darunter fällt auch die Capped Collection. Ihre Aufgabe ist es, nicht unendlich zu wachsen, sondern nur eine bestimmte Anzahl an Dokumenten aufzunehmen. Dies ist besonders interessant für Logging-Anwendungen. Hier wird garantiert, dass nicht irgendwann die Festplatte überläuft, nur weil zuviel geloggt wurde.

Kurz gesagt, die Capped Collection ist eine Collection, die eine bestimmte Größe hat, und beim Erreichen dieser Größe werden alte Dokumente automatisch entfernt, damit neue Dokumente Platz haben. Das Verfahren dahinter nennt sich „Least Recently Used" oder auch LRU.

Einsatzgebiete[1]

- Logging: Hier spart man sich den Aufwand, ein Log Rotating einzubauen.
- Caching: Speichern von schon berechneten Informationen, die sich automatisch erneuern, weil nämlich alte Informationen ablaufen.
- Archivierung: Hinzufügen von Versionen eines Dokuments, bei dem nach einer gewissen Anzahl an Versionen automatisch die alten Versionen gelöscht werden.

Was darf man mit Capped Collections machen?

- Man kann diese ganz einfach mit neuen Dokumenten befüllen. Erreicht man das Limit, also entweder Dokumentenanzahl oder Größe, werden alte Dokumente entfernt, damit die neuen Platz haben.
- Man kann Dokumente updaten, jedoch muss das neue Dokument kleiner oder genauso groß sein wie das alte. Größere Dokumente können nicht eingefügt werden. Fährt der Smart aus der Parklücke, kommt die S-Klasse in diese schwer rein. Und wenn ja, wird es teuer.
- Man darf keine Dokumente löschen, sondern kann stattdessen nur die gesamte Collection entfernen und neu erstellen. Alles-oder-Nichts-Prinzip.

Erstellen einer neuen Capped Collection

```
db.createCollection("log", {capped: true, size: <bytes>, max: <docs>});
```

Listing 3.3: Erstellen einer Capped Collection mit dem Namen log

Die Größe muss beim Erstellen in Bytes angegeben werden. Allerdings sind dies nicht nur die Bytes der einzelnen Daten, die man darin speichern möchte, sondern es kommt

1 Capped-Collections Einsatzgebiete: *bit.ly/capped_collections*
www.mongodb.org/display/DOCSDE/Capped+Collections#CappedCollections-Applications

zusätzlich noch der Overhead von BSON hinzu. Um die Dokumente auf eine gewisse Anzahl zu beschränken, kann man die Option max: <number-of-docs> verwenden.

3.1.5 System-Collections

Die System-Collections sind Pseudo-Collections, die nicht vom User für die Datenspeicherung genutzt werden können. In ihnen werden Metainformationen zur Datenbank abgelegt. Dies hat den Vorteil, dass man keine extra Konfigurationsdateien erstellen muss, um die Funktion der Datenbank anzupassen. Denn man nutzt einfach die System-Collection in jeder Datenbank, um Änderungen an der aktuellen Konfiguration vorzunehmen.

Da es eine ganze List an verschiedenen System-Collections gibt, findest du hier eine Tabelle mit einer kleinen Übersicht:

Collection Name	Aufgabe
system.namespaces	Hier werden die Namespaces einer Datenbank verwaltet. Diese basieren auch auf den Indizes, die man zuvor vergeben hat.
system.indexes	Indizes, die für die einzelnen Collections angelegt wurden, werden hier abgelegt. Dort kann man schnell nachprüfen, ob für ein spezielles Feld auch ein Index existiert.
system.profile	Möchte man die Datenbank genauer unter die Lupe nehmen, muss man das Profiling aktivieren, das seine Informationen in der *system.profile* Collection ablegt.
system.users	Verwendet man die Zugriffskontrolle, greift MongoDB auf diese Collection zu, um zu sehen, ob der aktuelle User berechtigt ist, auf eine bestimmte Datenbank zugreifen zu können.

Tabelle 3.1: Übersicht über die System-Collections

System-Collections sind schreibgeschützt. Das heißt man kann sie zwar ganz normal abfragen (wie das geht, ist in Kapitel 3.2 beschrieben), aber nicht einfach Dokumente in ihnen löschen, hinzufügen oder ändern. Möchte man zum Beispiel einen neuen Index hinzufügen, nutzt man dafür einen extra Command, der den Index erstellt und in der dafür vorgesehenen System-Collection abspeichert. Mehr zum Thema Indizes findest du in Kapitel 4.1.

3.1.6 Ein Dokument

Das Dokument ist die kleinste Einheit im MongoDB-Universum. Eine Collection besteht aus einer beliebigen Anzahl von Dokumenten. Dokumente sind zu vergleichen mit Datensätzen in einer Tabelle, um wieder die Brücke zu den RDBMS herzustellen.

Der Unterschied zu starren Tabellen ist, dass Dokumente zwar ein identisches Schema aufweisen können, nicht aber explizit müssen. Es ist durchaus möglich, alle Informatio-

nen, die bei einer Webapp anfallen, in eine einzige Collection zu speichern. Ob das aber sinnvoll ist, ist mehr als fraglich. Man schmeißt ja auch nicht sein McDonalds-Menü in den Mixer, bevor man es isst. Jedoch kommt es oft vor, dass ein Dokument in der gleichen Collection nicht alle, oder sogar mehr Informationen besitzt, z. B. ein anderes Dokument. Mit solchen Situationen kommen gewöhnliche RDBMS nicht sonderlich gut klar. MongoDB interessiert das eigentlich herzlich wenig, welche Informationen ein Dokument besitzt.

Natürlich werden jetzt die Kritiker sagen: „Ja aber dann ist meine Datenbasis nicht mehr konsistent, wenn jedes Dokument unterschiedlich sein kann." Aber ist es wirklich Aufgabe der Datenbank, für absolute Konsistenz zu sorgen und eine absolute Validität herzustellen? Wenn eine Webapp das Alter eines Users speichern soll, wieso muss dann die Datenbank feststellen, ob es sich bei dem vom User eingegebenen Wert um eine Zahl und wenn ja um eine im Bereich 1-99 Jahre handelt? Es ist doch normalerweise Aufgabe der Applikation sicherzustellen, dass nur valide Daten die Datenbank erreichen?

Daher gibt es in MongoDB auch keine Validierung auf Daten- sondern nur auf BSON-Ebene. Damit wird sichergestellt, dass der MongoDB-Server kein ungültiges BSON abspeichert, nicht jedoch, ob gewisse Werte valide, bezogen auf ihren Inhalt, sind.

3.1.7 Wie ist ein Dokument aufgebaut?

```
{
    "_id": ObjectId("4baa0023a917bae567000001"),
    "string": "Value",
    "integer": 1234,
    "float": 1.2,
    "array": [1, 2, 3],
    "doc": {"a": 1, "b": 2}
    ...
}
```

Listing 3.4: Aufbau eines Dokuments

In einem Dokument kann es fast beliebig viele Felder (Keys) geben, denen ein Wert zugeordnet ist. Dieser Wert kann unterschiedliche Datentypen enthalten, wie man in obigem Beispiel sieht.

HINWEIS: Wichtig ist, dass jedes Dokument, egal wie klein es ist, immer einen Key _id mit einem Wert vom Datentyp ObjectId enthält. Gibt man keine _id an, wird automatisch eine erzeugt. Dieser Key wird später z. B. auch für Referenzen zwischen Dokumenten verwendet.

In der aktuellen MongoDB-Implementation ist es jedoch so, dass ein Dokument nicht mehr als 8 MB inkl. des BSON Overheads, der durch die Keys entsteht, haben darf. Dies

hört sich jetzt vielleicht kontraproduktiv an, wenn man doch auch Dateien größer 8 MB speichern möchte. Wie das Problem umschifft werden kann, wird aber im GridFS Kapitel 4.3 genauer erklärt.

Welche Datentypen gibt es?

Datentyp	Beispiel
String	`{ "_id" : ObjectId("4c431220310eda114fe14f77"), "string" : "hans" }`
Integer	`{ „_id“ : ObjectId(„4c43123b310eda114fe14f78“), „integer“ : 123 }`
Float	`{ "_id" : ObjectId("4c431015310eda114fe14f76"), "float" : 111.11 }`
Boolean	`{ "_id" : ObjectId("4c431258310eda114fe14f79"), "boolean" : false }`
Array	`{ "_id" : ObjectId("4c431285310eda114fe14f7a"), "array" : [ 1, 2, 3 ] }`
Embedded-Doc	`{ "_id" : ObjectId("4c43128f310eda114fe14f7b"), "doc" : { "a" : 1 } }`
Date	`{ "_id" : ObjectId("4c4312ae310eda114fe14f7c"), "date" : "Sun Jul 18 2010 16:41:50 GMT+0200 (CEST)" }`
DBRef	`{ "_id" : ObjectId("4c4314c0310eda114fe14f7d"), "ref" : { "$ref" : "users", "$id" : ObjectId("4c43128f310eda114fe14f7b") } }`
ObjectId	`{ "_id" : ObjectId("4c4314e1310eda114fe14f7e") }`

Tabelle 3.2: Beispiele für die Darstellung der Datentypen in der mongo JavaScript Shell

Die Datentypen ObjectId und DBRef werden in den folgenden Kapiteln noch genauer erklärt.

String Encoding

Strings können beim Speichern und bei der Abfrage UTF-8 enkodiert sein. MongoDB kann diese dann automatisch verarbeiten. Es muss also nichts vorher umständlich ins ASCII Format konvertiert werden.

Integer Attribute

Wer jetzt Attribute für Integer-Werte wie unsigned und Co. vermisst, dem sei ans Herz gelegt, dass MongoDB einem das Leben vereinfachen möchte. Schwerer machen es uns sowieso schon genügend andere Dinge. Und soviel Speicherverlust erleidet man nicht, wenn man ein Integer-Feld einmal nicht mit „enthält nur positive Werte“ markiert.

Ein Beispiel für ein komplexeres Dokument

```
{
    "_id" : ObjectId("4c121a2ca917ba523e000002"),
    "activated_at" : "Fri Jun 11 2010 13:13:13 GMT+0200 (CEST)",
    "data" : { "style" : "green" },
    "created_at" : "Fri Jun 11 2010 13:12:44 GMT+0200 (CEST)",
    "template_id" : ObjectId("4c122095a917ba583d000001"),
    "details" : {
        "salution" : "Frau",
        "title" : "Dr.",
        "last_name" : "Grünspan",
        "first_name" : "Gisela"
        "zip" : "70000",
        "city" : "Stuttgart",
        "address" : "Einbahnstraße 123",
    },
    "updated_at" : "Tue Jun 15 2010 13:03:59 GMT+0200 (CEST)",
    "profile_id" : ObjectId("4c121a2ca917ba523e000001"),
    "active" : true
}
```

Listing 3.5: Auch umfangreichere Sachen lassen sich mit MongoDB gut bewältigen

3.1.8 Embedded Document

Embedded Documents sind nichts anderes als Teildokumente, die in ein Top-Level-Dokument eingebettet sind. Zwar kann man Dokumente auch bis zur Unendlichkeit verschachteln, aber das wäre dann sehr hirnfrei.

Wenn man aus der RDBMS-Ecke kommt, dann ist man oft gewohnt, alles zu normalisieren und möglichst viele Informationen auszulagern, um Redundanzen zu vermeiden. Daraus resultieren dann viele Joins, um an die entsprechenden Daten zu kommen, die wiederum die Abfragegeschwindigkeit nach unten ziehen.

Daher verfolgt MongoDB den Ansatz, Referenzen durch Embedding zu ersetzen. Simpel und einfach. Eingebettete Dokumente verursachen nämlich keine aufwändigen Joins.

Ein Beispiel für den sinnvollen Einsatz von Embedded Documents

```
{
    "_id" : ObjectId("4c121a2ca917ba523e000002"),
    "email" : "moo@moobar.com",
    "password": "45974ec63b36e340af22369f9466e9ebc5f2b163"
}
```

Listing 3.6: Dokument in der users Collection

Referenzierung mit einem Dokument in der phone_numbers-Collection:

```
{
    "_id" : ObjectId("4c121a2ca917ba533e000003"),
    "user_id": ObjectId("4c121a2ca917ba523e000002"),
    "home": "123456789",
    "work": "987654321",
    "mobile": "47110815"
}
```

Listing 3.7: Dokument in der Collection phone_numbers, das zu einem User gehört

Man sieht hier, dass zwei Abfragen benötigt würden, um an die Telefonnummern des Users mit der E-Mail-Adresse moo@moobar.com zu kommen. Daher macht es hier mehr als Sinn, die Telefonnummern in das Userdokument einzubetten. In Listing 3.7 sieht man dann das Ergebnis harter Arbeit:

```
{
    "_id" : ObjectId("4c121a2ca917ba523e000002"),
    "email" : "moo@moobar.com",
    "password": "45974ec63b36e340af22369f9466e9ebc5f2b163",
    "phone_numbers": {
        "home": "123456789",
        "work": "987654321",
        "mobile": "47110815"
    }
}
```

Listing 3.8: Die Telefonnummern wurden in das Userdokument eingebettet

Ein weiterer Vorteil, man spart Platz, da die Referenzen entfallen.

3.1.9 Eine ObjectId

Die ObjectId ist wohl das Herzstück eines jeden Dokuments. Diese wird immer unter dem Key _id gespeichert. Dabei muss die ID eines Dokuments nicht unbedingt eine ObjectId sein, sondern kann einen beliebigen Typ annehmen, jedoch empfiehlt es sich, die ObjectId als einheitlichen Standard zu verwenden.

Die ObjectId ist genau 12 Byte lang und setzt sich folgendermaßen zusammen:

Byte	0	1	2	3	4	5	6	7	8	9	10	11
Wert	Timestamp				System-ID			Prozess-ID		Inkrement. Counter		

Tabelle 3.3: Aufbau einer ObjectId

Aber bitte jetzt keine Panik bekommen. Denn die ObjectId wird entweder vom Client Driver oder der Datenbank automatisch generiert. Also absolut stressfrei. Man muss sich um nichts kümmern. Natürlich kann man auch seine eigene ObjectId nach einem ganz anderen Schema generieren, ob das jedoch wirklich den Mehraufwand wert ist?

Bitte beim Generieren daran denken, dass die selbsterdachte ObjectId auf jeden Fall nur einmal in der Collection vorkommt und genau 12 Byte lang ist. Denn dies sind die beiden Anforderungen an eine ObjectId.

> **HINWEIS:** Beim Erstellen von Referenzen und beim Abfragen bitte immer darauf achten, dass die ObjectId auch von Datentyp ObjectId und nicht nur ein einfacher String ist.

Am Anfang meiner MongoDB-Tage hatte ich bei einer Abfrage einen 12 Zeichen langen Hex-String mit der vermeintlichen ObjectId verwendet und mich gewundert, wieso ich kein Ergebnis bekam. Das Dokument mit der eigentlichen ObjectId hat aber in der Collection existiert. Das Problem war, dass ich bei der Abfrage als Datentyp String und nicht ObjectId verwendet habe:

```
// Lookup by _id as an ObjectId
> db.users.count(ObjectId("4c121a2ca917ba523e000002"));
1



// Lookup as plain text
> db.users.count("4c121a2ca917ba523e000002");
0
```

Listing 3.9: Der Unterschied zwischen String und ObjectId bei der Abfrage

3.1.10 Eine DBRef(erence)

Der letzte Begriff in unserem MongoDB-Einmaleins ist die Datenbankreferenz, zu vergleichen mit einem Fremdschlüssel in einem RDBMS.

Um es vorweg zu nehmen, man braucht DBRef nicht, um eine Referenz zwischen zwei Dokumenten herzustellen. Dies kann auch einfacher mit der ObjectId erfolgen. Trotzdem der Vollständigkeit halber hier die Erklärung zur DBRef.

DBRef erwartet zwei Parameter, nämlich die Collection und die ObjectId. Damit ist der Sinn auch eigentlich schon klar. Eine DBRef zeigt auf ein Dokument in der gleichen oder einer anderen Collection.

Greifen wir wieder das vorherige Beispiel auf, setzen aber diesmal auf eine DBRef. Nämlich der User, dem wir Telefonnummern zuordnen möchten:

```
{
    "_id" : ObjectId("4c121a2ca917ba523e000002"),
    "email" : "moo@moobar.com",
    "password": "45974ec63b36e340af22369f9466e9ebc5f2b163"
}
```

Listing 3.10: Dokument in der Users-Collection

```
{
    "_id" : ObjectId("4c121a2ca917ba533e000003"),
    "user": {
        "$ref" : "users",
        "$id" : ObjectId("4c121a2ca917ba523e000002")
    },
    "home": "123456789",
    "work": "987654321",
    "mobile": "47110815"
}
```

Listing 3.11: Hier kommt der DBRef-Datentyp zum Einsatz

3.2 Erste Schritte

Jetzt wird es ernst. Denn in den nachfolgenden Unterkapiteln muss der MongoDB-Server, oder man selbst, richtig schwitzen. Die folgenden Beispiele bitte in der mongo JavaScript Shell ausführen. Dabei ist die Syntax für die Commands nach einem einheitlichen Schema aufgebaut:

```
db.<collection>.<action>(<parameters>);
```

Listing 3.12: Genereller Command, der für die aktuell ausgewählte Datenbank gilt

Jedem Command wird immer das Kürzel db vorangestellt. Dieses sagt aus, dass sich der folgende Befehl auf die aktuell ausgewählte Datenbank bezieht. Danach folgt immer die Collection, in deren Kontext der Befehl ausgeführt wird. Nun folgt endlich die Aktion, die wir verwenden wollen. Jede Aktion erfordert unterschiedliche Parameter, die der Methode übergeben werden. Dazu aber gleich mehr.

3.2.1 Daten rein

Ohne Daten drin kommt man aber auch nicht an Daten ran und kann somit auch nichts löschen oder updaten. Die Daten (in Form von Dokumenten) werden dabei immer in einem sehr der JSON Notation ähnelnden (BSON-)Format definiert und an die Datenbank geschickt. In der mongo JavaScript-Shell benutzt man dazu folgenden Command:

```
> db.moo.insert({
      name: "fritz",
      age: 20,
      hobbies: ["soccer", "mongodb", "cheeseburger"]
  });

ObjectId("4c43717b310eda114fe14f7f")
```

Listing 3.13: Mit der insert-Methode kann man einfach Dokumente anlegen

Man erkennt an dem Rückgabewert der insert-Methode, nämlich der ObjectId des erzeugten Dokuments, dass der Vorgang erfolgreich war. Bekommt man einen JSError, dann hat man eventuell einen Syntaxfehler gemacht oder die Daten sind nicht valide. Aber anhand der Fehlermeldung kann man gut erkennen, was genau nicht funktioniert hat. Danke liebe Fehlermeldung.

Manchmal kann es aber auch vorkommen, dass man gerne einen Punkt (.) im Key verwenden möchte. Dies würde mit obigem Command nicht funktionieren, da der Punkt eigentlich nicht Bestandteil eines Keys sein darf. Daher muss man die interne Validierung deaktivieren, um einen Punkt nutzen zu können. Dies funktioniert folgendermaßen:

```
> db.moo.insert({'name.prename': "fritz"}, true);

ObjectId("4c43717b310eda114fe34f7a")
```

Listing 3.14: Verwenden von Punkten im Key

Entscheidend ist hier der zweite Parameter der insert-Methode. Dieser erwartet einen Boolean-Wert und entscheidet, ob die Punkt-Notation verwendet werden soll.

Eine weitere interessante Methode heißt save und setzt auf der insert Method auf. Übergibt man der save-Methode nun ein JSON-Dokument, das bereits eine ObjectId enthält, wird ein Update des Dokuments durchgeführt, anstatt dieses neu zu erzeugen:

```
// Show collection (collection is empty)
> db.moo.find()

// Insert new document
> db.moo.save({name: "hans", age: 30});
ObjectId("4c4377b7310eda114fe14f82")
```

```
// Get document by ObjectId
> db.moo.find()
{
    "_id" : ObjectId("4c4377b7310eda114fe14f82"),
    "name" : "hans",
    "age" : 30
}

// Update document, as it contains an ObjectId
> db.moo.save({
    "_id" : ObjectId("4c4377b7310eda114fe14f82"),
    "name" : "hans",
    "age" : 50
});

// Demonstrate, that the update has worked
> db.moo.find()
{
    "_id" : ObjectId("4c4377b7310eda114fe14f82"),
    "name" : "hans",
    "age" : 50
}
```

Listing 3.15: Demonstration der save-Methode

In obigem Beispiel erkennt man, dass der zweite save Command kein neues Dokument erstellt, sondern das bereits vorhandene Dokument aktualisiert hat. Der Grund liegt in der ObjectId, die beim zweiten Speichern übergeben wurde.

3.2.2 Daten raus

Daten rein war noch schön einfach. Perfekt, um warm zu werden. Daten raus ist dagegen etwas kniffliger, aber auch noch gut verständlich. Los geht's!

Beim Abfragen von Daten unterscheidet man zwischen mehreren Arten. Möchte ich ein Dokument gezielt oder mehrere Dokumente abfragen? Frage ich anhand einer mir bekannten ObjectId ab oder verwende ich ein frei wählbares Query? Auf die Frage, ob ich einen oder mehrere Dokumente abfragen möchte, gibt es eine einfache Antwort. findOne ist, wie der Name schon sagt, für ein Dokument die richtige Wahl, find kommt bei mehreren Dokumenten zum Einsatz. Natürlich kann ich auch find für ein einzelnes Dokument nutzen, aber dieses liegt dann als Cursor-Objekt (ähnlich einem intelligenten Array) vor, bei dem ich zuerst das erste Dokument selektieren muss. Kurz gesagt, unnötiger Mehraufwand. Daher lieber findOne für ein einzelnes Dokument.

Die find-Method als auch die findOne-Methode erwarten als Parameter entweder eine ObjectId oder ein Query als JavaScript Object. Hört sich kompliziert an, ist es aber nicht.

Beispiel : Selektieren eines Dokuments anhand der ObjectId

```
// Show collection (collection is empty)
> db.moo.find()

// Insert new document
> db.moo.insert({name: 'hans', age: 40});
ObjectId("4c448e940d897c6443e2f70b")

// Get document by ObjectId
> db.moo.findOne(ObjectId("4c448e940d897c6443e2f70b"));
{
    "_id" : ObjectId("4c448e940d897c6443e2f70b"),
    "name" : "hans",
    "age" : 40
}

// Assign result to variable
var res = db.moo.findOne(ObjectId("4c448e940d897c6443e2f70b"));

// Print object attributes
> res.name;
hans
> res.age;
40
```

Listing 3.16: Abfragen eines Dokuments anhand der ObjectId mit findOne

So weit so gut. Jetzt ist es aber zu 99,9 % so, dass man die ObjectId nicht im Kopf hat, daher bietet MongoDB auch die Möglichkeit, nach einzelnen Werten direkt zu suchen. Damit aber die folgenden Beispiele auch Spaß machen, brauchen wir richtige Testdaten. Daher importieren wir einen Haufen amerikanischer Städte, mit denen wir im ganzen Buch arbeiten wollen. Wenn man aus Versehen die Daten versaut, kann man folgenden Schritt immer wiederholen und hat danach einen frischen Datenpool:

```
# Download
$ wget http://bit.ly/zips_json

# Import
$ <path-to-mongodb>/mongoimport –d cities –c cities zips.json
```

Listing 3.17: Import der Städtedaten

Um zu überprüfen, ob alles funktioniert hat, starten wir die mongo JavaScript Shell und wechseln zur Datenbank cities. Sollte die Methode count eine vernünftig große Zahl ausspucken, hat der Import geklappt:

```
// Switch to database cities
> use cities;
```

```
switched to db cities

// Get the total number of documents in the cities collection
> db.cities.count();
29470
```

Listing 3.18: Anzeigen der Anzahl an Dokumenten in der Datenbank

Als Erstes möchten wir uns alle Städte anzeigen lassen, die ATLANTA heißen:

```
// Get cities with name ATLANTA
> db.cities.find({city: "ATLANTA"});
{
    "_id" : ObjectId("4c449a2cd187be742001a154"),
    "city" : "ATLANTA",
    "zip" : "30303",
    "loc" : { "y" : 33.752504, "x" : 84.388846 },
    "pop" : 1845,
    "state" : "GA"
}
{
    "_id" : ObjectId("4c449a2cd187be742001a155"),
    "city" : "ATLANTA",
    "zip" : "30305",
    "loc" : { "y" : 33.831963, "x" : 84.385145 },
    "pop" : 19122,
    "state" : "GA"
}
...
// There are many more entries
```

Listing 3.19: Abfrage aller Städte, die Atlanta heißen

Jetzt fragt sich natürlich jeder, der in Erdkunde gut aufgepasst hat, wieso man so viel mehr Einträge zum Suchbegriff ATLANTA erhält. Atlanta ist doch nur eine Großstadt in den USA? Die Antwort ist einfach. Es gibt mehrere Atlanten und das allseits bekannte Atlanta hat wiederum mehrere Stadtbezirke, die unterschiedliche Postleitzahlen haben. Und genau diese Stadtbezirke sind in den Testdaten enthalten. Tadaa!

Wir haben für die Abfrage die find-Methode verwendet, die für mehrere Dokumente geeignet ist. Den Parameter, den wir der Methode übergeben haben, ist ein JavaScript-Objekt im Key/Value-Format. Der Key heißt in unserem Fall city und die Value, also der Name der Stadt, die wir abfragen möchten, heißt ATLANTA. Queries sind generell immer in der Form Key:Value aufgebaut. Möchte man mehrere Bedingungen nutzen, trennt man diese einfach durch ein Komma:

```
{key_1: "value_1", key_2: "value_2", key_blah: "value_blubb"}
```

Listing 3.20: Mehrere Bedingungen in einer Abfrage

Nun wollen wir dies mit unseren Testdaten ausprobieren. Dazu kommt als zweite Bedingung noch die Postleitzahl hinzu. Wenn die Datenbasis korrekt ist, sollte jetzt nur noch ein Dokument erscheinen:

```
// Combine city and zip in one query
> db.cities.find({city: "ATLANTA", zip: "30312"});
{
    "_id" : ObjectId("4c449a2cd187be742001a15c"),
    "city" : "ATLANTA",
    "zip" : "30312",
    "loc" : { "y" : 33.746749, "x" : 84.378125 },
    "pop" : 17683,
    "state" : "GA"
}
```

Listing 3.21: Eine Abfrage mit mehreren Bedingungen in der Praxis

Siehe da, wie auch nicht anders zu erwarten war, gibt MongoDB nun nur noch ein Dokument zurück. Übrigens funktioniert das Abfragen mit dem Key/Value-Schema nicht nur mit den Methoden find und findOne, sondern auch mit Methoden update, remove und count:

```
// Query only by one key
> db.cities.count({city: "ATLANTA"});
41

// Take two keys for querying (reduces number of documents)
> db.cities.count({city: "ATLANTA", state: "GA"});
31
```

Listing 3.22: Eine Abfrage mit der count Method

In obigem Beispiel kann man gut erkennen, wie die Auswahl an verfügbaren Dokumenten immer mehr eingeschränkt wird, je mehr Bedingungen hinzugefügt werden.

find kann aber noch ein bisschen mehr als einfach nur eine Liste mit Ergebnissen auszuspucken. Man kann z. B. festlegen, welche Felder ausgegeben werden sollen. Dies ist sinnvoll, wenn man umfangreiche Dokumente hat und von diesen nur ein einziges Feld benötigt. Denn dann muss nicht das komplette Dokument vom Server zum Client geschickt werden, sondern es wird nur das benötigte Feld übermittelt.

Beispiele für die weiteren Parameter der find-Methode:

```
// Get only the field "zip"
> db.cities.find({city: "ATLANTA", state: "GA"}, {zip: true});
{ "_id" : ObjectId("4c449a2cd187be742001a154"), "zip" : "30303" }
{ "_id" : ObjectId("4c449a2cd187be742001a155"), "zip" : "30305" }
{ "_id" : ObjectId("4c449a2cd187be742001a156"), "zip" : "30306" }
```

```
... // 17 more documents, stripped for clarity
has more

// Let's select only 2 documents
> db.cities.find({city: "ATLANTA", state: "GA"}, {zip: true}, 2);
{ "_id" : ObjectId("4c449a2cd187be742001a154"), "zip" : "30303" }
{ "_id" : ObjectId("4c449a2cd187be742001a155"), "zip" : "30305" }

// Get 2 documents and skip the first 10 documents (pagination)
> db.cities.find({city: "ATLANTA", state: "GA"}, {zip: true}, 2, 10);
{ "_id" : ObjectId("4c449a2cd187be742001a15e"), "zip" : "30314" }
{ "_id" : ObjectId("4c449a2cd187be742001a15f"), "zip" : "30315" }
```

Listing 3.23: Beispiele für die weiteren Parameter von find

Hier nochmal die genaue Syntax der find-Methode:

```
find(query, fields, num-docs, skip-docs);
```

HINWEIS: Vielleicht fragt sich der eine oder andere, wieso man nicht einen Index setzt, bevor man eine Abfrage auf die Collection ausführt? Das macht natürlich mehr als Sinn, aber dazu mehr in Kapitel 4.2.
Auch stellt sich hier die Frage, ob man nicht auch kompliziertere Queries mit MongoDB ausführen kann?
Kurzversion: Ja, das ist kein Problem.
Langversion: Kapitel 4 hilft!

3.2.3 Daten ändern

Nun haben wir schon Dokumente angelegt und abgefragt, aber noch keine Dokumente verändert. Man kann entweder ein einzelnes Dokument verändern oder mehrere auf einmal. Um Dokumente zu ändern, wird die update-Methode verwendet. Diese erwartet auf jeden Fall zwei, maximal aber vier Parameter:

```
update(query, new-document/update-document, upsert, multiple)
```

Listing 3.24: Parameter der update-Methode

Parameter	Funktion
Query	Ein Query, das bestimmt, welche Dokumente geupdated werden sollen. Das Format des Queries ist identisch mit dem der *find*-Methode.
new-document/ update-document	Ein neues Document, das das alte/die alten ersetzen soll oder eine Kombination aus Update-Modifiern mit den dazugehörigen Werten.
upsert(true/false)	Soll das neue Dokument erzeugt werden, wenn es nicht gefunden wurde?
multiple(true/ false)	Wenn multiple *true* ist, dann wird nicht nur das erste gefundene Dokument geupdated, sondern alle Dokumente.

Tabelle 3.4: Parameter der update-Methode

Beispiel: So funktioniert die update-Methode:

```
// Insert sample document
> db.cows.insert({name: "boo", sound: "moo"});
ObjectId("4c44d02ef1990b697f1c81ef")

// Ge tone document
> db.cows.findOne();
{
    "_id" : ObjectId("4c44d02ef1990b697f1c81ef"),
    "name" : "boo",
    "sound" : "moo"
}

// Update document and set new name and sound values
> db.cows.update(
    {name: "boo"},
    {name: "foo", sound: "moomoo"}
);

// Show changes
> db.cows.find();
{
    "_id" : ObjectId("4c44d02ef1990b697f1c81ef"),
    "name" : "foo",
    "sound" : "moomoo"
}
```

Listing 3.25: Beispiel für die update-Methode

Ganz schön aufwändig, wenn man bei jedem Update immer das komplette Dokument inkl. Änderungen übergeben muss, oder? Ja das stimmt, aber auch dafür hat MongoDB eine geniale Lösung, die ich in Kapitel 4 genauer unter die Lupe nehme. Stichwort: Update-Modifiers. Also schön dranbleiben.

3.2.4 Daten löschen

Dies ist mein Lieblingspart des Kapitels. Denn Daten löschen ist die Kür des Ganzen. Daten löschen ist im Vergleich zu Daten speichern wirklich einfach. Wie schon in „Daten raus“ erwähnt, erwartet die Methode remove ebenfalls ein Query, wie auch ihre Kollegin find.

Kurze Rede, langer Sinn:

```
// Show current number of documents
> db.cities.count()
29470

// Remove all cities named "ATLANTA"
> db.cities.remove({city: 'ATLANTA'});

// Show number of documents again to see, if the removal has worked
> db.cities.count()
29429

// Get one document for later removal
> db.cities.findOne();
{
    "_id" : ObjectId("4c449a2bd187be7420018e5c"),
    "city" : "ACMAR",
    "zip" : "35004",
    "loc" : {
        "y" : 33.584132,
        "x" : 86.51557
    },
    "pop" : 6055,
    "state" : "AL"
}

// Remove a document by its ObjectId
> db.cities.remove(ObjectId("4c449a2bd187be7420018e5c"));

// Verify, that the document has been deleted
> db.cities.count()
29428
```

Listing 3.26: Beispiel für das Löschen von Dokumenten mit einem Query oder via ObjectId

remove löscht gnadenlos all das, was man ihm übergibt. Daher ist das Löschen anhand der ObjectId die sicherste Methode, denn diese identifiziert immer nur ein Dokument.