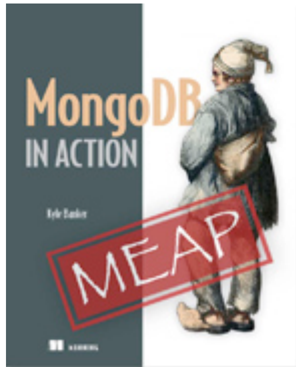
MANNING PUBLICATIONS

Diving into the MongoDB shell

An article from



MongoDB in Action
EARLY ACCESS EDITION

Kyle Banker
MEAP Release: June 2010
Softbound print: Spring 2011 | 375 pages
ISBN: 9781935182870

This article is taken from the book MongoDB in Action*. The author provides a practical tour of the most common tasks performed from the MongoDB shell, which lets users examine and manipulate data and administer the database server itself using the JavaScript programming language and a simple API.*

Tweet this button! (instructions here)

Get **35% off** any version of MongoDB in Action with the checkout code **fcc35**.
Offer is only valid through www.manning.com.

MongoDB's JavaScript shell makes it easy to play with data and get a tangible sense for documents, collections, and the database's particular query language.

We'll begin by getting the shell up and running. Then, we'll look at how JavaScript represents documents, and we'll go ahead and learn how to insert documents into a MongoDB collection. To verify these inserts, we'll practice querying the collection and take a brief tour of MongoDB's query language. Then it's on to updates, using some slightly more intricate documents. And, finally, we'll learn how to drop collections.

Starting the shell

You should have a working MongoDB installation on your computer. Make sure you have a running `mongod` instance; once you do, start the MongoDB shell by running the `mongo` executable.

```
./mongo
```

If the shell program starts successfully, your screen will look like figure 1. The shell heading displays the version of MongoDB you're running along with some additional information about the currently selected database.

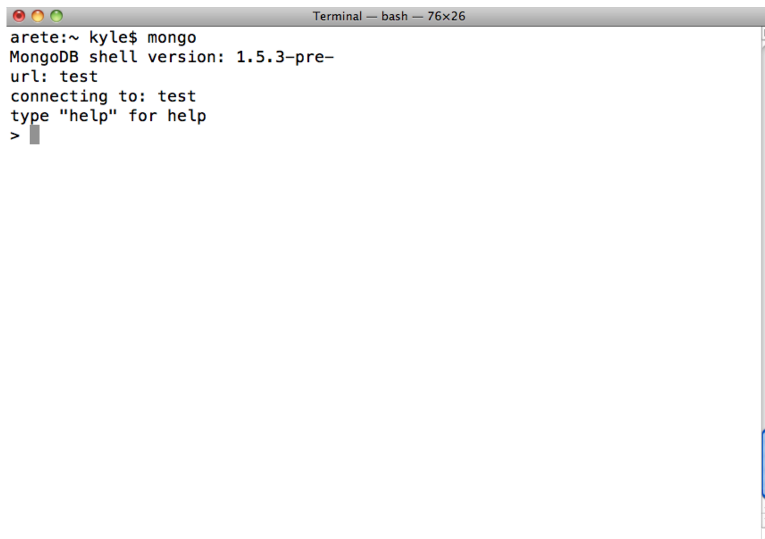


Figure 1 The MongoDB JavaScript shell on startup.

If you know some JavaScript, you can enter some code and start exploring the shell right away. Otherwise, read on to see how to start inserting data.

Inserts and queries

By default, you'll be connected to a database called `test`. However, as a way of keeping all of the subsequent tutorial exercises under the same namespace, let's start by switching to the `tutorial` database:

```
> use tutorial
switched to db tutorial
```

You'll see a message verifying that you've switched to databases.

On creating databases and collections

You may be wondering how it is that we can switch to the tutorial database without explicitly creating it. In fact, this simply isn't required. Databases and collection are created only when documents are first inserted. This behavior is consistent with MongoDB's dynamic approach to data; just as the structure of documents need not be defined in advance, individual collections and databases can be created at runtime. This can lead to a simplified and accelerated development process and essentially facilitates dynamic namespace allocation, which can frequently be useful. If you're concerned about databases or collection being created accidentally, there are settings in most of the drivers that implement a "strict" mode, preventing careless errors.

It's time to create our first document. Since we're using a JavaScript shell, our documents will be specified in JSON. For instance, the simplest imaginable document describing a user might look like this:

```
{username: "jones"}
```

The document contains a single key and value for storing Jones' username. To save this document, we need to choose a collection to save it to. Appropriately enough, let's save it to the `users` collection. Here's how:

```
db.users.insert({username: "smith"});
```

You may notice a slight delay after entering this bit of code. At this point, neither the `tutorial` database nor the `users` collection has been created on disk. The delay is caused by the allocation of the initial data files for both.

If the insert operation succeeded, then you've just saved your first document. A simple query can be issued to verify that the document has been saved:

```
db.users.find();
```

The response will look something like this:

```
{ "_id" : ObjectId("4bf9bec50e32f82523389314"), "username" : "smith" }
```

Notice that, in addition to the `username` field, an `_id` field has been added to the document. You can think of the `_id` value as the document's primary key. Every MongoDB document requires an `_id` and, if one is not present at the time of creation, a special MongoDB object id will be generated and added to the document on save. The object id that appears in your console won't be the same as the one in the code listing; however, it will be unique among all `_id` fields in the collection, which is the only hard requirement for the field.

We'll have more to say about object ids in the next chapter. For the moment, we're going to work through a few more commands. First, let's add a second user to the collection.

```
> db.users.save({username: "jones"});
```

There should now be two documents in the collection. Go ahead and verify this by running the `count` command.

```
> db.users.count();
2
```

Now that we have more than one document in the collection, we can look at some slightly more sophisticated queries. We can still query for all the documents in the collection:

```
> db.users.find()
{ "_id" : ObjectId("4bf9bec50e32f82523389314"), "username" : "smith" }
{ "_id" : ObjectId("4bf9bec90e32f82523389315"), "username" : "jones" }
```

But now we can also pass a simple query selector to the `find` method. A query selector is a document used to match against all of the documents in the collection.

To query for all documents where the username is "jones", we pass simple document that acts as our query selector, like so:

```
> db.users.find({"username": "jones"})
{ "_id" : ObjectId("4bf9bec90e32f82523389315"), "username" : "jones" }
```

The query selector `{"username": "jones"}` shouldn't present any surprises, and the results of the query are as you would expect.

We've just presented the basics of creating and reading data. Now it's time to look at how to update that data.

Updating documents

With some data in the database, we can now start playing with updates. All updates require at least two parameters. The first specifies which documents to update, and the second defines how the selected documents shall be modified. There are in fact two styles of modification; in this section we're to focus on modification by operator, which is by far the most common case and most representative of MongoDB's distinct features.

To take an example, suppose that our user "smith" decides to add her country of residence. We can record this using the following operation:

```
> db.users.update({username: "smith"}, {"$set": {country: "Canada"}});
```

This update tells MongoDB to find a document where the username is "smith" and then to set the value of the "country" property to "Canada." If we now issue a query, we'll see the document has been updated accordingly.

```
> db.users.find({username: "smith"});
{ "_id" : ObjectId("4bf9ec440e32f82523389316"), "country" : "Canada", "username" : "smith"
```

If the user later decides that she no longer wants her country stored in her profile, that value can be removed just as easily using the `$unset` operator:

```
> db.users.update({username: "smith"}, {"$unset": {country: 1}});
```

But let's enrich our example a bit. We are, after all, representing our data with documents, which can contain complex data structures. So let's suppose that, in addition to storing profile information, our users can also store lists of their favorite things. A good document representation might look something like this:

```
{username: "smith",
 favorites: {
   cities: ["Chicago", "Cheyenne"],
   movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
 }
}
```

The "favorites" key points to an object containing two other keys pointing to lists of favorite cities and movies. Given what you know already, can you think of a way to modify the "smith" document to look like this? The `$set` operator should come mind. Notice in this example that we're practically rewriting the document and that this is a perfectly acceptable use of $set.

```
db.users.update( {username: "smith"},
  {"$set": {favorites:
    {
       cities: ["Chicago", "Cheyenne"],
       movies: ["Casablanca", "The Sting"]
    }
  }
});
```

Let's modify the "jones" similarly; in this case, however, we'll just add a couple of favorite movies.

```
db.users.update( {username: "jones"},
  {"$set": {favorites:
    {
       movies: ["Casablanca", "Rocky"]
    }
  }
});
```

Now query the `users` collection to make sure that both updates have succeeded.

```
> db.users.find()
```

With a couple of rich document at our fingertips, we can really begin to see the power of MongoDB's query language. In particular, the query engine's ability to reach into nested inner object and to match against array elements proves especially useful in this situation. To demonstrate this, let's take a look at a very simple example. Suppose we're building a lottery application where all user guesses are stored per document. Run the following two commands, which will save a couple of illustrative guess documents to the "guesses" collection:

```
db.guesses.save( {list: [1, 2, 3] } );
db.guesses.save( {list: {name: "lottery", items: [2, 4, 6]}} );
```

Notice that we've represented a list of integers in a couple of ways: first, as a simple array, and second, as an inner document with two attributes, `name` and `items`.

The query for matching against the first document is trivial.

```
> db.guesses.find( {list: 2} );
{ "_id" : ObjectId("4bffd7da2f95a56b5581efe6"), "list" : [ 1, 2, 3 ] }
```

In plain English, this query says, "Find me a document whose `list` key includes the number 2 as one of its elements." Queries like these, which search across array elements, can take advantage of indexing, assuring efficiency.

But what about the second example, where `list` points to an inner object? It turns out that we can query for this document using a simple dot notation.

```
> db.guesses.find( {'list.items': 2} );
{ "_id" : ObjectId("4bffd83e2f95a56b5581efe7"), "list" : { "name" : "even", "items" : [ 2,
```

Querying on the `name` element is equally straightforward and returns the same result:

```
> db.guesses.find( {'list.name': 'lottery'} );
```

This dot notation can be applied to our `users` collection. To take a slightly more involved scenario, let's just take it for granted that any user who likes "Casablanca" also likes "The Maltese Falcon." How might we represent this as an update?

We could conceivably use the `$set` operator again, but that would require us to rewrite and send the entire array of movies. Since all we want to do is add an element to the list, we're better off using either `$push` or `$addToSet`. Both operators add an item to an array, but the second does so uniquely, preventing a duplicate addition. Here, then, is the update we're looking for:

```
db.users.update( {"favorites.movies": "Casablanca"},
  {$addToSet: {"favorites.movies": "The Maltese Falcon"} },
  {multi: true} );
```

Most of this should be decipherable. The first parameter, a query selector, matches against users who have "Casablanca" in their movies list. Next, the update document adds "The Maltese Falcon" to the list. The third parameter, which specifies `{multi: true}`, indicates that this is a multi-update. By default, a MongoDB update operation will apply only to the first document matched by the query selector. If we want the operation to apply to all documents matched, we must specify `{multi: true}`. Since we want our update to apply to both "smith" and "jones," the multi-update is necessary.

Now, we're going to move on to see how to delete documents.

Deleting data

You now know the basics of creating, reading, and updating data through the MongoDB shell. We've saved the simplest operation—removing data—for the end of our discussion. If given no parameters, a remove operation will clear a collection of its documents. So, to get rid of our `guesses` collection, we simply enter:

```
> db.guesses.remove();
```

Of course, we often need to remove only a certain set of documents and, for that, we can pass a query selector to the `remove()` method. If we wanted to remove all users whose favorite city is Cheyenne, we could pass a simple query selector:

```
> db.users.remove({"favorites.cities": "Cheyenne"});
```

Do keep in mind that the `remove()` operation doesn't actually delete the collection; it merely removes documents from a collection, which makes it analogous to SQL's `DELETE` and `TRUNCATE TABLE` directives.

If your intent is to delete the collection along with all of its indexes, use the `drop()` command:

```
> db.users.drop()
```

Creating, reading, updating, and deleting form the basic operations of any database; if you've followed along, you should be in a position to continue practicing basic CRUD operations in MongoDB.

About MongoDB

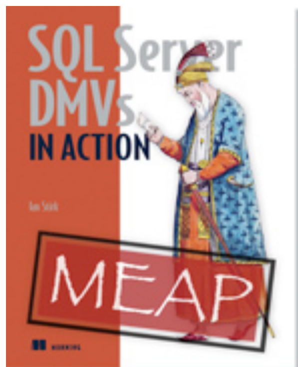
MongoDB is an open-source, document-based database management system. Designed for the data and scalability requirements of modern Internet applications, MongoDB features dynamic queries and secondary indexes; fast atomic updates and complex aggregations; and support for both master-slave replication and automatically-managed sharding for distributing data across multiple machines.

Here are some other Manning titles you might be interested in:

SQL Server MVP Deep Dives

IN PRINT

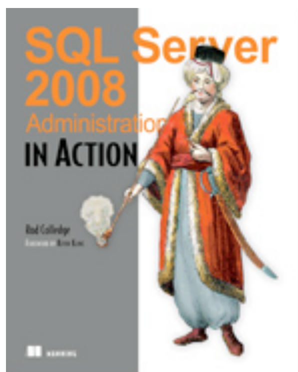
Edited by Paul Nielsen, Kalen Delaney, Greg Low, Adam Machanic, Paul S. Randal, and Kimberly L. Tripp
November 2009 | 848 pages
ISBN: 9781935182047



SQL Server DMVs in Action

EARLY ACCESS EDITION

Ian W. Stirk
MEAP Release: February 2010
Softbound print: Early 2011 | 375 pages
ISBN: 9781935182733



SQL Server 2008 Administration in Action

IN PRINT

Rod Colledge
August 2009 | 464 pages
ISBN: 193398872X