About The Author

Karl Seguin is a developer with experience across various fields and technologies. He's an expert .NET and Ruby developer. He's a semi-active contributor to OSS projects, a technical writer and an occasional speaker. With respect to MongoDB, he was a core contributor to the C# MongoDB library NoRM, wrote the interactive tutorial mongly as well as the Mongo Web Admin. His free service for casual game developers, mogade.com, is powered by MongoDB.

Karl has since written The Little Redis Book

His blog can be found at http://openmymind.net, and he tweets via [@karlseguin](http://twitter.com/karlseguin)

Chapter 3

Getting Started

Most of this book will focus on core MongoDB functionality. We'll therefore rely on the MongoDB shell. While the shell is useful to learn as well as being a useful administrative tool, your code will use a MongoDB driver.

This does bring up the first thing you should know about MongoDB: its drivers. MongoDB has a number of official drivers for various languages. These drivers can be thought of as the various database drivers you are probably already familiar with. On top of these drivers, the development community has built more language/framework-specific libraries. For example, NoRM is a C# library which implements LINQ and MongoMapper is a Ruby library which is ActiveRecord-friendly. Whether you choose to program directly against the core MongoDB drivers or some higher-level library is up to you. I point this out only because many people new to MongoDB are confused as to why there are both official drivers and community libraries - the former generally focuses on core communication/connectivity with MongoDB and the latter with more language and framework specific implementations.

As you read through this, I encourage you to play with MongoDB to replicate what I demonstrate as well as to explore questions that you might come up with on your own. It's easy to get up and running with MongoDB, so let's take a few minutes now to set things up.

1. Head over to the official download page and grab the binaries from the first row (the recommended stable version) for your operating system of choice. For development purposes, you can pick either 32-bit or 64-bit.

2. Extract the archive (wherever you want) and navigate to the `bin` subfolder. Don't execute anything just yet, but know that `mongod` is the server process and `mongo` is the client shell - these are the two executables we'll be spending most of our time with.

3. Create a new text file in the `bin` subfolder named `mongodb.config`.

4. Add a single line to your mongodb.config: `dbpath=PATH_TO_WHERE_YOU_WANT_TO_STORE_YOUR_DATABASE_FILES`. For example, on Windows you might do `dbpath=c:\mongodb\data` and on Linux you might do `dbpath=/var/lib/mongodb/data`.

5. Make sure the `dbpath` you specified exists.

6. Launch mongod with the `--config /path/to/your/mongodb.config` parameter.

Although this is important to understand, don't worry if things aren't yet clear. It won't take more than a couple of inserts to see what this truly means. Ultimately, the point is that a collection isn't strict about what goes in it (it's schema-less). Fields are tracked with each individual document. The benefits and drawbacks of this will be explored in a future chapter.

Let's get hands-on. If you don't have it running already, go ahead and start the `mongod` server as well as a mongo shell. The shell runs JavaScript. There are some global commands you can execute, like `help` or `exit`. Commands that you execute against the current database are executed against the `db` object, such as `db.help()` or `db.stats()`. Commands that you execute against a specific collection, which is what we'll be doing a lot of, are executed against the `db.COLLECTION_NAME` object, such as `db.unicorns.help()` or `db.unicorns.count()`.

Go ahead and enter `db.help()`, you'll get a list of commands that you can execute against the `db` object.

A small side note: Because this is a JavaScript shell, if you execute a method and omit the parentheses `()`, you'll see the method body rather than executing the method. I only mention it so that the first time you do it and get a response that starts with `function (...){` you won't be surprised. For example, if you enter `db.help` (without the parentheses), you'll see the internal implementation of the `help` method.

First we'll use the global `use` helper to switch databases, so go ahead and enter `use learn`. It doesn't matter that the database doesn't really exist yet. The first collection that we create will also create the actual `learn` database. Now that you are inside a database, you can start issuing database commands, like `db.getCollectionNames()`. If you do so, you should get an empty array (`[ ]`). Since collections are schema-less, we won't explicitly need to create them. We can simply insert a document into a new collection. To do so, use the `insert` command, supplying it with the document to insert:

```
db.unicorns.insert({name: 'Aurora',
    gender: 'f', weight: 450})
```


The above line is executing `insert` against the `unicorns` collection, passing it a single parameter. Internally MongoDB uses a binary serialized JSON format called BSON. Externally, this means that we use JSON a lot, as is the case with our parameters. If we execute `db.getCollectionNames()` now, we'll see a `unicorns` collection.

You can now use the `find` command against `unicorns` to return a list of documents:

```
db.unicorns.find()
```

Notice that, in addition to the data you specified, there's an `_id` field. Every document must have a unique `_id` field. You can either generate one yourself or let MongoDB generate a value for you which has the type `ObjectId`. Most of the time you'll probably want to let MongoDB generate it for you. By default, the `_id` field is indexed. You can verify this through the `getIndexes` command:

```
db.unicorns.getIndexes()
```

What you're seeing is the name of the index, the database and collection it was created against and the fields included in the index.

Now, back to our discussion about schema-less collections. Insert a totally different document into `unicorns`, such as:

```
db.unicorns.insert({name: 'Leto',
```

```
      gender: 'm',
      home: 'Arrakeen',
      worm: false})
```

And, again use `find` to list the documents. Once we know a bit more, we'll discuss this interesting behavior of MongoDB, but hopefully you are starting to understand why the more traditional terminology wasn't a good fit.

```
    dob: new Date(1998, 2, 7, 8, 30),
    loves: ['strawberry', 'lemon'],
    weight: 733,
    gender: 'f',
    vampires: 40});
db.unicorns.insert({name:'Kenny',
    dob: new Date(1997, 6, 1, 10, 42),
    loves: ['grape', 'lemon'],
    weight: 690,
    gender: 'm',
    vampires: 39});
db.unicorns.insert({name: 'Raleigh',
    dob: new Date(2005, 4, 3, 0, 57),
    loves: ['apple', 'sugar'],
    weight: 421,
    gender: 'm',
    vampires: 2});
db.unicorns.insert({name: 'Leia',
    dob: new Date(2001, 9, 8, 14, 53),
    loves: ['apple', 'watermelon'],
    weight: 601,
    gender: 'f',
    vampires: 33});
db.unicorns.insert({name: 'Pilot',
    dob: new Date(1997, 2, 1, 5, 3),
    loves: ['apple', 'watermelon'],
    weight: 650,
    gender: 'm',
    vampires: 54});
db.unicorns.insert({name: 'Nimue',
    dob: new Date(1999, 11, 20, 16, 15),
    loves: ['grape', 'carrot'],
    weight: 540,
    gender: 'f'});
db.unicorns.insert({name: 'Dunx',
    dob: new Date(1976, 6, 18, 18, 18),
    loves: ['grape', 'watermelon'],
    weight: 704,
    gender: 'm',
    vampires: 165});
```


Now that we have data, we can master selectors. `{field: value}` is used to find any documents where `field` is equal to `value`. `{field1: value1, field2: value2}` is how we do an and statement. The special `$lt`, `$lte`, `$gt`, `$gte`

Update Operators

In addition to `$set`, we can leverage other operators to do some nifty things. All update operators work on fields - so your entire document won't be wiped out. For example, the `$inc` operator is used to increment a field by a certain positive or negative amount. If Pilot was incorrectly awarded a couple vampire kills, we could correct the mistake by executing:

```
db.unicorns.update({name: 'Pilot'},
    {$inc: {vampires: -2}})
```

If Aurora suddenly developed a sweet tooth, we could add a value to her `loves` field via the `$push` operator:

```
db.unicorns.update({name: 'Aurora'},
    {$push: {loves: 'sugar'}})
```

The Update Operators section of the MongoDB manual has more information on the other available update operators.

Upserts

One of the more pleasant surprises of using `update` is that it fully supports `upserts`. An `upsert` updates the document if found or inserts it if not. Upserts are handy to have in certain situations and when you run into one, you'll know it. To enable upserting we pass a third parameter to update `{upsert:true}`.

A mundane example is a hit counter for a website. If we wanted to keep an aggregate count in real time, we'd have to see if the record already existed for the page, and based on that decide to run an update or insert. With the upsert option omitted (or set to false), executing the following won't do anything:

```
db.hits.update({page: 'unicorns'},
    {$inc: {hits: 1}});
db.hits.find();
```

However, if we add the upsert option, the results are quite different:

```
db.hits.update({page: 'unicorns'},
    {$inc: {hits: 1}}, {upsert:true});
db.hits.find();
```

Since no documents exists with a field `page` equal to `unicorns`, a new document is inserted. If we execute it a second time, the existing document is updated and `hits` is incremented to 2.

```
db.hits.update({page: 'unicorns'},
    {$inc: {hits: 1}}, {upsert:true})
db.hits.find();
```

Count

The shell makes it possible to execute a `count` directly on a collection, such as:

```
db.unicorns.count({vampires: {$gt: 50}})
```

In reality, `count` is actually a `cursor` method, the shell simply provides a shortcut. Drivers which don't provide such a shortcut need to be executed like this (which will also work in the shell):

```
db.unicorns.find({vampires: {$gt: 50}})
    .count()
```

In This Chapter

Using `find` and `cursors` is a straightforward proposition. There are a few additional commands that we'll either cover in later chapters or which only serve edge cases, but, by now, you should be getting pretty comfortable working in the mongo shell and understanding the fundamentals of MongoDB.

Flexible Schema

An oft-touted benefit of document-oriented database is that they don't enforce a fixed schema. This makes them much more flexible than traditional database tables. I agree that flexible schema is a nice feature, but not for the main reason most people mention.

People talk about schema-less as though you'll suddenly start storing a crazy mishmash of data. There are domains and data sets which can really be a pain to model using relational databases, but I see those as edge cases. Schema-less is cool, but most of your data is going to be highly structured. It's true that having an occasional mismatch can be handy, especially when you introduce new features, but in reality it's nothing a nullable column probably wouldn't solve just as well.

For me, the real benefit of dynamic schema is the lack of setup and the reduced friction with OOP. This is particularly true when you're working with a static language. I've worked with MongoDB in both C# and Ruby, and the difference is striking. Ruby's dynamism and its popular ActiveRecord implementations already reduce much of the object-relational impedance mismatch. That isn't to say MongoDB isn't a good match for Ruby, it really is. Rather, I think most Ruby developers would see MongoDB as an incremental improvement, whereas C# or Java developers would see a fundamental shift in how they interact with their data.

Think about it from the perspective of a driver developer. You want to save an object? Serialize it to JSON (technically BSON, but close enough) and send it to MongoDB. There is no property mapping or type mapping. This straightforwardness definitely flows to you, the end developer.

This just barely scratches the surface of what you can do with aggregations. In 2.6 aggregation got more powerful as the aggregate command returns either a cursor to the result set (which you already know how to work with from Chapter 1) or it can write your results into a new collection using the `$out` pipeline operator. You can see a lot more examples as well as all of the supported pipeline and expression operators in the MongoDB manual.