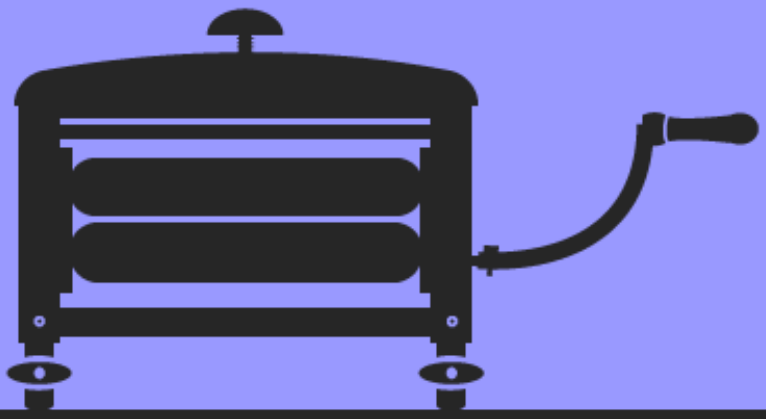


MongoDB 3

Succinctly®

by Zoran Maksimovic

Syncfusion | Technology Resource Portal

MongoDB 3 Succinctly

By
Zoran Maksimovic

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Zoran Maksimovic is a solution architect and software developer with over 16 years of professional experience. He is passionate about programming and web platforms, especially Microsoft technologies. Among other things, he is specialized and has interest in Microsoft.NET, OOD, TDD, DDD (Domain Driven Development), CQRS/ES, and event streaming. He is also a certified Scrum Master.

He spent most of his professional life working as a consultant on various projects for clients based in Switzerland, Italy, Germany, and France.

In his spare time, he contributes to his personal blog and is active on Twitter as @zoranmax.

Zoran is also the author of the *ServiceStack Succinctly* e-book.

When not coding or spending time with his family, he enjoys guitar playing, baroque music, good food, and Italian wine. He is married and the father of Alexei, Xenia, and Sofia.

For more info, visit http://zoran.me.

Introduction

Purpose of the book

The purpose of this book is to make you aware of the MongoDB NoSQL database so you can start working with this database in the fastest way.

My hope is that after reading this book, you will have enough knowledge to start coding and using the technology effectively.

Target audience

This book is intended for software developers who are already working with relational databases and want to know a bit more about MongoDB. Readers with a good understanding of database concepts, the Microsoft.NET Framework (most of the examples are written in C#), and JSON will benefit the most. Some shortcuts have been taken in this book, as the intention is not to go too deep into the content, but rather to show the various options, possibilities, and concepts.

Additional information and resources

Additional information about MongoDB can be found directly on the MongoDB website.

If you want to know more about the technologies mentioned in this book, see the following resources:

- Database
- JSON
- BSON
- NoSQL
- C# language

Source code

MongoDB is an open-source database written in C++, and at the time of writing, it's hosted here on GitHub.

MongoDB groups and communities

There are several groups on the web where additional information is hosted and shared, and common questions are answered. The following table contains a few that you might find useful.

Table 1: MongoDB additional information

MongoDB	https://www.mongodb.org/community
MongoDB University	https://university.mongodb.com
Twitter	https://twitter.com/mongodb
StackOverflow	http://stackoverflow.com/questions/tagged/mongodb
Google Plus	https://plus.google.com/communities/115421122548465808444
Meetup	http://www.meetup.com/pro/mongodb

Software requirements

To get the most out of this book and the included examples, you will need to have the Microsoft Visual Studio IDE installed on your computer, and Microsoft.NET v4 or later. At the time of writing, the most current available and stable edition of Visual Studio is Visual Studio 2015. You can download **Visual Studio Community Edition 2015** for free directly from the Microsoft website. Please pay attention to the licensing notes, as some restrictions might apply depending on the usage.

All of the examples in this book have been written and tested by installing MongoDB on Microsoft Windows 10 (on the local machine) and using Microsoft Visual Studio 2015.

Conventions used in the book

There are specific formats that you will see throughout this book to illustrate tips and tricks or other important concepts.

Note: This will identify things to note throughout the book.

Tip: This will identify tips and tricks throughout the book.

Source code

C# is used in a lot of the source code examples as follows:

Code Listing 1: C# code style

```
[SomeAttribute]
public class MongoDbConnector
{
    public string Connection { get; set; }
```

```
}
```

Or simply a command prompt (or terminal) code:

Code Listing 2: Command prompt code style

```
C:\mongodb\bin> mongod
```

Resources

The code mentioned in this book can be checked out from this website.

MongoDB version

All of the examples and explanations apply to version 3.4.1 of MongoDB, which is the latest stable version at the time of writing.

Chapter 1 MongoDB Overview

MongoDB is an open-source *document database* that provides high performance, high availability, and automatic scaling. MongoDB is available under the General Public License (GPL) for free, and it's also available under commercial license as part of the commercial offering of the company. In this book, we will discuss the functionalities offered by the free version.

MongoDB is one of many implementations of the so-called NoSQL databases, and it's currently one of the biggest players in this segment of the market.

NoSQL and document databases

If you are new to the NoSQL world, we need to introduce a few concepts.

Broadly speaking, there are three categories of databases:

- RDBMS (relational database management system)
- OLAP (online analytical processing)
- NoSQL

As developers, we are typically more familiar with the relational databases, such as Microsoft SQL Server, Oracle, MySQL, and Postgres, and with the way those databases organize data in a tabular format. It is also true that, historically, relational databases are most widely used, especially in the corporate world.

NoSQL (originally referring to "non SQL," "non-relational," or "not only SQL") is another type of database that offers a mechanism for storing and retrieving data, and it usually handles data in a different way than relational databases.

NoSQL databases exist in order to solve particular problems for particular domains, and are not a "silver bullet" for any kind of issue, as they have their pros and cons. Some of the main problems that NoSQL databases try to solve are issues of scalability and quantity of data.

In regards to the CAP theorem, NoSQL databases often compromise consistency in favor of availability and partition tolerance. In the NoSQL world, the "eventual consistency" is often used to achieve speed and scalability.

Scalability

One of the advantages the NoSQL databases have is the support for *horizontal* scalability (or *scaling out)*, which is available—but more limited and expensive—in the RDBMS systems. Horizontal scalability means that we can expand the capacity of the system by adding more servers (nodes). The performance is then almost linearly proportional to the number of nodes that are part of the system.

This idea of horizontal scalability is different from *vertical scalability*, where typically in order to handle more data, we are upgrading the server itself by adding more memory, HDD space, CPU, etc.

Scaling out is generally the cheaper and more flexible choice because it uses regular commodity hardware, while scaling up is typically much more expensive because the cost of the hardware tends to exponentially increase as it becomes more sophisticated, and in the end its expansion has more limitations.

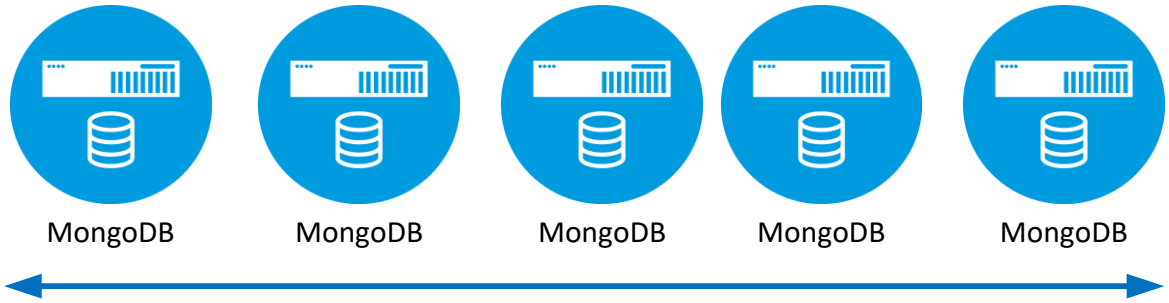


Figure 1: Horizontal Scalability

Implementations

NoSQL databases include a wide variety of implementations (typically not encompassing the tabular format) that were developed in response to a rise in the volume of data. Listed here are the various flavors:

- **Document databases** pair each key with a complex data structure known as a document (MongoDB, Couchbase Server, CouchDB, RavenDB, and others).
- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value (DynamoDB, Windows Azure Table Storage, Riak, Redis, LevelDB, Dynomite).
- **Wide-column** stores such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.
- **Graph** stores are used to store information about networks, such as social connections. Graph stores include Neo4J and HyperGraphDB.

NoSQL: What is missing?

Compared to the RDBMS, usually in the NoSQL databases there is no or little support for the following:

- **Limited or no support for JOINS (INNER, OUTER, etc**.): The access to the data is done at the document level, and therefore the handling of the links between objects has to be done at the application level.

- **No complex transactions support**: NoSQL databases are often supporting eventual consistency transactions, and are typically not supporting batches of updates, but work on single items.
- **No support for constraints**: Constraints are not implemented at the database level, but at the application level.

Database structure

MongoDB, as we mentioned previously, is a document database, and it's quite simple when it comes to the data representation. The database in its simplest form consists of two items: **`document`**, which contains data, and **`collection`**, which is a container of documents.

Documents

A **`document`** is a data structure composed of **`field`** and **`value`** pairs. **`Document`** is basically a JSON object that MongoDB stores on disk in binary (BSON) format.

Figure 2 shows an example of a **`document`** representing a user. It is not different from any JSON representation, so you should be familiar with the format. As we are going to see later, there are some conventions used, such the **`_id`** field, which is the primary key of this document, and in that sense, **`<User1>`** is simply a value of the primary key.

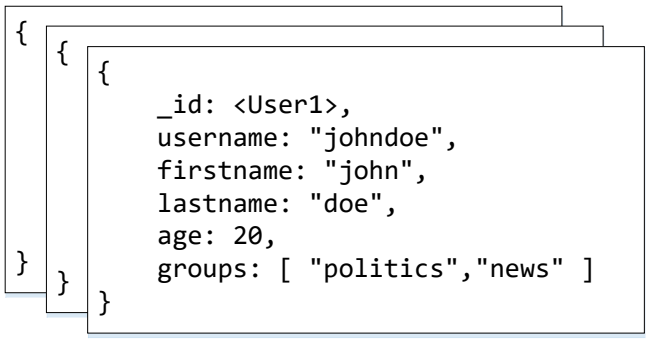


Figure 2: Example of a very simple document.

BSON is a binary-encoded serialization representation of the JSON. However, BSON supports more data types than JSON (for example, the **`Date`** type), and it can be compared to Google's ProtoBuf. (You can find more information about the BSON format here.)

Tip: In the RDBMS world, we can think of a document as representing a "record" of a table.

There is one hard limit of the document size, which is a maximum of 16 megabytes. This limit hasn't changed, even on the latest version of the database. That makes sense, as the maximum size limitation ensures that a single document cannot use an excessive amount of RAM or bandwidth. To store documents larger than the maximum size, MongoDB provides the **`GridFS`** API, which will be discussed in Chapter 10.

Collections

MongoDB stores documents in collections. A collection can be seen as analogous to a table in RDBMS. Every document in the collection, unless otherwise specified, has an **`_id`** automatically assigned by the database. One thing to note is that the collection is not like a table in which the set of columns (attributes) has to be predefined; collections are schema-less; therefore, a collection can contain any kind of content. However, it is not very practical to have disparate sets of data all in one collection (as this is technically possible), unless in some very particular use cases (data collection, logs, etc). Typically, what happens is that the objects are serialized at the application level and then stored in the database. Therefore, even though the schema is not enforced, some sort of control over the data in a collection will exist.

Thinking in documents

One of the biggest and most fundamental differences between relational databases and MongoDB is data modeling and the way to represent the structure of the data.

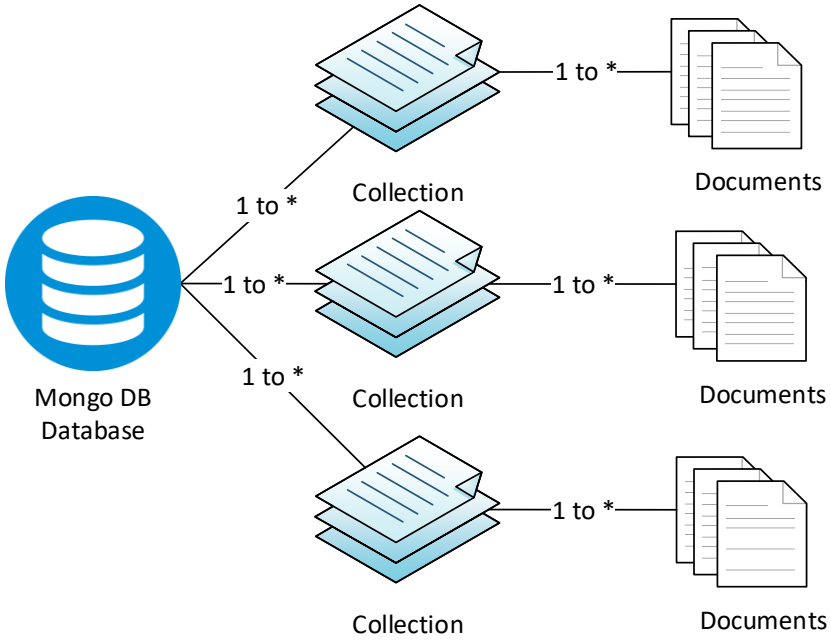


Figure 3: MongoDB data structure organization.

In MongoDB, data relationships can be represented either by embedded documents or by references. References pretty much correspond to the usage of foreign keys in RDMBS; however, the support for joins in MongoDB exists, but is quite limited.

Technically there is a way to join the two collections by using the **`lookup`** functionality in the aggregation framework, or via **`LINQ`** queries (which underneath use the aggregation framework in order to construct queries).

Referencing documents

Referencing documents can be seen as a standard way of normalizing data in the RDMBS, where the tables are linked by the foreign key. MongoDB, in this sense, is not any different.

In a nutshell, by normalizing data into individual collections, we are able to link the data in a very efficient manner by using the primary key (as a foreign key).

Let's consider the example shown in Figure 4, where we have a user document linked to an address and to a contact. We can see how the **`user_id`** (primary key of the user document) is used to link the documents together.

By using this way of linking collections together, we are forced to issue multiple queries in order to retrieve information, as there is no equivalent way of joining information together as we would in a RDMBS. (However, there is the **`$lookup`** command, which acts as a **`LEFT JOIN`**, introduced in version 3.2 of MongoDB.)

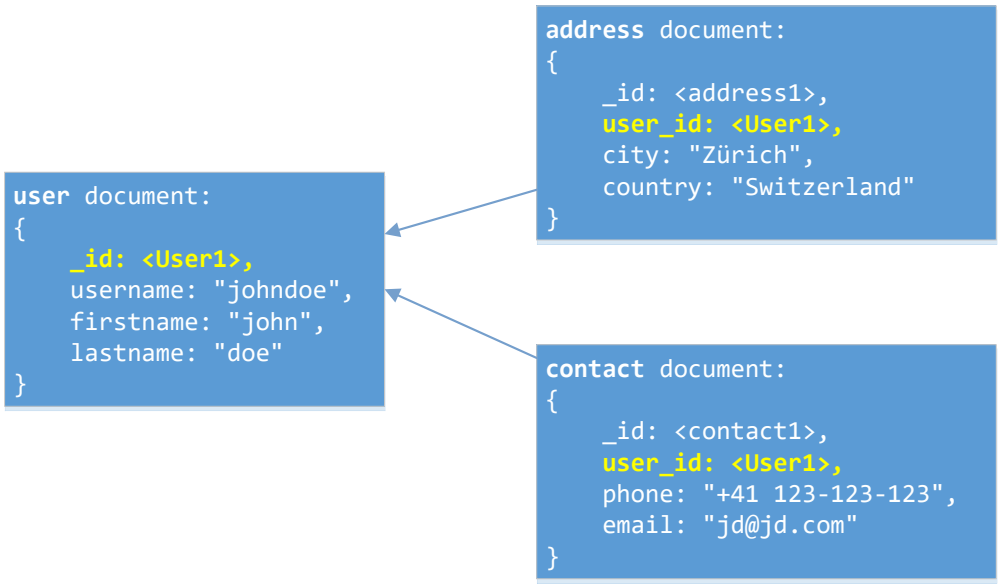


Figure 4: Referencing documents.

Embedding documents

By embedding documents, we are able to concatenate all the content into one document.

The same example can be represented simply, as shown in Figure 5. MongoDB offers a way to update the address or contact information directly, but this also means issuing an update to a document.

Figure 5: Embedding documents.

Document design strategy

As we have seen, there are mainly two ways of linking the documents. However, the need for one or the other would have to be carefully weighed, as it obviously can have some side effects. Here are some recommendations to follow:

- **Embed as much as possible:** The document database should eliminate quite a lot of joins, and therefore, the option we have is to put as much as possible in a single document. This way, the advantage is that saving and retrieving a document is atomic and very fast. There is no need to normalize data. Therefore, embed as much as possible, *especially the data that is not being used by other documents*.
- **Normalize:** Normalize data that can be referred to from multiple places into its own collection. This means creating reusable collections (for example, **`country`** or **`user`**). This is a more efficient way to handle duplicate values in only one place.
- **Document size:** The maximum document size in MongoDB is 16 MB. The limit is imposed mainly in order to ensure that a single document cannot use an excessive amount of RAM or bandwidth. This is quite a large quantity of text data (just think how much data is usually displayed on a single web page). In most cases, this limit is not a problem; however, it's good to keep it in mind and avoid premature optimizations.

- **Complex data structures and queries**: MongoDB can store arbitrary, deep-nested data structures, but cannot search them efficiently. If your data forms a tree, forest, or graph, you effectively need to store each node and its edges in a separate document.

- **Consistency**: MongoDB makes a trade-off between efficiency and consistency. The rule is that changes to a single document are atomic, while updates to multiple documents should never be assumed to be atomic. When designing the schema, consider how to keep your data consistent. Generally, the more that you keep in a document, the better, as stated in the first point of this list.

Pluggable storage engine

As modern applications need to support a variety of workloads with different price and performance profiles—from low-latency, in-memory read-and-write applications, to real-time analytics—MongoDB started offering support for pluggable storage engines to achieve the goal of having the same programming API model, but with different implementations.

At the time of writing, MongoDB supports the following engines:

- **MongoDB built-in engine**: MMAPv1 engine, which is an improved version of the engine used in prior MongoDB releases.
- **MongoDB built-in default engine**: The new WiredTiger storage engine, which provides significant benefits in terms of lower storage costs (better compression), greater hardware utilization, higher throughput, and more predictable performance than the related MMAP engine. Some benchmarks are showing from 7-10x higher performance of this engine.
- **MongoDB engine (only enterprise edition)**: The in-memory storage engine designed to serve ultra-high throughput.
- Facebook is supporting MongoRocks, a MongoDB storage engine based on Facebook's RocksDB-embedded database project.

With these choices in mind, developers can choose the appropriate engine based on their application needs.

Sharding

We have seen that one of the advantages of the NoSQL database is the ability to scale horizontally, and the technique used in MongoDB is *sharding*.

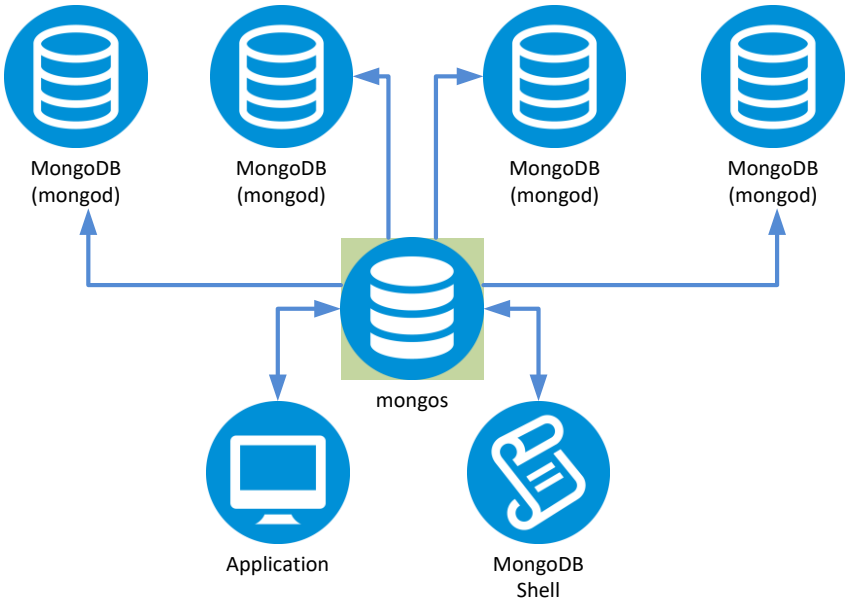


Figure 6: Sharding.

`Sharding` is a type of database partitioning that separates large databases the into smaller, faster, more easily managed parts (data shards).

In other words, instead of having to run a huge database on one server, MongoDB offers the ability to separate the load and partition (divide) the data into smaller chunks that could run independently on their own. When writing and reading data from the database, the MongoDB engine will make sure the data gets collected or distributed to the nodes in question accordingly.

Conclusion

In this chapter, we have seen what MongoDB is and how it correlates to the relational database. We looked at the database structure, and how the data gets organized within the database. The emphasis has been placed on the fact that documents are quite different from the normal tabular form.

In the end, we saw how to install the database and which tools come as part of the database. We are now ready to start using the database and its features.

Chapter 2 MongoDB Installation

MongoDB is a cross-platform database, and the current version supports a variety of operating systems—pretty much all main Linux distributions, Microsoft Windows, and OSX.

As the idea of this book was to use Microsoft.NET and MongoDB, we will only briefly touch on Linux installation as a reference, and we will mainly concentrate on running and configuring MongoDB on Microsoft Windows. While all concepts apply equally on every platform, it's not in the scope of this book to get into particular platform details.

For Mac and Windows platforms, you can download MongoDB and install it directly from the project website.

In this book, we will be covering the MongoDB Community Server, which is a free version of MongoDB.

Installation on Windows (single node)

Examples in this book were tested in Microsoft Windows 10; the exact version downloaded and used is version 3.4.1 for "Windows Server 2008 R2 64-bit and later, with SSL support x64."

MongoDB can run as a standalone application or as a Windows service. For learning purposes, it is interesting to see MongoDB running as the standalone application (console), as it outputs some debug information that can be useful. Otherwise, on production systems, it's highly recommended to run it as a Windows service.

MongoDB installation

1. Install the MongoDB by double-clicking on the downloaded file. Follow the installer instructions and choose the custom installation.
    a. Install MongoDB in the **c:\mongodb** folder. (You can choose any folder you like; for this book's purpose, I've chosen this path.)
    b. Choose all the features available.
2. We need to manually create the directory where the database data will reside. The default location where MongoDB will place the files is c:\data\db, so, let's just keep these default settings and create the folder by running **`md c:\data\db`** on the command line. The location of this folder can be changed by using the **`--dbpath`** option when starting the database.
3. The same thing applies to the application logs. The default place that MongoDB will be looking at is c:\data\log. So, let's run the **`md c:\data\log`** on the command line. This option can be controlled by using the **`--logpath`** option. Let's use the default and create this folder.

Running MongoDB as a standalone application

Starting the database is as simple as running the following command.

Code Listing 3: Starting MongoDB

```
C:\mongodb\bin>mongod
```

After running this command (which stands for “mongo daemon”) for the first time, you might be asked to open the firewall. Accept this option.

The output will be similar to the following in Figure 7, from which we may learn quite a few things (as highlighted).

- MongoDB operates by default on port: 27017.
- The database engine used by default is **`wiredTiger`**.
- Logs are enabled.
- Last message: “waiting for connections on port 27017” means that the database is ready to be used.

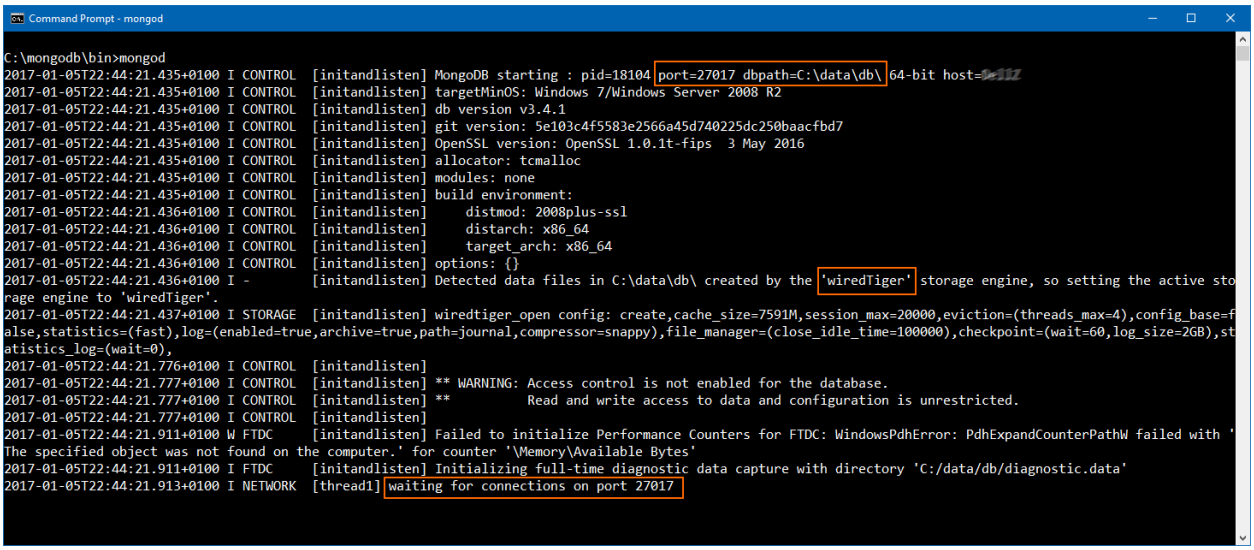


Figure 7: Starting MongoDB.

Installing MongoDB as a Windows service

By running the following command, where we specify explicitly where to place the logs, we will install MongoDB as a Windows Service. Make sure to run the command prompt with administrator rights.

In order to install MongoDB on Windows as a service, keeping the previously used settings is as simple as adding the **`--install`** parameter and the location of the log specified by the value of **`--logpath`**.

Code Listing 4: Installing MongoDB as a Windows service

```
C:\mongodb\bin>mongod --logpath c:\data\log\log.log --install
```

If the installation is successful, you will see MongoDB in the Windows Services panel, as shown in Figure 8.

For more information about installing MongoDB on Windows with the full option list, see this tutorial.

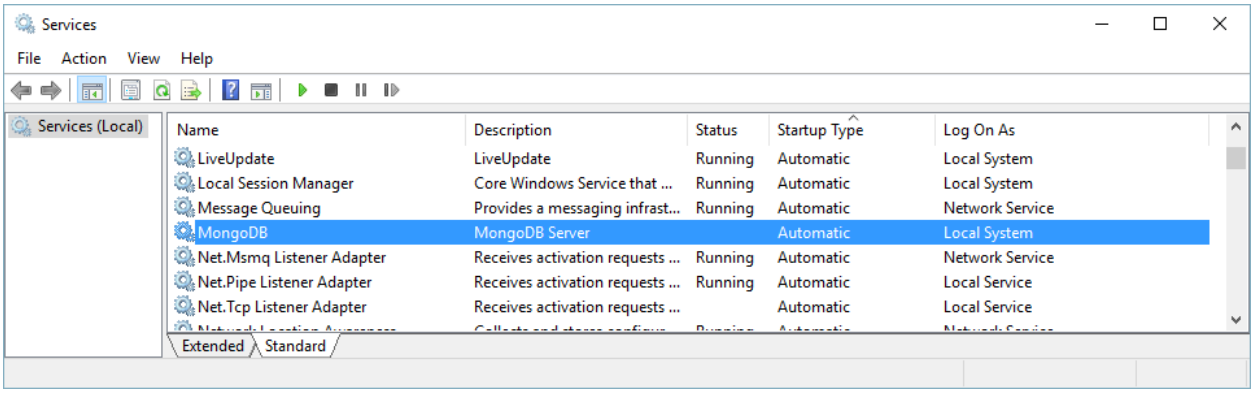


Figure 8: MongoDB service after complete installation.

Single node installation on Linux (Ubuntu)

Here we will demonstrate how to install MongoDB on Ubuntu (version 16.04 at the time of writing) by using the popular **apt** package management tool.

Using the terminal window, let's follow the following steps:

1. Import the public key used by the apt package manager.

Code Listing 5: Importing the public key

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
```

2. Create a list file for MongoDB.

Code Listing 6: List file creation

```
echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
```

3. Reload local package database.

Code Listing 7: Reload local package database

```
sudo apt-get update
```

4. Install the MongoDB package.

At this step, after preparing the **apt** package manager, we can install the latest version of MongoDB by running:

Code Listing 8: Install MongoDB package

```
sudo apt-get install -y mongodb-org
```

Or, alternatively, install a specific version of MongoDB (in our case, version 3.4.10).

Code Listing 9: Installing a specific version of MongoDB

```
sudo apt-get install -y mongodb-org=3.4.10 mongodb-org-server=3.4.10
mongodb-org-shell=3.4.10 mongodb-org-mongos=3.4.10 mongodb-org-tools=3.4.10
```

5. Make sure the file **/lib/systemd/system/mongod.service** is present and that it contains the following content:

Code Listing 10: MongoDB configuration file content

```
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target
Documentation=https://docs.mongodb.org/manual

[Service]
User=mongodb
Group=mongodb
ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf

[Install]
WantedBy=multi-user.target
```

6. At this point, we can start MongoDB and test that it works correctly.

Code Listing 11: Starting the MongoDB

```
sudo service mongod start
```

Let’s check to see if it’s possible to connect to the database:

Code Listing 12: Starting the client via terminal

```
mongo
```

This command should successfully connect to the “test” database.

What comes with the MongoDB installation?

We have just seen how to install MongoDB, together with the database itself. There are also a few extra components that are part of the installation package. In this chapter, we are going to go through the list and explain briefly what each of them does.

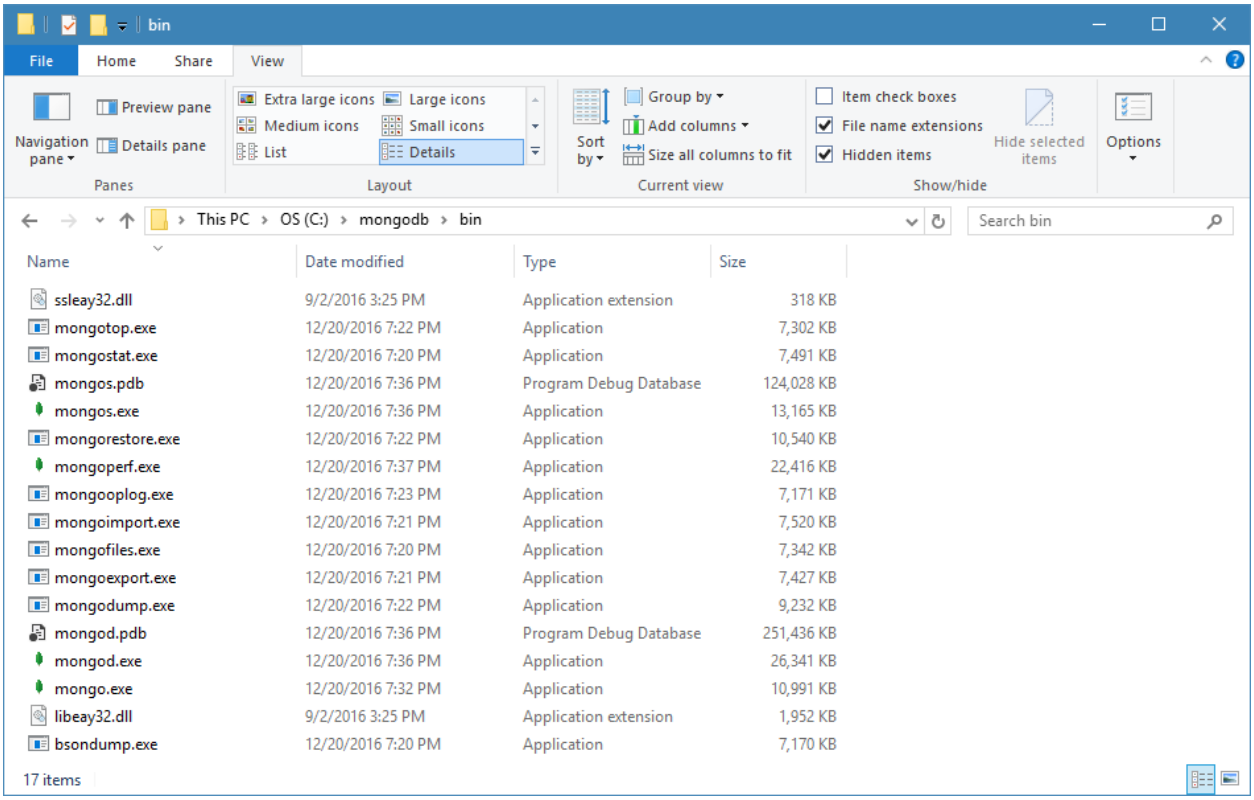


Figure 9: MongoDB installed utilities.

Don't worry if you're not familiar with some of the concepts (such as **`GridFS`**) that we haven't touched yet—we'll be looking into those in the next chapters.

Table 2: MongoDB installation files

Component	Category	Description
mongod	Core Process	The core database process. It is responsible for running (hosting) the **`mongodb`** instance on a single machine. This process is responsible for handling connections, requests, data access, and any other operations typical of a database.
mongos	Core Process	Short for "MongoDB Shard," **`mongos`** is a service that sits between the application and the MongoDBs. It communicates with the configuration server to determine where the

Component	Category	Description
		requested data lives (on which shard). It then fetches, aggregates, and returns the data in JSON form.
mongo	Core Process	An interactive JavaScript shell interface to MongoDB that provides an interface for system administrators and allows any kind of operation against the MongoDB instance.
mongodump	Binary Import/Export	A utility for creating a binary export of the contents of a database.
mongorestore	Binary Import/Export	Writes data from a binary database dump created by **mongodump** to a MongoDB instance.
bsondump	Binary Import/Export	**bsondump** is a diagnostic tool for inspecting BSON files, and it converts BSON files into human-readable formats, including JSON, which is very useful for reading the output files generated by **mongodump**.
mongooplog	Binary Import/Export	A simple tool that polls operations from the **replication oplog** of a remote server, and applies them to the local server.
mongoimport	Data Import/Export	Imports content from a JSON, CSV, or TSV export created by **mongoexport**.
mongoexport	Data Import/Export	Produces a JSON or CSV export of data stored in a MongoDB instance.
mongostat	Diagnostics	Provides a quick overview of the status of a currently

Component	Category	Description
		running **mongod** or **mongos** instance. It allows the monitoring of the MongoDB. It provides information such as the server status, database statistics, and collection statistics.
mongotop	Diagnostics	Another monitoring tool that provides the method to track the amount of time a MongoDB instance spends reading and writing data. It provides statistics per collection.
mongoreplay	Diagnostics	A traffic capture and replay tool for MongoDB that you can use to inspect and record commands sent to a MongoDB instance, and then replay those commands back onto another host at a later time.
mongoperf	Diagnostics	A utility that checks disk I/O performance independently of MongoDB.
mongofiles	GridFS	Manipulates files stored in the MongoDB instance in **GridFSobjects** from the command line. It is particularly useful, as it provides an interface between objects stored in your file system and **GridFS**.

Chapter 3 The Mongo Shell

Some of the first things a developer might try to find out is how to run queries and administer the database.

MongoDB comes with a set of utilities, all of them installed in the **bin** folder. One of the utilities is **mongo.exe**, also known as the "mongo shell," which is an interactive JavaScript interface to the database. The mongo shell is used to query or manipulate data, as well as to perform administrative operations.

To start the mongo shell, we have to simply type **mongo** in the command prompt. Obviously, in order for the shell to work, the database has to be up and running.

Code Listing 13: Starting the Mongo Shell

```
C:\mongodb\bin>mongo
MongoDB shell version v3.4.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.1
```

Once in the shell, we are automatically connected to the database instance on the local machine (localhost), and by definition the default port 27017 is used.

Tip: By including the** c:\mongodb\bin **directory in the system variable path, referencing mongodb utilities is much easier across the system.

If the database is not located on the local machine (which will almost always be the case in any production system), we can specify the remote server name (**--host**), port, and the username and password, as shown in the example that follows:

Code Listing 14: Connecting to the MongoDB remote server

```
C:\mongodb\bin>mongo --host <remoteServerName> --port 27017 –-u <username>
--p <password>
```

By default, MongoDB has no default user or password. These can be created with the **db.createUser** command.

Searching for help

One of the first things that can be done in the mongo shell is to check for help, which is available by simply typing **help** in the shell. Help will bring the list of available commands on the various objects such as database, collection, users, etc.

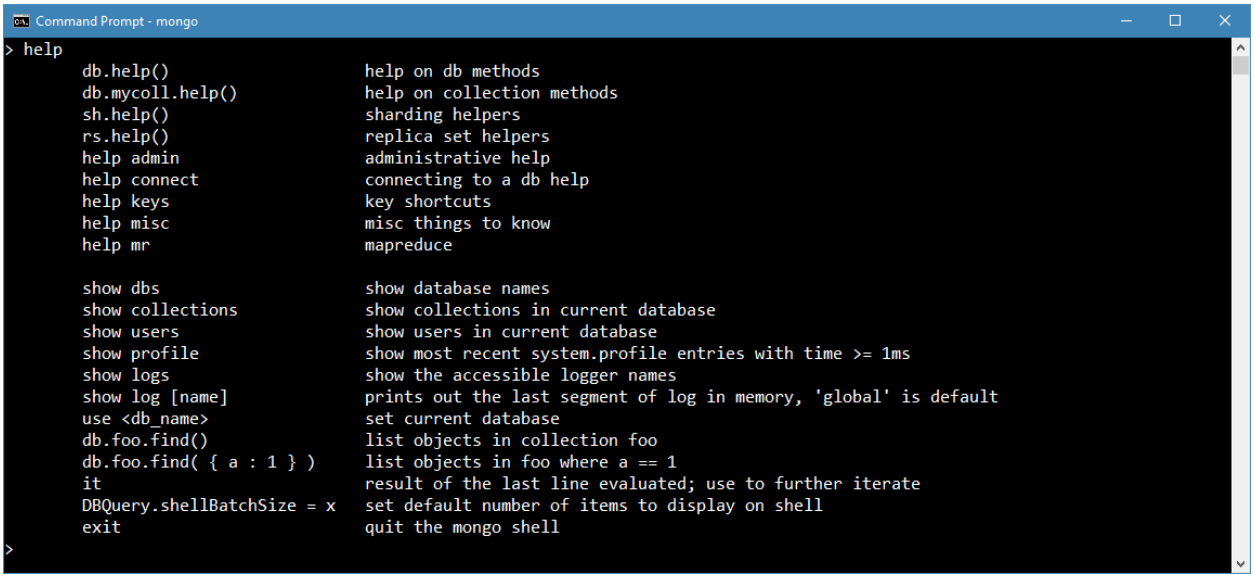


Figure 10: Showing help.

When you type **`db.help()`**, all of the operations available on the database object will be shown. The same logic would apply for other objects as well.

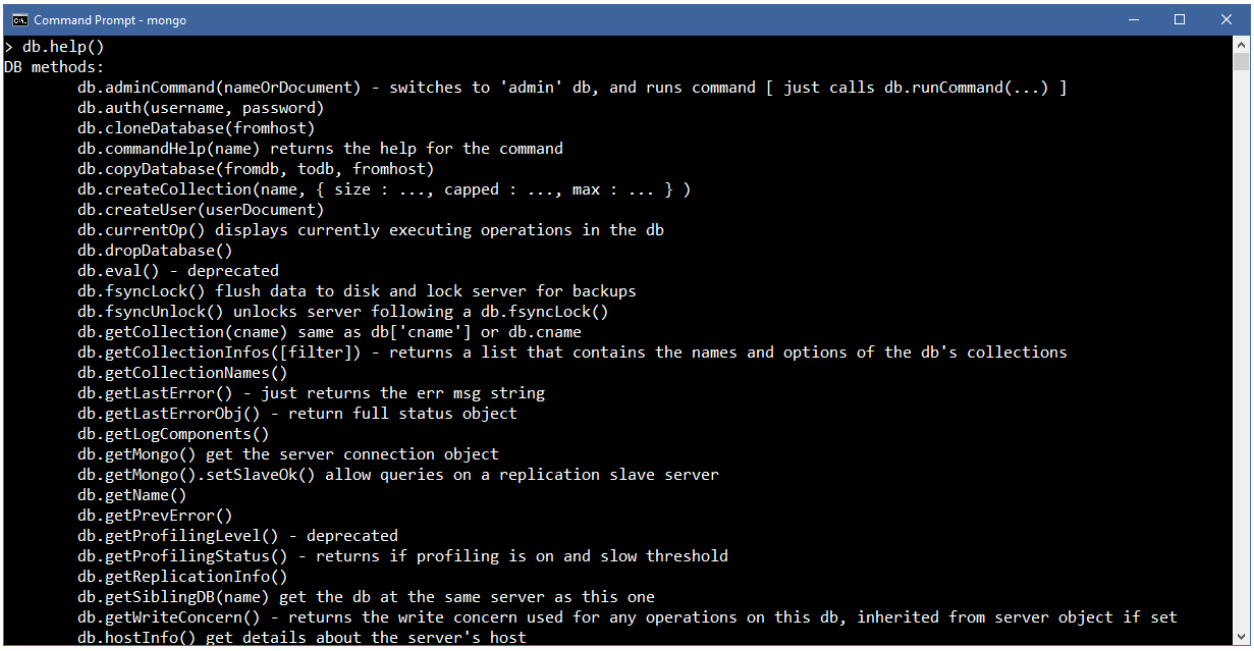


Figure 11: Showing help at the database level.

Databases

The MongoDB shell doesn't explicitly provide commands for creating databases. When you insert at least one document into a collection, MongoDB will create a database if one doesn't exist.

First, let's see how to get the list of databases in MongoDB by running the **`show dbs`** command:

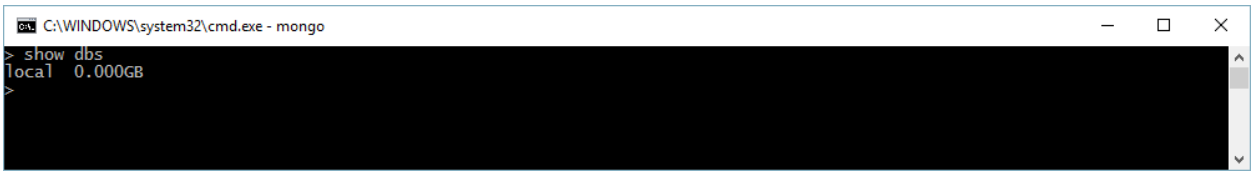


Figure 12: Getting the list of databases.

In the freshly installed MongoDB, there will be only one database, called **local**. Every **`mongod`** instance has its own **local** database, which stores data used in the replication process, and other instance-specific data.

You might also see an **`admin`** database being shown, depending on whether the security has been applied. The **`admin`** database will hold the information about users and passwords. In my case, there is no security applied.

Database creation

Just to play a bit with the shell, we will create a database called **`mydb`** by trying to insert a document. This action will make MongoDB create a database. To navigate to our new database, use the **`use mydb`** command. Even though the database doesn't exist yet, the shell will allow us to switch to it.

So, as an exercise, let's insert a fictitious user entry into the **`mydb`** database in the **`users`** collection:

Code Listing 15: Inserting a new document

```
db.users.insert({firstname: 'john', lastname: 'doe'})
```

As we can see in Figure 13, the result shows **`WriteResult({"nInserted" : 1 })`**, which tells us that one row has been written. All good so far.

Note: Inserting documents will be further explained in one of the following chapters.

As the last step, let's check to see if the database is now being created by running **`show dbs`** once again:

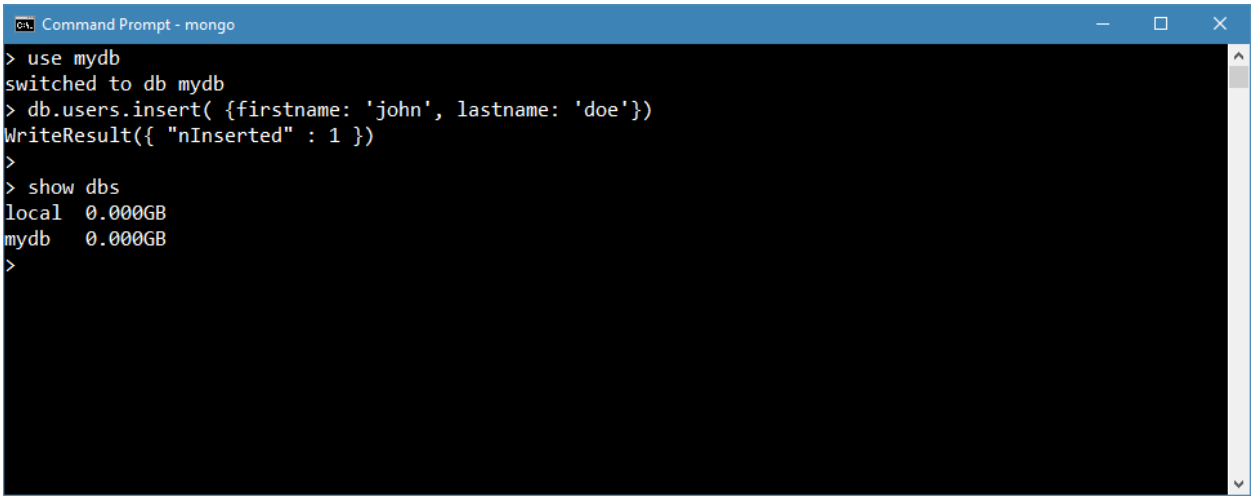


Figure 13: Creation of the database by inserting data into collection.

Congratulations, you successfully created your first MongoDB database!

Dropping databases

Sometimes there is a need to delete a database that has already been created. The MongoDB **`db.dropDatabase()`** command is used for this purpose.

In order to delete a database, we have to call the **`use database`** command. Once the shell switches to the selected database, we can call the **`db.dropDatabase()`** command, which will effectively delete the database.

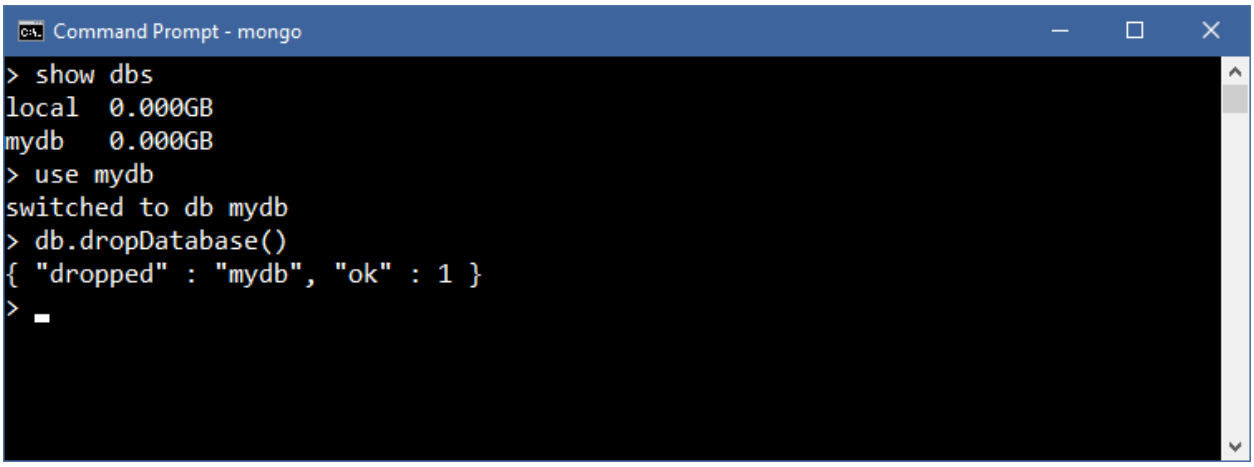


Figure 14: Deleting a database.

As in the previous case when we inserted an entry in the collection, the shell returns the result of the command, telling us that the command succeeded.

Collections

When we created the database, we inserted the first document in the **`users`** collection. To show that the **`users`** collection is there, we can use the command **`show collections`**, which is very similar to the one used to show the database list.

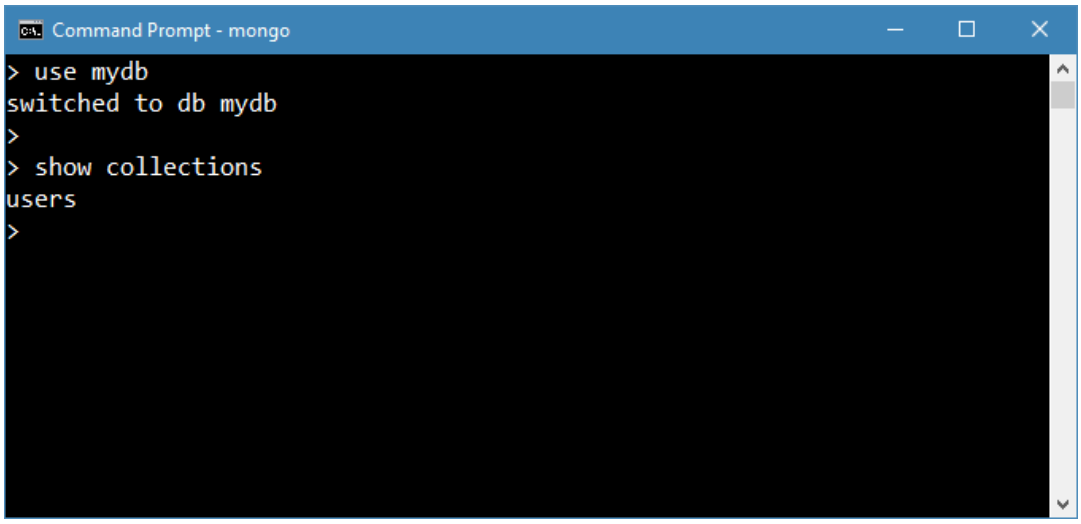


Figure 15: Showing the list of collections.

We can see that the **`users`** collection is available.

An alternative to this would be to run the **`db.getCollectionNames()`** command, which then returns a BSON document.

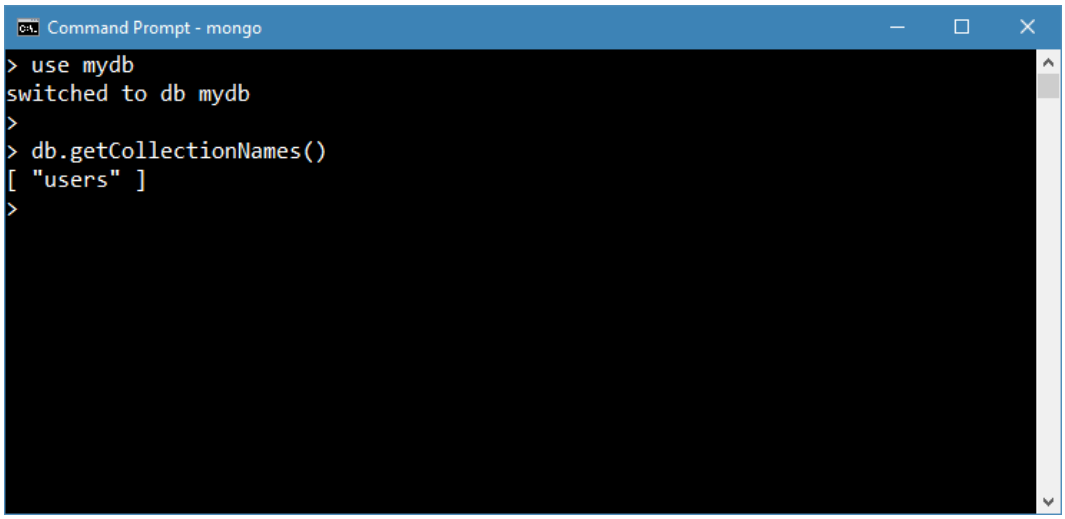


Figure 16: Alternative way of showing collections.

To create a collection, we can simply run the following command (the semicolon terminator is optional for single commands):

Code Listing 16: Creating Collections signature

```
db.createCollection(name, options);
```

Name is a mandatory parameter, while **options** is not. There are various options that can be set, such as **capped**, **autoIndexId**, and **size**; we will be looking into those in the next chapter.

In order to create the collection called **person**, we need to run the following command:

Code Listing 17: Create Collection called person

```
db.createCollection(“person”);
```

And to drop a collection, we would simply call **drop**, which will empty the collection and completely delete it from the database.

Code Listing 18: Dropping a person collection

```
db.person.drop();
```

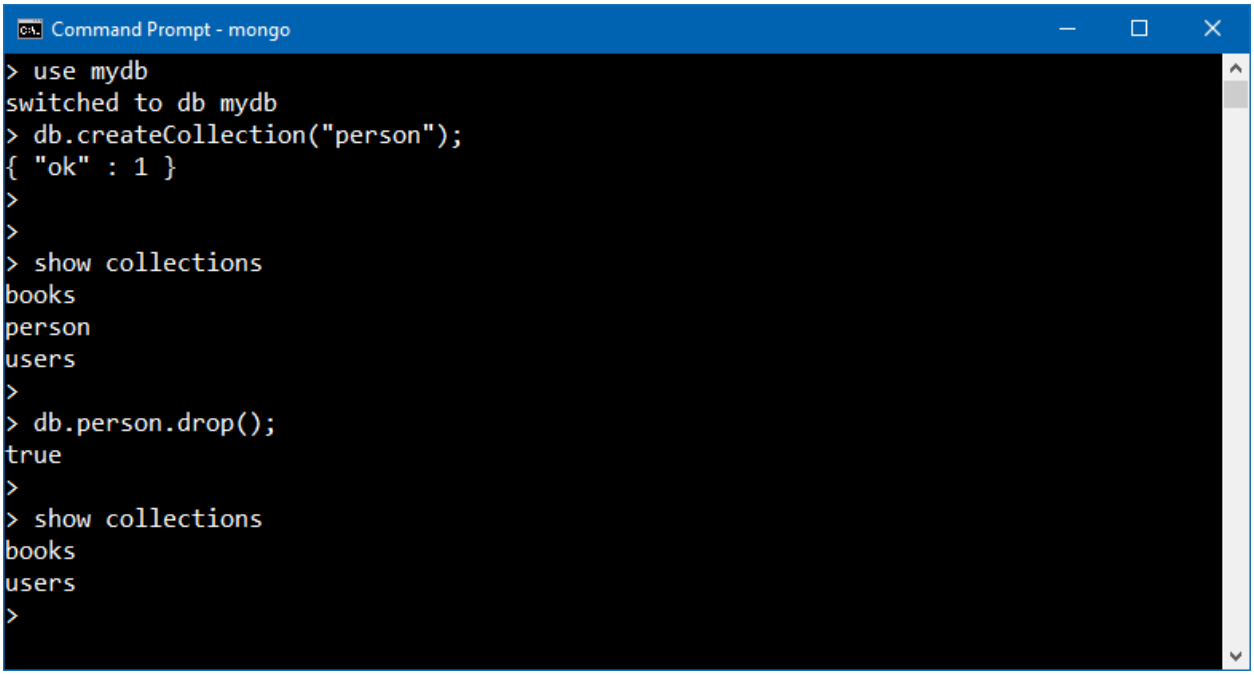


Figure 17: Creating and dropping a collection.

Tip: As a reminder: if you need to know which operations are available when working with collections, you can always refer to the help by using the `db.<collectionName>.help()` command, which will return a list of available commands.

Capped collections

Capped collections are a special kind of collection: fixed-size, circular collections.

- *Fixed-size* refers to the fact that there is a predefined (configurable) limit on the maximum number of items this table will support.
- *Circular* refers to the fact that once the maximum amount is reached, the oldest of the items gets deleted to make room to the new one.

One of the interesting properties of capped collections is that the collection itself preserves the order in which the items get inserted. This is a very important aspect, especially if this kind of table gets used for a logging-type of problem where the order of entries should be preserved. On the other hand, it has some limitations: we cannot remove a document from a capped collection, and updates on the documents won't work if an update or a replacement operation changes the document size.

Graphically, we can represent a capped collection as seen in Figure 18.

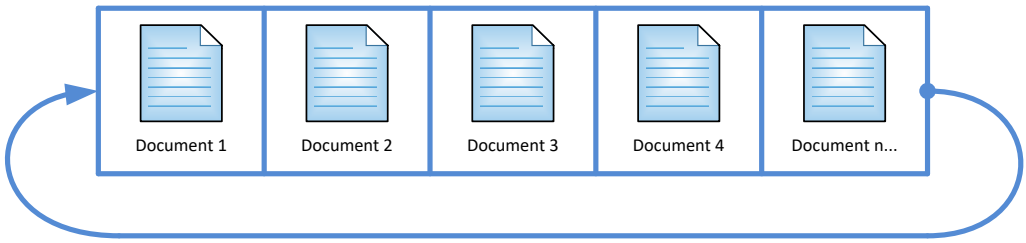


Figure 18: Capped Collection.

Capped collections can be used for several purposes, such as:

- Logging (for example, the latest activity performed on the website).
- Caching (preserving the latest items).
- Acting as a queue: Capped collection might be also used to act as a queue where the first-in-first-out logic applies.

Creating a capped collection

Capped collections are created in the same way as normal collections, but specifying the options parameter:

Code Listing 19: Creating capped collections

```
db.createCollection(“LogCollection”, { capped:true, size:10000, max:1000}) ;
```

There are 14 optional parameters. Three of the most common are:

- **`Capped`**: **`True`** sets the type of the collection as capped (the default is **`false`**).
- **`Size`**: Sets the maximum size in bytes for this particular collection.
- **`Max`**: Specifies the maximum number of documents allowed for the given collection.

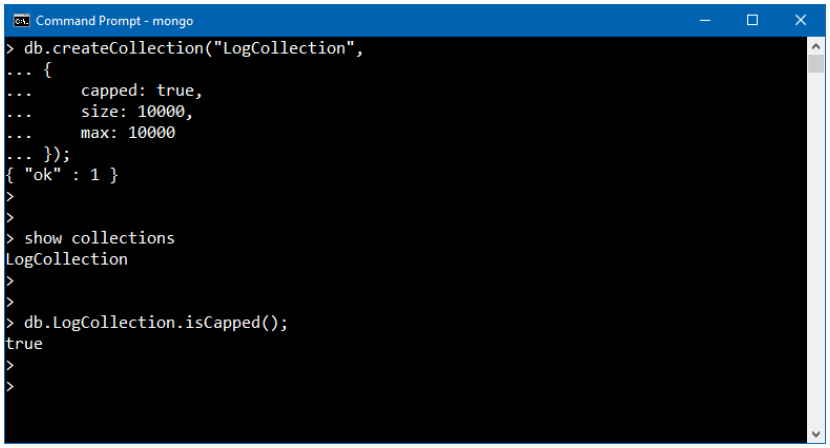


Figure 19: Creation of a capped collection.

Conclusion

In this chapter, we have seen what the MongoDB shell is and how to perform basic operations, such as the creation of a database or collections, with some more detail on collections themselves.

In the following chapters, we will see how to use the MongoDB shell further.

Chapter 4 Manipulating Documents

Now that we are able to find out the available collections, let's take a look at the following commands, which are used for manipulating the documents:

- **Insert**
- **Update**
- **Remove**

Simple data retrieval

Before we actually start with these operations, let's look at how to query the database in the simplest terms, as this method will be used as an example in upcoming chapters.

The MongoDB shell offers the ability to query for data; this is mainly achieved by using the **db.<collection>.find()** method.

The **db.collection.find()** will retrieve all the documents in a given collection (actually, the top 20). For more information, please see the full chapter on how to find and project data in MongoDB.

Inserting a document

There are several ways to insert data into a MongoDB collection.

Table 3: Methods of document creation

db.<collection>.insert()	Inserts a document or collection of documents into a collection. Returns a **BulkWriteResult** object back to the caller.
db.<collection>.insertOne()	New in v3.2. Inserts a single document into a collection.
db.<collection>.insertMany()	New in version 3.2. Inserts multiple documents into a collection. Returns a document containing the object IDs and information if the insert is acknowledged.
db.<collection>.save()	Updates an existing document or inserts a new document, depending on its **document** parameter. Returns a **WriteResult** object.

The goal in the following example is to add a new user in the **users** collection.

As we have seen, it's fairly easy to insert a simple document representing the **`user`**.

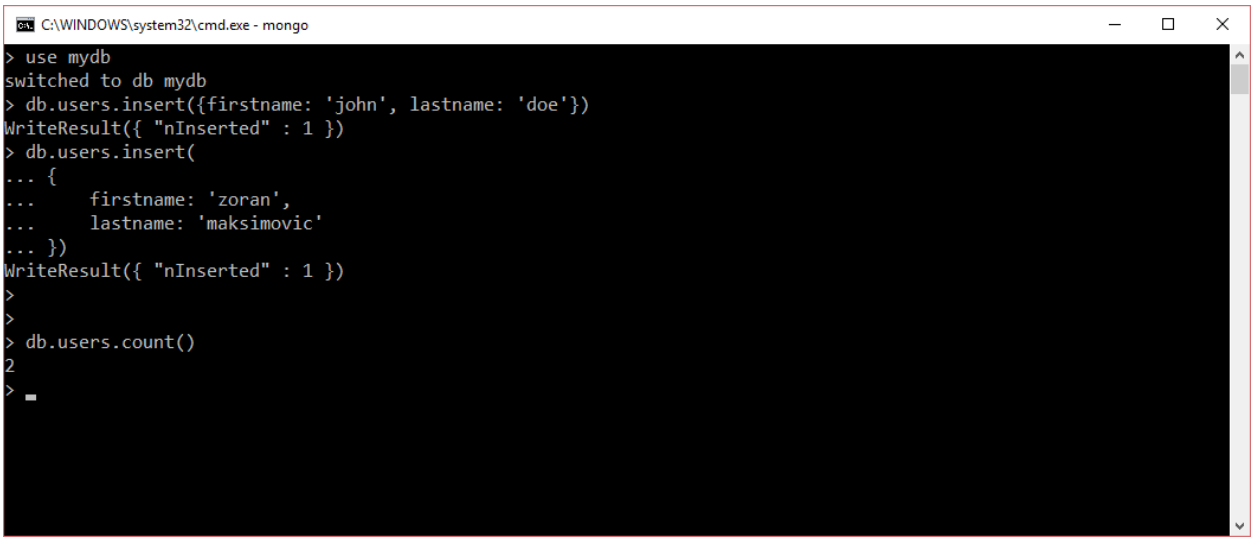


Figure 20: Inserting a user.

Using a single line vs multiline when specifying a JSON string doesn't really make any difference. The important point is to close the function with the right parenthesis. As soon as the document is inserted, the MongoDB shell informs us with the number of affected rows in the form of **`WriteResult ({ "nInserted": 1 })`**.

Just to prove that we have inserted two users, we can call **`db.users.count()`**, which will return the count of documents currently present in the **`users`** collection.

It is also possible to insert more than one document at a time, by using an array of JSON documents and passing this as a parameter to the **`insert`** function. In JSON, brackets [ ] are used to specify an array of objects.

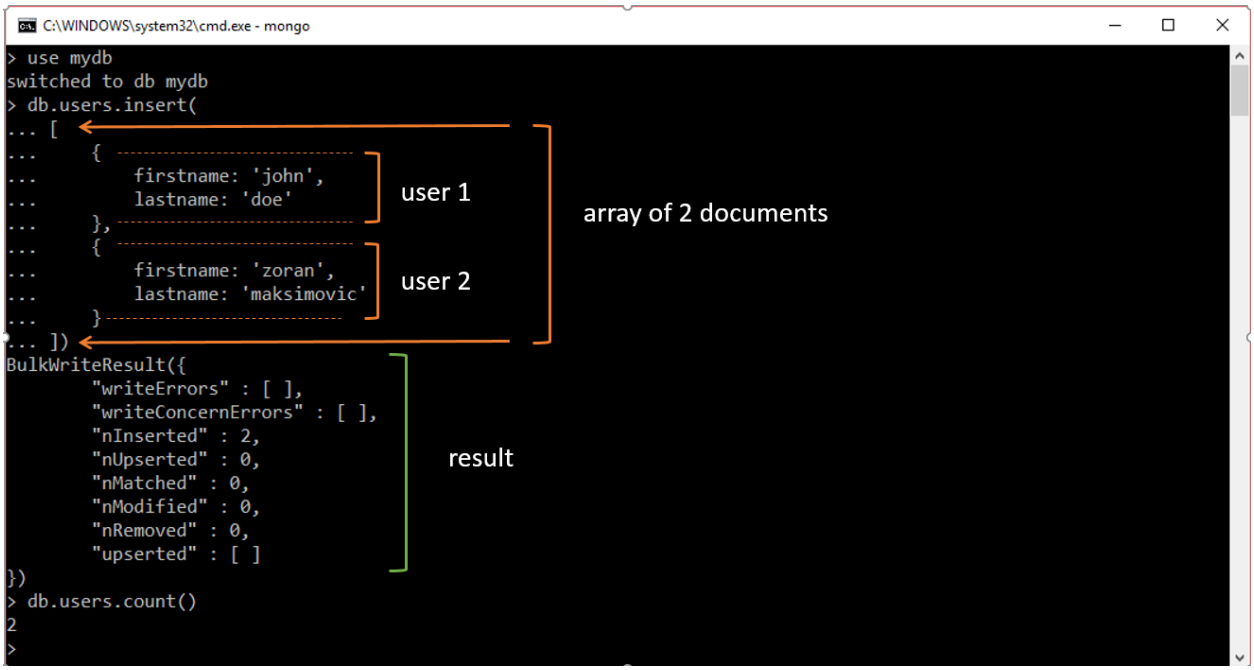


Figure 21: Inserting multiple documents into a collection.

It's also possible to use the **`db.users.save()`** to insert a new document into the collection. In reality, the **`save()`** command can be used both for inserting or updating documents.

If a document does not exist with the specified **`_id`** value, the **`save()`** method performs an insert with the specified fields in the document; otherwise, an update is performed, replacing all fields in the existing record with the fields from the document.

Let's see an example of how to use the **`save()`** command to insert data into a collection. It's pretty much straightforward, and it looks very similar to the previous examples.

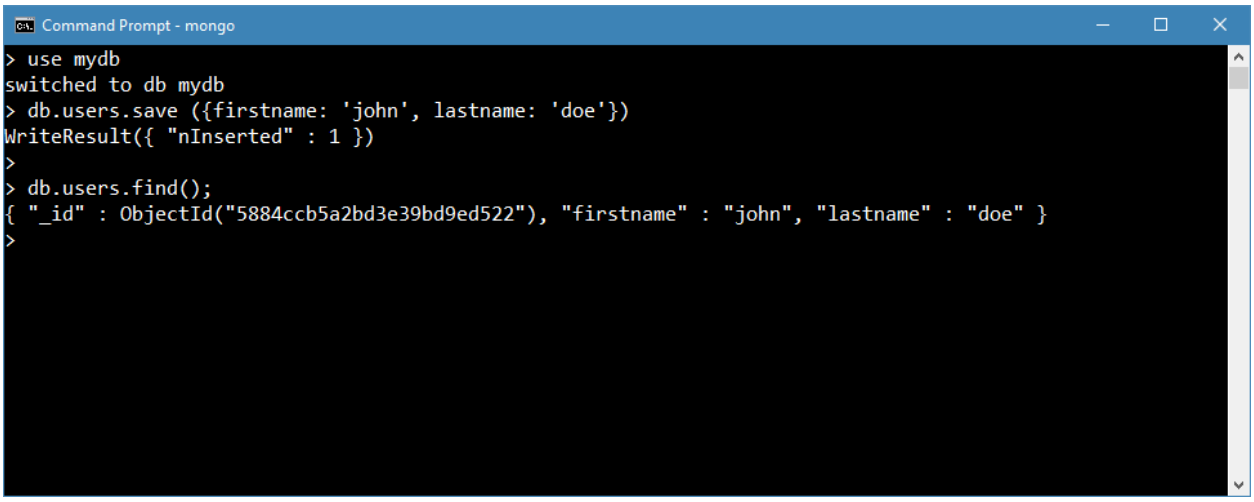


Figure 22: Inserting data into a collection by using save().

Document primary key

When we retrieved the item from the **users** collection, you might have noticed that there is a field called **_id**, which we did not explicitly mention when we inserted the document.

As in any database, MongoDB provides a way of handling the primary keys for the documents. Documents stored in a collection require a unique **_id** field that acts as a primary key. We can explicitly set the value of the **_id**; alternatively, the database will assign one by default. MongoDB uses the **ObjectId** type as the default to store the value for the **_id** field; however, the **_id** field may contain values of any BSON data type, other than an array.

The **ObjectId** is a BSON type, and its value consists of 12 bytes, where the first four bytes are a timestamp that reflect the **ObjectId**'s creation, specifically:

- a 4-byte value representing the seconds since the Unix epoch
- a 3-byte machine identifier
- a 2-byte process ID
- a 3-byte counter, starting with a random value

So, it's a very good candidate for a unique value. Let's look at a small example on how to force our own primary key:

Code Listing 20: Inserting user data by forcing the primary key

```
db.users.insert({_id: 1, firstname: 'john', lastname: 'doe'})
```

Assigning an **_id** is as easy as handling any other attribute. As you can see in Figure 23, duplicate primary keys will be avoided.

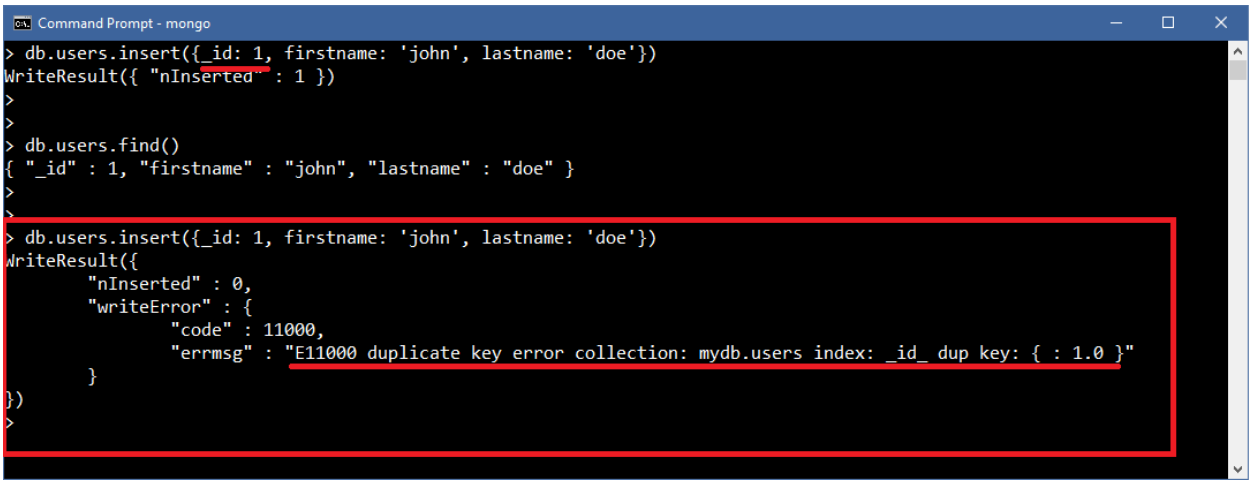


Figure 23: Duplicate primary keys not allowed.

Updating a document

MongoDB allows you to change existing documents in the following ways:

- Update the value of an existing field.
- Change the document by adding or removing attributes (fields).
- Replace the document entirely.

To manipulate the documents, MongoDB mainly offers three different flavors of the **`update()`** methods to be performed at the collection level.

Table 4: MongoDB document update methods

`db.<collection>.update()`	Modifies document(s) in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely.
`db.<collection>.updateOne()`	New as of v3.2. Updates one document within a collection.
`db.<collection>.updateMany()`	New as of version 3.2. Updates multiple documents within a collection.

Updating a value of an existing attribute

The following seems to be the most obvious operation to perform, so let's try to change the **`firstname`** of a user in the **`users`** collection. In order to do this, let's just create a new user with a simpler **`_id`**, by running the following command:

Code Listing 21: Creating a user with a fixed primary key

```
db.users.insert({_id: 1, firstname: 'john', lastname: 'doe'})
```

Once the user is in the collection, we can quickly query the collection and check if the user has been properly inserted by running the **`find()`** method. The **`find()`** method will return the full list of items in the collection.

Code Listing 22: Searching for user data (basic usage)

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

You should be able to see the user with the primary key **`_id = 1`**.

In order to change the **`firstname`** of the user, we can use the following command:

Code Listing 23: Update user's firstname

```
db.users.update(
    { _id : 1},                          //filter
    { $set: { firstname: "andrew" } }  //update action
)
```

The first parameter (in red) represents the *filter* that will be applied to items with the update, a bit like a **`WHERE`** clause when updating records in the RDBMS.

The second parameter (in blue) defines the operation to be performed. In our case, we use the operator **`$set`** in order to set the value of the **`firstname`**.

If the field we are passing is not in the document, it will be automatically created. If we would like to add the age of the user, then it becomes as simple as:

Code Listing 24: Updating user's age

```
db.users.update( { _id : 1}, { $set: { age: 40 } })
```

On the command line, you should see something similar to the following:

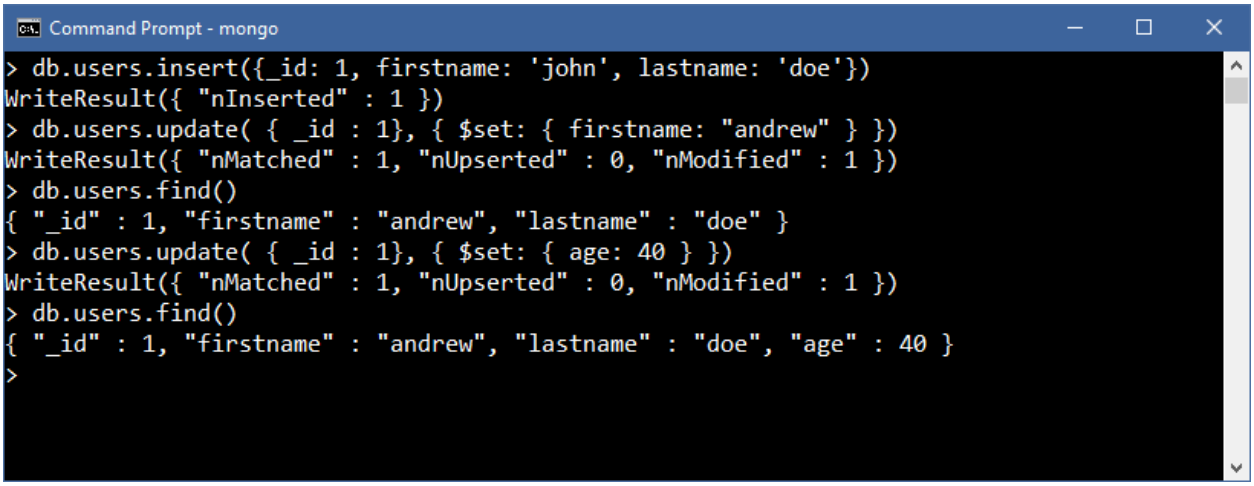


Figure 24: Applying the update methods.

To give an idea of which parameters the **`update()`** method supports, let's take a look at the general method signature:

Code Listing 25: Collection update method signature

```
db.collection.update(
   <query>,
   <update>,
   {
     upsert: <boolean>,
```

```
    multi: <boolean>,
    writeConcern: <document>
  }
)
```

Table 5: Update method's options

query	Defines the selection criteria for the update. This corresponds pretty much to the **WHERE** clause in the RDBMS.
update	Specifies the action to be executed on the document that matches the query criteria. This is where we can update a field or create a new one.
update-options	In the third parameter, there are few options that can be specified: **upsert**: If set to **true**, it creates a new document in case no documents have been found by the query criteria. By default, it is set to **false**. **multi**: If set to **true**, it updates multiple documents; otherwise, only one. **writeConcern**: Defines how the database will handle the write part. For instance, we can define the timeout for the query, or the acknowledgment that the data has been propagated to one or more nodes (in case of a multinode setup).

Update operators

There are quite a few update operators that can be used to manipulate values. We have already seen the usage of the **$set** operator, which performs the change on the document by setting the value explicitly, but there are others:

Table 6: Update operators

$set	Sets the value of a field in a document.
$unset	Removes the specified field from a document.
$min	Updates the field if the value is less than the existing field value.
$setOnInsert	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
$max	Updates the field if the value is greater than the existing field value.

$rename	Renames a field.
$mul	Multiplies the value of the field by a specified amount.
$inc	Increases (increments) the value by a specific amount.
$currentDate	Sets the value of a field to the current date.

If we are dealing with an array of items (a collection of items), then there are several more operators, including:

Table 7: Update operators for arrays

$pull	Removes all list elements that match a specified query.
$push	Adds an item to an array.
$addToSet	Adds an item to an array only if the item does not already exist.

For the full list of available operators, please visit the official MongoDB documentation or check the available options directly in the MongoDB shell with the following command:

Code Listing 26: Use of help on a collection

```
db.users.help()
```

Also, check the method signatures for the these update methods.

Deleting a document

In a very similar way as it happens for updates, MongoDB offers various ways for deleting documents. The methods follow pretty much the same principle.

Table 8: Data removal methods

db.<collection>.drop()	Completely empties the collection by deleting the collection itself. While this is not exactly deleting a document, it can be considered as a way of cleaning up the data.
db.<collection>.remove()	Deletes a single document or multiple documents that match a specified filter.
db.<collection>.deleteOne()	Deletes at most a single document that matches a specified filter, even though multiple documents may

	match the specified filter. It will delete the first document.
db.<collection>.deleteMany()	Deletes all documents that match a specified filter.

Let's demonstrate a deletion of a single document by using the **deleteOne** method. The signature of the **deleteOne** method is as follows. It accepts the filter and the **writeConcern** as discussed previously. **WriteConcern** is not mandatory, and if not specified, will take the default values.

Code Listing 27: DeleteOne method signature

```
db.collection.deleteOne( <filter>,  writeConcern: <document> )
```

Let's assume we already have an entry in the **users** table, and we try to delete it. This might look like the following:

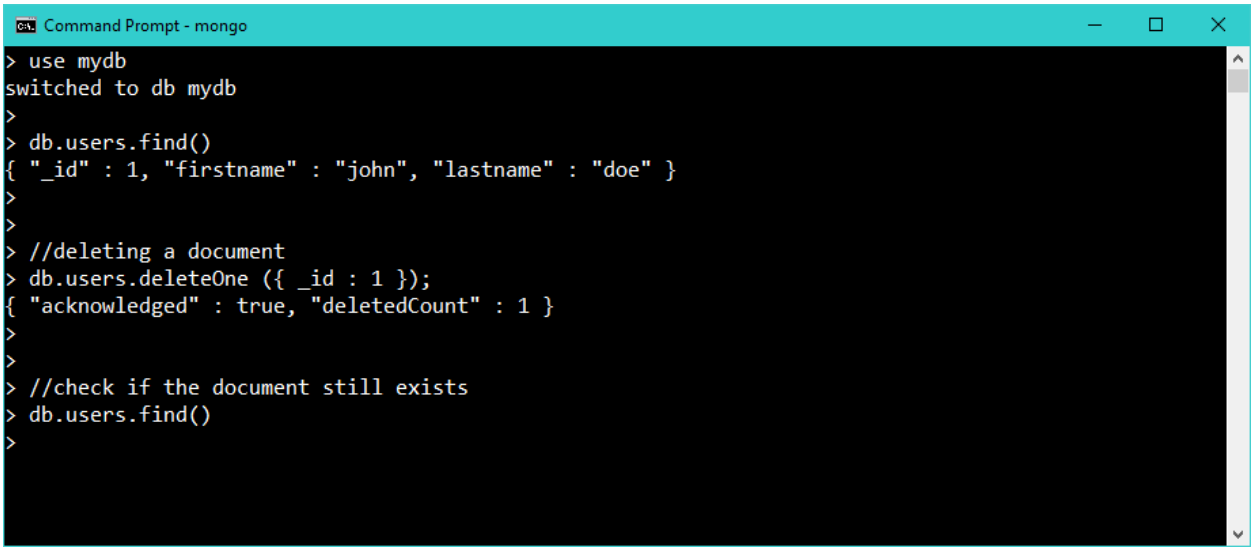


Figure 25: Deleting a document.

What is interesting here is the filter part, where we specify the document to be deleted. In our case, this is directly the primary key, **_id = 1**.

Chapter 5 Data Retrieval

We briefly mentioned how to retrieve data in the previous chapter, in order to support operations such as create, delete, and update. MongoDB is quite rich when it comes to data retrieval, as you will see.

Querying a collection

The starting point for the data retrieval is the **`db.<collection>.find()`** method. By just quickly using the **`help()`** method, we can see that the **`find()`** method offers several other modifiers for performing any kind of query over a collection.

It is possible to **`sort`** and apply the **`filters`** to the search itself. The following image shows the modifiers the **`find()`** contains:

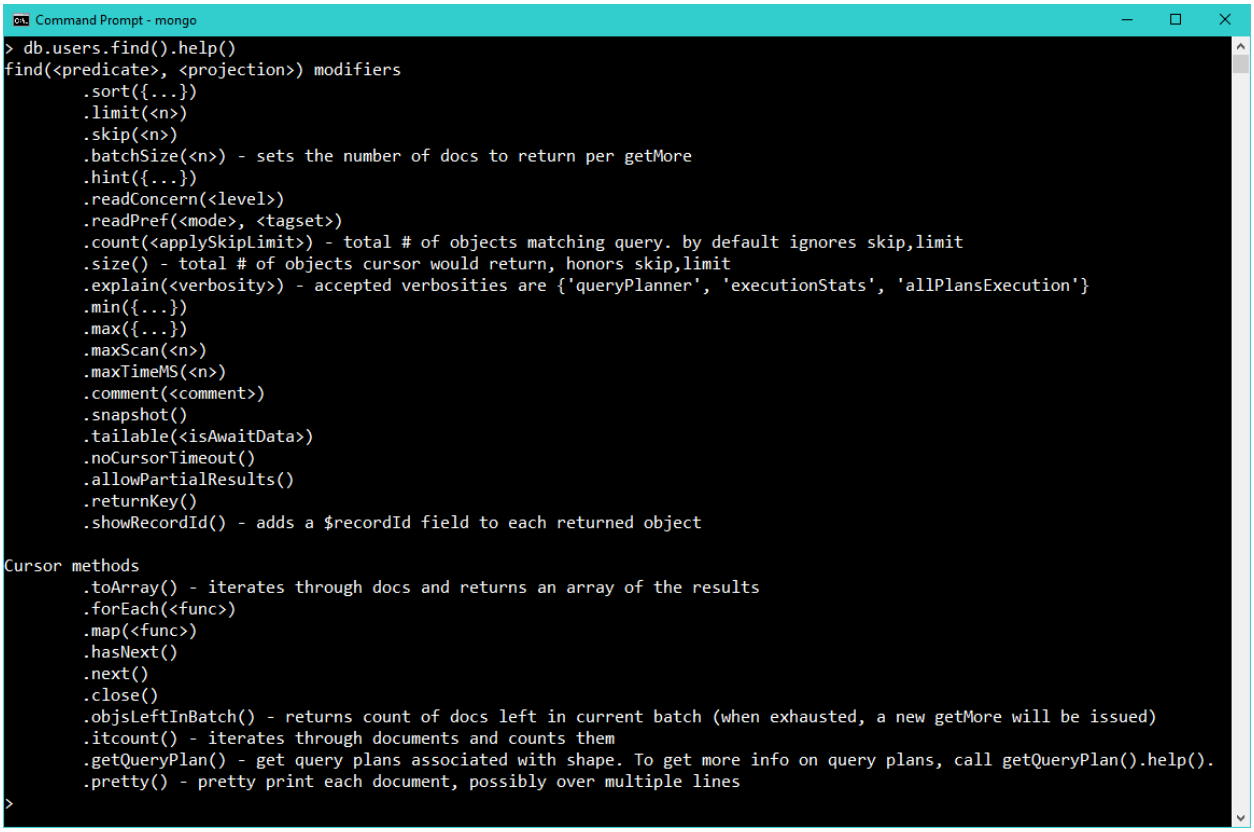


Figure 26: Find() method.

By definition, the queries are issued on a single collection; therefore, joining to other tables, as we would expect at this stage, is not possible. In order to achieve this, we need to either use the aggregation or the **`MapReduce`** mechanism, which will be discussed in the coming chapters.

In order to perform a query, we would usually need to specify a filter, unless we are retrieving all the documents of a collection. MongoDB uses some **comparison operators** that really come in handy for data comparison and matching:

Table 9: Query comparison operators

Name	**Description**
`$eq`	Matches values that are **equal** to a specified value.
`$gt`	Matches values that are **greater** than a specified value.
`$gte`	Matches values that are **greater than or equal** to a specified value.
`$lt`	Matches values that are **less than** a specified value.
`$lte`	Matches values that are **less than or equal** to a specified value.
`$ne`	Matches all values that are **not equal** to a specified value.
`$in`	Matches any of the values specified **in** an array.
`$nin`	Matches **none** of the values specified in an array.

And in addition to these, we have some **logical operators**, such as:

Table 10: Logical operators

Name	**Description**
`$or`	Joins query clauses with a logical **`OR`**; returns all documents that match the conditions of either clause.
`$and`	Joins query clauses with a logical **`AND`**; returns all documents that match both the conditions.
`$not`	Joins query clauses with a logical **`AND`**; returns all documents that match the conditions of both clauses.
`$nor`	Joins query clauses with a logical **`NOR`**; returns all documents that fail to match both clauses.

For additional information, you may check the reference card directly on the MongoDB website.

Using the method **`find()`** without any parameter will simply return anything available in the given collection. While this is useful in some simple scenarios, the tendency is to search for specific information by matching some of the attributes (fields) of a document to be returned. This is achieved by specifying the **`<query filter>`**, the first parameter of the method.

The basic signature of the **`find()`** method accepts two parameters. Both parameters are optional:

Code Listing 28: Collection find method signature

```
db.collection.find(<query filter> , <projection>)
```

`Query filter`: Represents the filter that is going to be applied while searching for data (exactly the same thing as the **`WHERE`** clause in the RDBMS database).

`Projection`: Provides a mechanism of specifying which data we want to return back. If we only want to return a specific field(s), this is the place to specify it.

As an example, if we were to retrieve users whose **`firstname`** and **`lastname`** match a given value, we would use the following notation (the formatting on multiple lines is simply for better readability):

Code Listing 29: Usage of an AND operator on a query

```
db.users.find(
{
     $and: [
               { firstname: "john" },
               { lastname : "doe" }
            ]
 });
```

Square brackets would contain an array of arguments. As demonstrated in Figure 27, we can see that the query returned the document that matches the values we specified.

Figure 27: Searching for firstname and lastname.

At the same time, if we were to combine the **`$and`** and **`$or`**, the following query is made:

Code Listing 30: Usage of AND and OR on a query

```
db.users.find(
{
     $and: [
```

```
                { firstname: "john" },
                { lastname : "doe" }
            ],
     $or:   [
                { _id : 1 }
            ]
});
```

When this runs in the MongoDB shell, the result is as follows:

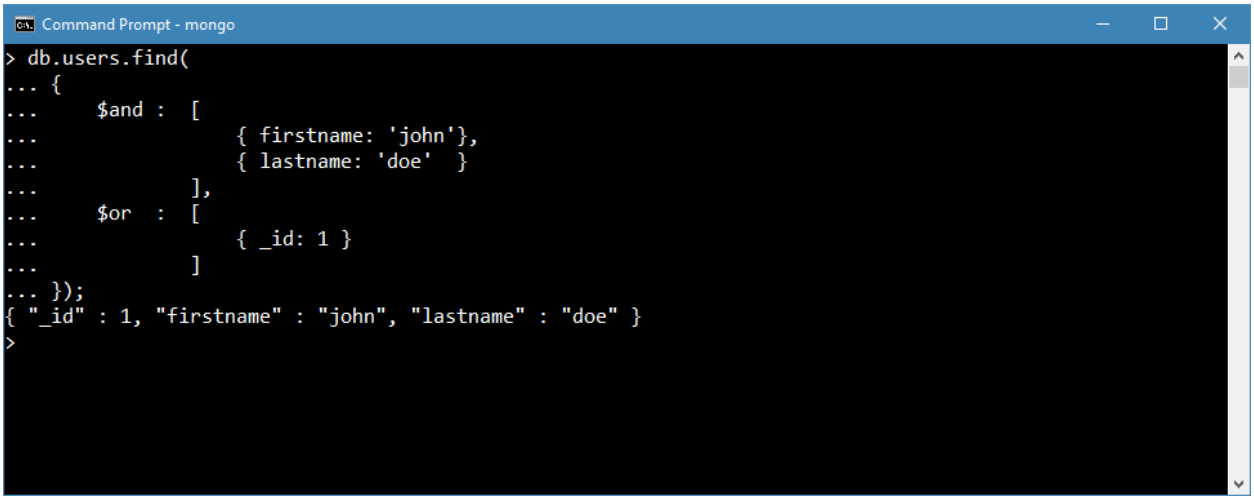


Figure 28: Specifying the filter containing AND and OR.

At the same time, if we want to use some comparison operators, the basic usage is to place the operator inside the curly brackets together with the value in question:

Code Listing 31: Searching for a user where the id is greater than 1

```
db.users.find( { _id : {$gt: 1} });
```

This query searches for items where the **_id** value is greater than **1**.

Table 11: Examples of query filters

Name	Description
{lastname : "doe"}	Docs in which **lastname** is equal to **doe**, or an array containing the value **doe**.
{age: 10, lastname: "doe"}	Docs in which age is equal to **10** and **lastname** is equal to **doe**.
{age: {$gt: 10}}	Docs in which age is greater than **10**. Also available: **$lt (<)**, **$gte (>=)**, **$lte (<=)**, and **$ne (!=)**.

Name	Description
{lastname: {$in: [“doe”, “foo”]}}	Docs in which **lastname** is either **doe** or **foo**.
{a: {$all: [10, “hello”]}}	Docs in which **a** is an array containing both **10** and **hello**.
{“name.lastname": “doe”}.	Docs in which **name** is an embedded document with **lastname** equal to **doe**.
{a: {$elemMatch: {b: 1, c: 2}}}	Docs in which **a** is an array that contains an element with both **b** equal to **1** and **c** equal to **2**.
{$or: [{a: 1}, {b: 2}]}	Docs in which **a** is **1** or **b** is **2**.
{a: /^m/}	Docs in which **a** begins with the letter m. One can also use the regex operator: **{a: {$regex: “^m”}}**.
{a: {$mod: [10, 1]}}	Docs in which a remainder after division with **10** is **1**.
{a: {$type: 2}}	Docs in which **a** is a string. (See bsonspec.org for more.) **{ $text: { $search: “hello” } }** Docs that contain **hello** on a text search. Requires a text index.

Projections

Let’s imagine a situation in which we have a document with a very large number of attributes (fields), but we only need to return one or two. This can be done by specifying the projection parameter in the **find()** method.

In the this chapter, we have seen how to filter for some data, and we have also seen that all the attributes get returned.

The projection is actually very simple; we simply need to specify the list of attributes that we want to be included or excluded in the output. And this works as follows:

In order to return only the **firstname** attribute from the query, we will specify the following:

Code Listing 32: Finding all users and returning the firstname only

```
db.users.find( { },  {_id: 0, firstname: 1} );
```

The number **1** in this case actually means **true** or include, and **0** means **false** or exclude.

The **_id** field will be always returned, unless explicitly excluded from the query.

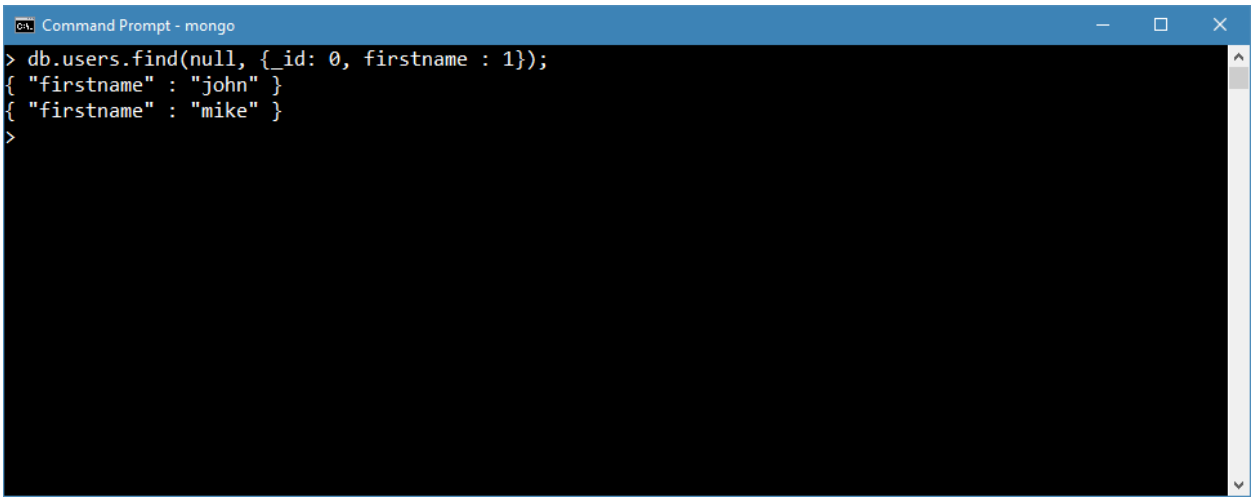


Figure 29: Returning only the firstname from the query.

Sorting

In order to sort out the result, we can use the **sort()** modifier on top of the **find()**. This looks like the following:

Code Listing 33: Specifying the sorting

```
db.users.find({}, {}).sort( { _id : -1} );
```

The **sort()** modifier accepts a list of fields on which we can assign two values, negative (**-1**) or positive (**1**), where:

- **Negative (-1)**: descending order
- **Positive (1)**: ascending order

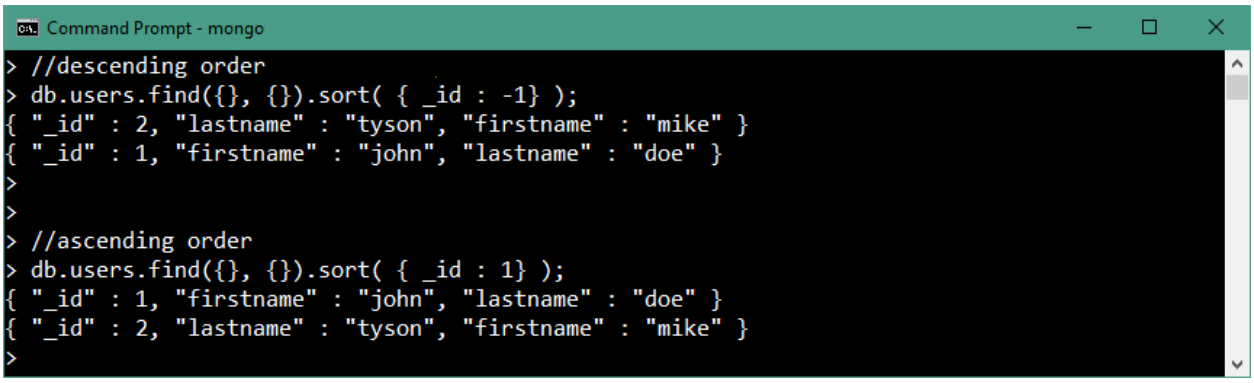


Figure 30: Sorting by descending or ascending order.

Limiting the output

In order to limit the number of documents being returned, we can use the **`limit()`** modifier. By specifying the value on the **`limit()`**, only that number of documents will be returned.

Code Listing 34: Limiting the number of documents returned by a query

```
db.users.find({}, {}).limit( 10 );
```

In this example, only the first 10 documents will be returned.

Cursor

When invoking the **`find()`** method, a cursor is being returned, and it can be used to iterate through documents quite easily, as demonstrated in Code Listing 35.

The query returns all the documents, and we are using the cursor to implement some specific logic at a document level. Our example here is used purely to demonstrate few things:

- We can iterate through documents.
- We can instantiate a single document.
- We can access document internal values.

Code Listing 35: Iterating through the cursor

```
var cursor = db.users.find({}, {});

while (cursor.hasNext()) {

    var document = cursor.next();

    if(document) {
        var firstname = document.firstname;
        print (tojson(firstname));
    }
};
```

In the MongoDB shell, we can see that **`"John"`** and **`"Mike"`** are returned as part of the query.

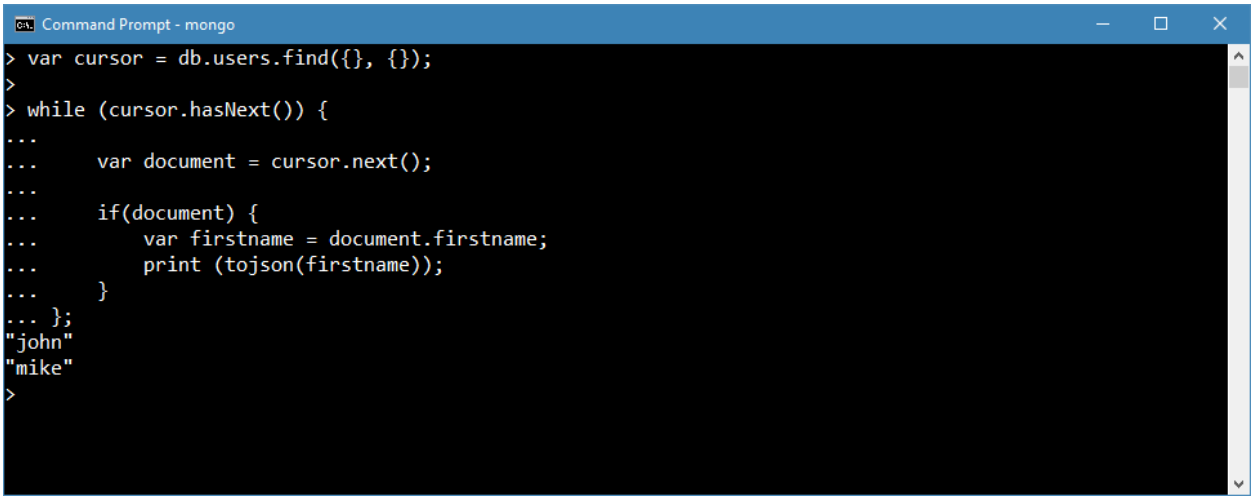


Figure 31: cursor in MongoDB shell.

The same can be achieved by using the **`forEach()`** function that will iterate through all the documents. It makes it a bit easier to code, but is substantially the same as the previous technique.

Code Listing 36: Using forEach on a cursor

```
var cursor = db.users.find({}, {});

cursor.forEach( function(document) {
        var firstname = document.firstname;
        print (tojson(firstname));
});
```

Aggregations

What we saw in the previous chapter was mainly related to retrieving documents from a given collection by filtering and projecting. What we haven't seen is the way to *aggregate* data; and this is simply because the **`find()`** method doesn't offer a way to do this.

The MongoDB documentation defines aggregations as follows: *Aggregations are operations that process data records and return computed results*.

Therefore, MongoDB offers a way to perform operations (computations) and to group values together in order to return a single result. With this functionality, we are able to perform any kind of analytic tasks on the data available in the database.

If you are coming from the RDBMS world, this is pretty much corresponding to the way of using operations around **`GROUP BY`** (**`sum()`**, **`avg()`**, **`count()`**, etc.).

MongoDB offers three ways to achieve this:

- Aggregation pipeline

- Map-reduce function
- Single-purpose aggregation methods

The aggregation pipeline

The MongoDB Aggregation Framework offers a way of processing documents in various stages (as a pipeline). Each stage transforms the documents as they are passing through the pipeline.

When executing a pipeline, MongoDB pipes operators into each other. If you are familiar with the Linux concept of a pipe, then that's the closest analogy. In a nutshell, this means that the **output** of an operator becomes the **input** of the following operator. The result of each operator is a new collection of documents. For instance, one possible pipeline would be as follows:

Figure 32: Aggregation pipeline example.

We can add as many operators in the pipeline as we like; we can also add the same operator more than once, and at a different position in the pipeline.

Figure 33: Aggregation example with repeating operators.

When it comes to implementation, MongoDB offers the **`aggregate()`** function that accepts the list of stages to be applied. The application of stages happens as a sequence is passed to the method. The method signature is as follows:

Code Listing 37: Aggregate method signature.

```
db.collection.aggregate([ { <stage 1> }, { <stage 2>}, … ]);
```

Possible stages are described in the following table from the MongoDB documentation, so let's dive into more details:

Table 12: Pipeline operators

Operator	Description	SQL equivalent
`$project`	Changes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.	**`SELECT`**
`$match`	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline	**`WHERE`**

Operator	Description	SQL equivalent
	stage. **$match** uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).	
$redact	Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of **$project** and **$match**. Can be used to implement field-level redaction. For each input document, outputs either one document or zero documents.	N/A
$limit	Passes the first **n** documents unmodified to the pipeline where **n** is the specified limit. For each input document, outputs either one document (for the first **n** documents) or zero documents (after the first **n** documents).	**LIMIT** or **TOP xxx**
$skip	Skips the first **n** documents where **n** is the specified skip number, and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first **n** documents) or one document (if after the first documents).	**LIMIT ..OFFSET**
$unwind	Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs **n** documents where **n** is the number of array elements and can be zero for an empty array.	N/A
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.	**GROUP BY**
$sample	Randomly selects the specified number of documents from its input.	N/A
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.	**ORDER BY**
$geoNear	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of **$match**, **$sort**, and **$limit** for geospatial data. The output documents include an additional distance field and can include a location identifier field.	N/A

Operator	Description	SQL equivalent
$lookup	Performs a left-outer join to another collection in the same database to filter in documents from the "joined" collection for processing.	**JOIN**
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the **$out** stage, it must be the last stage in the pipeline.	**SELECT**
$indexStats	Returns statistics regarding the use of each index for the collection.	**EXPLAIN**

To demonstrate the usage of the aggregation, we need to have some data to work against; therefore, we will be working on a collection of books, which will be in a very simple format: title, author, number of pages, and language.

So, let's populate the database with some sample data:

Code Listing 38: Simple data to be used for aggregation

```
db.books.insert(
[
   {_id: 1, title: "Anna Karenina", author: "Leo Tolstoy", pages : 500,
language: "russian"},
   {_id: 2, title: "Madame Bovary", author: "Gustave Flaubert", pages :
450, language: "french"},
   {_id: 3, title: "War and Peace", author: "Leo Tolstoy", pages : 470,
language: "russian"},
   {_id: 4, title: "The Great Gatsby", author: "F. Scott Fitzgerald", pages
: 300, language: "english"},
   {_id: 5, title: "Hamlet", author: "William Shakespeare", pages : 150,
language: "english"}
]);
```

Let's try to return the book with the largest number of pages, per language.

In order to achieve this, one of the strategies would be to:

1. Group the documents by language (in blue), by specifying the item on which the grouping will be performed.
2. Use the **$max** operator to find out which book has the largest number of pages (in red) for that given language.

Code Listing 39: Data aggregation (grouping)

```
db.books.aggregate(
   [
     {
       $group:
```

```
            {
              _id: {language: "$language"},
              pages: { $max: "$pages" }
            }
        }
    ]
);
```

One thing to note in this query is that we are using the **`$field`** notation to refer to the value of the field being processed (**`$max : "$pages"`**).

Execution returns the result shown in Figure 34.

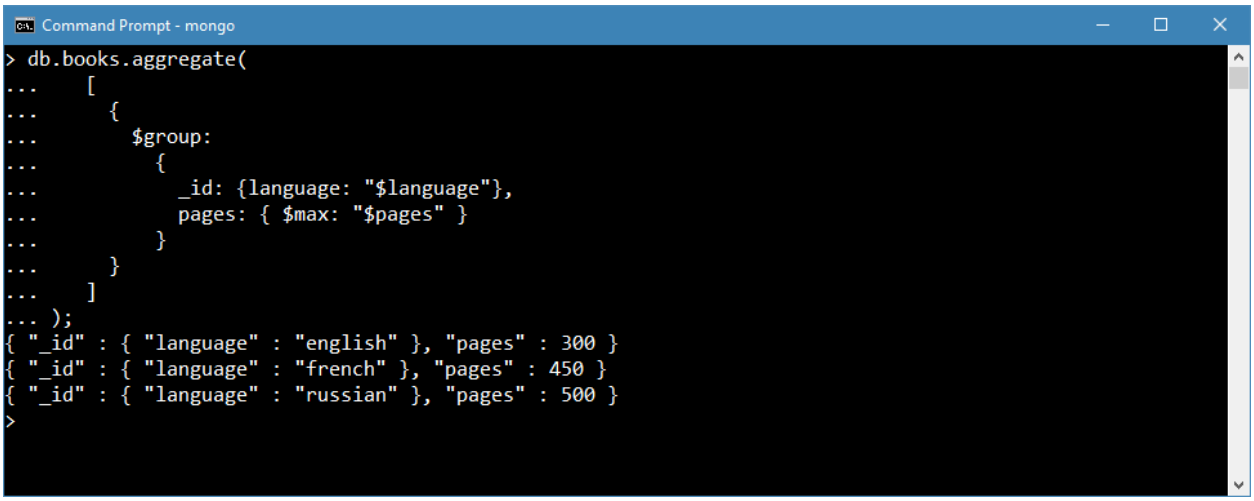


Figure 34: Data aggregation (grouping).

In order to apply the stages concept described at the beginning of the chapter, let's simply add one more requirement: let's make the previous calculation, but only apply it to the English language. That means we are not interested in knowing anything about languages other than English.

We can rewrite the query and divide it into two stages:

1) **`$match`**: We are filtering out all the items we don't want to perform the calculation on. That means we limit our data to only "english language" before starting the calculation.

2) **`$group`**: Everything is as before; the only difference is that the input to this stage would be consisting of books only written in English.

Now, the query looks as follows:

Code Listing 40: Data aggregation (grouping) with filtering ($match)

```
db.books.aggregate(
      [
            {
```

```
                $match : { language : "english" }
            },
            {
                $group:
                    {
                        _id: {language: "$language"},
                        pages: { $max: "$pages" },
                        title: { $first: "$title"}
                    }
            },
        ]
);
```

As shown in the MongoDB shell result, we can only see the values for **`english`**. All the rest is not visible.

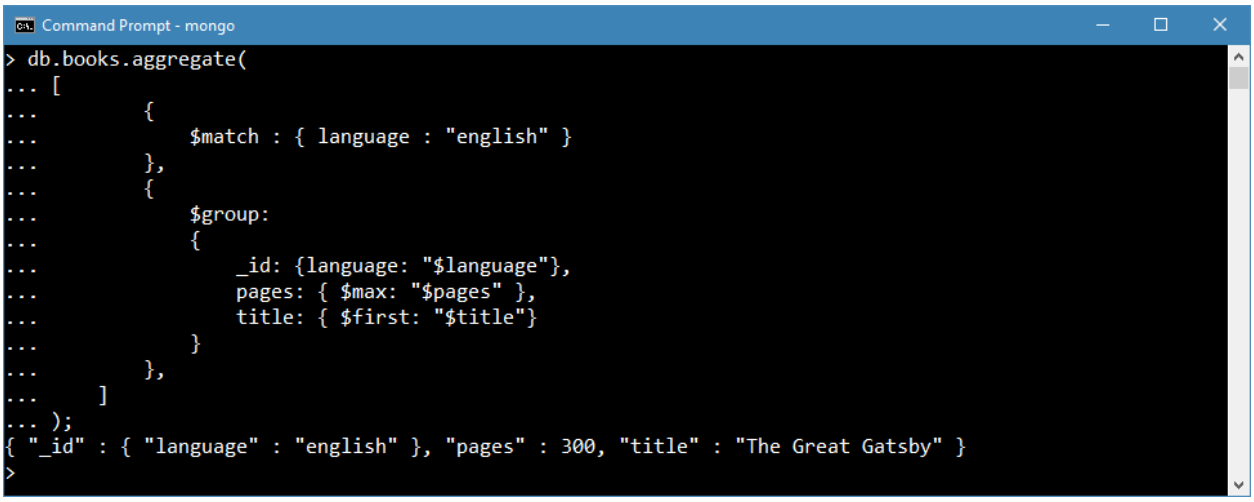


Figure 35: Aggregated Pipeline: group books by prefiltering collection.

The interesting point in the query above is the order of the operations we used. We could have certainly moved the **`$match`** part to be after the **`$group`**, in which case the result would be exactly the same. What would change is the processing of the data between stages. For instance, it makes sense to filter as much as possible *before* performing calculations, as the amount of work to be done by the engine would be smaller than if we only filtered the result at the end. This part about efficiency should be always considered when writing queries.

MapReduce

MapReduce is yet another way of aggregating data, and it is typically used for processing a large amount of data in order to obtain a condensed view of the same.

Every MongoDB collection has a **`mapReduce()`** command as part of it, with the following signature:

Code Listing 41: MapReduce method's signature

```
db.<collection>.mapReduce (
    mapFunction,
    reduceFunction,
    {
        <out>,
        <query>,
        <sort>,
        <limit>,
        <finalize>,
        <scope>,
        <jsMode>,
        <verbose>,
        <bypassDocumentValidation>
    }
);
```

Map and **reduce** are actually JavaScript functions, which we have to define.

Map

The **map** function is responsible for transforming each input document into zero or more documents. The context of the **map** function is the collection itself; therefore, you can use **this** within the function. **Emit()** is responsible for creating the output. The **map** function can be also seen as a mechanism of a **GROUP BY** in the RDBM world. The main goal is to return values normalized and grouped by a common **key**, which can be any of the properties of the collection, and a set of values that belong to this key.

Code Listing 42: Basic map function

```
var map = function(){
  /* emit values for each document */
  emit(key, <values>);
}
```

For Microsoft.NET developers, the output of the **map** function can be seen as **IDictionary<object, IList<object>>**, or a key associated with a set of values.
We can call **emit**() more than once if we want, or do any logic to manipulate the data we want to group.

Note: If an item contains only one record after the `map()` command is executed, MongoDB won't perform any `reduce` function, as it's already considered to be reduced.

Reduce

The **reduce** function simply gathers results and does something with them, such as reducing or grouping the items based on the same key, and doing something with the values, such as calculating the sum or quantity. The basic signature is as follows:

Code Listing 43: Basic reduce function definition

```
var reduce = function(key, value){
  /* reduce emitted values into result */
  return {result1: one, result2: two};
}
```

If we execute the same example as we have done previously with the aggregation pipeline (to get the books with the largest number of pages), then the **mapReduce** would look similar to the following:

Code Listing 44: MapReduce full example

```
/* 1. defining the map function */
var map = function() {
      emit(this.language,
      {
            pages: this.pages,
            title: this.title
      });
};

/* 2. defining the reduce function */
var reduce = function(key, values) {

      var max = values[0].pages;
      var title = values[0].title;

      values.forEach(function(value) {
            if(value.pages > max){
                  max = value.pages;
                  title = value.title;
            }
      });

      return  {pages: max, title: title} ;
};

/* 3. calling the map reduce against the books collection a*/
db.books.mapReduce(map, reduce, {out: { reduce:"biggest_books" }});

/* 4. Retrieving the result */
db.biggest_books.find();
```

The output of this query is as follows, and it's pretty much the same as the previously obtained one:

Code Listing 45: MapReduce data returned (result)

```
{ "_id" : "english", "value" : { "pages" : 300, "title" : "The Great Gatsby" } }
{ "_id" : "french",  "value" : { "pages" : 450, "title" : "Madame Bovary"    } }
{ "_id" : "russian", "value" : { "pages" : 500, "title" : "Anna Karenina"    } }
```

About the sequence of the execution:

1. In step 1, we define the **map** function. For every row in the **books** table, the function will return an object that will contain the **key = language** itself, and the values associated will be the book title and the number of pages.

Code Listing 46: Example of data returned

```
{ english } => [
                    { pages: 300, title: "The Great Gatsby"},
                    { pages: 150, title: "Hamlet"}
                ]
{ russian } => [
                    { pages: 500, title: "Anna Karenina"},
                     { pages: 470, title: "War and Peace"}
                ]
{ french } =>  [    {pages: 450, title: "Madame Bovary"} ]
```

2. In step 2, we define a **reduce** function. The **reduce** function's responsibility is to loop through the array of values and find out which number of pages has the biggest value. In order to return the values of the title, we have to map the title as well on every loop cycle.
3. Finally, we call the **mapReduce()** function, where the functions declared previously are set as arguments. One interesting aspect of this query is that it outputs the result in a new table called **biggest_books**.

As part of the output options, we could also choose to return the result as a cursor, in which case we would have something like the following:

Code Listing 47: MapReduce with cursor

```
 var values = db.books.mapReduce(map, reduce, {out: { inline: 1 }} );

print(tojson(values));
```

Once we have the **values** cursor filled in, we can iterate through values as we did in the previous chapter where we discussed cursors.

For extensive information about **MapReduce** in MongoDB, see the official documentation.

Single-purpose aggregation operations

MongoDB also offers some more aggregation commands that can be directly executed against a collection.

Table 13: "Simple" aggregation commands

Name	Description
`count`	Counts the number of documents that match the query; optional parameters are: **`limit`**, **`skip`**, **`hint`**, **`maxTimeMS`**.
`distinct`	Displays the distinct values found for a specific key in a collection.
`group`	Groups documents in a collection by the specified key and performs simple aggregation.

Count

`Count` is a very simple function to execute against a collection. By just calling the **`count()`** function, we get back the result.

Code Listing 48: Count method signature

```
db.books.count(query, options);
```

As we can see in Figure 36, we can also specify the query that acts as a data filter.

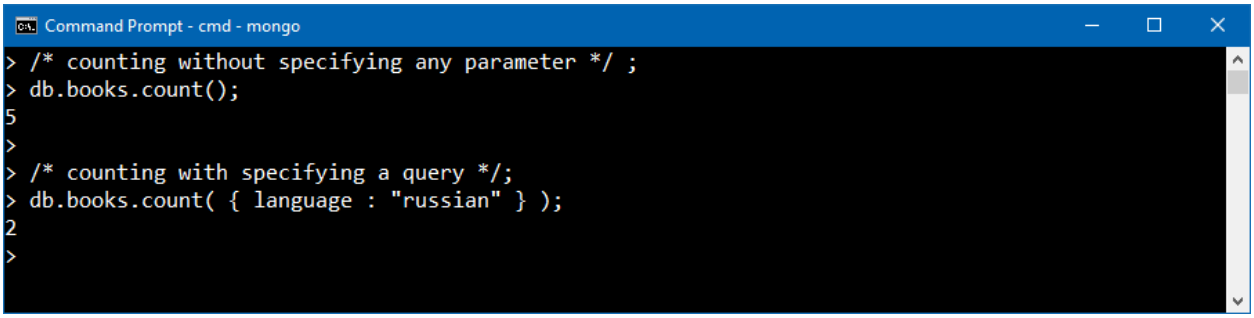


Figure 36: Counting books with or without query.

There are also a few options that can be specified, such a **`limit`**, **`skip`**, and **`maxTimeMS`** (max time in milliseconds), that could help us further specify the options of the count.

Keep in mind that **`count()`** can also be used on a cursor after we use the **`find()`** function.

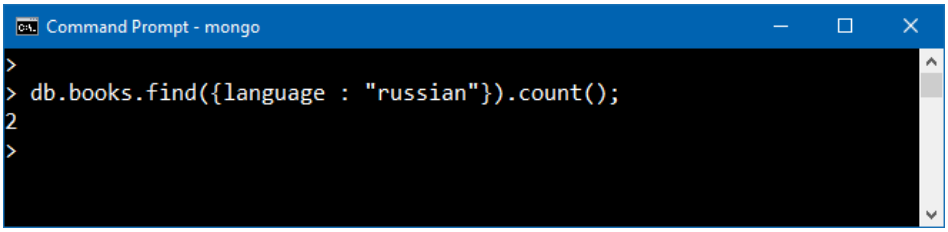


Figure 37: Using count() after searching for data (cursor).

Distinct

`Distinct` will return the list of distinct values as specified in the **`field`** parameter. **`Query`** can also be used to further specify the data worked against.

Code Listing 49: Distinct method signature

```
db.books.distinct(field, query);
```

Figure 38: Returning distinct values from a collection.

Group

The **`group`** method groups documents in a collection by the specified keys and performs simple aggregation functions, such as computing counts and sums. The method is analogous to a **`SELECT`** <...> **`GROUP BY`** statement in SQL. The **`group()`** method returns an array.

The signature of the method is as follows:

Code Listing 50: Group method signature

```
db.<collection>.group ( { key, reduce, initial [, keyf] [, cond] [,
finalize]})
```

Let's explain the most useful parts of the method:

Table 14: Group method parameters

Name	**Description**
key	Represents the field or fields to group. This will be used as a key of the group, to which all of the other values will be referenced.
reduce	An aggregation function that operates on documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document, and an aggregation result document for that group.
initial	Initializes the aggregation result document.
keyf	Optional. Alternative to the key field. Specifies a function that creates a "key object" for use as the grouping key. Use **keyf** instead of **key** to group by calculated fields rather than existing document fields.
cond	Optional. Contains the query if we want to work only against a subset of data.
finalize	Optional. Before the result is returned, it can perform formatting of the data. It's the last method executed within the query.

Let's see the example of if we were to calculate the sum of all of the pages per given language:

Code Listing 51: Grouping example

```
db.books.group (
{
    key : {language: 1 },
    reduce : function( currentDocument, result) {
        result.total +=  currentDocument.pages;
    },
    initial: {total: 0}
});
```

As executed in the shell, it looks like the following:

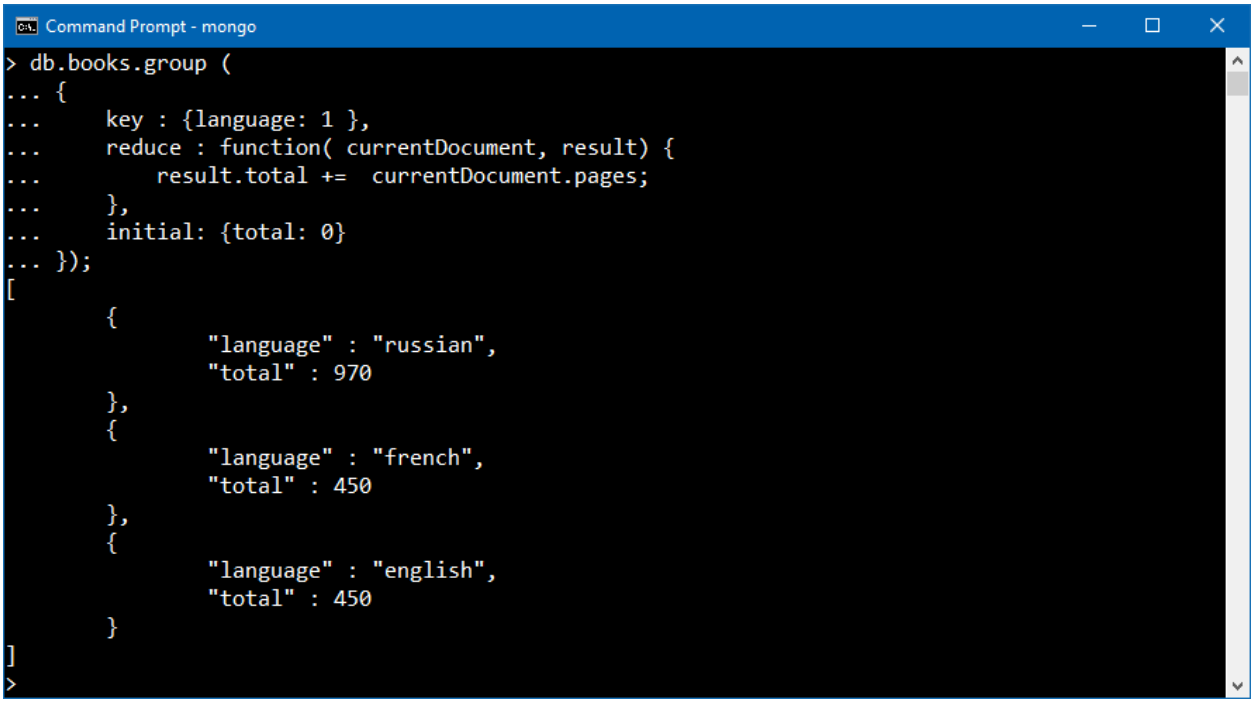


Figure 39: Grouping by language and summing up the number of pages.

Conclusion

In this chapter, we have briefly seen how MongoDB is rich with functionalities when it comes to querying and aggregating data. For the sake of brevity, we haven't seen all the cases, as this probably would require a document on its own. However, I hope that with this the reader can at least identify the elements that might be needed. For further information, see either the official documentation or the myriad of blog posts that explain some corner cases.

Chapter 6 Basic MongoDB with C#

Over the years, as MongoDB has become more of a mainstream database, a large number of technologies have been embraced and extended so that there is a possibility of interaction. It's perfectly normal to interact with MongoDB from Java, Microsoft.NET, Node.js, and many other platforms.

In this chapter, we will be discussing how Microsoft.NET and C# can be used to manipulate data in MongoDB. The idea is to show some basic operations (and bring some building blocks), and then later on to dive into a more complex solution.

To be able to use MongoDB from Microsoft.NET, we need a specific driver. In this book, we will be using the official driver supplied by MongoDB.

The driver can be downloaded from the MongoDB website. However, the preferred way to install the driver is by using the NuGet package, which is available with the following command:

Code Listing 52: Install MongoDB driver on NuGet

```
Install-Package MongoDB.Driver
```

There are several other official (legacy) MongoDB drivers available on NuGet; however, the previously mentioned package is the latest supported, and it should be used for any new project. At the time of writing this book, the latest version of the driver is **2.4**, which requires at least Microsoft.NET v4.5 to be used; however, it also supports the .NET Core.

Table 15: C# driver versions

C#/.NET Driver Version	MongoDB 2.4	MongoDB 2.6	MongoDB 3.0	MongoDB 3.2	MongoDB 3.4
Version 2.4				✓	✓
Version 2.3	✓	✓	✓	✓	✓
Version 2.2	✓	✓	✓	✓	

Since version 2.0 of the driver, there has been support for **`async`** API. This feature allows us to use the resource more efficiently. Compared to the previous version, this version brings a new, more simplified API to work with. For instance, instead of having overloaded methods, there are now **`Option`** classes that allow having the different overloads configured in such a way.

The driver implements the three most important top-level interfaces:

Table 16: MongoDB driver's main interfaces

IMongoClient	Defines the root object that will interact with the MongoDB server. Responsible for connecting, monitoring, and performing operations against the server.
IMongoDatabase	Represents a **database** in the MongoDB server instance.
IMongoCollection	Represents a **collection** in the MongoDB database.

In the upcoming chapters, we will take a look at how to use the classes implementing the above interfaces.

Connecting to the database

In this chapter, you will learn how to connect to a MongoDB database and perform basic operations such as referencing, creating, and deleting (dropping) the database.

Connecting to the database is very simple and it involves using an instance of the **MongoClient**.

Note: As we have seen in previous chapters, unless otherwise configured, MongoDB runs on port 27017 by default.

The first thing to do is to install MongoDB C# driver by running **Install-Package MongoDB.Driver** directly in Visual Studio, using the Package Manager Console.

The easiest way, without using any additional settings, is to create an instance of the **MongoClient** class and pass the **connectionString** as the argument. Note that the configuration string has the format of "**mongodb://servername:port**".

Code Listing 53: Connecting to MongoDB without authentication

```
using System;
using MongoDB.Driver;

public static void ConnectWithoutAuthentication()
{
    string connectionString = "mongodb://localhost:27017";

    MongoClient client = new MongoClient(connectionString);

    Console.WriteLine("Connected");
}
```

The connection string can contain quite a few parameters, such as a list of hosts in case we use a MongoDB cluster, or to specify connection options. The full signature is as follows:

Code Listing 54: MongoDB connection string

```
mongodb://[username:password@]host1[:port1][,hostN[:portN]]
[/[database][?options]]
```

Creating an instance of the **MongoClient** automatically creates a connection to the MongoDB. MongoClient in the background uses a connection pool.

The MongoDB recommendation is to create only one instance of the **MongoClient** per application so that the connection can be better reused. The **MongoClient** instance can be safely configured as a singleton lifetime if an inversion of control (IoC) framework is used.

Authentication

MongoDB itself supports several authentication mechanisms, such as SCRAM-SHA-1, MONGODB-CR, X.509 Certificate, LDAP Proxy, and Kerberos. By default, the **SCRAM-SHA-1** is used. Sending the username and password is as easy as creating an instance of **MongoCredential** and supplying the **database**, **username**, and **password**, and wrapping it within an instance of the **MongoClientSettings** to be passed to the **MongoClient**. This is the second overload that can be used. It's probably more complex to describe than to see in action, as follows:

Code Listing 55: Connecting to MongoDB with authentication

```
using MongoDB.Driver;

public static void ConnectWithAuthentication()
{
    string dbName = "ecommlight";
    string userName = "some_user";
    string password = "pwd";

    var credentials = MongoCredential.CreateCredential(dbName, userName, password);

    MongoClientSettings clientSettings = new MongoClientSettings()
    {
        Credentials = new[] { credentials },
        Server = new MongoServerAddress("localhost", 27017)
    };

    MongoClient client = new MongoClient(clientSettings);

    Console.WriteLine("Connected as {0}", userName);
}
```

As quickly referenced, MongoDB has a rich model of authentication mechanism, which goes far beyond the scope of this book. For more detailed information about the various authentication methods, see this documentation.

Database operations

Through the MongoDB driver, it is possible to create, list, or drop a database, which we will demonstrate in this section.

Referencing a database

Creating a reference to a database is a basic operation that will be needed in almost all the data manipulation examples. The hierarchy defined in MongoDB is:

Server > Database > Collection > Document > Data

In order to get to the data, there should be a reference to the database, and then all the way down.

In order to keep the examples easy and simplify the code, we will create a simple method that will return an instance of **`MongoClient`**, which will then be used instead of manually creating the connection every time.

Code Listing 56: Generic method of instantiating MongoClient

```
public static MongoClient GetMongoClient(string hostName)
{
    string connectionString = string.Format("mongodb://{0}:27017", hostName);
    return new MongoClient(connectionString);
}
```

We use the **`GetDatabase()`** method to get a reference to a database directly from the **`MongoClient`** instance, as follows:

Code Listing 57: Generic method of getting the database reference

```
public static IMongoDatabase GetDatabaseReference(string hostName, string
dbName)
{
    MongoClient client = GetMongoClient(hostName);
    IMongoDatabase database = client.GetDatabase(dbName);
    return database;
}
```

`GetDatabase()` returns an object that implements the **`IMongoDatabase`** interface, in this case **`MongoDatabaseImpl`**, which is the concrete implementation.

Database creation

MongoDB driver doesn't have an explicit method to create a database, but this can be done by referencing a database, as we have seen in the **`GetDatabaseReference`** method.

If the database name does not exist, MongoDB driver will create it automatically the first time we add a document to it, or by simply creating a new collection.

Code Listing 58: Create a new database in C#

```
public static IMongoDatabase CreateDatabase(string databaseName, string collectionName)
{
    MongoClient client = GetMongoClient(hostName);
    IMongoDatabase database = client.GetDatabase(databaseName);
    database.CreateCollection(collectionName);
    return database;
}
```

Which we can call:

Code Listing 59: Calling CreateDatabase method

```
public static void Main(string[] args)
{
    CreateDatabase("newDatabaseName", "newCollectionName");
}
```

Getting the list of databases

Getting the list of databases is straightforward. In the same style as getting the reference to the single database, we can call **`ListDatabases()`** or **`ListDatabasesAsync()`** on the **`MongoClient`** instance and obtain an instance of the **`BsonDocument`**, which will contain the basic information about the databases available on the server. Both of the methods would return an instance of an object that implements **`IAsyncCursor`**; however, the **`Async`** version would return the **`Task<IAsyncCursor>`**. Let's see this in an example:

Code Listing 60: Show list of available databases

```
using System;
using MongoDB.Driver;
using System.Threading.Tasks;

public static async Task GetListOfDatabasesAsync()
{
    MongoClient client = GetMongoClient("localhost");

    Console.WriteLine("Getting the list of databases asynchronously…");
    using (var cursor = await client.ListDatabasesAsync())
    {
        await cursor.ForEachAsync(d => Console.WriteLine(d.ToString()));
    }
}
```

```
public static void GetListOfDatabasesSync()
{
    MongoClient client = GetMongoClient("localhost");

    Console.WriteLine("Getting the list of databases synchronously…");
    var databases = client.ListDatabases().ToList();
    databases.ForEach(d => Console.WriteLine(d.GetElement("name").Value));
}
```

The output in this case is as follows:

```
Getting the list of databases asynchronously.
{ "name" : "admin", "sizeOnDisk" : 81920.0, "empty" : false }
{ "name" : "ecomlight", "sizeOnDisk" : 65536.0, "empty" : false }
{ "name" : "local", "sizeOnDisk" : 73728.0, "empty" : false }
{ "name" : "mydb", "sizeOnDisk" : 65536.0, "empty" : false }
Getting the list of databases synchronously.
admin
ecomlight
local
mydb
```

Figure 40: Returning the list of databases via C#.

I would like to emphasize the fact that in the first case (where we use the **async** method), a **BsonDocument** is returned and printed out to console. Being in JSON format, it's very easily recognized.

In the second case, by using **Console.WriteLine(d.GetElement("name").Value))**, we only return the **name** portion of it.

Deleting a database

To delete a database from MongoDB server, we call the **DropDatabase()** or **DropDatabaseAsync()** method from **MongoClient** object.

Code Listing 61: Deleting the database (sync and async versions)

```
public static void DropDatabase(string databaseName)
{
    MongoClient client = GetMongoClient("localhost");
    client.DropDatabase(databaseName);
}

public static async void DropDatabaseAsync(string databaseName)
{
    MongoClient client = GetMongoClient("localhost");
    await client.DropDatabaseAsync(databaseName);
```

```
}
```

Working with collections

As we have seen in the previous chapters, we can list and manipulate the collections by using the MongoDB shell. This is also possible by using the C# driver. In the following examples, we will show how to list, create, and drop a collection.

There are mainly three methods available on the **`IMongoDatabase`**:

- **`CreateCollection`** and **`CreateCollectionAsync`**: Creates a new collection if not already available.
- **`ListCollections`** and **`ListCollectionsAsync`**: Lists the already available collections on the database.
- **`DropCollection`** and **`DropCollectionAsync`**: Deletes (drops) a collection from the given database.

We can summarize all the operations within the **`CreateListAndDropCollections`** method as follows:

Code Listing 62: Show the list of available collections

```
public static void CreateListAndDropCollections(string databaseName)
{
    var database = GetDatabaseReference(hostName, databaseName);
    var collectionName = "some_collection";

    //create a new collection.
    database.CreateCollection(collectionName);

    //showing the list of collections before deleting.
    ListCollections(hostName, databaseName);

    //delete a collection.
    database.DropCollection(collectionName);

    //showing the list of collections after deleting.
    ListCollections(hostName, databaseName);
}

public static void ListCollections(string hostName, string databaseName)
{
    var database = GetDatabaseReference(hostName, databaseName);

    var collectionsList = database.ListCollections();

    Console.WriteLine("List of collections in the {0} database:",

database.DatabaseNamespace);
```

```
    foreach (var collection in collectionsList.ToList())
    {
        Console.WriteLine(collection.ToString());
    }
}
```

And by calling the previous code from:

Code Listing 63: Calling the method to show collections

```
public static void Main(string[] args)
{
    var newDatabaseName = "newDbName";

    CreateListAndDropCollections(newDatabaseName);
}
```

The output in this case is as follows:

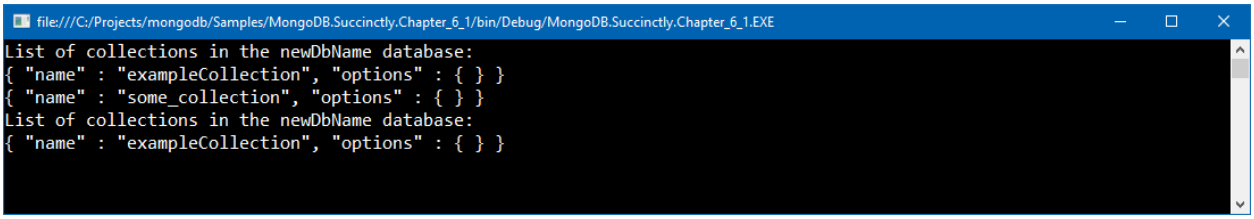
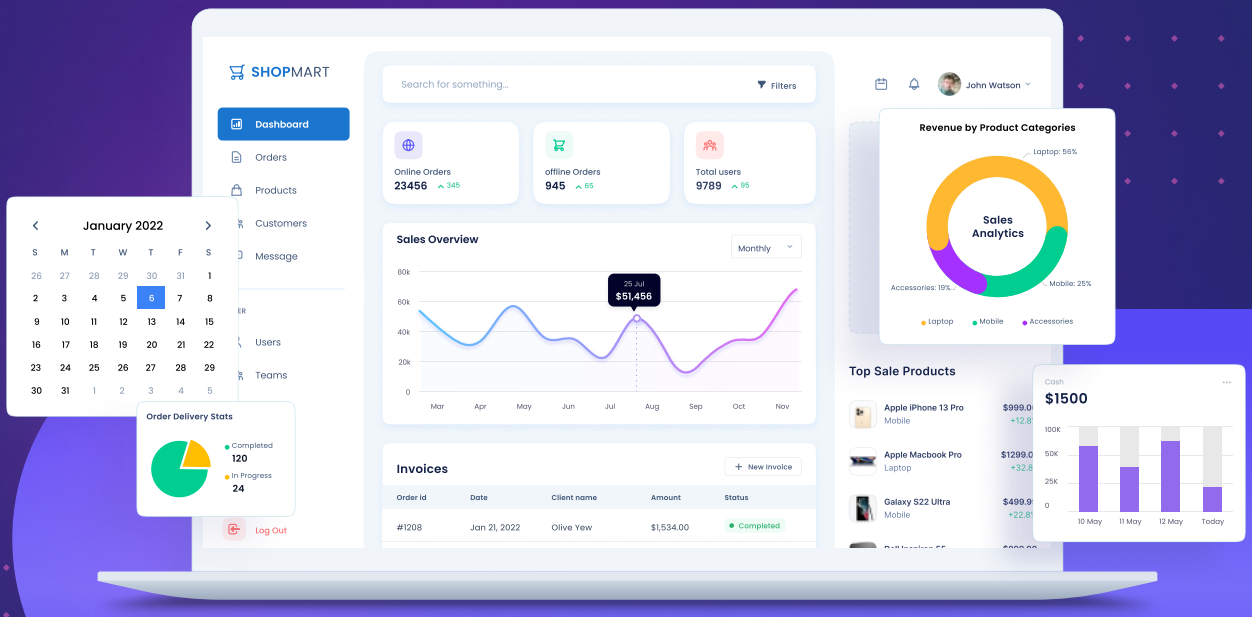


Figure 41: List of available collections.

Keep in mind that creating a collection will cause an error if the collection already exists.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 7 Data Handling in C#

Data representation

Before starting with various examples, we should spend a few words on the **BSON** representation of the data as stored in the MongoDB database, which is achieved through the **BsonDocument** object. There is a very high chance of encountering the **BsonDocument** when working with the driver, so it's crucial to understand what it is and what it represents.

We can think of **BsonDocument** as an equivalent to a table row in an RDBMS database, as it represents the document as it is stored in the database. This is also a way to dynamically express the data being returned or passed to the database.

The following code shows an example of creating a new **BsonDocument**, in our case representing the movie *The Seven Samurai*. As we would expect, the document is declaring the attributes **Name**, **Director**, **Actors**, and **Year**, with the respective values. The array of values is being injected by using another structure, the **BsonArray** object, and the items of the array are themselves declared as **BsonDocument** (a sort of a subdocument).

Code Listing 64: BsonDocument creation

```
BsonDocument sevenSamurai = new BsonDocument()
{
    { "Name" , "The Seven Samurai" },
    { "Director" , "Akira Kurosawa" },
    { "Actors", new BsonArray {
        new BsonDocument("Name", "Toshiro Mifune"),
        new BsonDocument("Name", "Takashi Shimura")}},
    { "Year" , 1954 }
};
```

In the MongoDB C# driver, using the **BsonDocument** as a means of sending or returning data is perfectly legal, and is fully supported. **BsonDocument** can be directly used when we want to work on a "lower" level with the data, without the need for deserialization seen in other formats, such as POCO ("plain old CLR object"), or when the schema is fluid and dynamic. However, most applications are built with a schema modeled in the application itself rather than the database. In these cases, it is likely that the application uses classes. Therefore, an alternative to the **BsonDocument** is to work directly with C# POCO objects.

Code Listing 65: C# representation of the BsonDocument

```
public class Movie
{
    public string MovieId { get; set; }
    public string Name { get; set; }
    public string Director { get; set; }
    public Actor[] Actors { get; set; }
    public int Year { get; set; }
```

```
}

public class Actor
{
    public string Name { get; set; }
}
```

The following instance of the **Movie** class corresponds exactly to the previously shown **BsonDocument** that represents the same movie.

Code Listing 66: Creation of a movie object in C#

```
Movie sevenSamurai = new Movie()
{
    Name = "Seven Samurai",
    Director = "Akira Kurosawa",
    Year = 1954,
    Actors = new Actor[]()
    {
        new Actor { Name = "Toshiro Mifune"},
        new Actor { Name = "Takashi Shimura"},
    }
};
```

Object mapping

The MongoDB driver offers a few ways of controlling the mapping of the values from the database to the POCO class, and vice versa. This is very important in case the names of the attributes as shown in the class differ from what is really stored (persisted) in the database, so it is crucial that there is a mechanism through which we can control exactly what comes in and out.

Serialization is the process of mapping an object to a BSON document that can be saved in MongoDB, and deserialization is the reverse process of reconstructing an object from a BSON document. For that reason, the serialization process is also often referred to as *object mapping*. The default BSON serializer available as part of the driver will take care of this conversion under the hood.

The MongoDB driver mainly supports two ways of mapping properties from and to the BSON representation: using the .NET attributes assigned to the class members, and using the **BsonClassMap** class.

We will use the previous **Media** and **Actor** classes in order to show the various options. One of the most frequently used options is property mapping, which means making sure that each member of the class is mapped to a particular value, but is not limited to it.

Let's see what the **Movie** class would look like when "decorated" with some extra (previously mentioned) attributes. Please note that there are two additional attributes, **Age** and **Metadata**, which will be explained further down the line.

Code Listing 67: POCO object with BSON attributes

```
[BsonIgnoreExtraElements]
public class Movie
{
    [BsonId(IdGenerator = typeof(StringObjectIdGenerator))]
    public string MovieId { get; set; }

    [BsonElement("name")]
    public string Name { get; set; }

    [BsonElement("directorName")]
    public string Director { get; set; }

    [BsonElement("actors")]
    public Actor[] Actors { get; set; }

    [BsonElement("year")]
    public int Year { get; set; }

    [BsonIgnore]
    public int Age
    {
        get { return DateTime.Now.Year - this.Year; }
    }

    [BsonExtraElements]
    public BsonDocument Metadata { get; set; }
}
```

An alternative to these attributes is the **`BsonClassMap`**, which is the entry point for the mapping.

***Tip:** `BsonClassMap` should be called only once per application.*

BsonElement attribute

The **`BsonElement`** attribute will make sure to map the class property to a given name, which means that when the class gets (de)serialized, the names specified as the parameter will be used. As an example, the property **`Director`**, when serialized, will actually be stored in MongoDB as **`directorName`** rather than with the property name that would be the default behavior.

If we were to use the **`BsonClassMap`**, we could achieve the same property-name mapping by chaining the **`SetElementName`** method on the **`MapProperty`** method.

Code Listing 68: Mapping done via BsonClassMap

```
BsonClassMap.RegisterClassMap<Movie>(movie =>
{
    movie.MapProperty(p => p.Name).SetElementName("name");
```

```
    movie.MapProperty(p => p.Director).SetElementName("directorName");
    movie.MapProperty(p => p.Year).SetElementName("year");
    movie.MapProperty(p => p.Actors).SetElementName("actors");
});
```

BsonId attribute

The **BsonId** attribute does mainly two things. It makes the property act as a primary key of the class and, on the other side, it allows us to assign the **IdGenerator** to the property. In our case, it maps the **MovieId** property, which (when serialized) will be transformed into the standard **_id**, as we have seen previously. **IdGenerator** is responsible for assigning the new value to the class when this is serialized. There are several serializers that are already part of the MongoDB C# driver, such as **BsonObjectIdGenerator**, **CombGuidGenerator**, **GuidGenerator**, **ObjectIdGenerator**, **StringObjectIdGenerator**, and **ZeroIdChecker<T>**. We are using the **StringObjectIdGenerator**, as in our case the **MovieId** is of type **string**; if it were of type **ObjectID**, then we would need to choose among the other generators.

We can also create our own unique key generators by implementing the **IIdGenerator** interface. What follows is a very naïve implementation of the primary key where we use the **Movie_** prefix followed by a **Guid**:

Code Listing 69: Example of a custom ID generator

```
public class MovieIdGenerator : IIdGenerator
{
    public object GenerateId(object container, object document)
    {

        return "Movie_" + System.Guid.NewGuid().ToString();
    }

    public bool IsEmpty(object id)
    {
        return id == null || string.IsNullOrEmpty(id.ToString());
    }
}
```

Which then would be used as follows:

Code Listing 70: Using the custom ID generator via attributes

```
public class Movie
{
    [BsonId(IdGenerator = typeof(MovieIdGenerator))]
    public string MovieId { get; set; }
…
```

If we were to use the mapping class, then instead of using the **MapProperty** method, we would use the **MapIdProperty**. **MapIdProperty** has the ability to set the **IdGenerator** by using the **SetIdGenerator** method.

Code Listing 71: Setting the custom generator in the BsonClassMap

```
BsonClassMap.RegisterClassMap<Movie>(movie =>
```

```
{
     movie.MapIdProperty(p => p.MovieId).SetIdGenerator(new MovieIdGenerator());
});
```

BsonIgnore attribute

BsonIgnore will simply make sure that the **BSON** serializer will ignore and not serialize the element to the **BSON** format. So, in our case the **Age** property will be never stored into the database, and the mapper will simply not fill this property if the value is available in the database. The corresponding notation of the **BsonClassMap** uses the **UnmapProperty** method.

Code Listing 72: Making the attribute not BSON serializable (ignored)

```
BsonClassMap.RegisterClassMap<Movie>( movie =>
{
     movie.UnmapProperty(p => p.Age);
});
```

BsonIgnoreExtraElements attribute

When a BSON document is deserialized back to a POCO, the name of each element is used to look up a matching field or property; when deserializer doesn't find the mapping property, it throws an exception. This is when the **BsonIgnoreExtraElements** attribute becomes very handy, as it will ignore those extra properties and won't try to link them back to the class.

The **SetIgnoreExtraElements** method on **BsonClassMap** achieves the same.

Code Listing 73: Instructing the serializer to ignore extra elements

```
BsonClassMap.RegisterClassMap<Movie>( movie =>
{
     movie.SetIgnoreExtraElements(true);
});
```

BsonExtraElements attribute

This attribute is one of the two ways of allowing the mapping of extra elements (those that are not part of the original object) dynamically. In fact, this is the way to mix static and dynamic data. In order for this to work, the dynamic property in a class should be of type **BsonDocument**. In our **Movie** class, we have the **Metadata** property that is of type **BsonDocument**.

The corresponding **BsonClassMap** method to be used is the **MapExtraElementsMember**.

Code Listing 74: Enabling the mapping of extra elements

```
BsonClassMap.RegisterClassMap<Movie>( movie =>
{
     movie.MapExtraElementsMember(p => p.Metadata);
});
```

The following is the example of how the movie will be serialized if we specify the **Metadata** as a new **BsonDocument**.

Code Listing 75: Instance of a movie to be serialized

```
Movie theGodFather = new Movie()
{
    Name = "The Godfather",
    Director = "Francis Ford Coppola",
    Year = 1972,
    Actors = new Actor[]
    {
        new Actor { Name = "Marlon Brando" },
        new Actor { Name = "Al Pacino" },
    },
    Metadata = new BsonDocument("href", "http://thegodfather.com")
};
```

It will be stored in the database, such as the following:

Code Listing 76: Serialized movie

```
{
    "_id" : "587a4496c6d11b31a0a6b829",
    "name" : "The Godfather",
    "directorName" : "Francis Ford Coppola",
    "actors" : [
            {
                "Name" : "Marlon Brando"
            },
            {
                "Name" : "Al Pacino"
            }
    ],
    "year" : 1972,
    "href" : "http://thegodfather.com"
}
```

We can clearly see that the **`href`** looks like an ordinary attribute, and the **`Metadata`** property, as it is in the C# file, is not even mentioned.

If we would omit the **`BsonExtraElements`** attribute, then the class would be serialized as follows:

Code Listing 77: Serialized movie without BsonExtraElements specified

```
{
    "_id" : "587a45d9c6d11b40944c32f6",
    "name" : "The Godfather",
    "directorName" : "Francis Ford Coppola",
    "actors" : [
            {
                "Name" : "Marlon Brando"
            },
            {
```

```
                    "Name" : "Al Pacino"
            }
    ],
    "year" : 1972,
    "Metadata" : {
            "href" : "http://thegodfather.com"
    }
}
```

Please note the difference from the previous example. Now, the **Metadata** attribute is shown.

Chapter 8 Inserting Data in C#

In this exercise, we will insert some data into the database called **moviesDb**. As we will be looking at two ways of inserting data, we will store the data into two collections (**movies_bson** for **BsonDocument**-based objects and **movies_poco** for data based on a POCO model). Technically, there should be no difference once the data is saved.

In order to do this, we need some sample data, as defined in the **GetBsonMovies()** method, which returns an array of **BsonDocuments**.

Code Listing 78: Return a BsonDocument array of movies

```
public static BsonDocument[] GetBsonMovies()
{
    BsonDocument sevenSamurai = new BsonDocument()
    {
        { "name" , "The Seven Samurai" },
        { "directorName" , " Akira Kurosawa" },
        { "actors", new BsonArray {
            new BsonDocument("name", "Toshiro Mifune"),
            new BsonDocument("name", "Takashi Shimura")}},
        { "year" , 1954 }
    };

    BsonDocument theGodfather = new BsonDocument()
    {
        { "name" , "The Godfather" },
        { "directorName" , "Francis Ford Coppola" },
        { "actors", new BsonArray {
            new BsonDocument("name", "Marlon Brando"),
            new BsonDocument("name", "Al Pacino"),
            new BsonDocument("name", "James Caan")} },
        { "year" , 1972 }
    };

    return new BsonDocument[] { sevenSamurai,  theGodfather };
}
```

We can insert the data by using the **InsertMany** or **InsertManyAsync** methods available at the collection level. The **GetDatabaseReference** method has been mentioned previously, and this is just a helper method to automate the referencing of a database.

In a case of inserting one document, we would use the **InsertOne** or **InsertOneAsync** method as opposed to the above-mentioned one.

Code Listing 79: Inserting a movie into the database

```
public static async Task Insert<T>(T[] movies, string dbName, string tableN
ame)
{
    var db = DatabaseHelper.GetDatabaseReference("localhost", dbName);

    var moviesCollection = db.GetCollection<T>(tableName);
    await moviesCollection.InsertManyAsync(movies);
}
```

The **insert** method is a generic method accepting a different kind of **movie** array, as we will be using the same method to send the list of movies based on POCO objects.

Tip: The general rule for using any method in MongoDB C# driver is to use the *async* ***method if available, rather than the synchronous ones. This book might not always follow this rule.***

We call the **Insert** method as follows:

Code Listing 80: Calling the inserting method

```
BsonDocument[] movies = MovieManager.GetBsonMovies();
MovieManager.Insert<BsonDocument>(movies, "moviesDb", "movies_bson").Wait()
;
```

Don't be confused by the **MovieManager** class, which is created as a helper class and contains the methods **GetBsonMovies** and **Insert<T>(..)**.

After running this code, the result will be as follows when querying the **movies_bson** collection.

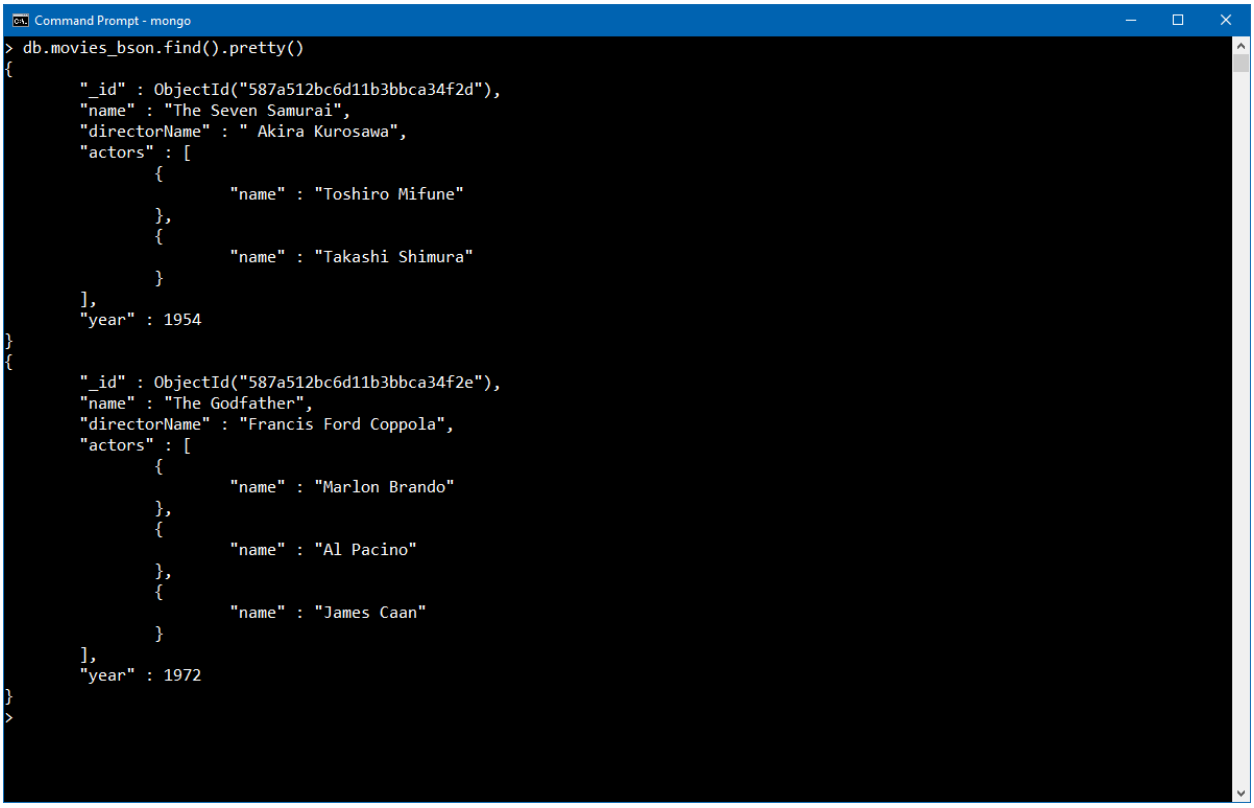


Figure 42 Movies from Bson format as stored in the database

As we have seen previously, the alternative method to the **BsonDocument** is to use **POCO** classes representing the objects, in which case we would first declare an array of **Movie** objects.

Code Listing 81 Get the list of Movies as array

```
public static Movie[] GetMovieList()
{
    Movie sevenSamurai = new Movie()
    {
        Name = "Seven Samurai",
        Director = "Akira Kurosawa",
        Year = 1954,
        Actors = new Actor[]
        {
            new Actor {Name = "Toshiro Mifune"},
            new Actor {Name = "Takashi Shimura"},
        }
    };

    Movie theGodFather = new Movie()
    {
        Name = "The Godfather",
        Director = "Francis Ford Coppola",
```

```
        Year = 1972,
        Actors = new Actor[]
        {
            new Actor {Name = "Marlon Brando"},
            new Actor {Name = "Al Pacino"},
        },
        Metadata = new BsonDocument("href", "http://thegodfather.com")
    };
    return new Movie[] { sevenSamurai, theGodFather };
}
```

However, before calling the **`Insert<Movie>()`** method, as we have seen previously, we need to call the **`RegisterClassMap`**, which will make the driver aware of the mapping between the POCO object and the desired serialization. The full code of the mapper and calling of the **`insert`** method follows.

Code Listing 82: Full object-to-BSON mapping

```
public class BsonMapper
{
    public static void Map()
    {
        if (!BsonClassMap.IsClassMapRegistered(typeof(Movie)))
        {
            BsonClassMap.RegisterClassMap<Movie>(movie =>
            {
                movie.MapIdProperty(p => p.MovieId)
                          .SetIdGenerator(new StringObjectIdGenerator());
                movie.MapProperty(p => p.Name).SetElementName("name");
                movie.MapProperty(p => p.Director).SetElementName("director
Name");
                movie.MapProperty(p => p.Year).SetElementName("year");
                movie.MapProperty(p => p.Actors).SetElementName("actors");
                movie.UnmapProperty(p => p.Age);
                movie.MapExtraElementsMember(p => p.Metadata);
                movie.SetIgnoreExtraElements(true);
            });
        }

       if (!BsonClassMap.IsClassMapRegistered(typeof(Actor)))
       {
           BsonClassMap.RegisterClassMap<Actor>(actor =>
           {
               actor.MapProperty(p => p.Name).SetElementName("name");
           });
       }
    }
}
```

How does the code look from the caller perspective?

Code Listing 83: Inserting movies by using POCO class mapping

```
private static void Main(string[] args)
{
    Movie[] movies = MovieManager.GetMovieList();
    BsonMapper.Map(); //map the class to the MongoDB representation.
    MovieManager.Insert<Movie>(movies, "moviesDb", "movies_poco").Wait();
}
```

Just to make a difference, we are storing the data to the **movies_poco** collection. This is purely for showing the functionality. The two methods are identical, and in a production system, we would use the same collection.

Let's also pay attention to the **BsonMappe.Map()** call. This should be executed only once, when the application starts.

The result of inserting the two movies is as follows. The conclusion is that the result obtained with the two methods is exactly the same.

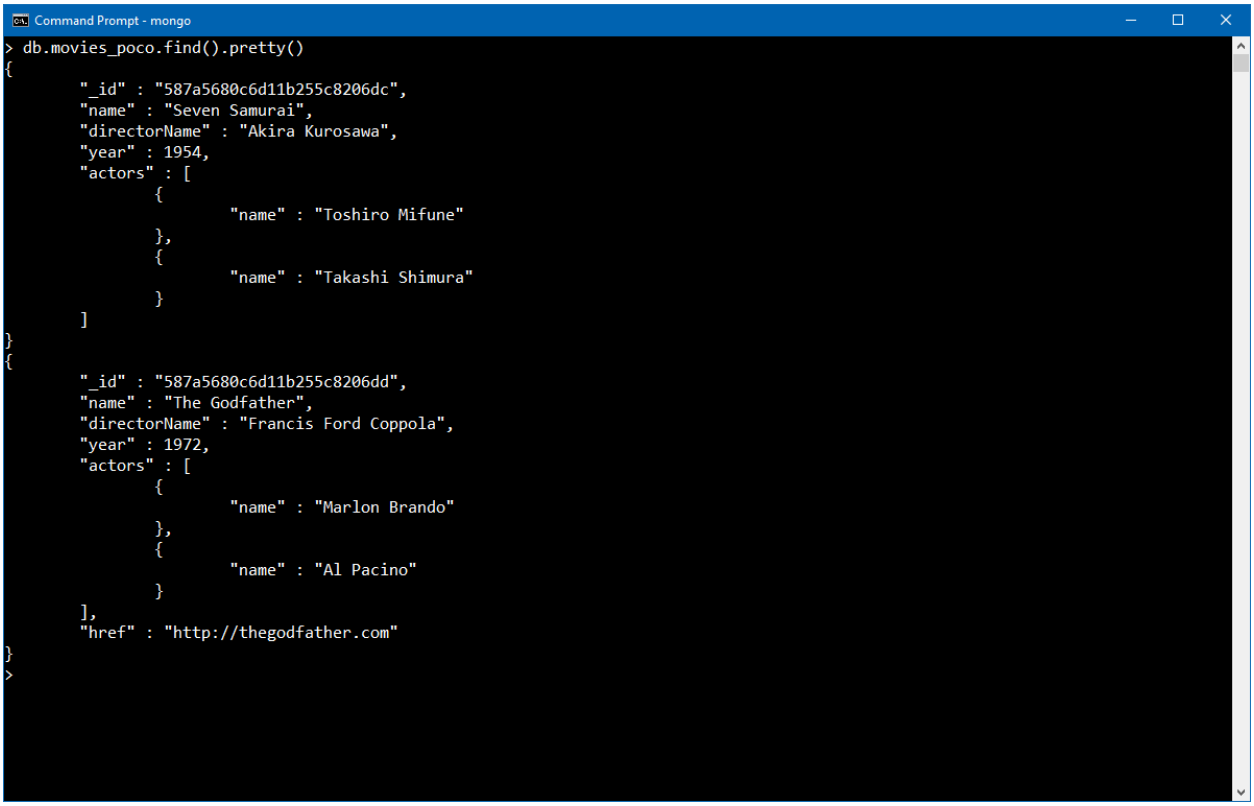


Figure 43: Movies as inserted in the database.

Chapter 9 Find (Query) Data in C#

In the previous chapters, we have gone through data retrieval and the various possibilities. We have also seen that the **find()** method was one of the very important entry points.

The MongoDB C# driver offers the **Find** and **FindAsync** methods to issue a query to retrieve data from a collection. As was the case when using the MongoDB shell, all queries made by using **Find** have the scope of a single collection.

The **FindAsync** method returns query results in a **IAsyncCursor**, while the **Find** method returns an object that implements the **IFindFluent** interface. To avoid iterating through the list, we can use the **ToListAsync** method to return the results as a list. This also means that all the documents will be held in memory; therefore, one must pay attention to the quantity of data being returned.

Returning all data from a collection

To return all the data from a collection, we simply don't have to specify any filter when calling the **FindAsync**() method. The following snippet shows a method that queries the collection and then uses the returned cursor object to iterate through the associated items. The important thing in this particular case is that the filter is actually a **BsonDocument**, which in this case is empty.

Code Listing 84: Finding movies as BsonDocuments

```
public async static void FindMoviesAsDocuments(string dbName, string collNa
me)
{
    var db = DatabaseHelper.GetDatabaseReference("localhost", dbName);
    var collection = db.GetCollection<BsonDocument>(collName);
    var filter = new BsonDocument();
    int count = 0;
    using (var cursor = await collection.FindAsync<BsonDocument>(filter))
    {
        while (await cursor.MoveNextAsync())
        {
            var batch = cursor.Current;
            foreach (var document in batch)
            {
                var movieName = document.GetElement("name").Value.ToString(
);
                Console.WriteLine("Movie Name: {0}", movieName);
                count++;
            }
        }
    }
}
```

```
/* Calling the method*/
MovieManager.FindMoviesAsDocuments(databaseName, "movies_bson");

/* Returns the following output*/

Movie Name: The Seven Samurai
Movie Name: The Godfather
```

Note that the result returned actually contains the **BsonDocuments**, and in order to get the values from the particular field, we need to use **document.GetElement("name").Value**.

If we were to query by using the **Movie** POCO object, then the search would look just a bit different. I've highlighted the differences in grey in the following code snippet.

Code Listing 85: Finding documents by using strongly typed movie collections

```
public async static void FindMoviesAsObjects(string dbName, string collName
)
{
    var db = DatabaseHelper.GetDatabaseReference("localhost", dbName);
    var collection = db.GetCollection<Movie>(collName);
    var filter = new BsonDocument();
    int count = 0;
    using (var cursor = await collection.FindAsync<Movie>(filter))
    {
        while (await cursor.MoveNextAsync())
        {
            var batch = cursor.Current;
            foreach (var movie in batch)
            {
                Console.WriteLine("Movie Name: {0}", movie.Name);
                count++;
            }
        }
    }
}

/* Calling the method*/
MovieManager.FindMoviesAsObjects(databaseName, "movies_poco");

/* Returns the following output*/

Movie Name: The Seven Samurai
Movie Name: The Godfather
```

Now that we have seen the basics, we can extend the example to contain the definition of the filter. The MongoDB C# driver offers the **FilterDefinitionBuilder**, which can help in defining the proper query without going into too many details of the MongoDB semantics (as we had to do by using the MongoDB shell). **FilterDefinition** supports both **BsonDocument** notation (name–value) and .NET expressions (lambda). The entry point for defining queries is

the **Builders<T>** class. As we will see, **Builders<T>** can support various types of operations such as filtering, sorting, or defining projections.

Let's quickly see some basic examples by using **Eq**, which stands for equal.

Code Listing 86: Statically typed vs. BsonDocument filter definition

```
/* Filter to retrieve movies where the name equals to "The Godfather" */
var expresssionFilter = Builders<Movie>.Filter.Eq(x => x.Name, "The Godfath
er");

/* Filter to retrieve movies where the name equals to "The Godfather"
 * by using BsonDocument notation */
var bsonFilter = Builders<BsonDocument>.Filter.Eq("name", "The Godfather");
```

Builder is quite flexible, and it allows defining all the supporting MongoDB query operators such as **Equal**, **Greater Than**, **Less Than**, and **Contains**. It is also possible to create various combinations between **Or** and **And**. The following code shows an example of an **Or** operator:

Code Listing 87: Building a filter by using the Or operator

```
/* find movies where the name is "The Godfather" OR "The Seven Samurai" */
var filter = Builders<Movie>.Filter.Or(new[]
{
    new ExpressionFilterDefinition<Movie>(x => x.Name == "The Godfather"),
    new ExpressionFilterDefinition<Movie>(x => x.Name == "The Seven Samurai
")
});
```

It is also very useful to know that is possible to use the C# operators **&** (binary **AND**) and **|** (binary **OR**) to build more complex and compile-time safe queries, such as the following:

Code Listing 88: Using C# conditional operators to build a filter

```
/* find movies where the name is "The Godfather" OR year > 1900 */
var builder = Builders<Movie>.Filter;
var query = builder.Eq("name", movieName) | builder.Gt("year", 1900);
var result = await collection.Find(query).ToListAsync();
```

By using this simple technique, we can change our original query so that we return the movies with a given name. The slight difference from the previous example is that we are converting the result to a list by using the **ToListAsync**() method.

In addition, we are also showing here how we can use the **Builder** to specify the sorting, which can be used to concatenate the various field sorting. In our case, we order **Ascending** by **Name**, and then **Descending** by **Year**.

Code Listing 89: Sorting the result

```
var collection = db.GetCollection<Movie>(collName);

var filter = Builders<Movie>.Filter.Eq(x => x.Name, movieName);
```

```
var movies = await collection.Find(filter).ToListAsync();
var sort = Builders<Movie>.Sort.Ascending(x => x.Name).Descending(x => x.Ye
ar);

foreach (var movie in movies)
{

Console.WriteLine("Match found: movie with name '{0}' exists", movie.Name);
}
```

The **Find** method also supports expressions as parameters, so this is also a valid query—and perhaps faster to write, as it doesn't need the **Builders** to be used.

Code Listing 90: Find with lambda expressions

```
var collection = db.GetCollection<Movie>(collName);

var movies = collection.Find(x => x.Name == "The Godfather");
```

Projecting data

In the RDBMS, one of the most natural things to do is to return just a subset of data for a given query. In MongoDB, this is called a projection of data. There are few ways of constructing the return data, but the best thing would be to start with an example, which we will then enhance slowly.

We can say that the main entry point for the projections is the **Builders** object—the same one used previously for filtering. **Builders** offers the possibility to define what data will be returned through the object **ProjectionDefinition**. There are two kinds of projections: one in which we know what the object to be returned (mapped to) is, and one in which we don't have the object representation of the data returned from the database.

Code Listing 91: Projection definition examples

```
/* using the ProjectDefinition object to specify an object that will only
return the _id and year as two attributes. */
ProjectionDefinition<Movie> projection = new BsonDocument("year", 1);

/* using the strongly typed Builders object to specify the return
attributes. */

var projection = Builders<Movie>.Projection
                            .Include("name")
                            .Include("year")
                            .Exclude("_id");
// or
var projection = Builders<Movie>.Projection
                            .Include(x => x.Name)
                            .Include(x => x.Year)
                            .Exclude(x => x.MovieId);
```

```
// or
var projection = Builders<Movie>.Projection.Expression(x =>
                            new { X = x.Name, Y = x.Year });
```

An example of specifying the return data in **BsonDocument** format is as follows:

Code Listing 92: Projection as BsonDocument

```
var collection = db.GetCollection<Movie>(collName);

var projection = Builders<Movie>.Projection
       .Include("name")
       .Include("year")
       .Exclude("_id");

var data = collection.Find(new BsonDocument())
           .Project<BsonDocument>(projection)
           .ToList();

foreach (var item in data)
{
    Console.WriteLine("Item retrieved {0}", item.ToString());
}
```

The structure being returned would be as follows:

Code Listing 93: data returned as defined by the projection

```
Item retrieved { "name" : "The Seven Samurai", "year" : 1954 }
Item retrieved { "name" : "The Godfather",     "year" : 1972 }
```

If we would like to have a strongly typed object, then it is as easy as specifying the **Project<Movie>** instead of **Project<BsonDocument>**. In the projection definition, we can use the expressions, which simplifies things since we don't have to remember the serialized attribute's name.

Code Listing 94: Projection defined as strongly typed Movie object

```
var collection = db.GetCollection<Movie>(collName);

var projection = Builders<Movie>.Projection
        .Include(x => x.Name)
        .Include(x => x.Year)
        .Exclude(x => x.MovieId);

var data = await collection.Find(new BsonDocument())
           .Project<Movie>(projection)
           .ToList();
```

```
foreach (Movie item in data)
{
    Console.WriteLine("Item retrieved {0}", item.ToString());
}
```

The objects returned will be of type **Movie**; however, only the name and year attributes will be populated with data, and other attributes will have the default value.

The **async** version of the method behaves a bit differently, and returns **IAsyncCursor**. In addition, there is no **Project** method, but the projection should be passed as part of the options.

Code Listing 95: Async version of the strongly typed projection definition

```
var collection = db.GetCollection<Movie>(collName);

var projection = Builders<Movie>.Projection
        .Include(x => x.Name)
        .Include(x => x.Year)
        .Exclude(x => x.MovieId);

var options = new FindOptions<Movie, BsonDocument>
{
    Projection = projection
};

var cursor = await collection.FindAsync(new BsonDocument(), options);
var data = cursor.ToList();

foreach (var item in data)
{
    Console.WriteLine("Item retrieved {0}", item.ToString());
}
```

Aggregation

In one of the previous chapters, we explained the aggregation and aggregation pipeline as it happens when using the MongoDB shell. As expected, the same can be done in C#.

The entry point of the functionality is the **Aggregate** method, which then can be expanded to specify the pipeline and the various options—pretty much what we have already gone through.

The following example shows how it is possible to group by a given field and calculate the count, in our case a count of the movies per year.

Code Listing 96: Aggregation of data by grouping

```
public static void AggregateMovies(string dbName, string collName)
{
    var db = DatabaseHelper.GetDatabaseReference("localhost", dbName);
```

```
    var collection = db.GetCollection<Movie>(collName);

    var data = collection.Aggregate()
        .Group(new BsonDocument
        {
            { "_id", "$year" },
            { "count", new BsonDocument("$sum", 1) }
        });

    foreach (var item in data.ToList())
    {
        Console.WriteLine("Item retrieved {0}", item.ToString());
    }
}
```

This query will return the **_id**, which will be the value of the year (hence the **$** sign in front of the attribute), and the **count** attribute with the actual value.

Code Listing 97: Aggregation result

```
Item retrieved { "_id" : 1972, "count" : 2 }
Item retrieved { "_id" : 1954, "count" : 1 }
```

It is also possible to specify the various stages in the aggregation pipeline. Therefore, we can add the **Match** before executing the grouping in order to prefilter the data we want to work against.

Code Listing 98: Aggregation by prefiltering the data to be grouped

```
var aggregate = collection.Aggregate()
    .Match(Builders<Movie>.Filter.Where(x => x.Name.Contains("Godfather")))
    .Group(new BsonDocument
    {
        {"_id", "$year"},
        {"count", new BsonDocument("$sum", 1)}
    });

var results = aggregate.ToList();
```

In this particular case, the **Match** works as we would normally filter the data. We have used the **Where** method in order to filter only the movies whose names include **Godfather**.

LINQ

The driver contains an implementation of LINQ that targets the underlying aggregation framework. The rich query language available in the aggregation framework maps very easily to the LINQ expression tree. This makes it possible to use the LINQ statements in order to perform queries.

The entry point is the **AsQueriable()** method that offers a world of possibilities in order to perform queries as we got used to with LINQ. **AsQueriable** is available at the collection level. There is support for the filtering, sorting, projecting, and grouping of data, and some basic functionality of joining to other collections.

Here's a quick example showing that both styles of LINQ are supported. The following code transforms the collection into **AsQueriable** and selects only the **Name** and **Age** from a movie. Therefore, we have only two attributes returned. This makes the projection of data very easy!

Code Listing 99: Using LINQ queries

```
var collection = db.GetCollection<Movie>(collName);

var query = from p in collection.AsQueryable()
            select new { p.Name, p.Age };

// both queries are equivalent.

var query = collection.AsQueryable().Select(p => new { p.Name, p.Age });
```

Applying a **Where** clause is as easy as calling the **Where** extension method.

Code Listing 100: LINQ query, data filtering by Where clause

```
var collection = db.GetCollection<Movie>(collName);

var query = collection.AsQueryable().Where(p => p.Age > 21);
```

Pagination becomes very easy with the **Take** and **Skip** methods.

Code Listing 101: Usage of Take and Skip

```
var collection = db.GetCollection<Movie>(collName);

var movies = collection.AsQueryable().Skip(10).Take(10).ToList();
```

Take will return a limited number of documents (in our case 10), while **Skip** makes sure to bypass the given number of documents. In our case, the documents from 11 to 20 will be returned.

Update data

The MongoDB C# driver offers quite a few ways to update the data. Here are just a few of the useful methods on a collection that can make it easy to search and manipulate documents:

Table 17: Update document methods

FindOneAndUpdate **FindOneAndUpdateAsync**	Updates one document based on a filter and, as a result, returns the updated document before or after the change.

UpdateMany **UpdateManyAsync**	Updates multiple documents based on a filter and returns the **UploadResult**.
UpdateOne **UpdateOneAsync**	Updates one document based on a filter and, as a result, returns the **UpdateResult**.
ReplaceOne **ReplaceOneAsync**	Replaces an entire document.
FindOneAndReplace **FindOneAndReplaceAsync**	Replaces one document based on a filter and, as a result, returns the replaced document before or after the change.

At first sight, the two methods **FindOneAndUpdate** and **UpdateOne** might seem to be pretty much identical; however, their use cases might differ. One returns a full document before or after the update operation, while the other just returns the information about the operation itself. If the data does not need to be returned, then **UpdateOne** is probably more efficient, as it doesn't have to perform another query and return data.

Updating a document doesn't differ very much from what we have already seen, as the patterns are pretty much the same: use the **Builders** object to define the query on which the update operation will be performed, and at the same time, use the **Builders** object to define what kind of update operation will be applied.

Let's go through the three different methods and show how it works in practice. The following example uses the **UpdateOne** method. As mentioned previously, we can see that in order to update the document by using the **Builers<T>.Filter**, we need to specify the query to find the document to be updated. Additionally, the interesting part is to define the **update** statement that happens through the **Set()** method, which can be written by using either the lambda expression and the value (as specified for the **Year**), or by manually supplying the name and the value. It is possible to specify more than one update by chaining multiple **Set** methods.

Code Listing 102: Updating a movie

```
public static void UpdateMovie(string dbName, string collName)
{
    var db = DatabaseHelper.GetDatabaseReference("localhost", dbName);
    var collection = db.GetCollection<Movie>(collName);

    var builder = Builders<Movie>.Filter;
    var filter = builder.Eq("name", "The Godfather");
    var update = Builders<Movie>.Update
                                .Set("name", "new name")
                                .Set(d => d.Year, 1900);

    UpdateResult result = collection.UpdateOne(filter, update);

    Console.WriteLine(result.ToBsonDocument());
}
```

UpdateMany comes with exactly the same signature; however, it would update multiple documents at the time.

FindOneAndUpdate

On the other side, **FindOneAndUpdate** becomes quite interesting with the options it can be supplied with. It makes it possible to do the following:

- Specify the projection: We can specify which attributes will be returned and excluded.
- Specify the return document: We can specify if we want the movie object in the state before or after the update has been performed.

Here is an example of using the **FindOneAndUpdate** method.

Code Listing 103: Example of using the FindOneAndUpdate

```
var filter = Builders<Movie>.Filter.Eq("name", "The Godfather");
var update = Builders<Movie>.Update
                            .Set("name", "new name")
                            .Set(d => d.Year, 1900);

var updateOptions = new FindOneAndUpdateOptions<Movie, Movie>()
{
    ReturnDocument = ReturnDocument.After,
    Projection = Builders<Movie>
        .Projection
        .Include(x => x.Year)
        .Include(x => x.Name)
};

Movie movie = collection.FindOneAndUpdate(filter, update, updateOptions);
```

As part of the update options, we have specified the **Projection**, which means that only the attributes specified as part of it will be returned by the method after the movie has been updated. So, when the **Movie** object is returned, it will be filled in only with the **_id**, **year**, and **name**. The rest of the attributes will have the default value. It is possible to choose between **ReturnDocument.After** and **ReturnDocument.Before**, which are the instructions to return the status of the movie before or after the update, respectively.

ReplaceOne

We use the **ReplaceOne** method to replace the entire document. Bear in mind that the **_id** field cannot be replaced, as it is immutable. The replacement document can have fields different from the original document. In the replacement document, you can omit the **_id** field, since the **_id** field is immutable. If you do include the **_id** field, it must be the same value as the existing value.

In the following example, we are showing how to replace a document. First, we are retrieving an already existing document from the database that will be replaced. In addition, there is a new instance of the **Movie**, which happens to be a different movie from the original. At the end, we call the **ReplaceOneAsync** method.

ReplaceOneAsync returns the **ReplaceOneResult** object, which contains properties such as **MatchedCount** and **ModifiedCount** that tell if the object to be modified has been found and modified.

Code Listing 104: Example of ReplaceOneAsync usage

```
var collection = db.GetCollection<Movie>(collName);

var builder = Builders<Movie>.Filter;
var filter = builder.Eq("name", "The Godfather");

//find the ID of the Godfather movie...
var theGodfather = await collection.FindAsync(filter);
var theGodfatherMovie = theGodfather.FirstOrDefault();

Movie replacementMovie = new Movie
{
    MovieId = theGodfatherMovie.MovieId,
    Name = "Mad Max: Fury Road",
    Year = 2015,
    Actors = new[]
    {
        new Actor {Name = "Tom Hardy"},
        new Actor {Name = "Charlize Theron"},
    },
    Director = "George Miller"
};
ReplaceOneResult r = await collection.ReplaceOneAsync(filter, replacementMo
vie);

Console.WriteLine(r.ToBsonDocument());
```

Delete data

In a very similar way to the update options, the MongoDB C# driver supports a few ways to delete data. The following methods on the collection can make it quite easy to delete the already existing documents:

Table 18: Methods that delete a document

FindOneAndDelete **FindOneAndDeleteAsync**	Deletes the first document in the collection that matches the filter. The sort parameter can be used to influence which document is updated.

DeleteMany **DeleteManyAsync**	Removes all documents that match the filter from a collection.
DeleteOne **DeleteOneAsync**	Deletes the first matching document based on a filter and, as a result, returns the **UpdateResult**.

DeleteOne and **DeleteMany** are pretty similar in their implementation. Both return the **DeleteResult**, which will inform us of the successful removal of the document through its **DeletedCount** property. Both of them accept either an expression or, as we have seen previously, a **FilterDefinition** that can be constructed using the **Builders** mechanism.

The following example uses the lambda expression as the **DeleteOneAsync** parameter:

Code Listing 105: Example of using the DeleteOneAsync and DeleteManyAsync

```
var collection = db.GetCollection<Movie>(collName);

DeleteResult result = await collection.DeleteOneAsync(m => m.Name == "The S
even Samurai");

//or

var result = await collection.DeleteManyAsync(m => m.Name == "The Seven Sam
urai" || m.Name == "Cabaret");
```

Here is an example that uses the **Builders**:

Code Listing 106: DeleteManyAsync with the specified filter

```
var collection = db.GetCollection<Movie>(collName);

var builder = Builders<Movie>.Filter;
var filter = builder.Eq("name", "The Godfather");
var result = await collection.DeleteManyAsync(filter);
```

FindOneAndDelete is a bit different, as it also offers the possibility to return the data as part of the deleting operation. It accepts the **FindOneAndDeleteOptions** to be passed in, with the ability to specify the sorting, which is the sorting on the collection before the delete happens (remember, only one document will be deleted), and the projection, which will return the data of the deleted document (obviously in the state it was in before being deleted).

Here's an example of how to use the **FindOneAndDeleteAsync** method:

Code Listing 107: FindOneAndDeleteAsync with options defined

```
var coll = db.GetCollection<Movie>(collName);

var options = new FindOneAndDeleteOptions<Movie, BsonDocument>
{
```

```
    Sort = Builders<Movie>.Sort.Ascending(x => x.Name),
    Projection = Builders<Movie>.Projection.Include(x => x.MovieId)
};
var result = await coll.FindOneAndDeleteAsync(m => m.Name == "Cabaret", opt
ions);
```

This will return a **`BsonDocument`** that contains only the attribute **`_id`**.

Conclusion

In this lengthy chapter, we have seen the most important aspects when it comes to manipulating data in MongoDB. Certainly, not every possible mechanism has been mentioned, as there are many other hidden features. I tried to illustrate the features that are going to be used the most and that have patterns we can follow.

After reading this chapter, the hope is that you have become aware of various possibilities and techniques for effectively using the MongoDB driver.

Chapter 10 Binary Data (File Handling) in C#

In Chapter 1, we mentioned the fact that the BSON documents stored in the MongoDB database have a hard limit of 16 megabytes. This is for both memory usage and performance reasons. MongoDB offers a possibility, however, to store files larger than 16 megabytes.

GridFS is a MongoDB specification and a way of storing binary information larger than the maximum document size. **GridFS** It is kind of a file system to store files, but its data is stored within MongoDB collections.

When the file is uploaded to **GridFS**, instead of storing a file in a single document, **GridFS** divides a file into parts called *chunks*. Each chunk is a separate document and has a maximum of 255 kilobytes of data. When the file is downloaded (retrieved) from **GridFS**, the original content is reassembled.

GridFS by default uses two collections to store the file's metadata (**fs.files)** and chunks (**fs.chunks)**. As happens for any other document in MongoDB, each chunk is identified by its unique **_id**, which is of type **ObjectId** field. The **fs.files** acts as a parent document. The **files_id** assigned to a chunk holds a reference to its parent.

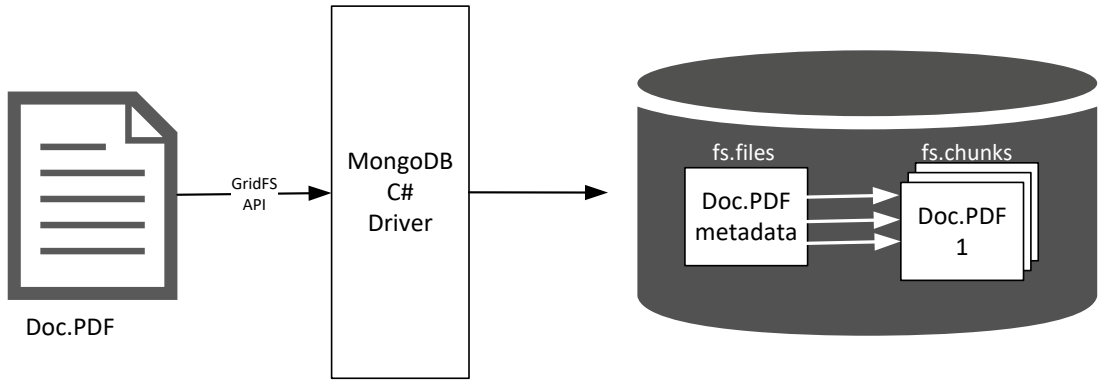


Figure 44: GridFS chunks.

When working in C#, we have to always use an object called **GridFSBucket** in order to interact with the underlying GridFS system. We should avoid directly accessing the underlying collections.

In order to use the GridFS from the MongoDB driver, we have to install the **MongoDB.Driver.GridFS** package from NuGet.

Uploading files

There are mainly two ways of uploading a file: either by specifying the location on the disk (from the client), or by submitting the data to a **Stream** object that the driver supplies.

One of the easiest ways to upload a file is by using the byte array. We will store the file in the database called **file_store**.

In the following example, we can see that we are instantiating a new instance of the **GridFSBucket** object, which will then be responsible for uploading the file via the **UploadFromBytes** method. To get the file byte data, we simply use the .NET standard **File.ReadAllBytes** method from the **System.IO** namespace.

Code Listing 108: Storing a file into MongoDB via GridFSBucket

```
public static void UploadFile()
{
    var database = DatabaseHelper.GetDatabaseReference("localhost", "file_s
tore");

    IGridFSBucket bucket = new GridFSBucket(database);

    byte[] source = File.ReadAllBytes("sample.pdf");

    ObjectId id =  bucket.UploadFromBytes("sample.pdf", source);

    Console.WriteLine(id.ToString());
}
```

This code generates the result shown in Figure 45. We can see that one entry in the **fs.files** collection has been created, and one entry (for the sake of space, we cannot see the full content) in the **fs.chunks**. We can also see that the chunk itself contains a pointer to the **fs.files** metadata through the **files_id** property.

The result returned by the method is the newly created **ObjectId**.

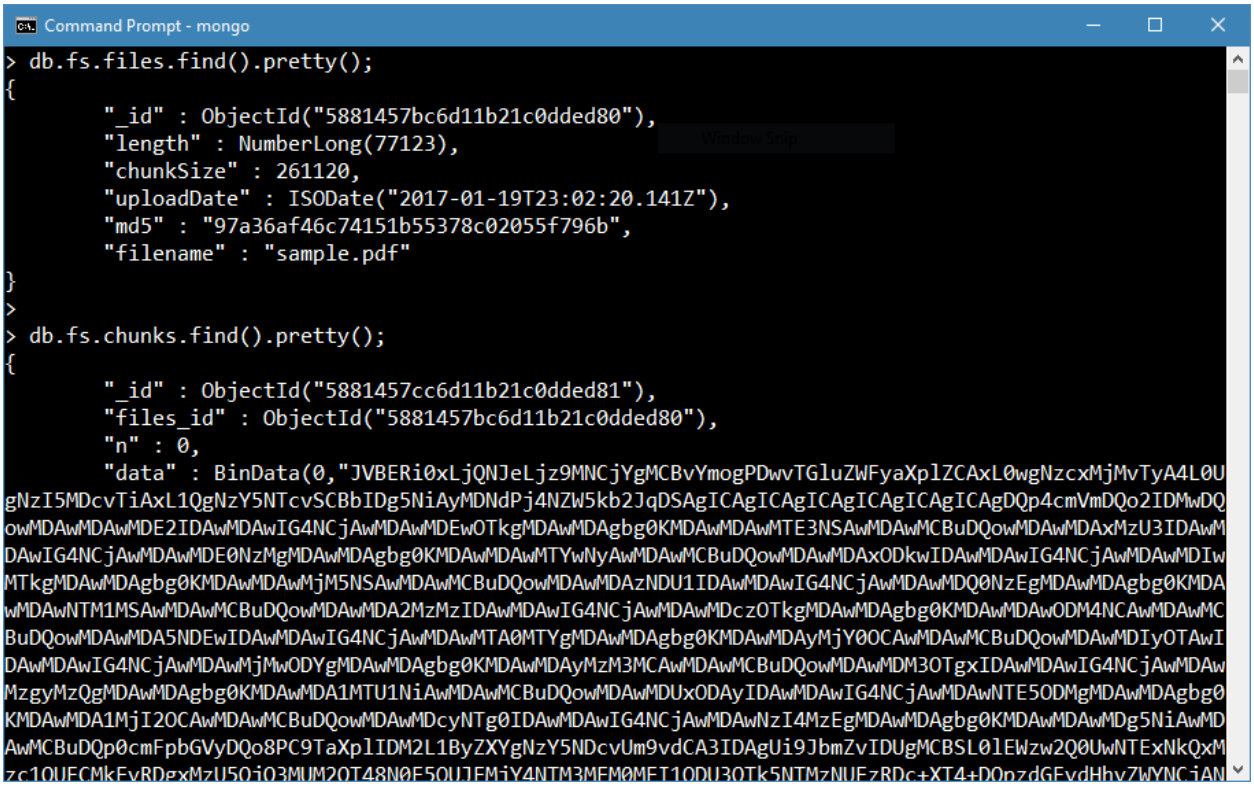


Figure 45: Database after uploading the file

Uploading files from a stream

There are also other ways of uploading the files, such as uploading from a file stream. The idea here is that instead of returning the byte array of a file, we supply the input information in the form of a **`Stream`** object.

Code Listing 109: Uploading a file from a stream

```
public static void UploadFileFromAStream()
{
    var database = DatabaseHelper.GetDatabaseReference("localhost", "file_s
tore");

    IGridFSBucket bucket = new GridFSBucket(database);
    Stream stream = File.Open("sample.pdf", FileMode.Open);

    var options = new GridFSUploadOptions()
    {
        Metadata = new BsonDocument()
        {
            {"author", "Mark Twain"},
            {"year", 1900}
        }
    };
```

```
    var id = bucket.UploadFromStream("sample.pdf", stream, options);

    Console.WriteLine(id.ToString());
}
```

In pretty much the same way as before, when updating a byte array by opening a file through **`File.Open`**, we obtain the stream object and pass it to the **`GridFSBucket`**.

One interesting thing is that there is another parameter available to pass the options to the **`UploadFromStream`**, where we can add some metadata to the file being uploaded. The metadata is again a **`BsonDocument`**, and it can have any structure we like.

After running the previous code, we get the following stored in the database. We can see that the metadata is now shown in the **`fs.files`** collection. The **`fs.chunks`**, as displayed in Figure 46, doesn't show the data attribute (which contains the byte code) intentionally.

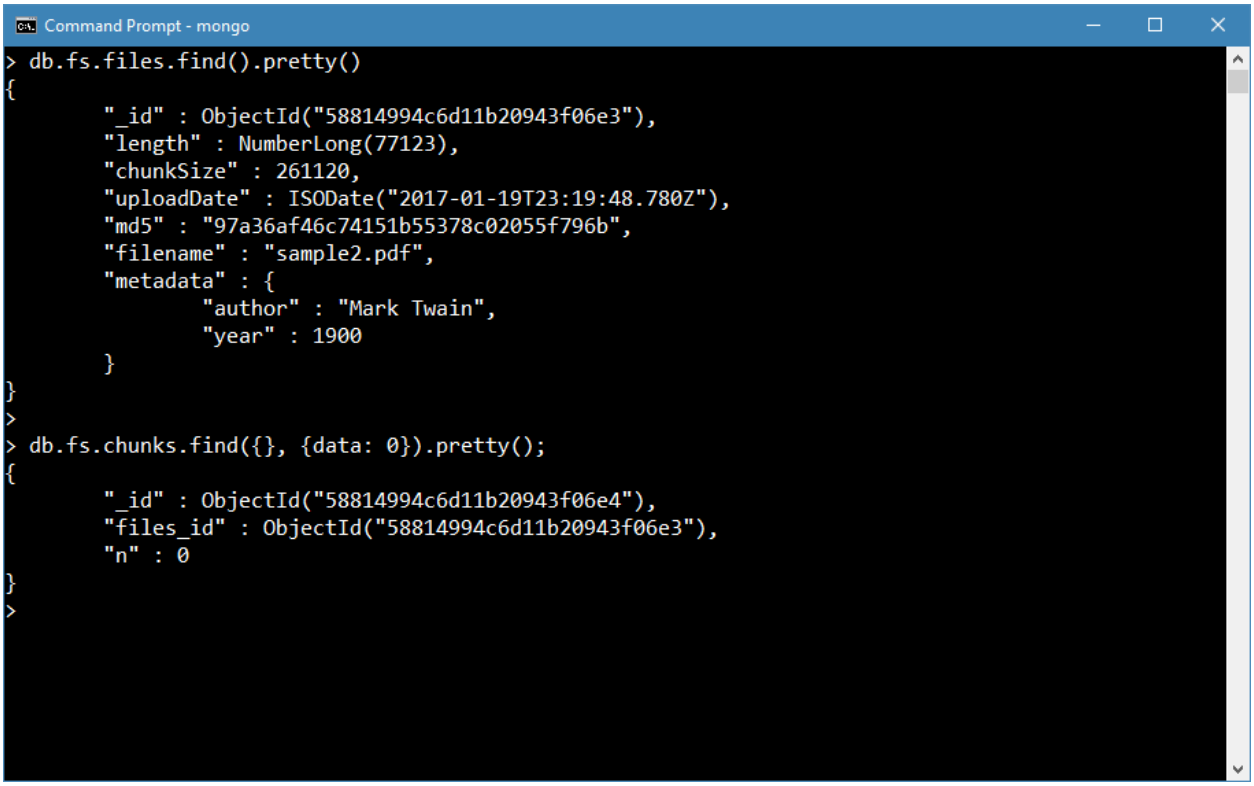


Figure 46: File uploaded with metadata

One very important thing to mention is the fact that if the file with the same name gets sent to the **`GridFS`**, then this file will be treated as a new version of that particular file.

Downloading files

There are a few ways to download a file from **GridFS**; the two main approaches are downloading the file as a byte array, and receiving back a **Stream** object from the driver.

It's possible to download files by using one of the following methods:

Table 19: Methods for downloading files

DownloadAsBytes **DownloadAsBytesAsync**	Downloads a file stored in GridFS and returns it as a byte array.
DownloadAsBytesByName **DownloadAsBytesByNameAsync**	Downloads a file stored in GridFS and returns it as a byte array.
DownloadToStream **DownloadToStreamAsync**	Downloads a file stored in GridFS and writes the contents to a stream.
DownloadToStreamByName **DownloadToStreamByNameAsync**	Downloads a file stored in GridFS and writes the contents to a stream.

DownloadAsBytes

One of the easiest ways to download files is to receive back the byte array. However, a bit of attention needs to be paid to the fact that the data of the byte array will be held in memory, so downloading large files can result in a high usage of memory.

Code Listing 110: Example of DownloadAsBytesAsync method

```
public static async Task DownloadFile()
{
    var database = DatabaseHelper.GetDatabaseReference("localhost", "file_s
tore");
    var bucket = new GridFSBucket(database);

    var filter = Builders<GridFSFileInfo<ObjectId>>

.Filter.Eq(x => x.Filename, "sample2.pdf");

    var searchResult = await bucket.FindAsync(filter);
    var fileEntry = searchResult.FirstOrDefault();

    byte[] content = await bucket.DownloadAsBytesAsync(fileEntry.Id);

    File.WriteAllBytes("C:\\temp\\sample2.pdf", content);
}
```

As the **DownloadAsBytes** requires the **ObjectId** of the file to be downloaded, we need to find some information about the file itself before calling the method. As we have done previously, we are creating a filter that will retrieve a file by searching by name. A particularity of the filter is that it uses **GridFSFileInfo<ObjectId>** as the generic parameter. This helps us to have strongly typed attributes of the file (**x.Filename**).

After receiving the data, we can safely save the file on the disk by calling the static methods in the standard .NET **File** class.

At the same time, downloading by name is very similar, with the only difference being that the name is being passed as the input parameter rather than the **ObjectId**. As we happen to know the name of the file, we don't need to search for its existence before downloading.

Code Listing 111: Example of DownloadAsBytesByNameAsync method

```
public static async Task DownloadFileAsBytesByName()
{
    var database = DatabaseHelper.GetDatabaseReference("localhost", "file_s
tore");

    IGridFSBucket bucket = new GridFSBucket(database);

    byte[] content = await bucket.DownloadAsBytesByNameAsync("sample2.pdf")
;

    File.WriteAllBytes("C:\\temp\\sample2.pdf", content);

    System.Diagnostics.Process.Start("C:\\temp\\sample2.pdf");
}
```

Download to a stream

Downloading to a stream is not that much different from the previously described methods. Obviously, the big difference is the object returned back, in this case, a **Stream**. In the following code, we are showing how to use the **DownloadToStreamAsync** method in order to download and store the data as a file.

Code Listing 112: Example usage of DownloadtoStreamAsync method

```
public static async Task DownloadFileToStream()
{
    var database = DatabaseHelper.GetDatabaseReference("localhost", "file_s
tore");

    IGridFSBucket bucket = new GridFSBucket(database);

    var filter = Builders<GridFSFileInfo<ObjectId>>

.Filter.Eq(x => x.Filename, "sample2.pdf");
```

```
    var searchResult = await bucket.FindAsync(filter);
    var fileEntry = searchResult.FirstOrDefault();

    var file = "c:\\temp\\mystream.pdf";
    using (Stream fs = new FileStream(file, FileMode.CreateNew, FileAccess.
Write))
    {
        await bucket.DownloadToStreamAsync(fileEntry.Id, fs);

        fs.Close();
    }
}
```

As we did for **DownloadAsBytes**, we require the **ObjectId** of the file to download; therefore, we are performing a search beforehand.

The interesting part of this method is the fact that the **Stream** gets created beforehand, and the stream is closed afterward. One very important aspect is that the stream management is left to the application itself.

Here is an example that uses the **DownloadToStreamByNameAsync**:

Code Listing 113: Example of using DownloadToStreamByNameAsync method

```
var file = "c:\\temp\\mystream2.pdf";
using (Stream fs = new FileStream(file, FileMode.CreateNew, FileAccess.Writ
e)
)
{
    await bucket.DownloadToStreamByNameAsync("sample2.pdf", fs);

    fs.Close();
}
```

Chapter 11 Back Up and Restore

In this chapter, you will learn how to back up and restore a database in MongoDB.

Back up

MongoDB provides a tool called **mongodump.exe** to back up your MongoDB database. By calling the **mongodump --help**, you will get quite a big list of options that can be used.

The most basic example would be backing up a database into a specific folder, which can be achieved as follows:

Code Listing 114: mongodump usage

```
mongodump -h localhost --db mydb -o c:\backup
```

Where:

- **-h** represents the host where the MongoDB runs.
- **--db** represents the database to be backed up.
- **-o** contains the output folder.

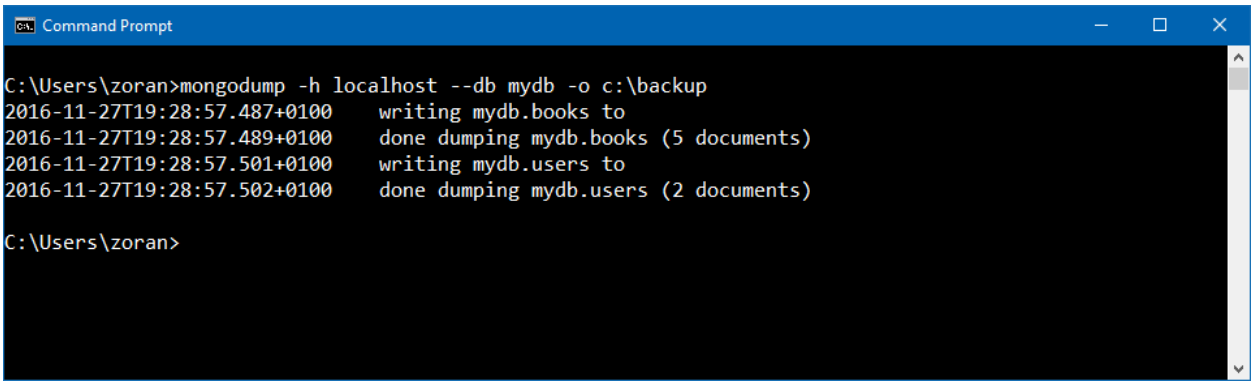


Figure 47: Backup command.

After running the command on the filesystem, we can see that the data has been exported correctly, and that MongoDB has created a folder called **mydb**.

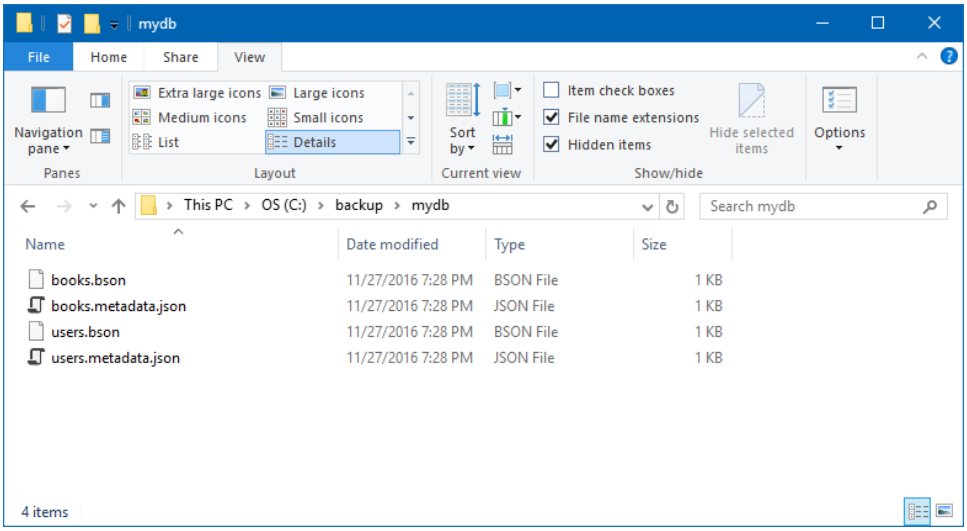


Figure 48: Backup folder.

The backup folder contains two types of files: **`.bson`**, which contains a **`bson`** copy of the data (bson being in binary format is not human-readable), and **`*metadata.json`**, which contains the configuration information of the collection itself, such as indexes.

Restore

After backing up the data, you can also restore it to MongoDB. To restore the data, you use **`mongorestore.exe`**. If you type it in the Command Prompt window and press **Enter**, you will get a response as shown in Figure 49.

The basic operation to perform is to restore the database from the previously taken backup. This is obtained by running the following command:

Code Listing 115: mongorestore usage

```
mongorestore -h localhost --db mydb --drop c:\backup
```

- **`-h`** represents the host where the MongoDB runs.
- **`--db`** represents the database to be restored.
- **`--drop`** contains the information of dropping the collection before recreating it.

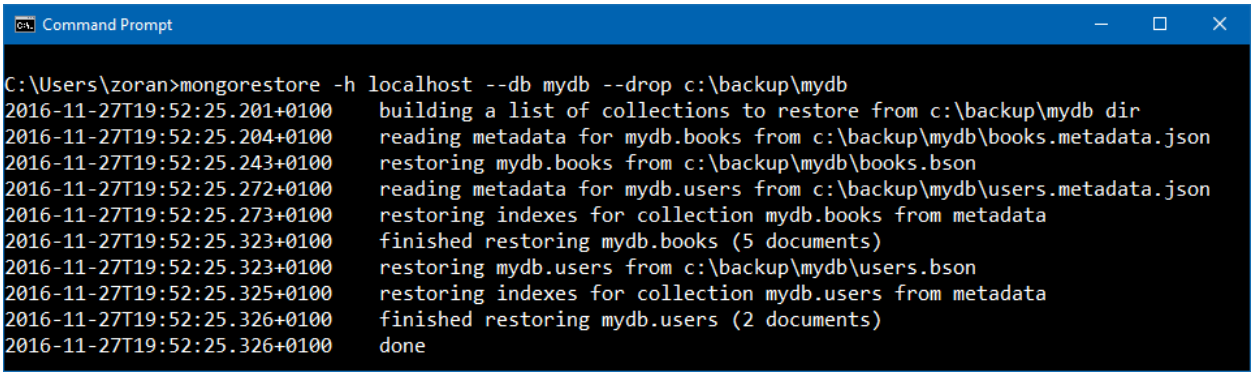


Figure 49: Restoring the database.

Final Words

In this book, we have touched on the most important aspects of the MongoDB database that the application developer should be aware of—from the theory about NoSQL and the document definition, to the usage of the Mongo Shell, and at the end, the examples of the MongoDB C# driver APIs for Microsoft .NET framework.

As with other books in the Syncfusion *Succinctly* series, what is mentioned in this book should give you a good starting point not only for designing an application, but also to start exploring more advanced options on your own, as many things are built around the mentioned concepts.

The administration part of the database was intentionally omitted (other than the short chapter about the backup and restore), as this book is more oriented to application developers.

I would like to thank you for reading this book, and hope that I've managed to fulfill the expectations you might have had when you started it.