

COM661 Full-Stack Strategies and Development

Practical C1: NoSQL and MongoDB

Aims

- To compare the relational and document store models for databases
- To provide guidelines for the design of document store databases
- To introduce the MongoDB server and shell
- To explain the purpose and structure of MongoDB `_id` values
- To introduce the `find()` method to search a MongoDB collections
- To demonstrate the `update()` and `remove()` methods for modifying documents and collections
- To demonstrate the `drop()` method for deleting collections
- To compare BSON and JSON as alternatives for data transfer

Contents

C1.1 NOSQL DATABASES	2
C1.1.1 RELATIONS VS DOCUMENTS	2
C1.1.2 DOCUMENT-BASED DESIGN	4
C1.2 INTRODUCING MONGODB.....	6
C1.2.1 MONGODB ID VALUES	7
C1.2.2 THE MONGODB SHELL	8
C1.2.3 LAUNCHING A MONGODB SERVER	8
C1.2.4 WORKING WITH DATABASES, COLLECTIONS AND DOCUMENTS	9
C1.3 BASIC MONGODB COMMANDS	12
C1.3.1 BASIC SEARCHING	12
C1.3.2 SORTING QUERY RESULTS	14
C1.3.3 UPDATING DOCUMENTS	14
C1.3.4 DELETING DOCUMENTS AND COLLECTIONS.....	16
C1.4 EXPORTING AND IMPORTING DATA	17
C1.4.1 EXPORTING AND IMPORTING BSON DATA.....	17
C1.4.2 EXPORTING AND IMPORTING JSON DATA	18
C1.5 FURTHER INFORMATION	21

C1.1 NoSQL Databases

NoSQL refers to a collection of database organisation models that offer alternatives to the tabular relations used in relational databases. Among the most popular NoSQL models are Key-value stores, Document Stores and Graph databases.

MongoDB is an example of a document store database, so we will begin by examining the main differences between this and the relational architecture that you are familiar with.

C1.1.1 Relations vs Documents

To illustrate the difference between relational and document store databases, consider a potential relational structure for a database to support a simple blog application, consisting of blog posts and comments.

The relational approach is to maintain separate data items in separate tables, with relationships between them identified by foreign key fields. For example, consider the structure illustrated in Figure C1.1 where the first two entries in the COMMENTS table can be seen to refer to the first entry in the POSTS table by use of the foreign key field **PostID**

POSTS			COMMENTS		
PostID	Title	Content	CommentID	PostID	Comment
1	My first post	Some content	1	1	Commenting on your first post
2	My second post	Some content	2	1	Also on your first post
3	My third post	Some content	3	2	Commenting on your second post

Figure C1.1 Relational Table Structure

In order to display a post and its comments, the application would need to query both tables, retrieving a row of data from the POSTS table as well as all rows in the COMMENTS table where the **PostID** value matches that in the row being retrieved from the POSTS table.

Relational databases are designed for **efficiency of storage** and a primary aim when designing a relational structure is to eliminate duplication of data. Each piece of data should only be stored once, so that when any information is updated, it only needs to be changed in one place. The rules that enforce this make up the process known as **normalisation**.

A document store database, on the other hand, is designed for **efficiency of retrieval**. The design is governed by the application in which the database will be used, so that (as far as possible) all of the information required should be retrieved in a single operation (i.e. one query, with no joins between different collections).

In the blog example, above, the most commonly performed operation will be the retrieval and display of posts and their comments, so the same dataset might be represented by the following JSON-style structure.

```
[
  {
    "PostID" : 1,
    "Title" : "My first post",
    "Content" : "Some content",
    "Comments" : [
      {
        "CommentID" : 1,
        "Content" : "Commenting on your first post"
      },
      {
        "CommentID" : 2,
        "Content" : "Also on your first post"
      }
    ]
  },
  {
    "PostID" : 2,
    "Title" : "My second post",
    "Content" : "Some content",
    "Comments" : [
      {
        "CommentID" : 3,
        "Content" : "Commenting on your second post"
      }
    ]
  },
  {
    "PostID" : 3,
    "Title" : "My third post",
    "Content" : "Some content"
  }
]
```

Here, the entire structure is represented as an array (enclosed within [...]), with each post defined as a JSON object (enclosed with { ... }). Each object (post) is specified as a key/value pair, where the key is the name of the element (i.e. the column name in the relational table) and the value is the data element that is stored. Note that comments on posts are embedded into the structure, with the **Comments** item defined as an array of comment objects. Note also, that when a field in an object is not required (e.g. the third post has no comments), then it can be simply left out. This is in contrast to the relational model which demands that each field has a value (even if that value is NULL).

C1.1.2 Document-based Design

The principle of efficiency of retrieval has a significant effect on the way in which we approach database design. In a relational architecture, there is typically one “correct” normalised structure for any given set of data, and all applications that make use of this database will interact with it according to this structure. However, in a document store database we must consider the application when designing the database so that we produce the most efficient data access platform. This may result in two applications that use the same raw data set with different preferred database architectures – because of the way in which they consume the data.

Consider the addition of a ‘users’ collection to our blog structure. We may want to store a set of various information about the user such as a name, email address, short bio, star sign, favourite colour, etc., – so an obvious approach would be to have a separate structure for users such as that shown below.

```
[
  {
    "UserID" : 1,
    "UserName" : "Adrian",
    "DisplayName" : "Adrian Moore",
    "Bio" : "Senior Lecturer"
  },
  {
    "UserID" : 2,
    "UserName" : "Jose",
    "DisplayName" : "Jose Santos",
    "Bio" : "Course Director"
  }
]
```

An application that needs to display all details pertaining to an individual user will simply read the appropriate object from this collection and retrieve each field.

However, it is obvious that we may want to display the name of a user alongside any posts that they have made, but maintaining this information in a separate collection goes against the principle of efficiency of retrieval – as the information on a post and the user who contributed it would be spread across two collections.

The solution is one that is alien to a relational database designer – we simply store the information in each place where it is needed – giving the following structure for the collection of posts and comments.

```
[
  {
    "PostID" : 1,
    "Title" : "My first post",
    "Content" : "Some content",
    "Comments" : [
      {
        "CommentID" : 1,
        "Content" : "Commenting on your first post"
      },
      {
        "CommentID" : 2,
        "Content" : "Also on your first post"
      }
    ],
    "Author" : {
      "UserId" : 1,
      "DisplayName" : "Adrian Moore"
    }
  },
  {
    "PostID" : 2,
    "Title" : "My second post",
    "Content" : "Some content",
    "Comments" : [
      {
        "CommentID" : 3,
        "Content" : "Commenting on your second post"
      }
    ],
    "Author" : {
      "UserId" : 1,
```

```

        "DisplayName" : "Adrian Moore"
    }
},
{
    "PostID" : 3,
    "Title" : "My third post",
    "Content" : "Some content" ,
    "Author" : {
        "UserId" : 2,
        "DisplayName" : "Jose Santos"
    }
}
]

```

Under this scheme, we are storing some of the user information twice – all user details are held in the “Users” collection so that the page that shows a user’s information can retrieve it all from one place, but some details are duplicated in the “Posts” collection so that we can satisfy the ‘view posts’ operation with a single database query.

The obvious drawback of this is that if a user decided to change their *DisplayName*, we would need to open the database and change every instance of it. This would be a very expensive operation, but its cost is outweighed by the benefit of the simple single-shot data retrieval for the most commonly performed operation.

In general, there are three main principles for design of document store database structures:

1. Keep the number of collections (tables) to a minimum
2. Keep each page or view to a single database query (no joins)
3. Optimise for the most common operations – even at the expense of less common tasks

C1.2 Introducing MongoDB

MongoDB (derived from **Humongous Database**) is a document store database that organises information as **collections**. In our Posts example from the previous section, the data represents a single collection, which is made up of a number of **documents** – each

document being a specification of a single post. Therefore, our **Posts** collection comprised three documents, while our **Users** collection was made up of two documents.

Although we have used JSON as the means of describing the collections, MongoDB actually uses a notation called **BSON** (pronounced *bi-son*) – a binary encoding of JSON that maintains the flexibility and ease of use of JSON, while adding the speed advantages of a binary format. However, MongoDB accepts JSON as input and produces JSON as output, so we as developers do not need to be concerned with the internal representation.

C1.2.1 MongoDB ID values

In our example, we included **ID** fields for **Posts**, **Comments**, **Authors** and **Users**. In fact, MongoDB will generate **ID** values automatically for each new document (or sub-document) that is created – so we never need to provide these ourselves.

In MongoDB, the ID field is always called **_id** and has a value that is defined as an **ObjectId()** with a long alphanumeric string derived from the date, time, machine identifier, process identifier and a counter – so no two **_id** values will be identical even across different installations. For example, a MongoDB implementation of our **Posts** collection might have auto-generated **_id** values as follows.

```
[
  {
    "_id" : ObjectId("165de4208b234b2140"),
    "Title" : "My first post",
    "Content" : "Some content",
    "Comments" : [
      {
        "_id" : ObjectId("567be1208b234b5778"),
        "Content" : "Commenting on your first post"
      },
      {
        "_id" : ObjectId("bc009208b2738f6e5"),
        "Content" : "Also on your first post"
      }
    ],
    "Author" : {
      "_id" : ObjectId("5e53c4208b234012b4"),
      "DisplayName" : "Adrian Moore"
    }
  },
  ...
]
```

C1.2.2 The MongoDB Shell

MongoDB has been provided for you on the lab machines, but if you want to install it on your own machine, you can obtain it free from

<https://www.mongodb.com/try/download/community2>.

Note: Installation notes for MongoDB on all platforms can be found at <https://www.mongodb.com/docs/v6.0/installation/>

The only piece of setup we need to do is to provide a data folder in which MongoDB will store the databases. On your personal machine, this will be a folder **C:\data\db** – but the lab installation should be set to **P:\data\db** so that your work will be available to you regardless of which lab machine you use.

Check your **P:** drive for the presence of a folder **P:\data\db** and, if it does not exist, create it now by opening a Command window, navigating to the **P:** drive and issuing the commands

```
P:\> mkdir data
P:\> cd data
P:\data> mkdir db
```

C1.2.3 Launching a MongoDB server

Note: If, during installation, you opt to “Run MongoDB as a service”, you will not need to launch the server – it will automatically run when your machine boots up.

In order to use MongoDB, we need to launch a MongoDB server that will service requests that are made either from the Mongo Shell or from other applications. You can launch the MongoDB server by issuing the command

```
P:\> mongod --dbpath=p:\data\db
```

which should result in output as illustrated in Figure C1.2 below.

Note: The **--dbPath** flag in the command above is only needed on the lab machines to indicate to MongoDB that you want to explicitly set the path to the database files. On your own machine, the command **mongod** should be sufficient.


```
adrianmoore — mongod — 80x24
[Adrians-Air-3:~ adrianmoore$ mongod
2017-02-01T20:41:43.237+0000 I JOURNAL [initandlisten] journal dir=/data/db/jou
rnal
2017-02-01T20:41:43.237+0000 I JOURNAL [initandlisten] recover : no journal fil
es present, no recovery needed
2017-02-01T20:41:43.259+0000 I JOURNAL [durability] Durability thread started
2017-02-01T20:41:43.259+0000 I JOURNAL [journal writer] Journal writer thread s
tarted
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] MongoDB starting : pid=5
49 port=27017 dbpath=/data/db 64-bit host=Adrians-Air-3.home
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten]
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] ** WARNING: soft rlimits
too low. Number of files is 256, should be at least 1000
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] db version v3.0.7
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] git version: nogitversio
n
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] build info: Darwin yosem
itevm.local 14.5.0 Darwin Kernel Version 14.5.0: Wed Jul 29 02:26:53 PDT 2015; r
oot:xnu-2782.40.9~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] allocator: system
2017-02-01T20:41:43.259+0000 I CONTROL [initandlisten] options: {}
2017-02-01T20:41:43.353+0000 I NETWORK [initandlisten] waiting for connections
on port 27017
]
```

Figure C1.2 MongoDB server

We leave this Command window open with the server application running while we interact with the database.

C1.2.4 Working with Databases, Collections and Documents

We can verify that MongoDB is installed and that a server is running by opening a new Command window (remember to leave the previous one open and running) and issuing the command to enter the Mongo Shell as follows.

```
P:\> mongo
```

Note: IN the latest version of MongoDB, the **mongo** shell has been replaced by one called **mongosh**. The new version has an improved layout and appearance but is otherwise functionally equivalent.

After some initial start-up and status messages, you should see the MongoDB prompt **>**, at which we can enter MongoDB Shell commands.

First, we will ask MongoDB to list the databases available to us by the command

```
> show dbs
```

By default, databases called **admin**, **config**, **local** are available on most installations (though this may vary). We can switch to our desired database by the command **use**, for example

```
> use local
```

To add a new database, we simply **use** one that is not already present. For example, to create a new database called **first**, we issue the command

```
> use first
```

and create a collection within this database by the command

```
> db.createCollection("fullStack")
```

where the object **db** refers to the currently selected database (the subject of the most recent **use** command) and the parameter **fullStack** is the name to be given to the new collection.

<p>Note: To list the collections in a database, you can issue the command</p> <pre>> show collections</pre>

To add a document to a collection, we need to **insert** it into a collection on the currently active database by the command

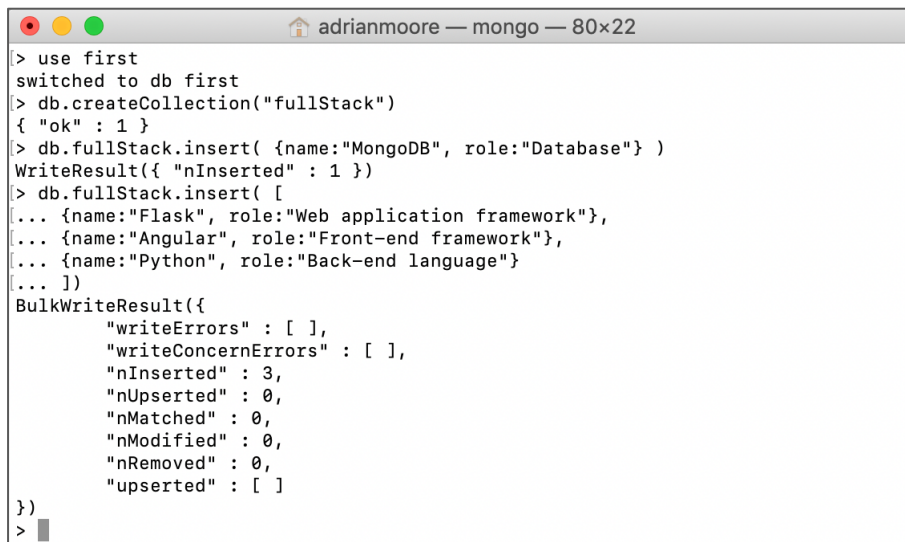
```
> db.fullStack.insert( { name:"MongoDB", role:"Database" } )
```

We should now see confirmation that have inserted one document into the collection.

To add multiple documents at once, we can simply pass an a JSON array as the parameter to **insert()**

```
> db.fullStack.insert( [
    { name: "Flask", role: "Web application framework" },
    { name: "Angular", role: "Front-end framework" },
    { name: "Python", role: "Back-end language" }
] )
```

Figure C1.3 illustrates the output generated by the above sequence. Note that you can enter JSON data over multiple lines by using the **<return>** key.



```
> use first
switched to db first
> db.createCollection("fullStack")
{ "ok" : 1 }
> db.fullStack.insert( {name:"MongoDB", role:"Database"} )
WriteResult({ "nInserted" : 1 })
> db.fullStack.insert( [
... {name:"Flask", role:"Web application framework"},
... {name:"Angular", role:"Front-end framework"},
... {name:"Python", role:"Back-end language"}
... ] )
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
>
```

Figure C1.3 Using the Mongo Shell

Do it now!	Follow the sequence of instructions above to enter the Mongo shell, create the database and collection and add the documents to the collection. Verify that you receive output similar to that shown in Figure C1.3 above
-------------------	---

Try it now!	Add a second collection of your choice to the first database and insert at least three documents to it. Each document should contain at least two fields – but remember that the fields do not have to be consistent across the documents in the collection.
--------------------	--

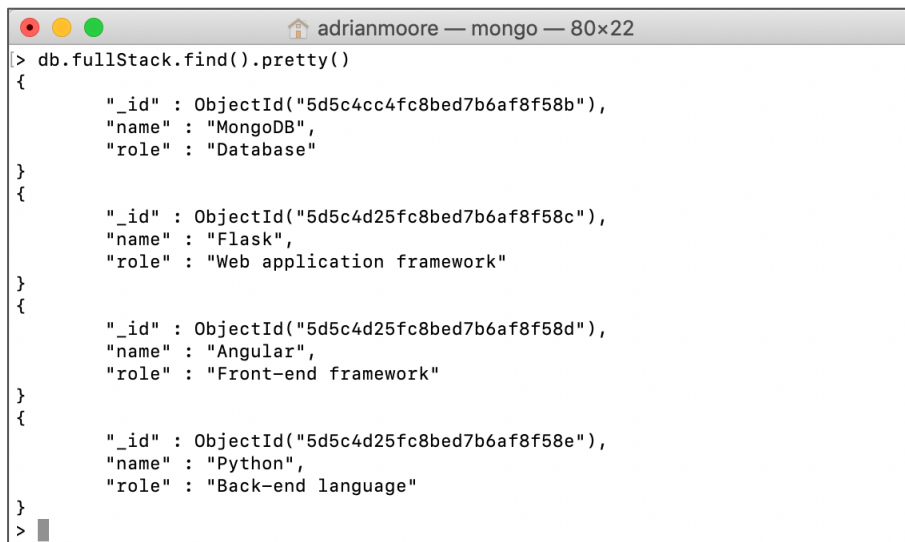
To retrieve documents from a collection, we chain the **find()** method to the database and collection

```
> db.fullStack.find()
```

In order to format this output in a more readable format, we can additionally chain the method **pretty()** to the command

```
> db.fullStack.find().pretty()
```

Figure C1.4 illustrates the output when retrieving the collection and passing the output through the **pretty()** method. Note the format of the **_id** field that was automatically assigned by MongoDB as each document was generated.



```
> db.fullStack.find().pretty()
{
  "_id" : ObjectId("5d5c4cc4fc8bed7b6af8f58b"),
  "name" : "MongoDB",
  "role" : "Database"
}
{
  "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58c"),
  "name" : "Flask",
  "role" : "Web application framework"
}
{
  "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58d"),
  "name" : "Angular",
  "role" : "Front-end framework"
}
{
  "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58e"),
  "name" : "Python",
  "role" : "Back-end language"
}
>
```

Figure C1.4 Retrieving the collection

To leave the Mongo Shell, either enter **CTRL-C** or issue the command **exit**. The **mongod** server can also be stopped by **CTRL-C**.

Do it now!	Chain the find() and pretty() methods to generate output similar to that shown in Figure C1.4 above. Note that some versions of the mongosh shell output in pretty format by default.
-------------------	--

C1.3 Basic MongoDB Commands

Now that we can create databases and collections and provide documents for them, we can begin to manipulate the data with some of the basic MongoDB commands.

C1.3.1 Basic searching

We have already seen how the **find()** method can be used to retrieve the documents within a collection. Where we want to retrieve a subset of the documents, we pass a JSON object to **find()** which contains the fields and values that we want to match.

For example, to return the document where the name field matches the value “Flask”, we can use the command

```
> db.fullStack.find ( { name : "Flask" } )
```

If we want to match multiple values, we simply add them to the object passed as a parameter to **find()**. For example, to retrieve documents where the **name** field is “MongoDB” and the **role** field is “Database”, we would issue the command

```
> db.fullStack.find ( { name:"MongoDB", role:"Database" } )
```

By default, the **find()** method returns whole documents – i.e. the object returned contains every field in matching documents. Where we only want specific values to be retrieved, we specify those fields in a second parameter (known in MongoDB terminology as a **projection**) to **find()** as shown in the following command which requests that only the **role** values should be returned.

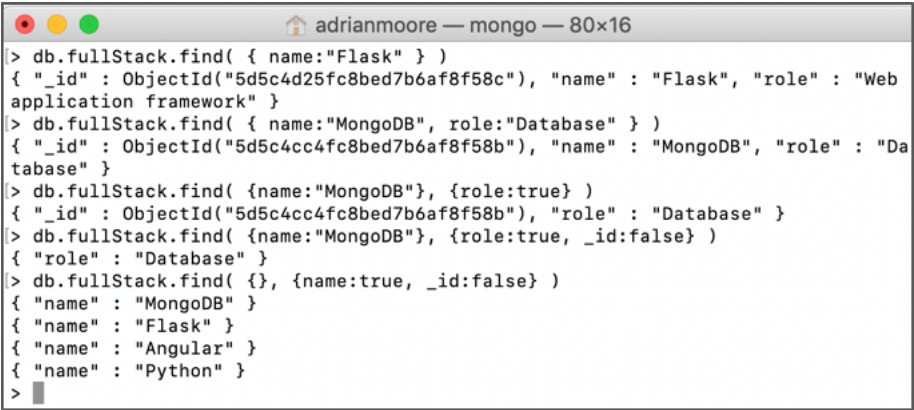
```
> db.fullStack.find ( {name:"MongoDB"}, {role:true} )
```

Note that by default, the **_id** field is ALWAYS returned – even when we do not specify it. If we want the **_id** to be excluded, we need to explicitly state that in the command as shown in the following

```
> db.fullStack.find( {name:"MongoDB"}, {role:true, _id:false} )
```

If we want to return specific fields from all documents, we can simply leave the first parameter of **find()** as an empty object.

```
> db.fullStack.find ( {}, {name:true, _id:false} )
```



```
adrianmoore — mongo — 80x16
> db.fullStack.find( { name:"Flask" } )
{ "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58c"), "name" : "Flask", "role" : "Web application framework" }
> db.fullStack.find( { name:"MongoDB", role:"Database" } )
{ "_id" : ObjectId("5d5c4cc4fc8bed7b6af8f58b"), "name" : "MongoDB", "role" : "Database" }
> db.fullStack.find( {name:"MongoDB"}, {role:true} )
{ "_id" : ObjectId("5d5c4cc4fc8bed7b6af8f58b"), "role" : "Database" }
> db.fullStack.find( {name:"MongoDB"}, {role:true, _id:false} )
{ "role" : "Database" }
> db.fullStack.find( {}, {name:true, _id:false} )
{ "name" : "MongoDB" }
{ "name" : "Flask" }
{ "name" : "Angular" }
{ "name" : "Python" }
>
```

Figure C1.5 Searching the collection

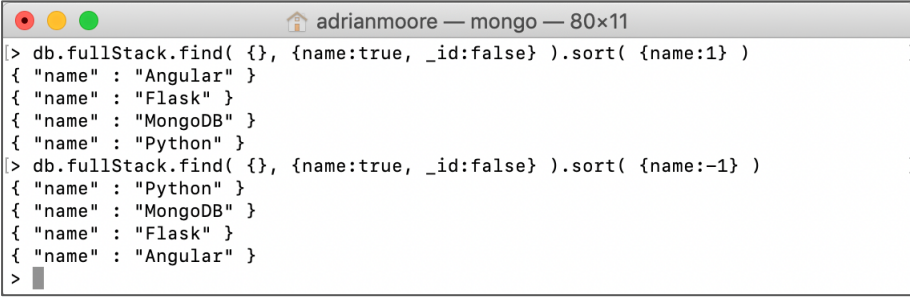
C1.3.2 Sorting query results

We can sort the results of MongoDB queries by chaining the **sort()** method to the results of a **find()** operation. We specify the field to sort by and the direction of sort by providing the information as a Javascript object, passed as a parameter to **sort()**. For example, to return the collection of name values sorted in ascending order, we would use the command

```
> db.fullStack.find ( { }, { "name" : true, "_id" : false } )  
      .sort( { "name" : 1 } )
```

To perform the sort in descending order, we change the sort key value from 1 to -1.

```
> db.fullStack.find ( { }, { "name" : true, "_id" : false } )  
      .sort( { "name" : -1 } )
```

A screenshot of a terminal window titled 'adrianmoore — mongo — 80x11'. The terminal shows two MongoDB commands and their results. The first command is `db.fullStack.find({}, {name:true, _id:false}).sort({name:1})`, which returns five documents with 'name' values: 'Angular', 'Flask', 'MongoDB', 'Python', and 'Angular' in ascending order. The second command is `db.fullStack.find({}, {name:true, _id:false}).sort({name:-1})`, which returns the same five documents but in descending order: 'Python', 'MongoDB', 'Flask', 'Angular', and 'Angular'.

```
|> db.fullStack.find( {}, {name:true, _id:false} ).sort( {name:1} )  
{ "name" : "Angular" }  
{ "name" : "Flask" }  
{ "name" : "MongoDB" }  
{ "name" : "Python" }  
{ "name" : "Angular" }  
|> db.fullStack.find( {}, {name:true, _id:false} ).sort( {name:-1} )  
{ "name" : "Python" }  
{ "name" : "MongoDB" }  
{ "name" : "Flask" }  
{ "name" : "Angular" }  
>
```

Figure C1.6 Sorting the collection

Do it now!	Try the searching and sorting commands shown in Sections C1.4.1 and C1.4.2 above and ensure that you receive output such as that shown in Figures C1.5 and C1.6
-------------------	---

Try it now!	Repeat the searching and sorting exercise with your own collection created earlier. Try different combinations of search fields, projections and sort options to ensure that you understand how the queries are constructed.
--------------------	--

C1.3.3 Updating documents

Updating a document is a two-stage command that consists of a **find** to locate the document(s) to be modified and a **set** to write the new values. These are passed as two parameters to the **update()** method as illustrated in the following command. Note the use of the special MongoDB command **\$set** in the second parameter.

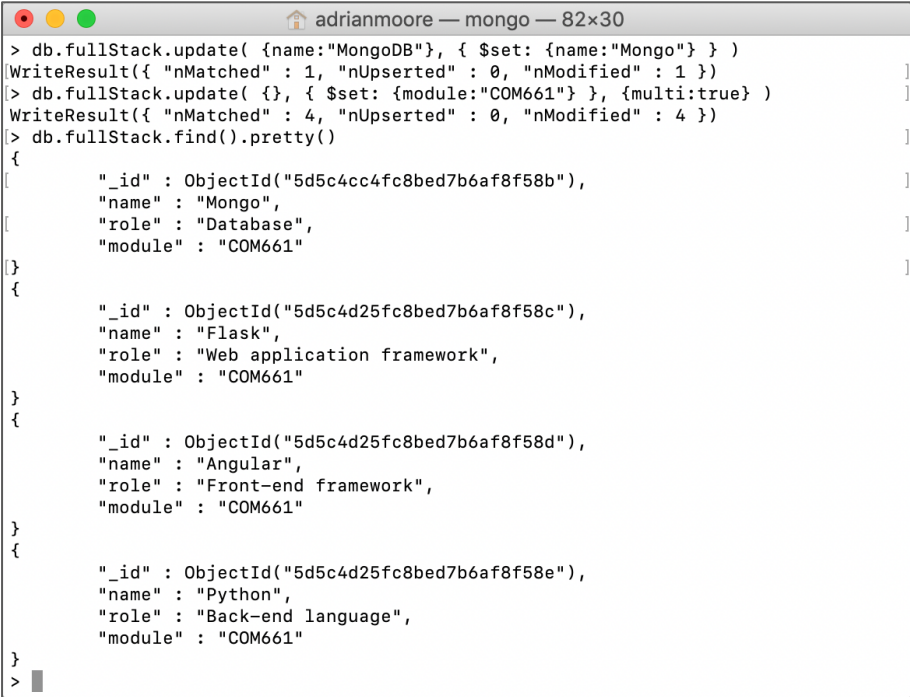
```
> db.fullStack.update ( { name:"MongoDB" },
                        { $set : { name:"Mongo" } }
                      )
```

This command locates all documents with a **name** value of “MongoDB” and updates those **name** fields to the new value “Mongo”. By default, the **update()** method will only update a single document – i.e. the first document that matches the search term. If we want to update multiple documents, we need to pass an extra options parameter to the **update()** method. We can illustrate this in our database by the following command that adds a new field to every document with a key of “module” and a value of “COM661”

```
> db.fullStack.update ( { } ,
                        { $set : { module:"COM661" } },
                        { multi : true }
                      )
```

Do it now!	Try the update commands shown above and ensure that you receive output such as that shown in Figures C1.7 below. Add a find() command to verify that the updates have been completed successfully
-------------------	---

Try it now!	Repeat the update exercise with your own collection to ensure that you understand how it works.
--------------------	--



```
adrianmoore — mongo — 82x30
> db.fullStack.update( {name:"MongoDB"}, { $set: {name:"Mongo"} } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
[> db.fullStack.update( {}, { $set: {module:"COM661"} }, {multi:true} )
WriteResult({ "nMatched" : 4, "nUpserted" : 0, "nModified" : 4 })
[> db.fullStack.find().pretty()
{
  {
    "_id" : ObjectId("5d5c4cc4fc8bed7b6af8f58b"),
    "name" : "Mongo",
    "role" : "Database",
    "module" : "COM661"
  }
  {
    "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58c"),
    "name" : "Flask",
    "role" : "Web application framework",
    "module" : "COM661"
  }
  {
    "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58d"),
    "name" : "Angular",
    "role" : "Front-end framework",
    "module" : "COM661"
  }
  {
    "_id" : ObjectId("5d5c4d25fc8bed7b6af8f58e"),
    "name" : "Python",
    "role" : "Back-end language",
    "module" : "COM661"
  }
}
```

Figure C1.7 Updating a collection

C1.3.4 Deleting documents and collections

In order to delete documents from a collection, MongoDB provides a **remove()** method. This method takes a JSON object as a parameter that specifies the documents to be deleted. For example, we could remove the “Angular” document from our example database by the command

```
> db.fullStack.remove ( { "name" : "Angular" } )
```

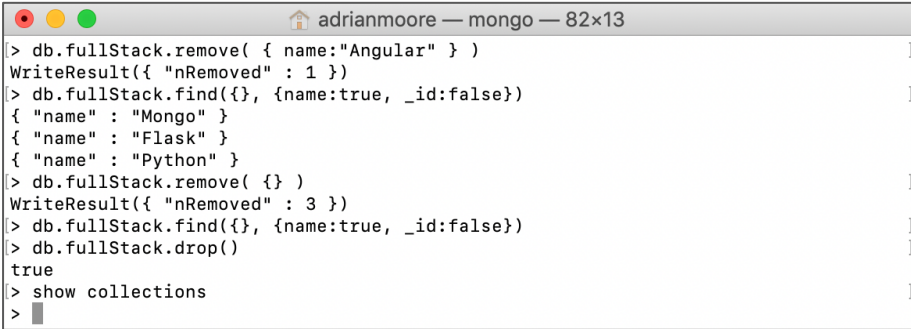
Note that the format of the parameter is exactly that already seen in the **find()** method – hence an empty object will match ALL documents and remove everything from the collection.

```
> db.fullStack.remove ( { } )
```

Deleting an entire collection is performed by the **drop()** method as illustrated by the command

```
> db.fullStack.drop ( )
```

Figure C1.8 demonstrates the **remove()** and **drop()** methods



```
adrianmoore — mongo — 82x13
> db.fullStack.remove( { name:"Angular" } )
WriteResult({ "nRemoved" : 1 })
> db.fullStack.find({}, {name:true, _id:false})
{ "name" : "Mongo" }
{ "name" : "Flask" }
{ "name" : "Python" }
> db.fullStack.remove( {} )
WriteResult({ "nRemoved" : 3 })
> db.fullStack.find({}, {name:true, _id:false})
> db.fullStack.drop()
true
> show collections
>
```

Figure C1.8 Remove and drop methods

Do it now!	Try the remove and drop commands shown above and ensure that you receive output such as that shown in Figure C1.8 above.
-------------------	--

Try it now!	Repeat the remove and drop exercise with your own collection to ensure that you understand how the commands are formed
--------------------	--

C1.4 Exporting and Importing Data

A common requirement in database-driven applications is the need to be able to move data in and out of the database. This might be to create backups or to populate an application with data before launch.

Do it now!	In preparation for this next section, re-create the fullStack collection and add the original four documents.
-------------------	--

C1.4.1 Exporting and Importing BSON data

Mongo provides options for export and import as both BSON (Binary JSON) and standard JSON. The option we choose to use depends on our requirement – if we are exporting the data for archive purposes, we will most likely use the binary format, while if we will want to read the data ourselves, then the JSON option is the most appropriate choice.

The Mongo tool for exporting database contents is **mongodump** which is run from the command prompt (not the MongoDB shell) as

```
P:\> mongodump --db first
```

where the name of the database is passed as the value of the **--db** flag.

The **mongodump** tool saves the exported data in a folder called **dump** within the current working directory. If you explore the **dump** folder you should find another sub-folder called **first** (the name of our database) and inside this folder you should see the BSON files that have been created.

The opposite action to **mongodump** is **mongorestore**, which allows us to create a database by importing a previously dumped data set. The **mongorestore** tool is run as

```
P:\> mongorestore --db second dump\first
```

which will create a new database called **second** by importing the data from our previous **mongodump** action. Figure C1.9 illustrates the dump and restore process used to make a copy of our database **first** in a new database called **second**.

```
Adrians-MBP:Desktop adrianmoore$ mongodump --db first
2019-08-21T10:17:08.323+0100    writing first.fullStack to
2019-08-21T10:17:08.324+0100    done dumping first.fullStack (4 documents)
Adrians-MBP:Desktop adrianmoore$ mongorestore --db second dump/first
2019-08-21T10:18:16.316+0100    the --db and --collection args should only be used when rest
oring from a BSON file. Other uses are deprecated and will not exist in the future; use --ns
Include instead
2019-08-21T10:18:16.316+0100    building a list of collections to restore from dump/first di
r
2019-08-21T10:18:16.317+0100    reading metadata for second.fullStack from dump/first/fullSt
ack.metadata.json
2019-08-21T10:18:16.351+0100    restoring second.fullStack from dump/first/fullStack.bson
2019-08-21T10:18:16.353+0100    no indexes to restore
2019-08-21T10:18:16.353+0100    finished restoring second.fullStack (4 documents)
2019-08-21T10:18:16.353+0100    done
Adrians-MBP:Desktop adrianmoore$
```

Figure C1.9 Dump and restore operations

Having restored the database, we can show that the operation has worked by entering the MongoDB shell, specify that as want to use the new database and display the data. Figure C1.10 illustrates this process.

```
Desktop — mongo — 80x12
> use second
switched to db second
> db.fullStack.find()
{ "_id" : ObjectId("5d5d0b31fc8bed7b6af8f58f"), "name" : "MongoDB", "role" : "Da
tabase" }
{ "_id" : ObjectId("5d5d0b48fc8bed7b6af8f590"), "name" : "Flask", "role" : "Web
Application Framework" }
{ "_id" : ObjectId("5d5d0b5dfc8bed7b6af8f591"), "name" : "Angular", "role" : "Fr
ont-end Framework" }
{ "_id" : ObjectId("5d5d0b72fc8bed7b6af8f592"), "name" : "Python", "role" : "Bac
k-end Language" }
>
```

Figure C1.10 Using the copied database

Note that **mongorestore** only performs **insert** operations – it does not do updates. If we are restoring to a database that already has content, only documents with **_id** values that are NOT already present will be added. It will not replace current values with new data from the file to be restored.

Do it now!	Try the dump and restore sequence shown above to create a copy of your database called first as a new database called second . Display the contents of the fullStack collection in second to verify that it has worked as expected.
-------------------	---

C1.4.2 Exporting and Importing JSON data

When we export JSON from a MongoDB database, we create a JSON representation of the data. We can see this by using the **mongoexport** tool and examining the JSON that is returned. The **mongoexport** tool is initiated as

```
P:\> mongoexport --db first --collection fullStack
```

Note that we need to specify the collection to be exported as well as the database and that, as illustrated in Figure C1.10, the default behavior is to output the JSON data to the console.

```
Adrians-MBP:Desktop adrianmoore$ mongoexport --db first --collection fullStack
2019-08-21T10:30:22.870+0100    connected to: localhost
{"_id":{"$oid":"5d5d0b31fc8bed7b6af8f58f"},"name":"MongoDB","role":"Database"}
{"_id":{"$oid":"5d5d0b48fc8bed7b6af8f590"},"name":"Flask","role":"Web Application Framework"}
{"_id":{"$oid":"5d5d0b5dfc8bed7b6af8f591"},"name":"Angular","role":"Front-end Framework"}
{"_id":{"$oid":"5d5d0b72fc8bed7b6af8f592"},"name":"Python","role":"Back-end Language"}
2019-08-21T10:30:22.871+0100    exported 4 records
Adrians-MBP:Desktop adrianmoore$
```

Figure C1.11 Using mongoexport

In order to save the data to a file we need to specify an additional flag to the **mongoexport** command as follows

```
P:\> mongoexport --db first --collection fullStack
--out data.json
```

If we examine the data file generated, we can see that it is still not quite usable as it does not represent a valid JSON structure as each document in the collection is represented as a separate JSON object, but it has failed to present the objects as an array.

Fortunately, there are another pair of flags we can specify, that will structure the data as an array (**--jsonArray**) and format it in a more human-readable form (**--pretty**)

```
P:\> mongoexport --db first --collection fullStack
--out data.json --jsonArray --pretty
```

Now, if we examine the JSON file generated, we see that it has a more useful appearance

```
[{
  "_id": {
    "$oid": "5898e88e6f3eb4a7bab4682"
  },
  "name": "MongoDB",
  "role": "Database"
},
...
]
```

Now, we can see the JSON representation of the data, and in particular how it treats the `_id` value. When we retrieved documents earlier using the `find()` method, we saw how the `_id` was represented as an `ObjectId()` value. However, as this is not valid JSON, `mongoexport` creates a valid representation by defining `_id` as a JSON object, with the unique identifier as the value of an `$oid` property.

The opposite command to `mongoexport` is `mongoimport`, which allows us to create a database and import a collection from a JSON file. The format of the `mongoimport` command is

```
P:\> mongoimport --db third --collection fullStack
--jsonArray data.json
```

where we specify the database to be used or created (**third**), the collection to be populated with the database (**fullStack**), that the file is a JSON array (**--jsonArray**) and finally the JSON file containing the data (**data.json**).

Figure C1.12 illustrates the `mongoimport` process and demonstrates that the database called **third** has been created and populated with the data.

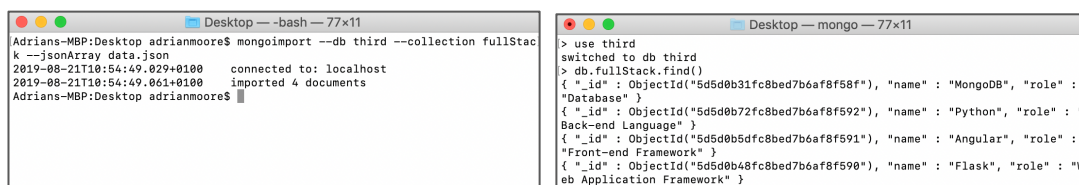


Figure C1.12 Using `mongoimport`

Do it now!	Try the export and import sequence shown above to export your fullStack collection from first into a new database called third . Display the contents of the fullStack collection in third to verify that it has worked as expected.
-------------------	---

C1.5 Further Information

- <https://www.youtube.com/watch?v=FwMwO8pXfq0>
How to install MongoDB on Windows 10 - YouTube
- <https://zellwk.com/blog/install-mongodb/>
Installing MongoDB on a Mac (Catalina and non-Catalina)
- <https://www.youtube.com/watch?v=2yQ9TGFpDuM>
NoSQL Databases for Beginners - YouTube
- https://www.youtube.com/watch?v=ZS_kXvOeQ5Y
SQL vs NoSQL - YouTube
- <https://www.w3resource.com/mongodb/nosql.php>
Introduction to NoSQL - HotJar
- <https://www.mongodb.com/>
MongoDB home page
- <https://www.tutorialspoint.com/mongodb/index.htm>
MongoDB Tutorial from TutorialsPoint
- <https://www.guru99.com/mongodb-tutorials.html>
MongoDB Tutorial for Beginners
- <https://docs.mongodb.com/manual/tutorial/query-documents/>
Searching MongoDB Documents
- <https://beginnersbook.com/2017/09/mongodb-sort-method/>
Sorting MongoDB documents
- <https://www.youtube.com/watch?v=KmjzcRRHD5s>
Importing and Exporting Data with MongoDB - YouTube