COM644 Full-Stack Web and App Development

Practical B2: MongoDB Commands

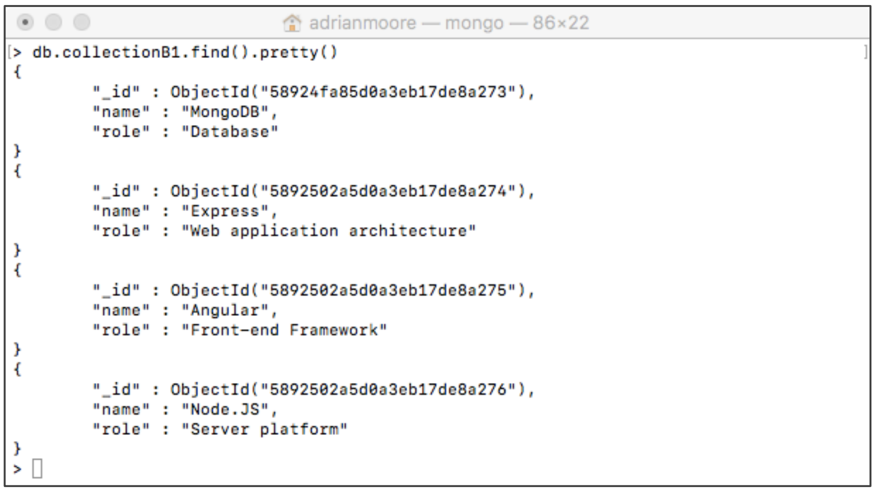
Aims

- To introduce the find() method to search a MongoDB collections
- To illustrate sorting of data returned by a find()
- To demonstrate the update() method for modifying documents
- To demonstrate the remove() method for removing documents from collections
- To demonstrate the drop() method for deleting collections
- To compare BSON and JSON as alternative notations for porting of data between databases
- To introduce export and import of BSON data
- To introduce export and import of JSON data

Contents

B2.1 Basic MongoDB Commands

In the previous practical, we created a simple document store database in MongoDB and populated it with a small collection of documents as illustrated in figure B2.1 below.



```
> db.collectionB1.find().pretty()
{
        "_id" : ObjectId("58924fa85d0a3eb17de8a273"),
        "name" : "MongoDB",
        "role" : "Database"
}
{
        "_id" : ObjectId("5892502a5d0a3eb17de8a274"),
        "name" : "Express",
        "role" : "Web application architecture"
}
{
        "_id" : ObjectId("5892502a5d0a3eb17de8a275"),
        "name" : "Angular",
        "role" : "Front-end Framework"
}
{
        "_id" : ObjectId("5892502a5d0a3eb17de8a276"),
        "name" : "Node.JS",
        "role" : "Server platform"
}
> 
```

Figure B2.1 Sample database in MongoDB

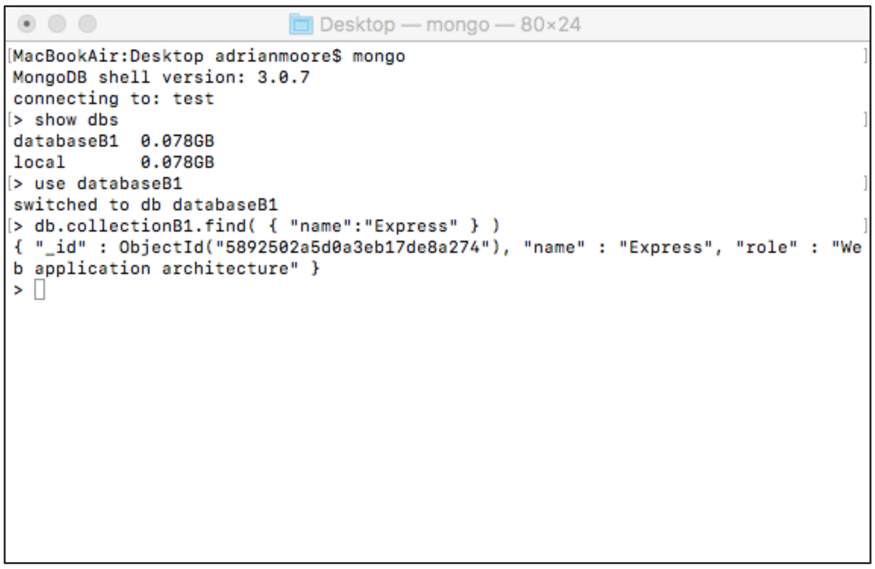
In this session, we will continue to work with this database in order to illustrate the main retrieval and update operations.

B2.1.1 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 Javascript 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 "Express", we can use the command

```
> db.collectionB1.find ( { "name" : "Express" } )
```



```
MacBookAir:Desktop adrianmoore$ mongo
MongoDB shell version: 3.0.7
connecting to: test
> show dbs
databaseB1  0.078GB
local       0.078GB
> use databaseB1
switched to db databaseB1
> db.collectionB1.find( { "name":"Express" } )
{ "_id" : ObjectId("5892502a5d0a3eb17de8a274"), "name" : "Express", "role" : "We
b application architecture" }
> 
```

Figure B2.2 The basic find() method

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.collectionB1.find ({ "name" : "Mongo", "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.collectionB1.find ({ "name" : "Express" }, { "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.collectionB1.find ({"name" : "Express"}, { "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.collectionB1.find ({ }, { "name" : true, "_id" : false })

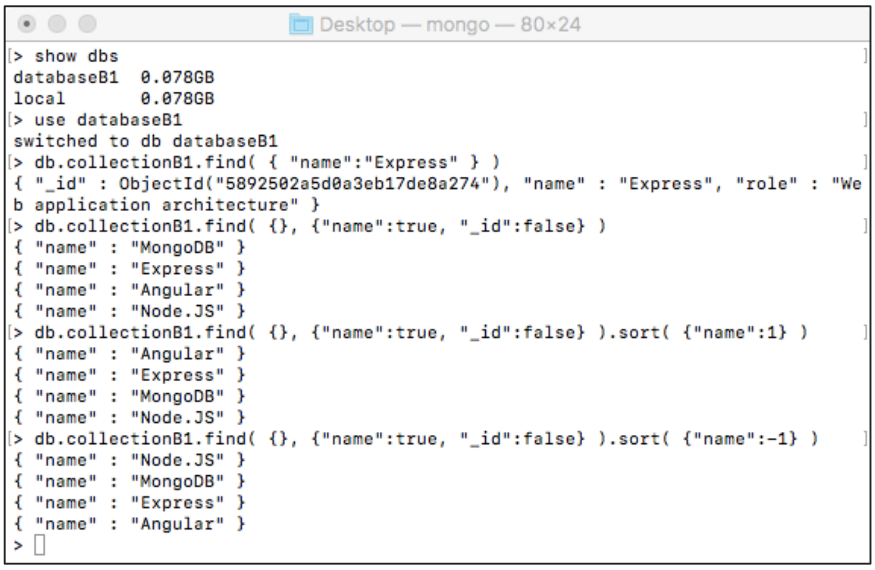
B2.1.2 Sorting query results

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.collectionB1.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.collectionB1.find ({ }, { "name" : true, "_id" : false }   ).sort({ "name" : -1 })**



```
> show dbs
databaseB1  0.078GB
local       0.078GB
> use databaseB1
switched to db databaseB1
> db.collectionB1.find( { "name":"Express" } )
{ "_id" : ObjectId("5892502a5d0a3eb17de8a274"), "name" : "Express", "role" : "We
b application architecture" }
> db.collectionB1.find( {}, {"name":true, "_id":false} )
{ "name" : "MongoDB" }
{ "name" : "Express" }
{ "name" : "Angular" }
{ "name" : "Node.JS" }
> db.collectionB1.find( {}, {"name":true, "_id":false} ).sort( {"name":1} )
{ "name" : "Angular" }
{ "name" : "Express" }
{ "name" : "MongoDB" }
{ "name" : "Node.JS" }
> db.collectionB1.find( {}, {"name":true, "_id":false} ).sort( {"name":-1} )
{ "name" : "Node.JS" }
{ "name" : "MongoDB" }
{ "name" : "Express" }
{ "name" : "Angular" }
> 
```

Figure B2.3 Searching and sorting

B2.1.3 Updating documents

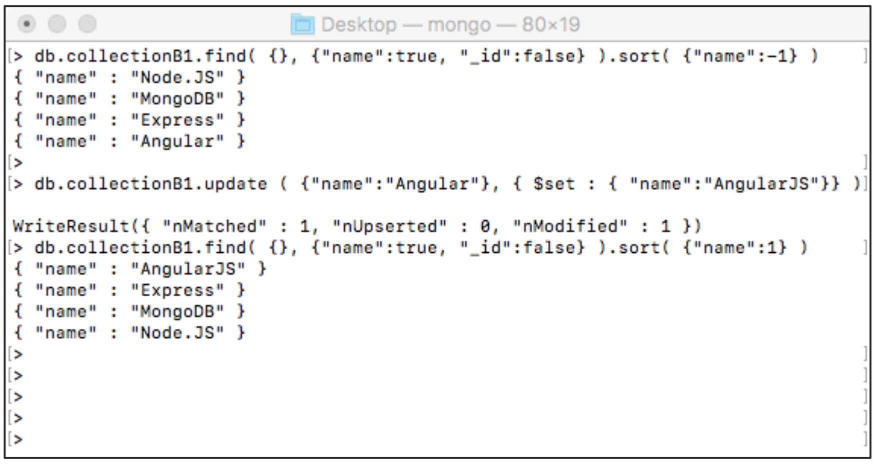
Updating a document is a two-stage command that consists of a **find** to locate the document(s) to be modified and a **set** to write the new values. These are passed as two parameters to the **`update()`** method as illustrated in the following command. Note the use of the special MongoDB command **`$set`**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

```
> db.collectionB1.update ( { "name" : "Angular" } ,
                           { $set : { "name" : "AngularJS"}  }
                         )
```

This command locates all documents with a **name** value of “Angular” and updates those **name** fields to the new value “AngularJS”.

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 “language” and a value of “Javascript”

```
> db.collectionB1.update ( { } ,
                           { $set : { “language” : “Javascript”} },
                           { multi : true }
                         )
```



```
> db.collectionB1.find( {}, {"name":true, "_id":false} ).sort( {"name":-1} )
{ "name" : "Node.JS" }
{ "name" : "MongoDB" }
{ "name" : "Express" }
{ "name" : "Angular" }
>
> db.collectionB1.update ( {"name":"Angular"}, { $set : { "name":"AngularJS"}} )

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.collectionB1.find( {}, {"name":true, "_id":false} ).sort( {"name":1} )
{ "name" : "AngularJS" }
{ "name" : "Express" }
{ "name" : "MongoDB" }
{ "name" : "Node.JS" }
>
>
>
>
>
```

Figure B2.4 Updating a collection

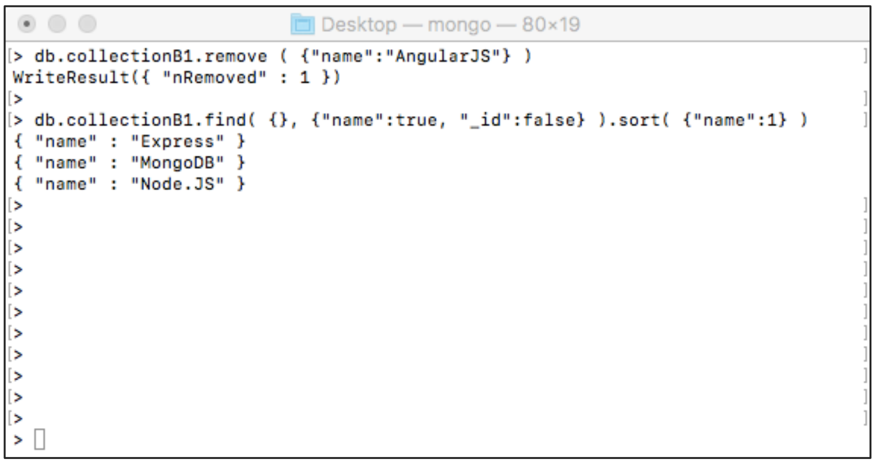
B2.1.4 Deleting documents and collections

In order to delete documents from a collection, MongoDB provides a **`remove()`** method. This method takes a Javascript 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.collectionB1.remove ( { “name” : “AngularJS” } )
```

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.collectionB1.remove ( { } )
```

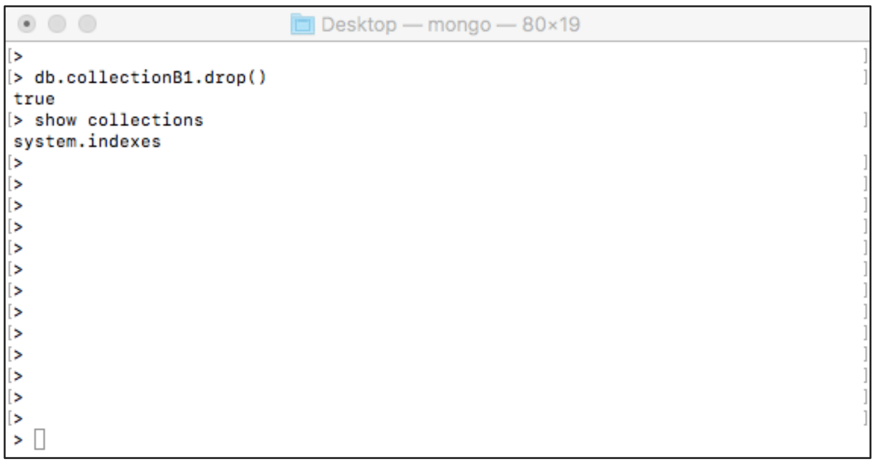


```
> db.collectionB1.remove ( {"name":"AngularJS"} )
WriteResult({ "nRemoved" : 1 })
>
> db.collectionB1.find( {}, {"name":true, "_id":false} ).sort( {"name":1} )
{ "name" : "Express" }
{ "name" : "MongoDB" }
{ "name" : "Node.JS" }
>
>
>
>
>
>
>
>
>
>
>
> 
```

Figure B2.5 Removing a document from a collection

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

> **db.collectionB1.drop ( )**



```
>
> db.collectionB1.drop()
true
> show collections
system.indexes
>
>
>
>
>
>
>
>
>
>
>
>
>
> 
```

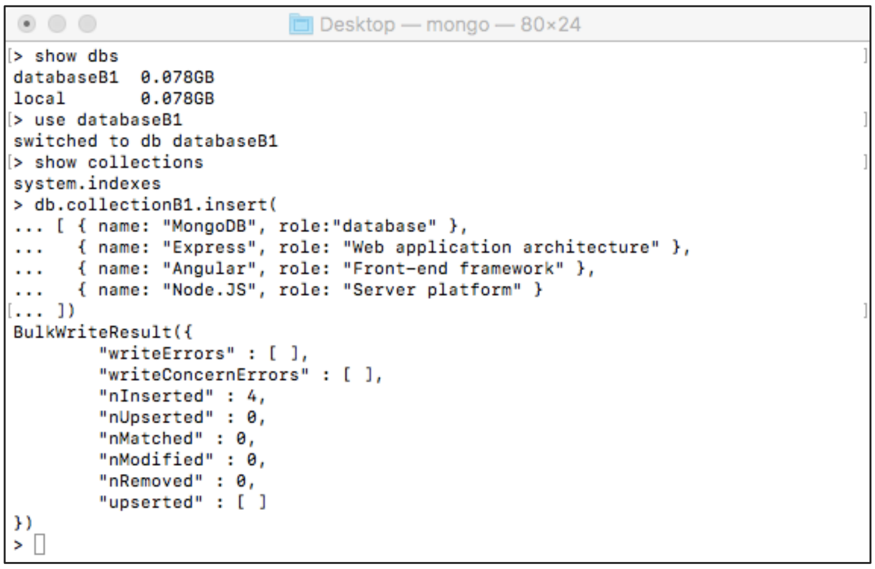
Figure B2.6 Dropping a collection from a database

B2.2 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.

Before exploring the options for data export and import, we will need to re-create the collection that we dropped in the previous example.

Copy the statement from the file *addCollection.txt* provided in the Practical Files and paste it to the Mongo command prompt as shown in Figure B2.7 below.



```
> show dbs
databaseB1  0.078GB
local       0.078GB
> use databaseB1
switched to db databaseB1
> show collections
system.indexes
> db.collectionB1.insert(
... [ { name: "MongoDB", role:"database" },
...   { name: "Express", role: "Web application architecture" },
...   { name: "Angular", role: "Front-end framework" },
...   { name: "Node.JS", role: "Server platform" }
... ])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 4,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> 
```

Figure B2.7 Re-create the collection

B2.2.1 Exporting and Importing BSON data

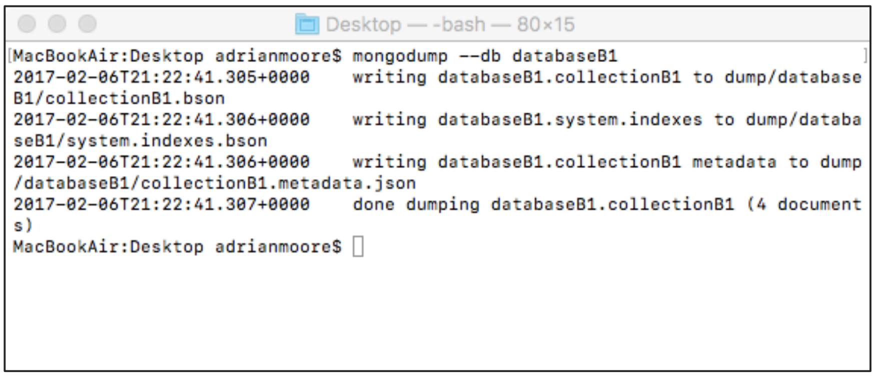
Mongo provides options for export and import as both BSON (Binary JSON) and standard JSON. The option we choose to use depends on our requirement – if we are exporting the data for archive purposes, we will most likely use the binary format, while if we will want to read the data ourselves, then the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

U:\B2> **mongodump --db databaseB1**

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 **databaseB1** (the name of our database) and inside this folder you should see the BSON files that have been created.



```
MacBookAir:Desktop adrianmoore$ mongodump --db databaseB1
2017-02-06T21:22:41.305+0000    writing databaseB1.collectionB1 to dump/database
B1/collectionB1.bson
2017-02-06T21:22:41.306+0000    writing databaseB1.system.indexes to dump/databa
seB1/system.indexes.bson
2017-02-06T21:22:41.306+0000    writing databaseB1.collectionB1 metadata to dump
/databaseB1/collectionB1.metadata.json
2017-02-06T21:22:41.307+0000    done dumping databaseB1.collectionB1 (4 document
s)
MacBookAir:Desktop adrianmoore$ 
```

Figure B2.8 Using mongodump

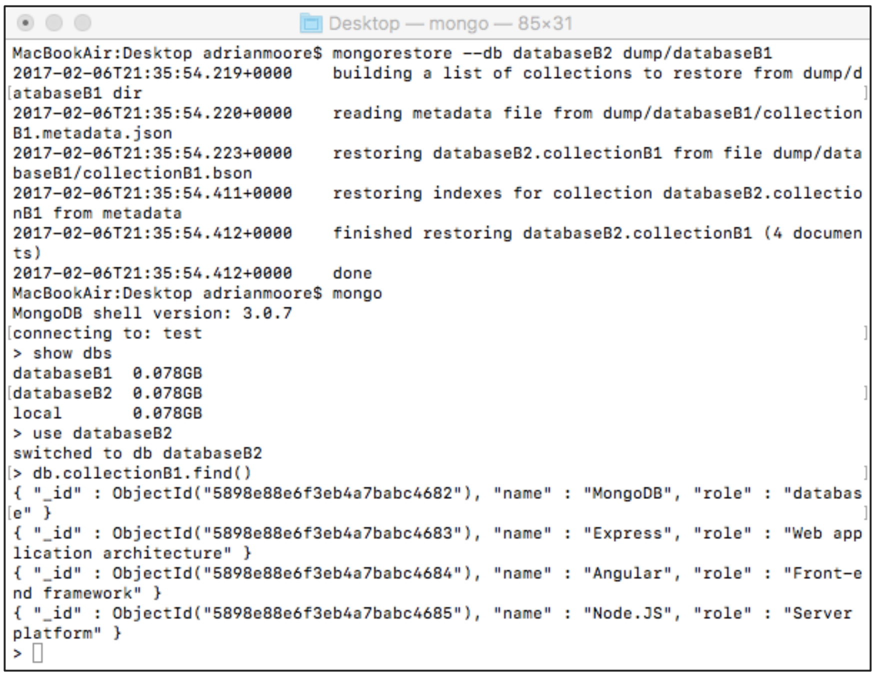
The opposite action to **mongodump** is **mongorestore**, which allows us to create a database by importing a previously dumped data set. The **mongorestore** tool is run as

U:\B2> **mongorestore --db databaseB2 dump\databaseB1**

which will create a new database called **databaseB2** by importing the data from our previous **mongodump** action.

Having restored the database, we can show that the operation has worked by entering the mongo shell, specify that as want to use the new database and display the data. Figure B2.9 illustrates this process.

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.



```
MacBookAir:Desktop adrianmoore$ mongorestore --db databaseB2 dump/databaseB1
2017-02-06T21:35:54.219+0000    building a list of collections to restore from dump/d
atabaseB1 dir
2017-02-06T21:35:54.220+0000    reading metadata file from dump/databaseB1/collection
B1.metadata.json
2017-02-06T21:35:54.223+0000    restoring databaseB2.collectionB1 from file dump/data
baseB1/collectionB1.bson
2017-02-06T21:35:54.411+0000    restoring indexes for collection databaseB2.collectio
nB1 from metadata
2017-02-06T21:35:54.412+0000    finished restoring databaseB2.collectionB1 (4 documen
ts)
2017-02-06T21:35:54.412+0000    done
MacBookAir:Desktop adrianmoore$ mongo
MongoDB shell version: 3.0.7
connecting to: test
> show dbs
databaseB1  0.078GB
databaseB2  0.078GB
local       0.078GB
> use databaseB2
switched to db databaseB2
> db.collectionB1.find()
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4682"), "name" : "MongoDB", "role" : "databas
e" }
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4683"), "name" : "Express", "role" : "Web app
lication architecture" }
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4684"), "name" : "Angular", "role" : "Front-e
nd framework" }
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4685"), "name" : "Node.JS", "role" : "Server
platform" }
> 
```

Figure B2.9 Using mongorestore

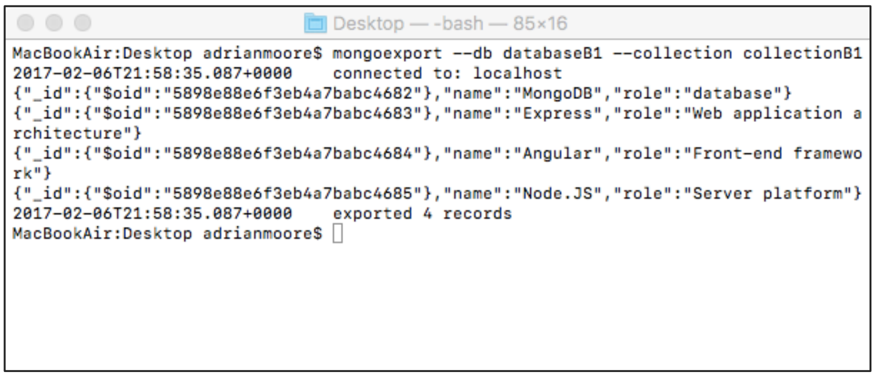
B2.2.2 Exporting and Importing JSON data

When we export JSON from a MongoDB database, we export 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

U:\B2> **mongoexport --db databaseB1 --collection collectionB1**

Note that we need to specify the collection to be exported as well as the database and that, as illustrated in Figure B2.10, the default behavior is to output the JSON data to the console.



```
MacBookAir:Desktop adrianmoore$ mongoexport --db databaseB1 --collection collectionB1
2017-02-06T21:58:35.087+0000    connected to: localhost
{"_id":{"$oid":"5898e88e6f3eb4a7babc4682"},"name":"MongoDB","role":"database"}
{"_id":{"$oid":"5898e88e6f3eb4a7babc4683"},"name":"Express","role":"Web application a
rchitecture"}
{"_id":{"$oid":"5898e88e6f3eb4a7babc4684"},"name":"Angular","role":"Front-end framewo
rk"}
{"_id":{"$oid":"5898e88e6f3eb4a7babc4685"},"name":"Node.JS","role":"Server platform"}
2017-02-06T21:58:35.087+0000    exported 4 records
MacBookAir:Desktop adrianmoore$ 
```

Figure B2.10 Using mongoexport

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

```
U:\B2> mongoexport --db databaseB1 --collection collectionB1 --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**)

```
U:\B2> mongoexport --db databaseB1 --collection collectionB1 --out data.json
       --jsonArray --pretty
```

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

```
[{
        "_id": {
            "$oid": "5898e88e6f3eb4a7babc4682"
        },
        "name": "MongoDB",
        "role": "database"
},
…
{
        "_id": {
            "$oid": "5898e88e6f3eb4a7babc4685"
        },
        "name": "Node.JS",
        "role": "Server platform"
}]
```

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 Javascript object, with the unique identifier as the value of an `$oid` property.

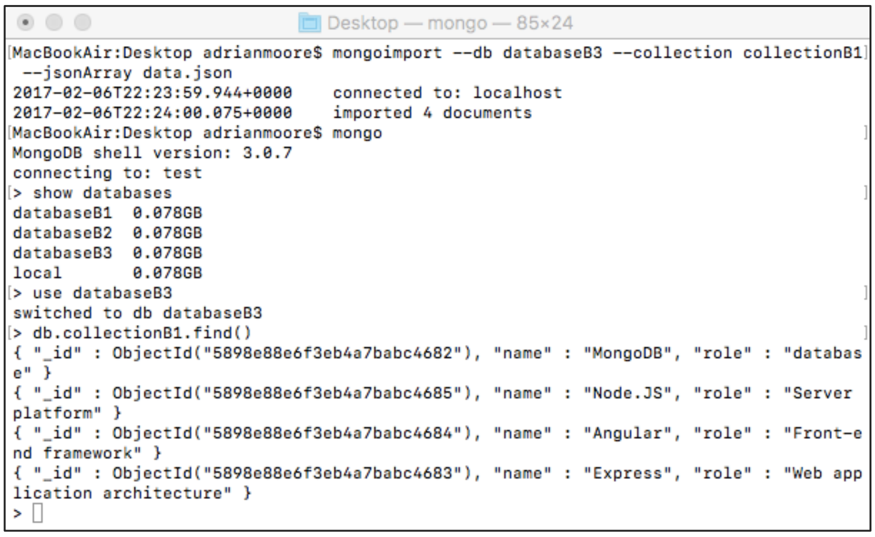
The opposite command to **mongoexport** is **mongoimport**, which allows us to create a database and import a collection from a JSON file that has previously been generated by **mongoexport**.

The format of the **mongoimport** command is

U:\B2> **mongoimport --db databaseB3 --collection collectionB1 --jsonArray data.json**

where we specify the database to be used or created (**databaseB3**), the collection to be populated with the database (**collectionB1**), that the file is a JSON array **(--jsonArray**) and finally the JSON file containing the data (**data.json**).

Figure B2.11 illustrates the **mongoimport** process and demonstrates that the database called databaseB3 has been created and populated with the data.



```
MacBookAir:Desktop adrianmoore$ mongoimport --db databaseB3 --collection collectionB1
 --jsonArray data.json
2017-02-06T22:23:59.944+0000    connected to: localhost
2017-02-06T22:24:00.075+0000    imported 4 documents
MacBookAir:Desktop adrianmoore$ mongo
MongoDB shell version: 3.0.7
connecting to: test
> show databases
databaseB1  0.078GB
databaseB2  0.078GB
databaseB3  0.078GB
local       0.078GB
> use databaseB3
switched to db databaseB3
> db.collectionB1.find()
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4682"), "name" : "MongoDB", "role" : "databas
e" }
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4685"), "name" : "Node.JS", "role" : "Server
platform" }
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4684"), "name" : "Angular", "role" : "Front-e
nd framework" }
{ "_id" : ObjectId("5898e88e6f3eb4a7babc4683"), "name" : "Express", "role" : "Web app
lication architecture" }
> 
```

Figure B2.11 Using mongoimport