COM644 Full-Stack Web and App Development

Practical B1: NoSQL and MongoDB

Aims

- To compare the relational and document store models for database architecture
- To provide guidelines for the design of document store databases
- To demonstrate manipulation of JSON data in Javascript
- To introduce the MongoDB database engine
- To explain the purpose and structure of MongoDB **`_id`** values
- To introduce the MongoDB server
- To demonstrate the MongoDB Shell
- To practice manipulating databases, collections and documents in the MongoDB Shell

Contents

B1.1 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

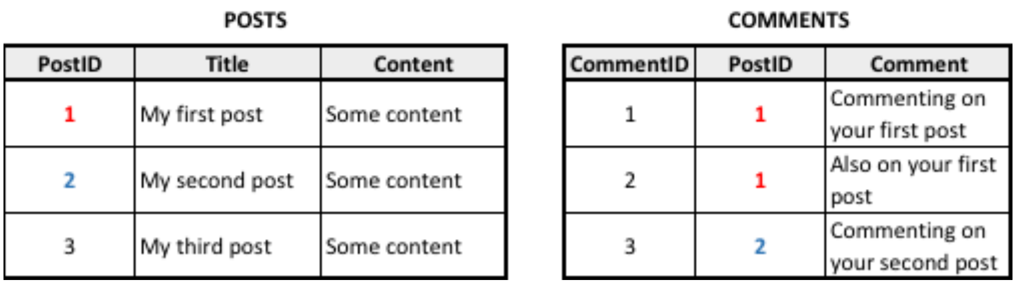
Key-value stores
Document Stores
Graph databases

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.

B1.1.1 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 B1.1 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

PostID	Title	Content
1	My first post	Some content
2	My second post	Some content
3	My third post	Some content

COMMENTS

CommentID	PostID	Comment
1	1	Commenting on your first post
2	1	Also on your first post
3	2	Commenting on your second post

Figure B1.1 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 Javascript 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 is it is 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).

B1.1.2 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 retrieval 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., etc. – so an obvious approach would be to have a separate structure such as that shown below.

```
[
  {
    “UserID” : 1,
    “UserName” : “Adrian”,
    “DisplayName” : “Adrian Moore”,
    “Bio” : “Web person”
  },
  {
    “UserID” : 2,
    “UserName” : “Therese”,
    “DisplayName” : “Therese Charles”,
    “Bio” : “Project management guru”
  }
]
```

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 does 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” : “Therese Charles”
   }
 }
]
```

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

B1.1.3 Data manipulation

Before continuing to the MongoDB document store database, it is useful to examine our Posts structure in code and see how Javascript can retrieve and manipulate individual elements.

Create a new folder called **B1** and create a new MEAN app within this folder by the command

```
U:\B1> npm init
```

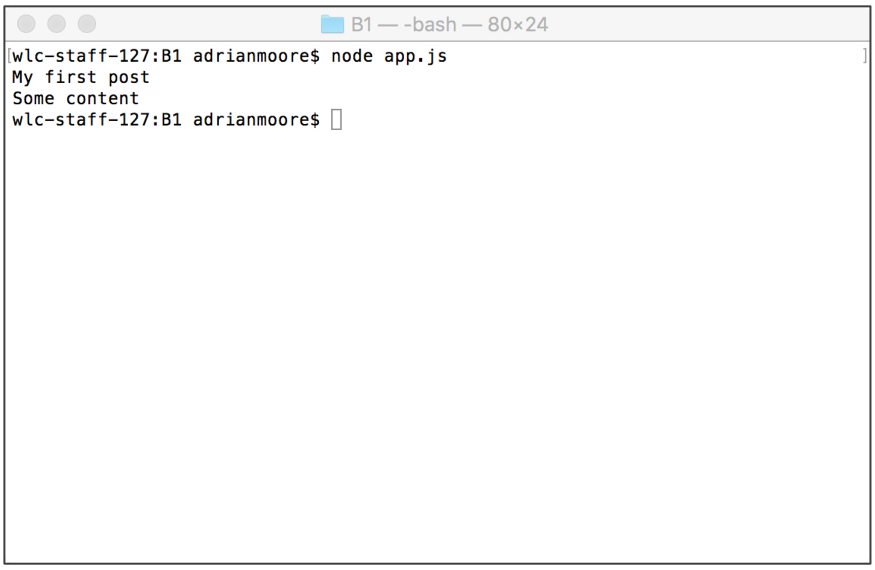
Now copy the file **posts.json** into the **B1** folder and create the file **app.js** with the following code.

File*: B1\app.js*

```
var posts = require('./posts.json');

console.log(posts[0].Title);
console.log(posts[0].Content);
```

Now run the application and verify that you retrieve and display the Title and Content values from the first Post element as shown in Figure B1.2 below.



```
wlc-staff-127:B1 adrianmoore$ node app.js
My first post
Some content
wlc-staff-127:B1 adrianmoore$ 
```

Figure B1.2 Retrieving from the JSON data structure

In order to loop through all posts and retrieve their **Title**, **Content** and author **Display Name** fields, repeat the previous exercise using the following code.

File: ***B1\app.js***

```
var posts = require('./posts.json');

for (var i=0; i < posts.length; i++ ) {
    console.log(posts[i].Title);
    console.log(posts[i].Content);
    console.log(posts[i].Author.DisplayName);
    console.log();
}
```

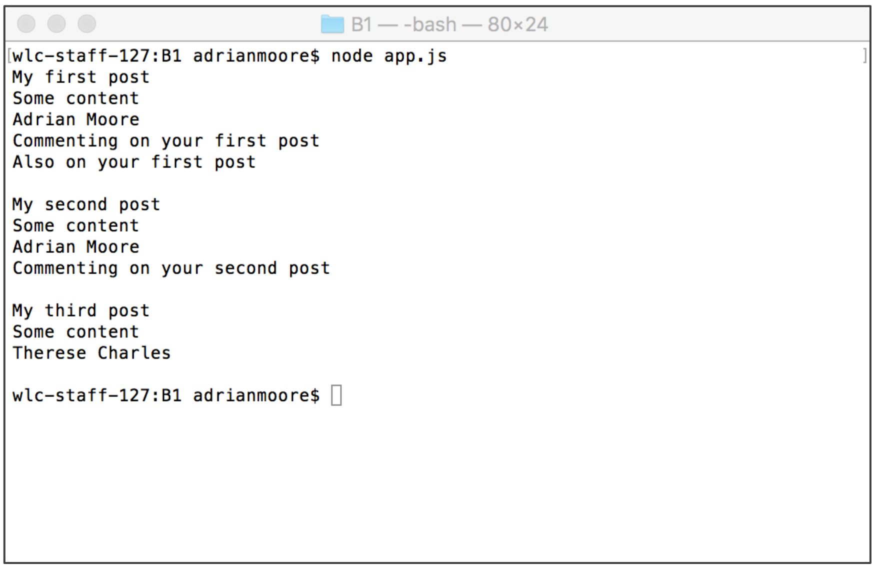
Finally, we will see how to deal with fields that may not be present in the data set. Modify **app.js** by including a test for the availability of a **Comments** field and, where it is present, loop across the array of comments, printing each **Content** value to the Console.

Check that you get output such as that in Figure B1.3 below.

File: *B1\app.js*

```
var posts = require('./posts.json');

for (var i=0; i < posts.length; i++ ) {
    console.log(posts[i].Title);
    console.log(posts[i].Content);
    console.log(posts[i].Author.DisplayName);
    if(posts[i].Comments) {
        for (var j=0; j<posts[i].Comments.length; j++) {
            console.log(posts[i].Comments[j].Content);
        }
    }
    console.log();
}
```



B1 — -bash — 80×24

```
wlc-staff-127:B1 adrianmoore$ node app.js
My first post
Some content
Adrian Moore
Commenting on your first post
Also on your first post

My second post
Some content
Adrian Moore
Commenting on your second post

My third post
Some content
Therese Charles

wlc-staff-127:B1 adrianmoore$ 
```

Figure B1.3 Dealing with optional JSON fields

Try it now!

Design your own document store database as a JSON structure (perhaps a dataset connected to your Final Year Project?) and…

i) Verify that the JSON is properly specified by requiring it into an **app.js** file

ii) Write Javascript code to iterate across your collection, writing selected values to the console.

B1.2 Introducing MongoDB

MongoDB (derived from Hu**mongo**us **D**ata**b**ase) is a document store database that organises information as **collections**. In our Posts example from the previous section, the file **posts.json** 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.

B1.2.1 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"
    }
  },
  ...
]
```

B1.3 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 http://www.mongodb.org.

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 has been set to **U:\data\db** so that your work will be available to you regardless of which lab machine you use.

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

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

B1.3.1 Launching a MongoDB server

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

```
U:\> mongod
```

which should result in output as illustrated in Figure B1.4 below.

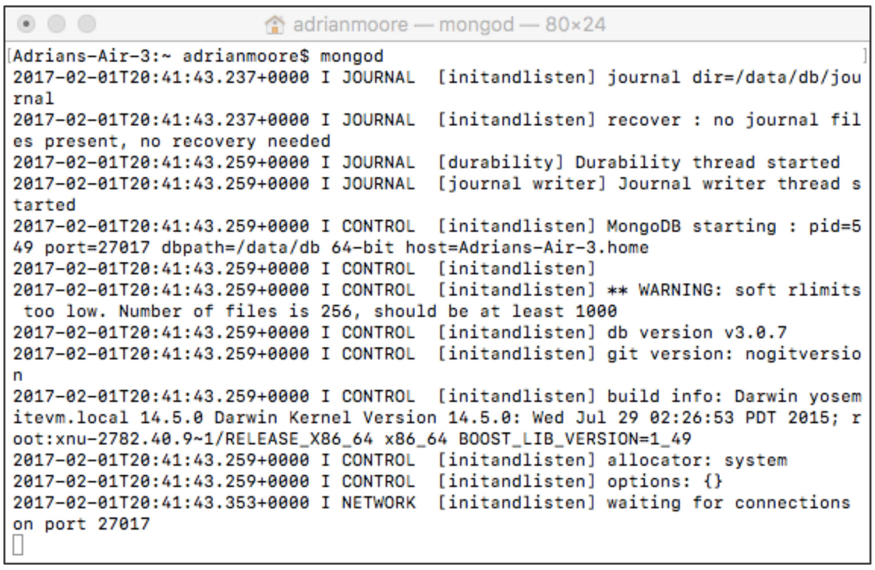

```
Adrians-Air-3:~ adrianmoore$ mongod
2017-02-01T20:41:43.237+0000 I JOURNAL  [initandlisten] journal dir=/data/db/jou
rnal
2017-02-01T20:41:43.237+0000 I JOURNAL  [initandlisten] recover : no journal fil
es present, no recovery needed
2017-02-01T20:41:43.259+0000 I JOURNAL  [durability] Durability 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 B1.4 MongoDB server

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

B1.3.2 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.

```
U:\> mongo
```

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, only one database, called **local**, is available so we can switch to this database by the command

```
> use local
```

To add a new database, we simply **use** one that is not already present. For example, to create a new database called **databaseB1**, we issue the command

```
> use databaseB1
```

and create a collection within this database by the command

```
> db.createCollection(“collectionB1”)
```

where the object **db** refers to the currently selected database (the subject of the most recent **use** command) and the parameter **collectionB1** is the name to be given to the new collection.

Running the command

```
> show collections
```

will now verify that our new collection has been added to the database.

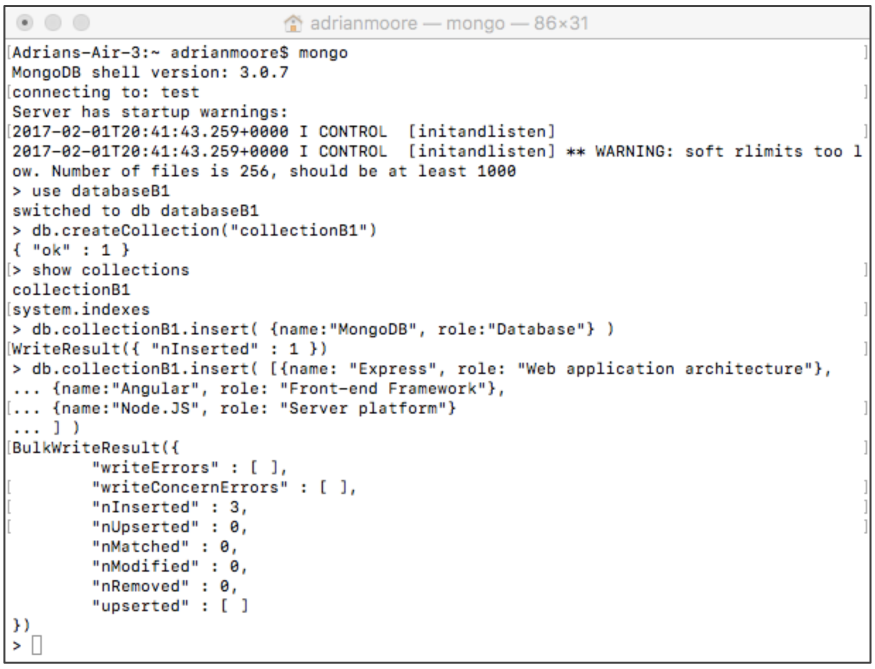
To add a document to a collection, we need to **insert** it into a collection on the currently active database by the command

> db.collectionB1.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.collectionB1.insert( [
            { name: “Express”, role: ”Web application architecture” },
            { name: “Angular”, role: “Front-end framework” },
            { name: “Node.JS”, role: “Server platform” }
       ] )
```

Figure B1.5 illustrates the output generated by the above sequence. Note that you can enter JSON data over multiple lines by using the ***<return>*** key.



```
Adrians-Air-3:~ adrianmoore$ mongo
MongoDB shell version: 3.0.7
connecting to: test
Server has startup warnings:
2017-02-01T20:41:43.259+0000 I CONTROL  [initandlisten]
2017-02-01T20:41:43.259+0000 I CONTROL  [initandlisten] ** WARNING: soft rlimits too l
ow. Number of files is 256, should be at least 1000
> use databaseB1
switched to db databaseB1
> db.createCollection("collectionB1")
{ "ok" : 1 }
> show collections
collectionB1
system.indexes
> db.collectionB1.insert( {name:"MongoDB", role:"Database"} )
WriteResult({ "nInserted" : 1 })
> db.collectionB1.insert( [{name: "Express", role: "Web application architecture"},
... {name:"Angular", role: "Front-end Framework"},
... {name:"Node.JS", role: "Server platform"}
... ] )
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 3,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> 
```

Figure B1.5 Using the Mongo Shell

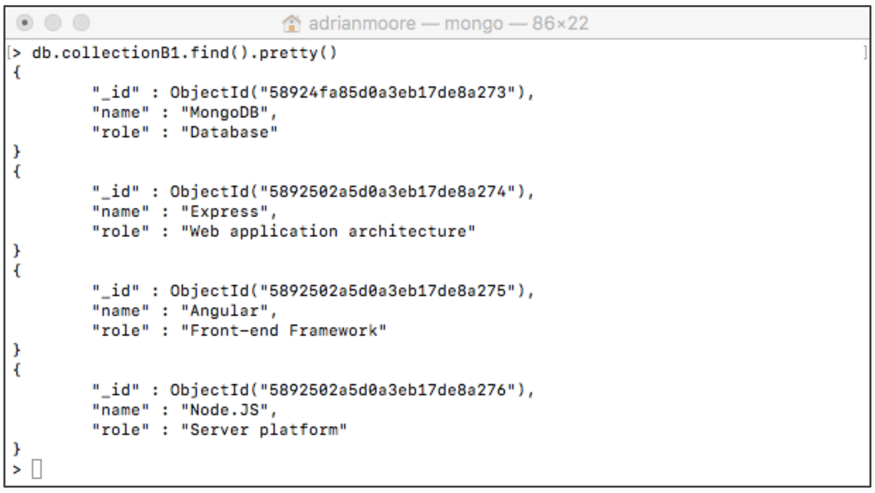
To retrieve documents from a collection, we chain the **find()** method to the database and collection

> db.collectionB1.find()

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

> db.collectionB1.find().pretty()

Figure B1.6 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 Mongo as each document was generated.



```
> 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 B1.6 Retrieving the collection

Try it now!

Revisit your JSON structure created earlier, and…

i) Create a new collection in the **databaseB1** database to store your data.

ii) Use **db.*collection*.insert()** to add your data to the new collection.
HINT: This will be quite a long command, so it is best to create it in a text editor before copying and pasting into the Console window.

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