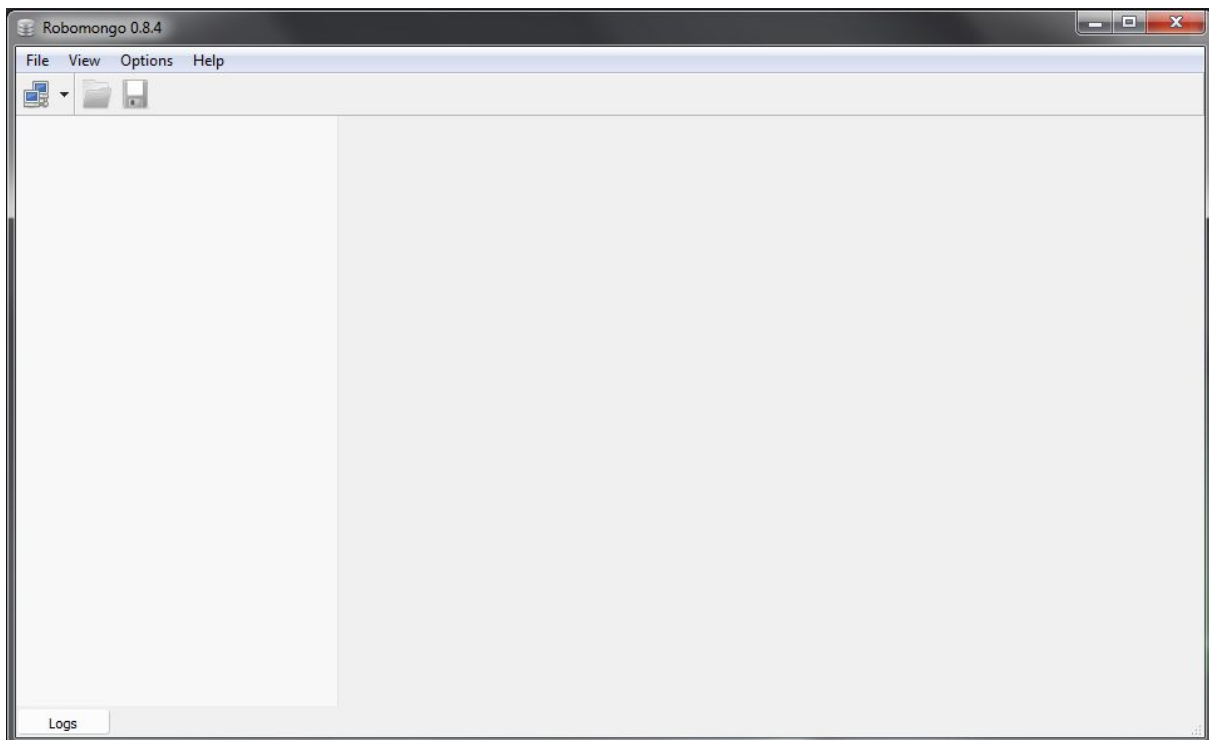


An Introduction to MongoDB

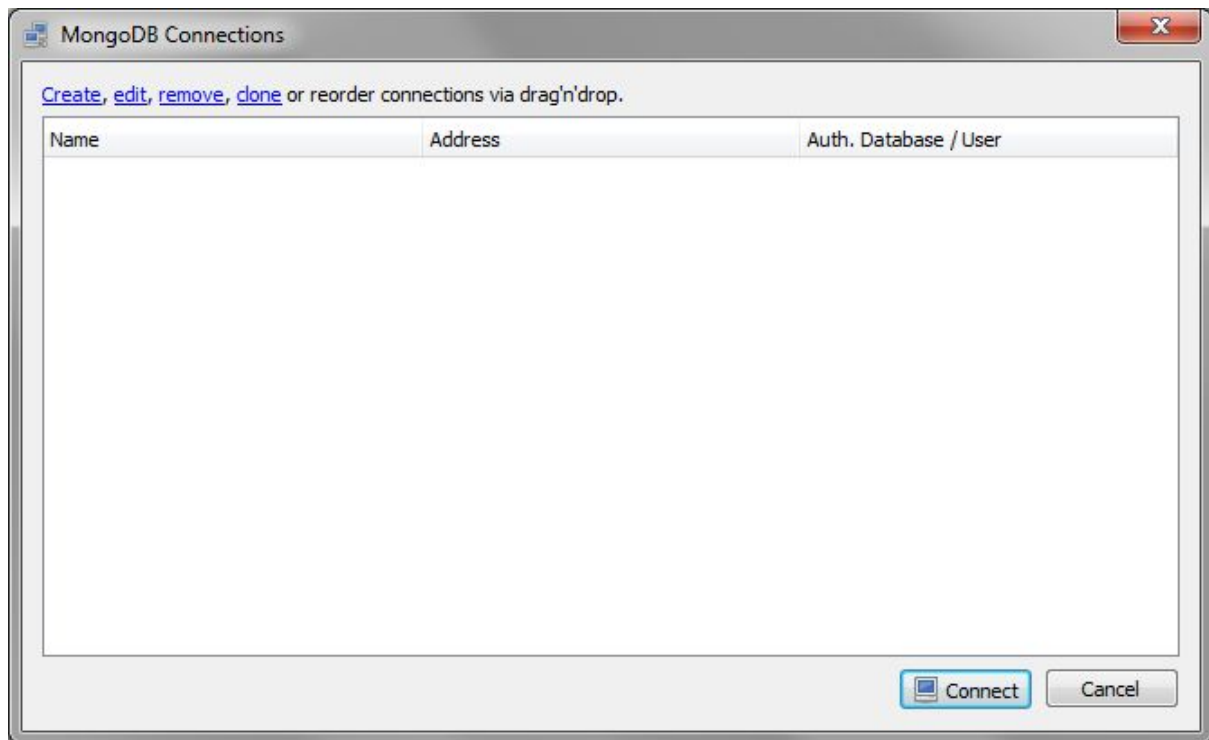
In this tutorial we will explore some of the basic concepts behind using MongoDB. We will use the freely available GUI Robo3T to run queries against the MongoDB server. There is a more advanced GUI available called Studio 3T which contains a similar shell (IntelliShell) for entering MongoDB queries, you may also use that if you wish.

Connecting to MongoDB

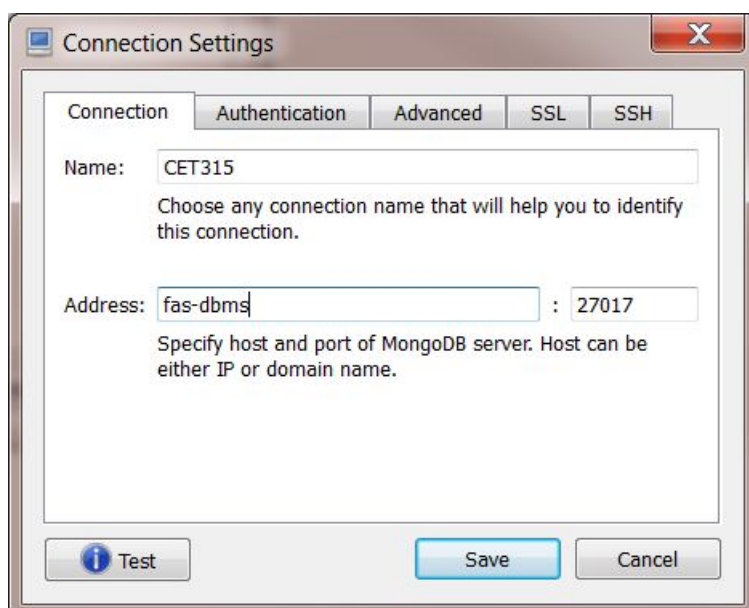
When you first run MongoDB, you should be presented with the following window.



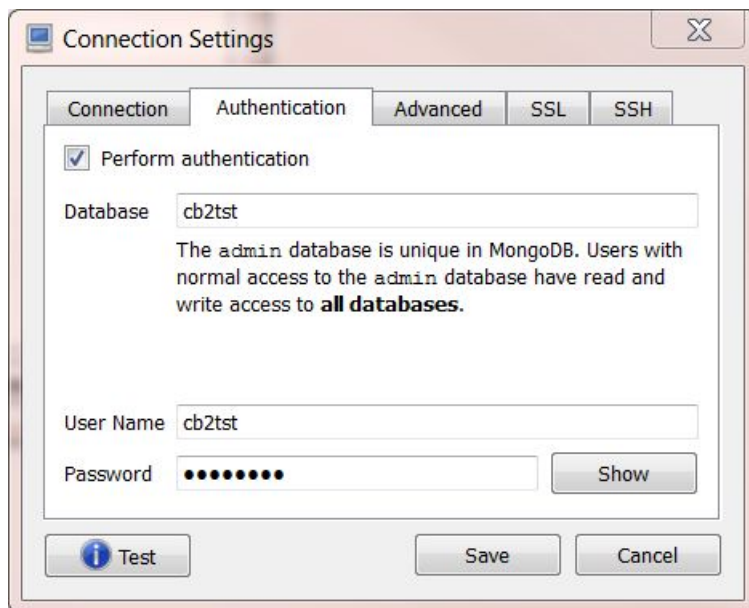
The first task is to create a connection to your MongoDB database. To do this, select the 'Connect' option within the 'File' menu, or alternatively press CTRL+O. The following window should appear:



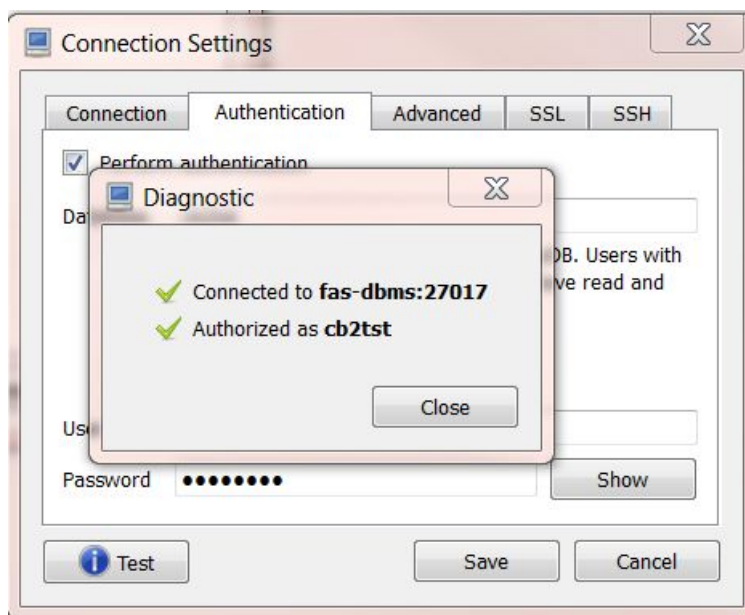
Choose Create. Specify a name for the Connection (it is recommended you enter the module code CET315, although you can enter any connection name), and the address fas-dbms in the Address field.



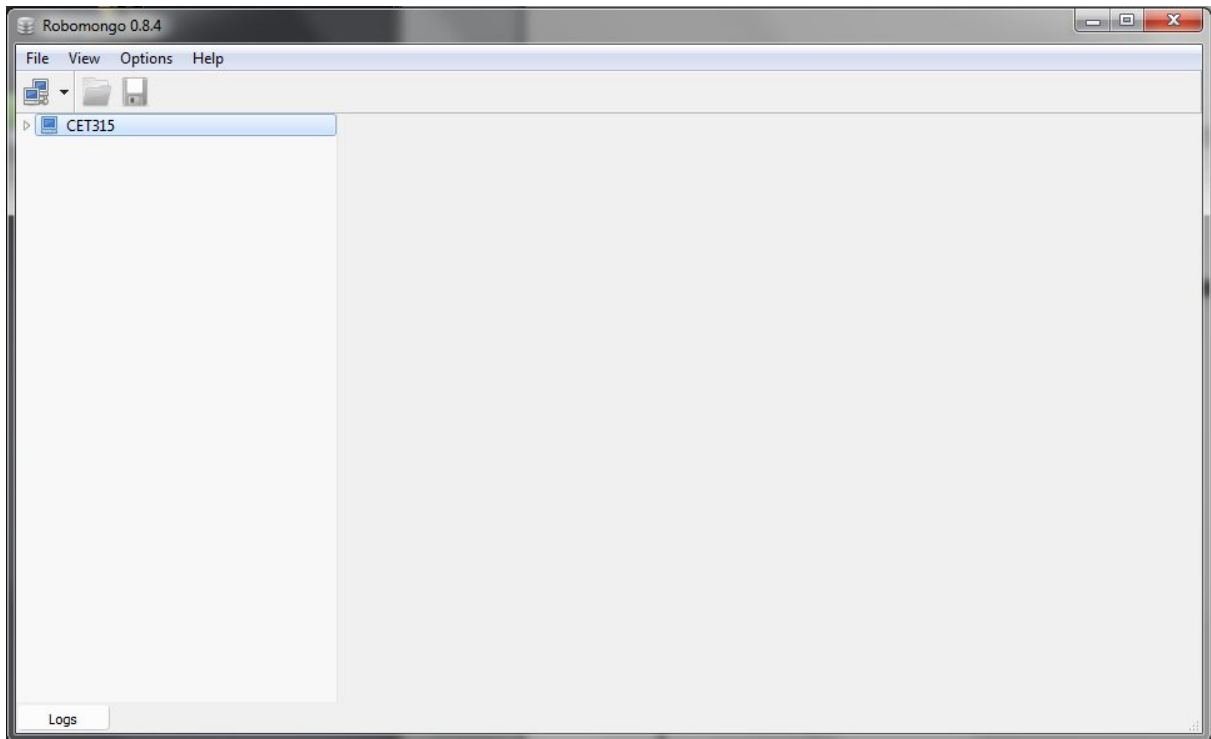
Then move to the Authentication tab. In the Database field, you should enter your user name. You should also enter your user name in the User Name field. Your password is your registration number.




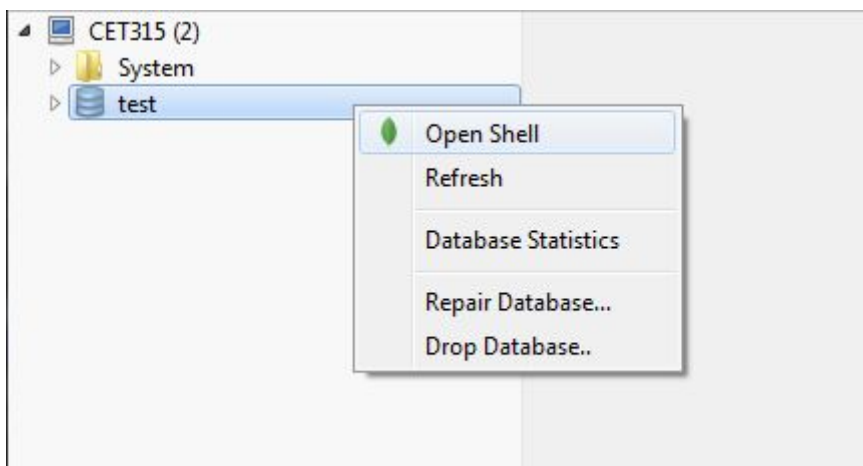
Before you click 'Save' you should test your connection with the 'Test' button. A dialog window, similar to the example shown below, should appear. You can then Save the connection.



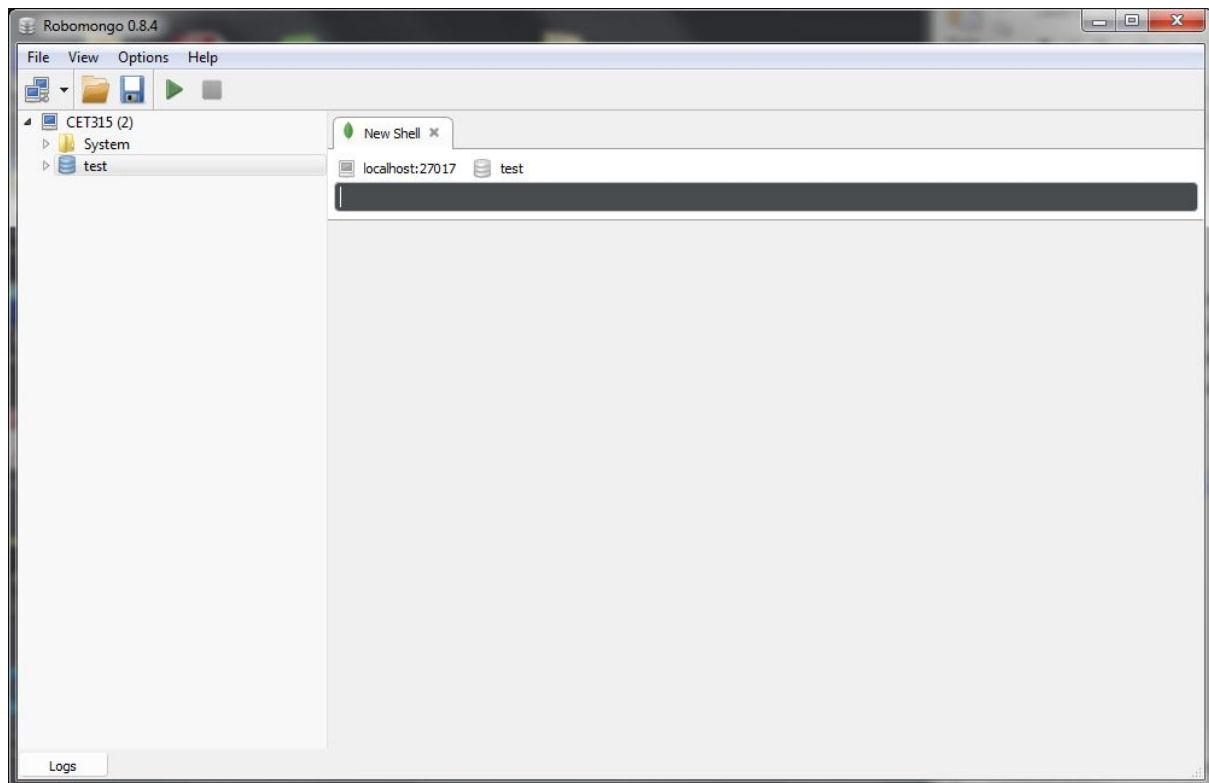
Once you have created and tested your connection, click the Connect button. If you have connected correctly, you should have a hierarchical menu appear on the left had side of your MongoDB application window, as shown below.



Click the  symbol next to the connection name to list your databases. You should see the name of your database listed. Move the cursor over the name of your database (in the example below the database is called test), click the right-hand mouse button and select the Open Shell option, as shown below.



MongoDB will then open a shell window in which you can enter and run MongoDB commands.



Creating Collections and Inserting Data

In MongoDB, we create **Collections** in which to store and group our data. Collections can loosely be thought of as similar to tables in a relational database, however as they are schemaless the format of data instances inside a collection can differ, unlike a relational database where each row is required to be the same structure. In MongoDB we call each data record a **document**. Within a document we have a collection of **key-value** pairs. For example, in the example below we are creating a new student **document** with one key-value pair, the **key** being **name**, and the **value** being **Joe**.

The simplest way to create a collection in MongoDB is to insert a document. To do this we use the **db.insert** command. Let's create a simple student document. In your shell, type the following MongoDB statement.

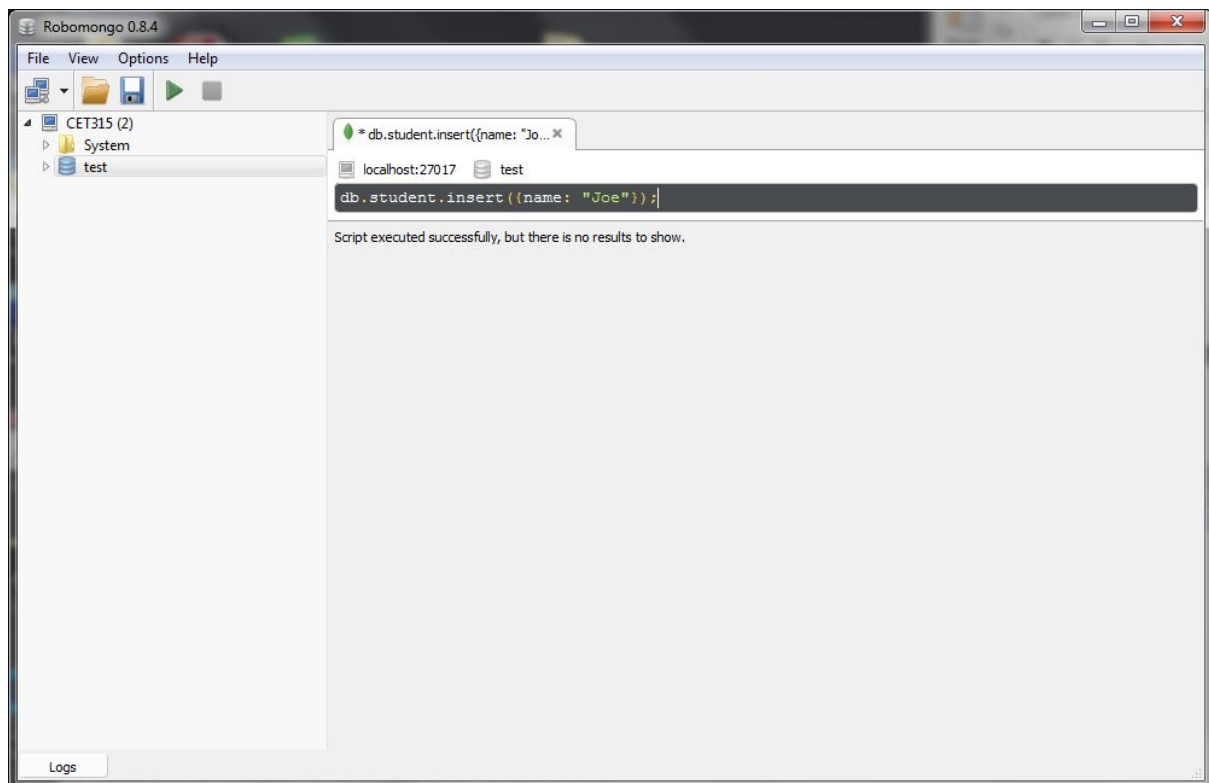
```
db.student.insert({name: "Joe"});
```

Just like in Oracle, don't forget the semi-colon at the end of the statement.



To execute the command, we use the button which can be found in the menu bar of MongoDB. Just like in Oracle SQL Developer, this will run all commands in the shell, or highlighted code if we have highlighted a particular command.

Run the command, and you should see the following result:



You should see the message:

```
Script executed successfully, but there is no results to show.
```

This means that your MongoDB statement has correctly run. If you get an error message, then correct the code to ensure it matches as above and run again.

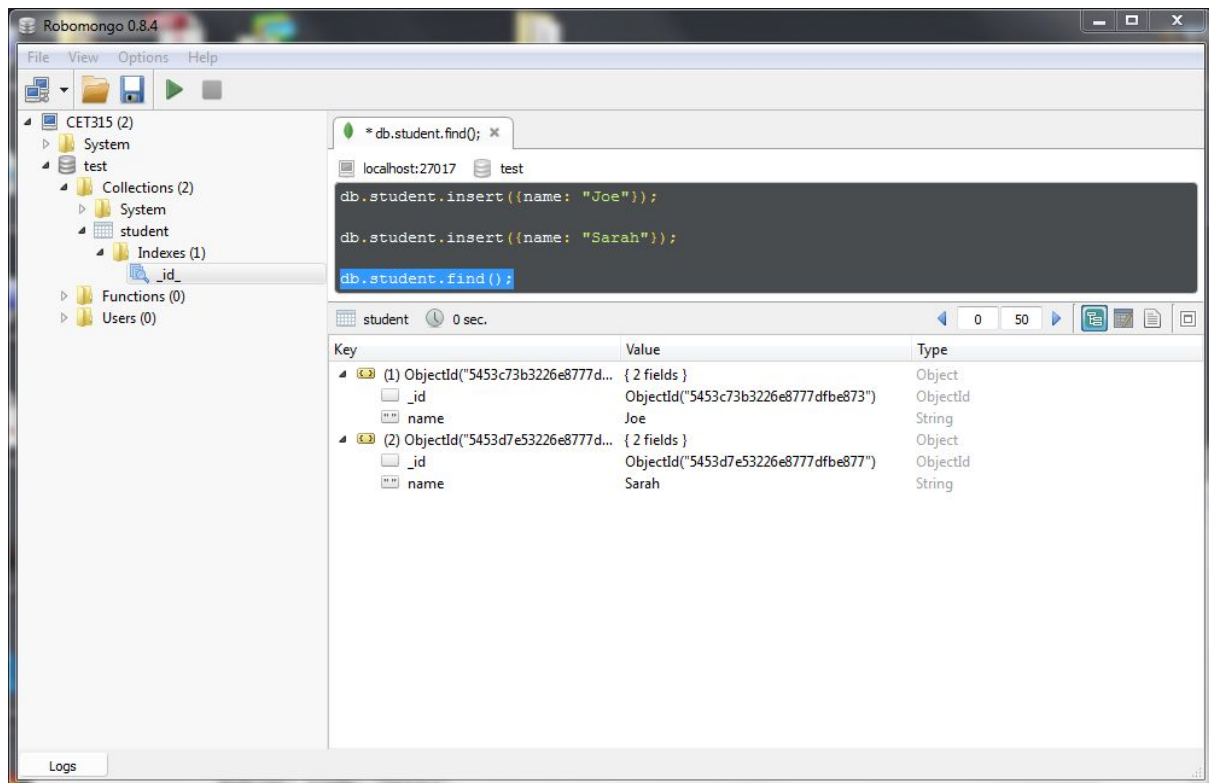
Congratulations! You have just inserted your first MongoDB document into a collection called student. Now, insert a second student, Sarah.


Querying

To query, we use the `find()` operation. As with the insert operation, we use the command

```
db.student.find();
```

to return all documents in the student collection, i.e. to return all students. Type and execute this command and you should get output similar to the example below:



You probably need to click the  icon next to each document to view its contents. You will also notice that the Object identifier for each document, i.e. the long hexadecimal number, will vary from the example output above. This is a system generated unique number so will vary from system to system.

To find a specific document we can specify a query condition to the find operation. For example, to find "Sarah" we would issue the query:

```
db.student.find({name: "Sarah"});
```

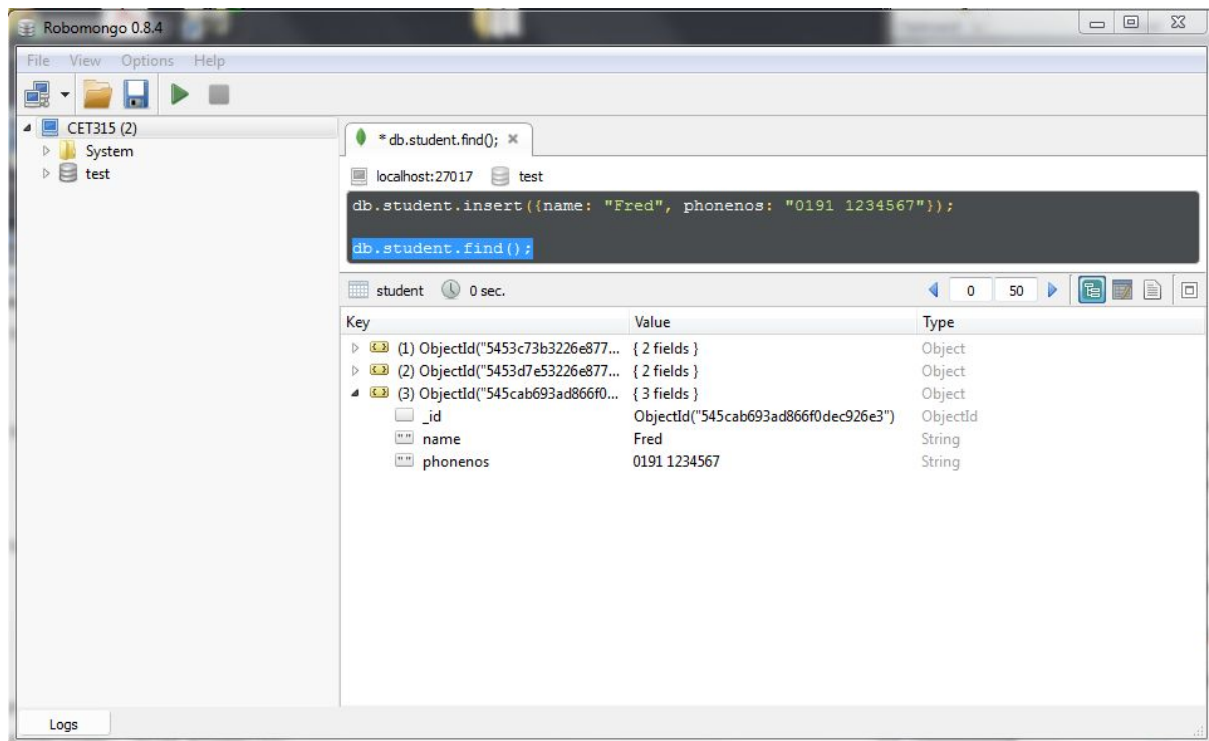
Run the query and ensure you take note of the results.

More Complex Documents

MongoDB allows us to create more structured documents that the ones we have seen so far. Say, for example, that we had a list of students where we want to store their contact details and information about the modules that they are taking. When we insert a record we can insert a comma separated list of values, for example run the command below:

```
db.student.insert({name: "Fred", phonenos: "0191 1234567"});
```

Now, use the command to retrieve all students, and compare the difference between the new student you have created compared to the previous two students, as shown below:



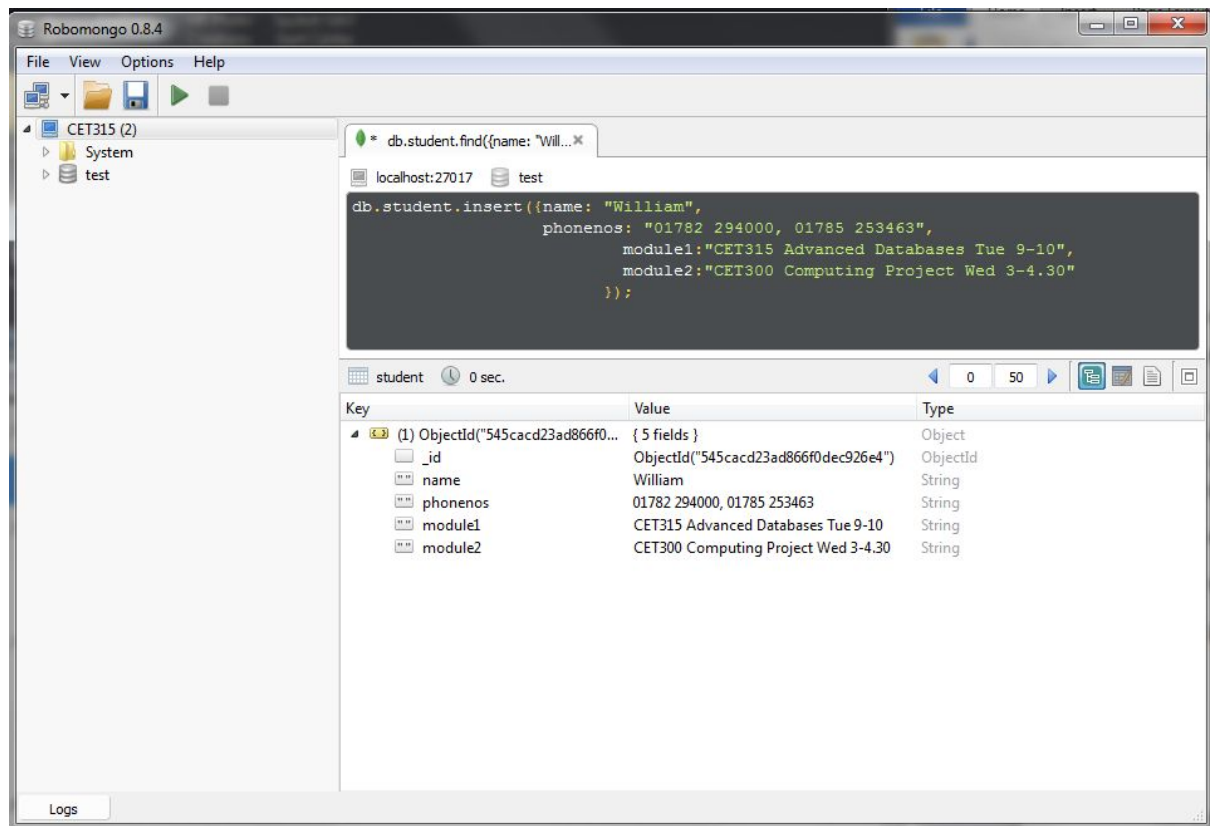
You will that your new student now has a telephone number value as well as a name.

Notice a big difference to relational databases, in that we are no longer restricted to all documents in the same collection having the same attributes. We are using a semi-structured schema, whereby we do not have a pre-defined schema (i.e. table definition as in Oracle), but our documents in a collection can have different structure.

Insert the following document.

```
db.student.insert({name: "William",
                    phonenos: "01782 294000, 01785 253463",
                    module1: "CET315 Advanced Databases Tue 9-10",
                    module2: "CET300 Computing Project Wed 3-4.30"
});
```

Run a query to retrieve only William, and view the results, which should appear as below.



This is a much more complex document,, however we can improve on this and introduce nesting of values, similar to the concept of a nested-relation in Oracle. Let's say, for example, for a module we want to include a list of assessments as well as the module name and information. Enter and run the following insert command:

```

db.student.insert({name: "Jane",

  phonenos: "01782 295000",

  module1:"Cet201 Information Systems Mon 2-6pm",

  assessments:{

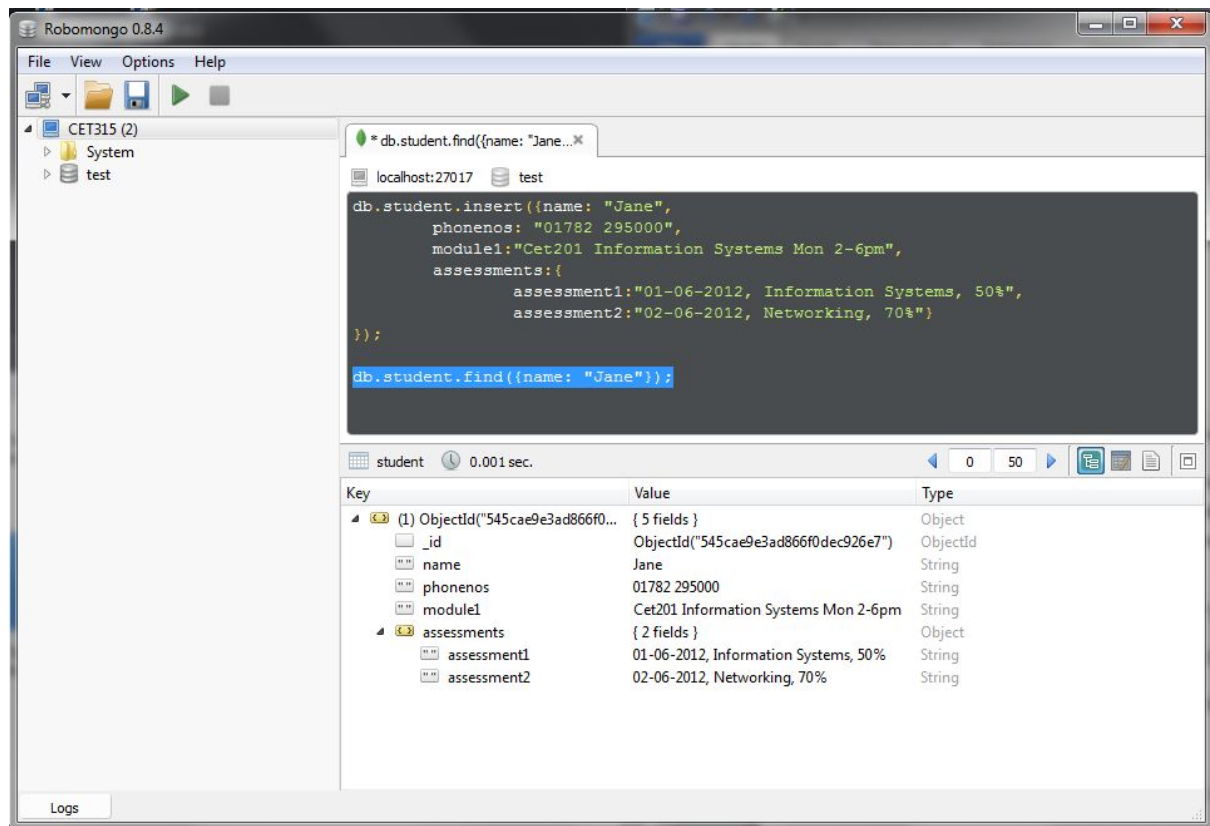
    assessment1:"01-06-2012, Information Systems, 50%",

    assessment2:"02-06-2012, Networking, 70%"}

});

```

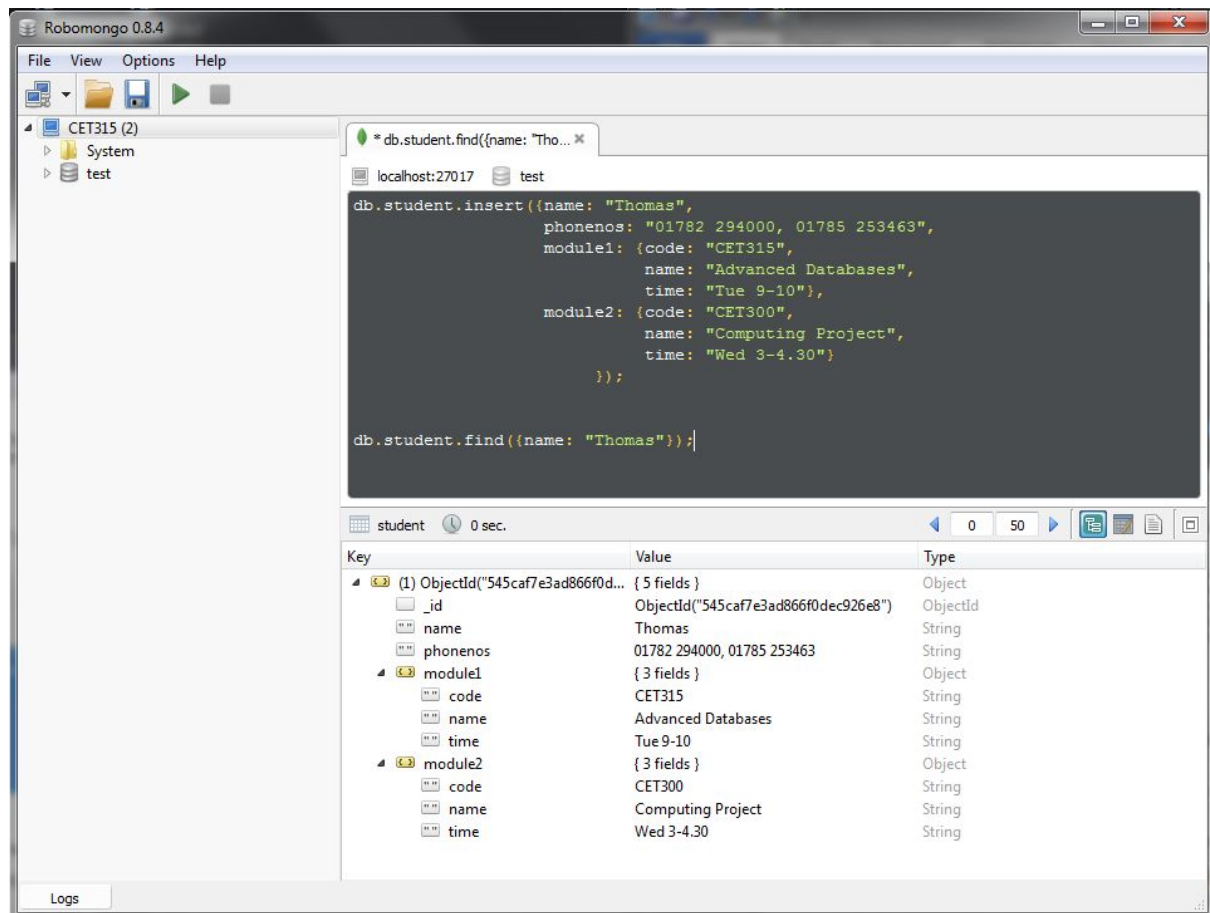
Run a query to retrieve the new document you have inserted, and you should see the result similar to the screenshot below:



And in our final example, we can have multiple values within a nested value, e.g.

```
db.student.insert({name: "Thomas",
  phonenos: "01782 294000, 01785 253463",
  module1: {code: "CET315",
    name: "Advanced Databases",
    time: "Tue 9-10"},
  module2: {code: "CET300",
    name: "Computing Project",
    time: "Wed 3-4.30"}
});
```

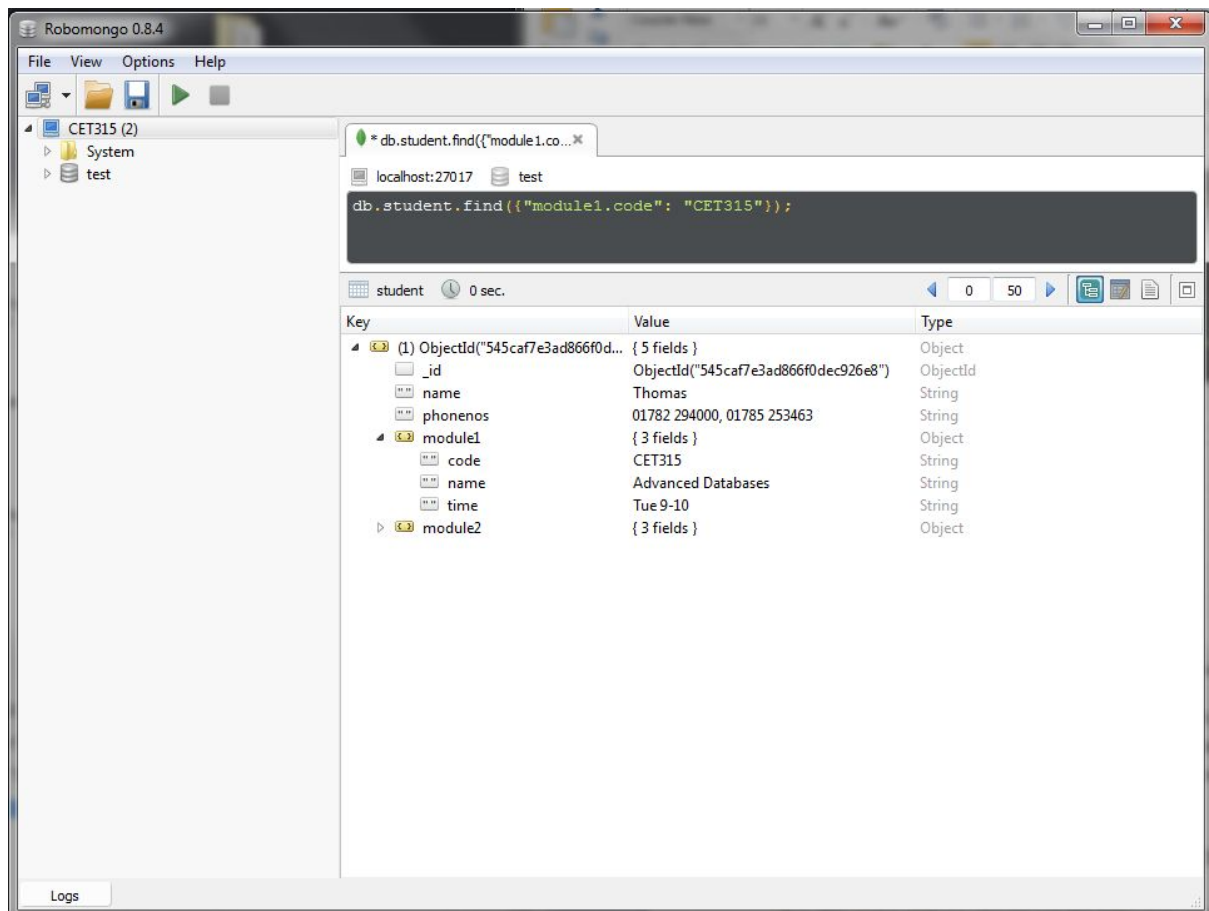
Retrieve your newly created document and the result should appear as below:



If we want to search within an embedded document, we use the nested dot notation similar to Oracle, however we must also insert the **key** in quotation marks. For example, to retrieve all students whose first module is CET315 would require the following query.

```
db.student.find({"module1.code": "CET315"});
```

Run this query and you should get the following result:



Other Queries

We can write more expressive queries, i.e. to search for multiple values or to test more than one condition. Run each of the following queries, and ensure you understand the result by viewing the resultant output:

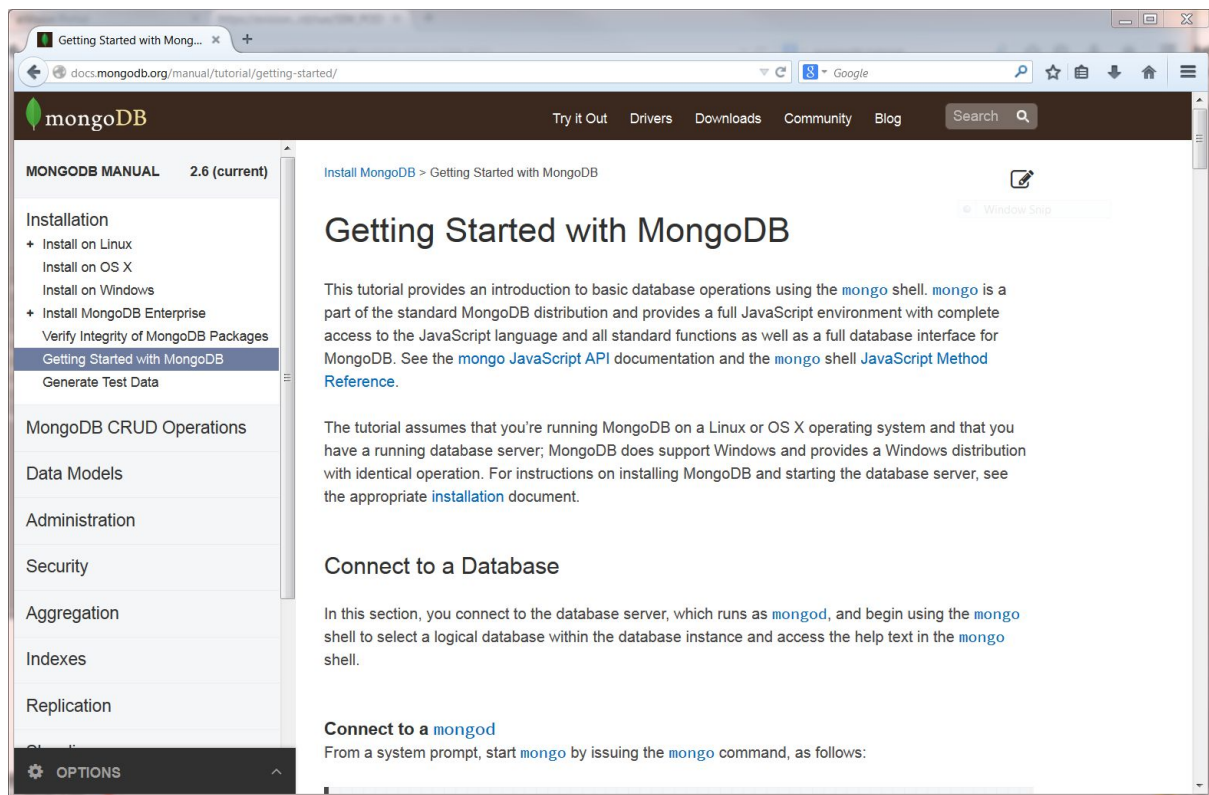
- `1. db.student.find({$or: [{name: "Jane"}, {phonenos: "0191 1234567"}]});`
- `2. db.student.find({$and: [{name: "Fred"}, {phonenos: "0191 1234567"}]});`
- `3. db.student.find({name: {"$in": ["Fred", "Jane"]} });`

Now Your Turn

This was only a very simple tutorial on MongoDB, to give you a taste of one of the more popular NoSQL systems. We have only considered very simple operations (create, retrieve) and not covered updates and deletions, nor how to link documents using ObjectIDs. These further operations are covered in the MongoDB tutorial which can be found at:

© David Nelson/Trijean Julien, version 2.0, April 2018

<http://docs.mongodb.org/manual/tutorial/getting-started/>



In this you should look at the sections 'Getting Started with MongoDB', 'Generate Test Data', and 'MongoDB CRUD Operations' which take you through more of the Create, Retrieve, Insert and Delete (CRUD) operations in MongoDB.

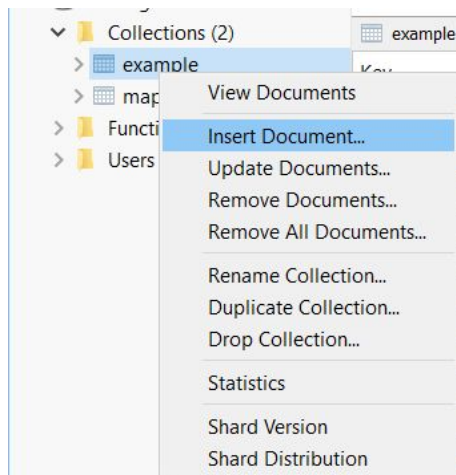
Learn how to manipulate data set

First of all you need to download a JSON Data set. For the example we will use a world bank data set (http://jsonstudio.com/wp-content/uploads/2014/02/world_bank.zip)

Unzip your word_bank.zip folder then follow these quick steps:

- 1- open JSON with notepad++
- 2- Select all (Ctrl+a)
- 3- Copy (Ctrl+c)
- 4- right click on the collection in Robot3T and then select "insert document"

© David Nelson/Trijean Julien, version 2.0, April 2018



5- Select all the content

6- Paste (Ctrl+v)

7- click on "save"

Then verify that all is well installed by using `db.test.find()` ; into Robo 3T

Filter and project data

1. Retrieve all data concerning “Kindom of Morocco” at the date of 2013. You should get 9 documents. Here is a screenshot of the data you have to find:

Key	Value	Type
> (1) ObjectId("52b213b3... { 43 fields }		Object
> (2) ObjectId("52b213b3... { 47 fields }		Object
> (3) ObjectId("52b213b3... { 49 fields }		Object
> (4) ObjectId("52b213b3... { 49 fields }		Object
> (5) ObjectId("52b213b3... { 47 fields }		Object
> (6) ObjectId("52b213b3... { 50 fields }		Object
> (7) ObjectId("52b213b3... { 47 fields }		Object
> (8) ObjectId("52b213b3... { 45 fields }		Object
▼ (9) ObjectId("52b213b3... { 45 fields }		Object
_id	ObjectId("52b213b38594d8a2b...	ObjectId
approvalfy	2013	String
board_approval_mo...	October	String
boardapprovaldate	2012-10-05T00:00:00Z	String
borrower	MOROCCAN ASSOCIATION FA...	String
closingdate	2013-08-31T00:00:00Z	String
country_namecode	Kingdom of Morocco!\$!MA	String
countrycode	MA	String
countryname	Kingdom of Morocco	String
countryshortname	Morocco	String

- Try now to keep the same data, but to include one more country like “Hashemite Kingdom of Jordan” at the same period. you should get 14 documents now. here a screenshot of what you may have:

Key	Value	Type
> (1) ObjectId("52b213b3...	{ 48 fields }	Object
> (2) ObjectId("52b213b3...	{ 43 fields }	Object
> (3) ObjectId("52b213b3...	{ 47 fields }	Object
> (4) ObjectId("52b213b3...	{ 49 fields }	Object
> (5) ObjectId("52b213b3...	{ 48 fields }	Object
> (6) ObjectId("52b213b3...	{ 49 fields }	Object
> (7) ObjectId("52b213b3...	{ 47 fields }	Object
> (8) ObjectId("52b213b3...	{ 50 fields }	Object
> (9) ObjectId("52b213b3...	{ 48 fields }	Object
> (10) ObjectId("52b213b3...	{ 47 fields }	Object
> (11) ObjectId("52b213b3...	{ 49 fields }	Object
▼ (12) ObjectId("52b213b3...	{ 44 fields }	Object
_id	ObjectId("52b213b38594d8a2b...	ObjectId
approvalfy	2013	String
board_approval_mo...	November	String
boardapprovaldate	2012-11-06T00:00:00Z	String
borrower	GOVERNMENT OF JORDAN	String
closingdate	2014-10-31T00:00:00Z	String
country_namecode	Hashemite Kingdom of Jordan!	String
countrycode	JO	String
countryname	Hashemite Kingdom of Jordan	String
countryshortname	Jordan	String

Now if we want to print out only certain values of the data we must add the key plus a value of 1 into a brace. This is an example with the key “lendprojectcost”:

trainingDataBase localhost:27017 trainingDataBase		
db.test.find({countryshortname:"Morocco"}, {lendprojectcost:1});		
test	0.001 sec.	0 50
Key	Value	Type
▼ (1) ObjectId("52b213b38...	{ 2 fields }	Object
_id	ObjectId("52b213b38594d8a2b...	ObjectId
lendprojectcost	200000000	Int32
▼ (2) ObjectId("52b213b38...	{ 2 fields }	Object
_id	ObjectId("52b213b38594d8a2b...	ObjectId
lendprojectcost	4550000	Int32
> (3) ObjectId("52b213b38...	{ 2 fields }	Object
> (4) ObjectId("52b213b38...	{ 2 fields }	Object
> (5) ObjectId("52b213b38...	{ 2 fields }	Object
> (6) ObjectId("52b213b38...	{ 2 fields }	Object
> (7) ObjectId("52b213b38...	{ 2 fields }	Object
> (8) ObjectId("52b213b38...	{ 2 fields }	Object
> (9) ObjectId("52b213b38...	{ 2 fields }	Object
> (10) ObjectId("52b213b3...	{ 2 fields }	Object
> (11) ObjectId("52b213b3...	{ 2 fields }	Object
> (12) ObjectId("52b213b3...	{ 2 fields }	Object

(if you don't want the id value in default you must add {"something":1, _id:0})

3. Print out only the name of their "mjsector_namecode" of the data you have selected just before. You may obtain something like this:

Key	Value	Type
> (1) ObjectId("52b213b38... { 2 fields }		Object
> (2) ObjectId("52b213b38... { 2 fields }		Object
> (3) ObjectId("52b213b38... { 2 fields }		Object
▼ (4) ObjectId("52b213b38... { 2 fields }		Object
_id	ObjectId("52b213b38594d8a2b...)	ObjectId
▼ mjsector_namecode	[5 elements]	Array
▼ [0]	{ 1 field }	Object
name	Education	String
▼ [1]	{ 1 field }	Object
name	Public Administration, Law, and ...	String
> [2]	{ 1 field }	Object
> [3]	{ 1 field }	Object
> [4]	{ 1 field }	Object
> (5) ObjectId("52b213b38... { 2 fields }		Object
> (6) ObjectId("52b213b38... { 2 fields }		Object
> (7) ObjectId("52b213b38... { 2 fields }		Object
> (8) ObjectId("52b213b38... { 2 fields }		Object
> (9) ObjectId("52b213b38... { 2 fields }		Object
> (10) ObjectId("52b213b3... { 2 fields }		Object
> (11) ObjectId("52b213b3... { 2 fields }		Object
> (12) ObjectId("52b213b3... { 2 fields }		Object
> (13) ObjectId("52b213b3... { 2 fields }		Object
> (14) ObjectId("52b213b3... { 2 fields }		Object

Now we must see all the operator we could use to filter the datas. we use them like that
 "something":{\$gt:10}.

Here is some operators:

\$gt \$gte	\$ls \$lse	\$ne	\$in \$nin	\$not	\$exists	\$size
Greater Than Equal	Less Than Equal	Not Equal	in not in	Negation	key Exists	size of a list

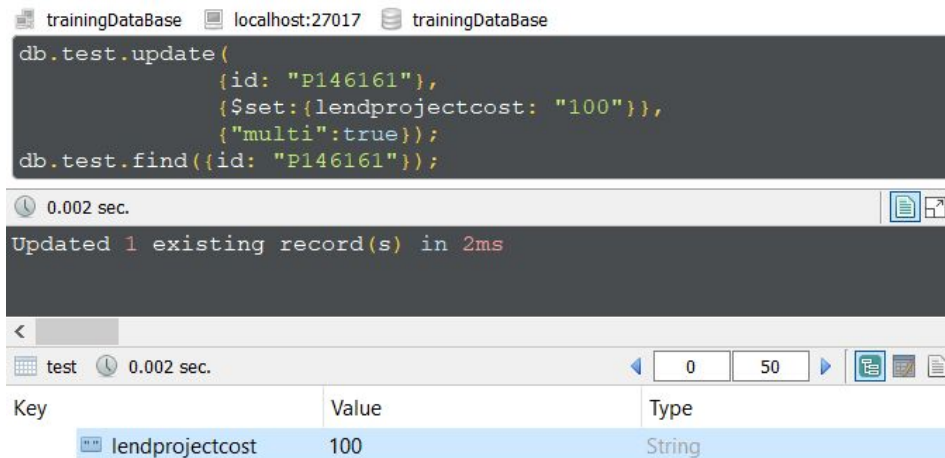
for example:

trainingDataBase localhost:27017 trainingDataBase		
db.test.find({countryshortname:"Morocco", sector:{ \$size: 5}});		
test	0.005 sec.	0 50
Key	Value	Type
> sector	[5 elements]	Array

4. Try to select any data concerning a project who cost more than 500 000 (the key is "lendprojectcost").

Update and delete datas

to update your data you must use `.update()` function and use the operator `$set` to precise witch values you want to change. For example:



```
trainingDataBase localhost:27017 trainingDataBase
db.test.update(
  {id: "P146161"},
  {$set:{lendprojectcost: "100"}},
  {"multi":true});
db.test.find({id: "P146161"});
```

0.002 sec.

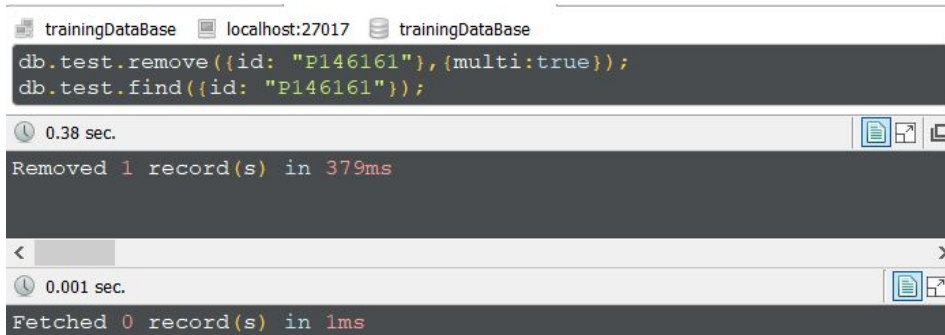
Updated 1 existing record(s) in 2ms

test 0.002 sec.

Key	Value	Type
lendprojectcost	100	String

The line containing `{"multi":true}` use to explain you can update multiple documents.

To delete a document you will use the `.remove()` function like the `.update` but without needing an operator. for example :



```
trainingDataBase localhost:27017 trainingDataBase
db.test.remove({id: "P146161"},{multi:true});
db.test.find({id: "P146161"});
```

0.38 sec.

Removed 1 record(s) in 379ms

0.001 sec.

Fetches 0 record(s) in 1ms

Exercise

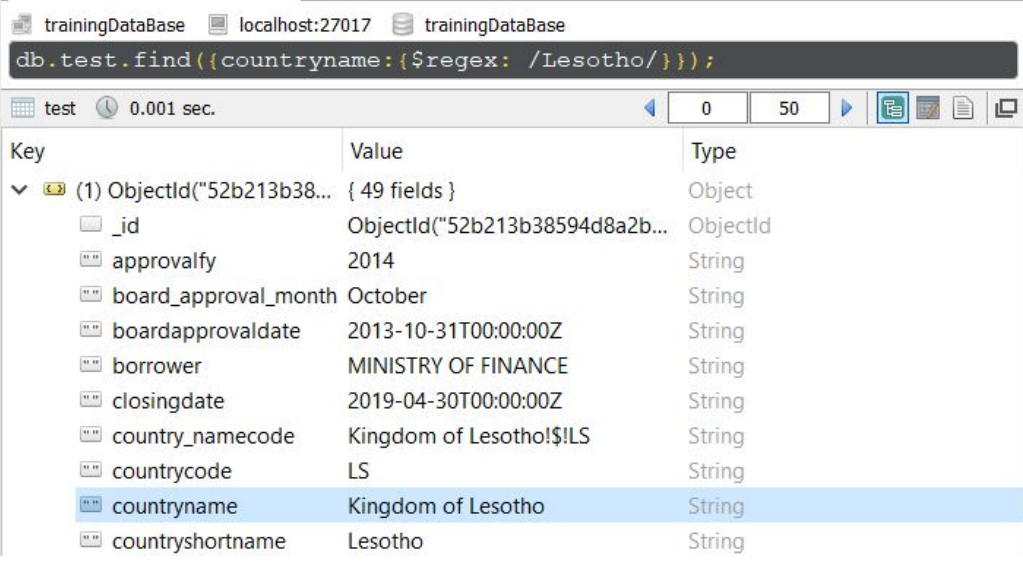
Think of a simple database that you would like to create, which would store one type of collection but individual records having different properties, for example a music collection where you could be storing information about CDs with differing numbers of tracks. Perform the following exercises, ensuring that you keep a record of your commands for each task in a text file. Once you have completed the exercise, mail your text file to the module tutor.

1. Create your own collection for your own database and insert at least 5 documents;
2. Ensure you can run a selection of queries to retrieve data from the documents you have inserted;
3. Update one of the documents you have created;
4. Delete one of the documents you have created;
5. Delete all of the documents you have created;

Pattern matching and Indexes

\$regex provides regular expression capabilities for pattern matching strings in queries.

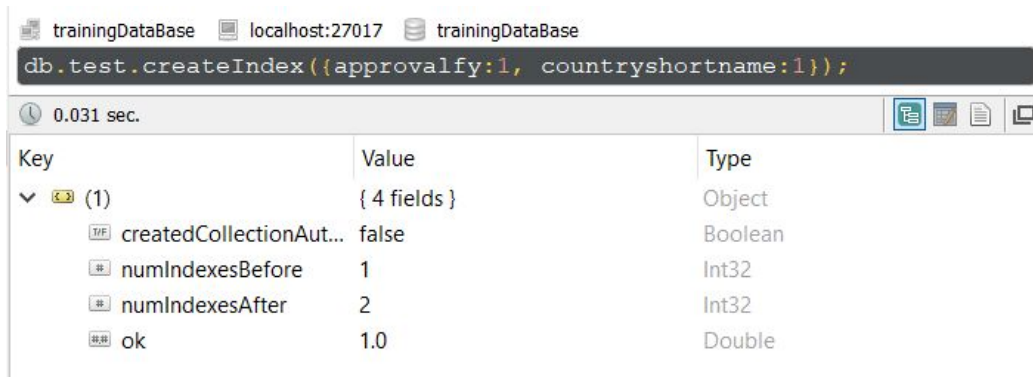
for example :



Key	Value	Type
(1) ObjectId("52b213b38...)	{ 49 fields }	Object
_id	ObjectId("52b213b38594d8a2b...)	ObjectId
approvalfy	2014	String
board_approval_month	October	String
boardapprovaldate	2013-10-31T00:00:00Z	String
borrower	MINISTRY OF FINANCE	String
closingdate	2019-04-30T00:00:00Z	String
country_namecode	Kingdom of Lesotho!\$!LS	String
countrycode	LS	String
countryname	Kingdom of Lesotho	String
countryshortname	Lesotho	String

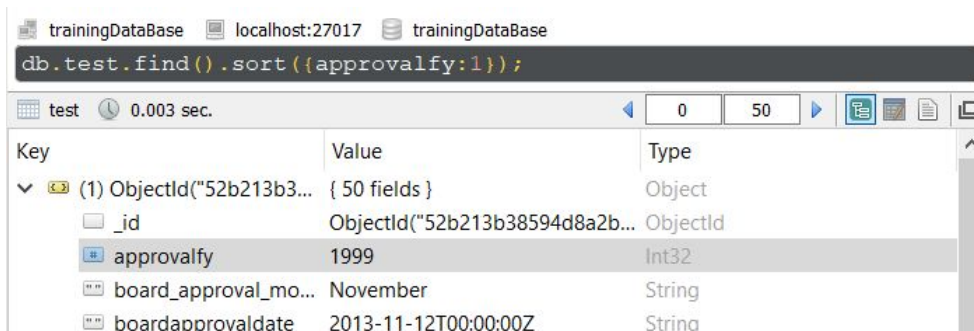
Sort operations that use an index often have better performance than those that do not use an index. In addition, sort operations that do not use an index will abort when they use 32 megabytes of memory.

to create an index you must do:



here I created multiple references in the index

To use on of the references you must do:



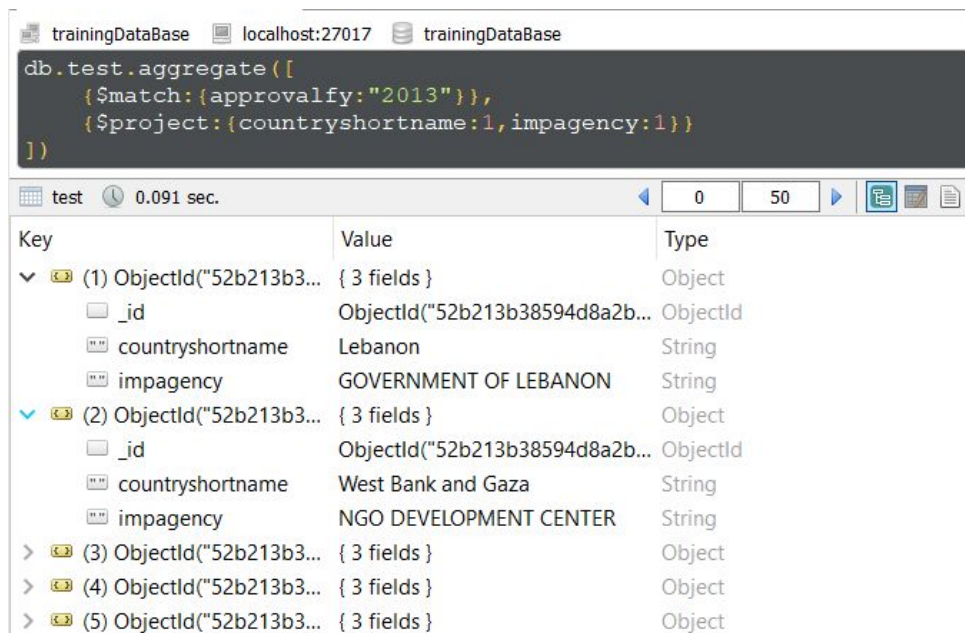
here I'm doing an ascending sort on "approvalfy" index.

Aggregate queries

the function .aggregate() use a pipeline (a sequence of data aggregation operations). Here is a description of the list of the operators that could be used into the aggregate:

<code>{ \$match: {} }</code>	<code>{ \$project: {} }</code>	<code>{ \$sort: {} }</code>	<code>{ \$group: {} }</code>	<code>{ \$unwind: {} }</code>
use like a filter	use like a print	use to sort the final result	group documents by value, the exit is a new collection	takes a list of value and produces for each element of the list a new document output with this element

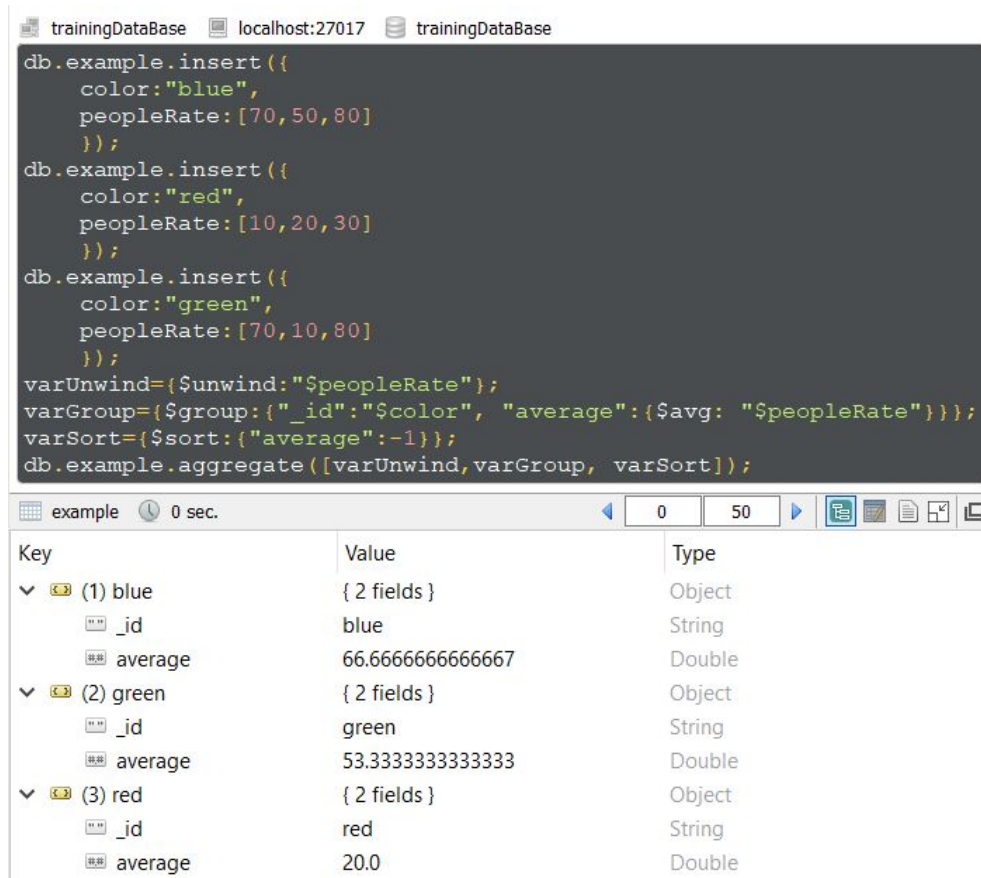
here is an example to do like a .find() with aggregation:



```
db.test.aggregate([
  {$match: {approvalfy: "2013"}},
  {$project: {countryshortname: 1, impagency: 1}}
])
```

Key	Value	Type
✓ (1) ObjectId("52b213b3...")	{ 3 fields }	Object
_id	ObjectId("52b213b38594d8a2b...")	ObjectId
countryshortname	Lebanon	String
impagency	GOVERNMENT OF LEBANON	String
✓ (2) ObjectId("52b213b3...")	{ 3 fields }	Object
_id	ObjectId("52b213b38594d8a2b...")	ObjectId
countryshortname	West Bank and Gaza	String
impagency	NGO DEVELOPMENT CENTER	String
> (3) ObjectId("52b213b3...")	{ 3 fields }	Object
> (4) ObjectId("52b213b3...")	{ 3 fields }	Object
> (5) ObjectId("52b213b3...")	{ 3 fields }	Object

Now I will show you an example of the 3 other operators, but to make the example more user-friendly, I put the operator into variable:



```
db.example.insert({
  color: "blue",
  peopleRate: [70, 50, 80]
});
db.example.insert({
  color: "red",
  peopleRate: [10, 20, 30]
});
db.example.insert({
  color: "green",
  peopleRate: [70, 10, 80]
});
varUnwind={$unwind: "$peopleRate"};
varGroup={$group: {"_id": "$color", "average": {$avg: "$peopleRate"}}};
varSort={$sort: {"average": -1}};
db.example.aggregate([varUnwind, varGroup, varSort]);
```

Key	Value	Type
✓ (1) blue	{ 2 fields }	Object
_id	blue	String
average	66.66666666666667	Double
✓ (2) green	{ 2 fields }	Object
_id	green	String
average	53.33333333333333	Double
✓ (3) red	{ 2 fields }	Object
_id	red	String
average	20.0	Double

© David Nelson/Trijean Julien, version 2.0, April 2018

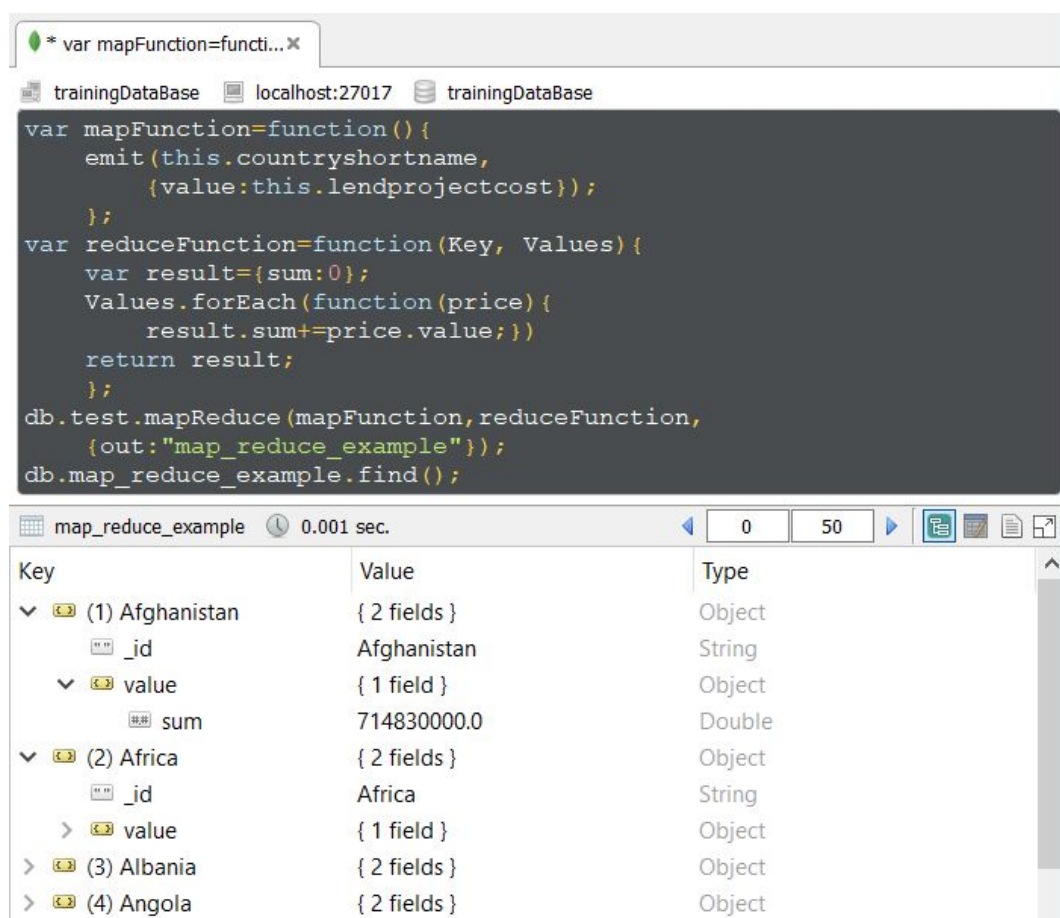
Here the \$key like "\$peopleRate" is to traverse over the value. So the unwind variable is to select each values from the list to get an average of them. If you want to sort the result in descending order you must do like me and use the "-1" value.

1. Try now to retrieve all average of "lendprojectcost" for a country in 2013 in the descending order.

Map-reduce

Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results.

for example I want to get the total price per country that the bank lend them:



The screenshot shows a MongoDB shell window with the following code executed:

```
var mapFunction=function() {
  emit(this.countryshortname,
    {value:this.lendprojectcost});
};
var reduceFunction=function(Key, Values){
  var result={sum:0};
  Values.forEach(function(price){
    result.sum+=price.value;})
  return result;
};
db.test.mapReduce(mapFunction,reduceFunction,
  {out:"map_reduce_example"});
db.map_reduce_example.find();
```

The results are displayed in a table with columns Key, Value, and Type. The results show the sum of lendprojectcost for each country.

Key	Value	Type
(1) Afghanistan	{ 2 fields }	Object
_id	Afghanistan	String
value	{ 1 field }	Object
sum	714830000.0	Double
(2) Africa	{ 2 fields }	Object
_id	Africa	String
value	{ 1 field }	Object
(3) Albania	{ 2 fields }	Object
(4) Angola	{ 2 fields }	Object

1. By helping you from the example try to retrieve the average percent of the "majorsector_percent.Name".