MongoDB

Getting MEAN 2 – A Practical Workshop

Sharif Malik | 2017

TABLE OF CONTENT

INTRODUCTION

What is MongoDB?

- MongoDB is an **open-source document database** and leading **NoSQL database**. MongoDB is written in C++.
- MongoDB is a **cross-platform**, **document oriented database** that provides, **high performance,** high availability, and easy scalability.
- MongoDB works on concept of **collection** and **document**.

Terminology

Database

- Database is a **physical container for collections**.
- Each database gets its own set of les on the le system. - A single MongoDB server typically has multiple databases.

Collection:

- Collection is a group of MongoDB documents.
- It is the equivalent of an **RDBMS table**.
- A collection exists within a single database.
- Collections do not enforce a schema.
- Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

- A document is a set of **key-value pairs**.
- Documents have **dynamic schema**.

Dynamic schema

- It means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Relationship between RDBMS terminology and MongoDB

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

Advantages of MongoDB over RDBMS

- **Schema less**:
  - MongoDB is a document database in which one collection holds different documents.
  - Number of fields, content and size of the document can differ from one document to another.
- **Structure** of a single object is clear. No complex joins.
- **Deep query-ability**. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- **Ease of scale-out**: MongoDB is easy to scale.

Why to Use?

- **Document Oriented Storage:** Data is stored in the form of JSON style documents.
- **Index on any attribute:** Replication and high availability - Rich queries - Fast in-place updates - Professional support by MongoDB

Where to Use?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Installation

- **Step 1: Installation steps**

You can find all the steps of installation on this below link:
https://docs.mongodb.com/manual/tutorial/install-mongodb-on- windows/

- **Step 2: Setup the MongoDB environment:**
  - MongoDB requires a data directory to store all data. MongoDB's default data directory is \data\db. Create this folder using the commands from command prompt:

  - $ md \data\db

  You can specify an alternative path for data files using the –**dbpath** option to **mongod.exe**

  - **Example:**

  C:\mongodb\bin\mongod.exe –dbpath d:\test\mongodb\data

  Note: If your path includes spaces, enclose the entire path in double quotes.
  For example:
  C:\mongodb\bin\mongod.exe –dbpath "d:\test\mongo db\data"

- **Step 3: Connect to MongoDB**
  To connect to mongodb through the mongo.exe shell, open another command prompt.

  C:\mongodb\bin\mongo.exe

Commands

Miscellaneous Commands

MongoDB Help

To get a list of commands, type **db.help()** in MongoDB client.
Description: This will give you a list of commands.

MongoDB Statistics

To get stats about MongoDB server, type the command **db.stats()** in MongoDB client.
Description: This will show the database name, number of collection and documents in the database.

Database commands

Create Database

- The **use** Command
  MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

  Syntax
  Basic syntax of use DATABASE statement is as follows:
  use DATABASE_NAME

  Example:
  If you want to create a database with name <testdb>, then use DATABASE statement would be as follows:
  >use testdb
  switched to db testdb

  To check your currently selected database, use the command **db**
  >db
  testdb

  If you want to check your databases list, use the command show dbs.
  > show dbs
  Your created database (testdb) is not present in list. To display database, you need to insert at least one document into it.

Note: In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

Drop Database

- The **dropDatabase()** Method
  MongoDB **db.dropDatabase()** command is used to drop a existing database.

  Syntax
  Basic syntax of dropDatabase() command is as follows:
  db.dropDatabase()
  This will delete the selected database. If you have not selected any database, then it will delete default **'test'** database.

  Example:
  First, check the list of available databases by using the command, **show dbs**.
  >show dbs
  local 0.78125GB
  mydb 0.23012GB
  test 0.23012GB
  >

  If you want to delete new database **<mydb>,** then dropDatabase() command would be as follows:
  >use mydb
  switched to db mydb
  >db.dropDatabase()
  >{ "dropped" : "mydb", "ok" : 1 }

  Now check list of databases.
  >show dbs local 0.78125GB test 0.23012GB

Collection commands

For more information(https://docs.mongodb.com/manual/reference/method/#collection)

Create

- The **createCollection()** Method
  MongoDB db.createCollection(name, options) is used to create collection.

- **Syntax**
  Basic syntax of createCollection() command is as follows:
  db.createCollection(name, options)

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Note: Options parameter is optional, so you need to specify only the name of the collection.

Following is the list of options you can use:

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
autoIndexID	Boolean	(Optional) If true, automatically create index on _id field. Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Drop

- The **drop()** Method
  MongoDB's db.collection.drop() is used to drop a collection from the database.

- **Syntax**
  db.COLLECTION_NAME.drop()

- **Example**
  First, check the available collections into your database mydb.
  >use mydb
  switched to db mydb
  >show collections
  users students persons

  Now drop the collection with the name mycollection.
  >db.users.drop()
  true

  Again, check the list of collections into database.
  >show collections
  students persons

- **Note:** drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

Insert

- **The insert() Method:**

To insert data into MongoDB collection, you need to use MongoDB's insert() or save()method.

- **Syntax**:
  >db.COLLECTION_NAME.insert(document)

- **Example**
  >db.mycol.insert({ "_id": ObjectId(57da4c69a8bb6d54a9d8b147), "title": 'MongoDB Example', "description": 'MongoDB is no sql database', "likes": 100 })

- **Explanation:**
  Here mycol is our collection name, as created in the previous section. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

- **Note:** In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document.

Find

❖ **The find() Method**

To query data from MongoDB collection, you need to use MongoDB's find() method.

❖ **Syntax:**

>db.COLLECTION_NAME. find()

Pretty

❖ **The pretty() Method**

To display the results in a formatted way, you can use pretty() method.

Syntax:

>db.mycol.find().pretty()

Example:

>db.mycol.find().pretty() {
"_id": ObjectId(57da4c69a8bb6d54a9d8b1
"title": "MongoDB Overview",
"description": "MongoDB is no sql database"
}

Note: Apart from find() method, there is findOne() method, that returns only one document.

Update

❖ **The update() Method**

The update() method is use to update the values in the existing document.

❖ **Syntax**:

> **db.COLLECTION_NAME.update (SELECTIOIN_CRITERIA, UPDATED_DATA)**

❖ **Example:**

Consider the mycol collection has the following data.

{ "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"Mean Workshop"}
{ "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"ng works"}
{ "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"meaningfull success"}

Following example will set the new title 'New MongoDB Update' of the documents whose title is 'Mean Workshop'.

>db.mycol.update({'title':'Mean Workshop'},{$set: {'title':'New MongoDB Update'}})

Remove

❖ **The remove() Method**

The remove() method is used to remove a document from the collection.remove() method accepts two parameters.

1. **deletion criteria:** (Optional) deletion criteria according to documents will be removed.
2. **justOne:** (Optional) if set to true or 1, then remove only one document.

❖ **Syntax:**

>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)

❖ **Example:**

Consider the mycol collection has the following data.

{ "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"Mean Workshop"}
{ "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"ng works "}
{ "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"meaningfull success"}

Following example will remove all the documents whose title is 'ng works'.

>db.mycol.remove({'title':'ng works'})
>db.mycol. find()
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"Mean Workshop"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"meaningfull success"}

❖ **RemoveOne Method:**

If there are multiple records but you want to delete only the first record, then set second parameter in remove() method as 1.

Example:

>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)

❖ **Remove All Documents:**

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. This is equivalent of SQL's truncate command.

>db.mycol.remove({})
>db.mycol.find()

Where Clause

RDBMS Where Clause Equivalents in MongoDB
To query the document based on some condition, you can use following operations:

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({ "by": "sharif malik" })	Where by = "sharif malik"
Less than	{<key>: {$lt:<value>}}	db.mycol.find({"age"{$lt:30}})	Where age < 30
Less Than Equals	{<key> {$lte :<value>}}	db.mycol.find({"age":{$lte:30}})	Where age <= 30
Greater than	{<key>:{$gt : <value>}}	db.mycol.find({"age":{$gt:20}})	Where age > 20
Greater than equals	{<key>:{$gte :<value>}}	db.mycol.find({"likes":{$gte:20}})	Where age >= 20
Not equals	{<key>: {$ne:<value>}}	db.mycol.find({"age": {$ne:24}})	

- **AND in MongoDB**

In the find() method, if you pass multiple keys by separating them by ',' then MongoDB treats it as AND condition.
Syntax:
>db.mycol.find({key1:value1, key2:value2})

- **OR in MongoDB**

To query documents based on the OR condition, you need to use $or keyword.
Syntax:
>db.mycol.find({ $or: [{key1: value1},{key2:value2}] })

Projection:

- **Explanation:**

Projection means selecting only the necessary data rather than selecting whole of the data of a document.
If a document has 15 fields and you need to show only 3, then select only 3 fields from them.

- **The find() Method**

MongoDB's find() method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you can execute find() method, then it displays all fields of a document.

To limit this, you need to set a list of fields with **value 1 or 0**.
1 is used to show the field while **0 is used to hide** the fields.

- **Syntax:**
  >db.COLLECTION_NAME.find({},{KEY:1})

- Example:
  Consider the collection mycol has the following data
  { "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"Mean Workshop"}
  { "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"ng works"}
  { "_id" : ObjectId(57da4c69a8bb6d54a9d8b147), "title":"meaningfull success"}

  Following example will display the title of the document while querying the document.

  >db.mycol.find({},{"title":1,_id:0})
  {"title":"mean workshop"} {"title":"ng works"} {"title":"meaningfull success"}

- **Note:** _id field is always displayed while executing find() method, if you don't want this field, then you need to set it as 0.

Datatypes

MongoDB supports many datatypes. Some of them are:

- **String**
  - This is the most commonly used datatype to store the data.
  - String in MongoDB must be UTF-8 valid.
- **Integer**
  - This type is used to store a numerical value.
  - Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean**
  - This type is used to store a Boolean (true/ false) value.
- **Double**
  - This type is used to store oating point values.
- **Min/Max Keys**
  - This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays**
  - This type is used to store arrays or list or multiple values into one key.
- **Timestamp**
  - This can be handy for recording when a document has been modified or added.
- **Null**
  - This type is used to store a Null value.
- **Symbol**
  - This datatype is used identically to a string;
  - however, it's generally reserved for languages that use a specific symbol type.
- **Date**
  - This datatype is used to store the current date or time in UNIX time format.
  - You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID**
  - This datatype is used to store the document's ID.
- **Binary data**
  - This datatype is used to store binary data.
- **Code**
  - This datatype is used to store JavaScript code into the document.
- **Regular expression**
  - This datatype is used to store regular expression.

Object Id:

- MongoDB uses ObjectIds as the default value of _id field of each document, which is generated while the creation of any document.
- The complex combination of ObjectId makes all the _id fields unique.
- An **ObjectId** is a 12-byte BSON type having the following structure:
  - The first 4 bytes representing the seconds since the unix epoch
  - The next 3 bytes are the machine identifier
  - The next 2 bytes consists of process id
  - The last 3 bytes are a random counter value

Creating New ObjectId

To generate a new ObjectId use the following code:

>myObjectId = ObjectId()

The above statement returned the following uniquely generated id:

> ObjectId("57da4c69a8bb6d54a9d8b147")

Instead of MongoDB generating the ObjectId, you can also provide a 12-byte id:

>myObjectId =ObjectId("57da4ca5a8bb6d54a9d8b148")

Fetching Timestamp of a Document

Since the _id ObjectId by default stores the 4-byte timestamp, in most cases you do not need to store the creation time of any document. You can fetch the creation time of a document using **getTimestamp()** method.

- **Example:**

  > myObjectId.getTimestamp()

  This will return the creation time of this document in ISO date format.

  > ISODate("2017-08-28T18:11:53Z")

Converting ObjectId to String

In some cases, you may need the value of ObjectId in a string format.
To convert the ObjectId in string, use the following code:

- **myObjectId.str**

OR

- **myObjectId.valueOf()**

The above code will return the string format of the ObjectId:

>59a45ce95ae0cf6e82241e3d

Mongoose

What is Mongoose

- Mongoose was built specifically as a Mongo DB Object-Document Modeller (ODM) for Node applications.
- One of the key principles is that you can manage your data model from within your application.
- You don't have to mess around directly with databases or external frameworks or relational mappers; you can just de ne your data model in the comfort of your application.

Why model the data?

- We had already talked about how Mongo DB is a document store, rather than a traditional table-based database using rows and columns.
- This allows Mongo DB great freedom and flexibility, but sometimes we need: **structure to our data**.
- **Example:**

```
{
  "firstname" : "Simon",
  "surname" : "Holmes",
  _id : ObjectId("52279effc62ca8b0c1000007")
}
```

Example MongoDB document

```
{
  firstname : String,
  surname : String
}
```

Corresponding Mongoose schema

- **Explanation**: The schema bears a very strong resemblance to the data itself. The schema defines the name for each data path, and the data type it will contain. In this example, we've simply declared the paths firstname and surname as strings.

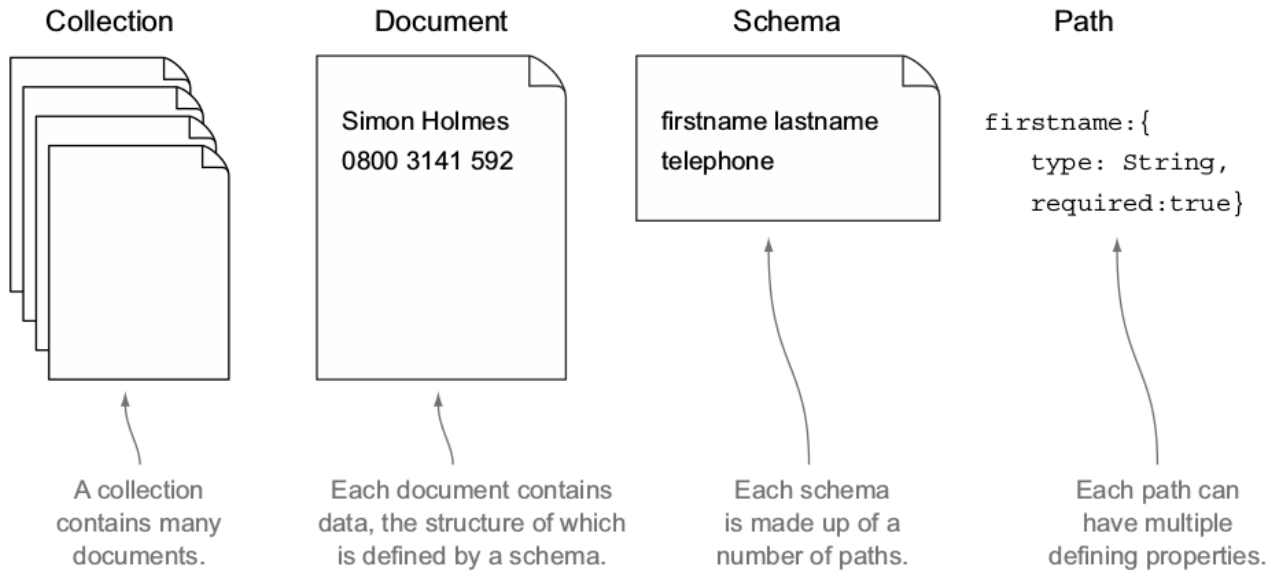
- **About the _id path:**
  - You may have noticed that we haven't declared the id path in the schema.
  - **_id** is the unique identifier: the primary key
  - MongoDB automatically creates this path when each document is created and assigns it a unique ObjectId value.

- The value is designed to always be unique by combining the time since the Unix epoch with machine and process identifiers and a counter.

❖ **Conclusion:**

Here is the naming conventions comparison:

- In MongoDB, **each entry in a database** is called a **document**.
- In MongoDB, a **collection of documents** is called a **collection.** (think about "table" if you're used to relational databases).
- In Mongoose, the definition of a **document** is called a **schema**.
- Each individual data entity defined in a **schema** is called a **path**.



Schema

❖ Everything in Mongoose starts with a Schema.

❖ Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

❖ **Example**:

```
1. var mongoose = require('mongoose');
2. var Schema = mongoose.Schema;
3.
4. var blogSchema = new Schema({
5.   title:  String,
6.   author: String,
7.   body:   String,
8.   comments: [{ body: String, date: Date }],
```

```
9.    date: { type: Date, default: Date.now },
10.   hidden: Boolean,
11.   meta: {
12.     votes: Number,
13.     favs:  Number
14.   }
15.});
```

Each key in our blogSchema defines a property in our documents which will be cast to its associated SchemaType. For example, we've defined a title which will be cast to the String SchemaType and date which will be cast to a Date SchemaType. Keys may also be assigned nested objects containing further key/type definitions ***(e.g. the `meta` property above).***

- **SchemaTypes**: The permitted SchemaTypes are

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array

Read more about them here (http://mongoosejs.com/docs/schematypes.html)

- **NOTE**:

  Schemas not only define the structure of your document and casting of properties, they also define document **instance** (http://mongoosejs.com/docs/guide.html#methods),**static Model methods**(http://mongoosejs.com/docs/guide.html#statics), **compound indexes** (http://mongoosejs.com/docs/guide.html#indexes), and document lifecycle hooks called **middleware** (http://mongoosejs.com/docs/middleware.html).

Models

- Models are fancy constructors compiled from our Schema definitions.
- Instances of these models represent documents which can be saved and retrieved from our database.
- All document creation and retrieval from the database is handled by these models.

❖ **Compiling first model:**

```
var schema = new mongoose.Schema({ name: 'string', size: 'string' });
var Tank = mongoose.model('Tank', schema);
```

❖ **Note:**

- The first argument is the *singular* name of the collection your model is for.
- **Mongoose automatically looks for the *plural* version of your model name.**
- Thus, for the example above, the model Tank is for the **tanks** collection in the database.
- The .model() function makes a copy of schema. Make sure that you've added everything you want to schema before calling .model() method.

❖ **Constructing documents:**

Documents are instances of our model. Creating them and saving to the database is easy.

```
var Tank = mongoose.model('Tank', yourSchema);

var small = new Tank({ size: 'small' });
small.save(function (err) {
  if (err) return handleError(err);
  // saved!
})

// or

Tank.create({ size: 'small' }, function (err, small) {
  if (err) return handleError(err);
  // saved!
})
```

Note: that no tanks will be created/removed until the connection your model uses is open. Every model has an associated connection. When you use mongoose.model(), your model will use the default mongoose connection. i.e.

```
mongoose.connect('localhost', 'gettingstarted');
```

If you create a custom connection, use that connection's model() function instead.

```
var connection = mongoose.createConnection('mongodb://localhost:27017/test');
var Tank = connection.model('Tank', yourSchema);
```

Querying

Finding documents is easy with Mongoose, which supports the rich query syntax of MongoDB. Documents can be retrieved using each models find, findById, findOne, or where static methods.

```
Tank.find({ size: 'small' }).where('createdDate').gt(oneYearAgo).exec(callback);
```

Removing

Models have a static remove method available for removing all documents matching conditions.

```
Tank.remove({ size: 'large' }, function (err) {
  if (err) return handleError(err);
  // removed!
});
```

Applications

Let's take an example of Users model.
Create database and insert some data to into it. Using the following file (https://github.com/virtualSharif/gettingMEAN2/blob/master/mongodb/examples/example1/mongodata.txt)

Example 1: You can download the source code from github (https://github.com/virtualSharif/gettingMEAN2/tree/master/mongodb/examples/example1)

Step 1: Mongoose import
You need Mongoose to de ne a Mongoose schema, naturally, so enter the following line into app.js (model directory)

```
var mongoose = require('mongoose');
```

Step 2: Mongoose connection to MongoDB URI

```
mongoose.connect('mongodb://localhost:27017/usertestdb', {useMongoClient: true}, function (error) {
    if (error) { console.log(error);
   }
});
```

Step 3: Mongoose Schema definition
Mongoose gives you a constructor function for defining new schemas, which you typically assign to a variable so that you can access it later. It looks like the following line:

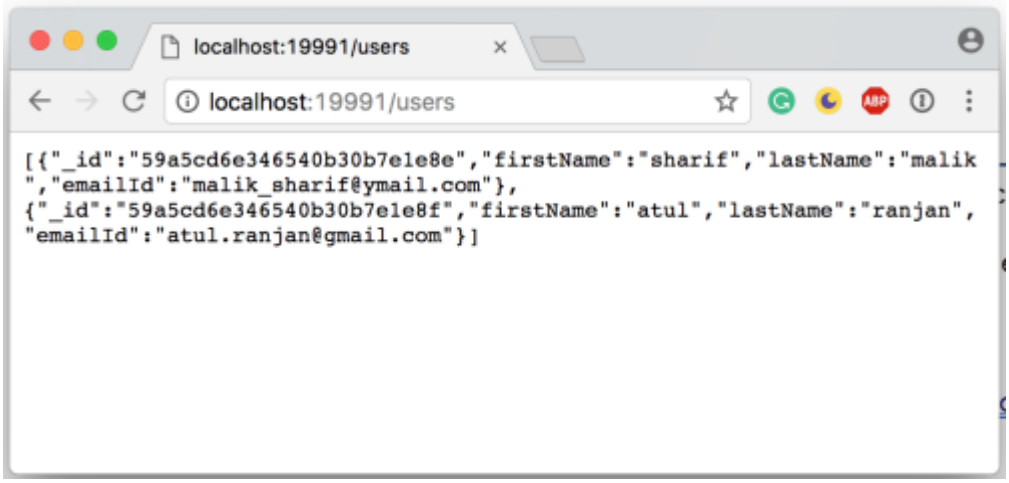
```
1.  var UserSchema = new mongoose.Schema({
2.                              firstName: String,
3.                              lastName: String,
4.                              emailId: String
5.                  });
```

Step 4: Mongoose Model definition: (Compiling a model from a schema)
Anything with the word "compiling" in it tends to sound a bit complicated. Compiling a Mongoose model from a schema is a simple one-line task. You just need to ensure that the schema is complete before you invoke the model command.

```
1.  var user = mongoose.model('user', UserSchema);
```

- Run the example :
  npm start or node app.js

- To see the output:
  Open the browser : http://localhost:19991/users



For fetching one user: http://localhost:19991/users/59a5cd6e346540b30b7e1e8e

Example 2:
Here you can have some enhancement to the example1.
Even the code is modularized.

You can download it from(https://github.com/virtualSharif/gettingMEAN2/tree/master/mongodb/examples/example2)
There are 4 files:

- **app.js** – main file which is require to run.
- **config.js** – which consists database properties eg. url, database name and so on.
- **userModel.js** – which consists the schema for the user model which will be used to relate with the mongodb.
- **mongodata.txt** – some db statements which can be used initially to setup database.

To run the application:
node app.js

To check the different routes, use POSTMAN chrome extension.
(for help checkout the express documentation).

References

https://docs.mongodb.com/manual/

https://docs.mongodb.com/getting-started/shell/

http://mongodb.github.io/node-mongodb-native/2.2/quick-start/quick-start/

https://university.mongodb.com/courses/M001/about

http://mongoosejs.com/docs/index.html