Extra Notes - Data Stores & APIs - using MongoDB and native driver

- Dr Nick Hayward

Contents

Data stores and APIs - using MongoDB and native driver - intro

As we build out our Node.js apps, we can start to add persistence to the data with the addition of data stores.

This primarily includes, but is not limited to, NoSQL data stores such as MongoDB. We can then use modules such as Mongoose to help connect, query, and structure how our apps work with MongoDB.

We can also create our own custom REST APIs for Node.js based web apps, offering various routes for GET, POST, PUT, DELETE, AND UPDATE.

data stores - install mongodb

Further information on MongoDB is available at the following URL,

- https://www.mongodb.com

Direct downloads are available from this site for Windows, Linux, and OS X. Download the latest *MongoDB Community Edition* executable for your OS, and click install.

Further details on installing MongoDB can be found at the following URL,

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

including OS specific help docs.

On OS X, it's also possible to install using Homebrew,

```
brew update
brew install mongodb
```

Unless specifically set, MongoDB will set a data directory for the installed DBs at `/data/db/`.

It will also use this directory to store `*.data` files for the current MongoDB install.

data stores - running mongodb

We can start the daemon for MongoDB using the following terminal command,

```
mongod
```

This command will output some debug information, and the final line will usually detail *network* information, including the port number waiting for connections. This means the daemon is running OK, and is now ready for connections.

With the daemon running, we can then connect to MongoDB using the following terminal command,

```
mongo
```

With the DB running, and a client connected, we can then start to query, update, monitor, maintain &c. our installed MongoDB databases.

n.b. it's also possible to specify a custom `data` directory for local databases, e.g.

```
mongod --dbpath ~/dev-data/mongo-data
```

This would start the MongoDB daemon and specify a working data directory in the home directory.

data stores - using MongoDB

We can now add a DB, some initial content, query for results, and so on.

For example, if we wanted to add some initial data, we can use the following command,

```
db.NodeTodo.insert({text: 'our first todo item'})
```

We can then query the `NoteTodo` DB for all current data,

```
db.NodeTodo.find()
```

This query will return the following type of output, e.g.

```
{ "_id" : ObjectId("5970f5384aadb18eed551dc5"), "text" : "our first todo item..." }
```

This is an object, which includes a unique ID for this record, and the *name:value* pair we just inserted.

data stores - Robo 3T MongoDB GUI

We can also install a GUI, *Robo 3T* (formerly Robomongo), for managing MongoDB,

- https://robomongo.org/

Download and install the applicable package for your current OS, including Windows, Linux, and OS X,

- https://robomongo.org/download

data stores - basic intro to NoSQL

So, what is NoSQL? In particular, it's often useful to consider NoSQL against a familiar SQL database structure. e.g.

SQL Structure	NoSQL Structure
database	database
table - users	collection - e.g. a library
row / records - user	document - e.g. a book
column - userID, name	field - e.g. author, title

data stores - connect to MongoDB from Node.js

We can test using MongoDB with Node.js by creating an initial API.

To connect to MongoDB from Node.js, there are various options available including a driver provided by the developers of MongoDB, *node mongodb native*. URL is as follows,

- https://mongodb.github.io/node-mongodb-native/

We'll create our api, e.g. `node-todo-api/`, and setup an initial Node.js app,

```
npm init
```

We'll then add a separate directory for ongoing MongoDB tests, `mongo-tests`, at the root of the `node-todo-api` directory.

We can also install the Mongo Client itself using NPM,

```
npm install mongodb --save
```

We can then add a test connect script to this `mongo-tests` directory, e.g. `mongodb-connect.js`

```
/ mongodb client
const client = require('mongodb').MongoClient;

// client connect - parameters = url and callback function
client.connect('mongodb://localhost:27017/NodeTodo', (err, db) => { // handle error
and db connection
  // handle return errors for connection
  if (err) {
    return console.log("connection to mongodb failed..."); // return - causes exit
from function...
  }
  console.log("now connected to mongodb...");

  // close connection to mongodb
  db.close()
});
```

So, we've successfully connected to the local running MongoDB daemon, specified a DB name, and then closed the connection. However, MongoDB will not create a new DB unless we specify some data to write. i.e. Mongo will not create an empty DB, which we can then populate later. (this is in contrast to other DBs such as SQL...).

If we want to save some content to a new DB, we'll need to add a collection, and specify the content to save, e.g.

```
...
db.collection('TodoItems').insertOne ({
  text: 'a simple todo item...',
  completed: false
}, (err, result) => {
  if (err) {
    return console.log('error returned for insert', err);
  }
  console.log('todo successfully saved', JSON.stringify(result.ops, undefined, 2));
// log output to console
});
...
```

The first time we call this updated code, we can now connect to MongoDB, create a new DB, add a collection, and then save a document with some JSON data for a single todo item.

`insertOne` will add a single document to the specified collection. It accepts two arguments, including an object for the single document, and a callback function with a return error or success response. The callback function will be executed for either of the error or success response, and we can add some code to execute to handle either scenario. In this example, we can log to the console a returned error, and a simple message for the success response with the document saved.

data stores - MongoDB ObjectId

One of the interesting aspects of working with documents in MongoDB is its use of *ObjectId* per document. e.g.

```
[
  {
    "name": "winifred",
    "location": "devon",
    "_id": "5973b06480ba9c0bacce7314"
  }
]
```

For this new document, MongoDB has automatically created an `_id` value. This is not an auto-incremented value, such as SQL.

So, the `_id` value is random, and not reliant on knowing the previous incremented value. This means we can quickly and easily scale DBs and servers without having to reference or check other existing DBs and servers. In effect, implementations of MongoDB can be separate nodes of an overall, scaled system.

The ObjectId constitute a few separate parts, contained within a 12 byte value. For example, these include

- a timestamp for the moment the `_id` was created
  - the first 4 bytes within the `_id`
- machine identifier for the host of the DB where the `_id` was created
  - next 3 bytes
- process ID - another way to create a unique identifier
  - next 2 bytes in the overall `_id` value
- a counter value for the item
  - takes up the final 3 bytes within the `_id` value

However, this structure is the default for the `_id`. This is not fixed, and a developer may set this `_id` as required.

As a document is inserted into a collection in a DB, we can also specify our own custom `_id`. It could be as simple as `day123` &c.

For the generated ObjectId, we can log to the console the generated ObjectId, e.g.

```
...
console.log(result.ops[0]._id); // use first inserted document - retrieve the
generated _id
...
```

We can then extract the *timestamp* from the first 4 bytes. e.g.

```
console.log(result.ops[0]._id.getTimestamp()); // use first inserted document - get
time stamp from first 4 bytes...
```

So, this will return a timestamp in GMT, e.g. `2017-07-23T14:37:05.000Z`.

ES6 - destructuring objects

Another new feature of ES6 is a shorthand way to extract a *value* from an object based upon its *name*, and then save to a variable. e.g.

```
var authors = {
  name: 'emma',
  books: 7,
  location: 'uk'
};

var {name} = authors;
console.log(name);
```

So, we can quickly access an object value, and save it to a matching variable. This new variable can be used as expected within the app's logic.

We can also use destructuring to get any properties from a specified object, e.g.

```
const {MongoClient, ObjectID} = require('mongodb');
```

So, the variables are now being set relative to the specified properties for the MongoDB module.

data stores - fetch data from MongoDB

Fetching data, such as a note or single todo item or simply all documents.

e.g. we might fetch all documents from a specified collection,

```
db.collection('Todos').find()
```

The `find()` method on the collection `Todos` returns a *cursor*, a pointer to all of the available documents. A *cursor* has many methods available, and we use them to fetch all or any of the available documents. It also returns a *promise*, which we can then use. e.g.

```
db.collection(`Todos`).find().toArray().then((docs) => {
  // log docs to console
  console.log(JSON.stringify(docs, undefined, 2));
}, (err) => {
  console.log('error in fetch documents', err);
});
```

So, an example method on `find()` is `toArray()`. This returns the document data from the collection as an array instead of the cursor.

```
[
  {
    "_id": "5973abfa23a0560b7351bbe0",
    "text": "walk the Inca trail...",
    "completed": false
  },
  {
    "_id": "5975dde85ac520dc59d77448",
    "text": "sail around the world...",
    "completed": false
  },
  {
    "_id": "5975e6b15ac520dc59d77525",
    "text": "visit the Grand Canyon...",

    "completed": true
  }
]
```

So, all documents have now been returned. We might then want to only return documents that have been completed or not.

```
// connect to collection - find documents by query - e.g. completed: false
db.collection('Todos').find({completed: false}).toArray().then((docs) => {
  ...
});
```

We're now querying the Todos collection for all documents with a *name:value* pair matching `completed: false`. However, if we want to query by a specific ObjectId, we have to create it first instead of simply copying and pasting the existing string. e.g. we create the `_id` as follows,

```
_id: new ObjectID('5975e6b15ac520dc59d77525')
```

So, we still need to know the existing string value, but it needs to be structured as an ObjectID value that MongoDB will recognise and use to search the specified collection. We can then pass this `_id` to the `find()` method and query MongoDB.

Lots of other methods are available for the cursor in MongoDB,

- mongodb.github.io/node-mongodb-native/2.2/api/Cursor.html

Some popular methods include `count()`, which can be used as follows,

```
// connect to collection - use count method on cursor
db.collection('Todos').find().count().then((count) => {
  // log count total to console
  console.log(`Total todos = ${count}`);
}, (err) => {
  console.log('error in count of documents', err);
});
```

So, we get a return total for the number of documents in the specified collection.

data stores - delete data in MongoDB

We can start by querying MongoDB, and then deleting any duplicate documents. e.g.

```
db.collection('Todos').deleteMany({text: 'sail around the world...'})
```

As MongoDB is returning a promise object, we can also add a `then()` method with a success and fail callback, e.g.

```
db.collection('Todos').deleteMany({text: 'sail around the world...'}).then((result)
=> {
  console.log(result); // result object returned for deletion
}, (err) => {
  console.log('error in deletion...', err);
});
```

We can log to the console the return `result` object. This helps us check the deletion and return from MongoDB. However, this will also return a lot of data for the single execution. The main point of interest is,

```
CommandResult {
result: {n: 4, ok: 1},
...
}
```

This basically tells us the number of documents deleted, and that the request was successful.

Likewise, we can delete a single document as follows, e.g.

```
db.collection('Todos').deleteOne({text: 'visit Antarctica'}).then((result) => {
  console.log(result); // result object returned for deletion
}, (err) => {
  console.log('error in deletion...', err);
});
```

This will delete the first document that matches the passed *name:value* pair.

Another popular method is `findOneAndDelete()`, which returns the data of the document and then deletes it from the collection, e.g.

```
db.collection('Todos').findOneAndDelete({completed: false}).then((result) => {
  console.log(result);
});
```

This will find the first document that matches the passed *name:value* pair, return the data, and then delete that first matched document. The return `result` object will return data for the document just deleted, e.g.

```
{ lastErrorObject: { n: 1 },
  value:
   { _id: 5973abfa23a0560b7351bbe0,
     text: 'walk the Inca trail...',
     completed: false },
  ok: 1 }
```

The `value{}` object can then be used to inform a user, for example, of the document that has just been deleted &c.

data stores - update data in MongoDB

We can now update data in a specified collection in MongoDB, including a single document to update.

We can start with the collection method, `findOneAndUpdate()`, which expects multiple parameters. e.g.

```
findOneAndUpdate(filter, update, options, callback)
```

and then returns a promise for the updated document. So, we might use this method as follows,

```
db.collection('Todos').findOneAndUpdate({
  _id: new ObjectID("5975e6b15ac520dc59d77525")
}, {
  // use update operator - items to actually update for matched document
  $set: {
    completed:true
  }
}, {
  // specify options - e.g. whether to return original document or not...
  returnOriginal: false
}).then((result) => {
  console.log(result);
});
```

This will now update the specified document, and return the updated data. e.g.

```
{ lastErrorObject: { updatedExisting: true, n: 1 },
  value:
   { _id: 5975e6b15ac520dc59d77525,

     text: 'visit the Grand Canyon...',
     completed: true },
  ok: 1 }
```

We might choose to return just the updated data, e.g.

```
console.log(`updated document: ${JSON.stringify(result.value, undefined, 2)}`);
```

We might also add a further property to update the document, such as incrementing a given value. e.g.

```
db.collection('Todos').findOneAndUpdate({
  _id: new ObjectID("597731be14ac1af4c0f16fe3")
}, {
  // use update operator - items to actually update for matched document
  $set: {
    completed:true
  },
  // use increment operator - specify value to increment by, e.g. -1, 1, 2 &c.
  $inc: {
    members: 1
  }
}, {
  // specify options - e.g. whether to return original document or not...
  returnOriginal: false
}).then((result) => {
  console.log(result);
});
```