NodeJS and MongoDB

Introduction

NodeJS works very well with MongoDB. As you saw last week, MongoDB uses a JavaScript-like syntax to query the database. In NodeJS, we use very similar code, as real JavaScript, to interact with a MongoDB database.

Using MongoDB from NodeJS

Retrieving data from the database

Here is an example to display all songs in the 'hits' collection:

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect('mongodb://localhost:27017/test',
    (err,db) =>
    {
        if (!err)
        {
            var hits = db.collection('hits');
            hits.find().toArray(
                (err,results) =>
                {
                    for(var i=0; i<results.length; i++)
                    {
                        console.log(results[i].song + '
                    }
                    db.close();
                });
        }
        else
        {
            console.log("ERROR: " + err);
        }
    }
);
```

How is this working?

- Firstly we need to obtain a MongoClient object. This is done by importing the 'mongodb' module (this needs to be installed using npm, though it's available on Neptune) and then using its ***MongoClient*** property.

- We then connect to the database. Note the connection URL:

```
mongodb://localhost:27017/test
```

  This means we are connecting to the ***test*** database on the local machine (localhost) on port 27017.
- The second parameter to ***connect()*** is a callback function which runs when the connection is made. Note the use of asynchronous development again: connecting to the database might take time, so we have to specify a callback to run when we get a connection, to avoid the application "hanging" waiting for a response.
- Similar to the ***fs*** module, the callback takes two parameters, an error and an object representing the database. So we check whether the error object exists, if it does we report the error while if it doesn't we continue.
- The next thing we do is to obtain an object representing the collection 'hits':

```
var hits = db.collection('hits');
```

- Having got this object, we then perform the query by calling the ***find()*** method:

```
hits.find().toArray( callback );
```

  Again we have to use a callback here as performing the query might be a time-consuming process. The callback contains two parameters once again - an error object and an array representing the actual results themselves. Note how we loop through the results and display them.
- Finally we close the connection to the database. **Note that you do NOT do this if you are writing an HTTP server otherwise it disconnects your server from the database, meaning any subsequent queries do not connect.**

Exercise

Adapt the previous example to find all songs by the artist specified by the user on the command-line. ***Hint:*** you specify what to search in ***find()*** in exactly the same way as using the MongoDB client directly.

Doing an update

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect('mongodb://localhost:27017/test',
    (err,db) =>
    {
        if (!err)
        {
```

```
            var hits = db.collection('hits');
            hits.update({_id: 123}, { $set:{quantity:34
                (err,results) =>
                {
                    if(!err)
                    {
                        if(results.result.n==1)
                        {
                            console.log("updated");
                        }
                        else
                        {
                            console.log("not updated -
                        }
                    }
                    else
                    {
                        console.log(err);
                    }
                    db.close();
                });
        }
        else
        {
            console.log("ERROR: " + err);
        }
    }
);
```

The update works similarly to the search, however we no longer have to display the results so it's a bit simpler. We call the ***update()*** method of the collection which takes three parameters:

- The criteria to match the record (here, that the _id should be 123);
- The updates to be performed (here, to update the quantity to 345);
- The callback function to run once the update has been completed. As in the previous case, we can use the ***err*** parameter to check for errors connecting to the database (e.g. the database server has shut down unexpectedly).
- Note however this will not check whether the song with the specified _id exists; to do this, we need to check the ***results.result.n*** property of the ***results*** object. This gives us the number of records that were updated successfully. If this is 1, we know the song was updated. However, if this is 0, it means that no records were updated - because no song had the specified ID.

As in the standard MongoDB command-line client, you can have multiple search criteria and updates by supplying objects with more than one field to the update() method. For example:

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect('mongodb://localhost:27017/test',
    (err,db) =>
    {
        if (!err)
        {
            var hits = db.collection('hits');
            hits.update({song: "Wonderwall", artist:"Oa
                (err,results)=>
                {
                    if(!err)
                    {
                        if(results.result.n==1)
                        {
                            console.log("updated");
                        }
                        else
                        {
                            console.log("not updated -
                        }
                    }
                    else
                    {
                        console.log(err);
                    }
                    db.close();
                });
        }
        else
        {
            console.log("ERROR: " + err);
        }
    }
);
```

This example will update both the quantity and the price of Wonderwall by Oasis.

Adding records

Adding records is similar in concept to updating, eg:

```
var MongoClient = require('mongodb').MongoClient;
MongoClient.connect('mongodb://localhost:27017/test',
    (err,db) =>
    {
        if (!err)
        {
            var hits = db.collection('hits');
            hits.insert({_id: 12345, song:'I Am The Wa
                (err,results)=>
                {
                    if(!err)
                    {
                        console.log("added");
                    }
                    else
                    {
                        console.log(err);
                    }
                    db.close();
                });
        }
        else
        {
            console.log("ERROR: " + err);
        }
    }
);
```

Note that ***insert()*** only requires one parameter before the callback: the record to insert, rather than the two (the search criteria and data to update) that we need for ***update()***.

Exercises

1. Write a Node script which adds a new song to the database specified by the user on the command-line. The user should be able to specify the _id, song, artist, year, quantity and price. Note that ***you will need to use parseInt() to convert the ID, year and quantity from string to integer*** and ***parseFloat() to convert the price to a float*** as these values are stored as numbers in the database but command-line arguments are treated as strings.
2. Write a Node script which updates an existing song. The user should be able to specify the _id and the price and quantity to update to. Again the ID will have to be converted from string to integer.

Advanced exercise

1. Write a Node ***server*** which searches for all songs by a given artist, specified as a query string. Note that when using MongoDB in a Node server, you should ***not*** use ***db.close()***.
2. Modify your example to generate ***JSON***. Hint: use ***JSON.stringify()*** on the array which toArray() gives you back.
3. If you have time write a full AJAX HitTastic! client using Node and MongoDB, allowing the user to first search for songs by a given artist, and then buy them by clicking a "Buy" button beside each search result. The "buy" button should link to a new API endpoint which "buys" a song by reducing the quantity by one. Note that ***any numeric values will be passed to the "buy" API endpoint as a string*** and therefore ***you will need to use parseInt() or parseFloat() to convert them from string to integer or float***.

IMPORTANT! Before doing this you should take note that you will need to implement ***CORS*** on your Node server - see the AJAX notes. This is because your AJAX front end will be delivered from the Apache server but will talk to your Node back-end - so your AJAX is coming from a different server to the backend. You'll need to allow access from your Apache-delivered front-end to your Node server and this can be done by adding an ***Access-control-allow-origin*** line:

```
response.writeHeader(200, { "Access-control-allow-origi
                            "Content-type": (your
```