ADDITIONAL READING

Using MongoDB's Node.js Driver

Visit our website

WRITE CODE THAT USES MONGODB'S NODE.JS DRIVER DIRECTLY

- **Step 1: Create project structure and install driver**

  The first thing we have to do to be able to use Node.js to interface with MongoDB, is to install the MongoDB driver. Find out more about this driver **here**. To install the driver, type the following instructions into your command line interface:

1. **mkdir myHypProject**
   Create a directory for your project.

2. **cd myHypProject**

3. **npm init**
   Create the package.json file for your project

4. **npm install mongodb --save**
   Install the driver, mongodb using NPM.

5. **mkdir data**
   Create a directory called data where you will store all your database related modules.

6. **mongod --dbpath=/data**
   Install and start a mongod process.

- **Step 2: Connect to the database**

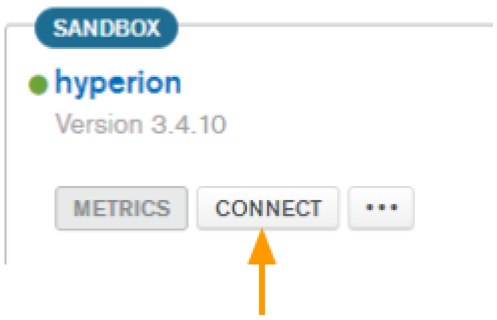
  We are now ready to start creating our db.js module. However, before we can do anything else, we need to connect to the database. The code used to connect to the database is shown below:

```
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');
```

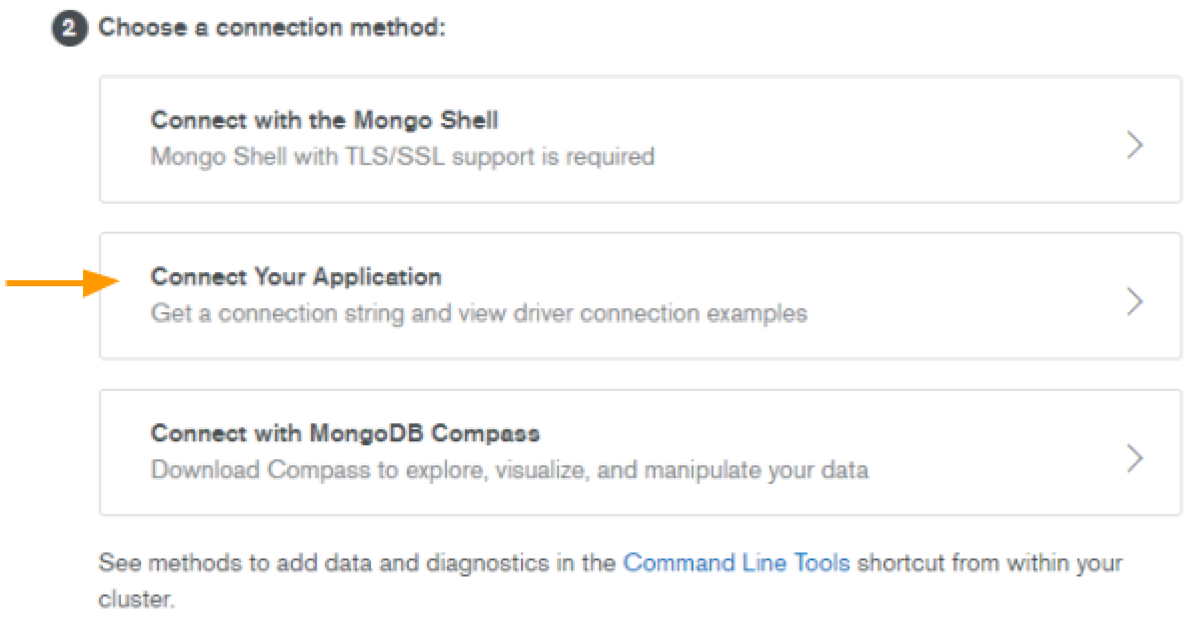
```
// Connection URL. Get this from Atlas. Click on Connect
const url =
'mongodb+srv://hyperionDB:mypassword@hyperion-78c.mongodb.net/test';

// Database Name
const dbName = 'myproject';

// Use connect method to connect to the server
MongoClient.connect(url, function(err, client) {
  assert.equal(null, err);
  console.log("Connected successfully to server");

  const db = client.db(dbName);

  client.close();
});
```

Remember, the connection URL used to connect to your database will be different from the one shown in the example. To find your connection string for your database on Atlas, use the same process you used to get your connection string for connecting using the mongo shell.

From your Atlas interface, choose 'connect' as shown below.



From the window that opens, choose to "Connect Your Application" as shown below. Select the appropriate options and copy the connection URL that Atlas provides you with. Remember to replace the placeholder for your password, with your actual password in your db.js file.


2 Choose a connection method:

Connect with the Mongo Shell
Mongo Shell with TLS/SSL support is required

Connect Your Application
Get a connection string and view driver connection examples

Connect with MongoDB Compass
Download Compass to explore, visualize, and manipulate your data

See methods to add data and diagnostics in the Command Line Tools shortcut from within your cluster.

Save the changes to your db.js file and then run it from your command line interface by typing: node /data/db.js. You should get a message saying, "Successfully connected to the database" on your command line interface.

You will learn more about encoding your password using "encodeURIComponent" in the "Testing and Refactoring Your Code" task.

- **Step 3: CRUD operations**

All the basic CRUD operations that we performed using the mongo shell, can also be coded using the MongoDB driver for Node.js. We give some code examples of functions that can be used for each of these operations. Since you are using JavaScript to create these functions, remember that you can modify the code to make the functions work however you like. The code examples for the rest of this task are from the official documentation for mongodb driver for Node.js **here. For more information, visit this site**.

Create:

An example of the code that can be used in order to insert a document into a collection is shown below.

```
const insertDocuments = function(db, callback) {
  // Get the collection named nameOfCollection
  const collection = db.collection('nameOfCollection');
  // Insert some documents
  collection.insertMany([
    {a : 1}, {a : 2}, {a : 3}
  ], function(err, result) {
    assert.equal(err, null);
    assert.equal(3, result.result.n);
    assert.equal(3, result.ops.length);
    console.log("Inserted 3 documents into the collection");
    callback(result);
  });
}
```

Points to note:

- This code uses a Node.js method that you may not be familiar with, assert.equal(). This method tests if the two arguments are equal. If not, the program terminates.

- The insertMany() method returns two objects, err and result.
    - The err object contains information about any errors that may have occurred as you attempted to insert documents into the database.
    - The result object contains information about how the insertMany() method affected the database. It includes information such as whether the insert was successful or not and how many documents were inserted.

- As with the mongo shell, you can also use the insertOne() method in Node.js.

- The MongoDB driver is asynchronous, so all instructions are executed without waiting for the previous instruction to finish. As you may imagine, this could cause problems, e.g. you could have unexpected results if you try to insert a document into a collection and you haven't yet finished connecting to the database. Notice how, in the code example above, callbacks are used to handle this.

Read:

We use the find() method to read files in a similar manner to how this was done when using the find command with the mongo shell. As with the find command in the mongo shell, if you use the find() method without specifying any arguments, all the documents in the collection are returned.

```
const findDocuments = function(db, callback) {
  // Get the documents collection
  const collection = db.collection('documents');
  // Find some documents
  collection.find({'a': 3}).toArray(function(err, docs) {
    assert.equal(err, null);
    console.log("Found the following records");
    console.log(docs);
    callback(docs);
  });
}
```

Points to note:

- The find() method also returns two objects, err and result.
  - The err object contains information about any errors that may have occurred when attempting to find documents in the database.
  - The result object (called 'docs' in the code example above) contains all the returned documents that can be converted to an array which contains each document as an object.

Update:

The updateOne() method is used to update documents in a collection, as shown in the example below:

```
const updateDocument = function(db, callback) {
  // Get the documents collection
  const collection = db.collection('documents');
  // Update document where a is 2, set b equal to 1
  collection.updateOne({ a : 2 }
    , { $set: { b : 1 } }, function(err, result) {
    assert.equal(err, null);
    assert.equal(1, result.result.n);
    console.log("Updated the document with the field a equal to 2");
    callback(result);
  });
}
```

Points to note:

- The updateOne() method again returns two objects, err and result.
    - The err object contains information about any errors that may have occurred as you attempted to update documents in the database.
    - The result object contains information about how the updateOne() method affected the database. It includes information such as whether the update was successful or not and how many documents were affected.

- As with the mongo shell, you can also use a method to updateMany() in Node.js.

<u>Delete:</u>

To delete documents from a collection, we use the deleteMany() or the deleteOne() method as shown in the example below:

```
 const removeDocument = function(db, callback) {
   // Get the documents collection
   const collection = db.collection('documents');
   // Delete document where a is 3
   collection.deleteOne({ a : 3 }, function(err, result) {
     assert.equal(err, null);
     assert.equal(1, result.result.n);
     console.log("Removed the document with the field a equal to 3");
     callback(result);
   });
 }
```