MongoDB JavaScript tutorial

In this tutorial, we show how to work with MongoDB in JavaScript. This tutorial uses the native mongodb driver. (There are also other solutions such as Mongoose or Monk.) This tutorial uses new features of JavaScript known as ES6.

MongoDB is a NoSQL cross-platform document-oriented database. It is one of the most popular databases available. MongoDB is developed by MongoDB Inc. and is published as free and open-source software.

A *record* in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB *documents* are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents. MongoDB stores documents in collections. *Collections* are analogous t tables in relational databases and documents to rows.

Installing MongoDB

The following command can be used to install MongoDB on a Debian-based Linux.

```
$ sudo apt-get install mongodb
```

The command installs the necessary packages that come with MongoDB.

```
$ sudo service mongodb status
mongodb start/running, process 975
```

With the `sudo service mongodb status` command we check the status of the `mongodb` server.

```
$ sudo service mongodb start
mongodb start/running, process 6448
```

The `mongodb` server is started with the `sudo service mongodb start` command.

Creating a database

The `mongo` tool is an interactive JavaScript shell interface to MongoDB, which provides an interface for systems administrators as well as a way for developers to test queries and operations directly with the database.

```
$ mongo testdb
MongoDB shell version: 2.4.9
```

```
connecting to: testdb
> db
testdb
> db.cars.insert({name: "Audi", price: 52642})
> db.cars.insert({name: "Mercedes", price: 57127})
> db.cars.insert({name: "Skoda", price: 9000})
> db.cars.insert({name: "Volvo", price: 29000})
> db.cars.insert({name: "Bentley", price: 350000})
> db.cars.insert({name: "Citroen", price: 21000})
> db.cars.insert({name: "Hummer", price: 41400})
> db.cars.insert({name: "Volkswagen", price: 21600})
```

We create a `testdb` database and insert eight documents in the `cars` collection.

Installing Node.js and MongoDB driver

Node.js is an open-source, cross-platform runtime environment for developing server-side Web applications. The runtime environment interprets JavaScript using Google's V8 JavaScript engine. We use Node.js to run our JavaScript applications.

```
$ curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
$ sudo apt-get install nodejs
```

We install Node.js.

```
$ npm install mongodb
```

We install the `mongodb` native JavaScript driver. The `npm` is a Node.js package manager. The MongoDB Node.js driver provides both callback based as well as Promised based interaction with MongoDB allowing applications to take full advantage of the new features in ES6.

Listing database collections

The `listCollections()` method lists available collections in a database.

list_collections.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;
var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    db.listCollections().toArray((err, collections) => {

        assert.equal(err, null);
        console.dir(collections);

        db.close();
    });
});
```

The example connects to the `testdb` database and retrieves all its collections.

```
var mongo = require('mongodb');
var assert = require('assert');
```

We use `mongodb` and `assert` modules.

```
var MongoClient = mongo.MongoClient;
```

`MongoClient` is used to connect to the MongoDB server.

```
var url = 'mongodb://localhost:27017/testdb';
```

This is the URL to the database. The 27017 is the default port on which the MongoDB server listens.

```
MongoClient.connect(url, (err, db) => {

...
});
```

A connection to the database is made with the `connect()` method.

```
db.listCollections().toArray((err, collections) => {

    assert.equal(err, null);

    console.dir(collections);

    db.close();
});
```

The `listCollection()` method finds all the collections in the `testdb` database; they are printed to the console. Generally, it is not recommended to close the database connection (long running applications using MongoDB) but since our applications are simple one-off instances, we do close the connections with the `close()` method.

```
$ node list_collections.js
[ { name: 'system.indexes' },
  { name: 'continents' },
  { name: 'cars' },
  { name: 'test' } ]
```

In our database, we have these four collections.

Database statistics

The `dbstats()` method gets statistics of a database.

dbstats.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;
var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    db.stats((err, stats) => {

        assert.equal(null, err);

        console.dir(stats);

        db.close();
    })
});
```

The example connects to the `testdb` database and shows its statistics.

```
$ node dbstats.js
{ db: 'testdb',
  collections: 5,
  objects: 30,
  avgObjSize: 41.86666666666667,
  dataSize: 1256,
  storageSize: 20480,
  numExtents: 5,
  indexes: 3,
  indexSize: 24528,
  fileSize: 201326592,
  nsSizeMB: 16,
  dataFileVersion: { major: 4, minor: 5 },
  ok: 1 }
```

This is the output of the `dbstats.js` example.

Reading data

The `find()` method creates a cursor for a query that can be used to iterate over results from MongoDB.

read_all.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    findCars(db, () => {

        db.close();
    });
});

var findCars = (db, callback) => {

   var cursor = db.collection('cars').find({});

   cursor.each((err, doc) => {

      assert.equal(err, null);

      if (doc != null) {

         console.dir(doc);

      } else {

         callback();
      }
   });
};
```

In the example, we iterate over all data of the `cars` collection.

```
var cursor = db.collection('cars').find({});
```

Passing an empty query returns all documents.

```
cursor.each((err, doc) => {

    assert.equal(err, null);

    if (doc != null) {

        console.dir(doc);

    } else {

        callback();
    }
});
```

We iterate through the documents of the collection using the `each()` method.

```
$ node read_all.js
{ _id: 1, name: 'Audi', price: 52642 }
{ _id: 2, name: 'Mercedes', price: 57127 }
{ _id: 3, name: 'Skoda', price: 9000 }
{ _id: 4, name: 'Volvo', price: 29000 }
{ _id: 5, name: 'Bentley', price: 350000 }
{ _id: 6, name: 'Citroen', price: 21000 }
{ _id: 7, name: 'Hummer', price: 41400 }
{ _id: 8, name: 'Volkswagen', price: 21600 }
```

This is the output of the `read_all.js` example.

Counting documents

The `count()` method returns the number of matching documents in the collection.

count_documents.js

```
var mongo = require('mongodb');
var assert = require('assert');
var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    db.collection('cars').find({}).count().then((n) => {

        console.log(`There are ${n} documents`);
        db.close();
    });
});
```

The example counts the number of documents in the `cars` collection.

```
db.collection('cars').find({}).count().then((n) => {

    console.log(`There are ${n} documents`);
    db.close();
});
```

We retrieve all documents from the `cars` collection and count them with `count()`. Note the usage of back ticks in the string interpolation.

```
$ node count_documents.js
There are 8 documents
```

There are eight documents in the cars collection now.

Reading one document

The `findOne()` method returns one document that satisfies the specified query criteria. If multiple documents satisfy the query, this method returns the first document according to the natural order whi reflects the order of documents on the disk.

read_one.js

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

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    var collection = db.collection('cars');

    var query = { name: 'Volkswagen' }

    collection.findOne((query), (err, doc) => {

        if (err) {

            console.log(err);
        } else {

            console.log(doc);
        }

        db.close();
    });
});
```

The example reads one document from the `cars` collection.

```
var query = { name: 'Volkswagen' }
```

The query contains the name of the car—Volkswagen.

```
collection.findOne((query), (err, doc) => {
```

The query is passed to the `findOne()` method.

```
$ node read_one.js
{ _id: 8, name: 'Volkswagen', price: 21600 }
```

This is the output of the example.

Promises

Promise is an object used for deferred and asynchronous computations. It represents an operation that has not completed yet, but is expected in the future.

```
asyncFunc()
.then(value => { /* success */ })
.catch(error => { /* failure */ });
```

The `then()` method always returns a Promise, which enables us to chain method calls.

promises.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    db.collection('cars').find({}).toArray().then((docs) => {

       docs.forEach((item, idx, array) => { console.log(item) });

       db.close();

   }).catch((err) => {

       console.log(err.stack);
    });
});
```

The example reads all documents from the `cars` collection; it utilizes a promise.

```
db.collection('cars').find({}).toArray().then((docs) => {
```

Note that we should be careful about using `toArray()` method because it can cause a lot of memory usag

Query operators

It is possible to filter data using MongoDB query operators such as `$gt`, `$lt`, or `$ne`.

read_gt.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    query = { price: { $gt : 30000 } };
    var cursor = db.collection('cars').find(query).toArray().then((docs) => {

        docs.forEach((item, idx, array) => { console.log(item)} );
        db.close();

    }).catch((err) => {

        console.log(err.stack);
    });
});
```

The example prints all documents whose car prices' are greater than 30,000.

```
query = { price: { $gt : 30000 } };
```

The `$gt` operator is used to get cars whose prices are greater than 30,000.

```
$ node read_gt.js
{ _id: 1, name: 'Audi', price: 52642 }
{ _id: 2, name: 'Mercedes', price: 57127 }
{ _id: 5, name: 'Bentley', price: 350000 }
{ _id: 7, name: 'Hummer', price: 41400 }
```

This is the output of the example. Only cars more expensive than 30,000 are included.

The `$and` logical operator can be used to combine multiple expressions.

read_gt_lt.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    findCars(db, () => {

        db.close();
    });
});

var findCars = (db, callback) => {

   query =  { $and: [ { price: { $gt: 20000 } }, { price: { $lt: 50000 } } ] };
   var cursor = db.collection('cars').find(query);

   cursor.each((err, doc) => {

      assert.equal(err, null);

      if (doc != null) {
          console.dir(doc);

      } else {

          callback();
      }
   });
};
```

In the example, we retrieve cars whose prices fall between 20,000 and 50,000.

```
query =  { $and: [ { price: { $gt: 20000 } }, { price: { $lt: 50000 } } ] };
```

The `$and` operator combines `$gt` and `$lt` to get the results.

```
$ node read_gt_lt.js
{ _id: 4, name: 'Volvo', price: 29000 }
{ _id: 6, name: 'Citroen', price: 21000 }
{ _id: 7, name: 'Hummer', price: 41400 }
{ _id: 8, name: 'Volkswagen', price: 21600 }
```

This is the output of the example.

Projections

Projections determine which fields are passed from the database.

projection.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    findCars(db, () => {

        db.close();
    });
});

var findCars = (db, callback) => {

   var cursor = db.collection('cars').find({}).project({_id: 0});

   cursor.each((err, doc) => {

      assert.equal(err, null);

      if (doc != null) {
            console.dir(doc);

        } else {

            callback();
        }
    });
};
```

The example excludes the `_id` field from the output.

```
var cursor = db.collection('cars').find({}).project({_id: 0});
```

The `project()` method sets a projection for the query; it excludes the `_id` field.

```
$ node projection.js
{ name: 'Audi', price: 52642 }
{ name: 'Mercedes', price: 57127 }
{ name: 'Skoda', price: 9000 }
{ name: 'Volvo', price: 29000 }
{ name: 'Bentley', price: 350000 }
{ name: 'Bentley', price: 350000 }
{ name: 'Citroen', price: 21000 }
{ name: 'Hummer', price: 41400 }
{ name: 'Volkswagen', price: 21600 }
```

This is the output for the example.

Limiting data output

The `limit()` method specifies the number of documents to be returned and the `skip()` method the numb
of documents to skip.

skip_limit.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    db.collection('cars').find({}).skip(2).limit(5).toArray().then((docs) => {

        docs.forEach((item, idx, array) => { console.log(item)} );

        db.close();

    }).catch((err) => {
        console.log(err.stack);
    });
});
```

The example reads from the `testdb.cars` collection, skips the first two documents, and limits the output five documents.

```
db.collection('cars').find({}).skip(2).limit(5).toArray().then((docs) => {
```

The `skip()` method skips the first two documents and the `limit()` method limits the output to five documents.

```
$ node skip_limit.js
{ _id: 3, name: 'Skoda', price: 9000 }
{ _id: 4, name: 'Volvo', price: 29000 }
{ _id: 5, name: 'Bentley', price: 350000 }
{ _id: 6, name: 'Citroen', price: 21000 }
{ _id: 7, name: 'Hummer', price: 41400 }
```

This is the output of the example.

Aggregations

Aggregations calculate aggregate values for the data in a collection.

sum_all_cars.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    sumCars(db, () => {

        db.close();
    });
});

var sumCars = (db, callback) => {

    var agr = [{$group: {_id: 1, all: { $sum: "$price" } }}];

    var cursor = db.collection('cars').aggregate(agr).toArray( (err, res) => {

       assert.equal(err, null);
       console.log(res);

       callback(res);
    });
};
```

The example calculates the prices of all cars in the collection.

```
var agr = [{$group: {_id: 1, all: { $sum: "$price" } }}];
```

The `$sum` operator calculates and returns the sum of numeric values. The `$group` operator groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, t each group.

```
var cursor = db.collection('cars').aggregate(agr).toArray( (err, res) => {
```

The `aggregate()` function applies the aggregation operation on the `cars` collection.

```
$ node sum_all_cars.js
[ { _id: 1, all: 619369 } ]
```

The sum of all prices is 619,369.

We can use the `$match` operator to select specific cars to aggregate.

sum_two_cars.js

```
var mongo = require('mongodb');
var assert = require('assert');

var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    sumCars(db, () => {

        db.close();
    });
});

var sumCars = (db, callback) => {

    var agr = [{ $match: {$or: [ { name: "Audi" }, { name: "Volvo" }] }},
               { $group: {_id: 1, sum2cars: { $sum: "$price" } }}];

    var cursor = db.collection('cars').aggregate(agr).toArray( (err, res) => {

       assert.equal(err, null);
       console.log(res);

       callback(res);
    });
};
```

The example calculates the sum of prices of Audi and Volvo cars.

```
var agr = [{ $match: {$or: [ { name: "Audi" }, { name: "Volvo" }] }},
            { $group: {_id: 1, sum2cars: { $sum: "$price" } }}];
```

The expression uses `$match`, `$or`, `$group`, and `$sum` operators to do the task.

```
$ node sum_two_cars.js
[ { _id: 1, sum2cars: 81642 } ]
```

The sum of the two cars' prices is 81,642.

Inserting a document

The `insert()` method inserts a single document into a collection.

insert_doc.js

```
var mongo = require('mongodb');
var assert = require('assert');
var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    insertDocument(db, () => {

        db.close();
    });
});

var insertDocument = (db, callback) => {

    var collection = db.collection('cars');

    var doc = {_id: 9, name: "Toyota", price: 37600 };

    collection.insert(doc, (err, result) => {

        assert.equal(err, null);
        assert.equal(1, result.result.n);
        console.log("A document was inserted into the collection");

        callback(result);
    });
}
```

The example inserts one car into the cars collection.

```
var doc = {_id: 9, name: "Toyota", price: 37600 };
```

This is a document to be inserted.

```
collection.insert(doc, (err, result) => {
```

The `insert()` method inserts the document into the collection.

```
> db.cars.find({_id: 9})
{ "_id" : 9, "name" : "Toyota", "price" : 37600 }
```

We confirm the insertion with the `mongo` tool.

Inserting multiple documents

The `insertMany()` method inserts multiple documents into a collection.

create_collection.js

```
var mongo = require('mongodb');
var assert = require('assert');
var MongoClient = mongo.MongoClient;

var url = 'mongodb://localhost:27017/testdb';
ObjectID = mongo.ObjectID;

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    insertDocuments(db, () => {

        db.close();
    });
});

var insertDocuments = (db, callback) => {

    var collection = db.collection('continents');

    var continents = [ {_id: new ObjectID(), name: "Africa"}, {_id: new ObjectID(), name: "America
        {_id: new ObjectID(), name: "Europe"}, {_id: new ObjectID(), name: "Asia"},
        {_id: new ObjectID(), name: "Australia"}, {_id: new ObjectID(), name: "Antarctica"} ]

    collection.insertMany(continents, (err, result) => {

        assert.equal(err, null);
        assert.equal(6, result.result.n);
        assert.equal(6, result.ops.length);
        console.log("6 documents inserted into the collection");

        callback(result);
    });
}
```

The example creates a `continents` collection and inserts six documents into it.

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

The `collection()` method retrieves a collection; if the collection does not exist, it is created.

```
var continents = [ {_id: new ObjectID(), name: "Africa"}, {_id: new ObjectID(), name: "America"},
var continents = [ {_id: new ObjectID(), name: "Africa"}, {_id: new ObjectID(), name: "America"},
    {_id: new ObjectID(), name: "Europe"}, {_id: new ObjectID(), name: "Asia"},
    {_id: new ObjectID(), name: "Australia"}, {_id: new ObjectID(), name: "Antarctica"} ]
```

This is an array of six records to be inserted into the new collection. The `ObjectID()` creates a new ObjectID, which is a unique value used to identify documents instead of integers.

```
collection.insertMany(continents, (err, result) => {

    assert.equal(err, null);
    assert.equal(6, result.result.n);
    assert.equal(6, result.ops.length);
    console.log("6 documents inserted into the collection");

    callback(result);
});
```

The `insertMany()` method inserts the array of documents into the `continents` collection.

```
> db.continents.find()
{ "_id" : ObjectId("5725df84a7f6376e0d6ae018"), "name" : "Africa" }
{ "_id" : ObjectId("5725df84a7f6376e0d6ae019"), "name" : "America" }
{ "_id" : ObjectId("5725df84a7f6376e0d6ae01a"), "name" : "Europe" }
{ "_id" : ObjectId("5725df84a7f6376e0d6ae01b"), "name" : "Asia" }
{ "_id" : ObjectId("5725df84a7f6376e0d6ae01c"), "name" : "Australia" }
{ "_id" : ObjectId("5725df84a7f6376e0d6ae01d"), "name" : "Antarctica" }
```

The `continents` collection has been successfully created.

Modifying documents

The `deleteOne()` method is used to delete a document and `updateOne()` to update a document.

modify.js

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

var url = 'mongodb://localhost:27017/testdb';

MongoClient.connect(url, (err, db) => {

    assert.equal(null, err);

    var collection = db.collection('cars');

    var query1 = { name: "Skoda" };

    collection.deleteOne(query1, (err, results) => {

        assert.equal(null, err);
    });

    var query2 = { name: "Audi" };
    collection.updateOne(query2, { $set: { "price": 52000 } }, (err, doc) => {

        assert.equal(null, err);
    });

    setTimeout( () => {

        console.log("Timeout");
        db.close();
    }, 5000);

});
```

The example deletes a document containing Skoda and updates the price of Audi.

```
var query1 = { name: "Skoda" };

collection.deleteOne(query1, (err, results) => {

    assert.equal(null, err);
});
```

The `deleteOne()` deletes the document of `Skoda`.

```
var query2 = { name: "Audi" };

collection.updateOne(query2, { $set: { "price": 52000 } }, (err, doc) => {

    assert.equal(null, err);
});
```

The price of Audi is changed to 52,000 with the `updateOne()` method. The `$set` operator is used to chang the price.

```
setTimeout( () => {

    console.log("Timeout");
    db.close();
}, 5000);
```

A timeout function is created with `setTimeout()`. We assume that five seconds is enough to execute the modifications. After that, we close the database connection.

```
> db.cars.find()
{ "_id" : 1, "name" : "Audi", "price" : 52000 }
{ "_id" : 2, "name" : "Mercedes", "price" : 57127 }
{ "_id" : 4, "name" : "Volvo", "price" : 29000 }
{ "_id" : 5, "name" : "Bentley", "price" : 350000 }
{ "_id" : 6, "name" : "Citroen", "price" : 21000 }
{ "_id" : 7, "name" : "Hummer", "price" : 41400 }
{ "_id" : 8, "name" : "Volkswagen", "price" : 21600 }
```

We confirm the changes with the `mongo` tool.

In this tutorial, we have worked with MongoDB and JavaScript.