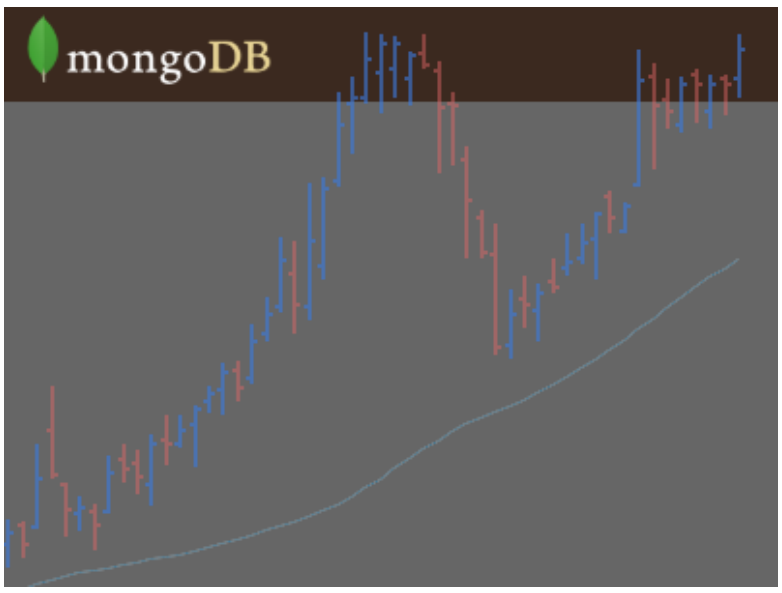
← Testing JavaScript With Jest | Node.js Blog Tutorial →

Node MongoDB Native



In this tutorial we'll take a look at creating a simple application to track stocks in a watchlist or portfolio. In order to store this data, we'll be using MongoDB – a popular NoSQL storage solution that is often used with Node.js. We'll see how to create the directory structure to hold this stock watchlist application, and then install MongoDB Native using npm to save it as a dependency to the package.json file. Once everything is in place, we'll see how to connect to Mongo and insert a new stock into our watchlist. We'll discuss a bit about the ObjectId and how it works in Mongo, and then finish up with fetching documents, counting documents, updating documents, and deleting them as well.

Create Directory and Package.json

First up, we'll just create a simple directory to hold all of our code and build a package.json file using npm init.

Setup Notifications For Subscribed Users

A Simple React.js Form Example

ES6 let vs var vs const

Mentions And Notifications

WordPress Toolbar Tutorial

Simple Styling Techniques For React Elements

What is WordPress?

Vue.js Tutorial

Install Twitter Bootstrap

What is the IoC Container in Laravel?

Linux Filesystem Hierarchy Explained

JavaScript Module Pattern

Laravel File Structure

```
node $mkdir stock-app
node $cd stock-app
stock-app $npm init
This utility will walk you through creating a package.json
file.
It only covers the most common items, and tries to guess
sensible defaults.

See npm help json for definitive documentation on these
fields
and exactly what they do.

Use npm install  afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (stock-app)
version: (1.0.0)
description: Simple Stock Watchlist
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\node\stock-app\package.json:

{
  "name": "stock-app",
  "version": "1.0.0",
  "description": "Simple Stock Watchlist",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
stock-app $
```

Installing MongoDB Native

We now have a package.json file, and we can install the **MongoDB Native** package like so.

```
stock-app $npm install mongodb --save
npm notice created a lockfile as package-lock.json. You
should commit this file.npm WARN stock-app@1.0.0 No
repository field.

+ mongodb@3.1.1
added 7 packages from 5 contributors and audited 7 packages
in 3.009s
found 0 vulnerabilities
```

When that completes, you'll have a fresh package.json file in the directory which will look something like this.

How To Add Methods To A Class In C#

Open Closed Principle

Mongoose Relationships Tutorial

How To Highlight New Content For Returning Visitors

What Is NodeJS?

Introduction To Laravel Controllers

How To Write Validation Test Cases

JSON in Laravel

JavaScript Revealing Prototype Pattern

PHP Code Structure

Escape Strings For MySQL To Avoid SQL Injection

Combine PHP Functions To Make Your Own

Developing With VueJS and PHP

Twitter Bootstrap Modal Tutorial

Build Your Own News Aggregator with Twitter Bootstrap and SimplePie

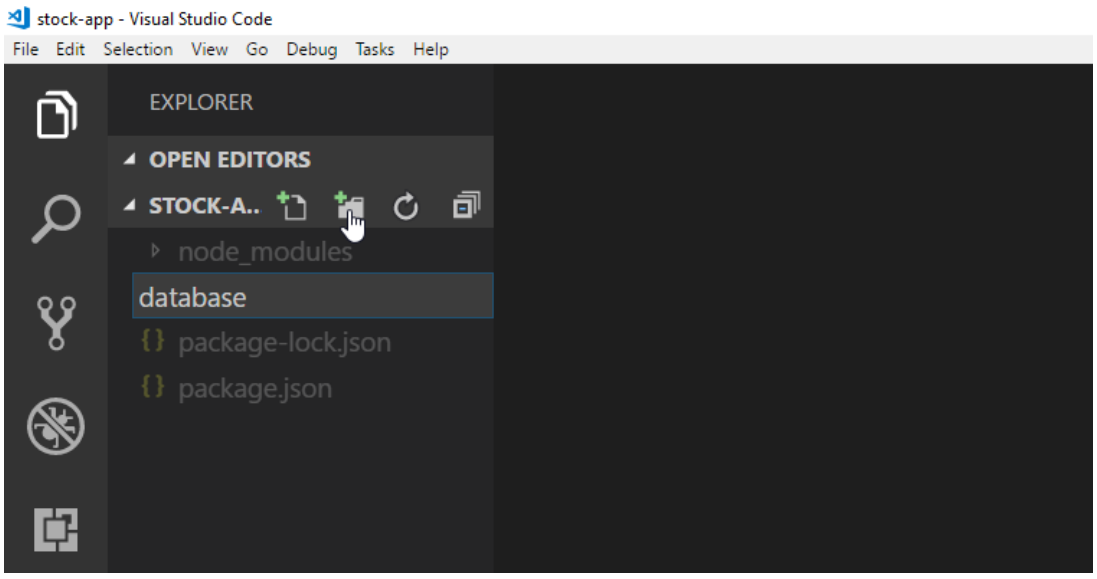
VegiThemes Twitter Bootstrap Themes

Laravel Form Class

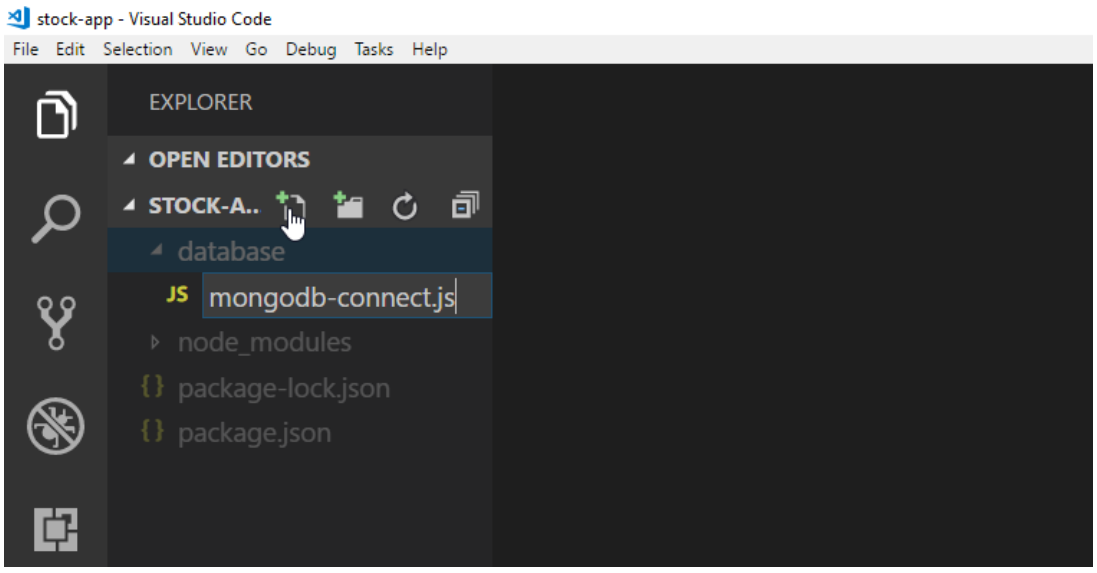
```
{
  "name": "stock-app",
  "version": "1.0.0",
  "description": "Simple Stock Watchlist",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mongodb": "^3.1.1"
  }
}
```

Connecting To MongoDB using MongoClient

We going to put our database specific files in a dedicated directory called `database`. I know, so original.



In this new directory, we can create a JavaScript file named **mongodb-connect.js**.



In this file, let's add the following JavaScript code.

WordPress Links and Images

Using Test Authentication To Allow Logged In Users To Post Replies

Douglas Crockford The Good Parts Examples

How To Use View Composers

ES6 Generators

jQuery Effects and Animation Methods

Angular Service Dependency Injection

Node MongoDB Native

What is a MySQL Join?

Introduction To The D3 JavaScript Library

VueJS Subnet Calculator

How To Use VueJS with Laravel Blade

Introduction To VirtualBox and Vagrant

C# Control Statements

ES6 Promises Tutorial

Most Popular JavaScript Frameworks

How To Use The Laravel Query Builder

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

MongoClient.connect('mongodb://localhost:27017/StockA
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

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

Before we run this snippet, do ensure you have **MongoDB installed and running** on your machine. As long as that checks out, we can cd into that database folder and just run the file.

```
stock-app $cd database
database $node mongodb-connect.js(node:14560)
DeprecationWarning: current URL string parser is
deprecated, and wi
ll be removed in a future version. To use the new parser,
pass option { useNewUr
lParser: true } to MongoClient.connect.
Connected to MongoDB
```

Interesting. It looks like we did connect, however we see that deprecation warning as well. A quick search on Stack Overflow indicates we can update the code like so – which does make the error clear.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

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

Insert Record Into Collection Using MongoDB Native

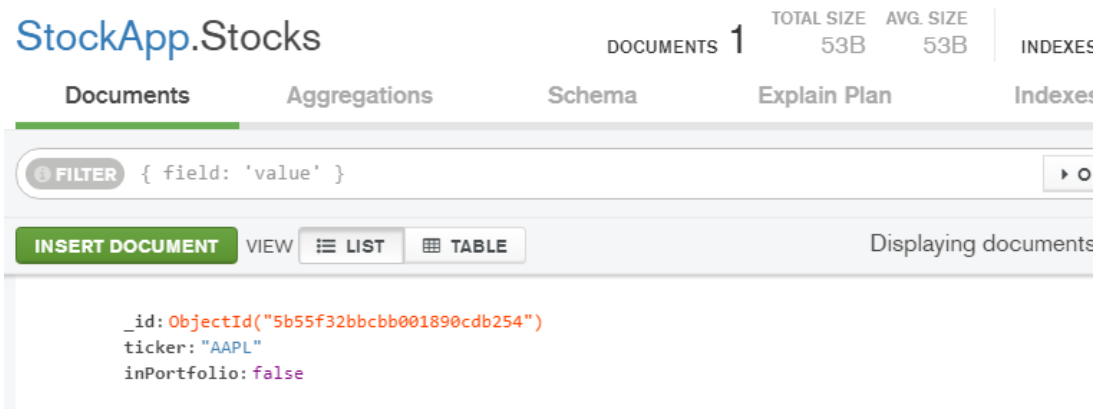
Let's now try to insert a document into the collection. We are representing adding a stock ticker to a watchlist here. The **insertOne()** function is used to complete this task.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').insertOne({
        ticker: 'AAPL',
        inPortfolio: false
    }, (err, result) => {
        if (err) {
            return console.log('Unable to insert stock', err);
        }

        console.log(JSON.stringify(result.ops, undefined, 2)
    });

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

Once again we can run this file and see the result. Since we are getting an instance of the new Stock, and not an error – it looks like all went well.

```
database $node mongodb-connect.js
Connected to MongoDB[  {
    "ticker": "AAPL",    "inPortfolio": false,
    "_id": "5b55f32bbcbb001890cdb254"
  }
]
```

We can also use **Mongo Compass** to check the results.



Viewing the new document using Compass is looking good as well.

The ObjectId

When we inserted a new document into Mongo, notice the _id field that we got back as part of our result. We did not specify that value, Mongo created it for us.

Understanding PHP Variable Scope

Twitter Bootstrap Navigation Elements

Adding Game Reviews With Eloquent Relationships

MySQL Group By Having Limit Offset and More!

The Ultimate PHP String Functions List

Record User Activity To The Database

Laravel hasMany and belongsTo Tutorial

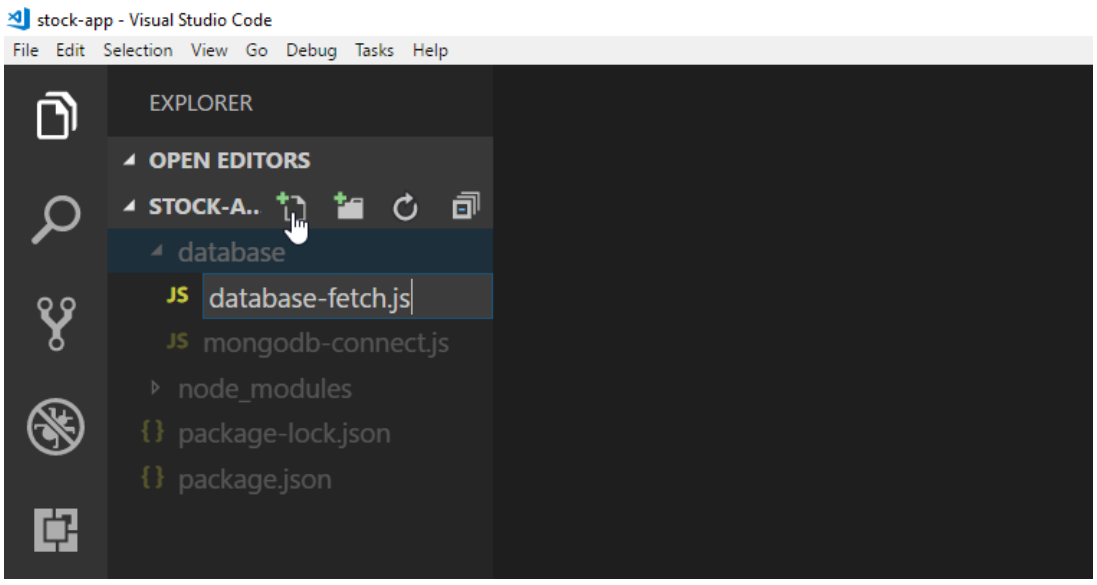
Create a Twitter Bootstrap Page

```
"_id": "5b55f32bbcbb001890cdb254"
```

As with most database systems, there is a unique identifier for any piece of data in the database. In Mongo, that is the ObjectId. The ObjectId is *not* an auto incrementing integer like you often see in SQL based systems. Mongo uses a randomly generated id. One reason for this is as a large system scales, there is no need to first check what that largest auto incrementing id is before inserting a new document. Mongo just generates a new random id, and away it goes. The ObjectId itself is a 12 byte value with the first 4 bytes being a timestamp. This is why there is no created_at field like you see in SQL based systems. The next 3 bytes are a machine identifier. Then there are 2 bytes to represent the process id. The last 3 bytes are a counter. All of these work together to create that random value. Learn more about **ObjectId** in the docs if you are curious.

Fetching Data Using MongoDB Native

Now we can start fetching some data from Mongo. We can add a new file in out database directory for this. We'll call it **database-fetch.js**.



We inserted a few more stocks into our database behind the scenes so we can demonstrate the **find().toArray()** function here.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').find().toArray().then((docs) =>
        console.log('Stocks');
        console.log(JSON.stringify(docs, undefined, 2));
    }, (err) => {
        console.log('Unable to fetch stocks', err);
    });

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

When we run the **database-fetch.js** file, we see the following array of results. It is an array of three objects, each having a unique _id, ticker, and inPortfolio properties.

```
database $node database-fetch.js
Connected to MongoDB
Stocks
[
  {
    "_id": "5b55f32bbcbb001890cdb254",
    "ticker": "AAPL",
    "inPortfolio": false
  },
  {
    "_id": "5b55f90ed81f3928c0423f41",
    "ticker": "MSFT",
    "inPortfolio": true
  },
  {
    "_id": "5b55f925330ad628406bcab3",
    "ticker": "NFLX",
    "inPortfolio": false
  },
  {
    "_id": "5b55f9575e8bc54238453153",
    "ticker": "MDB",
    "inPortfolio": true
  }
]
```

The above query might represent all stocks in a watchlist. However, maybe we want to see only those that are in your portfolio. You might be watching four stocks, but you have only actually purchased two for your portfolio. We can modify the query like so to be more granular.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').find({
        inPortfolio: true
    }).toArray().then((docs) => {
        console.log('Stocks');
        console.log(JSON.stringify(docs, undefined, 2));
    }, (err) => {
        console.log('Unable to fetch stocks', err);
    });

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

Now when we run the file, we can see that the query returns only stocks where the inPortfolio property is set to true. So it looks like we currently have Microsoft and Mongodb Inc in our portfolio.

```
database $node database-fetch.js
Connected to MongoDB
Stocks
[
  {
    "_id": "5b55f90ed81f3928c0423f41",
    "ticker": "MSFT",
    "inPortfolio": true
  },
  {
    "_id": "5b55f9575e8bc54238453153",
    "ticker": "MDB",
    "inPortfolio": true
  }
]
```

Counting Documents In The Collection

If you would like to query the number of documents in a collection, you can do so like we see below. In our case, this shows us all of the stocks on our watchlist whether we have purchased them or not.

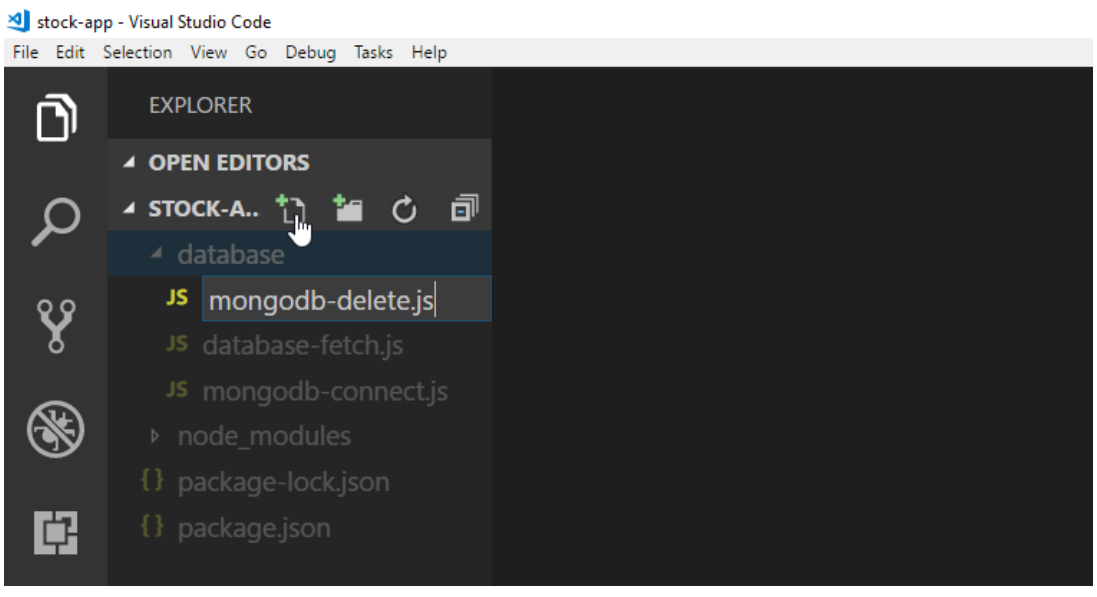
```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').find().count().then((count) =>
        console.log(`Stocks count: ${count}`);
    }, (err) => {
        console.log('Unable to fetch stocks', err);
    });

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

Here is the output from that query. Just what we would expect.

```
database $node database-fetch.js
Connected to MongoDB
Stocks count: 4
```

Deleting Documents Using MongoDB Native

Moving on to delete functionality, we can create a new file named **mongodb-delete.js** in our database directory.



Let's imagine we want to simply remove any stocks on our watchlist, that are not in our portfolio. We can use the **deleteMany()** function to do just that.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').deleteMany({
        inPortfolio: false
    }).then((result) => {
        console.log(result);
    });

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

We can run the JavaScript file and we get the result below. There is actually a huge chunk of output we omitted, since all we really care about is that n = 2, and ok = 1. This means two documents were effected(deleted), and the result went ok (it worked).

```
database $node mongodb-delete.js
Connected to MongoDB
CommandResult {
  result: { n: 2, ok: 1 }
```

What that means for us is that Apple and Netflix are no longer on our watchlist. Now we only have Microsoft and Mongo Inc.

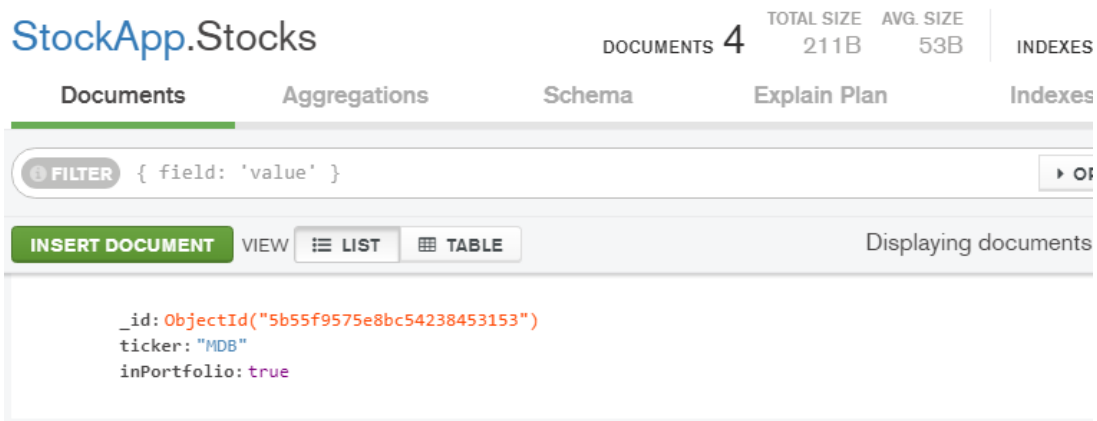
Perhaps we want to sell our Microsoft stock to remove it from both our watchlist and portfolio. We can do that with **deleteOne()**.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').deleteOne({
        ticker: 'MSFT'
    }).then((result) => {
        console.log(result);
    });

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

Running the file shows that 1 document was deleted and all went ok.

```
database $node mongodb-delete.js
Connected to MongoDB
CommandResult {
  result: { n: 1, ok: 1 }
```

Let's look at what we have in the database now using Compass. It looks like MDB stock is the last stock we own.



Turns out we need access to some funds, so we'll need to sell MDB in order to get our capital back. This time we'll use **findOneAndDelete()** like so.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').findOneAndDelete({
        ticker: 'MDB'
    }).then((result) => {
        console.log(result);
    });

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

When we run the program, we see that MDB was deleted and we now have no stocks in our watchlist or portfolio.

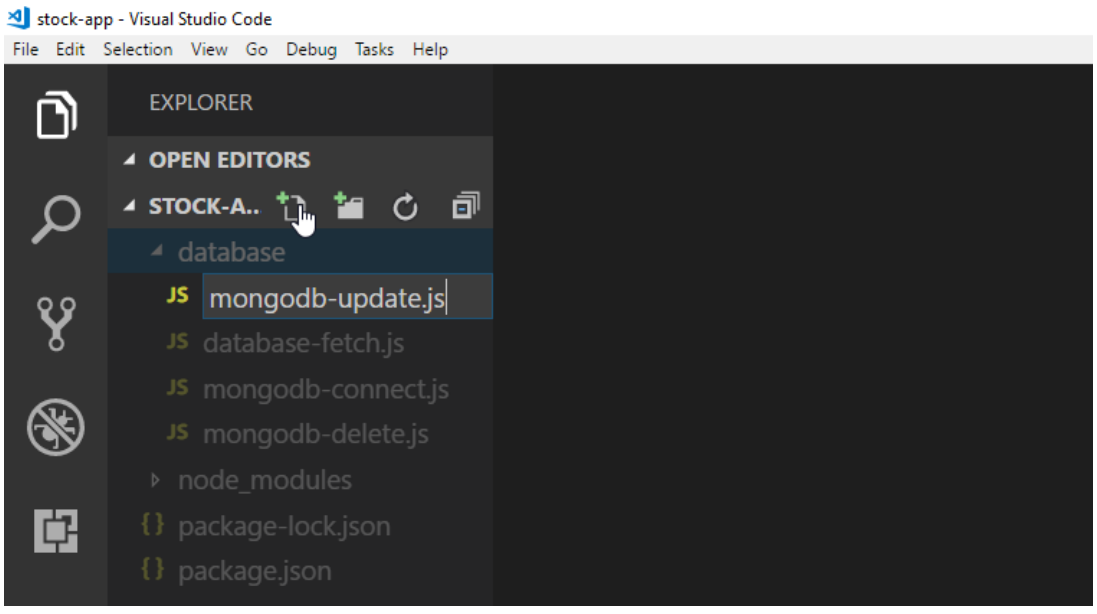
```
database $node mongodb-delete.js
Connected to MongoDB
{ lastErrorObject: { n: 1 },
  value:
   { _id: 5b55f9575e8bc54238453153,
     ticker: 'MDB',
     inPortfolio: true },
  ok: 1 }
```

Updating Documents Using MongoDB Native

Lastly, we can see how to update documents in MongoDB. You might see a problem however. We have no more stocks in our watchlist or portfolio. Let's add a bunch in one sweep using insertMany().

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').insertMany([{
        ticker: 'MDB',
        inPortfolio: true
    }, {
        ticker: 'NFLX',
        inPortfolio: true
    }, {
        ticker: 'AAPL',
        inPortfolio: true
    }, {
        ticker: 'MSFT',
        inPortfolio: true
    }, {
        ticker: 'IQ',
        inPortfolio: true
    }]).then((result) => {
        console.log(result);
    });

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

Great, all of those stocks are back in the watchlist and portfolio. Now we want to update the portfolio. Let's add a new file to the project named **mongodb-update.js**.



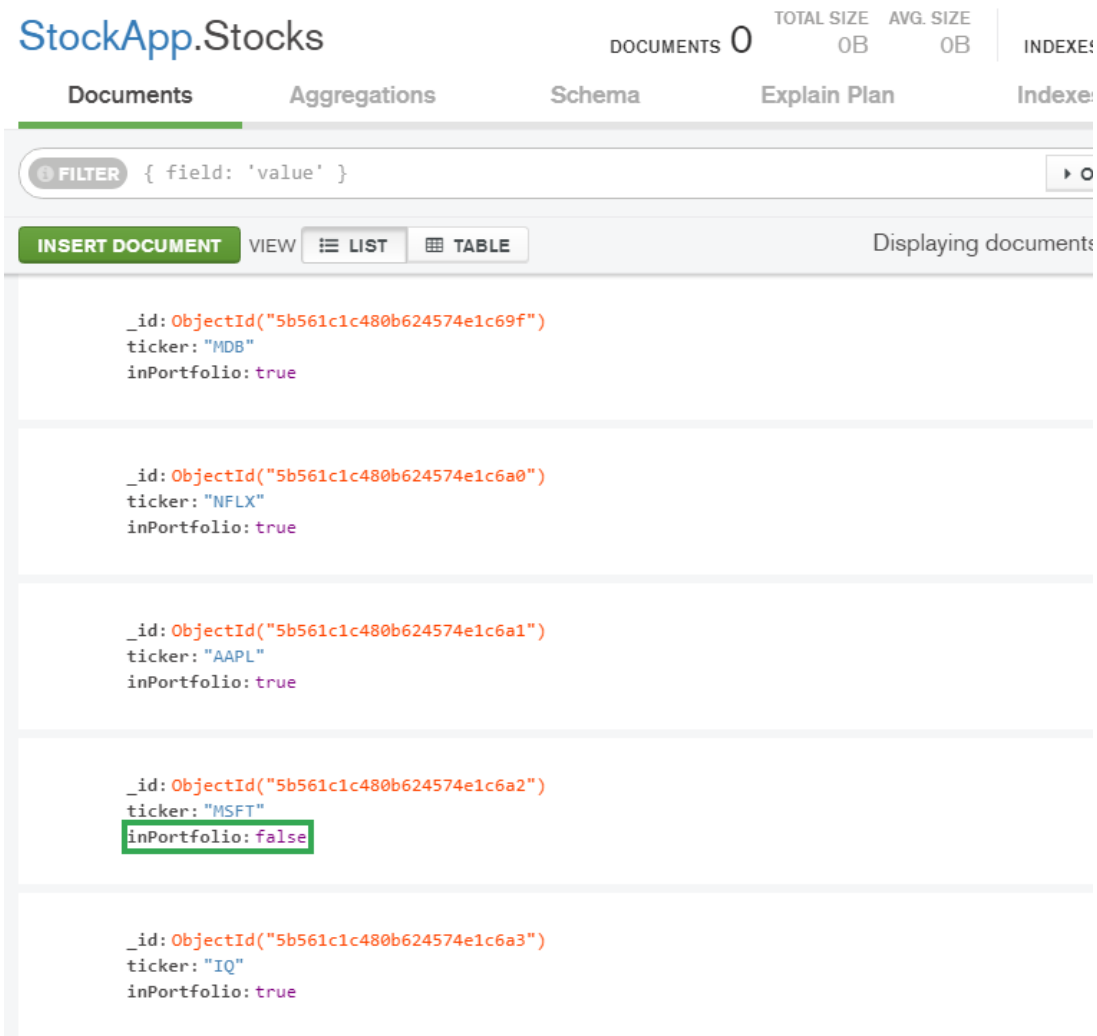
The function that we will use in this section is **findOneAndUpdate()**. In this scenario, we want to remove Microsoft from our portfolio, but keep it on our watchlist. So that means we need to find MSFT and set it's inPortfolio property to false.

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

MongoClient.connect('mongodb://localhost:27017/StockA
    useNewUrlParser: true
}, (err, client) => {
    if (err) {
        return console.log('Unable to connect to MongoDB'
    }
    console.log('Connected to MongoDB');

    const db = client.db('StockApp');
    db.collection('Stocks').findOneAndUpdate({
        ticker: 'MSFT'
    }, {
        $set: {
            inPortfolio: false
        }
    }, {
        returnOriginal: false
    }).then((result) => {
        console.log(result);
    });

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

Running the mongodb-update.js file gives us this response.

```
database $node mongodb-update.js
Connected to MongoDB
{ lastErrorObject: { n: 1, updatedExisting: true },
  value:
   { _id: 5b561c1c480b624574e1c6a2,
     ticker: 'MSFT',
     inPortfolio: false },
  ok: 1 }
```

This looks like it did the trick. Let’s also check in Compass.



Node MongoDB Native Summary

That about wraps it up with our tutorial on using MongoDB Native with Node.js. Many times we'll be using a dedicated ORM like **Mongoose** for this type of work, but as we've see in this tutorial – it can also be done with MongoDB Native just as efficiently.

#javascript #nodejs

← Testing JavaScript With Jest | Node.js Blog Tutorial →