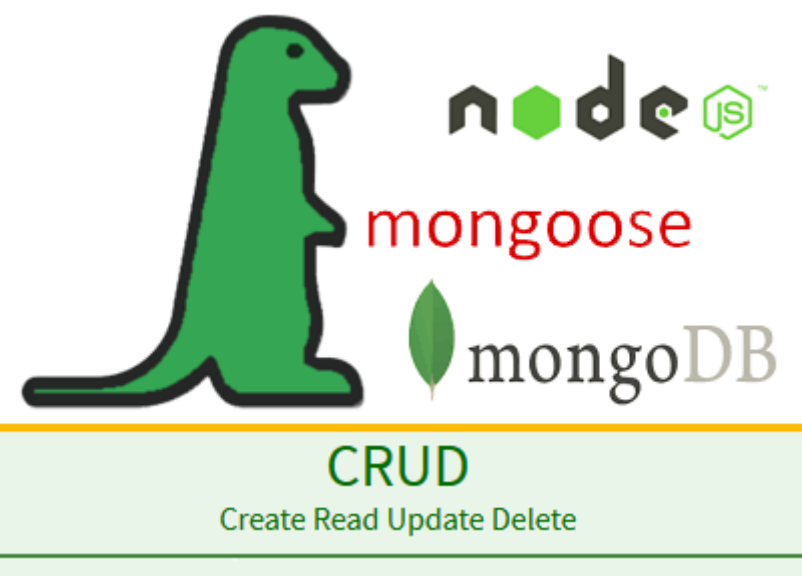


Mongoose Crud Tutorial



In this tutorial we are going to use Mongoose in our Node.js and

MongoDB environment to create, read, update, and delete documents from the database. We will cover connecting to MongoDB using Mongoose, explore MongoDB Schemas, and review Models in MongoDB. From there we'll move on to saving a document, as well as querying documents using comparison and logical expressions. We'll also see how to update documents using Mongoose both with the Query First and Update First approaches. Deleting documents from the database using Mongoose will be covered as well. Let's start working on CRUD with Mongoose now.

Connecting to MongoDB with Mongoose




We can start with a fresh project structure to get moving with MongoDB. Here we run `mkdir mongo-crud`, `cd mongo-crud`, and then `npm init --yes` to initialize a new project to work with.


```
node $mkdir mongo-crud
node $cd mongo-crud
mongo-crud $npm init --yes
Wrote to C:\node\mongo-crud\package.json:


{
  "name": "mongo-crud",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```


DEDICATED SERVER AGILE S


1 GBPS BANDWIDTH


   DELIVERED IN 1H

 1 CPU (4C/4T) @3 GHZ

 GEFORCE GT 710 1 GB

 1 TB SATA 3 ou 240 GB SSD

 8 GB RAM DDR4

 €29⁹⁹ / MTH .EX. VAT

DISCOVER

www.ikoula.com

Advertise Here



[How To Set Up Form Submission In Laravel](#)



[Most Useful JavaScript Array Functions](#)



[Vue.js Tutorial](#)



[How To Validate Form Submissions In Laravel](#)



[How To Highlight New Content For Returning Visitors](#)



[Building A Vue Front End For A Laravel API](#)



[Building A Forum With Laracasts](#)



[How To Pass Data To Views In Laravel](#)



[Using Operators in PHP](#)



[Adding Game Reviews With Eloquent Relationships](#)



[How To Test Your Laravel Application](#)

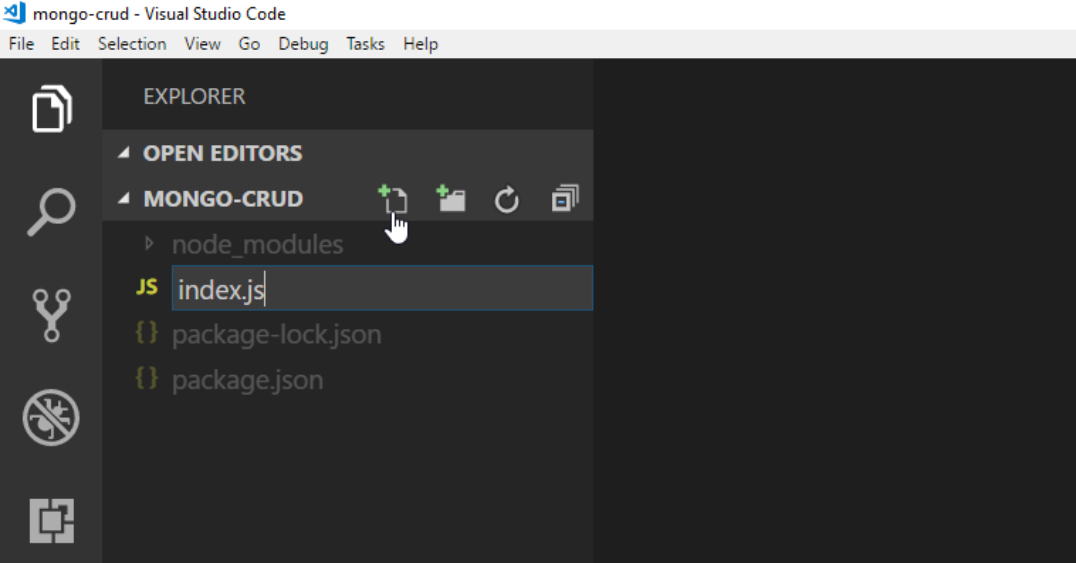
Let’s now install Mongoose using [npm package manager](#) by typing `npm i mongoose`. [Mongoose](#) is a MongoDB object modeling tool designed to work in an asynchronous environment.

```
mongo-crud $npm i mongoose
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN mongo-crud@1.0.0 No description
npm WARN mongo-crud@1.0.0 No repository field.

+ mongoose@5.1.4
added 20 packages from 15 contributors and audited 23 packages in 7.812s
found 0 vulnerabilities

mongo-crud $
```

Now we can go ahead and open the **mongo-crud** folder in Visual Studio Code or your favorite IDE program.



We are ready to **connect to MongoDB** from Node.js using this code.

index.js

```
1 const mongoose = require('mongoose');
2
3 mongoose.connect('mongodb://localhost/mongo-games
4   .then(() => 'You are now connected to Mongo!')
5   .catch(err => console.error('Something went wrong'
```

We launch the index.js using nodemon and we get a doozy!

```
mongo-crud $npm i mongoose
mongo-crud $nodemon index.js
[nodemon] 1.17.5
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Something went wrong { Error: write ECONNABORTED
  at _errnoException (util.js:1022:11)
  at Socket.writeGeneric (net.js:767:25)
  at Socket.write (net.js:786:8)
  at doWrite (_stream_writable.js:387:12)
  at writeOrBuffer (_stream_writable.js:373:5)
  at Socket.Writable.write (_stream_writable.js:290:11)
  at Socket.write (net.js:704:40)
  at Connection.write (C:\node\mongo-crud\node_modules\mongodb-core\lib\connection\connection.js:733:57)
  at C:\node\mongo-crud\node_modules\mongodb-core\lib\connection\pool.js:1571:44
  at waitForAuth (C:\node\mongo-crud\node_modules\mongodb-core\lib\connection\
```

[How To Delete A Record](#)



[From The Database](#)



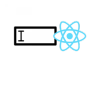
[27 Best Node.js Frameworks](#)



[Underscore JS sortBy Function](#)



[How To Fix The N+1 Problem](#)



[A Simple React.js Form Example](#)



[What Is An Assembly In C#?](#)



[Basic Laravel Routing and Views](#)



[Scheduling Commands and Tasks In Laravel](#)



[What Is NodeJS?](#)



[ES6 Arrow Function Tutorial](#)



[CSS Crash Course Tutorial For Beginners](#)



[Add Featured Image Support To Your WordPress Theme](#)



[How To Create A C# Unit Test In Visual Studio](#)



[Introduction to MySQL](#)



[You Might Still Need jQuery.](#)



[The Most Popular Content Manipulation Methods in jQuery.](#)



[Feature Test vs Unit Test And](#)

```

mongo-crud $nodemon index.js[nodemon] 1.17.5
[nodemon] to restart at any time, enter rs
[nodemon] watching: *.*
[nodemon] starting node index.js
Something went wrong { Error: write ECONNABORTED
  at _errnoException (util.js:1022:11)
  at Socket._writeGeneric (net.js:767:25)
  at Socket._write (net.js:786:8)
  at doWrite (_stream_writable.js:387:12)
  at writeOrBuffer (_stream_writable.js:373:5)
  at Socket.Writable.write (_stream_writable.js:290:11)
  at Socket.write (net.js:704:40)
  at Connection.write (C:\node\mongo-
crud\node_modules\mongodb-
core\lib\connection\connection.js:733:57)
  at C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\pool.js:1571:44
  at waitForAuth (C:\node\mongo-
crud\node_modules\mongodb-
core\lib\connection\pool.js:1425:40)
  at C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\pool.js:1433:5
  at C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\pool.js:1286:23
  at _combinedTickCallback
(internal/process/next_tick.js:131:7)
  at process._tickCallback
(internal/process/next_tick.js:180:9)
  name: 'MongoNetworkError',
  message: 'write ECONNABORTED',
  stack: 'Error: write ECONNABORTED\n    at
_errnoException (util.js:1022:11)\n    at
Socket._writeGeneric (net.js:767:25)\n    at
Socket._write (net.js:786:8)\n    at doWrite
(_stream_writable.js:387:12)\n    at writeOrBuffer
(_stream_writable.js:373:5)\n    at Socket.Writable.write
(_stream_writable.js:290:11)\n    at Socket.write
(net.js:704:40)\n    at Connection.write
(C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\connection.js:733:57)\n    at
C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\pool.js:1571:44\n    at
waitForAuth (C:\node\mongo-
crud\node_modules\mongodb-
core\lib\connection\pool.js:1425:40)\n    at
C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\pool.js:1433:5\n    at
C:\node\mongo-crud\node_modules\mongodb-
core\lib\connection\pool.js:1286:23\n    at
_combinedTickCallback
(internal/process/next_tick.js:131:7)\n    at
process._tickCallback
(internal/process/next_tick.js:180:9)' }
[nodemon] clean exit - waiting for changes before restart

```

We forgot to start the Mongo Daemon. Let's run **mongod** at the command prompt to launch MongoDB. Once complete, we can try to connect one more time and now things look much better.

Adding Replies To Threads



[VueJS Dynamic Components](#)



[ES6 let vs var vs const](#)



[7 Examples of the Every Function in Underscore JS](#)



[D3 DOM Manipulation And Data Binding With D3 JavaScript](#)



[How Does The Filter Function Work In Underscore JS?](#)



[Data Types in MySQL](#)



[D3 DOM Selection With D3 JavaScript](#)



[An Example of JavaScript Closure Using The Date Object](#)



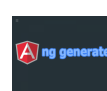
[Laravel API Resource Tutorial](#)



[VueJS Subnet Calculator](#)



[Flarum Forum Software](#)



[How To Create New Components In Angular Using The CLI](#)



[The Ultimate PHP String Functions List](#)



[How To Add Search To A WordPress Theme](#)



[WordPress Widgets Tutorial](#)


```
mongo-crud $node index.js
Now connected to MongoDB!
mongo-crud $nodemon index.js
[nodemon] 1.17.5
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Now connected to MongoDB!
```

Perfect! We’ve got Mongoose installed in a Node project, and we were able to connect to our [Mongo Database that we had installed](#) earlier.

Mongodb Schemas

To start working with inserting data into MongoDB, we need to create a [Schema](#). What is a Schema in the context of MongoDB? A Schema defines the shape of documents within a collection in MongoDB. A **Collection** in MongoDB is like a *table* in the relational database world. Within a collection, you can have one or many **Documents**. A Document in MongoDB is somewhat analogous to a row in an SQL based database. *Collections* have many *Documents* just like Tables have many Rows. Each document is a container of key/value pairs. Let’s see how to create a Schema in Mongoose.

```
1 const gameSchema = new mongoose.Schema({
2   title: String,
3   publisher: String,
4   tags: [String],
5   date: {
6     type: Date,
7     default: Date.now
8   },
9   onSale: Boolean,
10  price: Number
11 });
```

We want to insert a new game into our Mongo database, so we need to define the Schema of the data in Mongoose first. That is exactly what we did in the code above. Here is what the code does.

- `gameSchema` Defines the shape of game documents in MongoDB
- `new mongoose.Schema({})` Creates a new instance of the Mongoose Schema class where an object is passed in.
- `title: String` Each game needs a title, which is a string value.
- `publisher: String` Each game also has a publisher, also a string value.



[Linux](#)



[Laravel Form Class](#)



[Vegithemes Twitter Bootstrap Themes](#)



[How To Create User Registration in Laravel](#)



[Setting Up A Database With Seeding](#)



[The Ultimate Guide To Object Oriented PHP](#)



[Route Model Binding In Laravel](#)



[Angular Service Dependency Injection](#)



[Autoloading For Code Organization](#)



[The Top 9 Most Popular PHP String Functions](#)



[ES6 Object Literal Enhancements](#)



[JavaScript Code Structure](#)



[Laravel belongsToMany Example](#)



[What are MySQL Operators?](#)



[What are Getters and Setters?](#)



[Document Object Model Tutorial](#)

- **tags:** `[String]` We can assign multiple tags to a game, each being a string.
- **date:** `{ type: Date, default: Date.now }` We want to include a date value when inserting a game.
- **onSale:** `Boolean` Is the game currently on sale? Yes or no (true or false) represented by a Boolean.
- **price:** `Number` Finally, each game has a price – represented by a number.

When creating a Schema, the types which can be used are String, Number, Date, Buffer, Boolean, ObjectId, and Array.

Models In Mongodb

At this point, we have a clear definition of what a game should be when it is added to the Mongo database. In other words, we have a Schema that defines the shape of game documents to be added to a collection in MongoDB. This Schema must now be compiled down to a **Model**. You can think of a Model in MongoDB as an **Instance** of the given Schema. It's the same idea as creating a new Object from a Class in Object Oriented Programming. Here is how we create a Model.

```
1 const Game = mongoose.model('Game', gameSchema);
```

Mongoose provides the [model\(\) method which accepts two arguments](#). The first argument is the singular name of the collection your model is for. Mongoose automatically looks for the plural version of your model name by convention. Therefore, in the code above, the model 'Game' is for the 'games' collection in the database. The second argument to the model() method is the schema that defines the *shape of documents* in this **Collection**. This gives us a new `Game` class in our application. This is why in the line of code above, we are assigning the result of the call to `mongoose.model()` to that `Game` constant. It is capitalized because what is stored here is a Class, not an object.

Since this is a class, we can new up and object from this class. Check it out.

```
1 const game = new Game({
2   title: "The Legend of Zelda: Breath of the Wild",
3   publisher: "Nintendo",
4   tags: ["adventure", "action"],
5   onSale: false,
6   price: 59.99,
7 });
```



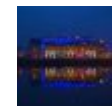
[The Top 11 Most Important HTML Text Features to](#)

[Learn](#)



[How To Remember Form](#)

[Data](#)



[Process Returned MySQL Query](#)

[Results In PHP](#)



[Getting Started With Python 3](#)



[Html Tutorials For Beginners](#)



[The Most Awesome PHP](#)

[Script to Highlight Your Code](#)



[The Top 15 Most Popular](#)

[JavaScript String Functions](#)



[CSS Grid Tutorial](#)



[HTML Hyperlinks Tutorial](#)



[CSS Position Tutorial](#)



[Vue Sibling Component](#)

[Communication](#)



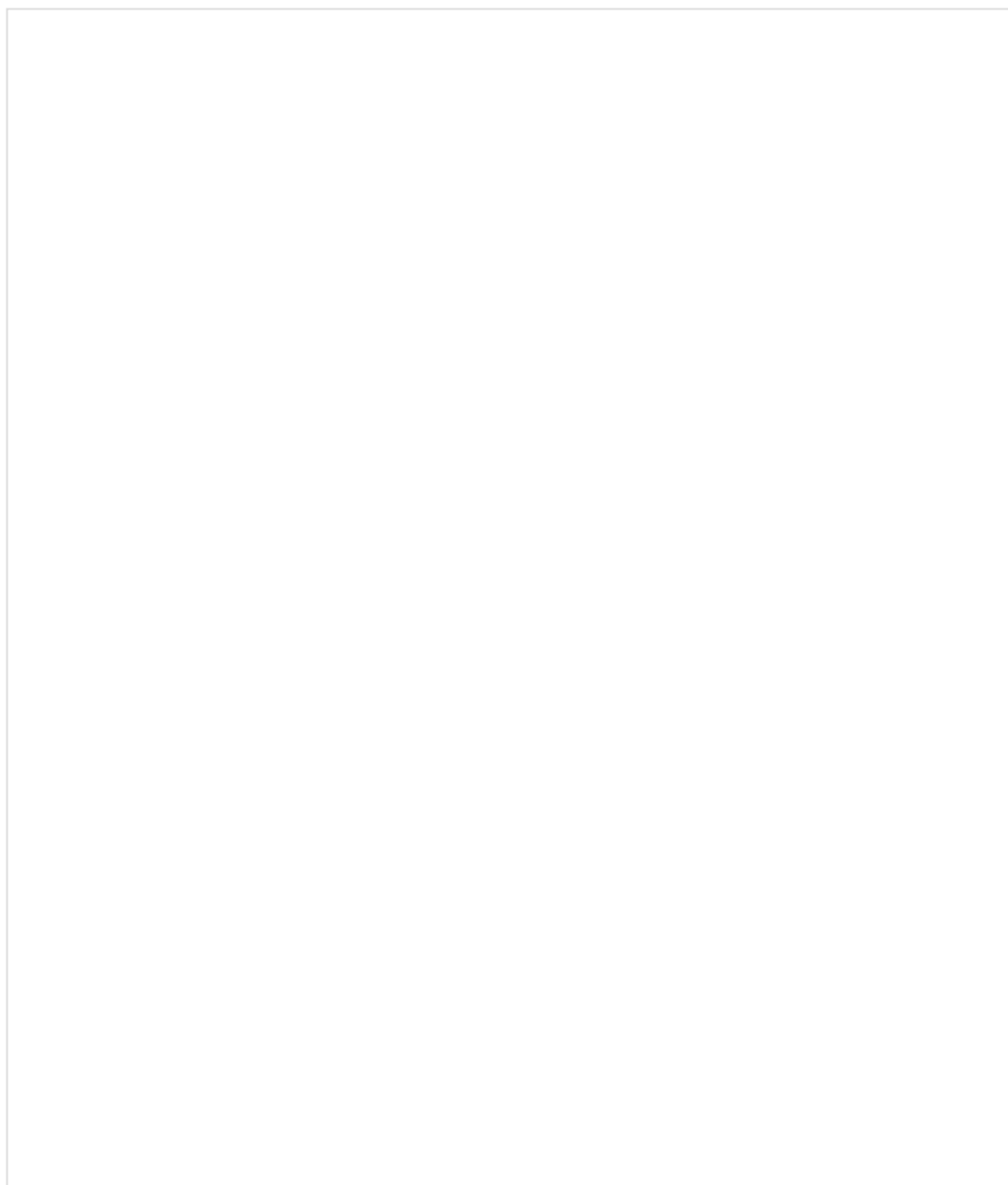
[What Are Scalable Vector Graphics?](#)

What we do above is to create a new instance of the Game class and pass an object during instantiation. We can simply set the properties of that object as we would like to. This is getting ready to save this document to the Mongo database. To review the steps involved so far.

- Define a Schema using `mongoose.Schema()`
- Use that result to create a model using `mongoose.model()`
- You now have a class you can create instances(objects) from
- These objects map to documents in the MongoDB database

Saving(Create) a Document In MongoDB

We have done all the legwork to save this document to the database. You might feel like all we have to do is something like `game.save();` and we're good. This is almost true. Since we are dealing with [Asynchronous JavaScript](#), we need to account for that and make use of either a callback, promise, or async await. In the code below, we use the async/await approach. Notice the highlighted code defines an async function of **saveGame()**. Inside that function we make use of the `await` keyword to indicate we are waiting on the result of a Promise. Finally, we call the `saveGame()` function at the very end of the code.



```
1 const mongoose = require('mongoose');
2
3 mongoose.connect('mongodb://localhost/mongo-game', {
4   .then(() => console.log('Now connected to MongoDB'))
5   .catch(err => console.error('Something went wrong'))
6
7 const gameSchema = new mongoose.Schema({
8   title: String,
9   publisher: String,
10  tags: [String],
11  date: {
12    type: Date,
13    default: Date.now
14  },
15  onSale: Boolean,
16  price: Number
17 });
18
19 const Game = mongoose.model('Game', gameSchema);
20
21 async function saveGame() {
22   const game = new Game({
23     title: "The Legend of Zelda: Breath of the Wild",
24     publisher: "Nintendo",
25     tags: ["adventure", "action"],
26     onSale: false,
27     price: 59.99,
28   });
29
30   const result = await game.save();
31   console.log(result);
32 }
33
34 saveGame();
```

Running the application at the command line shows us that the game was created!

```
mongo-crud $node index.js
Now connected to MongoDB!
{ tags: [ 'adventure', 'action' ],
  _id: 5b1588507bd7b004ac79206b,
  title: 'The Legend of Zelda: Breath of the Wild',
  publisher: 'Nintendo',
  onSale: false,
  price: 59.99,
  date: 2018-06-04T18:43:28.423Z,
  __v: 0 }
```

We can also now browse this new document in MongoDB using Compass.

The image shows a screenshot of the MongoDB Compass interface and a code editor. In the Compass interface, the 'mongo-games' database and 'games' collection are highlighted. A document is shown with the following fields: `_id` (ObjectId), `tags` (Array with 'adventure' and 'action'), `title` ('The Legend of Zelda: Breath of the Wild'), `publisher` ('Nintendo'), `onSale` (false), `price` (59.99), `date` (2018-06-04 14:43:28.423), and `__v` (0). The code editor shows the following JavaScript code:

```
mongoose.connect('mongodb://localhost:27017/mongo-games');

const Game = mongoose.model('Game', gameSchema);

const game = new Game({
  title: "The Legend of Zelda: Breath of the Wild",
  publisher: "Nintendo",
  tags: ["adventure", "action"],
  onSale: false,
  price: 59.99,
});
```

Did you notice that we never had to manually create a database, nor did we manually create any tables ahead of time like you must do with a relational database system. In MongoDB, you simply create a document, then store it in the database. It is a very flexible system. In the example above, by simply including 'mongo-games' in the connection string, a database was created for us automatically once we saved a document. Now, if you like, go ahead and change the values of the document to insert so that we can have many games to work with. Then we can practice queries on those documents.

Querying(Read) Documents in MongoDB

After adding a handful of games in the prior section, we can now read them from the database. There are many ways to query for data in MongoDB. We can start with the simple [find\(\)](#) method. In the below code we create a new async function named `getGames()`. In it, we want to fetch all the games in the database.


```
1  const mongoose = require('mongoose');
2
3  mongoose.connect('mongodb://localhost/mongo-game', {
4    .then(() => console.log('Now connected to MongoDB'))
5    .catch(err => console.error('Something went wrong'))
6  });
7  const gameSchema = new mongoose.Schema({
8    title: String,
9    publisher: String,
10   tags: [String],
11   date: {
12     type: Date,
13     default: Date.now
14   },
15   onSale: Boolean,
16   price: Number
17 });
18
19 const Game = mongoose.model('Game', gameSchema);
20
21 async function getGames() {
22   const games = await Game.find();
23   console.log(games);
24 }
25
26 getGames();
```

The code looks good so we run our node application. As you can see, we get a listing of all the games.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { tags: [ 'adventure', 'action' ],   _id:
5b1588507bd7b004ac79206b,
  title: 'The Legend of Zelda: Breath of the Wild',
  publisher: 'Nintendo',    onSale: false,
  price: 59.99,    date: 2018-06-04T18:43:28.423Z,
  __v: 0 },
{ tags: [ 'adventure', 'action' ],
  _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  publisher: 'Nintendo',
  onSale: true,
  price: 45,
  date: 2018-06-04T20:47:29.661Z,
  __v: 0 },
{ tags: [ 'racing', 'sports' ],
  _id: 5b15a5c610cf6b30dcdec47d,
  title: 'Mario Kart 8 Deluxe',
  publisher: 'Nintendo',
  onSale: false,
  price: 59.99,
  date: 2018-06-04T20:49:10.180Z,
  __v: 0 },
{ tags: [ 'action', 'shooter' ],
  _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  publisher: 'Nintendo',
  onSale: true,
  price: 35.99,
  date: 2018-06-04T20:51:17.812Z,
  __v: 0 },
{ tags: [ 'side scroller', 'platformer' ],
  _id: 5b15a6783863d22508a9703d,
  title: 'Rayman Legends',
  publisher: 'Ubisoft',
  onSale: false,
  price: 49.99,
  date: 2018-06-04T20:52:08.460Z,
  __v: 0 },
{ tags: [ 'simulation', 'farming' ],
  _id: 5b15a6ba75eabb4234ee9446,
  title: 'Stardew Valley',
  publisher: 'Chucklefish',
  onSale: false,
  price: 19.99,
  date: 2018-06-04T20:53:14.535Z,
  __v: 0 },
{ tags: [ 'adventure', 'platformer' ],
  _id: 5b15a6efa42bc928702bd749,
  title: 'Shovel Knight: Treasure Trove',
  publisher: 'Yacht Club Games',
  onSale: true,
  price: 10.99,
  date: 2018-06-04T20:54:07.257Z,
  __v: 0 } ]
```

Passing a filter to find()

Now we want to filter the results a bit. We can do so by passing an object. We can pass in an object with the publisher property set to Nintendo, and the onSale property set to true. We only want to see games that are published by Nintendo and also on sale. Here is the updated code to do that.

```
1 async function getGames() {
2   const games = await Game.find({
3     publisher: 'Nintendo',
4     onSale: true
5   });
6   console.log(games);
7 }
8
9 getGames();
```

Now, we can run at the command line again and we see Super Mario Odyssey and Splatoon 2 – both of which are on sale. Cool!

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { tags: [ 'adventure', 'action' ],
  _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  publisher: 'Nintendo',
  onSale: true,
  price: 45,
  date: 2018-06-04T20:47:29.661Z,
  __v: 0 },
  { tags: [ 'action', 'shooter' ],
  _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  publisher: 'Nintendo',
  onSale: true,
  price: 35.99,
  date: 2018-06-04T20:51:17.812Z,
  __v: 0 } ]
```

Adding a sort() to the query

Let's say you want to sort the results. In our case, let's sort the two games by price. We want the lowest price game first. We can pass an object to the sort() method with the price property set to 1. 1 indicates ascending order while -1 represents descending order.

```
1 async function getGames() {
2   const games = await Game
3     .find({ publisher: 'Nintendo', onSale: true })
4     .sort({ price: 1 });
5
6   console.log(games);
7 }
8
9 getGames();
```

This gives us the desired result as we can see Splatoon 2 has a cost of 35.99 and comes first in the list of results.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { tags: [ 'action', 'shooter' ],
  _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  publisher: 'Nintendo',
  onSale: true,
  price: 35.99,
  date: 2018-06-04T20:51:17.812Z,
  __v: 0 },
  { tags: [ 'adventure', 'action' ],
  _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  publisher: 'Nintendo',
  onSale: true,
  price: 45,
  date: 2018-06-04T20:47:29.661Z,
  __v: 0 } ]
```

Selecting only certain properties

We don't really need to see every single property of each document, let's just display the title of the game and the price. We can do this by adding a `select()` to the query. Again we pass an object and set the properties that we want to include to 1.

```
1 async function getGames() {
2   const games = await Game
3     .find({ publisher: 'Nintendo', onSale: true })
4     .sort({ price: 1 })
5     .select({ title: 1, price: 1 });
6
7   console.log(games);
8 }
9
10 getGames();
```

Now we see that we get the title and the price only in the query results. The id of the document is still included as this is how MongoDB works.


```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  price: 35.99 },
  { _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  price: 45 } ]
```

Comparison Query Operators

MongoDB has several comparison operators for querying.

- **\$eq** Matches values that are equal to a given value.
- **\$ne** Matches all values that are not equal to a given value.
- **\$gt** Matches values that are greater than a given value.
- **\$gte** Matches values that are greater than or equal to a given value.
- **\$lt** Matches values that are less than a given value.
- **\$lte** Matches values that are less than or equal to a given value.
- **\$in** Matches any of the values contained in an array.
- **\$nin** Matches none of the values contained in an array.

With that knowledge, let's find all games that are less than 25 dollars. We do this by passing an object to the price property within the find() function.

```
1 async function getGames() {
2   const games = await Game
3     .find({ price: { $lt: 25 } })
4     .sort({ price: 1 })
5     .select({ title: 1, price: 1 });
6
7   console.log(games);
8 }
9
10 getGames();
```

We can also check for several specific values in the database using the \$in operator. We want only games that are exactly 19.99, 35.99, or 59.99. Just pass an array to the \$in property like so.

```
1  async function getGames() {  
2    const games = await Game  
3      .find({ price: { $in: [19.99, 35.99, 59.99] } })  
4      .sort({ price: 1 })  
5      .select({ title: 1, price: 1 });  
6  
7    console.log(games);  
8  }  
9  
10 getGames();
```

That query looks to be working great!

```
mongo-crud $node index.js  
Now connected to MongoDB!  
[ { _id: 5b15a6ba75eabb4234ee9446,  
  title: 'Stardew Valley',  
  price: 19.99 },  
  { _id: 5b15a645a28b042f0822271f,  
    title: 'Splatoon 2',  
    price: 35.99 },  
  { _id: 5b1588507bd7b004ac79206b,  
    title: 'The Legend of Zelda: Breath of the Wild',  
    price: 59.99 },  
  { _id: 5b15a5c610cf6b30dcdec47d,  
    title: 'Mario Kart 8 Deluxe',  
    price: 59.99 } ]
```

This might look a little confusing because you would expect something like `price < 25` as you might see in the sql based world. In Mongo `price < 25` is replaced with something like `price: { $lt: 25 }`. Different syntax, same results. When running the program, we can see we have two games for less than 25 dollars.

```
mongo-crud $node index.js  
Now connected to MongoDB!  
[ { _id: 5b15a6efa42bc928702bd749,  
  title: 'Shovel Knight: Treasure Trove',  
  price: 10.99 },  
  { _id: 5b15a6ba75eabb4234ee9446,  
    title: 'Stardew Valley',  
    price: 19.99 } ]
```

Let's modify the query to now give us any games that are between 10 and 50 dollars. We do this by specifying two properties in the object passed to price.

```
1 async function getGames() {
2   const games = await Game
3     .find({ price: { $gt: 10, $lt: 50 } })
4     .sort({ price: 1 })
5     .select({ title: 1, price: 1 });
6
7   console.log(games);
8 }
9
10 getGames();
```

Now we have 5 games returned. All of which are more expensive than 10 dollars per game but less expensive than 50 dollars per game.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b15a6efa42bc928702bd749,
  title: 'Shovel Knight: Treasure Trove',
  price: 10.99 },
  { _id: 5b15a6ba75eabb4234ee9446,
  title: 'Stardew Valley',
  price: 19.99 },
  { _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  price: 35.99 },
  { _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  price: 45 },
  { _id: 5b15a6783863d22508a9703d,
  title: 'Rayman Legends',
  price: 49.99 } ]
```

Logical Query Operators

Let's now look at some logical operators. Imagine we want to see all the games published by Nintendo, OR, all the games that are on sale. How can we do that? We use the `.or()` function in the query chain like so.

```
1 async function getGames() {
2   const games = await Game
3     .find()
4     .or([{ publisher: 'Nintendo' }, { onSale: true }])
5     .sort({ price: 1 })
6     .select({ publisher: 1, title: 1, onSale: 1 });
7
8   console.log(games);
9 }
10
11 getGames();
```

We get the result we are looking for. All listed games are either published by Nintendo, or they are on sale. As long as one of those conditions is met for each game, then it is included in the results.

```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b15a6efa42bc928702bd749,
  title: 'Shovel Knight: Treasure Trove',
  publisher: 'Yacht Club Games',
  onSale: true },
  { _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  publisher: 'Nintendo',
  onSale: true },
  { _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  publisher: 'Nintendo',
  onSale: true },
  { _id: 5b1588507bd7b004ac79206b,
  title: 'The Legend of Zelda: Breath of the Wild',
  publisher: 'Nintendo',
  onSale: false },
  { _id: 5b15a5c610cf6b30dcdec47d,
  title: 'Mario Kart 8 Deluxe',
  publisher: 'Nintendo',
  onSale: false } ]
```

Let's now add another condition to the query. We want all games published by Nintendo, OR, all the games that are on sale, AND, they must be less than 50 dollars.

```
1  async function getGames() {
2    const games = await Game
3      .find()
4      .or([ { publisher: 'Nintendo' }, { onSale: true } ])
5      .and([ { price: { $lt: 50 } } ])
6      .sort({ price: 1 })
7      .select({ publisher: 1, title: 1, onSale: 1, price: 1 })
8
9    console.log(games);
10 }
11
12 getGames();
```

We can see that we have refined the query successfully. We get less games because any game that is more than 50 dollars is now excluded.


```
mongo-crud $node index.js
Now connected to MongoDB!
[ { _id: 5b15a6efa42bc928702bd749,
  title: 'Shovel Knight: Treasure Trove',
  publisher: 'Yacht Club Games',
  onSale: true,
  price: 10.99 },
  { _id: 5b15a645a28b042f0822271f,
  title: 'Splatoon 2',
  publisher: 'Nintendo',
  onSale: true,
  price: 35.99 },
  { _id: 5b15a561d34346325cd8e564,
  title: 'Super Mario Odyssey',
  publisher: 'Nintendo',
  onSale: true,
  price: 45 } ]
```

Updating(Update) in MongoDB

Now we get to updating documents in MongoDB using Mongoose. There are a couple of ways to do this and we'll look at both.

Updating Documents using Query First

In the query first approach, you find the document by it's id first using **.findById()**. Then you modify any properties you like, and lastly make a call to the **.save()** function. You are likely quite familiar with this approach as it's pretty much the same in all languages and frameworks. Here is how we can do that.

```
1 async function updateGame(id) {
2   const game = await Game.findById(id);
3   if (!game) return
4
5   game.price = 24;
6
7   const result = await game.save();
8   console.log(result);
9 }
```

In the above code, we find the game by it's unique id. If something doesn't work, or we don't have the right id, the function just returns. Otherwise, we update the price and set it to 24 dollars. Finally we save the game and output the result. So we track down the id of our Zelda game and find it is 5b1588507bd7b004ac79206b. Let's make a call to the function like so.

```
1 updateGame('5b1588507bd7b004ac79206b');
```

When we run the application we now see the game has a price of 24 dollars. Excellent!

```
mongo-crud $node index.js
Now connected to MongoDB!
{ tags: [ 'adventure', 'action' ],
  _id: 5b1588507bd7b004ac79206b,
  title: 'The Legend of Zelda: Breath of the Wild',
  publisher: 'Nintendo',
  onSale: false,
  price: 24,
  date: 2018-06-04T18:43:28.423Z,
  __v: 0 }
```

Updating a Document using Update First

There is also an update first approach when using Mongoose. In this approach, you simply dive into the database and update the document directly. You might also return the updated document when complete. This approach makes use of the **.update()** method.

```
1 async function updateGame(id) {
2   const result = await Game.update({ _id: id }, {
3     $set: {
4       title: 'A Link Between Worlds',
5       price: 55
6     }
7   });
8   console.log(result);
9 }
10
11 updateGame('5b1588507bd7b004ac79206b');
```

When we run the program, we are shown that one document was updated.

```
mongo-crud $node index.js
Now connected to MongoDB!
{ n: 1, nModified: 1, ok: 1 }
```

When we use Compass to check out the data of this document, we can see it worked. It now has a new title as well as a new price.

mongo-games.games

Documents Schema Explain Plan Indexes

FILTER { field: 'value' } **OPTIONS**

INSERT DOCUMENT VIEW **LIST** **TABLE** Displaying documents 1 - 7 of 7

```
_id: ObjectId("5b1588507bd7b004ac79206b")
> tags: Array
  title: "A Link Between Worlds"
  publisher: "Nintendo"
  onSale: false
  price: 55
  date: 2018-06-04 14:43:28.423
  __v: 0
```

If you do want to see the document after it was updated, you can use `.findByIdAndUpdate()` like we see here.

```
1 async function updateGame(id) {
2   const game = await Game.findByIdAndUpdate({ _id: id },
3     $set: {
4       title: 'Ocarina of Time ',
5       price: 35
6     }
7   }, { new: true });
8   console.log(game);
9 }
10
11 updateGame('5b1588507bd7b004ac79206b');
```

```
mongo-crud $node index.js
Now connected to MongoDB!
{ tags: [ 'adventure', 'action' ],
  _id: 5b1588507bd7b004ac79206b,
  title: 'Ocarina of Time ',
  publisher: 'Nintendo',
  onSale: false,
  price: 35,
  date: 2018-06-04T18:43:28.423Z,
  __v: 0 }
```

Remove(Delete) Documents in MongoDB

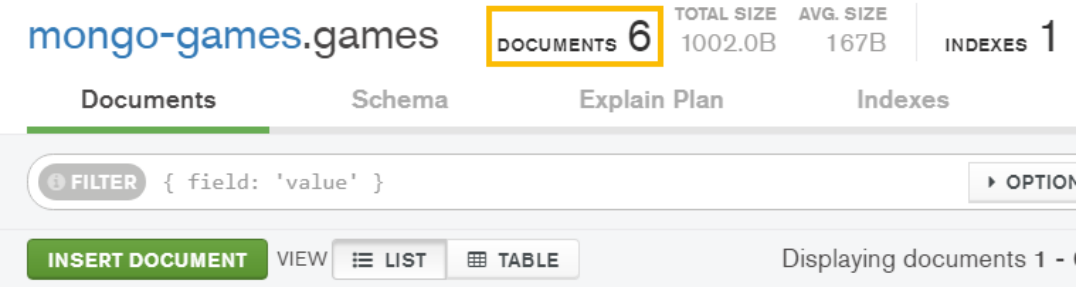
Finally! We get to see how to delete a document from MongoDB using Mongoose. We'll use the **.deleteOne()** method to do so.

```
1 async function deleteGame(id) {
2   const result = await Game.deleteOne({ _id: id })
3   console.log(result);
4 }
5
6 deleteGame('5b1588507bd7b004ac79206b');
```

When running the application, it shows us one document was removed.

```
mongo-crud $node index.js
Now connected to MongoDB!
{ n: 1, ok: 1 }
```

If we look in Compass, now there are only 6 documents when we had 7 to start. The remove was successful.



Mongoose Crud Tutorial Summary

To recap, here are some key points to remember when working with Mongoose to perform crud operation in MongoDB.

- Before you can store objects in MongoDB, you need to first define a **Mongoose schema**.
- The schema defines the *shape of documents* in MongoDB.
- A SchemaType object is used to provide details about the data.
- The types you can use are: **String**, **Number**, **Date**, **Buffer**, **Boolean** and **ObjectId**.
- Once a mongoose schema is defined, it must be compiled into a **Model**.
- A Model is similar to a Class. It's a blueprint for creating objects.
- Useful methods are **.save()**, **.findById()**, **.set()**, **.update()**, **.findByIdAndUpdate()**, **.deleteOne()**, **.deleteMany()**, and **.findByIdAndRemove()**.

#nodejs