MongoDB

MongoDB data types.

Null

The null type is used to represent a null and a field that does not exist. For example:

```
{

"isbn": null

}
```

Code language: JSON / JSON with Comments (json)

Boolean

The boolean type has two values true and false. For example:

```
{

"best_seller": true

}
```

Code language: JSON / JSON with Comments (json)

Number

By default, the mongo shell uses the 64-bit floating-point numbers. For example:

```
{

"price": 9.95,

"pages": 851

}
```

Code language: JSON / JSON with Comments (json)

The NumberInt and NumberLong classes represent 4-byte and 8-byte integers respectively. For example:

{

"year": NumberInt("2020"),

"words": NumberLong("95403")

}

Code language: JSON / JSON with Comments (json)

String

The string type represents any string of UTF-8 characters. For example:

{

"title": "MongDB Data Types"

}

Code language: JSON / JSON with Comments (json)

Date

The date type stores dates as 64-bit integers that represents milliseconds since the Unix epoch (January 1, 1970). It does not store the time zone. For example:

{

"updated_at": new Date()

}

Code language: JSON / JSON with Comments (json)

In JavaScript, the Date class is used to represent the date type in MongoDB.

Note that you should always call the new Date(), not just Date() when you create a new Date object because the Date() returns a string representation of the date, not the Date object.

The mongo shell displays dates using local time zone settings. However, MongoDB does not store date with the time zone. To store the time zone, you can use another key e.g., timezone.

Regular Expression

MongoDB allows you to store JavaScript regular expressions. For example:

```
{
"pattern": /\\d+/
}
```

Code language: JSON / JSON with Comments (json)

In this example, /\\d+/ is a regular expression that matches one or more digits.

Array

The array type allows you to store a list of values of any type. The values do not have to be in the same type, for example:

```
{
"title": "MongoDB Array",
"reviews": ["John", 3.5, "Jane", 5]
}
```

Code language: JSON / JSON with Comments (json)

The good thing about arrays in the document is that MongoDB understands their structures and allows you to carry operations on their elements.

For example, you can query all documents where 5 is an element of the reviews array. Also, you can create an index on the reviews array to improve the query performance.

Embeded Document

A value of a document can be another document that is often referred to as an embedded document.

The following example shows a book document that contains the author document as an embedded document:

```
{
"title": "MongoDB Tutorial",
"pages": 945,
"author": {
```

```json
"first_name": "John",
"last_name": "Doe"
}
}
```

Code language: JSON / JSON with Comments (json)

In this example, the author document has its own key/value pairs including first_name and last_name.

Object ID

In MongoDB, every document has an "_id" key. The value of the "_id" key can be any type. However, it defaults to an ObjectId.

The value of the "_id" key must be unique within a collection so that MongoDB can identify every document in the collection.

The ObjectId class is the default type for "_id". It is used to generate unique values globally across servers. Since MongoDB is designed to be distributed, it is important to ensure the identifiers are unique in the shared environment.

The ObjectId uses 12 bytes for storage, where each byte represents 2 hexadecimal digits. In total, an ObjectId is 24 hexadecimal digits.

The 12-byte ObjectId value consists of:

- A 4-byte timestamp value that represents the ObjectId‘s generated time measured in seconds since the Unix epoch.
- A 5-byte random value.
- A 3-byte increment counter, initialized to a random value.

These first 9 bytes of an ObjectId guarantee its uniqueness across servers and processes for a single second. The last 3 bytes guarantee uniqueness within a second in a single process.

As a result, these 12-bytes allow for up to 2563 (16,777,216) unique ObjectIds values to be generated per process in a single second.

When you insert a document without specifying a value for the "_id" key, MongoDB automatically generates a unique id for the document. For example:

```
db.books.insertOne({

"title": "MongoDB Basics"

});
```

Code language: JavaScript (javascript)

Output:

```
{

"acknowledged" : true,

"insertedId" : ObjectId("5f2fcae09b58c38603442a4f")

}
```

Code language: JSON / JSON with Comments (json)

MongoDB generated the id with the value ObjectId("5f2fcae09b58c38603442a4f"). You can view the inserted document like this:

```
db.books.find().pretty()
```

Code language: CSS (css)

Output:

```
{

"_id" : ObjectId("5f2fcae09b58c38603442a4f"),

"title" : "MongoDB Basics"

}
```

Code language: JSON / JSON with Comments (json)

In this tutorial, you have learned the most commonly used MongoDB data types including null, number, string, array, regular expression, date, and ObjectId.

1) Insert a document without an _id field example

The following example uses the insertOne() method to insert a new document into the books collection:

```
db.books.insertOne({

title: 'MongoDB insertOne',
isbn: '0-7617-6154-3'

});
```

Code language: CSS (css)

Output:

```
{

"acknowledged" : true,

"insertedId" : ObjectId("5f31cf00902f22de3464ddc4")

}
```

Code language: JSON / JSON with Comments (json)

In this example, we passed a document to the insertOne() method without specifying the _id field. Therefore, MongoDB automatically added the _id field and assigned it a unique ObjectId value.

Note that you will see a different ObjectId value from this example because ObjectId values are specific to machine and time when the insertOne() method executes.

To select the document that you have inserted, you can use the [find()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/>) method like this:

```
db.books.find()
```

Code language: CSS (css)

Output:

```
[

{

_id: ObjectId("621489fcf514a446bf1a98ea"),

title: 'MongoDB insertOne',

isbn: '0-7617-6154-3'

}

]
```

Code language: JavaScript (javascript)

2) Insert a document with an _id field example

The following example uses the insertOne() method to insert a document that has an _id field into the books collection:

```
db.books.insertOne({
_id: 1,
title: "Mastering Big Data",
isbn: "0-9270-4986-4"
});
```

Code language: CSS (css)

Output:

```
{ "acknowledged" : true, "insertedId" : 1 }
```

Code language: JSON / JSON with Comments (json)

The following example attempts to insert another document whose _id field already exists into the books collection:

```
db.books.insertOne({
_id: 1,
title: "MongoDB for JS Developers",
isbn: "0-4925-3790-9"
});
```

Code language: CSS (css)

Since the _id: 1 already exists, MongoDB threw the following exception:

```
WriteError({
"index" : 0,
"code" : 11000,
"errmsg" : "E11000 duplicate key error collection: bookstore.books index: _id_ dup key:
{ _id: 1.0 }",
"op" : {
"_id" : 1,
"title" : "MongoDB for JS Developers",
"isbn" : "0-4925-3790-9"
}
})
```

Code language: JavaScript (javascript)

Using MongoDB insertMany() method to insert multiple documents without specifying _id fields

The following statement uses the insertMany() method to insert multiple documents without the _id fields:

```
db.books.insertMany([
{ title: "NoSQL Distilled", isbn: "0-4696-7030-4"},
{ title: "NoSQL in 7 Days", isbn: "0-4086-6859-8"},
{ title: "NoSQL Database", isbn: "0-2504-6932-4"},
]);
```

Code language: JavaScript (javascript)

The findOne() method has the following syntax:

```
db.collection.findOne(query, projection)
```

Code language: CSS (css)

The findOne() accepts two optional arguments: query and projection.

- The query is a document that specifies the selection criteria.
- The projection is a document that specifies the fields in the matching document that you want to return.

If you omit the query, the findOne() returns the first document in the collection according to the natural order which is the order of documents on the disk.

If you don’t pass the projection argument, then findOne() will include all fields in the matching documents.

Using MongoDB findOne() method to select some fields

The following example uses the findOne() method to find the document with _id 5. And it returns only the _id and name fields of the matching document:

```
db.products.findOne({_id: 5}, {name: 1})
```

Code language: JavaScript (javascript)

The query returned the following document:

```
{ "_id" : 5, "name" : "SmartPhone" }
```

Code language: JavaScript (javascript)

As you can see clearly from the output, MongoDB includes the _id field in the returned document by default.

To completely remove it from the returned document, you need to explicitly specify _id:0 in the projection document like this:

```
db.products.findOne({ _id: 5 }, {name: 1, _id: 0})
```

Using MongoDB find() method to return selected fields

The following example uses the find() method to search for documents whose category is Java. It returns the fields _id, title and isbn:

```
db.books.find({ categories: 'Java'}, { title: 1,isbn: 1})
```

Returning fields in embedded documents

The following example returns the name, price, and _id fields of document _id 1. It also returns the screen field on the spec embedded document:

```
db.products.find({_id:1}, {
name: 1,
price: 1,
"spec.screen": 1
})
```

Code language: CSS (css)

Using $eq operator to check if a field equals a specified value

The following example uses the $eq operator to query the products collection to select all documents where the value of the price field equals 899:

```
db.products.find({
price: {
$eq: 899
}
}, {
name: 1,
price: 1
})
```

Code language: CSS (css)

The query is equivalent to the following:

```
db.products.find({
price: 899
}, {
name: 1,
price: 1
})
```

Using $gt to select documents where the value of a field is greater than a specified value

The following example uses the $gt operator to select documents from the products collection where price is greater than 699:

```
db.products.find({
price: {
$gt: 699
}
}, {
```

```
name: 1,
price: 1
})
```

Using $gte operator to select documents where a field is greater than or equal to a specified value

The following example uses the $gte operator to select documents from the products collection where price is greater than 799:

```
db.products.find({
price: {
$gte: 799
}
}, {
name: 1,
price: 1
})
```

Using the $gte operator to check if an array element is greater than or equal to a value

The following example uses the $gte operator to query the products collection to find all documents where the array storage has at least one element greater than or equal to 512:

```
db.products.find({
storage: {
$gte: 512
}
}, {
name: 1,
storage: 1
})
db.products.find({ price: { $lt: 799 } }, { name: 1, price: 1 })
```

Using the $ne operator to select documents where the value of a field is greater than a specified value

The following example uses the $ne operator to select documents from the products collection where the price is not equal to 899:

```
db.products.find({
price: {
$ne: 899
}
}, {
name: 1,
price: 1
})
```

Using the $in opeator to match values

The following example uses the $in operator to select documents from the products collection whose the price is either 599 or 799:

```
db.products.find({
price: {
$in: [699, 799]
}
}, {
name: 1,
price: 1
})
```

Using the MongoDB $nin opeator to match values

The following query uses the $nin operator to select documents from the products collection whose price is neither 599 or 799:

```
db.products.find({
price: {
$nin: [699, 799]
}
}, {
name: 1,
price: 1
})
```

Using MongoDB $and operator example

The following example uses the $and operator to select all documents in the products collection where:

- the value in the price field is equal to 899 **and**
- the value in the color field is either "white" or "black"

```
db.products.find({
$and: [{
price: 899
}, {
color: {
$in: ["white", "black"]
}
}]
}, {
name: 1,
price: 1,
color: 1
})
```

Using MongoDB $or operator example

The following example uses the $or operator to select all documents in the products collection where the value in the price field equals 799 or 899:

```
db.products.find({
$or: [{
price: 799
}, {
price: 899
}]
}, {
name: 1,
price: 1
})
```

Using MongoDB $not operator to select documents

The following example shows how to use the $not operator to find documents where:

- the price field is not greater than 699.
- do not contain the price field.

```
db.products.find({
price: {
$not: {
$gt: 699
}
}
}, {
name: 1,
```

price: 1

})

Using the MongoDB $exists operator example

The following example uses the $exists operator to select documents where the price field exists:

```
db.products.find(
{
price: {
$exists: true
}
},
{
name: 1,
price: 1
}
)
```

MongoDB provides you with three ways to identify a BSON type: string, number, and alias. The following table lists the BSON types identified by these three forms:

Type	**Number**	**Alias**
Double	1	“double”
String	2	“string”
Object	3	“object”
Array	4	“array”
Binary data	5	“binData”
ObjectId	7	“objectId”
Boolean	8	“bool”
Date	9	“date”
Null	10	“null”

Regular Expression	11	“regex”
JavaScript	13	“javascript”
32-bit integer	16	“int”
Timestamp	17	“timestamp”
64-bit integer	18	“long”
Decimal128	19	“decimal”
Min key	-1	“minKey”
Max key	127	“maxKey”

The $type operator also supports the number alias that matches against the following BSON types:

- double
- 32-bit integer
- 64-bit integer
- decimal

MongoDB $type operator examples

We’ll use the following products collection:

db.products.insertMany([

{ "_id" : 1, "name" : "xPhone", "price" : "799", "releaseDate" : ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 }, "color" : ["white", "black"], "storage" : [64, 128, 256] },

{ "_id" : 2, "name" : "xTablet", "price" : NumberInt(899), "releaseDate" : ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 }, "color" : ["white", "black", "purple"], "storage" : [128, 256, 512] },

{ "_id" : 3, "name" : "SmartTablet", "price" : NumberLong(899), "releaseDate" : ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 }, "color" : ["blue"], "storage" : [16, 64, 128] },

{ "_id" : 4, "name" : "SmartPad", "price" : [599, 699, 799], "releaseDate" : ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 }, "color" : ["white", "orange", "gold", "gray"], "storage" : [128, 256, 1024] },

{ "_id" : 5, "name" : "SmartPhone", "price" : ["599",699], "releaseDate" : ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" : 1.66 }, "color" : ["white", "orange", "gold", "gray"], "storage" : [128, 256] },

{ "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7, "cpu" : 3.66 }, "color" : ["black"], "storage" : [1024] }

])

Code language: JavaScript (javascript)

This products collection contains the price field that has int, double, long values.

1) Using the $type operator example

The following example uses the $type operator to query documents from the products collection where the price field is the string type or is an array containing an element that is a string type.

db.products.find({

price: {

$type: "string"

}

}, {

name: 1,

price: 1

})

Code language: CSS (css)

It returned the following documents:

{ "_id" : 1, "name" : "xPhone", "price" : "799" }

{ "_id" : 5, "name" : "SmartPhone", "price" : ["599", 699] }

Code language: JSON / JSON with Comments (json)

Since the string type corresponds to the number 2 (see the BSON types table above), you can use the number 2 in the query instead:

db.products.find({

```
price: {
$type: 2
}
}, {
name: 1,
price: 1
})
```

Using MongoDB $all operator to match values

The following example uses the $all operator to query the products collection for documents where the value of the color field is an array that includes "black" and "white":

```
db.products.find({
color: {
$all: ["black", "white"]
}
}, {
name: 1,
color: 1
})
```

We’ll use the following products collection for the demonstration:

```
db.products.insertMany([
{ "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 }, "color" : [ "white", "black" ], "storage" : [ 64, 128, 256 ] },
{ "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 }, "color" : [ "white", "black", "purple" ], "storage" : [ 128, 256, 512 ] },
```

{ "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 }, "color" : ["blue"], "storage" : [16, 64, 128] },

{ "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 }, "color" : ["white", "orange", "gold", "gray"], "storage" : [128, 256, 1024] },

{ "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" : 1.66 }, "color" : ["white", "orange", "gold", "gray"], "storage" : [128, 256] }

])

Code language: JavaScript (javascript)

1) Using the MongDB $elemMatch opeator example

The following example uses the $elemMatch operator to query documents from the products collection:

```
db.products.find({
storage: {
$elemMatch: {
$lt: 128
}
}
}, {
name: 1,
storage: 1
});
```

MongoDB sort() method examples

We’ll use the following products collection to illustrate how the sort() method works.

db.products.insertMany([

{ "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate" : ISODate("2011-05-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 }, "color" : ["white", "black"], "storage" : [64, 128, 256] },

{ "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate" : ISODate("2011-09-01T00:00:00Z"), "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 }, "color" : ["white", "black", "purple"], "storage" : [128, 256, 512] },

{ "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate" : ISODate("2015-01-14T00:00:00Z"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 }, "color" : ["blue"], "storage" : [16, 64, 128] },

{ "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate" : ISODate("2020-05-14T00:00:00Z"), "spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 }, "color" : ["white", "orange", "gold", "gray"], "storage" : [128, 256, 1024] },

{ "_id" : 5, "name" : "SmartPhone", "price" : 599, "releaseDate" : ISODate("2022-09-14T00:00:00Z"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" : 1.66 }, "color" : ["white", "orange", "gold", "gray"], "storage" : [128, 256] },

{ "_id" : 6, "name" : "xWidget", "spec" : { "ram" : 64, "screen" : 9.7, "cpu" : 3.66 }, "color" : ["black"], "storage" : [1024] },

{ "_id" : 7, "name" : "xReader", "price" : null, "spec" : { "ram" : 64, "screen" : 6.7, "cpu" : 3.66 }, "color" : ["black", "white"], "storage" : [128] }

])

Code language: JavaScript (javascript)

1) Sorting document by one field examples

The following query returns all documents from the products collection where the price field exists. For each document, it selects the _id, name, and price fields:

db.products.find({

'price': {

$exists: 1

}

}, {

name: 1,

```
price: 1
})
```

Code language: PHP (php)

Output:

```
[
{ _id: 1, name: 'xPhone', price: 799 },
{ _id: 2, name: 'xTablet', price: 899 },
{ _id: 3, name: 'SmartTablet', price: 899 },
{ _id: 4, name: 'SmartPad', price: 699 },
{ _id: 5, name: 'SmartPhone', price: 599 },
{ _id: 7, name: 'xReader', price: null }
]
```

Code language: JavaScript (javascript)

To sort the products by prices in ascending order, you use the sort() method like this:

```
db.products.find({
'price': {
$exists: 1
}
}, {
name: 1,
price: 1
}).sort({
price: 1
})
```

Code language: PHP (php)

Output:

```
[
{ _id: 7, name: 'xReader', price: null },
{ _id: 5, name: 'SmartPhone', price: 599 },
{ _id: 4, name: 'SmartPad', price: 699 },
{ _id: 1, name: 'xPhone', price: 799 },
{ _id: 2, name: 'xTablet', price: 899 },
{ _id: 3, name: 'SmartTablet', price: 899 }
]
```

Code language: JavaScript (javascript)

In this example, the sort() method places the document whose price is null first, and then the documents with the prices from lowest to highest.

To sort the documents in descending order, you change the value of the price field to -1 as shown in the following query:

```
db.products.find({
'price': {
$exists: 1
}
}, {
name: 1,
price: 1
}).sort({
price: -1
})
```

Code language: PHP (php)

Output:

```
[
{ _id: 2, name: 'xTablet', price: 899 },
{ _id: 3, name: 'SmartTablet', price: 899 },
{ _id: 1, name: 'xPhone', price: 799 },
{ _id: 4, name: 'SmartPad', price: 699 },
{ _id: 5, name: 'SmartPhone', price: 599 },
{ _id: 7, name: 'xReader', price: null }
]
```

Code language: JavaScript (javascript)

In this example, the sort() method places the document with the highest price first and the one whose price is null last. (See the sort order above)

2) Sorting document by two or more fields example

The following example uses the sort() method to sort the products by name and price in ascending order. It selects only documents where the price field exists and includes the _id, name, and price fields in the matching documents.

```
db.products.find({
'price': {
$exists: 1
}
}, {
name: 1,
price: 1
}).sort({
price: 1,
name: 1
});
```

Code language: PHP (php)

Output:

```
[
{ _id: 7, name: 'xReader', price: null },
{ _id: 5, name: 'SmartPhone', price: 599 },
{ _id: 4, name: 'SmartPad', price: 699 },
{ _id: 1, name: 'xPhone', price: 799 },
{ _id: 3, name: 'SmartTablet', price: 899 },
{ _id: 2, name: 'xTablet', price: 899 }
]
```

Code language: JavaScript (javascript)

In this example, the sort() method sorts the products by prices first. Then it sorts the sorted result set by names.

If you look at the result set more closely, you'll see that the products with _id 3 and 2 have the same price 899. The sort() method places the SmartTablet before the xTablet based on the ascending order specified by the name field.

The following example sorts the products by prices in ascending order and sorts the sorted products by names in descending order:

```
db.products.find({
'price': {
$exists: 1
}
}, {
name: 1,
price: 1
}).sort({
price: 1,
name: -1
})
```

Code language: PHP (php)

Output:

```
[
{ _id: 7, name: 'xReader', price: null },

{ _id: 5, name: 'SmartPhone', price: 599 },

{ _id: 4, name: 'SmartPad', price: 699 },

{ _id: 1, name: 'xPhone', price: 799 },

{ _id: 2, name: 'xTablet', price: 899 },

{ _id: 3, name: 'SmartTablet', price: 899 }

]
```

Code language: JavaScript (javascript)

In this example, the sort() method sorts the products by prices in ascending order. However, it sorts sorted products by names in descending order.

Unlike the previous example, the sort() places the xTable before SmartTablet.

3) Sorting documetns by dates

The following example sorts the documents from the products collection by values in the releaseDate field. It selects only document whose releaseDate field exists and includes the _id, name, and releaseDate fields in the matching documents:

```
db.products.find({

releaseDate: {

$exists: 1

}

}, {

name: 1,

releaseDate: 1

}).sort({

releaseDate: 1

});
```

Code language: CSS (css)

Output:

```
[
{
_id: 1,
name: 'xPhone',
releaseDate: ISODate("2011-05-14T00:00:00.000Z")
},
{
_id: 2,
name: 'xTablet',
releaseDate: ISODate("2011-09-01T00:00:00.000Z")
},
{
_id: 3,
name: 'SmartTablet',
releaseDate: ISODate("2015-01-14T00:00:00.000Z")
},
{
_id: 4,
name: 'SmartPad',
releaseDate: ISODate("2020-05-14T00:00:00.000Z")
},
{
_id: 5,
name: 'SmartPhone',
releaseDate: ISODate("2022-09-14T00:00:00.000Z")
}
]
```

Code language: JavaScript (javascript)

In this example, the sort() method places the documents with the releaseDate in ascending order.

The following query sorts the products by the values in the releaseDate field in descending order:

```
db.products.find({
releaseDate: {
$exists: 1
}
}, {
name: 1,
releaseDate: 1
}).sort({
releaseDate: -1
});
```

Code language: CSS (css)

Output:

```
[
{
_id: 5,
name: 'SmartPhone',
releaseDate: ISODate("2022-09-14T00:00:00.000Z")
},
{
_id: 4,
name: 'SmartPad',
releaseDate: ISODate("2020-05-14T00:00:00.000Z")
},
{
_id: 3,
name: 'SmartTablet',
releaseDate: ISODate("2015-01-14T00:00:00.000Z")
},
{
_id: 2,
name: 'xTablet',
releaseDate: ISODate("2011-09-01T00:00:00.000Z")
},
{
_id: 1,
name: 'xPhone',
releaseDate: ISODate("2011-05-14T00:00:00.000Z")
}
]
```

Code language: JavaScript (javascript)

4) Sorting documents by fields in embedded documents

The following example sorts the products by the values in the ram field in the spec embedded documents. It includes the _id, name, and spec fields in the matching documents.

```
db.products.find({}, {
name: 1,
spec: 1
}).sort({
"spec.ram": 1
});
```

Code language: JavaScript (javascript)

Output:

```
[
{ _id: 1, name: 'xPhone', spec: { ram: 4, screen: 6.5, cpu: 2.66 } },
{
_id: 5,
name: 'SmartPhone',
spec: { ram: 4, screen: 9.7, cpu: 1.66 }
},
{
_id: 4,
name: 'SmartPad',
spec: { ram: 8, screen: 9.7, cpu: 1.66 }
},
{
_id: 3,
name: 'SmartTablet',
spec: { ram: 12, screen: 9.7, cpu: 3.66 }
},
{
_id: 2,
name: 'xTablet',
spec: { ram: 16, screen: 9.5, cpu: 3.66 }
},
{
_id: 6,
name: 'xWidget',
spec: { ram: 64, screen: 9.7, cpu: 3.66 }
},
{
_id: 7,
name: 'xReader',
spec: { ram: 64, screen: 6.7, cpu: 3.66 }
}
]
```

Using MongoDB limit() and skip() to get the paginated result

Suppose you want to divide the products collection into pages, each has 2 products.

The following query uses the skip() and limit() to get the documents on the second page:

```
db.products.find({}, {
name: 1,
price: 1
}).sort({
price: -1,
name: 1
}).skip(2).limit(2);
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, name: 'xPhone', price: 799 },
{ _id: 4, name: 'SmartPad', price: 699 }
]
```

Using the MongoDB updateOne() method to update a single document

The following example uses the updateOne() method to update the price of the document with _id: 1:

```
db.products.updateOne({
_id: 1
}, {
$set: {
price: 899
}
})
```

Code language: PHP (php)

In this query:

- The { _id : 1 } is the filter argument that matches the documents to update. In this example, it matches the document whose _id is 1.
- The { $set: { price: 899 } } specifies the change to apply. It uses the $set operator to set the value of the price field to 899.

The query returns the following result:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

In this result document, the matchedCount indicates the number of matching documents (1) and the modifiedCount shows the number of the updated documents (1).

To verify the update, you can use the [findOne()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/>) method to retrieve the document _id: 1 as follows:

db.products.findOne({ _id: 1 }, { name: 1, price: 1 })

Code language: CSS (css)

It returned the following document:

{ _id: 1, name: 'xPhone', price: 899 }

Code language: CSS (css)

As you can see clearly from the output, the price has been updated successfully.

2) Using the MongoDB updateOne() method to update the first matching document

The following query selects the documents from the products collection in which the value of the price field is 899:

db.products.find({ price: 899 }, { name: 1, price: 1 })

Code language: CSS (css)

It returned the following documents:

[

{ _id: 1, name: 'xPhone', price: 899 },

{ _id: 2, name: 'xTablet', price: 899 },

{ _id: 3, name: 'SmartTablet', price: 899 }

]

Code language: JavaScript (javascript)

The following example uses the updateOne() method to update the first matching document where the price field is 899:

db.products.updateOne({ price: 899 }, { $set: { price: null } })

Code language: PHP (php)

It updated one document as shown in the following result:

{

```
acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}
```

Code language: CSS (css)

If you query the document with _id: 1, you’ll see that its price field is updated:

```
db.products.find({ _id: 1}, { name: 1, price: 1 })
```

Code language: CSS (css)

Output:

```
[ { _id: 1, name: 'xPhone', price: null } ]
```

Code language: JavaScript (javascript)

3) Using the updateOne() method to update embedded documents

The following query uses the [find()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/>) method to select the document with _id: 4:

```
db.products.find({ _id: 4 }, { name: 1, spec: 1 })
```

Code language: CSS (css)

It returned the following document:

```
[

{

_id: 4,

name: 'SmartPad',

spec: { ram: 8, screen: 9.7, cpu: 1.66 }

}

]
```

Code language: JavaScript (javascript)

The following example uses the updateOne() method to update the values of the ram, screen, and cpu fields in the spec embedded document of the document _id: 4:

```
db.products.updateOne({
_id: 4
}, {
$set: {
"spec.ram": 16,
"spec.screen": 10.7,
"spec.cpu": 2.66
}
})
```

Code language: PHP (php)

It returned the following document:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

If you query the document with _id 4 again, you’ll see the change:

```
db.products.find({ _id: 4 }, { name: 1, spec: 1 })
```

Code language: CSS (css)

Output:

```
[
{
```

```
_id: 4,
name: 'SmartPad',
spec: { ram: 16, screen: 10.7, cpu: 2.66 }
}
]
```

Code language: JavaScript (javascript)

4) Using the MongoDB updateOne() method to update array elements

The following example uses the updateOne() method to update the first and second elements of the storage array in the document with _id 4:

```
db.products.updateOne(
{ _id: 4},
{
$set: {
"storage.0": 16,
"storage.1": 32
}
}
)
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
```

}

Code language: CSS (css)

If you query the document with _id 4 from the products collection, you'll see that the first and second elements of the storage array have been updated:

```
db.products.find({ _id: 4 }, { name: 1, storage: 1 });
```

Code language: CSS (css)

Output:

```
[ { _id: 4, name: 'SmartPad', storage: [ 16, 32, 1024 ] } ]
```

Introduction to MongoDB updateMany() method

The updateMany() method allows you to update all documents that satisfy a condition.

The following shows the syntax of the updateMany() method:

```
db.collection.updateMany(filter, update, options)
```

Code language: CSS (css)

In this syntax:

- The filter is a document that specifies the condition to select the document for update. If you pass an empty document ({}) into the method, it'll update all the documents the collection.
- The update is a document that specifies the updates to apply.
- The options argument provides some options for updates that won't be covered in this tutorial.

The updateMany() method returns a document that contains multiple fields. The following are the notable ones:

- The matchedCount stores the number of matched documents.
- The modifiedCount stores the number of modified documents.

To form the update argument, you typically use the $set operator.

$set operator

The $set operator replaces the value of a field with a specified value. It has the following syntax:

```
{ $set: { <field1>: <value1>, <field2>: <value2>, ...}}
```

Code language: HTML, XML (xml)

If the field doesn’t exist, the $set operator will add the new field with the specified value to the document. If you specify the field with the dot notation e.g., embededDoc.field and the field does not exist, the $set operator will create the embedded document (embedded).

MongoDB updateMany() method examples

We’ll use the following products collection:

```
db.products.insertMany([
{ "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 },"color":["white","black"],"storage":[64,128,256]},
{ "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
{ "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 },"color":["blue"],"storage":[16,64,128]},
{ "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
{ "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
])
```

Code language: JavaScript (javascript)

1) Using the MongoDB updateMany() method to update multiple documents

The following example uses the updateMany() method to update the documents where the value of the price field is 899:

```
db.products.updateMany(
{ price: 899},
{ $set: { price: 895 }}
)
```

Code language: PHP (php)

In this query:

The { price : 899 } is the filter argument that specified the documents to update.

The { $set: { price: 895} } specifies the update to apply, which uses the $set operator to set the value of the price field to 895.

The query returns the following result:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 2,
modifiedCount: 2,
upsertedCount: 0
}
```

Code language: CSS (css)

In this result document, the matchedCount stores the number of matching documents (2) and the modifiedCount stores the number of the updated documents (2).

To check the update, you can use the [find()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/>) method to select the documents where the value of the price field is 895 as follows:

```
db.products.find({
price: 895
}, {
```

name: 1,

price: 1

})

Code language: CSS (css)

The query returned the following documents:

[

{ _id: 2, name: 'xTablet', price: 895 },

{ _id: 3, name: 'SmartTablet', price: 895 }

]

Code language: JavaScript (javascript)

The price field values have been updated successfully.

2) Using the updateMany() method to update embedded documents

The following query uses the [find()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/>) method to select the documents where the value in the price field is greater than 700:

db.products.find({

price: { $gt: 700}

}, {

name: 1,

price: 1,

spec: 1

})

Code language: CSS (css)

The query returned the following documents:

[

{

```
_id: 1,
name: 'xPhone',
price: 799,
spec: { ram: 4, screen: 6.5, cpu: 2.66 }
},
{
_id: 2,
name: 'xTablet',
price: 895,
spec: { ram: 16, screen: 9.5, cpu: 3.66 }
},
{
_id: 3,
name: 'SmartTablet',
price: 895,
spec: { ram: 12, screen: 9.7, cpu: 3.66 }
}
]
```

Code language: JavaScript (javascript)

The following example uses the updateMany() method to update the values of the ram, screen, and cpu fields in the spec embedded documents of these documents:

```
db.products.updateMany({
price: { $gt: 700}
}, {
$set: {
"spec.ram": 32,
"spec.screen": 9.8,
```

"spec.cpu": 5.66

}

})

Code language: PHP (php)

The query returned the following document indicating that the three documents have been updated successfully:

{

acknowledged: true,

insertedId: null,

matchedCount: 3,

modifiedCount: 3,

upsertedCount: 0

}

Code language: CSS (css)

3) Using the MongoDB updateMany() method to update array elements

The following example uses the updateMany() method to update the first and second elements of the storage array of the documents where the _id is 1, 2 and 3:

db.products.updateMany({

_id: {

$in: [1, 2, 3]

}

}, {

$set: {

"storage.0": 16,

"storage.1": 32

}

```
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 3,
modifiedCount: 3,
upsertedCount: 0
}
```

Code language: CSS (css)

If you query the documents whose _id is 1, 2, and 3 from the products collection, you'll see that the first and second elements of the storage array have been updated:

```
db.products.find(
{ _id: { $in: [1,2,3]}},
{ name: 1, storage: 1}
)
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, name: 'xPhone', storage: [ 16, 32, 256 ] },
{ _id: 2, name: 'xTablet', storage: [ 16, 32, 512 ] },
{ _id: 3, name: 'SmartTablet', storage: [ 16, 32, 128 ] }
]
```

MongoDB $min operator example

We'll use the following products collection:

```
db.products.insertMany([
{ "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("2011-05-14"),
"spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 },"color":["white","black"],"storage":
[64,128,256]},
{ "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("2011-09-01") ,
"spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 },"color":
["white","black","purple"],"storage":[128,256,512]},
{ "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate("2015-01-14"),
"spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 },"color":["blue"],"storage":[16,64,128]},
{ "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2020-05-
14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 },"color":
["white","orange","gold","gray"],"storage":[128,256,1024]},
{ "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("2022-09-14"),
"spec" : { "ram" : 4, "screen" : 5.7, "cpu" : 1.66 },"color":
["white","orange","gold","gray"],"storage":[128,256]}
])
```

Code language: JavaScript (javascript)

The following example uses the $min operator to update the price of the document _id 5:

```
db.products.updateOne({
_id: 5
}, {
$min: {
price: 699
}
})
```

Code language: PHP (php)

The query found a matching document. However, it didn’t update any:

```
{
```

```
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 0,
upsertedCount: 0
}
```

Code language: CSS (css)

The reason is that the new value 699 is greater than the current value 599.

The following example uses the $min operator to update the price of the document _id 5:

```
db.products.updateOne({
_id: 5
}, {
$min: {
price: 499
}
})
```

Code language: PHP (php)

In this case, the price field of the document _id 5 is updated to 499:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

This query verifies the update:

```
db.products.find({ _id: 5 }, { name: 1, price: 1 })
```

Code language: CSS (css)

Output:

```
[ { _id: 5, name: 'SmartPhone', price: 499 } ]
```

1) Using the MongoDB $mul to multiply the value of a field

The following example uses the $mul operator to multiply the price of the document _id 5 by 10%:

```
db.products.updateOne({ _id: 5 }, { $mul: { price: 1.1 } })
```

Code language: PHP (php)

The output document showed that the query matched one document and updated it:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query verifies the update:

```
db.products.find({
_id: 5
}, {
name: 1,
price: 1
})
```

Code language: CSS (css)

Output:

```
[ { _id: 5, name: 'SmartPhone', price: 658.9000000000001 } ]
```

Code language: CSS (css)

2) Using the MongoDB $mul to multiply the values of array elements

The following query uses the $mul operator to double the values of the first, second, and third array elements in the storage array of the document _id 1:

```
db.products.updateOne({
_id: 1
}, {
$mul: {
"storage.0": 2,
"storage.1": 2,
"storage.2": 2
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query uses the [findOne()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-findone/>) method to select the document with _id 1 to verify the update:

```
db.products.findOne({ _id: 1 }, { name: 1, storage: 1 })
```

Code language: CSS (css)

Output:

```
{ _id: 1, name: 'xPhone', storage: [ 128, 256, 512 ] }
```

Code language: CSS (css)

3) Using the $mul operator to multiply the values of a field in embedded documents

This example uses the $mul operator to double the values of the ram field in the spec embedded documents of all documents from the products collection:

```
db.products.updateMany({}, {
$mul: {
"spec.ram": 2
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 5,
modifiedCount: 5,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query returns all documents from the products collection:

```
db.products.find({}, { name: 1, "spec.ram": 1 })
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, name: 'xPhone', spec: { ram: 8 } },
{ _id: 2, name: 'xTablet', spec: { ram: 32 } },
{ _id: 3, name: 'SmartTablet', spec: { ram: 24 } },
{ _id: 4, name: 'SmartPad', spec: { ram: 16 } },
{ _id: 5, name: 'SmartPhone', spec: { ram: 8 } }
]
```

MongoDB $unset operator examples

We’ll use the following products collection:

```
db.products.insertMany([
{ "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 },"color":["white","black"],"storage":[64,128,256]},
{ "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
{ "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 },"color":["blue"],"storage":[16,64,128]},
{ "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
{ "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
])
```

Code language: JavaScript (javascript)

1) Using the MongoDB $unset operator to remove a field from a document

The following example uses the $unset operator to remove the price field from the document _id 1 in the products collection:

```
db.products.updateOne({
_id: 1
}, {
$unset: {
price: ""
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

The modifiedCount indicated that one document has been modified. This query returns all documents from the products collection to verify the update:

```
db.products.find({}, { name: 1, price: 1 })
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, name: 'xPhone' },
{ _id: 2, name: 'xTablet', price: 899 },
{ _id: 3, name: 'SmartTablet', price: 899 },
{ _id: 4, name: 'SmartPad', price: 699 },
{ _id: 5, name: 'SmartPhone', price: 599 }
]
```

Code language: JavaScript (javascript)

As you can see clearly from the output, the $unset operator has completely removed the price field from the document with _id 1.

2) Using the MongoDB $unset operator to remove a field in an embedded document

The following statement uses the $unset operator to remove the ram field from the spec embedded documents of all documents in the products collection:

```
db.products.updateMany({}, {
$unset: {
"spec.ram": ""
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 5,
modifiedCount: 5,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query returns all documents from the products collection:

```
db.products.find({}, {
spec: 1
})
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, spec: { screen: 6.5, cpu: 2.66 } },
{ _id: 2, spec: { screen: 9.5, cpu: 3.66 } },
{ _id: 3, spec: { screen: 9.7, cpu: 3.66 } },
{ _id: 4, spec: { screen: 9.7, cpu: 1.66 } },
{ _id: 5, spec: { screen: 5.7, cpu: 1.66 } }
]
```

As you can see, the ram field has been removed from spec embedded document in all documents.

3) Using the MongoDB $unset operator to set array elements to null

The following example uses the $unset operator to set the first elements of the storage arrays to null:

```
db.products.updateMany({}, { $unset: { "storage.0": "" } })
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 5,
modifiedCount: 5,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query selects the storage array from all documents in the products collection:

```
db.products.find({}, { "storage":1})
```

Code language: JavaScript (javascript)

Output:

```
[
{ _id: 1, storage: [ null, 128, 256 ] },
{ _id: 2, storage: [ null, 256, 512 ] },
{ _id: 3, storage: [ null, 64, 128 ] },
{ _id: 4, storage: [ null, 256, 1024 ] },
{ _id: 5, storage: [ null, 256 ] }
]
```

Code language: JavaScript (javascript)

In this example, the $unset operator sets the first elements of the storage arrays to null instead of removing them completely.

4) Using the MongoDB $unset operator to remove multiple fields from a document

The following statement uses the $unset operator to remove the releaseDate and spec fields from all the documents in the products collection:

```
db.products.updateMany({}, {
$unset: {
releaseDate: "",
spec: ""
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 5,
modifiedCount: 5,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query verifies the update:

```
db.products.find({}, {
name: 1,
storage: 1,
releaseDate: 1,
spec: 1
})
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, name: 'xPhone', storage: [ null, 128, 256 ] },
{ _id: 2, name: 'xTablet', storage: [ null, 256, 512 ] },
{ _id: 3, name: 'SmartTablet', storage: [ null, 64, 128 ] },
{ _id: 4, name: 'SmartPad', storage: [ null, 256, 1024 ] },
```

{ _id: 5, name: 'SmartPhone', storage: [null, 256] }

]

Introduction to the MongoDB $rename operator

Sometimes, you want to rename a field in a document e.g., when it is misspelled or not descriptive enough. In this case, you can use the $rename operator.

The $rename is a field update operator that allows you to rename a field in a document to the new one.

The $rename operator has the following syntax:

{ $rename: { <field_name>: <new_field_name>, ...}}

Code language: HTML, XML (xml)

In this syntax, the <new_field_name> must be different from the <field_name>.

If the document has a field with the same name as the <new_field_name>, the $rename operator removes that field and renames the specified <field_name> to <new_field_name>.

In case the <field_name> doesn't exist in the document, the $rename operator does nothing. It also won't issue any warnings or errors.

The $rename operator can rename fields in embedded documents. In addition, it can move these fields in and out of the embedded documents.

MongoDB $rename field operator examples

We'll use the following products collection:

db.products.insertMany([

{ "_id" : 1, "nmea" : "xPhone", "price" : 799, "releaseDate": ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 },"color":["white","black"],"storage":[64,128,256]},

{ "_id" : 2, "nmea" : "xTablet", "price" : 899, "releaseDate": ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 },"color":["white","black","purple"],"storage":[128,256,512]},

```
{ "_id" : 3, "nmea" : "SmartTablet", "price" : 899, "releaseDate": ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 },"color":["blue"],"storage":[16,64,128]},

{ "_id" : 4, "nmea" : "SmartPad", "price" : 699, "releaseDate": ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},

{ "_id" : 5, "nmea" : "SmartPhone", "price" : 599,"releaseDate": ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}

])
```

Code language: JavaScript (javascript)

1) Using MongoDB $rename to rename a field in a document

The following example uses the $rename operator to rename the misspelled field nmea to name:

```
db.products.updateMany({}, {

$rename: {

nmea: "name"

}

})
```

Code language: PHP (php)

In this example, the $rename operator changed the field name from nmea to name as indicated in the following returned document:

```
{

acknowledged: true,

insertedId: null,

matchedCount: 5,

modifiedCount: 5,

upsertedCount: 0

}
```

Code language: CSS (css)

To verify the update, you can use the [find()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/>) method to select all documents from the products collection:

```
db.products.find({}, { name: 1 })
```

Code language: CSS (css)

Output:

```
[
{ _id: 1, name: 'xPhone' },
{ _id: 2, name: 'xTablet' },
{ _id: 3, name: 'SmartTablet' },
{ _id: 4, name: 'SmartPad' },
{ _id: 5, name: 'SmartPhone' }
]
```

Code language: JavaScript (javascript)

2) Using MongoDB $rename operator to rename fields in embedded documents

The following example uses the $rename operator to change the size field of the spec embedded document to screenSize:

```
db.products.updateMany({}, {
$rename: {
"spec.screen": "spec.screenSize"
}
})
```

Code language: PHP (php)

It returned the following result:

```
{
```

```
acknowledged: true,

insertedId: null,

matchedCount: 5,

modifiedCount: 0,

upsertedCount: 0

}
```

Code language: CSS (css)

This query uses the [find()](<https://www.mongodbtutorial.org/mongodb-crud/mongodb-find/>) method to select all documents from the products collection:

```
db.products.find({}, {

spec: 1

})
```

Code language: CSS (css)

Here is the output:

```
[

{ _id: 1, spec: { ram: 4, cpu: 2.66, screenSize: 6.5 } },

{ _id: 2, spec: { ram: 16, cpu: 3.66, screenSize: 9.5 } },

{ _id: 3, spec: { ram: 12, cpu: 3.66, screenSize: 9.7 } },

{ _id: 4, spec: { ram: 8, cpu: 1.66, screenSize: 9.7 } },

{ _id: 5, spec: { ram: 4, cpu: 1.66, screenSize: 5.7 } }

]
```

As you can see from the output, the screen fields in the spec embedded documents have been renamed to screenSize.

3) Using the MongoDB $rename to move field out of the embedded document

The following example uses the $rename operator to move the cpu field out of the spec embedded document in the document _id 1:

```
db.products.updateOne({
_id: 1
},
{
$rename: {
"spec.cpu": "cpu"
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

Code language: CSS (css)

The following query selects the document with _id 1 to verify the rename:

```
db.products.find({ _id: 1})
```

Code language: CSS (css)

Output:

```
[
{
_id: 1,
price: 799,
releaseDate: ISODate("2011-05-14T00:00:00.000Z"),
spec: { ram: 4, screenSize: 6.5 },
color: [ 'white', 'black' ],
storage: [ 64, 128, 256 ],
name: 'xPhone',
cpu: 2.66
}
]
```

Code language: JavaScript (javascript)

As you can see clearly from the output, the cpu field becomes the top-level field.

4) Using the MongoDB $rename to rename a field to an existing field

The following example uses the $rename operator to rename the field color to storage in the document with _id 2.

However, the storage field already exists. Therefore, the $rename operator removes the storage field and renames the field color to storage:

```
db.products.updateOne({
_id: 2
}, {
$rename: {
"color": "storage"
}
})
```

Code language: PHP (php)

Output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 0,
upsertedCount: 0
}
```

Code language: CSS (css)

Let’s check the document _id 2:

```
db.products.find({ _id: 2 })
```

Code language: CSS (css)

Output:

```
[
{
_id: 2,
price: 899,
releaseDate: ISODate("2011-09-01T00:00:00.000Z"),
spec: { ram: 16, cpu: 3.66, screenSize: 9.5 },
storage: [ 'white', 'black', 'purple' ],
name: 'xTablet'
}
]
```

Introduction to the MongoDB upsert

Upsert is a combination of **up**date and in**sert**. Upsert performs two functions:

- Update data if there is a matching document.
- Insert a new document in case there is no document matches the query criteria.

To perform an upsert, you use the following updateMany() method with the upsert option set to true:

```
document.collection.updateMany(query, update, { upsert: true} )
```

Code language: CSS (css)

The upsert field in the third argument is set to false by default. This means that if you omit it, the method will only update the documents that match the query.

Notice that the updateOne() method also can upsert with the { upsert: true }.

MongoDB upsert examples

We'll use the following products collection.

db.products.insertMany([

{ "_id" : 1, "nmea" : "xPhone", "price" : 799, "releaseDate": ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 },"color":["white","black"],"storage": [64,128,256]},

{ "_id" : 2, "nmea" : "xTablet", "price" : 899, "releaseDate": ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 },"color": ["white","black","purple"],"storage":[128,256,512]},

{ "_id" : 3, "nmea" : "SmartTablet", "price" : 899, "releaseDate": ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 },"color":["blue"],"storage":[16,64,128]},

{ "_id" : 4, "nmea" : "SmartPad", "price" : 699, "releaseDate": ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 },"color": ["white","orange","gold","gray"],"storage":[128,256,1024]},

{ "_id" : 5, "nmea" : "SmartPhone", "price" : 599,"releaseDate": ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 5.7, "cpu" : 1.66 },"color": ["white","orange","gold","gray"],"storage":[128,256]}

])

Code language: JavaScript (javascript)

The following query uses the update() method to update the price for the document _id 6:

db.products.updateMany(

{_id: 6 },

{ $set: {price: 999} }

)

Code language: PHP (php)

The query found no match and didn't update any document as shown in the following output:

```
{
acknowledged: true,
insertedId: null,
matchedCount: 0,
modifiedCount: 0,
upsertedCount: 0
}
```

Code language: CSS (css)

If you pass the { upsert: true } to the updateMany() method, it'll insert a new document. For example:

```
db.products.updateMany(
{ _id: 6 },
{ $set: {price: 999} },
{ upsert: true}
)
```

Code language: PHP (php)

The query returns the following document:

```
{
acknowledged: true,
insertedId: 6,
matchedCount: 0,
modifiedCount: 0,
upsertedCount: 1
}
```

Code language: CSS (css)

The output indicates that there was no matching document (matchedCount is zero) and the updateMany() method didn't update any document.

However, the updateMany() method inserted one document and returned the id of the new document stored in the upsertedId field.

If you query the document with _id 6 from the products collection, you'll see the new document with the price field:

```
db.products.find({_id:6})
```

Code language: CSS (css)

Output:

```
[ { _id: 6, price: 999 } ]
```

Code language: CSS (css)

Summary

- Use the { upsert: true } argument in the updateMany() or updateOne() method to perform an upsert.

Introduction to the MongoDB deleteOn() method

The deleteOne() method allows you to delete a single document from a collection.

The deleteOne() method has the following syntax:

```
db.collection.deleteOne(filter, option)
```

Code language: CSS (css)

The deleteOne() method accepts two arguments:

- filter is an required argument. The filter is a document that specifies the condition for matching the document to remove. If you pass an empty document {} into the method, it'll delete the first document in the collection.
- option is an optional argument. The option is a document that specifies the deletion options.

The deleteOne() method returns a document containing the deleteCount field that stores the number of deleted documents.

To delete all documents that match a condition from a collection, you use the deleteMany() method.

MongoDB deleteCount() method examples

We'll use the following products collection:

```
db.products.insertMany([
{ "_id" : 1, "name" : "xPhone", "price" : 799, "releaseDate": ISODate("2011-05-14"), "spec" : { "ram" : 4, "screen" : 6.5, "cpu" : 2.66 },"color":["white","black"],"storage":[64,128,256]},
{ "_id" : 2, "name" : "xTablet", "price" : 899, "releaseDate": ISODate("2011-09-01") , "spec" : { "ram" : 16, "screen" : 9.5, "cpu" : 3.66 },"color":["white","black","purple"],"storage":[128,256,512]},
{ "_id" : 3, "name" : "SmartTablet", "price" : 899, "releaseDate": ISODate("2015-01-14"), "spec" : { "ram" : 12, "screen" : 9.7, "cpu" : 3.66 },"color":["blue"],"storage":[16,64,128]},
{ "_id" : 4, "name" : "SmartPad", "price" : 699, "releaseDate": ISODate("2020-05-14"),"spec" : { "ram" : 8, "screen" : 9.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256,1024]},
{ "_id" : 5, "name" : "SmartPhone", "price" : 599,"releaseDate": ISODate("2022-09-14"), "spec" : { "ram" : 4, "screen" : 9.7, "cpu" : 1.66 },"color":["white","orange","gold","gray"],"storage":[128,256]}
])
```

Code language: JavaScript (javascript)

1) Using the deleteOne() method to delete a single document

The following example uses the deleteOne() method to delete a document with the _id is 1 from the products collection:

```
db.products.deleteOne({ _id: 1 })
```

Code language: CSS (css)

The query returned the following document:

{ "acknowledged" : true, "deletedCount" : 1 }

Code language: JSON / JSON with Comments (json)

The deleteCount field returned 1 indicating the number of deleted documents.

2) Using the deleteOne() method to delete the first document from a collection

The following query uses the deleteOne() method to remove the first document returned from the products collection:

db.products.deleteOne({})

Code language: CSS (css)

It returned the following document:

{ "acknowledged" : true, "deletedCount" : 1 }

Code language: JSON / JSON with Comments (json)

In this example, we passed an empty document {} to the deleteOne() method. Therefore, it removed the first document from the products collection.

Create One Document

Let's begin by creating a new Airbnb listing. We can do so by calling **Collection**'s **insertOne()**. insertOne() will insert a single document into the collection. The only required parameter is the new document (of type object) that will be inserted. If our new document does not contain the _id field, the MongoDB driver will automatically create an _id for the document.

Our function to create a new listing will look something like the following:

Code Snippet

The output would be something like the following:

Code Snippet

1

async function createListing(client, newListing){

```
const result = await
client.db("sample_airbnb").collection("listingsAndReviews").insertOne(newListing);
console.log(`New listing created with the following id: ${result.insertedId}`);
}
await createListing(client,
{
name: "Lovely Loft",
summary: "A charming loft in Paris",
bedrooms: 1,
bathrooms: 1
}
);
```

!https://webimages.mongodb.com/_com_assets/cms/ks9ey91u6mgoo8on4-copyIcon.svg?auto=format%252Ccompress

Note that since we did not include a field named _id in the document, the MongoDB driver automatically created an _id for us. The _id of the document you create will be different from the one shown above. For more information on how MongoDB generates _id, see **Quick Start: BSON Data Types - ObjectId**.

If you're not a fan of copying and pasting, you can get a full copy of the code above in the **Node.js Quick Start GitHub Repo**.

Create Multiple Documents

Sometimes you will want to insert more than one document at a time. You could choose to repeatedly call insertOne(). The problem is that, depending on how you've structured your code, you may end up waiting for each insert operation to return before beginning the next, resulting in slow code.

Instead, you can choose to call **Collection**'s **insertMany()**. insertMany() will insert an array of documents into your collection.

One important option to note for insertMany() is ordered. If ordered is set to true, the documents will be inserted in the order given in the array. If any of the inserts fail (for example, if you attempt to insert a document with an _id that is already being used by another document in the collection), the remaining documents will not be inserted. If ordered is set to false, the documents may not be inserted in the order given in the array. MongoDB will attempt to insert all of the documents in the given array—regardless of whether any of the other inserts fail. By default, ordered is set to true.

Let's write a function to create multiple Airbnb listings.

```
async function createMultipleListings(client, newListings){
const result = await
client.db("sample_airbnb").collection("listingsAndReviews").insertMany(newListings);
console.log(`${result.insertedCount} new listing(s) created with the following id(s):`);
console.log(result.insertedIds);
}
async function findOneListingByName(client, nameOfListing) {
const result = await
client.db("sample_airbnb").collection("listingsAndReviews").findOne({ name:
nameOfListing });
if (result) {
console.log(`Found a listing in the collection with the name '${nameOfListing}':`);
console.log(result);
} else {
console.log(`No listings found with the name '${nameOfListing}'`);
}
}
client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
);
```

```
const cursor = client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 });
const cursor = client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 })
.limit(maximumNumberOfResults);
const cursor = client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 })
.limit(maximumNumberOfResults);
const results = await cursor.toArray();
async function
findListingsWithMinimumBedroomsBathroomsAndMostRecentReviews(client, {
minimumNumberOfBedrooms = 0,
minimumNumberOfBathrooms = 0,
maximumNumberOfResults = Number.MAX_SAFE_INTEGER
} = {}) {
```

```
const cursor = client.db("sample_airbnb").collection("listingsAndReviews").find(
{
bedrooms: { $gte: minimumNumberOfBedrooms },
bathrooms: { $gte: minimumNumberOfBathrooms }
}
).sort({ last_review: -1 })
.limit(maximumNumberOfResults);
const results = await cursor.toArray();
if (results.length > 0) {
console.log(`Found listing(s) with at least ${minimumNumberOfBedrooms} bedrooms
and ${minimumNumberOfBathrooms} bathrooms:`);
results.forEach((result, i) => {
date = new Date(result.last_review).toDateString();
console.log();
console.log(`${i + 1}. name: ${result.name}`);
console.log(` _id: ${result._id}`);
console.log(` bedrooms: ${result.bedrooms}`);
console.log(` bathrooms: ${result.bathrooms}`);
console.log(` most recent review date: ${new Date(result.last_review).toDateString()}`);
});
} else {
console.log(`No listings found with at least ${minimumNumberOfBedrooms} bedrooms
and ${minimumNumberOfBathrooms} bathrooms`);
}
}
await findListingsWithMinimumBedroomsBathroomsAndMostRecentReviews(client, {
minimumNumberOfBedrooms: 4,
minimumNumberOfBathrooms: 2,
maximumNumberOfResults: 5
});
async function upsertListingByName(client, nameOfListing, updatedListing) {
const result = await client.db("sample_airbnb").collection("listingsAndReviews")
.updateOne({ name: nameOfListing },
{ $set: updatedListing },
{ upsert: true });
console.log(`${result.matchedCount} document(s) matched the query criteria.`);
if (result.upsertedCount > 0) {
console.log(`One document was inserted with the id ${result.upsertedId._id}`);
} else {
console.log(`${result.modifiedCount} document(s) was/were updated.`);
}
}
```

Example

Here's a small code snippet to illustrate some of the terminology above:

```
const puppySchema = new mongoose.Schema({
name: {
type: String,
required: true
},
age: Number
});
const Puppy = mongoose.model('Puppy', puppySchema);
```

In the code above, puppySchema defines the shape of the document which has two fields, name, and age.

The SchemaType for name is String and for age is Number. Note that you can define the SchemaType for a field by using an object with a type property like with name. Or you can apply a SchemaType directly to the field like with age.

Also, notice that the SchemaType for name has the option required set to true. To use options like required and lowercase for a field, you need to use an object to set the SchemaType.

At the bottom of the snippet, puppySchema is compiled into a model named Puppy, which can then be used to construct documents in an application.

Getting Started

Mongo Installation

Before we get started, let's setup Mongo. You can choose from **one of the following options** (we are using option #1 for this article):

1. Download the appropriate MongoDB version for your Operating System from the MongoDB Website and follow their installation instructions
2. Create a free sandbox database subscription on mLab
3. Install Mongo using Docker if you prefer to use Docker

Let's navigate through some of the basics of Mongoose by implementing a model that represents data for a simplified address book.

I am using Visual Studio Code, Node 8.9, and NPM 5.6. Fire up your favorite IDE, create a blank project, and let's get started! We will be using the limited ES6 syntax in Node, so we won't be configuring Babel.

NPM Install

Let's go to the project folder and initialize our project

npm init -y

Let's install Mongoose and a validation library with the following command:

npm install mongoose validator

The above install command will install the latest version of the libraries. The Mongoose syntax in this article is specific to Mongoose v5 and beyond.

Database Connection

Create a file ./src/database.js ******under the project root.

Next, we will add a simple class with a method that connects to the database.

Your connection string will vary based on your installation.

```
let mongoose = require('mongoose');
const server = '127.0.0.1:27017'; // REPLACE WITH YOUR DB SERVER
const database = 'fcc-Mail'; // REPLACE WITH YOUR DB NAME
class Database {
constructor() {
this._connect();
}
_connect() {
mongoose
.connect(`mongodb://${server}/${database}`)
.then(() => {
console.log('Database connection successful');
})
.catch((err) => {
console.error('Database connection error');
});
}
}
module.exports = new Database();
```

The require('mongoose') ****call above returns a Singleton object. It means that the first time you call require('mongoose'), it is creating an instance of the Mongoose class and returning it. On subsequent calls, it will return the same instance that was created and returned to you the first time because of how module import/export works in ES6.

https://cdn-media-1.freecodecamp.org/images/0*RvVsD_byUakUzuCj.

Module import/require work-flow

Similarly, we have turned our Database class into a singleton by returning an instance of the class in the module.exports statement because we only need a single connection to the database.

ES6 makes it very easy for us to create a singleton (single instance) pattern because of how the module loader works by caching the response of a previously imported file.

Mongoose Schema vs. Model

A Mongoose model is a wrapper on the Mongoose schema. A Mongoose schema defines the structure of the document, default values, validators, etc., whereas a Mongoose model provides an interface to the database for creating, querying, updating, deleting records, etc.

Creating a Mongoose model comprises primarily of three parts:

1. Referencing Mongoose

let mongoose = require('mongoose');

This reference will be the same as the one that was returned when we connected to the database, which means the schema and model definitions will not need to explicitly connect to the database.

2. Defining the Schema

A schema defines document properties through an object where the key name corresponds to the property name in the collection.

let emailSchema = new mongoose.Schema({

email: String

});

Here we define a property called email ****with a schema type String ****which maps to an internal validator that will be triggered when the model is saved to the database. It will fail if the data type of the value is not a string type.

The following Schema Types are permitted:

- Array

- Boolean
- Buffer
- Date
- Mixed (A generic / flexible data type)
- Number
- ObjectId
- String

Mixed and ObjectId are defined under require(‘mongoose’).Schema.Types.

3. Exporting a Model

We need to call the model constructor on the Mongoose instance and pass it the name of the collection and a reference to the schema definition.

```
module.exports = mongoose.model('Email', emailSchema);
```

Let’s combine the above code into ./src/models/email.js ****to define the contents of a basic email model:

```
let mongoose = require('mongoose');
let emailSchema = new mongoose.Schema({
email: String
});
module.exports = mongoose.model('Email', emailSchema);
```

A schema definition should be simple, but its complexity is usually based on application requirements. Schemas can be reused and they can contain several child-schemas too. In the example above, the value of the email property is a simple value type. However, it can also be an object type with additional properties on it.

We can create an instance of the model we defined above and populate it using the following syntax:

```
let EmailModel = require('./email');
let msg = new EmailModel({
email: 'ada.lovelace@gmail.com'
});
```

Let’s enhance the Email schema to make the email property a unique, required field and convert the value to lowercase before saving it. We can also add a validation function that will ensure that the value is a valid email address. We will reference and use the validator library installed earlier.

```
let mongoose = require('mongoose');
let validator = require('validator');
let emailSchema = new mongoose.Schema({
email: {
type: String,
required: true,
unique: true,
lowercase: true,
validate: (value) => {
return validator.isEmail(value);
}
}
});
module.exports = mongoose.model('Email', emailSchema);
```

Basic Operations

Mongoose has a flexible API and provides many ways to accomplish a task. We will not focus on the variations because that is out of scope for this article, but remember that most of the operations can be done in more than one way either syntactically or via the application architecture.

Create Record

Let’s create an instance of the email model and save it to the database:

```
let EmailModel = require('./email');
let msg = new EmailModel({
```

```
email: 'ADA.LOVELACE@GMAIL.COM'
});
msg
.save()
.then((doc) => {
console.log(doc);
})
.catch((err) => {
console.error(err);
});
```

The result is a document that is returned upon a successful save:

```
{
_id: 5a78fe3e2f44ba8f85a2409a,
email: 'ada.lovelace@gmail.com',
__v: 0
}
```

The following fields are returned (internal fields are prefixed with an underscore):

1. The _id field is auto-generated by Mongo and is a primary key of the collection. Its value is a unique identifier for the document.
2. The value of the email field is returned. Notice that it is lower-cased because we specified the lowercase: true attribute in the schema.
3. __v is the versionKey property set on each document when first created by Mongoose. Its value contains the internal revision of the document.

If you try to repeat the save operation above, you will get an error because we have specified that the email field should be unique.

Fetch Record

Let’s try to retrieve the record we saved to the database earlier. The model class exposes several static and instance methods to perform operations on the database. We will now try to find the record that we created previously using the find method and pass the email as the search term.

```
EmailModel.find({
email: 'ada.lovelace@gmail.com' // search query
})
.then((doc) => {
console.log(doc);
})
.catch((err) => {
console.error(err);
});
```

The document returned will be similar to what was displayed when we created the record:

```
{
_id: 5a78fe3e2f44ba8f85a2409a,
email: 'ada.lovelace@gmail.com',
__v: 0
}
```

Update Record

Let’s modify the record above by changing the email address and adding another field to it, all in a single operation. For performance reasons, Mongoose won’t return the updated document so we need to pass an additional parameter to ask for it:

```
EmailModel.findOneAndUpdate(
{
email: 'ada.lovelace@gmail.com' // search query
},
{
email: 'theoutlander@live.com' // field:values to update
},
{
new: true, // return updated doc
runValidators: true // validate before update
}
)
.then((doc) => {
console.log(doc);
})
.catch((err) => {
console.error(err);
});
```

The document returned will contain the updated email:

```
{
_id: 5a78fe3e2f44ba8f85a2409a,
email: 'theoutlander@live.com',
__v: 0
}
```

Delete Record

We will use the findOneAndRemove call to delete a record. It returns the original document that was removed:

```
EmailModel.findOneAndRemove({
email: 'theoutlander@live.com'
})
.then((response) => {
```

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

Helpers

We have looked at some of the basic functionality above known as CRUD (Create, Read, Update, Delete) operations, but Mongoose also provides the ability to configure several types of helper methods and properties. These can be used to further simplify working with data.

Let’s create a user schema in ./src/models/user.js with the fieldsfirstName and lastName:

```
let mongoose = require('mongoose');
let userSchema = new mongoose.Schema({
firstName: String,
lastName: String
});
module.exports = mongoose.model('User', userSchema);
```

Virtual Property

A virtual property is not persisted to the database. We can add it to our schema as a helper to get and set values.

Let’s create a virtual property called fullName which can be used to set values on firstName and lastName and retrieve them as a combined value when read:

```
userSchema.virtual('fullName').get(function () {
return this.firstName + ' ' + this.lastName;
});
userSchema.virtual('fullName').set(function (name) {
let str = name.split(' ');
```

this.firstName = str[0];

this.lastName = str[1];

});

Callbacks for get and set must use the function keyword as we need to access the model via the this ****keyword. Using fat arrow functions will change what this refers to.

Now, we can set firstName and lastName by assigning a value to fullName:

let model = new UserModel();

model.fullName = 'Thomas Anderson';

console.log(model.toJSON()); // Output model fields as JSON

console.log();

console.log(model.fullName); // Output the full name

The code above will output the following:

{ _id: 5a7a4248550ebb9fafd898cf,

firstName: 'Thomas',

lastName: 'Anderson' }

Thomas Anderson

Instance Methods

We can create custom helper methods on the schema and access them via the model instance. These methods will have access to the model object and they can be used quite creatively. For instance, we could create a method to find all the people who have the same first name as the current instance.

In this example, let’s create a function to return the initials for the current user. Let’s add a custom helper method called getInitials to the schema:

userSchema.methods.getInitials = function () {

return this.firstName[0] + this.lastName[0];

};

This method will be accessible via a model instance:

let model = new UserModel({

```
firstName: 'Thomas',
lastName: 'Anderson'
});
let initials = model.getInitials();
console.log(initials); // This will output: TA
```

Static Methods

Similar to instance methods, we can create static methods on the schema. Let’s create a method to retrieve all users in the database:

```
userSchema.statics.getUsers = function () {
return new Promise((resolve, reject) => {
this.find((err, docs) => {
if (err) {
console.error(err);
return reject(err);
}
resolve(docs);
});
});
};
```

Calling getUsers on the Model class will return all the users in the database:

```
UserModel.getUsers()
.then((docs) => {
console.log(docs);
})
.catch((err) => {
console.error(err);
});
```

Adding instance and static methods is a nice approach to implement an interface to database interactions on collections and records.

Middleware

Middleware are functions that run at specific stages of a pipeline. Mongoose supports middleware for the following operations:

- Aggregate
- Document
- Model
- Query

For instance, models have pre and post functions that take two parameters:

1. Type of event (‘init’, ‘validate’, ‘save’, ‘remove’)
2. A callback that is executed with **this** referencing the model instance

https://cdn-media-1.freecodecamp.org/images/0*iZwmyy25FSxuxXlH.

Example of Middleware (a.k.a. pre and post hooks)

Let’s try an example by adding two fields called createdAt and updatedAt to our schema:

```
let mongoose = require('mongoose');
let userSchema = new mongoose.Schema({
firstName: String,
lastName: String,
createdAt: Date,
updatedAt: Date
});
module.exports = mongoose.model('User', userSchema);
```

When model.save() is called, there is a pre(‘save’, …) and post(‘save’, …) event that is triggered. For the second parameter, you can pass a function that is called when the event is triggered. These functions take a parameter to the next function in the middleware chain.

Let’s add a pre-save hook and set values for createdAt and updatedAt:

```
userSchema.pre('save', function (next) {
let now = Date.now();
this.updatedAt = now;
// Set a value for createdAt only if it is null
if (!this.createdAt) {
this.createdAt = now;
}
// Call the next function in the pre-save chain
next();
});
```

Let’s create and save our model:

```
let UserModel = require('./user');
let model = new UserModel({
fullName: 'Thomas Anderson'
});
msg
.save()
.then((doc) => {
console.log(doc);
})
.catch((err) => {
console.error(err);
});
```

You should see values for createdAt and updatedAt when the record that is created is printed:

```
{ _id: 5a7bbbeebc3b49cb919da675,
firstName: 'Thomas',
lastName: 'Anderson',
updatedAt: 2018-02-08T02:54:38.888Z,
createdAt: 2018-02-08T02:54:38.888Z,
__v: 0 }
```

Plugins

Suppose that we want to track when a record was created and last updated on every collection in our database. Instead of repeating the above process, we can create a plugin and apply it to every schema.

Let’s create a file ./src/model/plugins/timestamp.js and replicate the above functionality as a reusable module:

```
module.exports = function timestamp(schema) {
// Add the two fields to the schema
schema.add({
createdAt: Date,
updatedAt: Date
});
// Create a pre-save hook
schema.pre('save', function (next) {
let now = Date.now();
this.updatedAt = now;
// Set a value for createdAt only if it is null
if (!this.createdAt) {
this.createdAt = now;
}
// Call the next function in the pre-save chain
next();
});
};
```

To use this plugin, we simply pass it to the schemas that should be given this functionality:

```
let timestampPlugin = require('./plugins/timestamp');
emailSchema.plugin(timestampPlugin);
userSchema.plugin(timestampPlugin);
```

Query Building

Mongoose has a very rich API that handles many complex operations supported by MongoDB. Consider a query where we can incrementally build query components.

In this example, we are going to:

1. Find all users
2. Skip the first 100 records
3. Limit the results to 10 records
4. Sort the results by the firstName field
5. Select the firstName
6. Execute that query