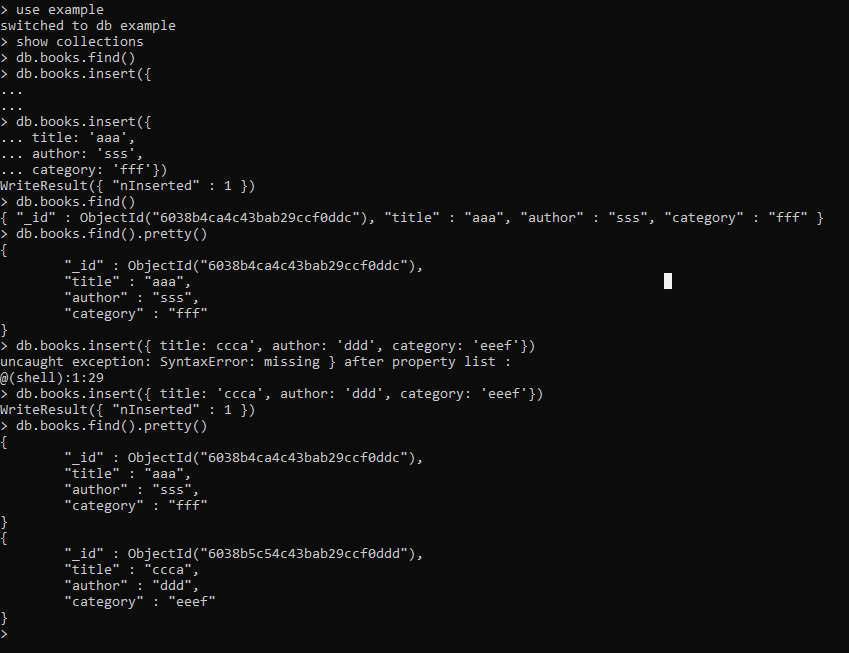
**MongoDB**

Insert data



All types :

**const** schema = **new** Schema({

name: String,

binary: Buffer,

living: Boolean,

updated: { type: Date, default: Date.now },

age: { type: Number, min: 18, max: 65 },

mixed: Schema.Types.Mixed,

\_someId: Schema.Types.ObjectId,

decimal: Schema.Types.Decimal128,

array: [],

ofString: [String],

ofNumber: [Number],

ofDates: [Date],

ofBuffer: [Buffer],

ofBoolean: [Boolean],

ofMixed: [Schema.Types.Mixed],

ofObjectId: [Schema.Types.ObjectId],

ofArrays: [[]],

ofArrayOfNumbers: [[Number]],

nested: {

stuff: { type: String, lowercase: true, trim: true }

},

map: Map,

mapOfString: {

type: Map,

of: String

}

})

*// example use*

**const** Thing = mongoose.model('Thing', schema);

**const** m = **new** Thing;

m.name = 'Statue of Liberty';

m.age = 125;

m.updated = **new** Date;

m.binary = Buffer.alloc(0);

m.living = false;

m.mixed = { any: { thing: 'i want' } };

m.markModified('mixed');

m.\_someId = **new** mongoose.Types.ObjectId;

m.array.push(1);

m.ofString.push("strings!");

m.ofNumber.unshift(1,2,3,4);

m.ofDates.addToSet(**new** Date);

m.ofBuffer.pop();

m.ofMixed = [1, [], 'three', { four: 5 }];

m.nested.stuff = 'good';

m.map = **new** Map([['key', 'value']]);

m.save(callback);

SchemaType Options :

**const** schema1 = **new** Schema({

test: String *// `test` is a path of type String*

});

**const** schema2 = **new** Schema({

*// The `test` object contains the "SchemaType options"*

test: { type: String } *// `test` is a path of type string*

});

want to lowercase a string before saving

**const** schema2 = **new** Schema({

test: {

type: String,

lowercase: true *// Always convert `test` to lowercase*

}

});

The **JSON**. **stringify**() method converts a JavaScript object or value to a **JSON** string,

**All Schema Types**

* required: boolean or function, if true adds a [required validator](https://mongoosejs.com/docs/validation.html#built-in-validators) for this property
* default: Any or function, sets a default value for the path. If the value is a function, the return value of the function is used as the default.
* select: boolean, specifies default [projections](https://docs.mongodb.com/manual/tutorial/project-fields-from-query-results/) for queries
* validate: function, adds a [validator function](https://mongoosejs.com/docs/validation.html#built-in-validators) for this property
* get: function, defines a custom getter for this property using [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).
* set: function, defines a custom setter for this property using [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).
* alias: string, mongoose >= 4.10.0 only. Defines a [virtual](https://mongoosejs.com/docs/guide.html#virtuals) with the given name that gets/sets this path.
* immutable: boolean, defines path as immutable. Mongoose prevents you from changing immutable paths unless the parent document has isNew: true.
* transform: function, Mongoose calls this function when you call [Document#toJSON()](https://mongoosejs.com/docs/api/document.html" \l "document_Document-toJSON) function, including when you [JSON.stringify()](https://thecodebarbarian.com/the-80-20-guide-to-json-stringify-in-javascript) a document.
* **const** numberSchema = **new** Schema({
* integerOnly: {
* type: Number,
* get: v => Math.round(v),
* set: v => Math.round(v),
* alias: 'i'
* }
* });
* **const** Number = mongoose.model('Number', numberSchema);
* **const** doc = **new** Number();
* doc.integerOnly = 2.001;
* doc.integerOnly; *// 2*
* doc.i; *// 2*
* doc.i = 3.001;
* doc.integerOnly; *// 3*
* doc.i; *// 3*

##### Indexes

You can also define [MongoDB indexes](https://docs.mongodb.com/manual/indexes/) using schema type options.

* index: boolean, whether to define an [index](https://docs.mongodb.com/manual/indexes/) on this property.
* unique: boolean, whether to define a [unique index](https://docs.mongodb.com/manual/core/index-unique/) on this property.
* sparse: boolean, whether to define a [sparse index](https://docs.mongodb.com/manual/core/index-sparse/) on this property.

##### String

* lowercase: boolean, whether to always call .toLowerCase() on the value
* uppercase: boolean, whether to always call .toUpperCase() on the value
* trim: boolean, whether to always call .trim() on the value
* match: RegExp, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value matches the given regular expression
* enum: Array, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is in the given array.
* minLength: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value length is not less than the given number
* maxLength: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value length is not greater than the given number
* populate: Object, sets default [populate options](https://mongoosejs.com/docs/populate.html#query-conditions)

##### Number

* min: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is greater than or equal to the given minimum.
* max: Number, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is less than or equal to the given maximum.
* enum: Array, creates a [validator](https://mongoosejs.com/docs/validation.html) that checks if the value is strictly equal to one of the values in the given array.
* populate: Object, sets default [populate options](https://mongoosejs.com/docs/populate.html#query-conditions)

##### Date

* min: Date
* max: Date

#### String

**const** schema1 = **new** Schema({ name: String }); *// name will be cast to string*

**const** schema2 = **new** Schema({ name: 'String' }); *// Equivalent*

**const** Person = mongoose.model('Person', schema2);

**new** Person({ name: 42 }).name; *// "42" as a string*

**new** Person({ name: { toString: () => 42 } }).name; *// "42" as a string*

*// "undefined", will get a cast error if you `save()` this document*

**new** Person({ name: { foo: 42 } }).name;

#### Number

**const** schema1 = **new** Schema({ age: Number }); *// age will be cast to a Number*

**const** schema2 = **new** Schema({ age: 'Number' }); *// Equivalent*

**const** Car = mongoose.model('Car', schema2);

**new** Car({ age: '15' }).age; *// 15 as a Number*

**new** Car({ age: true }).age; *// 1 as a Number*

**new** Car({ age: false }).age; *// 0 as a Number*

**new** Car({ age: { valueOf: () => 83 } }).age; *// 83 as a Number*

#### Dates

 If you must modify Date types using built-in methods, tell mongoose about the change with doc.markModified('pathToYourDate') before saving.

**const** Assignment = mongoose.model('Assignment', { dueDate: Date });

Assignment.findOne(**function** (err, doc) {

doc.dueDate.setMonth(3);

doc.save(callback); *// THIS DOES NOT SAVE YOUR CHANGE*

doc.markModified('dueDate');

doc.save(callback); *// works*

})

#### Buffer

**const** schema1 = **new** Schema({ binData: Buffer }); *// binData will be cast to a Buffer*

**const** schema2 = **new** Schema({ binData: 'Buffer' }); *// Equivalent*

**const** Data = mongoose.model('Data', schema2);

**const** file1 = **new** Data({ binData: 'test'}); *// {"type":"Buffer","data":[116,101,115,116]}*

**const** file2 = **new** Data({ binData: 72987 }); *// {"type":"Buffer","data":[27]}*

**const** file4 = **new** Data({ binData: { type: 'Buffer', data: [1, 2, 3]}}); *// {"type":"Buffer","data":[1,2,3]}*

#### ObjectIds

An [ObjectId](https://docs.mongodb.com/manual/reference/method/ObjectId/) is a special type typically used for unique identifiers. Here's how you declare a schema with a path driver that is an ObjectId:

**const** mongoose = require('mongoose');

**const** carSchema = **new** mongoose.Schema({ driver: mongoose.ObjectId });

**const** Car = mongoose.model('Car', carSchema);

**const** car = **new** Car();

car.driver = **new** mongoose.Types.ObjectId();

**typeof** car.driver; *// 'object'*

car.driver **instanceof** mongoose.Types.ObjectId; *// true*

car.driver.toString(); *// Something like "5e1a0651741b255ddda996c4"*

#### Boolean

Booleans in Mongoose are [plain JavaScript booleans](https://www.w3schools.com/js/js_booleans.asp). By default, Mongoose casts the below values to true:

* true
* 'true'
* 1
* '1'
* 'yes'

Mongoose casts the below values to false:

* false
* 'false'
* 0
* '0'
* 'no'
* **const** M = mongoose.model('Test', **new** Schema({ b: Boolean }));
* console.log(**new** M({ b: 'nay' }).b); *// undefined*
* *// Set { false, 'false', 0, '0', 'no' }*
* console.log(mongoose.Schema.Types.Boolean.convertToFalse);
* mongoose.Schema.Types.Boolean.convertToFalse.add('nay');
* console.log(**new** M({ b: 'nay' }).b); *// false*

#### Arrays

Mongoose supports arrays of [SchemaTypes](https://mongoosejs.com/docs/api.html" \l "schema_Schema.Types) and arrays of [subdocuments](https://mongoosejs.com/docs/subdocs.html). Arrays of SchemaTypes are also called primitive arrays, and arrays of subdocuments are also called document arrays.

**const** ToySchema = **new** Schema({ name: String });

**const** ToyBoxSchema = **new** Schema({

toys: [ToySchema],

buffers: [Buffer],

strings: [String],

numbers: [Number]

*// ... etc*

});

**const** ToyBox = mongoose.model('ToyBox', ToyBoxSchema);

console.log((**new** ToyBox()).toys); *// []*

Note: specifying an empty array is equivalent to Mixed. The following all create arrays of Mixed:

**const** Empty1 = **new** Schema({ any: [] });

**const** Empty2 = **new** Schema({ any: Array });

**const** Empty3 = **new** Schema({ any: [Schema.Types.Mixed] });

**const** Empty4 = **new** Schema({ any: [{}] });

## Maps vs Objects

**const** obj = {

name: 'Jean-Luc Picard',

age: 59,

rank: 'Captain'

};

obj.name; *// 'Jean-Luc Picard'*

**const** map = **new** Map([

*// You define a map via an array of 2-element arrays. The first*

*// element of each nested array is the key, and the 2nd is the value*

['name', 'Jean-Luc Picard'],

['age', 59],

['rank', 'Captain']

]);

*// To get the value associated with a given `key` in a map, you*

*// need to call `map.get(key)`. Using `map.key` will \*\*not\*\* work.*

map.get('name'); *// 'Jean-Luc Picard'*

Suppose you wanted to get Captain Picard's age. With an object, you can use obj.age. With a map, you would use map.get('age').

obj.constructor; *// [Function: Object]*

map.get('constructor'); *// undefined*

**const** map = **new** Map([]);

**const** n1 = **new** Number(5);

**const** n2 = **new** Number(5);

map.set(n1, 'One');

map.set(n2, 'Two');

*// `n1` and `n2` are objects, so `n1 !== n2`. That means the map has*

*// separate keys for `n1` and `n2`.*

map.get(n1); *// 'One'*

map.get(n2); *// 'Two'*

map.get(5); *// undefined*

*// If you were to do this with an object, `n2` would overwrite `n1`*

**const** obj = {};

obj[n1] = 'One';

obj[n2] = 'Two';

obj[n1]; *// 'Two'*

obj[5]; *// 'Two'*

#### Maps

**const** userSchema = **new** Schema({

*// `socialMediaHandles` is a map whose values are strings. A map's*

*// keys are always strings. You specify the type of values using `of`.*

socialMediaHandles: {

type: Map,

of: String

}

});

**const** user = **new** User({

socialMediaHandles: {}

});

*// Good*

user.socialMediaHandles.set('github', 'vkarpov15');

*// Works too*

user.set('socialMediaHandles.twitter', '@code\_barbarian');

*// Bad, the `myspace` property will \*\*not\*\* get saved*

user.socialMediaHandles.myspace = 'fail';

*// 'vkarpov15'*

console.log(user.socialMediaHandles.get('github'));

*// '@code\_barbarian'*

console.log(user.get('socialMediaHandles.twitter'));

*// undefined*

user.socialMediaHandles.github;

*// Will only save the 'github' and 'twitter' properties*

user.save();

### [Schemas](https://mongoosejs.com/docs/schematypes.html#schemas)

To declare a path as another [schema](https://mongoosejs.com/docs/guide.html#definition), set type to the sub-schema's instance.

To set a default value based on the sub-schema's shape, simply set a default value, and the value will be cast based on the sub-schema's definition before being set during document creation.

**const** subSchema = **new** mongoose.Schema({

*// some schema definition here*

});

**const** schema = **new** mongoose.Schema({

data: {

type: subSchema

default: {}

}

});

## [**Connections**](https://mongoosejs.com/docs/connections.html#connections)

You can connect to MongoDB with the mongoose.connect() method.

mongoose.connect('mongodb://localhost:27017/myapp', {useNewUrlParser: true});

This is the minimum needed to connect the myapp database running locally on the default port (27017). If connecting fails on your machine, try using 127.0.0.1 instead of localhost.

You can also specify several more parameters in the uri:

mongoose.connect('mongodb://username:password@host:port/database?options...', {useNewUrlParser: true});

### [option: capped](https://mongoosejs.com/docs/guide.html#capped)

Mongoose supports MongoDBs [capped](http://www.mongodb.org/display/DOCS/Capped+Collections) collections. To specify the underlying MongoDB collection be capped, set the capped option to the maximum size of the collection in [bytes](http://www.mongodb.org/display/DOCS/Capped+Collections#CappedCollections-size.).

**new** Schema({..}, { capped: { size: 1024, max: 1000, autoIndexId: true } });

**const** schema = **new** Schema({

name: String

}, {

capped: { size: 1024 },

bufferCommands: false,

autoCreate: false *// disable `autoCreate` since `bufferCommands` is false*

});

**const** Model = mongoose.model('Test', schema);

*// Explicitly create the collection before using it*

*// so the collection is capped.*

**await** Model.createCollection();

### [Error Handling](https://mongoosejs.com/docs/connections.html#error-handling)

mongoose.connect('mongodb://localhost:27017/test', { useNewUrlParser: true }).

catch(error => handleError(error));

*// Or:*

**try** {

**await** mongoose.connect('mongodb://localhost:27017/test', { useNewUrlParser: true });

} **catch** (error) {

handleError(error);

}

* bufferCommands - This is a mongoose-specific option (not passed to the MongoDB driver) that disables [Mongoose's buffering mechanism](http://mongoosejs.com/docs/faq.html#callback_never_executes)
* user/pass - The username and password for authentication. These options are Mongoose-specific, they are equivalent to the MongoDB driver's auth.user and auth.password options.
* autoIndex - By default, mongoose will automatically build indexes defined in your schema when it connects. This is great for development, but not ideal for large production deployments, because index builds can cause performance degradation. If you set autoIndex to false, mongoose will not automatically build indexes for **any** model associated with this connection.
* dbName - Specifies which database to connect to and overrides any database specified in the connection string. This is useful if you are unable to specify a default database in the connection string like with [some mongodb+srv syntax connections](https://stackoverflow.com/questions/48917591/fail-to-connect-mongoose-to-atlas/48917626#48917626).

Below are some of the options that are important for tuning Mongoose.

* useNewUrlParser - The underlying MongoDB driver has deprecated their current [connection string](https://docs.mongodb.com/manual/reference/connection-string/) parser. Because this is a major change, they added the useNewUrlParser flag to allow users to fall back to the old parser if they find a bug in the new parser. You should set useNewUrlParser: true unless that prevents you from connecting. Note that if you specify useNewUrlParser: true, you **must** specify a port in your connection string, like mongodb://localhost:27017/dbname. The new url parser does *not* support connection strings that do not have a port, like mongodb://localhost/dbname.

**Important in connections**

**const** options = {

useNewUrlParser: true,

useUnifiedTopology: true,

useCreateIndex: true,

useFindAndModify: false,

autoIndex: false, *// Don't build indexes*

poolSize: 10, *// Maintain up to 10 socket connections*

serverSelectionTimeoutMS: 5000, *// Keep trying to send operations for 5 seconds*

socketTimeoutMS: 45000, *// Close sockets after 45 seconds of inactivity*

family: 4 *// Use IPv4, skip trying IPv6*

};

mongoose.connect(uri, options);

## Documents

### Documents vs Models

[Document](https://mongoosejs.com/docs/api.html#Document) and [Model](https://mongoosejs.com/docs/api.html#Model) are distinct classes in Mongoose. The Model class is a subclass of the Document class. When you use the [Model constructor](https://mongoosejs.com/docs/api.html#Model), you create a new document.

**const** MyModel = mongoose.model('Test', **new** Schema({ name: String }));

**const** doc = **new** MyModel();

doc **instanceof** MyModel; *// true*

doc **instanceof** mongoose.Model; *// true*

doc **instanceof** mongoose.Document; *// true*

## **Queries**

Mongoose [models](https://mongoosejs.com/docs/models.html) provide several static helper functions for [CRUD operations](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete). Each of these functions returns a [mongoose Query object](http://mongoosejs.com/docs/api.html#Query).

* [Model.deleteMany()](https://mongoosejs.com/docs/api.html#model_Model.deleteMany)
* [Model.deleteOne()](https://mongoosejs.com/docs/api.html#model_Model.deleteOne)
* [Model.find()](https://mongoosejs.com/docs/api.html#model_Model.find)
* [Model.findById()](https://mongoosejs.com/docs/api.html#model_Model.findById)
* [Model.findByIdAndDelete()](https://mongoosejs.com/docs/api.html#model_Model.findByIdAndDelete)
* [Model.findByIdAndRemove()](https://mongoosejs.com/docs/api.html#model_Model.findByIdAndRemove)
* [Model.findByIdAndUpdate()](https://mongoosejs.com/docs/api.html#model_Model.findByIdAndUpdate)
* [Model.findOne()](https://mongoosejs.com/docs/api.html#model_Model.findOne)
* [Model.findOneAndDelete()](https://mongoosejs.com/docs/api.html#model_Model.findOneAndDelete)
* [Model.findOneAndRemove()](https://mongoosejs.com/docs/api.html#model_Model.findOneAndRemove)
* [Model.findOneAndReplace()](https://mongoosejs.com/docs/api.html#model_Model.findOneAndReplace)
* [Model.findOneAndUpdate()](https://mongoosejs.com/docs/api.html#model_Model.findOneAndUpdate)
* [Model.replaceOne()](https://mongoosejs.com/docs/api.html#model_Model.replaceOne)
* [Model.updateMany()](https://mongoosejs.com/docs/api.html#model_Model.updateMany)
* [Model.updateOne()](https://mongoosejs.com/docs/api.html#model_Model.updateOne)

A mongoose query can be executed in one of two ways. First, if you pass in a callback function, Mongoose will execute the query asynchronously and pass the results to the callback.

*// find each person with a last name matching 'Ghost'*

**const** query = Person.findOne({ 'name.last': 'Ghost' });

*// selecting the `name` and `occupation` fields*

query.select('name occupation');

*// execute the query at a later time*

query.exec(**function** (err, person) {

**if** (err) **return** handleError(err);

*// Prints "Space Ghost is a talk show host."*

console.log('%s %s is a %s.', person.name.first, person.name.last,

person.occupation);

});

### Executing

**const** Person = mongoose.model('Person', yourSchema);

*// find each person with a last name matching 'Ghost', selecting the `name` and `occupation` fields*

Person.findOne({ 'name.last': 'Ghost' }, 'name occupation', **function** (err, person) {

**if** (err) **return** handleError(err);

*// Prints "Space Ghost is a talk show host".*

console.log('%s %s is a %s.', person.name.first, person.name.last,

person.occupation);

});

Mongoose executed the query and passed the results to callback. All callbacks in Mongoose use the pattern: callback(error, result). If an error occurs executing the query, the error parameter will contain an error document, and result will be null. If the query is successful, the error parameter will be null, and the result will be populated with the results of the query.

*// With a JSON doc*

Person.

find({

occupation: /host/,

'name.last': 'Ghost',

age: { $gt: 17, $lt: 66 },

likes: { $in: ['vaporizing', 'talking'] }

}).

limit(10).

sort({ occupation: -1 }).

select({ name: 1, occupation: 1 }).

exec(callback);

*// Using query builder*

Person.

find({ occupation: /host/ }).

where('name.last').equals('Ghost').

where('age').gt(17).lt(66).

where('likes').in(['vaporizing', 'talking']).

limit(10).

sort('-occupation').

select('name occupation').

exec(callback);

### [Queries are Not Promises](https://mongoosejs.com/docs/queries.html#queries-are-not-promises)

Mongoose queries are **not** promises. They have a .then() function for [co](https://www.npmjs.com/package/co) and [async/await](http://thecodebarbarian.com/common-async-await-design-patterns-in-node.js.html) as a convenience. However, unlike promises, calling a query's .then() can execute the query multiple times.

For example, the below code will execute 3 updateMany() calls, one because of the callback, and two because .then() is called twice.

**const** q = MyModel.updateMany({}, { isDeleted: true }, **function**() {

console.log('Update 1');

});

q.then(() => console.log('Update 2'));

q.then(() => console.log('Update 3'));

Don't mix using callbacks and promises with queries, or you may end up with duplicate operations. That's because passing a callback to a query function immediately executes the query, and calling [then()](https://masteringjs.io/tutorials/fundamentals/then) executes the query again.

Mixing promises and callbacks can lead to duplicate entries in arrays. For example, the below code inserts 2 entries into the tags array, \*not just 1.

**const** BlogPost = mongoose.model('BlogPost', **new** Schema({

title: String,

tags: [String]

}));

*// Because there's both `await` \*\*and\*\* a callback, this `updateOne()` executes twice*

*// and thus pushes the same string into `tags` twice.*

**const** update = { $push: { tags: ['javascript'] } };

**await** BlogPost.updateOne({ title: 'Introduction to Promises' }, update, (err, res) => {

console.log(res);

});

\*\*\*\*\*\*\*

There are no joins in MongoDB but sometimes we still want references to documents in other collections. This is where [population](https://mongoosejs.com/docs/populate.html) comes in.

# Query Casting

**const** query = Character.find({ name: 'Jean-Luc Picard' });

query.getFilter(); *// `{ name: 'Jean-Luc Picard' }`*

*// Subsequent chained calls merge new properties into the filter*

query.find({ age: { $gt: 50 } });

query.getFilter(); *// `{ name: 'Jean-Luc Picard', age: { $gt: 50 } }`*

# How to Use findOneAndUpdate() in Mongoose

**const** Character = mongoose.model('Character', **new** mongoose.Schema({

name: String,

age: Number

}));

**await** Character.create({ name: 'Jean-Luc Picard' });

**const** filter = { name: 'Jean-Luc Picard' };

**const** update = { age: 59 };

*// `doc` is the document \_before\_ `update` was applied*

**let** doc = **await** Character.findOneAndUpdate(filter, update);

doc.name; *// 'Jean-Luc Picard'*

doc.age; *// undefined*

doc = **await** Character.findOne(filter);

doc.age; *// 59*

You should set the new option to true to return the document **after** update was applied.

**const** filter = { name: 'Jean-Luc Picard' };

**const** update = { age: 59 };

*// `doc` is the document \_after\_ `update` was applied because of*

*// `new: true`*

**let** doc = **await** Character.findOneAndUpdate(filter, update, {

new: true

});

doc.name; *// 'Jean-Luc Picard'*

doc.age; *// 59*

As an alternative to the new option, you can also use the returnOriginal option. returnOriginal: false is equivalent to new: true.

**const** filter = { name: 'Jean-Luc Picard' };

**const** update = { age: 59 };

*// `doc` is the document \_after\_ `update` was applied because of*

*// `returnOriginal: false`*

**let** doc = **await** Character.findOneAndUpdate(filter, update, {

returnOriginal: false

});

doc.name; *// 'Jean-Luc Picard'*

doc.age; *// 59*

## Atomic Updates

**const** filter = { name: 'Jean-Luc Picard' };

**const** update = { age: 59 };

**let** doc = **await** Character.findOne({ name: 'Jean-Luc Picard' });

*// Document changed in MongoDB, but not in Mongoose*

**await** Character.updateOne(filter, { name: 'Will Riker' });

*// This will update `doc` age to `59`, even though the doc changed.*

doc.age = 59;

**await** doc.save();

doc = **await** Character.findOne();

doc.name; *// Will Riker*

doc.age; *// 59*

## Upsert

Using the upsert option, you can use findOneAndUpdate() as a find-and-[upsert](https://docs.mongodb.com/manual/reference/method/db.collection.update/" \l "db.collection.update) operation. An upsert behaves like a normal findOneAndUpdate() if it finds a document that matches filter. But, if no document matches filter, MongoDB will insert one by combining filter and update as shown below.

**const** filter = { name: 'Will Riker' };

**const** update = { age: 29 };

**await** Character.countDocuments(filter); *// 0*

**let** doc = **await** Character.findOneAndUpdate(filter, update, {

new: true,

upsert: true *// Make this update into an upsert*

});

doc.name; *// Will Riker*

doc.age; *// 29*