<https://mongoosejs.com/docs/populate.html>

There are no joins in MongoDB but sometimes we still want references to documents in other collections. This is where population comes in.

## Populate

MongoDB has the join-like [$lookup](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/) aggregation operator in versions >= 3.2. Mongoose has a more powerful alternative called populate(), which lets you reference documents in other collections.

Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s). We may populate a single document, multiple documents, plain object, multiple plain objects, or all objects returned from a query. Let's look at some examples.

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

**var** Schema = mongoose.Schema;

**var** personSchema = Schema({

\_id: Schema.Types.ObjectId,

name: String,

age: Number,

stories: [{ type: Schema.Types.ObjectId, ref: 'Story' }]

});

**var** storySchema = Schema({

author: { type: Schema.Types.ObjectId, ref: 'Person' },

title: String,

fans: [{ type: Schema.Types.ObjectId, ref: 'Person' }]

});

**var** Story = mongoose.model('Story', storySchema);

**var** Person = mongoose.model('Person', personSchema);

So far we've created two [Models](https://mongoosejs.com/docs/models.html). Our Person model has its stories field set to an array of ObjectIds. The refoption is what tells Mongoose which model to use during population, in our case the Story model. All \_ids we store here must be document \_ids from the Story model.

**Note**: ObjectId, Number, String, and Buffer are valid for use as refs. However, you should use ObjectId unless you are an advanced user and have a good reason for doing so.

### [Saving refs](https://mongoosejs.com/docs/populate.html#saving-refs)

Saving refs to other documents works the same way you normally save properties, just assign the \_id value:

**var** author = **new** Person({

\_id: **new** mongoose.Types.ObjectId(),

name: 'Ian Fleming',

age: 50

});

author.save(**function** (err) {

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

**var** story1 = **new** Story({

title: 'Casino Royale',

author: author.\_id *// assign the \_id from the person*

});

story1.save(**function** (err) {

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

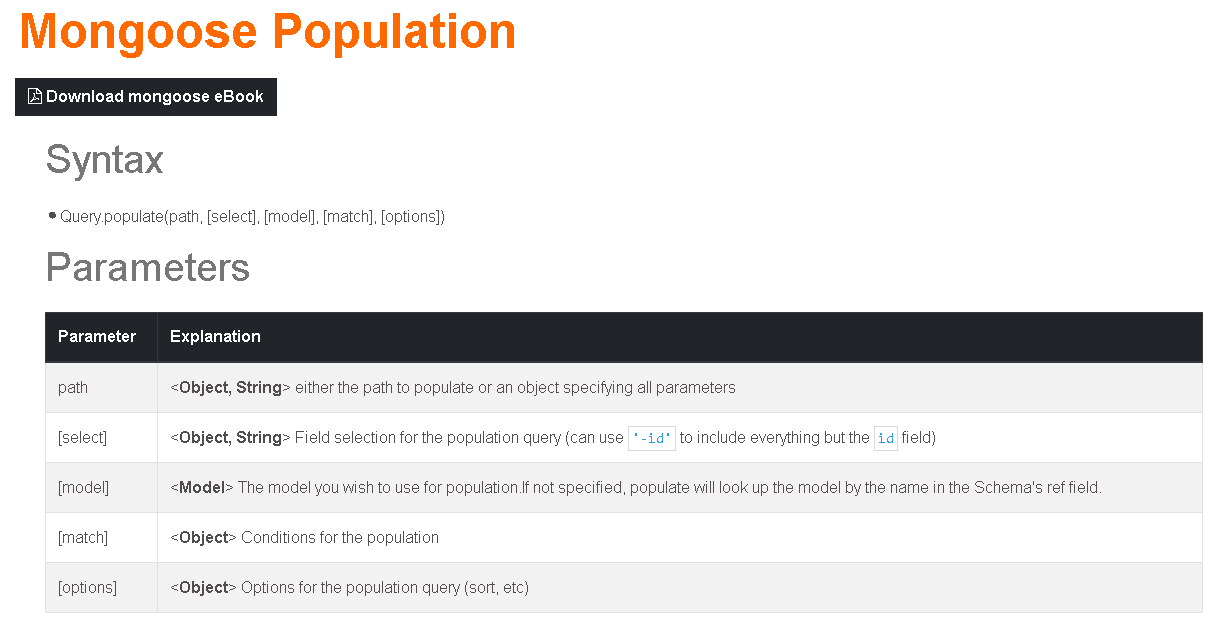
*// thats it!*

});

});

### [Population](https://mongoosejs.com/docs/populate.html#population)

So far we haven't done anything much different. We've merely created a Person and a Story. Now let's take a look at populating our story's author using the query builder:



Story.

findOne({ title: 'Casino Royale' }).

populate('author').

exec(**function** (err, story) {

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

console.log('The author is %s', story.author.name);

*// prints "The author is Ian Fleming"*

});

Populated paths are no longer set to their original \_id , their value is replaced with the mongoose document returned from the database by performing a separate query before returning the results.

Arrays of refs work the same way. Just call the [populate](https://mongoosejs.com/docs/api.html#query_Query-populate) method on the query and an array of documents will be returned in place of the original \_ids.

### [Setting Populated Fields](https://mongoosejs.com/docs/populate.html#setting-populated-fields)

In Mongoose >= 4.0, you can manually populate a field as well.

Story.findOne({ title: 'Casino Royale' }, **function**(error, story) {

**if** (error) {

**return** handleError(error);

}

story.author = author;

console.log(story.author.name); *// prints "Ian Fleming"*

});

### [Field Selection](https://mongoosejs.com/docs/populate.html#field-selection)

What if we only want a few specific fields returned for the populated documents? This can be accomplished by passing the usual [field name syntax](https://mongoosejs.com/docs/api.html#query_Query-select) as the second argument to the populate method:

Story.

findOne({ title: /casino royale/i }).

populate('author', 'name'). *// only return the Persons name*

exec(**function** (err, story) {

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

console.log('The author is %s', story.author.name);

*// prints "The author is Ian Fleming"*

console.log('The authors age is %s', story.author.age);

*// prints "The authors age is null'*

});

### [Populating Multiple Paths](https://mongoosejs.com/docs/populate.html#populating-multiple-paths)

What if we wanted to populate multiple paths at the same time?

Story.

find(...).

populate('fans').

populate('author').

exec();

If you call populate() multiple times with the same path, only the last one will take effect.

*// The 2nd `populate()` call below overwrites the first because they*

*// both populate 'fans'.*

Story.

find().

populate({ path: 'fans', select: 'name' }).

populate({ path: 'fans', select: 'email' });

*// The above is equivalent to:*

Story.find().populate({ path: 'fans', select: 'email' });

### [Query conditions and other options](https://mongoosejs.com/docs/populate.html#query-conditions)

What if we wanted to populate our fans array based on their age, select just their names, and return at most, any 5 of them?

Story.

find(...).

populate({

path: 'fans',

match: { age: { $gte: 21 }},

*// Explicitly exclude `\_id`, see http://bit.ly/2aEfTdB*

select: 'name -\_id',

options: { limit: 5 }

}).

exec();

### [Refs to children](https://mongoosejs.com/docs/populate.html#refs-to-children)

We may find however, if we use the author object, we are unable to get a list of the stories. This is because no story objects were ever 'pushed' onto author.stories.

There are two perspectives here. First, you may want the author know which stories are theirs. Usually, your schema should resolve one-to-many relationships by having a parent pointer in the 'many' side. But, if you have a good reason to want an array of child pointers, you can push() documents onto the array as shown below.

author.stories.push(story1);

author.save(callback);

This allows us to perform a find and populate combo:

Person.

findOne({ name: 'Ian Fleming' }).

populate('stories'). *// only works if we pushed refs to children*

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

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

console.log(person);

});

It is debatable that we really want two sets of pointers as they may get out of sync. Instead we could skip populating and directly find() the stories we are interested in.

Story.

find({ author: author.\_id }).

exec(**function** (err, stories) {

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

console.log('The stories are an array: ', stories);

});

The documents returned from [query population](https://mongoosejs.com/docs/api.html#query_Query-populate) become fully functional, removeable, saveable documents unless the [lean](https://mongoosejs.com/docs/api.html#query_Query-lean) option is specified. Do not confuse them with [sub docs](https://mongoosejs.com/docs/subdocs.html). Take caution when calling its remove method because you'll be removing it from the database, not just the array.

### [Populating an existing document](https://mongoosejs.com/docs/populate.html#populate_an_existing_mongoose_document)

If we have an existing mongoose document and want to populate some of its paths, **mongoose >= 3.6** supports the [document#populate()](https://mongoosejs.com/docs/api.html" \l "document_Document-populate) method.

### [Populating multiple existing documents](https://mongoosejs.com/docs/populate.html#populate_multiple_documents)

If we have one or many mongoose documents or even plain objects (\_like [mapReduce](https://mongoosejs.com/docs/api.html" \l "model_Model.mapReduce) output\_), we may populate them using the [Model.populate()](https://mongoosejs.com/docs/api.html" \l "model_Model.populate) method available in **mongoose >= 3.6**. This is what document#populate() and query#populate() use to populate documents.

### [Populating across multiple levels](https://mongoosejs.com/docs/populate.html#deep-populate)

Say you have a user schema which keeps track of the user's friends.

**var** userSchema = **new** Schema({

name: String,

friends: [{ type: ObjectId, ref: 'User' }]

});

Populate lets you get a list of a user's friends, but what if you also wanted a user's friends of friends? Specify the populate option to tell mongoose to populate the friends array of all the user's friends:

User.

findOne({ name: 'Val' }).

populate({

path: 'friends',

*// Get friends of friends - populate the 'friends' array for every friend*

populate: { path: 'friends' }

});

### [Populating across Databases](https://mongoosejs.com/docs/populate.html#cross-db-populate)

Let's say you have a schema representing events, and a schema representing conversations. Each event has a corresponding conversation thread.

**var** eventSchema = **new** Schema({

name: String,

*// The id of the corresponding conversation*

*// Notice there's no ref here!*

conversation: ObjectId

});

**var** conversationSchema = **new** Schema({

numMessages: Number

});

Also, suppose that events and conversations are stored in separate MongoDB instances.

**var** db1 = mongoose.createConnection('localhost:27000/db1');

**var** db2 = mongoose.createConnection('localhost:27001/db2');

**var** Event = db1.model('Event', eventSchema);

**var** Conversation = db2.model('Conversation', conversationSchema);

In this situation, you will **not** be able to populate() normally. The conversation field will always be null, because populate() doesn't know which model to use. However, [you can specify the model explicitly](https://mongoosejs.com/docs/api.html#model_Model.populate).

Event.

find().

populate({ path: 'conversation', model: Conversation }).

exec(**function**(error, docs) { */\* ... \*/* });

This is known as a "cross-database populate," because it enables you to populate across MongoDB databases and even across MongoDB instances.

### [Dynamic References via refPath](https://mongoosejs.com/docs/populate.html#dynamic-ref)

Mongoose can also populate from multiple collections based on the value of a property in the document. Let's say you're building a schema for storing comments. A user may comment on either a blog post or a product.

**const** commentSchema = **new** Schema({

body: { type: String, required: true },

on: {

type: Schema.Types.ObjectId,

required: true,

*// Instead of a hardcoded model name in `ref`, `refPath` means Mongoose*

*// will look at the `onModel` property to find the right model.*

refPath: 'onModel'

},

onModel: {

type: String,

required: true,

enum: ['BlogPost', 'Product']

}

});

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

**const** BlogPost = mongoose.model('BlogPost', **new** Schema({ title: String }));

**const** Comment = mongoose.model('Comment', commentSchema);

The refPath option is a more sophisticated alternative to ref. If ref is just a string, Mongoose will always query the same model to find the populated subdocs. With refPath, you can configure what model Mongoose uses for each document.

**const** book = **await** Product.create({ name: 'The Count of Monte Cristo' });

**const** post = **await** BlogPost.create({ title: 'Top 10 French Novels' });

**const** commentOnBook = **await** Comment.create({

body: 'Great read',

on: book.\_id,

onModel: 'Product'

});

**const** commentOnPost = **await** Comment.create({

body: 'Very informative',

on: post.\_id,

onModel: 'BlogPost'

});

*// The below `populate()` works even though one comment references the*

*// 'Product' collection and the other references the 'BlogPost' collection.*

**const** comments = **await** Comment.find().populate('on').sort({ body: 1 });

comments[0].on.name; *// "The Count of Monte Cristo"*

comments[1].on.title; *// "Top 10 French Novels"*

An alternative approach is to define separate blogPost and product properties on commentSchema, and then populate() on both properties.

**const** commentSchema = **new** Schema({

body: { type: String, required: true },

product: {

type: Schema.Types.ObjectId,

required: true,

ref: 'Product'

},

blogPost: {

type: Schema.Types.ObjectId,

required: true,

ref: 'BlogPost'

}

});

*// ...*

*// The below `populate()` is equivalent to the `refPath` approach, you*

*// just need to make sure you `populate()` both `product` and `blogPost`.*

**const** comments = **await** Comment.find().

populate('product').

populate('blogPost').

sort({ body: 1 });

comments[0].product.name; *// "The Count of Monte Cristo"*

comments[1].blogPost.title; *// "Top 10 French Novels"*

Defining separate blogPost and product properties works for this simple example. But, if you decide to allow users to also comment on articles or other comments, you'll need to add more properties to your schema. You'll also need an extra populate() call for every property, unless you use [mongoose-autopopulate](https://www.npmjs.com/package/mongoose-autopopulate). Using refPath means you only need 2 schema paths and one populate() call regardless of how many models your commentSchema can point to.

### [Populate Virtuals](https://mongoosejs.com/docs/populate.html#populate-virtuals)

New in 4.5.0

So far you've only populated based on the \_id field. However, that's sometimes not the right choice. In particular, [arrays that grow without bound are a MongoDB anti-pattern](https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/). Using mongoose virtuals, you can define more sophisticated relationships between documents.

**var** PersonSchema = **new** Schema({

name: String,

band: String

});

**var** BandSchema = **new** Schema({

name: String

});

BandSchema.virtual('members', {

ref: 'Person', *// The model to use*

localField: 'name', *// Find people where `localField`*

foreignField: 'band', *// is equal to `foreignField`*

*// If `justOne` is true, 'members' will be a single doc as opposed to*

*// an array. `justOne` is false by default.*

justOne: false,

options: { sort: { name: -1 }, limit: 5 } *// Query options, see http://bit.ly/mongoose-query-options*

});

**var** Person = mongoose.model('Person', PersonSchema);

**var** Band = mongoose.model('Band', BandSchema);

*/\*\**

*\* Suppose you have 2 bands: "Guns N' Roses" and "Motley Crue"*

*\* And 4 people: "Axl Rose" and "Slash" with "Guns N' Roses", and*

*\* "Vince Neil" and "Nikki Sixx" with "Motley Crue"*

*\*/*

Band.find({}).populate('members').exec(**function**(error, bands) {

*/\* `bands.members` is now an array of instances of `Person` \*/*

});

Keep in mind that virtuals are not included in toJSON() output by default. If you want populate virtuals to show up when using functions that rely on JSON.stringify(), like Express' [res.json() function](http://expressjs.com/en/4x/api.html" \l "res.json), set the virtuals: true option on your schema's toJSON options.

*// Set `virtuals: true` so `res.json()` works*

**var** BandSchema = **new** Schema({

name: String

}, { toJSON: { virtuals: true } });

If you're using populate projections, make sure foreignField is included in the projection.

Band.

find({}).

populate({ path: 'members', select: 'name' }).

exec(**function**(error, bands) {

*// Won't work, foreign field `band` is not selected in the projection*

});

Band.

find({}).

populate({ path: 'members', select: 'name band' }).

exec(**function**(error, bands) {

*// Works, foreign field `band` is selected*

});

### [Populate in Middleware](https://mongoosejs.com/docs/populate.html#populate-middleware)

You can populate in either pre or post [hooks](http://mongoosejs.com/docs/middleware.html). If you want to always populate a certain field, check out the [mongoose-autopopulate plugin](http://npmjs.com/package/mongoose-autopopulate).

*// Always attach `populate()` to `find()` calls*

MySchema.pre('find', **function**() {

**this**.populate('user');

});

*// Always `populate()` after `find()` calls. Useful if you want to selectively populate*

*// based on the docs found.*

MySchema.post('find', **async** **function**(docs) {

**for** (**let** doc **of** docs) {

**if** (doc.isPublic) {

**await** doc.populate('user').execPopulate();

}

}

});

*// `populate()` after saving. Useful for sending populated data back to the client in an*

*// update API endpoint*

MySchema.post('save', **function**(doc, next) {

doc.populate('user').execPopulate().then(**function**() {

next();

});

});

# mongoose-populate-example

An example of referencing schema in properties and arrays

When using a NoSQL database like MongoDb, most of the time you'll have documents that contain all properties by itself. But there are also scenarios where you might encounter the need for a more relational approach and need to reference other documents by the ObjectIds.

This post will show you how to deal with these references using Node.js and the [mongoose ODM](http://mongoosejs.com/).

Lets consider we'll have a users collection and a posts collection, thus we'll have a UserSchema as well as a PostSchema. Posts can be written by users and the can by commented by users.

In this example, well reference the users in posts and comments by their ObjectId reference.

The UserSchema is implemented straight forward and looks like this:

var mongoose = require('mongoose');

var UserSchema = new mongoose.Schema({

name: String

});

module.exports = mongoose.model("User", UserSchema);

Beside the title property, the PostSchema also defines the reference by ObjectId for the postedBy property of the PostSchema as well as the postedBy property of the comments inside the comments array property:

var mongoose = require('mongoose');

var PostSchema = new mongoose.Schema({

title: String,

postedBy: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User'

},

comments: [{

text: String,

postedBy: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User'

}

}]

});

module.exports = mongoose.model("Post", PostSchema);

Now lets create two users:

require("./database");

var User = require('./User'),

Post = require('./Post');

var alex = new User({

name: "Alex"

});

var gerry = new User({

name: "Gerry"

})

alex.save();

gerry.save();

The interesting part of course is the creation and even more the query for posts. The post is created with the ObjectId references to the users.

var post = new Post({

title: "Hello World",

postedBy: alex.\_id,

comments: [{

text: "Nice post!",

postedBy: gerry.\_id

}, {

text: "Thanks :)",

postedBy: alex.\_id

}]

})

Now lets save the Post and after it got created, query for all existing Posts.

post.save(function(error) {

if (!error) {

Post.find({})

.populate('postedBy')

.populate('comments.postedBy')

.exec(function(error, posts) {

console.log(JSON.stringify(posts, null, "t"))

})

}

});

As you can see, the we're using the [populate](http://mongoosejs.com/docs/populate.html) function of mongoose to join the documents when querying for Posts. The first call to populate joins the Users for the postedBy property of the posts whereas the second one joins the Users for the comments.

The Post document in the database looks like this:

{

"\_id" : ObjectId("54cd6669d3e0fb1b302e54e6"),

"title" : "Hello World",

"postedBy" : ObjectId("54cd6669d3e0fb1b302e54e4"),

"comments" : [

{

"text" : "Nice post!",

"postedBy" : ObjectId("54cd6669d3e0fb1b302e54e5"),

"\_id" : ObjectId("54cd6669d3e0fb1b302e54e8")

},

{

"text" : "Thanks :)",

"postedBy" : ObjectId("54cd6669d3e0fb1b302e54e4"),

"\_id" : ObjectId("54cd6669d3e0fb1b302e54e7")

}

],

"\_\_v" : 0

}

In contrast, the query result is a full document containing all User references for the Posts.

[

{

"\_id": "54cd6669d3e0fb1b302e54e6",

"title": "Hello World",

"postedBy": {

"\_id": "54cd6669d3e0fb1b302e54e4",

"name": "Alex",

"\_\_v": 0

},

"\_\_v": 0,

"comments": [

{

"text": "Nice post!",

"postedBy": {

"\_id": "54cd6669d3e0fb1b302e54e5",

"name": "Gerry",

"\_\_v": 0

},

"\_id": "54cd6669d3e0fb1b302e54e8"

},

{

"text": "Thanks :)",

"postedBy": {

"\_id": "54cd6669d3e0fb1b302e54e4",

"name": "Alex",

"\_\_v": 0

},

"\_id": "54cd6669d3e0fb1b302e54e7"

}

]

}

]