Populate

MongoDB has the join-like $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, a plain object, multiple plain objects, or all objects returned from a query. Let's look at some examples.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const personSchema = Schema({
  _id: Schema.Types.ObjectId,
  name: String,
  age: Number,
  stories: [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

const storySchema = Schema({
  author: { type: Schema.Types.ObjectId, ref: 'Person' },
  title: String,
  fans: [{ type: Schema.Types.ObjectId, ref: 'Person' }]
});

const Story = mongoose.model('Story', storySchema);
const Person = mongoose.model('Person', personSchema);
```

So far we've created two Models. Our `Person` model has its `stories` field set to an array of `ObjectId`s. The `ref` option is what tells Mongoose which model to use during population, in our case the `Story` model. All `_id`s we store here must be document `_id`s 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

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

```
const author = new Person({
  _id: new mongoose.Types.ObjectId(),
  name: 'Ian Fleming',
  age: 50
});

author.save(function (err) {
  if (err) return handleError(err);

  const story1 = new Story({
    title: 'Casino Royale',
    author: author._id    // assign the _id from the person
  });

  story1.save(function (err) {
    if (err) return handleError(err);
    // that's it!
  });
});
```

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 method on the query and an array of documents will be returned *in place* of the original `_id`s.

Setting Populated Fields

You can manually populate a property by setting it to a document. The document must be an instance of the model your `ref` property refers to.

```
Story.findOne({ title: 'Casino Royale' }, function(error, story) {
  if (error) {
    return handleError(error);
  }
  story.author = author;
  console.log(story.author.name); // prints "Ian Fleming"
});
```

Checking Whether a Field is Populated

You can call the `populated()` function to check whether a field is populated. If `populated()` returns a truthy value, you can assume the field is populated.

```
story.populated('author'); // truthy

story.depopulate('author'); // Make `author` not populated anymore
story.populated('author'); // undefined
```

A common reason for checking whether a path is populated is getting the `author` id. However, for your convenience, Mongoose adds a `_id` getter to ObjectId instances so you can use `story.author._id` regardless of whether `author` is populated.

```
story.populated('author'); // truthy
story.author._id; // ObjectId

story.depopulate('author'); // Make `author` not populated anymore
story.populated('author'); // undefined

story.author instanceof ObjectId; // true
story.author._id; // ObjectId, because Mongoose adds a special getter
```

What If There's No Foreign Document?

Mongoose populate doesn't behave like conventional SQL joins. When there's no document, `story.author` will be `null`. This is analogous to a left join in SQL.

```
await Person.deleteMany({ name: 'Ian Fleming' });

const story = await Story.findOne({ title: 'Casino Royale' }).populate('author');
story.author; // `null`
```

If you have an array of `authors` in your `storySchema`, `populate()` will give you an empty array instead.

```
const storySchema = Schema({
  authors: [{ type: Schema.Types.ObjectId, ref: 'Person' }],
  title: String
});

// Later

const story = await Story.findOne({ title: 'Casino Royale' }).populate('authors');
story.authors; // `[]`
```

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 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

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

What if we wanted to populate our fans array based on their age and select just their names?

```
Story.
  find().
  populate({
    path: 'fans',
    match: { age: { $gte: 21 } },
    // Explicitly exclude `_id`, see http://bit.ly/2aEfTdB
    select: 'name -_id'
  }).
  exec();
```

The `match` option doesn't filter out `Story` documents. If there are no documents that satisfy `match`, you'll get a `Story` document with an empty `fans` array.

For example, suppose you `populate()` a story's `author` and the `author` doesn't satisfy `match`. Then the story's `author` will be `null`.

```
const story = await Story.
  findOne({ title: 'Casino Royale' }).
  populate({ path: 'author', name: { $ne: 'Ian Fleming' } }).
  exec();
story.author; // `null`
```

In general, there is no way to make `populate()` filter stories based on properties of the story's `author`. For example, the below query won't return any results, even though `author` is populated.

```
const story = await Story.
  findOne({ 'author.name': 'Ian Fleming' }).
  populate('author').
  exec();
story; // null
```

If you want to filter stories by their author's name, you should use denormalization.

limit vs. perDocumentLimit

Populate does support a `limit` option, however, it currently does **not** limit on a per-document basis for backwards compatibility. For example, suppose you have 2 stories:

```
Story.create([
  { title: 'Casino Royale', fans: [1, 2, 3, 4, 5, 6, 7, 8] },
  { title: 'Live and Let Die', fans: [9, 10] }
]);
```

If you were to `populate()` using the `limit` option, you would find that the 2nd story has 0 fans:

```
const stories = Story.find().populate({
  path: 'fans',
  options: { limit: 2 }
});

stories[0].name; // 'Casino Royale'
stories[0].fans.length; // 2
```

```
// 2nd story has 0 fans!
stories[1].name; // 'Live and Let Die'
stories[1].fans.length; // 0
```

That's because, in order to avoid executing a separate query for each document, Mongoose instead queries for fans using `numDocuments * limit` as the limit. If you need the correct `limit`, you should use the `perDocumentLimit` option (new in Mongoose 5.9.0). Just keep in mind that `populate()` will execute a separate query for each story, which may cause `populate()` to be slower.

```
const stories = await Story.find().populate({
  path: 'fans',
  // Special option that tells Mongoose to execute a separate query
  // for each `story` to make sure we get 2 fans for each story.
  perDocumentLimit: 2
});

stories[0].name; // 'Casino Royale'
stories[0].fans.length; // 2

stories[1].name; // 'Live and Let Die'
stories[1].fans.length; // 2
```

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` to 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 become fully functional, `remove`able, `save`able documents unless the lean option is specified. Do not confuse them with sub docs. Take caution when calling its remove method because you'll be removing it from the database, not just the array.

Populating an existing document

If you have an existing mongoose document and want to populate some of its paths, you can use the Document#populate() method.

```
const person = await Person.findOne({ name: 'Ian Fleming' });

person.populated('stories'); // null

// Call the `populate()` method on a document to populate a path.
await person.populate('stories');

person.populated('stories'); // Array of ObjectIds
person.stories[0].name; // 'Casino Royale'
```

The `Document#populate()` method does not support chaining. You need to call `populate()` multiple times, or with an array of paths, to populate multiple paths

```
await person.populate(['stories', 'fans']);
person.populated('fans'); // Array of ObjectIds
```

Populating multiple existing documents

If we have one or many mongoose documents or even plain objects (*like mapReduce output*), we may populate them using the Model.populate() method. This is what `Document#populate()` and `Query#populate()` use to populate documents.

Populating across multiple levels

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

```
const 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' }
  });
```

Cross Database Populate

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

```
const db1 = mongoose.createConnection('mongodb://localhost:27000/db1');
const db2 = mongoose.createConnection('mongodb://localhost:27001/db2');

const conversationSchema = new Schema({ numMessages: Number });
const Conversation = db2.model('Conversation', conversationSchema);

const eventSchema = new Schema({
  name: String,
  conversation: {
    type: ObjectId,
    ref: Conversation // `ref` is a **Model class**, not a string
  }
});
const Event = db1.model('Event', eventSchema);
```

In the above example, events and conversations are stored in separate MongoDB databases. String `ref` will not work in this situation, because Mongoose assumes a string `ref` refers to a model name on the same connection. In the above example, the conversation model is registered on `db2`, not `db1`.

```
// Works
const events = await Event.
  find().
  populate('conversation');
```

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

If you don't have access to the model instance when defining your `eventSchema`, you can also pass the model instance as an option to `populate()`.

```
const events = await Event.
  find().
  // The `model` option specifies the model to use for populating.
  populate({ path: 'conversation', model: Conversation });
```

Dynamic References via `refPath`

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. 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

So far you've only populated based on the `_id` field. However, that's sometimes not the right choice. For example, suppose you have 2 models: `Author` and `BlogPost`.

```
const AuthorSchema = new Schema({
  name: String,
  posts: [{ type: mongoose.Schema.Types.ObjectId, ref: 'BlogPost' }]
});

const BlogPostSchema = new Schema({
  title: String,
  comments: [{
    author: { type: mongoose.Schema.Types.ObjectId, ref: 'Author' },
    content: String
  }]
});

const Author = mongoose.model('Author', AuthorSchema, 'Author');
const BlogPost = mongoose.model('BlogPost', BlogPostSchema, 'BlogPost');
```

The above is an example of **bad schema design**. Why? Suppose you have an extremely prolific author that writes over 10k blog posts. That `author` document will be huge, over 12kb, and large documents lead to performance issues on both server and client. The Principle of Least Cardinality states that one-to-many relationships, like author to blog post, should be stored on the "many" side. In other words, blog posts should store their `author`, authors should **not** store all their `posts`.

```
const AuthorSchema = new Schema({
  name: String
});

const BlogPostSchema = new Schema({
  title: String,
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'Author' },
  comments: [{
    author: { type: mongoose.Schema.Types.ObjectId, ref: 'Author' },
    content: String
  }]
});
```

Unfortunately, these two schemas, as written, don't support populating an author's list of blog posts. That's where *virtual populate* comes in. Virtual populate means calling `populate()` on a virtual property that has a `ref` option as shown below.

```
// Specifying a virtual with a `ref` property is how you enable virtual
// population
AuthorSchema.virtual('posts', {
  ref: 'BlogPost',
  localField: '_id',
  foreignField: 'author'
});

const Author = mongoose.model('Author', AuthorSchema, 'Author');
const BlogPost = mongoose.model('BlogPost', BlogPostSchema, 'BlogPost');
```

You can then `populate()` the author's `posts` as shown below.

```
const author = await Author.findOne().populate('posts');

author.posts[0].title; // Title of the first blog post
```

Keep in mind that virtuals are *not* included in `toJSON()` and `toObject()` output by default. If you want populate virtuals to show up when using functions like Express' `res.json()` function or `console.log()`, set the `virtuals: true` option on your schema's `toJSON` and `toObject()` options.

```
const authorSchema = new Schema({ name: String }, {
  toJSON: { virtuals: true }, // So `res.json()` and other `JSON.stringify()` functions include vi
  toObject: { virtuals: true } // So `console.log()` and other functions that use `toObject()` inc
});
```

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

```
let authors = await Author.
  find({}).
  // Won't work because the foreign field `author` is not selected
  populate({ path: 'posts', select: 'title' }).
  exec();
```

```
authors = await Author.
  find({}).
  // Works, foreign field `author` is selected
  populate({ path: 'posts', select: 'title author' }).
  exec();
```

Populate Virtuals: The Count Option

Populate virtuals also support counting the number of documents with matching `foreignField` as opposed to the documents themselves. Set the `count` option on your virtual:

```
const PersonSchema = new Schema({
  name: String,
  band: String
});

const BandSchema = new Schema({
  name: String
});
BandSchema.virtual('numMembers', {
  ref: 'Person', // The model to use
  localField: 'name', // Find people where `localField`
  foreignField: 'band', // is equal to `foreignField`
  count: true // And only get the number of docs
});

// Later
const doc = await Band.findOne({ name: 'Motley Crue' }).
  populate('numMembers');
doc.numMembers; // 2
```

Populating Maps

Maps are a type that represents an object with arbitrary string keys. For example, in the below schema, `members` is a map from strings to ObjectIds.

```
const BandSchema = new Schema({
  name: String,
  members: {
    type: Map,
    of: {
      type: 'ObjectId',
      ref: 'Person'
    }
  }
});
const Band = mongoose.model('Band', bandSchema);
```

This map has a `ref`, which means you can use `populate()` to populate all the ObjectIds in the map. Suppose you have the below `band` document:

```
const person1 = new Person({ name: 'Vince Neil' });
const person2 = new Person({ name: 'Mick Mars' });

const band = new Band({
  name: 'Motley Crue',
  members: {
    'singer': person1._id,
    'guitarist': person2._id
  }
});
```

You can `populate()` every element in the map by populating the special path `members.$*`. `$*` is a special syntax that tells Mongoose to look at every key in the map.

```
const band = await Band.findOne({ name: 'Motley Crue' }).populate('members.$*');

band.members.get('singer'); // { _id: ..., name: 'Vince Neil' }
```

You can also populate paths in maps of subdocuments using `$*`. For example, suppose you have the below `librarySchema`:

```
const librarySchema = new Schema({
  name: String,
  books: {
    type: Map,
    of: new Schema({
      title: String,
      author: {
        type: 'ObjectId',
        ref: 'Person'
      }
    })
  }
});
const Library = mongoose.model('Library, librarySchema');
```

You can `populate()` every book's author by populating `books.$*.author`:

```
const libraries = await Library.find().populate('books.$*.author');
```

Populate in Middleware

You can populate in either pre or post hooks. If you want to always populate a certain field, check out the mongoose-autopopulate plugin.

```
// 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');
    }
  }
});
```

```
// `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').then(function() {
    next();
  });
});
```

Next Up

Now that we've covered `populate()`, let's take a look at discriminators.