Table of Contents

Schemas

If you haven't yet done so, please take a minute to read the quickstart to get an idea of how Mongoose works. If you are migrating from 4.x to 5.x please take a moment to read the migration guide.

Defining your schema

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

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

 var blogSchema = new Schema({
   title:  String, // String is shorthand for {type: String}
   author: String,
   body:   String,
   comments: [{ body: String, date: Date }],
   date: { type: Date, default: Date.now },
   hidden: Boolean,
   meta: {
     votes: Number,
     favs:  Number
   }
 });
```

If you want to add additional keys later, use the Schema#add method.

Each key in our code `blogSchema` defines a property in our documents which will be cast to its associated SchemaType. For example, we've defined a property `title` which will be cast to the String SchemaType and property `date` which will be cast to a `Date` SchemaType.

Notice above that if a property only requires a type, it can be specified using a shorthand notation (contrast the `title` property above with the `date` property).

Keys may also be assigned nested objects containing further key/type definitions like the `meta` property above. This will happen whenever a key's value is a POJO that lacks a bona-fide `type` property. In these cases, only the leaves in a tree are given actual paths in the schema (like `meta.votes` and `meta.favs` above), and the branches do not have actual paths. A side-effect of this is that `meta` above cannot have its own validation. If validation is needed up the tree, a path needs to be created up the tree - see the Subdocuments section for more information on how to do this. Also read the Mixed subsection of the SchemaTypes guide for some gotchas.

The permitted SchemaTypes are:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

Read more about SchemaTypes here.

Schemas not only define the structure of your document and casting of properties, they also define document instance methods, static Model methods, compound indexes, and document lifecycle hooks called middleware.

Creating a model

To use our schema definition, we need to convert our `blogSchema` into a Model we can work with. To do so, we pass it into `mongoose.model(modelName, schema)`:

```
var Blog = mongoose.model('Blog', blogSchema);
 // ready to go!
```

Ids

By default, Mongoose adds an `_id` property to your schemas.

```
const schema = new Schema();

schema.path('_id'); // ObjectId { ... }
```

When you create a new document with the automatically added `_id` property, Mongoose creates a new `_id` of type ObjectId to your document.

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

const doc = new Model();
doc._id instanceof mongoose.Types.ObjectId; // true
```

You can also overwrite Mongoose's default `_id` with your own `_id`. Just be careful: Mongoose will refuse to save a document that doesn't have an `_id`, so you're responsible for setting `_id` if you define your own `_id` path.

```
const schema = new Schema({ _id: Number });
const Model = mongoose.model('Test', schema);

const doc = new Model();
await doc.save(); // Throws "document must have an _id before saving"

doc._id = 1;
await doc.save(); // works
```

Instance methods

Instances of `Models` are documents. Documents have many of their own built-in instance methods. We may also define our own custom document instance methods.

```
// define a schema
  var animalSchema = new Schema({ name: String, type: String });

  // assign a function to the "methods" object of our animalSchema
  animalSchema.methods.findSimilarTypes = function(cb) {
    return mongoose.model('Animal').find({ type: this.type }, cb);
  };
```

Now all of our `animal` instances have a `findSimilarTypes` method available to them.

```
var Animal = mongoose.model('Animal', animalSchema);
 var dog = new Animal({ type: 'dog' });

 dog.findSimilarTypes(function(err, dogs) {
   console.log(dogs); // woof
 });
```

- Overwriting a default mongoose document method may lead to unpredictable results. See this for more details.
- The example above uses the `Schema.methods` object directly to save an instance method. You can also use the `Schema.method()` helper as described here.
- Do **not** declare methods using ES6 arrow functions (`=>`). Arrow functions explicitly prevent binding `this`, so your method will **not** have access to the document and the above examples will not work.

Statics

You can also add static functions to your model. There are two equivalent ways to add a static:

- Add a function property to `schema.statics`
- Call the `Schema#static()` function

```
// Assign a function to the "statics" object of our animalSchema
 animalSchema.statics.findByName = function(name) {
   return this.find({ name: new RegExp(name, 'i') });
 };
 // Or, equivalently, you can call `animalSchema.static()`.
 animalSchema.static('findByBreed', function(breed) {
   return this.find({ breed });
 });

 const Animal = mongoose.model('Animal', animalSchema);
 let animals = await Animal.findByName('fido');
 animals = animals.concat(await Animal.findByBreed('Poodle'));
```

Do **not** declare statics using ES6 arrow functions (`=>`). Arrow functions explicitly prevent binding `this`, so the above examples will not work because of the value of `this`.

Query Helpers

You can also add query helper functions, which are like instance methods but for mongoose queries. Query helper methods let you extend mongoose's chainable query builder API.

```
animalSchema.query.byName = function(name) {
  return this.where({ name: new RegExp(name, 'i') });
};

var Animal = mongoose.model('Animal', animalSchema);

Animal.find().byName('fido').exec(function(err, animals) {
  console.log(animals);
});

Animal.findOne().byName('fido').exec(function(err, animal) {
  console.log(animal);
});
```

Indexes

MongoDB supports secondary indexes. With mongoose, we define these indexes within our `Schema` at the path level or the `schema` level. Defining indexes at the schema level is necessary when creating compound indexes.

```
var animalSchema = new Schema({
  name: String,
  type: String,
  tags: { type: [String], index: true } // field level
});

animalSchema.index({ name: 1, type: -1 }); // schema level
```

When your application starts up, Mongoose automatically calls `createIndex` for each defined index in your schema. Mongoose will call `createIndex` for each index sequentially, and emit an 'index' event on the model when all the `createIndex` calls succeeded or when there was an error. While nice for development, it is recommended this behavior be disabled in production since index creation can cause a significant performance impact. Disable the behavior by setting the `autoIndex` option of your schema to `false`, or globally on the connection by setting the option `autoIndex` to `false`.

```
mongoose.connect('mongodb://user:pass@localhost:port/database', { autoIndex: false
});
  // or
  mongoose.createConnection('mongodb://user:pass@localhost:port/database', {
autoIndex: false });
```

```
// or
animalSchema.set('autoIndex', false);
// or
new Schema({..}, { autoIndex: false });
```

Mongoose will emit an `index` event on the model when indexes are done building or an error occurred.

```
// Will cause an error because mongodb has an _id index by default that
// is not sparse
animalSchema.index({ _id: 1 }, { sparse: true });
var Animal = mongoose.model('Animal', animalSchema);

Animal.on('index', function(error) {
  // "_id index cannot be sparse"
  console.log(error.message);
});
```

See also the Model#ensureIndexes method.

Virtuals

Virtuals are document properties that you can get and set but that do not get persisted to MongoDB. The getters are useful for formatting or combining fields, while setters are useful for de-composing a single value into multiple values for storage.

```
// define a schema
var personSchema = new Schema({
  name: {
    first: String,
    last: String
  }
});

// compile our model
var Person = mongoose.model('Person', personSchema);

// create a document
var axl = new Person({
  name: { first: 'Axl', last: 'Rose' }
});
```

Suppose you want to print out the person's full name. You could do it yourself:

```
console.log(axl.name.first + ' ' + axl.name.last); // Axl Rose
```

But concatenating the first and last name every time can get cumbersome. And what if you want to do some extra processing on the name, like removing diacritics?

A virtual property getter lets you define a `fullName` property that won't get persisted to MongoDB.

```
personSchema.virtual('fullName').get(function () {
  return this.name.first + ' ' + this.name.last;
});
```

Now, mongoose will call your getter function every time you access the `fullName` property:

```
console.log(axl.fullName); // Axl Rose
```

If you use `toJSON()` or `toObject()` mongoose will *not* include virtuals by default. This includes the output of calling `JSON.stringify()` on a Mongoose document, because `JSON.stringify()` calls `toJSON()`. Pass `{ virtuals: true }` to either `toObject()` or `toJSON()`.

You can also add a custom setter to your virtual that will let you set both first name and last name via the `fullName` virtual.

```
personSchema.virtual('fullName').
  get(function() { return this.name.first + ' ' + this.name.last; }).
  set(function(v) {
    this.name.first = v.substr(0, v.indexOf(' '));
    this.name.last = v.substr(v.indexOf(' ') + 1);
  });

axl.fullName = 'William Rose'; // Now `axl.name.first` is "William"
```

Virtual property setters are applied before other validation. So the example above would still work even if the `first` and `last` name fields were required.

Only non-virtual properties work as part of queries and for field selection. Since virtuals are not stored in MongoDB, you can't query with them.

Aliases

Aliases are a particular type of virtual where the getter and setter seamlessly get and set another property. This is handy for saving network bandwidth, so you can convert a short property name stored in the database into a longer name for code readability.

```
var personSchema = new Schema({
  n: {
    type: String,
    // Now accessing `name` will get you the value of `n`, and setting `n` will set
the value of `name`
    alias: 'name'
  }
});
```

```
// Setting `name` will propagate to `n`
var person = new Person({ name: 'Val' });
console.log(person); // { n: 'Val' }
console.log(person.toObject({ virtuals: true })); // { n: 'Val', name: 'Val' }
console.log(person.name); // "Val"

person.name = 'Not Val';
console.log(person); // { n: 'Not Val' }
```

You can also declare aliases on nested paths. It is easier to use nested schemas and subdocuments, but you can also declare nested path aliases inline as long as you use the full nested path `nested.myProp` as the alias.

```
const childSchema = new Schema({
  n: {
    type: String,
    alias: 'name'
  }
}, { _id: false });

const parentSchema = new Schema({
  // If in a child schema, alias doesn't need to include the full nested path
  c: childSchema,
  name: {
    f: {
      type: String,
      // Alias needs to include the full nested path if declared inline
      alias: 'name.first'
    }
  }
});
```

Options

Schemas have a few configurable options which can be passed to the constructor or to the `set` method:

```
new Schema({..}, options);

// or

var schema = new Schema({..});
schema.set(option, value);
```

Valid options:

- autoIndex

- autoCreate
- bufferCommands
- capped
- collection
- id
- _id
- minimize
- read
- writeConcern
- shardKey
- strict
- strictQuery
- toJSON
- toObject
- typeKey
- useNestedStrict
- validateBeforeSave
- versionKey
- collation
- selectPopulatedPaths
- skipVersioning
- timestamps
- storeSubdocValidationError

option: autoIndex

By default, Mongoose's `init()` function creates all the indexes defined in your model's schema by calling `Model.createIndexes()` after you successfully connect to MongoDB. Creating indexes automatically is great for development and test environments. But index builds can also create significant load on your production database. If you want to manage indexes carefully in production, you can set `autoIndex` to false.

```
const schema = new Schema({..}, { autoIndex: false });
const Clock = mongoose.model('Clock', schema);
Clock.ensureIndexes(callback);
```

The `autoIndex` option is set to `true` by default. You can change this default by setting `mongoose.set('autoIndex', false);`

option: autoCreate

Before Mongoose builds indexes, it calls `Model.createCollection()` to create the underlying collection in MongoDB if `autoCreate` is set to true. Calling `createCollection()` sets the collection's default collation based on the collation option and establishes the collection as a capped collection if you set the `capped` schema option. Like `autoIndex`, setting `autoCreate` to true is helpful for development and test environments.

Unfortunately, `createCollection()` cannot change an existing collection. For example, if you add `capped: 1024` to your schema and the existing collection is not capped, `createCollection()` will throw an error. Generally, `autoCreate` should be `false` for production environments.

```
const schema = new Schema({..}, { autoCreate: true, capped: 1024 });
const Clock = mongoose.model('Clock', schema);
// Mongoose will create the capped collection for you.
```

Unlike `autoIndex`, `autoCreate` is `false` by default. You can change this default by setting `mongoose.set('autoCreate', true);`

option: bufferCommands

By default, mongoose buffers commands when the connection goes down until the driver manages to reconnect. To disable buffering, set `bufferCommands` to false.

```
var schema = new Schema({..}, { bufferCommands: false });
```

The schema `bufferCommands` option overrides the global `bufferCommands` option.

```
mongoose.set('bufferCommands', true);
// Schema option below overrides the above, if the schema option is set.
var schema = new Schema({..}, { bufferCommands: false });
```

option: capped

Mongoose supports MongoDBs capped collections. To specify the underlying MongoDB collection be `capped`, set the `capped` option to the maximum size of the collection in bytes.

```
new Schema({..}, { capped: 1024 });
```

The `capped` option may also be set to an object if you want to pass additional options like max or autoIndexId. In this case you must explicitly pass the `size` option, which is required.

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

option: collection

Mongoose by default produces a collection name by passing the model name to the utils.toCollectionName method. This method pluralizes the name. Set this option if you need a different name for your collection.

```
var dataSchema = new Schema({..}, { collection: 'data' });
```

option: id

Mongoose assigns each of your schemas an `id` virtual getter by default which returns the document's `_id` field cast to a string, or in the case of ObjectIds, its hexString. If you don't want an `id` getter added to your schema, you may disable it by passing this option at schema construction time.

```
// default behavior
var schema = new Schema({ name: String });
var Page = mongoose.model('Page', schema);
var p = new Page({ name: 'mongodb.org' });
console.log(p.id); // '50341373e894ad16347efe01'

// disabled id
var schema = new Schema({ name: String }, { id: false });
var Page = mongoose.model('Page', schema);
var p = new Page({ name: 'mongodb.org' });
console.log(p.id); // undefined
```

option: _id

Mongoose assigns each of your schemas an `_id` field by default if one is not passed into the Schema constructor. The type assigned is an ObjectId to coincide with MongoDB's default behavior. If you don't want an `_id` added to your schema at all, you may disable it using this option.

You can **only** use this option on subdocuments. Mongoose can't save a document without knowing its id, so you will get an error if you try to save a document without an `_id`.

```
// default behavior
var schema = new Schema({ name: String });
var Page = mongoose.model('Page', schema);
var p = new Page({ name: 'mongodb.org' });
console.log(p); // { _id: '50341373e894ad16347efe01', name: 'mongodb.org' }

// disabled _id
var childSchema = new Schema({ name: String }, { _id: false });
var parentSchema = new Schema({ children: [childSchema] });

var Model = mongoose.model('Model', parentSchema);
```

```
Model.create({ children: [{ name: 'Luke' }] }, function(error, doc) {
  // doc.children[0]._id will be undefined
});
```

option: minimize

Mongoose will, by default, "minimize" schemas by removing empty objects.

```
const schema = new Schema({ name: String, inventory: {} });
const Character = mongoose.model('Character', schema);

// will store `inventory` field if it is not empty
const frodo = new Character({ name: 'Frodo', inventory: { ringOfPower: 1 }});
await frodo.save();
let doc = await Character.findOne({ name: 'Frodo' }).lean();
doc.inventory; // { ringOfPower: 1 }

// will not store `inventory` field if it is empty
const sam = new Character({ name: 'Sam', inventory: {}});
await sam.save();
doc = await Character.findOne({ name: 'Sam' }).lean();
doc.inventory; // undefined
```

This behavior can be overridden by setting `minimize` option to `false`. It will then store empty objects.

```
const schema = new Schema({ name: String, inventory: {} }, { minimize: false });
const Character = mongoose.model('Character', schema);

// will store `inventory` if empty
const sam = new Character({ name: 'Sam', inventory: {} });
await sam.save();
doc = await Character.findOne({ name: 'Sam' }).lean();
doc.inventory; // {}
```

To check whether an object is empty, you can use the `$isEmpty()` helper:

```
const sam = new Character({ name: 'Sam', inventory: {} });
sam.$isEmpty('inventory'); // true

sam.inventory.barrowBlade = 1;
sam.$isEmpty('inventory'); // false
```

option: read

Allows setting query#read options at the schema level, providing us a way to apply default ReadPreferences to all queries derived from a model.

```
var schema = new Schema({..}, { read: 'primary' });           // also aliased as
'p'
var schema = new Schema({..}, { read: 'primaryPreferred' });   // aliased as 'pp'
var schema = new Schema({..}, { read: 'secondary' });         // aliased as 's'
var schema = new Schema({..}, { read: 'secondaryPreferred' }); // aliased as 'sp'
var schema = new Schema({..}, { read: 'nearest' });           // aliased as 'n'
```

The alias of each pref is also permitted so instead of having to type out 'secondaryPreferred' and getting the spelling wrong, we can simply pass 'sp'.

The read option also allows us to specify *tag sets*. These tell the driver from which members of the replica-set it should attempt to read. Read more about tag sets here and here.

NOTE: you may also specify the driver read pref strategy option when connecting:

```
// pings the replset members periodically to track network latency
var options = { replset: { strategy: 'ping' }};
mongoose.connect(uri, options);

var schema = new Schema({..}, { read: ['nearest', { disk: 'ssd' }] });
mongoose.model('JellyBean', schema);
```

option: writeConcern

Allows setting write concern at the schema level.

```
const schema = new Schema({ name: String }, {
  writeConcern: {
    w: 'majority',
    j: true,
    wtimeout: 1000
  }
});
```

option: shardKey

The `shardKey` option is used when we have a sharded MongoDB architecture. Each sharded collection is given a shard key which must be present in all insert/update operations. We just need to set this schema option to the same shard key and we'll be all set.

```
new Schema({ .. }, { shardKey: { tag: 1, name: 1 }})
```

Note that Mongoose does not send the `shardcollection` command for you. You must configure your shards yourself.

option: strict

The strict option, (enabled by default), ensures that values passed to our model constructor that were not specified in our schema do not get saved to the db.

```
var thingSchema = new Schema({..})
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing({ iAmNotInTheSchema: true });
thing.save(); // iAmNotInTheSchema is not saved to the db

// set to false..
var thingSchema = new Schema({..}, { strict: false });
var thing = new Thing({ iAmNotInTheSchema: true });
thing.save(); // iAmNotInTheSchema is now saved to the db!!
```

This also affects the use of `doc.set()` to set a property value.

```
var thingSchema = new Schema({..})
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing;
thing.set('iAmNotInTheSchema', true);
thing.save(); // iAmNotInTheSchema is not saved to the db
```

This value can be overridden at the model instance level by passing a second boolean argument:

```
var Thing = mongoose.model('Thing');
var thing = new Thing(doc, true);  // enables strict mode
var thing = new Thing(doc, false); // disables strict mode
```

The `strict` option may also be set to `"throw"` which will cause errors to be produced instead of dropping the bad data.

NOTE: Any key/val set on the instance that does not exist in your schema is always ignored, regardless of schema option.

```
var thingSchema = new Schema({..})
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing;
thing.iAmNotInTheSchema = true;
thing.save(); // iAmNotInTheSchema is never saved to the db
```

option: strictQuery

For backwards compatibility, the `strict` option does **not** apply to the `filter` parameter for queries.

```
const mySchema = new Schema({ field: Number }, { strict: true });
const MyModel = mongoose.model('Test', mySchema);

// Mongoose will **not** filter out `notInSchema: 1`, despite `strict: true`
MyModel.find({ notInSchema: 1 });
```

The `strict` option does apply to updates.

```
// Mongoose will strip out `notInSchema` from the update if `strict` is
// not `false`
MyModel.updateMany({}, { $set: { notInSchema: 1 } });
```

Mongoose has a separate `strictQuery` option to toggle strict mode for the `filter` parameter to queries.

```
const mySchema = new Schema({ field: Number }, {
  strict: true,
  strictQuery: true // Turn on strict mode for query filters
});
const MyModel = mongoose.model('Test', mySchema);

// Mongoose will strip out `notInSchema: 1` because `strictQuery` is `true`
MyModel.find({ notInSchema: 1 });
```

option: toJSON

Exactly the same as the toObject option but only applies when the document's `toJSON` method is called.

```
var schema = new Schema({ name: String });
schema.path('name').get(function (v) {
  return v + ' is my name';
});
schema.set('toJSON', { getters: true, virtuals: false });
var M = mongoose.model('Person', schema);
var m = new M({ name: 'Max Headroom' });
console.log(m.toObject()); // { _id: 504e0cd7dd992d9be2f20b6f, name: 'Max Headroom'
}
console.log(m.toJSON()); // { _id: 504e0cd7dd992d9be2f20b6f, name: 'Max Headroom is
my name' }
// since we know toJSON is called whenever a js object is stringified:
console.log(JSON.stringify(m)); // { "_id": "504e0cd7dd992d9be2f20b6f", "name":
"Max Headroom is my name" }
```

To see all available `toJSON/toObject` options, read this.

option: toObject

Documents have a toObject method which converts the mongoose document into a plain JavaScript object. This method accepts a few options. Instead of applying these options on a per-document basis, we may declare the options at the schema level and have them applied to all of the schema's documents by default.

To have all virtuals show up in your `console.log` output, set the `toObject` option to `{ getters: true }`:

```
var schema = new Schema({ name: String });
```

```
schema.path('name').get(function (v) {
  return v + ' is my name';
});
schema.set('toObject', { getters: true });
var M = mongoose.model('Person', schema);
var m = new M({ name: 'Max Headroom' });
console.log(m); // { _id: 504e0cd7dd992d9be2f20b6f, name: 'Max Headroom is my name'
}
```

To see all available `toObject` options, read this.

option: typeKey

By default, if you have an object with key 'type' in your schema, mongoose will interpret it as a type declaration.

```
// Mongoose interprets this as 'loc is a String'
var schema = new Schema({ loc: { type: String, coordinates: [Number] } });
```

However, for applications like geoJSON, the 'type' property is important. If you want to control which key mongoose uses to find type declarations, set the 'typeKey' schema option.

```
var schema = new Schema({
  // Mongoose interpets this as 'loc is an object with 2 keys, type and
coordinates'
  loc: { type: String, coordinates: [Number] },
  // Mongoose interprets this as 'name is a String'
  name: { $type: String }
}, { typeKey: '$type' }); // A '$type' key means this object is a type declaration
```

option: validateBeforeSave

By default, documents are automatically validated before they are saved to the database. This is to prevent saving an invalid document. If you want to handle validation manually, and be able to save objects which don't pass validation, you can set `validateBeforeSave` to false.

```
var schema = new Schema({ name: String });
schema.set('validateBeforeSave', false);
schema.path('name').validate(function (value) {
    return v != null;
});
var M = mongoose.model('Person', schema);
var m = new M({ name: null });
m.validate(function(err) {
    console.log(err); // Will tell you that null is not allowed.
});
m.save(); // Succeeds despite being invalid
```

option: versionKey

The `versionKey` is a property set on each document when first created by Mongoose. This keys value contains the internal revision of the document. The `versionKey` option is a string that represents the path to use for versioning. The default is `__v`. If this conflicts with your application you can configure as such:

```
const schema = new Schema({ name: 'string' });
const Thing = mongoose.model('Thing', schema);
const thing = new Thing({ name: 'mongoose v3' });
await thing.save(); // { __v: 0, name: 'mongoose v3' }

// customized versionKey
new Schema({..}, { versionKey: '_somethingElse' })
const Thing = mongoose.model('Thing', schema);
const thing = new Thing({ name: 'mongoose v3' });
thing.save(); // { _somethingElse: 0, name: 'mongoose v3' }
```

Note that Mongoose versioning is **not** a full optimistic concurrency solution. Use mongoose-update-if-current for OCC support. Mongoose versioning only operates on arrays:

```
// 2 copies of the same document
const doc1 = await Model.findOne({ _id });
const doc2 = await Model.findOne({ _id });

// Delete first 3 comments from `doc1`
doc1.comments.splice(0, 3);
await doc1.save();

// The below `save()` will throw a VersionError, because you're trying to
// modify the comment at index 1, and the above `splice()` removed that
// comment.
doc2.set('comments.1.body', 'new comment');
await doc2.save();
```

Document versioning can also be disabled by setting the `versionKey` to `false`. *DO NOT disable versioning unless you know what you are doing.*

```
new Schema({..}, { versionKey: false });
const Thing = mongoose.model('Thing', schema);
const thing = new Thing({ name: 'no versioning please' });
thing.save(); // { name: 'no versioning please' }
```

Mongoose *only* updates the version key when you use `save()`. If you use `update()`, `findOneAndUpdate()`, etc. Mongoose will **not** update the version key. As a workaround, you can use the below middleware.

```
schema.pre('findOneAndUpdate', function() {
  const update = this.getUpdate();
  if (update.__v != null) {
    delete update.__v;
  }
  const keys = ['$set', '$setOnInsert'];
  for (const key of keys) {
    if (update[key] != null && update[key].__v != null) {
      delete update[key].__v;
      if (Object.keys(update[key]).length === 0) {
        delete update[key];
      }
    }
  }
  update.$inc = update.$inc || {};
  update.$inc.__v = 1;
});
```

option: collation

Sets a default collation for every query and aggregation. Here's a beginner-friendly overview of collations.

```
var schema = new Schema({
  name: String
}, { collation: { locale: 'en_US', strength: 1 } });

var MyModel = db.model('MyModel', schema);

MyModel.create([{ name: 'val' }, { name: 'Val' }]).
  then(function() {
    return MyModel.find({ name: 'val' });
  }).
  then(function(docs) {
    // `docs` will contain both docs, because `strength: 1` means
    // MongoDB will ignore case when matching.
  });
```

option: skipVersioning

`skipVersioning` allows excluding paths from versioning (i.e., the internal revision will not be incremented even if these paths are updated). DO NOT do this unless you know what you're doing. For subdocuments, include this on the parent document using the fully qualified path.

```
new Schema({..}, { skipVersioning: { dontVersionMe: true } });
thing.dontVersionMe.push('hey');
```

```
thing.save(); // version is not incremented
```

option: timestamps

The `timestamps` option tells mongoose to assign `createdAt` and `updatedAt` fields to your schema. The type assigned is Date.

By default, the names of the fields are `createdAt` and `updatedAt`. Customize the field names by setting `timestamps.createdAt` and `timestamps.updatedAt`.

```
const thingSchema = new Schema({..}, { timestamps: { createdAt: 'created_at' } });
const Thing = mongoose.model('Thing', thingSchema);
const thing = new Thing();
await thing.save(); // `created_at` & `updatedAt` will be included

// With updates, Mongoose will add `updatedAt` to `$set`
await Thing.updateOne({}, { $set: { name: 'Test' } });

// If you set upsert: true, Mongoose will add `created_at` to `$setOnInsert` as
well
await Thing.findOneAndUpdate({}, { $set: { name: 'Test2' } });

// Mongoose also adds timestamps to bulkWrite() operations
// See https://mongoosejs.com/docs/api.html#model_Model.bulkWrite
await Thing.bulkWrite([
  insertOne: {
    document: {
      name: 'Jean-Luc Picard',
      ship: 'USS Stargazer'
      // Mongoose will add `created_at` and `updatedAt`
    }
  },
  updateOne: {
    filter: { name: 'Jean-Luc Picard' },
    update: {
      $set: {
        ship: 'USS Enterprise'
        // Mongoose will add `updatedAt`
      }
    }
  }
]);
```

By default, Mongoose uses `new Date()` to get the current time. If you want to overwrite the function Mongoose uses to get the current time, you can set the `timestamps.currentTime` option. Mongoose will call the `timestamps.currentTime` function whenever it needs to get the current time.

```
const schema = Schema({
  createdAt: Number,
  updatedAt: Number,
  name: String
}, {
  // Make Mongoose use Unix time (seconds since Jan 1, 1970)
  timestamps: { currentTime: () => Math.floor(Date.now() / 1000) }
});
```

option: useNestedStrict

Write operations like `update()`, `updateOne()`, `updateMany()`, and `findOneAndUpdate()` only check the top-level schema's strict mode setting.

```
var childSchema = new Schema({}, { strict: false });
var parentSchema = new Schema({ child: childSchema }, { strict: 'throw' });
var Parent = mongoose.model('Parent', parentSchema);
Parent.update({}, { 'child.name': 'Luke Skywalker' }, function(error) {
  // Error because parentSchema has `strict: throw`, even though
  // `childSchema` has `strict: false`
});

var update = { 'child.name': 'Luke Skywalker' };
var opts = { strict: false };
Parent.update({}, update, opts, function(error) {
  // This works because passing `strict: false` to `update()` overwrites
  // the parent schema.
});
If you set useNestedStrict to true, mongoose will use the child schema's strict
option for casting updates.
var childSchema = new Schema({}, { strict: false });
var parentSchema = new Schema({ child: childSchema },
  { strict: 'throw', useNestedStrict: true });
var Parent = mongoose.model('Parent', parentSchema);
Parent.update({}, { 'child.name': 'Luke Skywalker' }, function(error) {
  // Works!
});
```

option: selectPopulatedPaths

By default, Mongoose will automatically `select()` any populated paths for you, unless you explicitly exclude them.

```
const bookSchema = new Schema({
  title: 'String',
  author: { type: 'ObjectId', ref: 'Person' }
});
const Book = mongoose.model('Book', bookSchema);
```

```
// By default, Mongoose will add `author` to the below `select()`.
await Book.find().select('title').populate('author');

// In other words, the below query is equivalent to the above
await Book.find().select('title author').populate('author');
```

To opt out of selecting populated fields by default, set `selectPopulatedPaths` to `false` in your schema.

```
const bookSchema = new Schema({
  title: 'String',
  author: { type: 'ObjectId', ref: 'Person' }
}, { selectPopulatedPaths: false });
const Book = mongoose.model('Book', bookSchema);

// Because `selectPopulatedPaths` is false, the below doc will **not**
// contain an `author` property.
const doc = await Book.findOne().select('title').populate('author');
```

option: storeSubdocValidationError

For legacy reasons, when there is a validation error in subpath of a single nested schema, Mongoose will record that there was a validation error in the single nested schema path as well. For example:

```
const childSchema = new Schema({ name: { type: String, required: true } });
const parentSchema = new Schema({ child: childSchema });

const Parent = mongoose.model('Parent', parentSchema);

// Will contain an error for both 'child.name' _and_ 'child'
new Parent({ child: {} }).validateSync().errors;
```

Set the `storeSubdocValidationError` to `false` on the child schema to make Mongoose only reports the parent error.

```
const childSchema = new Schema({
  name: { type: String, required: true }
}, { storeSubdocValidationError: false }); // <-- set on the child schema
const parentSchema = new Schema({ child: childSchema });

const Parent = mongoose.model('Parent', parentSchema);

// Will only contain an error for 'child.name'
new Parent({ child: {} }).validateSync().errors;
```

With ES6 Classes

Schemas have a `loadClass()` method that you can use to create a Mongoose schema from an ES6 class:

- ES6 class methods become Mongoose methods
- ES6 class statics become Mongoose statics
- ES6 getters and setters become Mongoose virtuals

Here's an example of using `loadClass()` to create a schema from an ES6 class:

```
class MyClass {
  myMethod() { return 42; }
  static myStatic() { return 42; }
  get myVirtual() { return 42; }
}

const schema = new mongoose.Schema();
schema.loadClass(MyClass);

console.log(schema.methods); // { myMethod: [Function: myMethod] }
console.log(schema.statics); // { myStatic: [Function: myStatic] }
console.log(schema.virtuals); // { myVirtual: VirtualType { ... } }
```

Pluggable

Schemas are also pluggable which allows us to package up reusable features into plugins that can be shared with the community or just between your projects.

Further Reading

Here's an alternative introduction to Mongoose schemas.

To get the most out of MongoDB, you need to learn the basics of MongoDB schema design. SQL schema design (third normal form) was designed to minimize storage costs, whereas MongoDB schema design is about making common queries as fast as possible. The *6 Rules of Thumb for MongoDB Schema Design* blog series is an excellent resource for learning the basic rules for making your queries fast.

Users looking to master MongoDB schema design in Node.js should look into *The Little MongoDB Schema Design Book* by Christian Kvalheim, the original author of the MongoDB Node.js driver. This book shows you how to implement performant schemas for a laundry list of use cases, including e-commerce, wikis, and appointment bookings.

Next Up

Now that we've covered `Schemas`, let's take a look at SchemaTypes.

Schema Types

SchemaTypes handle definition of path defaults, validation, getters, setters, field selection defaults for queries, and other general characteristics for Mongoose document properties.

What is a SchemaType?

You can think of a Mongoose schema as the configuration object for a Mongoose model. A SchemaType is then a configuration object for an individual property. A SchemaType says what type a given path should have, whether it has any getters/setters, and what values are valid for that path.

```
const schema = new Schema({ name: String });
schema.path('name') instanceof mongoose.SchemaType; // true
schema.path('name') instanceof mongoose.Schema.Types.String; // true
schema.path('name').instance; // 'String'
```

A SchemaType is different from a type. In other words, `mongoose.ObjectId !== mongoose.Types.ObjectId`. A SchemaType is just a configuration object for Mongoose. An instance of the `mongoose.ObjectId` SchemaType doesn't actually create MongoDB ObjectIds, it is just a configuration for a path in a schema.

The following are all the valid SchemaTypes in Mongoose. Mongoose plugins can also add custom SchemaTypes like int32. Check out Mongoose's plugins search to find plugins.

- String
- Number
- Date

- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

Example

```
var 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

var Thing = mongoose.model('Thing', schema);

var 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);
```

The `type` Key

`type` is a special property in Mongoose schemas. When Mongoose finds a nested property named `type` in your schema, Mongoose assumes that it needs to define a SchemaType with the given type.

```
// 3 string SchemaTypes: 'name', 'nested.firstName', 'nested.lastName'
const schema = new Schema({
  name: { type: String },
  nested: {
    firstName: { type: String },
    lastName: { type: String }
  }
});
```

As a consequence, you need a little extra work to define a property named `type` in your schema. For example, suppose you're building a stock portfolio app, and you want to store the asset's `type` (stock, bond, ETF, etc.). Naively, you might define your schema as shown below:

```
const holdingSchema = new Schema({
  // You might expect `asset` to be an object that has 2 properties,
  // but unfortunately `type` is special in Mongoose so mongoose
  // interprets this schema to mean that `asset` is a string
  asset: {
    type: String,
    ticker: String
  }
});
```

However, when Mongoose sees `type: String`, it assumes that you mean `asset` should be a string, not an object with a property `type`. The correct way to define an object with a property `type` is shown below.

```
const holdingSchema = new Schema({
  asset: {
    // Workaround to make sure Mongoose knows `asset` is an object
    // and `asset.type` is a string, rather than thinking `asset`
    // is a string.
    type: { type: String },
    ticker: String
  }
});
```

SchemaType Options

You can declare a schema type using the type directly, or an object with a `type` property.

```
var schema1 = new Schema({
  test: String // `test` is a path of type String
});

var schema2 = new Schema({
  // The `test` object contains the "SchemaType options"
  test: { type: String } // `test` is a path of type string
});
```

In addition to the type property, you can specify additional properties for a path. For example, if you want to lowercase a string before saving:

```
var schema2 = new Schema({
  test: {
    type: String,
    lowercase: true // Always convert `test` to lowercase
  }
});
```

You can add any property you want to your SchemaType options. Many plugins rely on custom SchemaType options. For example, the mongoose-autopopulate plugin automatically populates paths if you set `autopopulate: true` in your SchemaType options. Mongoose comes with support for several built-in SchemaType options, like `lowercase` in the above example.

The `lowercase` option only works for strings. There are certain options which apply for all schema types, and some that apply for specific schema types.

All Schema Types

- `required`: boolean or function, if true adds a required validator 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 for queries
- `validate`: function, adds a validator function for this property
- `get`: function, defines a custom getter for this property using `Object.defineProperty()`.
- `set`: function, defines a custom setter for this property using `Object.defineProperty()`.
- `alias`: string, mongoose >= 4.10.0 only. Defines a virtual with the given name that gets/sets this path.

```
var numberSchema = new Schema({
  integerOnly: {
    type: Number,
    get: v => Math.round(v),
    set: v => Math.round(v),
    alias: 'i'
  }
});

var Number = mongoose.model('Number', numberSchema);

var 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 using schema type options.

- `index`: boolean, whether to define an index on this property.
- `unique`: boolean, whether to define a unique index on this property.
- `sparse`: boolean, whether to define a sparse index on this property.

```
var schema2 = new Schema({
  test: {
    type: String,
    index: true,
```

```
    unique: true // Unique index. If you specify `unique: true`
    // specifying `index: true` is optional if you do `unique: true`
  }
});
```

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 that checks if the value matches the given regular expression
- `enum`: Array, creates a validator that checks if the value is in the given array.
- `minlength`: Number, creates a validator that checks if the value length is not less than the given number
- `maxlength`: Number, creates a validator that checks if the value length is not greater than the given number

Number

- `min`: Number, creates a validator that checks if the value is greater than or equal to the given minimum.
- `max`: Number, creates a validator that checks if the value is less than or equal to the given maximum.
- `enum`: Array, creates a validator that checks if the value is strictly equal to one of the values in the given array.

Date

- `min`: Date
- `max`: Date

Usage Notes

String

To declare a path as a string, you may use either the `String` global constructor or the string `'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);
```

If you pass an element that has a `toString()` function, Mongoose will call it, unless the element is an array or the `toString()` function is strictly equal to `Object.prototype.toString()`.

```
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

To declare a path as a number, you may use either the `Number` global constructor or the string `'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);
```

There are several types of values that will be successfully cast to a Number.

```
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
```

If you pass an object with a `valueOf()` function that returns a Number, Mongoose will call it and assign the returned value to the path.

The values `null` and `undefined` are not cast.

NaN, strings that cast to NaN, arrays, and objects that don't have a `valueOf()` function will all result in a CastError.

Dates

Built-in `Date` methods are **not** hooked into the mongoose change tracking logic which in English means that if you use a `Date` in your document and modify it with a method like `setMonth()`, mongoose will be unaware of this change and `doc.save()` will not persist this modification. If you must modify `Date` types using built-in methods, tell mongoose about the change with `doc.markModified('pathToYourDate')` before saving.

```
var 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

To declare a path as a Buffer, you may use either the `Buffer` global constructor or the string `'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);
```

Mongoose will successfully cast the below values to buffers.

```
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]}
```

Mixed

An "anything goes" SchemaType. Mongoose will not do any casting on mixed paths. You can define a mixed path using `Schema.Types.Mixed` or by passing an empty object literal. The following are equivalent.

```
const Any = new Schema({ any: {} });
const Any = new Schema({ any: Object });
const Any = new Schema({ any: Schema.Types.Mixed });
const Any = new Schema({ any: mongoose.Mixed });
// Note that by default, if you're using `type`, putting _any_ POJO as the `type`
will
// make the path mixed.
const Any = new Schema({
  any: {
    type: { foo: String }
  } // "any" will be Mixed - everything inside is ignored.
});
// However, as of Mongoose 5.8.0, this behavior can be overridden with
typePojoToMixed.
// In that case, it will create a single nested subdocument type instead.
const Any = new Schema({
  any: {
    type: { foo: String }
  } // "any" will be a single nested subdocument.
}, {typePojoToMixed: false});
```

Since Mixed is a schema-less type, you can change the value to anything else you like, but Mongoose loses the ability to auto detect and save those changes. To tell Mongoose that the value of a Mixed type has changed, you need to call `doc.markModified(path)`, passing the path to the Mixed type you just changed.

To avoid these side-effects, a Subdocument path may be used instead.

```
person.anything = { x: [3, 4, { y: "changed" }] };
person.markModified('anything');
person.save(); // Mongoose will save changes to `anything`.
```

ObjectIds

An 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 });
```

`ObjectId` is a class, and ObjectIds are objects. However, they are often represented as strings. When you convert an ObjectId to a string using `toString()`, you get a 24-character hexadecimal string:

```
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. 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'`

Any other value causes a CastError. You can modify what values Mongoose converts to true or false using the `convertToTrue` and `convertToFalse` properties, which are JavaScript sets.

```
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 and arrays of subdocuments. Arrays of SchemaTypes are also called *primitive arrays*, and arrays of subdocuments are also called *document arrays*.

```
var ToySchema = new Schema({ name: String });
var ToyBoxSchema = new Schema({
  toys: [ToySchema],
  buffers: [Buffer],
  strings: [String],
  numbers: [Number]
  // ... etc
});
```

Arrays are special because they implicitly have a default value of `[]` (empty array).

```
var ToyBox = mongoose.model('ToyBox', ToyBoxSchema);
console.log((new ToyBox()).toys); // []
```

To overwrite this default, you need to set the default value to `undefined`

```
var ToyBoxSchema = new Schema({
  toys: {
    type: [ToySchema],
    default: undefined
  }
});
```

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

```
var Empty1 = new Schema({ any: [] });
var Empty2 = new Schema({ any: Array });
var Empty3 = new Schema({ any: [Schema.Types.Mixed] });
var Empty4 = new Schema({ any: [{}] });
```

Maps

New in Mongoose 5.1.0

A `MongooseMap` is a subclass of JavaScript's `Map` class. In these docs, we'll use the terms 'map' and `MongooseMap` interchangeably. In Mongoose, maps are how you create a nested document with arbitrary keys.

Note: In Mongoose Maps, keys must be strings in order to store the document in MongoDB.

```
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 = mongoose.model('User', userSchema);
// Map { 'github' => 'vkarpov15', 'twitter' => '@code_barbarian' }
console.log(new User({
  socialMediaHandles: {
    github: 'vkarpov15',
    twitter: '@code_barbarian'
  }
}).socialMediaHandles);
```

The above example doesn't explicitly declare `github` or `twitter` as paths, but, since `socialMediaHandles` is a map, you can store arbitrary key/value pairs. However, since `socialMediaHandles` is a map, you **must** use `.get()` to get the value of a key and `.set()` to set the value of a key.

```
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();
```

Map types are stored as BSON objects in MongoDB. Keys in a BSON object are ordered, so this means the insertion order property of maps is maintained.

Getters

Getters are like virtuals for paths defined in your schema. For example, let's say you wanted to store user profile pictures as relative paths and then add the hostname in your application. Below is how you would structure your `userSchema`:

```
const root = 'https://s3.amazonaws.com/mybucket';

const userSchema = new Schema({
  name: String,
  picture: {
    type: String,
    get: v => `${root}${v}`
  }
});

const User = mongoose.model('User', userSchema);

const doc = new User({ name: 'Val', picture: '/123.png' });
doc.picture; // 'https://s3.amazonaws.com/mybucket/123.png'
doc.toObject({ getters: false }).picture; // '123.png'
```

Generally, you only use getters on primitive paths as opposed to arrays or subdocuments. Because getters override what accessing a Mongoose path returns, declaring a getter on an object may remove Mongoose change tracking for that path.

```
const schema = new Schema({
  arr: [{ url: String }]
});

const root = 'https://s3.amazonaws.com/mybucket';

// Bad, don't do this!
schema.path('arr').get(v => {
  return v.map(el => Object.assign(el, { url: root + el.url }));
});
```

```
// Later
doc.arr.push({ key: String });
doc.arr[0]; // 'undefined' because every `doc.arr` creates a new array!
```

Instead of declaring a getter on the array as shown above, you should declare a getter on the `url` string as shown below. If you need to declare a getter on a nested document or array, be very careful!

```
const schema = new Schema({
  arr: [{ url: String }]
});

const root = 'https://s3.amazonaws.com/mybucket';

// Good, do this instead of declaring a getter on `arr`
schema.path('arr.0.url').get(v => `${root}${v}`);
```

Creating Custom Types

Mongoose can also be extended with custom SchemaTypes. Search the plugins site for compatible types like mongoose-long, mongoose-int32, and other types.

Read more about creating custom SchemaTypes here.

The `schema.path()` Function

The `schema.path()` function returns the instantiated schema type for a given path.

```
var sampleSchema = new Schema({ name: { type: String, required: true } });
console.log(sampleSchema.path('name'));
// Output looks like:
/**
 * SchemaString {
 *   enumValues: [],
 *   regExp: null,
 *   path: 'name',
 *   instance: 'String',
 *   validators: ...
 */
```

You can use this function to inspect the schema type for a given path, including what validators it has and what the type is.

Next Up

Now that we've covered `SchemaTypes`, let's take a look at Connections.

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});
```

See the mongodb connection string spec for more detail.

- Buffering
- Error Handling
- Options
- Connection String Options
- Connection Events
- A note about keepAlive
- Server Selection
- Replica Set Connections
- Replica Set Host Names
- Multi-mongos support
- Multiple connections
- Connection Pools
- Option Changes in v5.x

Operation Buffering

Mongoose lets you start using your models immediately, without waiting for mongoose to establish a connection to MongoDB.

```
mongoose.connect('mongodb://localhost:27017/myapp', {useNewUrlParser: true});
var MyModel = mongoose.model('Test', new Schema({ name: String }));
// Works
MyModel.findOne(function(error, result) { /* ... */ });
```

That's because mongoose buffers model function calls internally. This buffering is convenient, but also a common source of confusion. Mongoose will *not* throw any errors by default if you use a model without connecting.

```
var MyModel = mongoose.model('Test', new Schema({ name: String }));
// Will just hang until mongoose successfully connects
```

```
MyModel.findOne(function(error, result) { /* ... */ });

setTimeout(function() {
  mongoose.connect('mongodb://localhost:27017/myapp', {useNewUrlParser: true});
}, 60000);
```

To disable buffering, turn off the `bufferCommands` option on your schema. If you have `bufferCommands` on and your connection is hanging, try turning `bufferCommands` off to see if you haven't opened a connection properly. You can also disable `bufferCommands` globally:

```
mongoose.set('bufferCommands', false);
```

Note that buffering is also responsible for waiting until Mongoose creates collections if you use the `autoCreate` option. If you disable buffering, you should also disable the `autoCreate` option and use `createCollection()` to create capped collections or collections with collations.

```
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

There are two classes of errors that can occur with a Mongoose connection.

- Error on initial connection. If initial connection fails, Mongoose will **not** attempt to reconnect, it will emit an 'error' event, and the promise `mongoose.connect()` returns will reject.
- Error after initial connection was established. Mongoose will attempt to reconnect, and it will emit an 'error' event.

To handle initial connection errors, you should use `.catch()` or `try/catch` with async/await.

```
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);
}
```

To handle errors after initial connection was established, you should listen for error events on the connection. However, you still need to handle initial connection errors as shown above.

```
mongoose.connection.on('error', err => {
  logError(err);
});
```

Options

The `connect` method also accepts an `options` object which will be passed on to the underlying MongoDB driver.

```
mongoose.connect(uri, options);
```

A full list of options can be found on the MongoDB Node.js driver docs for `connect()`. Mongoose passes options to the driver without modification, modulo a few exceptions that are explained below.

- `bufferCommands` - This is a mongoose-specific option (not passed to the MongoDB driver) that disables mongoose's buffering mechanism
- `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.

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

- `useNewUrlParser` - The underlying MongoDB driver has deprecated their current 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`.

- `useCreateIndex` - False by default. Set to `true` to make Mongoose's default index build use `createIndex()` instead of `ensureIndex()` to avoid deprecation warnings from the MongoDB driver.
- `useFindAndModify` - True by default. Set to `false` to make `findOneAndUpdate()` and `findOneAndRemove()` use native `findOneAndUpdate()` rather than `findAndModify()`.
- `useUnifiedTopology`- False by default. Set to `true` to opt in to using the MongoDB driver's new connection management engine. You should set this option to `true`, except for the unlikely case that it prevents you from maintaining a stable connection.
- `promiseLibrary` - Sets the underlying driver's promise library.
- `poolSize` - The maximum number of sockets the MongoDB driver will keep open for this connection. By default, `poolSize` is 5. Keep in mind that, as of MongoDB 3.4, MongoDB only allows one operation per socket at a time, so you may want to increase this if you find you have a few slow queries that are blocking faster queries from proceeding. See Slow Trains in MongoDB and Node.js.
- `socketTimeoutMS` - How long the MongoDB driver will wait before killing a socket due to inactivity *after initial connection*. A socket may be inactive because of either no activity or a long-running operation. This is set to `30000` by default, you should set this to 2-3x your longest running operation if you expect some of your database operations to run longer than 20 seconds. This option is passed to Node.js `socket#setTimeout()` function after the MongoDB driver successfully completes.
- `family` - Whether to connect using IPv4 or IPv6. This option passed to Node.js' `dns.lookup()` function. If you don't specify this option, the MongoDB driver will try IPv6 first and then IPv4 if IPv6 fails. If your `mongoose.connect(uri)` call takes a long time, try `mongoose.connect(uri, { family: 4 })`
- `authSource` - The database to use when authenticating with `user` and `pass`. In MongoDB, users are scoped to a database. If you are getting an unexpected login failure, you may need to set this option.

The following options are important for tuning Mongoose only if you are running **without** the `useUnifiedTopology` option:

- `autoReconnect` - The underlying MongoDB driver will automatically try to reconnect when it loses connection to MongoDB. Unless you are an

extremely advanced user that wants to manage their own connection pool, do **not** set this option to `false`.

- `reconnectTries` - If you're connected to a single server or mongos proxy (as opposed to a replica set), the MongoDB driver will try to reconnect every `reconnectInterval` milliseconds for `reconnectTries` times, and give up afterward. When the driver gives up, the mongoose connection emits a `reconnectFailed` event. This option does nothing for replica set connections.
- `reconnectInterval` - See `reconnectTries`
- `bufferMaxEntries` - The MongoDB driver also has its own buffering mechanism that kicks in when the driver is disconnected. Set this option to 0 and set `bufferCommands` to `false` on your schemas if you want your database operations to fail immediately when the driver is not connected, as opposed to waiting for reconnection.
- `connectTimeoutMS` - How long the MongoDB driver will wait before killing a socket due to inactivity *during initial connection*. Defaults to 30000. This option is passed transparently to Node.js' `socket#setTimeout()` function.

The following options are important for tuning Mongoose only if you are running **with** the `useUnifiedTopology` option:

- `serverSelectionTimeoutMS` - With `useUnifiedTopology`, the MongoDB driver will try to find a server to send any given operation to, and keep retrying for `serverSelectionTimeoutMS` milliseconds. If not set, the MongoDB driver defaults to using `30000` (30 seconds).
- `heartbeatFrequencyMS` - With `useUnifiedTopology`, the MongoDB driver sends a heartbeat every `heartbeatFrequencyMS` to check on the status of the connection. A heartbeat is subject to `serverSelectionTimeoutMS`, so the MongoDB driver will retry failed heartbeats for up to 30 seconds by default. Mongoose only emits a `'disconnected'` event after a heartbeat has failed, so you may want to decrease this setting to reduce the time between when your server goes down and when Mongoose emits `'disconnected'`. We recommend you do **not** set this setting below 1000, too many heartbeats can lead to performance degradation.

The `serverSelectionTimeoutMS` option also handles how long `mongoose.connect()` will retry initial connection before erroring out. With `useUnifiedTopology`, `mongoose.connect()` will retry for 30 seconds by default (default `serverSelectionTimeoutMS`) before erroring out. To get faster feedback on failed operations, you can reduce `serverSelectionTimeoutMS` to 5000 as shown below.

Example:

```
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);
```

See this page for more information about `connectTimeoutMS` and `socketTimeoutMS`

Callback

The `connect()` function also accepts a callback parameter and returns a promise.

```
mongoose.connect(uri, options, function(error) {
  // Check error in initial connection. There is no 2nd param to the callback.
});

// Or using promises
mongoose.connect(uri, options).then(
  () => { /** ready to use. The `mongoose.connect()` promise resolves to mongoose
instance. */ },
  err => { /** handle initial connection error */ }
);
```

Connection String Options

You can also specify driver options in your connection string as parameters in the query string portion of the URI. This only applies to options passed to the MongoDB driver. You **can't** set Mongoose-specific options like `bufferCommands` in the query string.

```
mongoose.connect('mongodb://localhost:27017/test?connectTimeoutMS=1000&bufferComman
ds=false&authSource=otherdb');
// The above is equivalent to:
mongoose.connect('mongodb://localhost:27017/test', {
  connectTimeoutMS: 1000
  // Note that mongoose will **not** pull `bufferCommands` from the query string
});
```

The disadvantage of putting options in the query string is that query string options are harder to read. The advantage is that you only need a single configuration option, the URI, rather than separate options for `socketTimeoutMS`, `connectTimeoutMS`, etc. Best practice is to put options that likely differ between development and

production, like `replicaSet` or `ssl`, in the connection string, and options that should remain constant, like `connectTimeoutMS` or `poolSize`, in the options object.

The MongoDB docs have a full list of supported connection string options. Below are some options that are often useful to set in the connection string because they are closely associated with the hostname and authentication information.

- `authSource` - The database to use when authenticating with `user` and `pass`. In MongoDB, users are scoped to a database. If you are getting an unexpected login failure, you may need to set this option.
- `family` - Whether to connect using IPv4 or IPv6. This option passed to Node.js' `dns.lookup()` function. If you don't specify this option, the MongoDB driver will try IPv6 first and then IPv4 if IPv6 fails. If your `mongoose.connect(uri)` call takes a long time, try `mongoose.connect(uri, { family: 4 })`

Connection Events

Connections inherit from Node.js' `EventEmitter` class, and emit events when something happens to the connection, like losing connectivity to the MongoDB server. Below is a list of events that a connection may emit.

- `connecting`: Emitted when Mongoose starts making its initial connection to the MongoDB server
- `connected`: Emitted when Mongoose successfully makes its initial connection to the MongoDB server
- `open`: Equivalent to `connected`
- `disconnecting`: Your app called `Connection#close()` to disconnect from MongoDB
- `disconnected`: Emitted when Mongoose lost connection to the MongoDB server. This event may be due to your code explicitly closing the connection, the database server crashing, or network connectivity issues.
- `close`: Emitted after `Connection#close()` successfully closes the connection. If you call `conn.close()`, you'll get both a 'disconnected' event and a 'close' event.
- `reconnected`: Emitted if Mongoose lost connectivity to MongoDB and successfully reconnected. Mongoose attempts to automatically reconnect when it loses connection to the database.
- `error`: Emitted if an error occurs on a connection, like a `parseError` due to malformed data or a payload larger than 16MB.
- `fullsetup`: Emitted when you're connecting to a replica set and Mongoose has successfully connected to the primary and at least one secondary.

- `all`: Emitted when you're connecting to a replica set and Mongoose has successfully connected to all servers specified in your connection string.
- `reconnectFailed`: Emitted when you're connected to a standalone server and Mongoose has run out of `reconnectTries`. The MongoDB driver will no longer attempt to reconnect after this event is emitted. This event will never be emitted if you're connected to a replica set.

A note about keepAlive

For long running applications, it is often prudent to enable `keepAlive` with a number of milliseconds. Without it, after some period of time you may start to see `"connection closed"` errors for what seems like no reason. If so, after reading this, you may decide to enable `keepAlive`:

```
mongoose.connect(uri, { keepAlive: true, keepAliveInitialDelay: 300000 });
```

`keepAliveInitialDelay` is the number of milliseconds to wait before initiating `keepAlive` on the socket. `keepAlive` is true by default since mongoose 5.2.0.

Replica Set Connections

To connect to a replica set you pass a comma delimited list of hosts to connect to rather than a single host.

```
mongoose.connect('mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]' [, options]);
```

For example:

```
mongoose.connect('mongodb://user:pw@host1.com:27017,host2.com:27017,host3.com:27017/testdb');
```

To connect to a single node replica set, specify the `replicaSet` option.

```
mongoose.connect('mongodb://host1:port1/?replicaSet=rsName');
```

Server Selection

If you enable the `useUnifiedTopology` option, the underlying MongoDB driver will use server selection to connect to MongoDB and send operations to MongoDB. If the MongoDB driver can't find a server to send an operation to after `serverSelectionTimeoutMS`, you'll get the below error:

```
MongoTimeoutError: Server selection timed out after 30000 ms
```

You can configure the timeout using the `serverSelectionTimeoutMS` option to

```
mongoose.connect():
mongoose.connect(uri, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
```

```
  serverSelectionTimeoutMS: 5000 // Timeout after 5s instead of 30s
});
```

A `MongoTimeoutError` has a `reason` property that explains why server selection timed out. For example, if you're connecting to a standalone server with an incorrect password, `reason` will contain an "Authentication failed" error.

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

const uri = 'mongodb+srv://username:badpw@cluster0-OMITTED.mongodb.net/' +
  'test?retryWrites=true&w=majority';
// Prints "MongoError: bad auth Authentication failed."
mongoose.connect(uri, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  serverSelectionTimeoutMS: 5000
}).catch(err => console.log(err.reason));
```

Replica Set Host Names

MongoDB replica sets rely on being able to reliably figure out the domain name for each member. On Linux and OSX, the MongoDB server uses the output of the `hostname` command to figure out the domain name to report to the replica set. This can cause confusing errors if you're connecting to a remote MongoDB replica set running on a machine that reports its `hostname` as `localhost`:

```
// Can get this error even if your connection string doesn't include
// `localhost` if `rs.conf()` reports that one replica set member has
// `localhost` as its host name.
failed to connect to server [localhost:27017] on first connect
```

If you're experiencing a similar error, connect to the replica set using the `mongo` shell and run the `rs.conf()` command to check the host names of each replica set member. Follow this page's instructions to change a replica set member's host name.

Multi-mongos support

You can also connect to multiple mongos instances for high availability in a sharded cluster. You do not need to pass any special options to connect to multiple mongos in mongoose 5.x.

```
// Connect to 2 mongos servers
mongoose.connect('mongodb://mongosA:27501,mongosB:27501', cb);
```

Multiple connections

So far we've seen how to connect to MongoDB using Mongoose's default connection. Mongoose creates a *default connection* when you call

`mongoose.connect()`. You can access the default connection using `mongoose.connection`.

You may need multiple connections to MongoDB for several reasons. One reason is if you have multiple databases or multiple MongoDB clusters. Another reason is to work around slow trains. The `mongoose.createConnection()` function takes the same arguments as `mongoose.connect()` and returns a new connection.

```
const conn =
mongoose.createConnection('mongodb://[username:password@]host1[:port1][,host2[:port
2],...[,hostN[:portN]]][/[database][?options]]', options);
```

This connection object is then used to create and retrieve models. Models are **always** scoped to a single connection.

```
const UserModel = conn.model('User', userSchema);
```

If you use multiple connections, you should make sure you export schemas, **not** models. Exporting a model from a file is called the *export model pattern*. The export model pattern is limited because you can only use one connection.

```
const userSchema = new Schema({ name: String, email: String });

// The alternative to the export model pattern is the export schema pattern.
module.exports = userSchema;

// Because if you export a model as shown below, the model will be scoped
// to Mongoose's default connection.
// module.exports = mongoose.model('User', userSchema);
```

If you use the export schema pattern, you still need to create models somewhere. There are two common patterns. First is to export a connection and register the models on the connection in the file:

```
// connections/fast.js
const mongoose = require('mongoose');

const conn = mongoose.createConnection(process.env.MONGODB_URI);
conn.model('User', require('../schemas/user'));

module.exports = conn;

// connections/slow.js
const mongoose = require('mongoose');

const conn = mongoose.createConnection(process.env.MONGODB_URI);
conn.model('User', require('../schemas/user'));
conn.model('PageView', require('../schemas/pageView'));
```

```
module.exports = conn;
```

Another alternative is to register connections with a dependency injector or another inversion of control (IOC) pattern.

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

module.exports = function connectionFactory() {
  const conn = mongoose.createConnection(process.env.MONGODB_URI);

  conn.model('User', require('../schemas/user'));
  conn.model('PageView', require('../schemas/pageView'));

  return conn;
};
```

Connection Pools

Each `connection`, whether created with `mongoose.connect` or `mongoose.createConnection` are all backed by an internal configurable connection pool defaulting to a maximum size of 5. Adjust the pool size using your connection options:

```
// With object options
mongoose.createConnection(uri, { poolSize: 4 });

const uri = 'mongodb://localhost:27017/test?poolSize=4';
mongoose.createConnection(uri);
```

Option Changes in v5.x

You may see the following deprecation warning if upgrading from 4.x to 5.x and you didn't use the `useMongoClient` option in 4.x:

```
the server/replset/mongos options are deprecated, all their options are supported
at the top level of the options object
```

In older version of the MongoDB driver you had to specify distinct options for server connections, replica set connections, and mongos connections:

```
mongoose.connect(myUri, {
  server: {
    socketOptions: {
      socketTimeoutMS: 0,
      keepAlive: true
    },
    reconnectTries: 30
  },
```

```
  replset: {
    socketOptions: {
      socketTimeoutMS: 0,
      keepAlive: true
    },
    reconnectTries: 30
  },
  mongos: {
    socketOptions: {
      socketTimeoutMS: 0,
      keepAlive: true
    },
    reconnectTries: 30
  }
});
```

In mongoose v5.x you can instead declare these options at the top level, without all that extra nesting. Here's the list of all supported options.

```
// Equivalent to the above code
mongoose.connect(myUri, {
  socketTimeoutMS: 0,
  keepAlive: true,
  reconnectTries: 30
});
```

Next Up

Now that we've covered connections, let's take a look at models.

Models

Models are fancy constructors compiled from `Schema` definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

- Compiling your first model
- Constructing Documents
- Querying
- Deleting
- Updating
- Change Streams

Compiling your first model

When you call `mongoose.model()` on a schema, Mongoose *compiles* a model for you.

```
var schema = new mongoose.Schema({ name: 'string', size: 'string' });
var Tank = mongoose.model('Tank', schema);
```

The first argument is the *singular* name of the collection your model is for. ** Mongoose automatically looks for the plural, lowercased version of your model name. ** Thus, for the example above, the model Tank is for the **tanks** collection in the database.

Note: The `.model()` function makes a copy of `schema`. Make sure that you've added everything you want to `schema`, including hooks, before calling `.model()`!

Constructing Documents

An instance of a model is called a document. Creating them and saving to the database is easy.

```
var Tank = mongoose.model('Tank', yourSchema);

var small = new Tank({ size: 'small' });
small.save(function (err) {
  if (err) return handleError(err);
  // saved!
});

// or

Tank.create({ size: 'small' }, function (err, small) {
  if (err) return handleError(err);
  // saved!
});

// or, for inserting large batches of documents
Tank.insertMany([{ size: 'small' }], function(err) {

});
```

Note that no tanks will be created/removed until the connection your model uses is open. Every model has an associated connection. When you use `mongoose.model()`, your model will use the default mongoose connection.

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

If you create a custom connection, use that connection's `model()` function instead.

```
var connection = mongoose.createConnection('mongodb://localhost:27017/test');
var Tank = connection.model('Tank', yourSchema);
```

Querying

Finding documents is easy with Mongoose, which supports the rich query syntax of MongoDB. Documents can be retreived using each `models` find, findById, findOne, or where static methods.

```
Tank.find({ size: 'small' }).where('createdDate').gt(oneYearAgo).exec(callback);
```

See the chapter on queries for more details on how to use the Query api.

Deleting

Models have static `deleteOne()` and `deleteMany()` functions for removing all documents matching the given `filter`.

```
Tank.deleteOne({ size: 'large' }, function (err) {
  if (err) return handleError(err);
  // deleted at most one tank document
});
```

Updating

Each `model` has its own `update` method for modifying documents in the database without returning them to your application. See the API docs for more detail.

```
Tank.updateOne({ size: 'large' }, { name: 'T-90' }, function(err, res) {
  // Updated at most one doc, `res.modifiedCount` contains the number
  // of docs that MongoDB updated
});
```

If you want to update a single document in the db and return it to your application, use findOneAndUpdate instead.

Change Streams

New in MongoDB 3.6.0 and Mongoose 5.0.0

Change streams provide a way for you to listen to all inserts and updates going through your MongoDB database. Note that change streams do **not** work unless you're connected to a MongoDB replica set.

```
async function run() {
  // Create a new mongoose model
  const personSchema = new mongoose.Schema({
    name: String
  });
  const Person = mongoose.model('Person', personSchema, 'Person');

  // Create a change stream. The 'change' event gets emitted when there's a
  // change in the database
  Person.watch().
```

```
    on('change', data => console.log(new Date(), data));

  // Insert a doc, will trigger the change stream handler above
  console.log(new Date(), 'Inserting doc');
  await Person.create({ name: 'Axl Rose' });
}
```

The output from the above async function will look like what you see below.

```
2018-05-11T15:05:35.467Z 'Inserting doc'
2018-05-11T15:05:35.487Z 'Inserted doc'
2018-05-11T15:05:35.491Z { _id: { _data: ... },
  operationType: 'insert',
  fullDocument: { _id: 5af5b13fe526027666c6bf83, name: 'Axl Rose', __v: 0 },
  ns: { db: 'test', coll: 'Person' },
  documentKey: { _id: 5af5b13fe526027666c6bf83 } }
```

You can read more about change streams in mongoose in this blog post.

Yet more

The API docs cover many additional methods available like count, mapReduce, aggregate, and more.

Next Up

Now that we've covered `Models`, let's take a look at Documents.

Documents

Mongoose documents represent a one-to-one mapping to documents as stored in MongoDB. Each document is an instance of its Model.

- Documents vs Models
- Retrieving
- Updating
- Validating
- Overwriting

Documents vs Models

Document and Model are distinct classes in Mongoose. The Model class is a subclass of the Document class. When you use the Model constructor, 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
```

In Mongoose, a "document" generally means an instance of a model. You should not have to create an instance of the Document class without going through a model.

Retrieving

When you load documents from MongoDB using model functions like `findOne()`, you get a Mongoose document back.

```
const doc = await MyModel.findOne();

doc instanceof MyModel; // true
doc instanceof mongoose.Model; // true
doc instanceof mongoose.Document; // true
```

Updating

Mongoose documents track changes. You can modify a document using vanilla JavaScript assignments and Mongoose will convert it into MongoDB update operators.

```
doc.name = 'foo';

// Mongoose sends a `updateOne({ _id: doc._id }, { $set: { name: 'foo' } })`
// to MongoDB.
await doc.save();
```

If the document with the corresponding `_id` is not found, Mongoose will report a `DocumentNotFoundError`:

```
const doc = await MyModel.findOne();

// Delete the document so Mongoose won't be able to save changes
await MyModel.deleteOne({ _id: doc._id });

doc.name = 'foo';
await doc.save(); // Throws DocumentNotFoundError
```

The `save()` function is generally the right way to update a document with Mongoose. With `save()`, you get full validation and middleware.

For cases when `save()` isn't flexible enough, Mongoose lets you create your own MongoDB updates with casting, middleware, and limited validation.

```
// Update all documents in the `mymodels` collection
await MyModel.updateMany({}, { $set: { name: 'foo' } });
```

Note that `update()`, `updateMany()`, `findOneAndUpdate()`, etc. do not execute `save()` middleware. If you need save middleware and full validation, first query for the document and then `save()` it.

Validating

Documents are casted validated before they are saved. Mongoose first casts values to the specified type and then validates them. Internally, Mongoose calls the document's `validate()` method before saving.

```
const schema = new Schema({ name: String, age: { type: Number, min: 0 } });
const Person = mongoose.model('Person', schema);

let p = new Person({ name: 'foo', age: 'bar' });
// Cast to Number failed for value "bar" at path "age"
await p.validate();

let p2 = new Person({ name: 'foo', age: -1 });
// Path `age` (-1) is less than minimum allowed value (0).
await p2.validate();
```

Mongoose also supports limited validation on updates using the `runValidators` option. Mongoose casts parameters to query functions like `findOne()`, `updateOne()` by default. However, Mongoose does *not* run validation on query function parameters by default. You need to set `runValidators: true` for Mongoose to validate.

```
// Cast to number failed for value "bar" at path "age"
await Person.updateOne({}, { age: 'bar' });

// Path `age` (-1) is less than minimum allowed value (0).
await Person.updateOne({}, { age: -1 }, { runValidators: true });
```

Read the validation guide for more details.

Overwriting

There are 2 different ways to overwrite a document (replacing all keys in the document). One way is to use the `Document#overwrite()` function followed by `save()`.

```
const doc = await Person.findOne({ _id });

// Sets `name` and unsets all other properties
doc.overwrite({ name: 'Jean-Luc Picard' });
await doc.save();
```

The other way is to use `Model.replaceOne()`.

```
// Sets `name` and unsets all other properties
await Person.replaceOne({ _id }, { name: 'Jean-Luc Picard' });
```

Next Up

Now that we've covered Documents, let's take a look at Subdocuments.

Queries

Mongoose models provide several static helper functions for CRUD operations. Each of these functions returns a mongoose `Query` object.

- `Model.deleteMany()`
- `Model.deleteOne()`
- `Model.find()`
- `Model.findById()`
- `Model.findByIdAndDelete()`
- `Model.findByIdAndRemove()`
- `Model.findByIdAndUpdate()`
- `Model.findOne()`
- `Model.findOneAndDelete()`
- `Model.findOneAndRemove()`
- `Model.findOneAndReplace()`
- `Model.findOneAndUpdate()`
- `Model.replaceOne()`
- `Model.updateMany()`
- `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`.

A query also has a `.then()` function, and thus can be used as a promise.

- Executing
- Queries are Not Promises
- References to other documents
- Streaming
- Versus Aggregation

Executing

When executing a query with a `callback` function, you specify your query as a JSON document. The JSON document's syntax is the same as the MongoDB shell.

```
var 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.

Anywhere a callback is passed to a query in Mongoose, the callback follows the pattern `callback(error, results)`. What `results` is depends on the operation: For `findOne()` it is a potentially-null single document, `find()` a list of documents, `count()` the number of documents, `update()` the number of documents affected, etc. The API docs for Models provide more detail on what is passed to the callbacks.

Now let's look at what happens when no `callback` is passed:

```
// find each person with a last name matching 'Ghost'
var 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);
});
```

In the above code, the `query` variable is of type Query. A `Query` enables you to build up a query using chaining syntax, rather than specifying a JSON object. The below 2 examples are equivalent.

```
// 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);
```

A full list of Query helper functions can be found in the API docs.

Queries are Not Promises

Mongoose queries are **not** promises. They have a `.then()` function for co and async/await 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.

References to other documents

There are no joins in MongoDB but sometimes we still want references to documents in other collections. This is where population comes in. Read more about how to include documents from other collections in your query results here.

Streaming

You can stream query results from MongoDB. You need to call the Query#cursor() function to return an instance of QueryCursor.

```
const cursor = Person.find({ occupation: /host/ }).cursor();

for (let doc = await cursor.next(); doc != null; doc = await cursor.next()) {
  console.log(doc); // Prints documents one at a time
}
```

Iterating through a Mongoose query using async iterators also creates a cursor.

```
for await (const doc of Person.find()) {
  console.log(doc); // Prints documents one at a time
}
```

Cursors are subject to cursor timeouts. By default, MongoDB will close your cursor after 10 minutes and subsequent `next()` calls will result in a `MongoError: cursor id 123 not found` error. To override this, set the `noCursorTimeout` option on your cursor.

```
// MongoDB won't automatically close this cursor after 10 minutes.
const cursor = Person.find().cursor().addCursorFlag('noCursorTimeout', true);
```

However, cursors can still time out because of session idle timeouts. So even a cursor with `noCursorTimeout` set will still time out after 30 minutes of inactivity. You can read more about working around session idle timeouts in the MongoDB documentation.

Versus Aggregation

Aggregation can do many of the same things that queries can. For example, below is how you can use `aggregate()` to find docs where `name.last = 'Ghost'`:

```
const docs = await Person.aggregate([{ $match: { 'name.last': 'Ghost' } }]);
```

However, just because you can use `aggregate()` doesn't mean you should. In general, you should use queries where possible, and only use `aggregate()` when you absolutely need to.

Unlike query results, Mongoose does **not** `hydrate()` aggregation results. Aggregation results are always POJOs, not Mongoose documents.

```
const docs = await Person.aggregate([{ $match: { 'name.last': 'Ghost' } }]);

docs[0] instanceof mongoose.Document; // false
```

Also, unlike query filters, Mongoose also doesn't cast aggregation pipelines. That means you're responsible for ensuring the values you pass in to an aggregation pipeline have the correct type.

```
const doc = await Person.findOne();

const idString = doc._id.toString();

// Finds the `Person`, because Mongoose casts `idString` to an ObjectId
const queryRes = await Person.findOne({ _id: idString });

// Does **not** find the `Person`, because Mongoose doesn't cast aggregation
// pipelines.
const aggRes = await Person.aggregate([{ $match: { _id: idString } }])
```

Next Up

Now that we've covered `Queries`, let's take a look at Validation.

Validation

Before we get into the specifics of validation syntax, please keep the following rules in mind:

- Validation is defined in the SchemaType
- Validation is middleware. Mongoose registers validation as a `pre('save')` hook on every schema by default.
- You can manually run validation using `doc.validate(callback)` or `doc.validateSync()`
- Validators are not run on undefined values. The only exception is the `required` validator.
- Validation is asynchronously recursive; when you call Model#save, sub-document validation is executed as well. If an error occurs, your Model#save callback receives it
- Validation is customizable

```
var schema = new Schema({
  name: {
    type: String,
    required: true
  }
});
var Cat = db.model('Cat', schema);

// This cat has no name :(
var cat = new Cat();
cat.save(function(error) {
  assert.equal(error.errors['name'].message,
    'Path `name` is required.');
```

```
  error = cat.validateSync();
  assert.equal(error.errors['name'].message,
    'Path `name` is required.');
});
```

Built-in Validators

Mongoose has several built-in validators.

- All SchemaTypes have the built-in required validator. The required validator uses the SchemaType's `checkRequired()` function to determine if the value satisfies the required validator.
- Numbers have `min` and `max` validators.
- Strings have `enum`, `match`, `minlength`, and `maxlength` validators.

Each of the validator links above provide more information about how to enable them and customize their error messages.

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12
  },
  bacon: {
    type: Number,
    required: [true, 'Why no bacon?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea'],
    required: function() {
      return this.bacon > 3;
    }
  }
});
var Breakfast = db.model('Breakfast', breakfastSchema);

var badBreakfast = new Breakfast({
  eggs: 2,
  bacon: 0,
  drink: 'Milk'
});
var error = badBreakfast.validateSync();
assert.equal(error.errors['eggs'].message,
```

```
  'Too few eggs');
assert.ok(!error.errors['bacon']);
assert.equal(error.errors['drink'].message,
  '`Milk` is not a valid enum value for path `drink`.');

badBreakfast.bacon = 5;
badBreakfast.drink = null;

error = badBreakfast.validateSync();
assert.equal(error.errors['drink'].message, 'Path `drink` is required.');

badBreakfast.bacon = null;
error = badBreakfast.validateSync();
assert.equal(error.errors['bacon'].message, 'Why no bacon?');
```

The `unique` Option is Not a Validator

A common gotcha for beginners is that the `unique` option for schemas is *not* a validator. It's a convenient helper for building MongoDB unique indexes. See the FAQ for more information.

```
const uniqueUsernameSchema = new Schema({
  username: {
    type: String,
    unique: true
  }
});
const U1 = db.model('U1', uniqueUsernameSchema);
const U2 = db.model('U2', uniqueUsernameSchema);

const dup = [{ username: 'Val' }, { username: 'Val' }];
U1.create(dup, err => {
  // Race condition! This may save successfully, depending on whether
  // MongoDB built the index before writing the 2 docs.
});

// You need to wait for Mongoose to finish building the `unique`
// index before writing. You only need to build indexes once for
// a given collection, so you normally don't need to do this
// in production. But, if you drop the database between tests,
// you will need to use `init()` to wait for the index build to finish.
U2.init().
  then(() => U2.create(dup)).
  catch(error => {
    // Will error, but will *not* be a mongoose validation error, it will be
    // a duplicate key error.
```

```
    // See: https://masteringjs.io/tutorials/mongoose/e11000-duplicate-key
    assert.ok(error);
    assert.ok(!error.errors);
    assert.ok(error.message.indexOf('duplicate key error') !== -1);
  });
```

Custom Validators

If the built-in validators aren't enough, you can define custom validators to suit your needs.

Custom validation is declared by passing a validation function. You can find detailed instructions on how to do this in the `SchemaType#validate()` API docs.

```
var userSchema = new Schema({
  phone: {
    type: String,
    validate: {
      validator: function(v) {
        return /\d{3}-\d{3}-\d{4}/.test(v);
      },
      message: props => `${props.value} is not a valid phone number!`
    },
    required: [true, 'User phone number required']
  }
});

var User = db.model('user', userSchema);
var user = new User();
var error;

user.phone = '555.0123';
error = user.validateSync();
assert.equal(error.errors['phone'].message,
  '555.0123 is not a valid phone number!');

user.phone = '';
error = user.validateSync();
assert.equal(error.errors['phone'].message,
  'User phone number required');

user.phone = '201-555-0123';
// Validation succeeds! Phone number is defined
// and fits `DDD-DDD-DDDD`
error = user.validateSync();
assert.equal(error, null);
```

Async Custom Validators

Custom validators can also be asynchronous. If your validator function returns a promise (like an `async` function), mongoose will wait for that promise to settle. If the returned promise rejects, or fulfills with the value `false`, Mongoose will consider that a validation error.

```
const userSchema = new Schema({
  name: {
    type: String,
    // You can also make a validator async by returning a promise.
    validate: () => Promise.reject(new Error('Oops!'))
  },
  email: {
    type: String,
    // There are two ways for an promise-based async validator to fail:
    // 1) If the promise rejects, Mongoose assumes the validator failed with the
given error.
    // 2) If the promise resolves to `false`, Mongoose assumes the validator failed
and creates an error with the given `message`.
    validate: {
      validator: () => Promise.resolve(false),
      message: 'Email validation failed'
    }
  }
});

const User = db.model('User', userSchema);
const user = new User();

user.email = 'test@test.co';
user.name = 'test';
user.validate().catch(error => {
  assert.ok(error);
  assert.equal(error.errors['name'].message, 'Oops!');
  assert.equal(error.errors['email'].message, 'Email validation failed');
});
```

Validation Errors

Errors returned after failed validation contain an `errors` object whose values are `ValidatorError` objects. Each ValidatorError has `kind`, `path`, `value`, and `message` properties. A ValidatorError also may have a `reason` property. If an error was thrown in the validator, this property will contain the error that was thrown.

```
var toySchema = new Schema({
```

```
  color: String,
  name: String
});

var validator = function(value) {
  return /red|white|gold/i.test(value);
};
toySchema.path('color').validate(validator,
  'Color `{VALUE}` not valid', 'Invalid color');
toySchema.path('name').validate(function(v) {
  if (v !== 'Turbo Man') {
    throw new Error('Need to get a Turbo Man for Christmas');
  }
  return true;
}, 'Name `{VALUE}` is not valid');

var Toy = db.model('Toy', toySchema);

var toy = new Toy({ color: 'Green', name: 'Power Ranger' });

toy.save(function (err) {
  // `err` is a ValidationError object
  // `err.errors.color` is a ValidatorError object
  assert.equal(err.errors.color.message, 'Color `Green` not valid');
  assert.equal(err.errors.color.kind, 'Invalid color');
  assert.equal(err.errors.color.path, 'color');
  assert.equal(err.errors.color.value, 'Green');

  // This is new in mongoose 5. If your validator throws an exception,
  // mongoose will use that message. If your validator returns `false`,
  // mongoose will use the 'Name `Power Ranger` is not valid' message.
  assert.equal(err.errors.name.message,
    'Need to get a Turbo Man for Christmas');
  assert.equal(err.errors.name.value, 'Power Ranger');
  // If your validator threw an error, the `reason` property will contain
  // the original error thrown, including the original stack trace.
  assert.equal(err.errors.name.reason.message,
    'Need to get a Turbo Man for Christmas');

  assert.equal(err.name, 'ValidationError');
});
```

Cast Errors

Before running validators, Mongoose attempts to coerce values to the correct type. This process is called *casting* the document. If casting fails for a given path, the `error.errors` object will contain a `CastError` object.

Casting runs before validation, and validation does not run if casting fails. That means your custom validators may assume `v` is `null`, `undefined`, or an instance of the type specified in your schema.

```
const vehicleSchema = new mongoose.Schema({
  numWheels: { type: Number, max: 18 }
});
const Vehicle = db.model('Vehicle', vehicleSchema);

const doc = new Vehicle({ numWheels: 'not a number' });
const err = doc.validateSync();

err.errors['numWheels'].name; // 'CastError'
// 'Cast to Number failed for value "not a number" at path "numWheels"'
err.errors['numWheels'].message;
```

Required Validators On Nested Objects

Defining validators on nested objects in mongoose is tricky, because nested objects are not fully fledged paths.

```
var personSchema = new Schema({
  name: {
    first: String,
    last: String
  }
});

assert.throws(function() {
  // This throws an error, because 'name' isn't a full fledged path
  personSchema.path('name').required(true);
}, /Cannot.*'required'/);

// To make a nested object required, use a single nested schema
var nameSchema = new Schema({
  first: String,
  last: String
});

personSchema = new Schema({
  name: {
    type: nameSchema,
    required: true
```

```
  }
});

var Person = db.model('Person', personSchema);

var person = new Person();
var error = person.validateSync();
assert.ok(error.errors['name']);
```

Update Validators

In the above examples, you learned about document validation. Mongoose also supports validation for `update()`, `updateOne()`, `updateMany()`, and `findOneAndUpdate()` operations. Update validators are off by default - you need to specify the `runValidators` option.

To turn on update validators, set the `runValidators` option for `update()`, `updateOne()`, `updateMany()`, or `findOneAndUpdate()`. Be careful: update validators are off by default because they have several caveats.

```
var toySchema = new Schema({
  color: String,
  name: String
});

var Toy = db.model('Toys', toySchema);

Toy.schema.path('color').validate(function (value) {
  return /red|green|blue/i.test(value);
}, 'Invalid color');

var opts = { runValidators: true };
Toy.updateOne({}, { color: 'not a color' }, opts, function (err) {
  assert.equal(err.errors.color.message,
    'Invalid color');
});
```

Update Validators and `this`

There are a couple of key differences between update validators and document validators. In the color validation function above, `this` refers to the document being validated when using document validation. However, when running update validators, the document being updated may not be in the server's memory, so by default the value of `this` is not defined.

```
var toySchema = new Schema({
  color: String,
```

```
  name: String
});

toySchema.path('color').validate(function(value) {
  // When running in `validate()` or `validateSync()`, the
  // validator can access the document using `this`.
  // Does **not** work with update validators.
  if (this.name.toLowerCase().indexOf('red') !== -1) {
    return value !== 'red';
  }
  return true;
});

var Toy = db.model('ActionFigure', toySchema);

var toy = new Toy({ color: 'red', name: 'Red Power Ranger' });
var error = toy.validateSync();
assert.ok(error.errors['color']);

var update = { color: 'red', name: 'Red Power Ranger' };
var opts = { runValidators: true };

Toy.updateOne({}, update, opts, function(error) {
  // The update validator throws an error:
  // "TypeError: Cannot read property 'toLowerCase' of undefined",
  // because `this` is **not** the document being updated when using
  // update validators
  assert.ok(error);
});
```

The `context` option

The `context` option lets you set the value of `this` in update validators to the underlying query.

```
toySchema.path('color').validate(function(value) {
  // When running update validators with the `context` option set to
  // 'query', `this` refers to the query object.
  if (this.getUpdate().$set.name.toLowerCase().indexOf('red') !== -1) {
    return value === 'red';
  }
  return true;
});

var Toy = db.model('Figure', toySchema);
```

```
var update = { color: 'blue', name: 'Red Power Ranger' };
// Note the context option
var opts = { runValidators: true, context: 'query' };

Toy.updateOne({}, update, opts, function(error) {
  assert.ok(error.errors['color']);
});
```

Update Validators Only Run On Updated Paths

The other key difference is that update validators only run on the paths specified in the update. For instance, in the below example, because 'name' is not specified in the update operation, update validation will succeed.

When using update validators, `required` validators **only** fail when you try to explicitly `$unset` the key.

```
var kittenSchema = new Schema({
  name: { type: String, required: true },
  age: Number
});

var Kitten = db.model('Kitten', kittenSchema);

var update = { color: 'blue' };
var opts = { runValidators: true };
Kitten.updateOne({}, update, opts, function(err) {
  // Operation succeeds despite the fact that 'name' is not specified
});

var unset = { $unset: { name: 1 } };
Kitten.updateOne({}, unset, opts, function(err) {
  // Operation fails because 'name' is required
  assert.ok(err);
  assert.ok(err.errors['name']);
});
```

Update Validators Only Run For Some Operations

One final detail worth noting: update validators **only** run on the following update operators:

- `$set`
- `$unset`
- `$push` (>= 4.8.0)
- `$addToSet` (>= 4.8.0)

- `$pull` (>= 4.12.0)
- `$pullAll` (>= 4.12.0)

For instance, the below update will succeed, regardless of the value of `number`, because update validators ignore `$inc`.

Also, `$push`, `$addToSet`, `$pull`, and `$pullAll` validation does **not** run any validation on the array itself, only individual elements of the array.

```
var testSchema = new Schema({
  number: { type: Number, max: 0 },
  arr: [{ message: { type: String, maxlength: 10 } }]
});

// Update validators won't check this, so you can still `$push` 2 elements
// onto the array, so long as they don't have a `message` that's too long.
testSchema.path('arr').validate(function(v) {
  return v.length < 2;
});

var Test = db.model('Test', testSchema);

var update = { $inc: { number: 1 } };
var opts = { runValidators: true };
Test.updateOne({}, update, opts, function(error) {
  // There will never be a validation error here
  update = { $push: [{ message: 'hello' }, { message: 'world' }] };
  Test.updateOne({}, update, opts, function(error) {
    // This will never error either even though the array will have at
    // least 2 elements.
  });
});
```

On $push and $addToSet

New in 4.8.0: update validators also run on `$push` and `$addToSet`

```
var testSchema = new Schema({
  numbers: [{ type: Number, max: 0 }],
  docs: [{
    name: { type: String, required: true }
  }]
});

var Test = db.model('TestPush', testSchema);

var update = {
```

```
  $push: {
    numbers: 1,
    docs: { name: null }
  }
};
var opts = { runValidators: true };
Test.updateOne({}, update, opts, function(error) {
  assert.ok(error.errors['numbers']);
  assert.ok(error.errors['docs']);
});
```

Middleware

Middleware (also called pre and post *hooks*) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.

- Types of Middleware
- Pre
- Errors in Pre Hooks
- Post
- Asynchronous Post Hooks
- Define Middleware Before Compiling Models
- Save/Validate Hooks
- Naming Conflicts
- Notes on findAndUpdate() and Query Middleware
- Error Handling Middleware
- Aggregation Hooks
- Synchronous Hooks

Types of Middleware

Mongoose has 4 types of middleware: document middleware, model middleware, aggregate middleware, and query middleware. Document middleware is supported for the following document functions. In document middleware functions, `this` refers to the document.

- validate
- save
- remove
- updateOne

- deleteOne
- init (note: init hooks are synchronous)

Query middleware is supported for the following Model and Query functions. In query middleware functions, `this` refers to the query.

- count
- deleteMany
- deleteOne
- find
- findOne
- findOneAndDelete
- findOneAndRemove
- findOneAndUpdate
- remove
- update
- updateOne
- updateMany

Aggregate middleware is for `MyModel.aggregate()`. Aggregate middleware executes when you call `exec()` on an aggregate object. In aggregate middleware, `this` refers to the aggregation object.

- aggregate

Model middleware is supported for the following model functions. In model middleware functions, `this` refers to the model.

- insertMany

All middleware types support pre and post hooks. How pre and post hooks work is described in more detail below.

Note: If you specify `schema.pre('remove')`, Mongoose will register this middleware for `doc.remove()` by default. If you want to your middleware to run on `Query.remove()` use `schema.pre('remove', { query: true, document: false }, fn)`.

Note: Unlike `schema.pre('remove')`, Mongoose registers `updateOne` and `deleteOne` middleware on `Query#updateOne()` and `Query#deleteOne()` by default. This means that both `doc.updateOne()` and `Model.updateOne()` trigger `updateOne` hooks, but `this` refers to a query, not a document. To register `updateOne` or `deleteOne` middleware as document middleware, use `schema.pre('updateOne', { document: true, query: false })`.

Note: The `create()` function fires `save()` hooks.

Pre

Pre middleware functions are executed one after another, when each middleware calls `next`.

```
var schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

In mongoose 5.x, instead of calling `next()` manually, you can use a function that returns a promise. In particular, you can use `async/await`.

```
schema.pre('save', function() {
  return doStuff().
    then(() => doMoreStuff());
});

// Or, in Node.js >= 7.6.0:
schema.pre('save', async function() {
  await doStuff();
  await doMoreStuff();
});
```

If you use `next()`, the `next()` call does **not** stop the rest of the code in your middleware function from executing. Use the early `return` pattern to prevent the rest of your middleware function from running when you call `next()`.

```
var schema = new Schema(..);
schema.pre('save', function(next) {
  if (foo()) {
    console.log('calling next!');
    // `return next();` will make sure the rest of this function doesn't run
    /*return*/ next();
  }
  // Unless you comment out the `return` above, 'after next' will print
  console.log('after next');
});
```

Use Cases

Middleware are useful for atomizing model logic. Here are some other ideas:

- complex validation
- removing dependent documents (removing a user removes all his blogposts)
- asynchronous defaults
- asynchronous tasks that a certain action triggers

Errors in Pre Hooks

If any pre hook errors out, mongoose will not execute subsequent middleware or the hooked function. Mongoose will instead pass an error to the callback and/or reject the returned promise. There are several ways to report an error in middleware:

```
schema.pre('save', function(next) {
  const err = new Error('something went wrong');
  // If you call `next()` with an argument, that argument is assumed to be
  // an error.
  next(err);
});

schema.pre('save', function() {
  // You can also return a promise that rejects
  return new Promise((resolve, reject) => {
    reject(new Error('something went wrong'));
  });
});

schema.pre('save', function() {
  // You can also throw a synchronous error
  throw new Error('something went wrong');
});

schema.pre('save', async function() {
  await Promise.resolve();
  // You can also throw an error in an `async` function
  throw new Error('something went wrong');
});

// later...

// Changes will not be persisted to MongoDB because a pre hook errored out
myDoc.save(function(err) {
  console.log(err.message); // something went wrong
});
```

Calling `next()` multiple times is a no-op. If you call `next()` with an error `err1` and then throw an error `err2`, mongoose will report `err1`.

Post middleware

post middleware are executed *after* the hooked method and all of its `pre` middleware have completed.

```
schema.post('init', function(doc) {
```

```
  console.log('%s has been initialized from the db', doc._id);
});
schema.post('validate', function(doc) {
  console.log('%s has been validated (but not saved yet)', doc._id);
});
schema.post('save', function(doc) {
  console.log('%s has been saved', doc._id);
});
schema.post('remove', function(doc) {
  console.log('%s has been removed', doc._id);
});
```

Asynchronous Post Hooks

If your post hook function takes at least 2 parameters, mongoose will assume the second parameter is a `next()` function that you will call to trigger the next middleware in the sequence.

```
// Takes 2 parameters: this is an asynchronous post hook
schema.post('save', function(doc, next) {
  setTimeout(function() {
    console.log('post1');
    // Kick off the second post hook
    next();
  }, 10);
});

// Will not execute until the first middleware calls `next()`
schema.post('save', function(doc, next) {
  console.log('post2');
  next();
});
```

Define Middleware Before Compiling Models

Calling `pre()` or `post()` after compiling a model does **not** work in Mongoose in general. For example, the below `pre('save')` middleware will not fire.

```
const schema = new mongoose.Schema({ name: String });

// Compile a model from the schema
const User = mongoose.model('User', schema);

// Mongoose will **not** call the middleware function, because
// this middleware was defined after the model was compiled
schema.pre('save', () => console.log('Hello from pre save'));

new User({ name: 'test' }).save();
```

This means that you must add all middleware and plugins **before** calling `mongoose.model()`. The below script will print out "Hello from pre save":

```
const schema = new mongoose.Schema({ name: String });
// Mongoose will call this middleware function, because this script adds
// the middleware to the schema before compiling the model.
schema.pre('save', () => console.log('Hello from pre save'));

// Compile a model from the schema
const User = mongoose.model('User', schema);

new User({ name: 'test' }).save();
```

As a consequence, be careful about exporting Mongoose models from the same file that you define your schema. If you choose to use this pattern, you must define global plugins **before** calling `require()` on your model file.

```
const schema = new mongoose.Schema({ name: String });

// Once you `require()` this file, you can no longer add any middleware
// to this schema.
module.exports = mongoose.model('User', schema);
```

Save/Validate Hooks

The `save()` function triggers `validate()` hooks, because mongoose has a built-in `pre('save')` hook that calls `validate()`. This means that all `pre('validate')` and `post('validate')` hooks get called **before** any `pre('save')` hooks.

```
schema.pre('validate', function() {
  console.log('this gets printed first');
});
schema.post('validate', function() {
  console.log('this gets printed second');
});
schema.pre('save', function() {
  console.log('this gets printed third');
});
schema.post('save', function() {
  console.log('this gets printed fourth');
});
```

Naming Conflicts

Mongoose has both query and document hooks for `remove()`.

```
schema.pre('remove', function() { console.log('Removing!'); });

// Prints "Removing!"
```

```
doc.remove();

// Does **not** print "Removing!". Query middleware for `remove` is not
// executed by default.
Model.remove();
```

You can pass options to `Schema.pre()` and `Schema.post()` to switch whether Mongoose calls your `remove()` hook for `Document.remove()` or `Model.remove()`:

```
// Only document middleware
schema.pre('remove', { document: true }, function() {
  console.log('Removing doc!');
});

// Only query middleware. This will get called when you do `Model.remove()`
// but not `doc.remove()`.
schema.pre('remove', { query: true }, function() {
  console.log('Removing!');
});
```

Notes on findAndUpdate() and Query Middleware

Pre and post `save()` hooks are **not** executed on `update()`, `findOneAndUpdate()`, etc. You can see a more detailed discussion why in this GitHub issue. Mongoose 4.0 introduced distinct hooks for these functions.

```
schema.pre('find', function() {
  console.log(this instanceof mongoose.Query); // true
  this.start = Date.now();
});

schema.post('find', function(result) {
  console.log(this instanceof mongoose.Query); // true
  // prints returned documents
  console.log('find() returned ' + JSON.stringify(result));
  // prints number of milliseconds the query took
  console.log('find() took ' + (Date.now() - this.start) + ' millis');
});
```

Query middleware differs from document middleware in a subtle but important way: in document middleware, `this` refers to the document being updated. In query middleware, mongoose doesn't necessarily have a reference to the document being updated, so `this` refers to the **query** object rather than the document being updated.

For instance, if you wanted to add an `updatedAt` timestamp to every `updateOne()` call, you would use the following pre hook.

```
schema.pre('updateOne', function() {
  this.set({ updatedAt: new Date() });
});
```

You **cannot** access the document being updated in `pre('updateOne')` or `pre('findOneAndUpdate')` query middleware. If you need to access the document that will be updated, you need to execute an explicit query for the document.

```
schema.pre('findOneAndUpdate', async function() {
  const docToUpdate = await this.model.findOne(this.getQuery());
  console.log(docToUpdate); // The document that `findOneAndUpdate()` will modify
});
```

However, if you define `pre('updateOne')` document middleware, `this` will be the document being updated. That's because `pre('updateOne')` document middleware hooks into `Document#updateOne()` rather than `Query#updateOne()`.

```
schema.pre('updateOne', { document: true, query: false }, function() {
  console.log('Updating');
});
const Model = mongoose.model('Test', schema);

const doc = new Model();
await doc.updateOne({ $set: { name: 'test' } }); // Prints "Updating"

// Doesn't print "Updating", because `Query#updateOne()` doesn't fire
// document middleware.
await Model.updateOne({}, { $set: { name: 'test' } });
```

Error Handling Middleware

New in 4.5.0

Middleware execution normally stops the first time a piece of middleware calls `next()` with an error. However, there is a special kind of post middleware called "error handling middleware" that executes specifically when an error occurs. Error handling middleware is useful for reporting errors and making error messages more readable.

Error handling middleware is defined as middleware that takes one extra parameter: the 'error' that occurred as the first parameter to the function. Error handling middleware can then transform the error however you want.

```
var schema = new Schema({
  name: {
    type: String,
    // Will trigger a MongoError with code 11000 when
    // you save a duplicate
```

```
    unique: true
  }
});

// Handler **must** take 3 parameters: the error that occurred, the document
// in question, and the `next()` function
schema.post('save', function(error, doc, next) {
  if (error.name === 'MongoError' && error.code === 11000) {
    next(new Error('There was a duplicate key error'));
  } else {
    next();
  }
});

// Will trigger the `post('save')` error handler
Person.create([{ name: 'Axl Rose' }, { name: 'Axl Rose' }]);
```

Error handling middleware also works with query middleware. You can also define a post `update()` hook that will catch MongoDB duplicate key errors.

```
// The same E11000 error can occur when you call `update()`
// This function **must** take 3 parameters. If you use the
// `passRawResult` function, this function **must** take 4
// parameters
schema.post('update', function(error, res, next) {
  if (error.name === 'MongoError' && error.code === 11000) {
    next(new Error('There was a duplicate key error'));
  } else {
    next(); // The `update()` call will still error out.
  }
});

var people = [{ name: 'Axl Rose' }, { name: 'Slash' }];
Person.create(people, function(error) {
  Person.update({ name: 'Slash' }, { $set: { name: 'Axl Rose' } }, function(error)
{
    // `error.message` will be "There was a duplicate key error"
  });
});
```

Error handling middleware can transform an error, but it can't remove the error. Even if you call `next()` with no error as shown above, the function call will still error out.

Aggregation Hooks

You can also define hooks for the `Model.aggregate()` function. In aggregation middleware functions, `this` refers to the Mongoose `Aggregate` object. For example,

suppose you're implementing soft deletes on a `Customer` model by adding an `isDeleted` property. To make sure `aggregate()` calls only look at customers that aren't soft deleted, you can use the below middleware to add a `$match` stage to the beginning of each aggregation pipeline.

```
customerSchema.pre('aggregate', function() {
  // Add a $match state to the beginning of each pipeline.
  this.pipeline().unshift({ $match: { isDeleted: { $ne: true } } });
});
```

The `Aggregate#pipeline()` function lets you access the MongoDB aggregation pipeline that Mongoose will send to the MongoDB server. It is useful for adding stages to the beginning of the pipeline from middleware.

Synchronous Hooks

Certain Mongoose hooks are synchronous, which means they do **not** support functions that return promises or receive a `next()` callback. Currently, only `init` hooks are synchronous, because the `init()` function is synchronous. Below is an example of using pre and post init hooks.

```
const schema = new Schema({ title: String, loadedAt: Date });

schema.pre('init', pojo => {
  assert.equal(pojo.constructor.name, 'Object'); // Plain object before init
});

const now = new Date();
schema.post('init', doc => {
  assert.ok(doc instanceof mongoose.Document); // Mongoose doc after init
  doc.loadedAt = now;
});

const Test = db.model('Test', schema);

return Test.create({ title: 'Casino Royale' }).
  then(doc => Test.findById(doc)).
  then(doc => assert.equal(doc.loadedAt.valueOf(), now.valueOf()));
```

To report an error in an init hook, you must throw a **synchronous** error. Unlike all other middleware, init middleware does **not** handle promise rejections.

```
const schema = new Schema({ title: String });

const swallowedError = new Error('will not show');
// init hooks do **not** handle async errors or any sort of async behavior
schema.pre('init', () => Promise.reject(swallowedError));
schema.post('init', () => { throw Error('will show'); });
```

```
const Test = db.model('Test', schema);

return Test.create({ title: 'Casino Royale' }).
  then(doc => Test.findById(doc)).
  catch(error => assert.equal(error.message, 'will show'));
```

Next Up

Now that we've covered middleware, let's take a look at Mongoose's approach to faking JOINs with its query population helper.

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();
```

limit vs. perDocumentLimit

Populate does support a `limit` option, however, it currently does **not** limit on a per-document basis. 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().sort({ name: 1 }).populate({
  path: 'fans',
  options: { limit: 2 }
});

stories[0].fans.length; // 2
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.

```
const stories = await Story.find().sort({ name: 1 }).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].fans.length; // 2
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` 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. Just make sure you call `Document#execPopulate()` to execute the `populate()`.

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

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

// Call the `populate()` method on a document to populate a path.
// Need to call `execPopulate()` to actually execute the `populate()`.
await person.populate('stories').execPopulate();

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

The `Document#populate()` method supports chaining, so you can chain multiple `populate()` calls together.

```
await person.populate('stories').populate('fans').execPopulate();
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.

```
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' }
  });
```

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. In particular, arrays that grow without bound are a MongoDB anti-pattern. Using mongoose virtuals, you can define more sophisticated relationships between documents.

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

const 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
});

const Person = mongoose.model('Person', PersonSchema);
const 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` */
});
```

You can also use the populate `match` option to add an additional filter to the `populate()` query. This is useful if you need to split up `populate()` data:

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

const BandSchema = new Schema({
  name: String
});
BandSchema.virtual('activeMembers', {
  ref: 'Person',
  localField: 'name',
  foreignField: 'band',
  justOne: false,
  match: { isActive: true }
});
BandSchema.virtual('formerMembers', {
  ref: 'Person',
  localField: 'name',
  foreignField: 'band',
  justOne: false,
  match: { isActive: false }
});
```

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, set the `virtuals: true` option on your schema's `toJSON` options.

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

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').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();
  });
});
```

Next Up

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

The `model.discriminator()` function

Discriminators are a schema inheritance mechanism. They enable you to have multiple models with overlapping schemas on top of the same underlying MongoDB collection.

Suppose you wanted to track different types of events in a single collection. Every event will have a timestamp, but events that represent clicked links should have a URL. You can achieve this using the `model.discriminator()` function. This function takes 3 parameters, a model name, a discriminator schema and an optional key (defaults to the model name). It returns a model whose schema is the union of the base schema and the discriminator schema.

```
var options = {discriminatorKey: 'kind'};

var eventSchema = new mongoose.Schema({time: Date}, options);
var Event = mongoose.model('Event', eventSchema);

// ClickedLinkEvent is a special type of Event that has
// a URL.
var ClickedLinkEvent = Event.discriminator('ClickedLink',
  new mongoose.Schema({url: String}, options));

// When you create a generic event, it can't have a URL field...
var genericEvent = new Event({time: Date.now(), url: 'google.com'});
assert.ok(!genericEvent.url);

// But a ClickedLinkEvent can
var clickedEvent =
  new ClickedLinkEvent({time: Date.now(), url: 'google.com'});
assert.ok(clickedEvent.url);
```

Discriminators save to the Event model's collection

Suppose you created another discriminator to track events where a new user registered. These `SignedUpEvent` instances will be stored in the same collection as generic events and `ClickedLinkEvent` instances.

```
var event1 = new Event({time: Date.now()});
var event2 = new ClickedLinkEvent({time: Date.now(), url: 'google.com'});
var event3 = new SignedUpEvent({time: Date.now(), user: 'testuser'});

var save = function (doc, callback) {
  doc.save(function (error, doc) {
    callback(error, doc);
  });
};

Promise.all([event1.save(), event2.save(), event3.save()]).
  then(() => Event.countDocuments()).
  then(count => {
    assert.equal(count, 3);
  });
```

Discriminator keys

The way mongoose tells the difference between the different discriminator models is by the 'discriminator key', which is `__t` by default. Mongoose adds a String path called `__t` to your schemas that it uses to track which discriminator this document is an instance of.

```
var event1 = new Event({time: Date.now()});
var event2 = new ClickedLinkEvent({time: Date.now(), url: 'google.com'});
var event3 = new SignedUpEvent({time: Date.now(), user: 'testuser'});

assert.ok(!event1.__t);
assert.equal(event2.__t, 'ClickedLink');
assert.equal(event3.__t, 'SignedUp');
```

Discriminators add the discriminator key to queries

Discriminator models are special; they attach the discriminator key to queries. In other words, `find()`, `count()`, `aggregate()`, etc. are smart enough to account for discriminators.

```
var event1 = new Event({time: Date.now()});
var event2 = new ClickedLinkEvent({time: Date.now(), url: 'google.com'});
var event3 = new SignedUpEvent({time: Date.now(), user: 'testuser'});
```

```
Promise.all([event1.save(), event2.save(), event3.save()]).
  then(() => ClickedLinkEvent.find({})).
  then(docs => {
    assert.equal(docs.length, 1);
    assert.equal(docs[0]._id.toString(), event2._id.toString());
    assert.equal(docs[0].url, 'google.com');
  });
```

Discriminators copy pre and post hooks

Discriminators also take their base schema's pre and post middleware. However, you can also attach middleware to the discriminator schema without affecting the base schema.

```
var options = {discriminatorKey: 'kind'};

var eventSchema = new mongoose.Schema({time: Date}, options);
var eventSchemaCalls = 0;
eventSchema.pre('validate', function (next) {
  ++eventSchemaCalls;
  next();
});
var Event = mongoose.model('GenericEvent', eventSchema);

var clickedLinkSchema = new mongoose.Schema({url: String}, options);
var clickedSchemaCalls = 0;
clickedLinkSchema.pre('validate', function (next) {
  ++clickedSchemaCalls;
  next();
});
var ClickedLinkEvent = Event.discriminator('ClickedLinkEvent',
  clickedLinkSchema);

var event1 = new ClickedLinkEvent();
event1.validate(function() {
  assert.equal(eventSchemaCalls, 1);
  assert.equal(clickedSchemaCalls, 1);

  var generic = new Event();
  generic.validate(function() {
    assert.equal(eventSchemaCalls, 2);
    assert.equal(clickedSchemaCalls, 1);
  });
});
```

Handling custom _id fields

A discriminator's fields are the union of the base schema's fields and the discriminator schema's fields, and the discriminator schema's fields take precedence. There is one exception: the `_id` field. If a custom _id field is set on the base schema, that will always override the discriminator's _id field, as shown below.

```
var options = {discriminatorKey: 'kind'};

// Base schema has a custom String `_id` and a Date `time`...
var eventSchema = new mongoose.Schema({_id: String, time: Date},
  options);
var Event = mongoose.model('BaseEvent', eventSchema);

var clickedLinkSchema = new mongoose.Schema({
  url: String,
  time: String
}, options);
// The discriminator schema has a String `time` and an
// implicitly added ObjectId `_id`.
assert.ok(clickedLinkSchema.path('_id'));
assert.equal(clickedLinkSchema.path('_id').instance, 'ObjectID');
var ClickedLinkEvent = Event.discriminator('ChildEventBad',
  clickedLinkSchema);

var event1 = new ClickedLinkEvent({_id: 'custom id', time: '4pm'});
// clickedLinkSchema overwrites the `time` path, but **not**
// the `_id` path.
assert.strictEqual(typeof event1._id, 'string');
assert.strictEqual(typeof event1.time, 'string');
```

Using discriminators with `Model.create()`

When you use `Model.create()`, mongoose will pull the correct type from the discriminator key for you.

```
var Schema = mongoose.Schema;
var shapeSchema = new Schema({
  name: String
}, { discriminatorKey: 'kind' });

var Shape = db.model('Shape', shapeSchema);

var Circle = Shape.discriminator('Circle',
  new Schema({ radius: Number }));
var Square = Shape.discriminator('Square',
  new Schema({ side: Number }));
```

```
var shapes = [
  { name: 'Test' },
  { kind: 'Circle', radius: 5 },
  { kind: 'Square', side: 10 }
];
Shape.create(shapes, function(error, shapes) {
  assert.ifError(error);
  assert.ok(shapes[0] instanceof Shape);
  assert.ok(shapes[1] instanceof Circle);
  assert.equal(shapes[1].radius, 5);
  assert.ok(shapes[2] instanceof Square);
  assert.equal(shapes[2].side, 10);
});
```

Embedded discriminators in arrays

You can also define discriminators on embedded document arrays. Embedded discriminators are different because the different discriminator types are stored in the same document array (within a document) rather than the same collection. In other words, embedded discriminators let you store subdocuments matching different schemas in the same array.

As a general best practice, make sure you declare any hooks on your schemas **before** you use them. You should **not** call `pre()` or `post()` after calling `discriminator()`

```
var eventSchema = new Schema({ message: String },
  { discriminatorKey: 'kind', _id: false });

var batchSchema = new Schema({ events: [eventSchema] });

// `batchSchema.path('events')` gets the mongoose `DocumentArray`
var docArray = batchSchema.path('events');

// The `events` array can contain 2 different types of events, a
// 'clicked' event that requires an element id that was clicked...
var clickedSchema = new Schema({
  element: {
    type: String,
    required: true
  }
}, { _id: false });
// Make sure to attach any hooks to `eventSchema` and `clickedSchema`
// **before** calling `discriminator()`.
var Clicked = docArray.discriminator('Clicked', clickedSchema);
```

```
// ... and a 'purchased' event that requires the product that was purchased.
var Purchased = docArray.discriminator('Purchased', new Schema({
  product: {
    type: String,
    required: true
  }
}, { _id: false }));

var Batch = db.model('EventBatch', batchSchema);

// Create a new batch of events with different kinds
var batch = {
  events: [
    { kind: 'Clicked', element: '#hero', message: 'hello' },
    { kind: 'Purchased', product: 'action-figure-1', message: 'world' }
  ]
};

Batch.create(batch).
  then(function(doc) {
    assert.equal(doc.events.length, 2);

    assert.equal(doc.events[0].element, '#hero');
    assert.equal(doc.events[0].message, 'hello');
    assert.ok(doc.events[0] instanceof Clicked);

    assert.equal(doc.events[1].product, 'action-figure-1');
    assert.equal(doc.events[1].message, 'world');
    assert.ok(doc.events[1] instanceof Purchased);

    doc.events.push({ kind: 'Purchased', product: 'action-figure-2' });
    return doc.save();
  }).
  then(function(doc) {
    assert.equal(doc.events.length, 3);

    assert.equal(doc.events[2].product, 'action-figure-2');
    assert.ok(doc.events[2] instanceof Purchased);

    done();
  }).
  catch(done);
```

Recursive embedded discriminators in arrays

You can also define embedded discriminators on embedded discriminators. In the below example, `sub_events` is an embedded discriminator, and for `sub_event` keys with value 'SubEvent', `sub_events.events` is an embedded discriminator.

```
var singleEventSchema = new Schema({ message: String },
  { discriminatorKey: 'kind', _id: false });

var eventListSchema = new Schema({ events: [singleEventSchema] });

var subEventSchema = new Schema({
   sub_events: [singleEventSchema]
}, { _id: false });

var SubEvent = subEventSchema.path('sub_events').
  discriminator('SubEvent', subEventSchema);
eventListSchema.path('events').discriminator('SubEvent', subEventSchema);

var Eventlist = db.model('EventList', eventListSchema);

// Create a new batch of events with different kinds
var list = {
  events: [
    { kind: 'SubEvent', sub_events: [{kind:'SubEvent', sub_events:[],
message:'test1'}], message: 'hello' },
    { kind: 'SubEvent', sub_events: [{kind:'SubEvent',
sub_events:[{kind:'SubEvent', sub_events:[], message:'test3'}], message:'test2'}],
message: 'world' }
  ]
};

Eventlist.create(list).
  then(function(doc) {
    assert.equal(doc.events.length, 2);

    assert.equal(doc.events[0].sub_events[0].message, 'test1');
    assert.equal(doc.events[0].message, 'hello');
    assert.ok(doc.events[0].sub_events[0] instanceof SubEvent);

    assert.equal(doc.events[1].sub_events[0].sub_events[0].message, 'test3');
    assert.equal(doc.events[1].message, 'world');
    assert.ok(doc.events[1].sub_events[0].sub_events[0] instanceof SubEvent);

    doc.events.push({kind:'SubEvent', sub_events:[{kind:'SubEvent', sub_events:[],
message:'test4'}], message:'pushed'});
    return doc.save();
  }).
```

```
  then(function(doc) {
    assert.equal(doc.events.length, 3);

    assert.equal(doc.events[2].message, 'pushed');
    assert.ok(doc.events[2].sub_events[0] instanceof SubEvent);

    done();
  }).
  catch(done);
```

Single nested discriminators

You can also define discriminators on single nested subdocuments, similar to how you can define discriminators on arrays of subdocuments.

As a general best practice, make sure you declare any hooks on your schemas **before** you use them. You should **not** call `pre()` or `post()` after calling `discriminator()`

```
const shapeSchema = Schema({ name: String }, { discriminatorKey: 'kind' });
const schema = Schema({ shape: shapeSchema });

schema.path('shape').discriminator('Circle', Schema({ radius: String }));
schema.path('shape').discriminator('Square', Schema({ side: Number }));

const MyModel = mongoose.model('ShapeTest', schema);

// If `kind` is set to 'Circle', then `shape` will have a `radius` property
let doc = new MyModel({ shape: { kind: 'Circle', radius: 5 } });
doc.shape.radius; // 5

// If `kind` is set to 'Square', then `shape` will have a `side` property
doc = new MyModel({ shape: { kind: 'Square', side: 10 } });
doc.shape.side; // 10
```

Plugins

Schemas are pluggable, that is, they allow for applying pre-packaged capabilities to extend their functionality. This is a very powerful feature.

- Example
- Global Plugins
- Officially Supported Plugins

Example

Plugins are a tool for reusing logic in multiple schemas. Suppose you have several models in your database and want to add a `loadedAt` property to each one. Just create a plugin once and apply it to each `Schema`:

```
// loadedAt.js
module.exports = function loadedAtPlugin(schema, options) {
  schema.virtual('loadedAt').
    get(function() { return this._loadedAt; }).
    set(function(v) { this._loadedAt = v; });

  schema.post(['find', 'findOne'], function(docs) {
    if (!Array.isArray(docs)) {
      docs = [docs];
    }
    const now = new Date();
    for (const doc of docs) {
      doc.loadedAt = now;
    }
  });
};

// game-schema.js
const loadedAtPlugin = require('./loadedAt');
const gameSchema = new Schema({ ... });
gameSchema.plugin(loadedAtPlugin);

// player-schema.js
const loadedAtPlugin = require('./loadedAt');
const playerSchema = new Schema({ ... });
playerSchema.plugin(loadedAtPlugin);
```

We just added last-modified behavior to both our `Game` and `Player` schemas and declared an index on the `lastMod` path of our Games to boot. Not bad for a few lines of code.

Global Plugins

Want to register a plugin for all schemas? The mongoose singleton has a `.plugin()` function that registers a plugin for every schema. For example:

```
const mongoose = require('mongoose');
mongoose.plugin(require('./loadedAt'));

const gameSchema = new Schema({ ... });
const playerSchema = new Schema({ ... });
// `loadedAtPlugin` gets attached to both schemas
const Game = mongoose.model('Game', gameSchema);
```

```
const Player = mongoose.model('Player', playerSchema);
```

Officially Supported Plugins

The Mongoose team maintains several plugins that add cool new features to Mongoose. Here's a couple:

- mongoose-autopopulate: Always `populate()` certain fields in your Mongoose schemas.
- mongoose-lean-virtuals: Attach virtuals to the results of Mongoose queries when using `.lean()`.
- mongoose-cast-aggregation

You can find a full list of officially supported plugins on Mongoose's plugins search site.

Community!

Not only can you re-use schema functionality in your own projects, but you also reap the benefits of the Mongoose community as well. Any plugin published to npm and with 'mongoose' as an npm keyword will show up on our search results page.