Fork me on GitHub

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 3.x to 4.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,
  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 `blogSchema` defines a property in our documents which will be cast to its associated SchemaType. For example, we've defined a `title` which will be cast to the String SchemaType and `date` which will be cast to a `Date` SchemaType.

Keys may also be assigned nested objects containing further key/type definitions *(e.g. the `meta` property above).*

The permitted SchemaTypes are

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

Read more about them 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!
```

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 too.

```
// define a schema
var animalSchema = new Schema({ name: String, type
// assign a function to the "methods" object of ou
animalSchema.methods.findSimilarTypes = function (
  return this.model('Animal').find({ type: this.ty
}
```

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

```
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 unpredictible results. See this for more details.

Statics

Adding static methods to a `Model` is simple as well. Continuing with our `animalSchema`:

```
// assign a function to the "statics" object of ou
animalSchema.statics.findByName = function (name,
  return this.find({ name: new RegExp(name, 'i') }
}

var Animal = mongoose.model('Animal', animalSchema
Animal.findByName('fido', function (err, animals)
  console.log(animals);
});
```

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 l
});

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

When your application starts up, Mongoose automatically calls `ensureIndex` for each defined index in your schema. Mongoose will call `ensureIndex` for each index sequentially, and emit an 'index' event on the model when all the `ensureIndex` 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 `config.autoIndex` to `false`.

```
mongoose.connect('mongodb://user:pass@localhost:po
// or
mongoose.createConnection('mongodb://user:pass@loc
// or
animalSchema.set('autoIndex', false);
// or
new Schema({..}, { autoIndex: false });
```

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 bad = new Person({
    name: { first: 'Walter', last: 'White' }
});
```

Suppose we want to log the full name of `bad`. We could do this manually like so:

```
console.log(bad.name.first + ' ' + bad.name.last)
```

Or we could define a virtual property getter on our `personSchema` so we don't need to write out this string concatenation mess each time:

mongoose

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

Now, when we access our virtual "name.full" property, our getter function will be invoked and the value returned:

```
console.log('%s is insane', bad.name.full); // Wa
```

Note that if the resulting record is converted to an object or JSON, virtuals are not included by default. Pass `virtuals : true` to either toObject() or to toJSON() to have them returned.

It would also be nice to be able to set `this.name.first` and `this.name.last` by setting `this.name.full`. For example, if we wanted to change `bad`'s `name.first` and `name.last` to 'Breaking' and 'Bad' respectively, it'd be nice to just:

```
bad.name.full = 'Breaking Bad';
```

Mongoose lets you do this as well through its virtual property setters:

```
personSchema.virtual('name.full').set(function (na
  var split = name.split(' ');
  this.name.first = split[0];
  this.name.last = split[1];
});

...

mad.name.full = 'Breaking Bad';
console.log(mad.name.first); // Breaking
console.log(mad.name.last);  // Bad
```

Virtual property setters are applied before other validation. So the

- MongoDB Version Compatibility
- 3.6 release notes
- 3.8 release notes
- 4.0 release notes
- API docs
- quick start
- contributors
- prior releases

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.

Options

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

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

// or

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

Valid options:

- autoIndex
- capped
- collection
- emitIndexErrors
- id
- _id
- minimize
- read
- safe
- shardKey
- strict
- toJSON
- toObject
- typeKey
- validateBeforeSave
- versionKey
- skipVersioning
- timestamps

option: autoIndex

option: autoIndex

At application startup, Mongoose sends an `ensureIndex` command for each index declared in your `Schema`. As of Mongoose v3, indexes are created in the `background` by default. If you wish to disable the auto-creation feature and manually handle when indexes are created, set your `Schemas` `autoIndex` option to `false` and use the ensureIndexes method on your model.

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

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: fa
```

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: 100
```

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: 'c
```

option: emitIndexErrors

By default, mongoose will build any indexes you specify in your schema for you, and emit an 'index' event on the model when the index build either succeeds or errors out.

```
MyModel.on('index', function(error) {
  /* If error is truthy, index build failed */
});
```

However, this makes it tricky to catch when your index build fails. The `emitIndexErrors` option makes seeing when your index build fails simpler. If this option is on, mongoose will additionally emit an 'error' event on the model when an index build fails.

```
MyModel.schema.options.emitIndexErrors; // true
MyModel.on('error', function(error) {
  // gets an error whenever index build fails
});
```

Node.js' built-in event emitter throws an exception if an error event is emitted and there are no listeners, so its easy to configure your application to fail fast when an index build fails.

option: id

Mongoose assigns each of your schemas an `id` virtual getter by default which returns the documents _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 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: fa
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 sub-documents. 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: '50341373e894ad16347efe(

// disabled _id
var childSchema = new Schema({ name: String }, { _
var parentSchema = new Schema({ children: [childSc
var Model = mongoose.model('Model', parentSchema);

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

option: minimize

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

```
var schema = new Schema({ name: String, inventory
var Character = mongoose.model('Character', schema

// will store `inventory` field if it is not empty
var frodo = new Character({ name: 'Frodo', invento
Character.findOne({ name: 'Frodo' }, function(err,
  console.log(character); // { name: 'Frodo', inve
});

// will not store `inventory` field if it is empty
var sam = new Character({ name: 'Sam', inventory:
Character.findOne({ name: 'Sam' }, function(err, c
  console.log(character); // { name: 'Sam' }
});
```

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

```
var schema = new Schema({ name: String, inventory
var Character = mongoose.model('Character', schema

// will store `inventory` if empty
var sam = new Character({ name: 'Sam', inventory:
Character.findOne({ name: 'Sam' }, function(err, c
  console.log(character); // { name: 'Sam', invent
});
```

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' })
var schema = new Schema({..}, { read: 'primaryPref
var schema = new Schema({..}, { read: 'secondary'
var schema = new Schema({..}, { read: 'secondaryPr
var schema = new Schema({..}, { read: 'nearest' })
```

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
var options = { replset: { strategy: 'ping' }};
mongoose.connect(uri, options);

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

option: safe

This option is passed to MongoDB with all operations and specifies if errors should be returned to our callbacks as well as tune write behavior.

```
var safe = true;
new Schema({ .. }, { safe: safe });
```

By default this is set to `true` for all schemas which guarentees that any occurring error gets passed back to our callback. By setting `safe` to something else like `{ j: 1, w: 2, wtimeout: 10000 }` we can guarantee the write was committed to the MongoDB journal (j: 1), at least 2 replicas (w: 2), and that the write will timeout if it takes longer than 10 seconds (wtimeout: 10000). Errors will still be passed to our callback.

NOTE: In 3.6.x, you also need to turn versioning off. In 3.7.x and above, versioning will **automatically be disabled** when `safe` is set to `false`

**NOTE: this setting overrides any setting specified by passing db options while creating a connection.

There are other write concerns like `{ w: "majority" }` too. See the MongoDB docs for more details.

```
var safe = { w: "majority", wtimeout: 10000 };
new Schema({ .. }, { safe: safe });
```

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

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

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

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
```

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 str
var thing = new Thing(doc, false); // disables st
```

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

NOTE: do not set to false unless you have good reason.

NOTE: in mongoose v2 the default was false.

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
```

option: toJSON

Exactly the same as the toObject option but only applies when the documents `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: fa
var M = mongoose.model('Person', schema);
var m = new M({ name: 'Max Headroom' });
console.log(m.toObject()); // { _id: 504e0cd7dd992
console.log(m.toJSON()); // { _id: 504e0cd7dd992d9
// since we know toJSON is called whenever a js ob
console.log(JSON.stringify(m)); // { "_id": "504e0
```

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 here and have it applied to all of this schemas 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
```

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, coo
```

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
  loc: { type: String, coordinates: [Number] },
  // Mongoose interprets this as 'name is a Strin
  name: { $type: String }
}, { typeKey: '$type' }); // A '$type' key means t
```

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 i
});
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:

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

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

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 });
var Thing = mongoose.model('Thing', schema);
var thing = new Thing({ name: 'no versioning pleas
thing.save(); // { name: 'no versioning please' }
```

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 sub-documents, include this on the parent document using the fully qualified path.

```
new Schema({..}, { skipVersioning: { dontVersionMe
thing.dontVersionMe.push('hey');
thing.save(); // version is not incremented
```

option: timestamps

If set `timestamps`, mongoose assigns `createdAt` and `updatedAt` fields to your schema, the type assigned is Date.

By default, the name of two fields are `createdAt` and `updatedAt`, custom the field name by setting `timestamps.createdAt` and `timestamps.updatedAt`.

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

option: useNestedStrict

In mongoose 4, `update()` and `findOneAndUpdate()` only check the top-level schema's strict mode setting.

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

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.

Next Up

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

SchemaTypes.