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 5.x to 6.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.

```
import mongoose from 'mongoose';
const { Schema } = mongoose;

const 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 doesn't have a `type` property.

In these cases, Mongoose only creates actual schema paths for leaves in the tree. (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)`:

```
const 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
const 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.

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

dog.findSimilarTypes((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') })
};

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

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

Animal.findOne().byName('fido').exec((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.

```
const 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 });
const Animal = mongoose.model('Animal', animalSchema);

Animal.on('index', 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
const personSchema = new Schema({
  name: {
    first: String,
    last: String
  }
});

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

// create a document
const 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.

You can learn more about virtuals here.

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.

```
const personSchema = new Schema({
  n: {
    type: String,
```

```
    // Now accessing `name` will get you the value of `n`, and setting `name` will set the value o
    alias: 'name'
  }
});

// Setting `name` will propagate to `n`
const 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

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

Valid options:

- autoIndex
- autoCreate

- bufferCommands
- bufferTimeoutMS
- capped
- collection
- discriminatorKey
- id
- _id
- minimize
- read
- writeConcern
- shardKey
- strict
- strictQuery
- toJSON
- toObject
- typeKey
- useNestedStrict
- validateBeforeSave
- versionKey
- optimisticConcurrency
- 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.

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

option: bufferTimeoutMS

If `bufferCommands` is on, this option sets the maximum amount of time Mongoose buffering will wait before throwing an error. If not specified, Mongoose will use 10000 (10 seconds).

```
// If an operation is buffered for more than 1 second, throw an error.
const schema = new Schema({..}, { bufferTimeoutMS: 1000 });
```

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.

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

option: discriminatorKey

When you define a discriminator, Mongoose adds a path to your schema that stores which discriminator a document is an instance of. By default, Mongoose adds an `__t` path, but you can set `discriminatorKey` to overwrite this default.

```
const baseSchema = new Schema({}, { discriminatorKey: 'type' });
const BaseModel = mongoose.model('Test', baseSchema);

const personSchema = new Schema({ name: String });
const PersonModel = BaseModel.discriminator('Person', personSchema);

const doc = new PersonModel({ name: 'James T. Kirk' });
// Without `discriminatorKey`, Mongoose would store the discriminator
// key in `__t` instead of `type`
doc.type; // 'Person'
```

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

// disabled id
const schema = new Schema({ name: String }, { id: false });
const Page = mongoose.model('Page', schema);
const 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
const schema = new Schema({ name: String });
const Page = mongoose.model('Page', schema);
const p = new Page({ name: 'mongodb.org' });
console.log(p); // { _id: '50341373e894ad16347efe01', name: 'mongodb.org' }

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

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

Model.create({ children: [{ name: 'Luke' }] }, (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.

```
const schema = new Schema({..}, { read: 'primary' });            // also aliased as 'p'
const schema = new Schema({..}, { read: 'primaryPreferred' });   // aliased as 'pp'
const schema = new Schema({..}, { read: 'secondary' });          // aliased as 's'
const schema = new Schema({..}, { read: 'secondaryPreferred' }); // aliased as 'sp'
const 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
const options = { replset: { strategy: 'ping' }};
mongoose.connect(uri, options);

const 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.

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

// set to false..
const thingSchema = new Schema({..}, { strict: false });
const 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.

```
const thingSchema = new Schema({..})
const Thing = mongoose.model('Thing', thingSchema);
const 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:

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

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

option: strictQuery

Mongoose supports a separate `strictQuery` option to avoid strict mode for query filters. This is because empty query filters cause Mongoose to return all documents in the model, which can cause issues.

```
const mySchema = new Schema({ field: Number }, { strict: true });
const MyModel = mongoose.model('Test', mySchema);
// Mongoose will filter out `notInSchema: 1` because `strict: true`, meaning this query will retur
```

```
// _all_ documents in the 'tests' collection
MyModel.find({ notInSchema: 1 });
```

The `strict` option does apply to updates. The `strictQuery` option is **just** for query filters.

```
// 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: false // Turn off strict mode for query filters
});
const MyModel = mongoose.model('Test', mySchema);
// Mongoose will strip out `notInSchema: 1` because `strictQuery` is false
MyModel.find({ notInSchema: 1 });
```

In general, we do **not** recommend passing user-defined objects as query filters:

```
// Don't do this!
const docs = await MyModel.find(req.query);

// Do this instead:
const docs = await MyModel.find({ name: req.query.name, age: req.query.age }).setOptions({ sanitiz
```

In Mongoose 6, `strictQuery` is equal to `strict` by default. However, you can override this behavior globally:

```
// Set `strictQuery` to `false`, so Mongoose doesn't strip out non-schema
// query filter properties by default.
// This does **not** affect `strict`.
mongoose.set('strictQuery', false);
```

option: toJSON

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

```
const schema = new Schema({ name: String });
schema.path('name').get(function (v) {
  return v + ' is my name';
});
schema.set('toJSON', { getters: true, virtuals: false });
const M = mongoose.model('Person', schema);
const 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 m
```

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 }`:

```
const schema = new Schema({ name: String });
schema.path('name').get(function(v) {
  return v + ' is my name';
});
schema.set('toObject', { getters: true });
const M = mongoose.model('Person', schema);
const 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'
const 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.

```
const 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.

```
const schema = new Schema({ name: String });
schema.set('validateBeforeSave', false);
schema.path('name').validate(function (value) {
    return value != null;
});
const M = mongoose.model('Person', schema);
const 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's default versioning is **not** a full optimistic concurrency solution. Mongoose's default versioning only operates on arrays as shown below.

```
// 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();
```

If you need optimistic concurrency support for `save()`, you can set the `optimisticConcurrency` option

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: optimisticConcurrency

Optimistic concurrency is a strategy to ensure the document you're updating didn't change between when you loaded it using `find()` or `findOne()`, and when you update it using `save()`.

For example, suppose you have a `House` model that contains a list of `photos`, and a `status` that represents whether this house shows up in searches. Suppose that a house that has status `'APPROVED'` must have at least two `photos`. You might implement the logic of approving a house document as shown below:

```
async function markApproved(id) {
  const house = await House.findOne({ _id });
  if (house.photos.length < 2) {
    throw new Error('House must have at least two photos!');
  }

  house.status = 'APPROVED';
  await house.save();
}
```

The `markApproved()` function looks right in isolation, but there might be a potential issue: what if another function removes the house's photos between the `findOne()` call and the `save()` call? For example, the below code will succeed:

```
const house = await House.findOne({ _id });
if (house.photos.length < 2) {
  throw new Error('House must have at least two photos!');
}

const house2 = await House.findOne({ _id });
house2.photos = [];
await house2.save();

// Marks the house as 'APPROVED' even though it has 0 photos!
house.status = 'APPROVED';
await house.save();
```

If you set the `optimisticConcurrency` option on the `House` model's schema, the above script will throw an error.

```
const House = mongoose.model('House', Schema({
  status: String,
  photos: [String]
}, { optimisticConcurrency: true }));

const house = await House.findOne({ _id });
if (house.photos.length < 2) {
  throw new Error('House must have at least two photos!');
}

const house2 = await House.findOne({ _id });
house2.photos = [];
await house2.save();

// Throws 'VersionError: No matching document found for id "..." version 0'
house.status = 'APPROVED';
await house.save();
```

option: collation

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

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

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

MyModel.create([{ name: 'val' }, { name: 'Val' }]).
  then(() => {
    return MyModel.find({ name: 'val' });
  }).
  then((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.

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

const update = { 'child.name': 'Luke Skywalker' };
const 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.

```
const childSchema = new Schema({}, { strict: false });
const parentSchema = new Schema({ child: childSchema },
  { strict: 'throw', useNestedStrict: true });
const Parent = mongoose.model('Parent', parentSchema);
Parent.update({}, { 'child.name': 'Luke Skywalker' }, 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.