mongoose

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.

```
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);
```