SchemaTypes

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?
- SchemaType Options
- Creating Custom Types
- The `schema.path()` Function
- Further Reading

- What is a SchemaType?

- The `type` Key
- SchemaType Options
- Usage Notes
- Getters
- Custom Types
- The `schema.path()` Function

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
- Schema

Example

```
const 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

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

const 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.

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

```
const 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:

```
const 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.
- `immutable`: boolean, defines path as immutable. Mongoose prevents you from changing immutable paths unless the parent document has `isNew: true`.
- `transform`: function, Mongoose calls this function when you call `Document#toJSON()` function, including when you `JSON.stringify()` a document.

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

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

const 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.

```
const 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
- `populate`: Object, sets default populate options

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.
- `populate`: Object, sets default populate options

Date

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

ObjectId

- `populate`: Object, sets default populate options

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 once validated, meaning that it will not throw on initialization, only when validated.

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.

```
const 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"
```

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 });
```

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*.

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

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

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

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

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

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

Mongoose supports a special `$*` syntax to populate all elements in a map. For example, suppose your `socialMediaHandles` map contains a `ref`:

```
const userSchema = new Schema({
  socialMediaHandles: {
    type: Map,
```

```
    of: new Schema({
      handle: String,
      oauth: {
        type: ObjectId,
        ref: 'OAuth'
      }
    })
  }
});
const User = mongoose.model('User', userSchema);
```

To populate every `socialMediaHandles` entry's `oauth` property, you should populate on `socialMediaHandles.$*.oauth`:

```
const user = await User.findOne().populate('socialMediaHandles.$*.oauth');
```

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}`);
```

Schemas

To declare a path as another schema, set `type` to the sub-schema's instance.

To set a default value based on the sub-schema's shape, simply set a default value, and the value will be cast based on the sub-schema's definition before being set during document creation.

```
const subSchema = new mongoose.Schema({
  // some schema definition here
});

const schema = new mongoose.Schema({
  data: {
    type: subSchema
    default: {}
  }
});
```

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.

```
const 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.

Further Reading

- An Introduction to Mongoose SchemaTypes
- Mongoose Schema Types