**Schema Types**

SchemaTypes take care of validation, casting, defaults, and other general options in our models. We can specify our types one of two ways:

// directly without options

var Person = new Schema({

title : String

});

// or with options

var Person = new Schema({

title : { type: String, lowercase: true }

});

In the example above we specified the lowercase option for strings which will lowercase the string whenever it is set. Options are functions that are called on each SchemaType. Each SchemaType has its own set of custom options.

**Available Schema Types**

**String**

* lowercase: {Boolean}

Creates a setter which calls .toLowerCase() on the value

* uppercase: {Boolean}

Creates a setter which calls .toUpperCase() on the value

* trim: {Boolean}

Creates a setter which calls .trim() on the value

* match: {RegExp}

Creates a RegExp based [validator](http://mongoosejs.com/docs/2.7.x/docs/validation.html). The value being set is .test()ed against the RegExp. If it does not pass, validation will fail.

* enum: {Array}

Creates an enum validator. If the value being set is not in this array, validation will fail.

**Number**

* min: {Number}

Creates a validator which checks that the value being set is not less than the value specified.

* max: {Number}

Creates a validator which checks that the value being set is not greater than the value specified.

**Date**

* no custom options

**Boolean**

* no custom options

**Buffer (v2.x only)**

* no custom options

**ObjectId**

To specify a type of ObjectId, use Schema.ObjectId in your declaration.

var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var Car = new Schema({ driver: Schema.ObjectId })

* no custom options

**Mixed**

An "anything goes" SchemaType, its flexibility comes at a trade-off of it being harder to maintain. Mixed is available either through Schema.Types.Mixed or by passing an empty object literal. The following are equivalent:

var Any = new Schema({ any: {} });

var Any = new Schema({ any: Schema.Types.Mixed });

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

person.anything = { x: [3, 4, { y: "changed" }] };

person.markModified('anything');

person.save(); // anything will now get saved

* no custom options

**Array**

Creates an array of SchemaTypes or [Embedded Documents](http://mongoosejs.com/docs/2.7.x/docs/embedded-documents.html).

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

var ToyBox = new Schema({

toys: [ToySchema]

, buffers: [Buffer]

, string: [String]

, numbers: [Number]

... etc

});

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

var Empty1 = new Schema({ any: [] });

var Empty2 = new Schema({ any: Array });

var Empty3 = new Schema({ any: [Schema.Types.Mixed] });

var Empty4 = new Schema({ any: [{}] });

* no custom options

**Additional options**

Besides the options listed above, all SchemaTypes share the following additional options.

* default: {Function|value} - Determines the default value for the path. All values are casted. If using a function, the value it returns will be casted as the default value.
* required: {Boolean} - If true, creates a validation rule requiring this path be set before saving occurs.
* select: {Boolean} - Specifies default path selection behavior. In other words, you can specify if this path should be included or excluded from query results by default.
* // Excluding `bio` from query results by default
* var Person = new Schema({ bio: { type: String, select: false }})
* var P = db.model('Person', Person);
* P.findOne(function (err, p) {
* console.log(p.bio) // undefined
* });
* // Including `name` in query results by default
* var Person = new Schema({
* name: { type: String, select: true }
* , age: Number
* })
* var P = db.model('Person', Person);
* P.findOne().select('age').exec(function (err, p) {
* console.log(p.isSelected('age')) // true
* console.log(p.isSelected('name')) // true
* });
* // can also be overridden on a query by query basis
* var Person = new Schema({ bio: { type: String, select: false }})
* var P = db.model('Person', Person);
* P.findOne().select('bio').exec(callback); // bio will be selected
* get: {Function} - Adds a getter for this path. See the [getters / setters](http://mongoosejs.com/docs/2.7.x/docs/getters-setters.html) docs for more detail.
* set: {Function} - Adds a setter for this path. See the [getters / setters](http://mongoosejs.com/docs/2.7.x/docs/getters-setters.html) docs for more detail.
* index: {Boolean|Object} - Tells Mongoose to ensure an index is created for this path. An object can be passed as well.
* var Person = new Schema({ name: { type: String, index: true }})
* var name = { type: String, index: { unique: true }}
* var Person = new Schema({ name: name })

Note: if the index already exists on the db, it will *not* be replaced.

* unique: {Boolean} - Tells Mongoose to ensure a unique index is created for this path. The following are equivalent:
* var Person = new Schema({ name: { type: String, unique: true }})
* var name = { type: String, index: { unique: true }}
* var Person = new Schema({ name: name })

Note: if the index already exists on the db, it will *not* be replaced.

* sparse: {Boolean} - Tells Mongoose to ensure a sparse index is created for this path. The following are equivalent:
* var Person = new Schema({ name: { type: String, sparse: true }})
* var name = { type: String, index: { sparse: true }}
* var Person = new Schema({ name: name })

Note: if the index already exists on the db, it will *not* be replaced.

* validate: {Function|RegExp|Array} - Creates a [validator](http://mongoosejs.com/docs/2.7.x/docs/validation.html) for this path.
* // passing a function
* function hasNumber (v) {
* return v.length && /\d/.test(v);
* }
* var street = { type: String, validate: hasNumber }
* var Person = new Schema({ street: street });
* // passing a RegExp
* var street = { type: String, validate: /\d/ }
* var Person = new Schema({ street: street });
* // passing an array
* var street = {
* type: String
* , validate: [hasNumber, 'street number required']
* }
* var Person = new Schema({ street: street })
* // or
* var street = {
* type: String
* , validate: [/\d/, 'street number required']
* }
* var Person = new Schema({ street: street });

For more detail about validation including async validation, see the [validation](http://mongoosejs.com/docs/2.7.x/docs/validation.html) page.

**Alternate options definition**

Instead of defining options when instanciating your Schema we can also access keys through the path function and add options there:

Person.path('age').max(400);

Person.path('meta.birth').set(function (v) {

// this is a setter

});

Person.path('title').validate(function (v) {

return v.length > 50;

});