Schematype

SchemaType()

Parameters

- path «String»
- [options] «SchemaTypeOptions» See SchemaTypeOptions docs
- [instance] «String»

SchemaType constructor. Do **not** instantiate `SchemaType` directly. Mongoose converts your schema paths into SchemaTypes automatically.

Example:

```
const schema = new Schema({ name: String });
schema.path('name') instanceof SchemaType; // true
```

SchemaType.prototype.cast()

Parameters

- value «Object» value to cast
- doc «Document» document that triggers the casting
- init «Boolean»

The function that Mongoose calls to cast arbitrary values to this SchemaType.

SchemaType.prototype.default()

Parameters

- val «Function|any» the default value

Returns:

- «defaultValue»

Sets a default value for this SchemaType.

Example:

```
const schema = new Schema({ n: { type: Number, default: 10 })
const M = db.model('M', schema)
const m = new M;
console.log(m.n) // 10
```

Defaults can be either `functions` which return the value to use as the default or the literal value itself. Either way, the value will be cast based on its schema type before being set during document creation.

Example:

```
// values are cast:
const schema = new Schema({ aNumber: { type: Number, default: 4.815162342 }})
const M = db.model('M', schema)
const m = new M;
console.log(m.aNumber) // 4.815162342

// default unique objects for Mixed types:
const schema = new Schema({ mixed: Schema.Types.Mixed });
```

```
schema.path('mixed').default(function () {
  return {};
});

// if we don't use a function to return object literals for Mixed defaults,
// each document will receive a reference to the same object literal creating
// a "shared" object instance:
const schema = new Schema({ mixed: Schema.Types.Mixed });
schema.path('mixed').default({});
const M = db.model('M', schema);
const m1 = new M;
m1.mixed.added = 1;
console.log(m1.mixed); // { added: 1 }
const m2 = new M;
console.log(m2.mixed); // { added: 1 }
```

SchemaType.prototype.get()

Parameters

- fn «Function»

Returns:

- «SchemaType» this

Adds a getter to this schematype.

Example:

```
function dob (val) {
  if (!val) return val;
  return (val.getMonth() + 1) + "/" + val.getDate() + "/" + val.getFullYear();
}

// defining within the schema
const s = new Schema({ born: { type: Date, get: dob })

// or by retreiving its SchemaType
const s = new Schema({ born: Date })
s.path('born').get(dob)
```

Getters allow you to transform the representation of the data as it travels from the raw mongodb document to the value that you see.

Suppose you are storing credit card numbers and you want to hide everything except the last 4 digits to the mongoose user. You can do so by defining a getter in the following way:

```
function obfuscate (cc) {
  return '****-****-****-' + cc.slice(cc.length-4, cc.length);
}
```

```
const AccountSchema = new Schema({
  creditCardNumber: { type: String, get: obfuscate }
});

const Account = db.model('Account', AccountSchema);

Account.findById(id, function (err, found) {
  console.log(found.creditCardNumber); // '****-****-****-1234'
});
```

Getters are also passed a second argument, the schematype on which the getter was defined. This allows for tailored behavior based on options passed in the schema.

```
function inspector (val, priorValue, schematype) {
  if (schematype.options.required) {
    return schematype.path + ' is required';
  } else {
    return schematype.path + ' is not';
  }
}

const VirusSchema = new Schema({
  name: { type: String, required: true, get: inspector },
  taxonomy: { type: String, get: inspector }
})

const Virus = db.model('Virus', VirusSchema);

Virus.findById(id, function (err, virus) {
  console.log(virus.name);     // name is required
  console.log(virus.taxonomy); // taxonomy is not
})
```

SchemaType.prototype.immutable()

Parameters

- bool «Boolean»

Returns:

- «SchemaType» this

Defines this path as immutable. Mongoose prevents you from changing immutable paths unless the parent document has `isNew: true`.

Example:

```
const schema = new Schema({
  name: { type: String, immutable: true },
  age: Number
});
const Model = mongoose.model('Test', schema);

await Model.create({ name: 'test' });
const doc = await Model.findOne();

doc.isNew; // false
doc.name = 'new name';
doc.name; // 'test', because `name` is immutable
```

Mongoose also prevents changing immutable properties using `updateOne()` and `updateMany()` based on strict mode.

Example:

```
// Mongoose will strip out the `name` update, because `name` is immutable
Model.updateOne({}, { $set: { name: 'test2' }, $inc: { age: 1 } });

// If `strict` is set to 'throw', Mongoose will throw an error if you
// update `name`
const err = await Model.updateOne({}, { name: 'test2' }, { strict: 'throw' }).
  then(() => null, err => err);
err.name; // StrictModeError

// If `strict` is `false`, Mongoose allows updating `name` even though
// the property is immutable.
Model.updateOne({}, { name: 'test2' }, { strict: false });
```

SchemaType.prototype.index()

Parameters

- options «Object|Boolean|String»

Returns:

- «SchemaType» this

Declares the index options for this schematype.

Example:

```
const s = new Schema({ name: { type: String, index: true })
const s = new Schema({ loc: { type: [Number], index: 'hashed' })
const s = new Schema({ loc: { type: [Number], index: '2d', sparse: true })
const s = new Schema({ loc: { type: [Number], index: { type: '2dsphere', sparse: true }})
const s = new Schema({ date: { type: Date, index: { unique: true, expires: '1d' }})
s.path('my.path').index(true);
s.path('my.date').index({ expires: 60 });
s.path('my.path').index({ unique: true, sparse: true });
```

NOTE:

Indexes are created in the background by default. If `background` is set to `false`, MongoDB will not execute any read/write operations you send until the index build. Specify `background: false` to override Mongoose's default.

SchemaType.prototype.ref()

Parameters

- ref «String|Model|Function» either a model name, a Model, or a function that returns a model name or model.

Returns:

- «SchemaType» this

Set the model that this path refers to. This is the option that populate looks at to determine the foreign collection it should query.

Example:

```
const userSchema = new Schema({ name: String });
const User = mongoose.model('User', userSchema);

const postSchema = new Schema({ user: mongoose.ObjectId });
postSchema.path('user').ref('User'); // Can set ref to a model name
postSchema.path('user').ref(User); // Or a model class
postSchema.path('user').ref(() => 'User'); // Or a function that returns the model name
postSchema.path('user').ref(() => User); // Or a function that returns the model class

// Or you can just declare the `ref` inline in your schema
const postSchema2 = new Schema({
  user: { type: mongoose.ObjectId, ref: User }
});
```

SchemaType.prototype.required()

Parameters

- required «Boolean|Function|Object» enable/disable the validator, or function that returns required boolean, or options object
    - [options.isRequired] «Boolean|Function» enable/disable the validator, or function that returns required boolean
    - [options.ErrorConstructor] «Function» custom error constructor. The constructor receives 1 parameter, an object containing the validator properties.

- [message] «String» optional custom error message

Returns:

- «SchemaType» this

Adds a required validator to this SchemaType. The validator gets added to the front of this SchemaType's validators array using `unshift()`.

Example:

```
const s = new Schema({ born: { type: Date, required: true })

// or with custom error message

const s = new Schema({ born: { type: Date, required: '{PATH} is required!' })

// or with a function

const s = new Schema({
  userId: ObjectId,
  username: {
    type: String,
    required: function() { return this.userId != null; }
  }
})

// or with a function and a custom message
const s = new Schema({
  userId: ObjectId,
  username: {
    type: String,
    required: [
      function() { return this.userId != null; },
      'username is required if id is specified'
    ]
  }
})

// or through the path API

s.path('name').required(true);

// with custom error messaging

s.path('name').required(true, 'grrr :( ');

// or make a path conditionally required based on a function
const isOver18 = function() { return this.age >= 18; };
s.path('voterRegistrationId').required(isOver18);
```

The required validator uses the SchemaType's `checkRequired` function to determine whether a given value satisfies the required validator. By default, a value satisfies the required validator if `val != null` (that is, if the value is not null nor undefined). However, most built-in mongoose schema types override the default `checkRequired` function:

SchemaType.prototype.select()

Parameters

- val «Boolean»

Returns:

- «SchemaType» this

Sets default `select()` behavior for this path.

Set to `true` if this path should always be included in the results, `false` if it should be excluded by default. This setting can be overridden at the query level.

Example:

```
T = db.model('T', new Schema({ x: { type: String, select: true }}));
T.find(..); // field x will always be selected ..
// .. unless overridden;
T.find().select('-x').exec(callback);
```

SchemaType.prototype.set()

Parameters

- fn «Function»

Returns:

- «SchemaType» this

Adds a setter to this schematype.

Example:

```
function capitalize (val) {
  if (typeof val !== 'string') val = '';
  return val.charAt(0).toUpperCase() + val.substring(1);
}

// defining within the schema
const s = new Schema({ name: { type: String, set: capitalize }});

// or with the SchemaType
const s = new Schema({ name: String })
s.path('name').set(capitalize);
```

Setters allow you to transform the data before it gets to the raw mongodb document or query.

Suppose you are implementing user registration for a website. Users provide an email and password, which gets saved to mongodb. The email is a string that you will want to normalize to lower case, in order to avoid one email having more than one account -- e.g., otherwise, avenue@q.com can be registered for 2 accounts via avenue@q.com and AvEnUe@Q.CoM.

You can set up email lower case normalization easily via a Mongoose setter.

```
function toLower(v) {
  return v.toLowerCase();
}

const UserSchema = new Schema({
  email: { type: String, set: toLower }
});

const User = db.model('User', UserSchema);

const user = new User({email: 'AVENUE@Q.COM'});
console.log(user.email); // 'avenue@q.com'

// or
const user = new User();
user.email = 'Avenue@Q.com';
console.log(user.email); // 'avenue@q.com'
User.updateOne({ _id: _id }, { $set: { email: 'AVENUE@Q.COM' } }); // update to 'avenue@q.com'
```

As you can see above, setters allow you to transform the data before it stored in MongoDB, or before executing a query.

NOTE: we could have also just used the built-in `lowercase: true` SchemaType option instead of defining our own function.

```
new Schema({ email: { type: String, lowercase: true }})
```

Setters are also passed a second argument, the schematype on which the setter was defined. This allows for tailored behavior based on options passed in the schema.

```
function inspector (val, priorValue, schematype) {
  if (schematype.options.required) {
    return schematype.path + ' is required';
  } else {
    return val;
  }
}

const VirusSchema = new Schema({
  name: { type: String, required: true, set: inspector },
  taxonomy: { type: String, set: inspector }
})

const Virus = db.model('Virus', VirusSchema);
const v = new Virus({ name: 'Parvoviridae', taxonomy: 'Parvovirinae' });
```

```
console.log(v.name);      // name is required
console.log(v.taxonomy); // Parvovirinae
```

You can also use setters to modify other properties on the document. If you're setting a property `name` on a document, the setter will run with `this` as the document. Be careful, in mongoose 5 setters will also run when querying by `name` with `this` as the query.

```
const nameSchema = new Schema({ name: String, keywords: [String] });
nameSchema.path('name').set(function(v) {
  // Need to check if `this` is a document, because in mongoose 5
  // setters will also run on queries, in which case `this` will be a
  // mongoose query object.
  if (this instanceof Document && v != null) {
    this.keywords = v.split(' ');
  }
  return v;
});
```

SchemaType.prototype.sparse()

Parameters

- bool «Boolean»

Returns:

- «SchemaType» this

Declares a sparse index.

Example:

```
const s = new Schema({ name: { type: String, sparse: true } });
s.path('name').index({ sparse: true });
```

SchemaType.prototype.text()

Parameters

- bool «Boolean»

Returns:

- «SchemaType» this

Declares a full text index.

Example:

```
const s = new Schema({name : {type: String, text : true })
 s.path('name').index({text : true});
```

SchemaType.prototype.transform()

Parameters

- fn «Function»

Returns:

- «SchemaType» this

Defines a custom function for transforming this path when converting a document to JSON.

Mongoose calls this function with one parameter: the current `value` of the path. Mongoose then uses the return value in the JSON output.

Example:

```
const schema = new Schema({
  date: { type: Date, transform: v => v.getFullYear() }
});
const Model = mongoose.model('Test', schema);

await Model.create({ date: new Date('2016-06-01') });
const doc = await Model.findOne();

doc.date instanceof Date; // true

doc.toJSON().date; // 2016 as a number
JSON.stringify(doc); // '{"_id":...,"date":2016}'
```

SchemaType.prototype.unique()

Parameters

- bool «Boolean»

Returns:

- «SchemaType» this

Declares an unique index.

Example:

```
const s = new Schema({ name: { type: String, unique: true }});
s.path('name').index({ unique: true });
```

NOTE: violating the constraint returns an `E11000` error from MongoDB when saving, not a Mongoose validation error.

SchemaType.prototype.validate()

Parameters

- obj «RegExp|Function|Object» validator function, or hash describing options
  - [obj.validator] «Function» validator function. If the validator function returns `undefined` or a truthy value, validation succeeds. If it returns falsy (except `undefined`) or throws an error, validation fails.
  - [obj.message] «String|Function» optional error message. If function, should return the error message as a string
  - [obj.propsParameter=false] «Boolean» If true, Mongoose will pass the validator properties object (with the `validator` function, `message`, etc.) as the 2nd arg to the validator function. This is disabled by default because many validators rely on positional args, so turning this on may cause unpredictable behavior in external validators.
- [errorMsg] «String|Function» optional error message. If function, should return the error message as a string
- [type] «String» optional validator type

Returns:

- «SchemaType» this

Adds validator(s) for this document path.

Validators always receive the value to validate as their first argument and must return `Boolean`. Returning `false` or throwing an error means validation failed.

The error message argument is optional. If not passed, the default generic error message template will be used.

Examples:

```
// make sure every value is equal to "something"
function validator (val) {
  return val == 'something';
}
new Schema({ name: { type: String, validate: validator }});

// with a custom error message
```

```
const custom = [validator, 'Uh oh, {PATH} does not equal "something".']
new Schema({ name: { type: String, validate: custom }});

// adding many validators at a time

const many = [
    { validator: validator, msg: 'uh oh' }
  , { validator: anotherValidator, msg: 'failed' }
]
new Schema({ name: { type: String, validate: many }});

// or utilizing SchemaType methods directly:

const schema = new Schema({ name: 'string' });
schema.path('name').validate(validator, 'validation of `{PATH}` failed with value `{VALUE}`');
```

Error message templates:

From the examples above, you may have noticed that error messages support basic templating. There are a few other template keywords besides `{PATH}` and `{VALUE}` too. To find out more, details are available here.

If Mongoose's built-in error message templating isn't enough, Mongoose supports setting the `message` property to a function.

```
schema.path('name').validate({
  validator: function() { return v.length > 5; },
  // `errors['name']` will be "name must have length 5, got 'foo'"
  message: function(props) {
    return `${props.path} must have length 5, got '${props.value}'`;
  }
});
```

To bypass Mongoose's error messages and just copy the error message that the validator throws, do this:

```
schema.path('name').validate({
  validator: function() { throw new Error('Oops!'); },
  // `errors['name']` will be "Oops!"
  message: function(props) { return props.reason.message; }
});
```

Asynchronous validation:

Mongoose supports validators that return a promise. A validator that returns a promise is called an *async validator*. Async validators run in parallel, and `validate()` will wait until all async validators have settled.

```
schema.path('name').validate({
  validator: function (value) {
    return new Promise(function (resolve, reject) {
      resolve(false); // validation failed
    });
  }
});
```

You might use asynchronous validators to retreive other documents from the database to validate against or to meet other I/O bound validation needs.

Validation occurs `pre('save')` or whenever you manually execute document#validate.

If validation fails during `pre('save')` and no callback was passed to receive the error, an `error` event will be emitted on your Models associated db connection, passing the validation error object along.

```
const conn = mongoose.createConnection(..);
conn.on('error', handleError);

const Product = conn.model('Product', yourSchema);
const dvd = new Product(..);
dvd.save(); // emits error on the `conn` above
```

If you want to handle these errors at the Model level, add an `error` listener to your Model as shown below.

```
// registering an error listener on the Model lets us handle errors more locally
Product.on('error', handleError);
```

Schematype.cast()

Parameters

- caster «Function|false» Function that casts arbitrary values to this type, or throws an error if casting failed

Returns:

- «Function»

Get/set the function used to cast arbitrary values to this type.

Example:

```
// Disallow `null` for numbers, and don't try to cast any values to
// numbers, so even strings like '123' will cause a CastError.
mongoose.Number.cast(function(v) {
  assert.ok(v === undefined || typeof v === 'number');
  return v;
});
```

Schematype.cast()

Parameters

- caster «Function|false» Function that casts arbitrary values to this type, or throws an error if casting failed

Returns:

- «Function»

Get/set the function used to cast arbitrary values to this particular schematype instance. Overrides `SchemaType.cast()`.

Example:

```
// Disallow `null` for numbers, and don't try to cast any values to
// numbers, so even strings like '123' will cause a CastError.
const number = new mongoose.Number('mypath', {});
number.cast(function(v) {
  assert.ok(v === undefined || typeof v === 'number');
  return v;
});
```

Schematype.checkRequired()

Parameters

- fn «Function»

Returns:

- «Function»

Override the function the required validator uses to check whether a value passes the `required` check. Override this on the individual SchemaType.

Example:

```
// Use this to allow empty strings to pass the `required` validator
mongoose.Schema.Types.String.checkRequired(v => typeof v === 'string');
```

Schematype.get()

Parameters

- getter «Function»

Returns:

- «this»

Attaches a getter for all instances of this schema type.

Example:

```
// Make all numbers round down
mongoose.Number.get(function(v) { return Math.floor(v); });
```

Schematype.set()

Parameters

- option «String» The name of the option you'd like to set (e.g. trim, lowercase, etc...)
- value «*» The value of the option you'd like to set.

Returns:

- «void»

Sets a default option for this schema type.

Example:

```
// Make all strings be trimmed by default
mongoose.SchemaTypes.String.set('trim', true);
```