Validation

Before we get into the specifics of validation syntax, please keep the following rules in mind:

- Validation is defined in the SchemaType
- Validation is middleware. Mongoose registers validation as a `pre('save')` hook on every schema by default.
- You can disable automatic validation before save by setting the validateBeforeSave option
- You can manually run validation using `doc.validate(callback)` or `doc.validateSync()`
- You can manually mark a field as invalid (causing validation to fail) by using `doc.invalidate(...)`
- Validators are not run on undefined values. The only exception is the `required` validator.
- Validation is asynchronously recursive; when you call Model#save, sub-document validation is executed as well. If an error occurs, your Model#save callback receives it
- Validation is customizable

```
const schema = new Schema({
  name: {
    type: String,
    required: true
  }
});
const Cat = db.model('Cat', schema);

// This cat has no name :(
const cat = new Cat();
cat.save(function(error) {
  assert.equal(error.errors['name'].message,
    'Path `name` is required.');

  error = cat.validateSync();
  assert.equal(error.errors['name'].message,
    'Path `name` is required.');
});
```

Built-in Validators

Mongoose has several built-in validators.

- All SchemaTypes have the built-in required validator. The required validator uses the SchemaType's `checkRequired()` function to determine if the value satisfies the required validator.
- Numbers have `min` and `max` validators.
- Strings have `enum`, `match`, `minLength`, and `maxLength` validators.

Each of the validator links above provide more information about how to enable them and customize their error messages.

```
const breakfastSchema = new Schema({
  eggs: {
```

```
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12
  },
  bacon: {
    type: Number,
    required: [true, 'Why no bacon?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea'],
    required: function() {
      return this.bacon > 3;
    }
  }
});
const Breakfast = db.model('Breakfast', breakfastSchema);

const badBreakfast = new Breakfast({
  eggs: 2,
  bacon: 0,
  drink: 'Milk'
});
let error = badBreakfast.validateSync();
assert.equal(error.errors['eggs'].message,
  'Too few eggs');
assert.ok(!error.errors['bacon']);
assert.equal(error.errors['drink'].message,
  '`Milk` is not a valid enum value for path `drink`.');

badBreakfast.bacon = 5;
badBreakfast.drink = null;

error = badBreakfast.validateSync();
assert.equal(error.errors['drink'].message, 'Path `drink` is required.');

badBreakfast.bacon = null;
error = badBreakfast.validateSync();
assert.equal(error.errors['bacon'].message, 'Why no bacon?');
```

Custom Error Messages

You can configure the error message for individual validators in your schema. There are two equivalent ways to set the validator error message:

- Array syntax: `min: [6, 'Must be at least 6, got {VALUE}']`
- Object syntax: `enum: { values: ['Coffee', 'Tea'], message: '{VALUE} is not supported' }`

Mongoose also supports rudimentary templating for error messages. Mongoose replaces `{VALUE}` with the value being validated.

```
const breakfastSchema = new Schema({
  eggs: {
    type: Number,
```

```
    min: [6, 'Must be at least 6, got {VALUE}'],
    max: 12
  },
  drink: {
    type: String,
    enum: {
      values: ['Coffee', 'Tea'],
      message: '{VALUE} is not supported'
    }
  }
});
const Breakfast = db.model('Breakfast', breakfastSchema);

const badBreakfast = new Breakfast({
  eggs: 2,
  drink: 'Milk'
});
let error = badBreakfast.validateSync();
assert.equal(error.errors['eggs'].message,
  'Must be at least 6, got 2');
assert.equal(error.errors['drink'].message, 'Milk is not supported');
```

The `unique` Option is Not a Validator

A common gotcha for beginners is that the `unique` option for schemas is *not* a validator. It's a convenient helper for building MongoDB unique indexes. See the FAQ for more information.

```
const uniqueUsernameSchema = new Schema({
  username: {
    type: String,
    unique: true
  }
});
const U1 = db.model('U1', uniqueUsernameSchema);
const U2 = db.model('U2', uniqueUsernameSchema);

const dup = [{ username: 'Val' }, { username: 'Val' }];
U1.create(dup, err => {
  // Race condition! This may save successfully, depending on whether
  // MongoDB built the index before writing the 2 docs.
});

// You need to wait for Mongoose to finish building the `unique`
// index before writing. You only need to build indexes once for
// a given collection, so you normally don't need to do this
// in production. But, if you drop the database between tests,
// you will need to use `init()` to wait for the index build to finish.
U2.init().
  then(() => U2.create(dup)).
  catch(error => {
    // Will error, but will *not* be a mongoose validation error, it will be
    // a duplicate key error.
    // See: https://masteringjs.io/tutorials/mongoose/e11000-duplicate-key
    assert.ok(error);
    assert.ok(!error.errors);
```

```
    assert.ok(error.message.indexOf('duplicate key error') !== -1);
  });
```

Custom Validators

If the built-in validators aren't enough, you can define custom validators to suit your needs.

Custom validation is declared by passing a validation function. You can find detailed instructions on how to do this in the `SchemaType#validate()` API docs.

```
const userSchema = new Schema({
  phone: {
    type: String,
    validate: {
      validator: function(v) {
        return /\d{3}-\d{3}-\d{4}/.test(v);
      },
      message: props => `${props.value} is not a valid phone number!`
    },
    required: [true, 'User phone number required']
  }
});

const User = db.model('user', userSchema);
const user = new User();
let error;

user.phone = '555.0123';
error = user.validateSync();
assert.equal(error.errors['phone'].message,
  '555.0123 is not a valid phone number!');

user.phone = '';
error = user.validateSync();
assert.equal(error.errors['phone'].message,
  'User phone number required');

user.phone = '201-555-0123';
// Validation succeeds! Phone number is defined
// and fits `DDD-DDD-DDDD`
error = user.validateSync();
assert.equal(error, null);
```

Async Custom Validators

Custom validators can also be asynchronous. If your validator function returns a promise (like an `async` function), mongoose will wait for that promise to settle. If the returned promise rejects, or fulfills with the value `false`, Mongoose will consider that a validation error.

```
const userSchema = new Schema({
  name: {
    type: String,
    // You can also make a validator async by returning a promise.
```

```
    validate: () => Promise.reject(new Error('Oops!'))
  },
  email: {
    type: String,
    // There are two ways for an promise-based async validator to fail:
    // 1) If the promise rejects, Mongoose assumes the validator failed with the given error.
    // 2) If the promise resolves to `false`, Mongoose assumes the validator failed and creates ar
    validate: {
      validator: () => Promise.resolve(false),
      message: 'Email validation failed'
    }
  }
});

const User = db.model('User', userSchema);
const user = new User();

user.email = 'test@test.co';
user.name = 'test';
user.validate().catch(error => {
  assert.ok(error);
  assert.equal(error.errors['name'].message, 'Oops!');
  assert.equal(error.errors['email'].message, 'Email validation failed');
});
```

Validation Errors

Errors returned after failed validation contain an `errors` object whose values are `ValidatorError` objects. Each ValidatorError has `kind`, `path`, `value`, and `message` properties. A ValidatorError also may have a `reason` property. If an error was thrown in the validator, this property will contain the error that was thrown.

```
const toySchema = new Schema({
  color: String,
  name: String
});

const validator = function(value) {
  return /red|white|gold/i.test(value);
};
toySchema.path('color').validate(validator,
  'Color `{VALUE}` not valid', 'Invalid color');
toySchema.path('name').validate(function(v) {
  if (v !== 'Turbo Man') {
    throw new Error('Need to get a Turbo Man for Christmas');
  }
  return true;
}, 'Name `{VALUE}` is not valid');

const Toy = db.model('Toy', toySchema);

const toy = new Toy({ color: 'Green', name: 'Power Ranger' });

toy.save(function(err) {
  // `err` is a ValidationError object
```

```
  // `err.errors.color` is a ValidatorError object
  assert.equal(err.errors.color.message, 'Color `Green` not valid');
  assert.equal(err.errors.color.kind, 'Invalid color');
  assert.equal(err.errors.color.path, 'color');
  assert.equal(err.errors.color.value, 'Green');

  // This is new in mongoose 5. If your validator throws an exception,
  // mongoose will use that message. If your validator returns `false`,
  // mongoose will use the 'Name `Power Ranger` is not valid' message.
  assert.equal(err.errors.name.message,
    'Need to get a Turbo Man for Christmas');
  assert.equal(err.errors.name.value, 'Power Ranger');
  // If your validator threw an error, the `reason` property will contain
  // the original error thrown, including the original stack trace.
  assert.equal(err.errors.name.reason.message,
    'Need to get a Turbo Man for Christmas');

  assert.equal(err.name, 'ValidationError');
});
```

Cast Errors

Before running validators, Mongoose attempts to coerce values to the correct type. This process is called *casting* the document. If casting fails for a given path, the `error.errors` object will contain a `CastError` object.

Casting runs before validation, and validation does not run if casting fails. That means your custom validators may assume `v` is `null`, `undefined`, or an instance of the type specified in your schema.

```
const vehicleSchema = new mongoose.Schema({
  numWheels: { type: Number, max: 18 }
});
const Vehicle = db.model('Vehicle', vehicleSchema);

const doc = new Vehicle({ numWheels: 'not a number' });
const err = doc.validateSync();

err.errors['numWheels'].name; // 'CastError'
// 'Cast to Number failed for value "not a number" at path "numWheels"'
err.errors['numWheels'].message;
```

Required Validators On Nested Objects

Defining validators on nested objects in mongoose is tricky, because nested objects are not fully fledged paths.

```
let personSchema = new Schema({
  name: {
    first: String,
    last: String
  }
});
```

```
assert.throws(function() {
  // This throws an error, because 'name' isn't a full fledged path
  personSchema.path('name').required(true);
}, /Cannot.*'required'/);

// To make a nested object required, use a single nested schema
const nameSchema = new Schema({
  first: String,
  last: String
});

personSchema = new Schema({
  name: {
    type: nameSchema,
    required: true
  }
});

const Person = db.model('Person', personSchema);

const person = new Person();
const error = person.validateSync();
assert.ok(error.errors['name']);
```

Update Validators

In the above examples, you learned about document validation. Mongoose also supports validation for `update()`, `updateOne()`, `updateMany()`, and `findOneAndUpdate()` operations. Update validators are off by default - you need to specify the `runValidators` option.

To turn on update validators, set the `runValidators` option for `update()`, `updateOne()`, `updateMany()`, or `findOneAndUpdate()`. Be careful: update validators are off by default because they have several caveats.

```
const toySchema = new Schema({
  color: String,
  name: String
});

const Toy = db.model('Toys', toySchema);

Toy.schema.path('color').validate(function(value) {
  return /red|green|blue/i.test(value);
}, 'Invalid color');

const opts = { runValidators: true };
Toy.updateOne({}, { color: 'not a color' }, opts, function(err) {
  assert.equal(err.errors.color.message,
    'Invalid color');
});
```

Update Validators and `this`

There are a couple of key differences between update validators and document validators. In the color validation function above, `this` refers to the document being validated when using document validation. However, when running update validators, the document being updated may not be in the server's memory, so by default the value of `this` is not defined.

```
const toySchema = new Schema({
  color: String,
  name: String
});

toySchema.path('color').validate(function(value) {
  // When running in `validate()` or `validateSync()`, the
  // validator can access the document using `this`.
  // Does **not** work with update validators.
  if (this.name.toLowerCase().indexOf('red') !== -1) {
    return value !== 'red';
  }
  return true;
});

const Toy = db.model('ActionFigure', toySchema);

const toy = new Toy({ color: 'red', name: 'Red Power Ranger' });
const error = toy.validateSync();
assert.ok(error.errors['color']);

const update = { color: 'red', name: 'Red Power Ranger' };
const opts = { runValidators: true };

Toy.updateOne({}, update, opts, function(error) {
  // The update validator throws an error:
  // "TypeError: Cannot read property 'toLowerCase' of undefined",
  // because `this` is **not** the document being updated when using
  // update validators
  assert.ok(error);
});
```

The `context` option

The `context` option lets you set the value of `this` in update validators to the underlying query.

```
toySchema.path('color').validate(function(value) {
  // When running update validators, `this` refers to the query object.
  if (this.getUpdate().$set.name.toLowerCase().indexOf('red') !== -1) {
    return value === 'red';
  }
  return true;
});

const Toy = db.model('Figure', toySchema);

const update = { color: 'blue', name: 'Red Power Ranger' };
// Note the context option
const opts = { runValidators: true, context: 'query' };
```

```
Toy.updateOne({}, update, opts, function(error) {
  assert.ok(error.errors['color']);
});
```

Update Validators Only Run On Updated Paths

The other key difference is that update validators only run on the paths specified in the update. For instance, in the below example, because 'name' is not specified in the update operation, update validation will succeed.

When using update validators, `required` validators **only** fail when you try to explicitly `$unset` the key.

```
const kittenSchema = new Schema({
  name: { type: String, required: true },
  age: Number
});

const Kitten = db.model('Kitten', kittenSchema);

const update = { color: 'blue' };
const opts = { runValidators: true };
Kitten.updateOne({}, update, opts, function() {
  // Operation succeeds despite the fact that 'name' is not specified
});

const unset = { $unset: { name: 1 } };
Kitten.updateOne({}, unset, opts, function(err) {
  // Operation fails because 'name' is required
  assert.ok(err);
  assert.ok(err.errors['name']);
});
```

Update Validators Only Run For Some Operations

One final detail worth noting: update validators **only** run on the following update operators:

- `$set`
- `$unset`
- `$push` (>= 4.8.0)
- `$addToSet` (>= 4.8.0)
- `$pull` (>= 4.12.0)
- `$pullAll` (>= 4.12.0)

For instance, the below update will succeed, regardless of the value of `number`, because update validators ignore `$inc`.

Also, `$push`, `$addToSet`, `$pull`, and `$pullAll` validation does **not** run any validation on the array itself, only individual elements of the array.

```
const testSchema = new Schema({
  number: { type: Number, max: 0 },
```

```
  arr: [{ message: { type: String, maxlength: 10 } }]
});

// Update validators won't check this, so you can still `$push` 2 elements
// onto the array, so long as they don't have a `message` that's too long.
testSchema.path('arr').validate(function(v) {
  return v.length < 2;
});

const Test = db.model('Test', testSchema);

let update = { $inc: { number: 1 } };
const opts = { runValidators: true };
Test.updateOne({}, update, opts, function() {
  // There will never be a validation error here
  update = { $push: [{ message: 'hello' }, { message: 'world' }] };
  Test.updateOne({}, update, opts, function(error) {
    // This will never error either even though the array will have at
    // least 2 elements.
  });
});
```