Middleware

Middleware (also called pre and post *hooks*) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.

Types of Middleware

Mongoose has 4 types of middleware: document middleware, model middleware, aggregate middleware, and query middleware. Document middleware is supported for the following document functions. In document middleware functions, `this` refers to the document.

- validate
- save
- remove
- updateOne
- deleteOne
- init (note: init hooks are synchronous)

Query middleware is supported for the following Model and Query functions. In query middleware functions, `this` refers to the query.

- count
- countDocuments
- deleteMany
- deleteOne
- estimatedDocumentCount
- find
- findOne
- findOneAndDelete
- findOneAndRemove
- findOneAndReplace
- findOneAndUpdate
- remove
- replaceOne
- update

- updateOne
- updateMany

Aggregate middleware is for `MyModel.aggregate()`. Aggregate middleware executes when you call `exec()` on an aggregate object. In aggregate middleware, `this` refers to the aggregation object.

- aggregate

Model middleware is supported for the following model functions. In model middleware functions, `this` refers to the model.

- insertMany

All middleware types support pre and post hooks. How pre and post hooks work is described in more detail below.

Note: If you specify `schema.pre('remove')`, Mongoose will register this middleware for `doc.remove()` by default. If you want to your middleware to run on `Query.remove()` use `schema.pre('remove', { query: true, document: false }, fn)`.

Note: Unlike `schema.pre('remove')`, Mongoose registers `updateOne` and `deleteOne` middleware on `Query#updateOne()` and `Query#deleteOne()` by default. This means that both `doc.updateOne()` and `Model.updateOne()` trigger `updateOne` hooks, but `this` refers to a query, not a document. To register `updateOne` or `deleteOne` middleware as document middleware, use `schema.pre('updateOne', { document: true, query: false })`.

Note: The `create()` function fires `save()` hooks.

Pre

Pre middleware functions are executed one after another, when each middleware calls `next`.

```
const schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

In mongoose 5.x, instead of calling `next()` manually, you can use a function that returns a promise. In particular, you can use `async/await`.

```
schema.pre('save', function() {
  return doStuff().
    then(() => doMoreStuff());
});

// Or, in Node.js >= 7.6.0:
schema.pre('save', async function() {
  await doStuff();
  await doMoreStuff();
});
```

If you use `next()`, the `next()` call does **not** stop the rest of the code in your middleware function from executing. Use the early `return` pattern to prevent the rest of your middleware function from running when you call `next()`.

```
const schema = new Schema(..);
schema.pre('save', function(next) {
  if (foo()) {
    console.log('calling next!');
    // `return next();` will make sure the rest of this function doesn't run
    /*return*/ next();
  }
  // Unless you comment out the `return` above, 'after next' will print
  console.log('after next');
});
```

Use Cases

Middleware are useful for atomizing model logic. Here are some other ideas:

- complex validation
- removing dependent documents (removing a user removes all their blogposts)
- asynchronous defaults
- asynchronous tasks that a certain action triggers

Errors in Pre Hooks

If any pre hook errors out, mongoose will not execute subsequent middleware or the hooked function. Mongoose will instead pass an error to the callback and/or reject the returned promise. There are several ways to report an error in middleware:

```
schema.pre('save', function(next) {
  const err = new Error('something went wrong');
  // If you call `next()` with an argument, that argument is assumed to be
  // an error.
  next(err);
});

schema.pre('save', function() {
  // You can also return a promise that rejects
  return new Promise((resolve, reject) => {
    reject(new Error('something went wrong'));
  });
});

schema.pre('save', function() {
  // You can also throw a synchronous error
  throw new Error('something went wrong');
});

schema.pre('save', async function() {
  await Promise.resolve();
  // You can also throw an error in an `async` function
  throw new Error('something went wrong');
});

// later...

// Changes will not be persisted to MongoDB because a pre hook errored out
myDoc.save(function(err) {
  console.log(err.message); // something went wrong
});
```

Calling `next()` multiple times is a no-op. If you call `next()` with an error `err1` and then throw an error `err2`, mongoose will report `err1`.

Post middleware

post middleware are executed *after* the hooked method and all of its `pre` middleware have completed.

```
schema.post('init', function(doc) {
  console.log('%s has been initialized from the db', doc._id);
});
schema.post('validate', function(doc) {
  console.log('%s has been validated (but not saved yet)', doc._id);
});
schema.post('save', function(doc) {
  console.log('%s has been saved', doc._id);
});
schema.post('remove', function(doc) {
  console.log('%s has been removed', doc._id);
});
```

Asynchronous Post Hooks

If your post hook function takes at least 2 parameters, mongoose will assume the second parameter is a `next()` function that you will call to trigger the next middleware in the sequence.

```
// Takes 2 parameters: this is an asynchronous post hook
schema.post('save', function(doc, next) {
  setTimeout(function() {
    console.log('post1');
    // Kick off the second post hook
    next();
  }, 10);
});

// Will not execute until the first middleware calls `next()`
schema.post('save', function(doc, next) {
  console.log('post2');
  next();
});
```

Define Middleware Before Compiling Models

Calling `pre()` or `post()` after compiling a model does **not** work in Mongoose in general. For example, the below `pre('save')` middleware will not fire.

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

// Compile a model from the schema
const User = mongoose.model('User', schema);

// Mongoose will **not** call the middleware function, because
// this middleware was defined after the model was compiled
schema.pre('save', () => console.log('Hello from pre save'));

new User({ name: 'test' }).save();
```

This means that you must add all middleware and plugins **before** calling `mongoose.model()`. The below script will print out "Hello from pre save":

```
const schema = new mongoose.Schema({ name: String });
// Mongoose will call this middleware function, because this script adds
// the middleware to the schema before compiling the model.
schema.pre('save', () => console.log('Hello from pre save'));

// Compile a model from the schema
const User = mongoose.model('User', schema);

new User({ name: 'test' }).save();
```

As a consequence, be careful about exporting Mongoose models from the same file that you define your schema. If you choose to use this pattern, you must define global plugins **before** calling `require()` on your model file.

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

// Once you `require()` this file, you can no longer add any middleware
// to this schema.
module.exports = mongoose.model('User', schema);
```

Save/Validate Hooks

The `save()` function triggers `validate()` hooks, because mongoose has a built-in `pre('save')` hook that calls `validate()`. This means that all `pre('validate')` and `post('validate')` hooks get called **before** any `pre('save')` hooks.

```
schema.pre('validate', function() {
  console.log('this gets printed first');
});
schema.post('validate', function() {
  console.log('this gets printed second');
});
schema.pre('save', function() {
  console.log('this gets printed third');
});
schema.post('save', function() {
  console.log('this gets printed fourth');
});
```

Naming Conflicts

Mongoose has both query and document hooks for `remove()`.

```
schema.pre('remove', function() { console.log('Removing!'); });

// Prints "Removing!"
doc.remove();

// Does **not** print "Removing!". Query middleware for `remove` is not
// executed by default.
Model.remove();
```

You can pass options to `Schema.pre()` and `Schema.post()` to switch whether Mongoose calls your `remove()` hook for `Document.remove()` or `Model.remove()`. Note here that you need to set both `document` and `query` properties in the passed object:

```
// Only document middleware
schema.pre('remove', { document: true, query: false }, function() {
  console.log('Removing doc!');
});

// Only query middleware. This will get called when you do `Model.remove()`
// but not `doc.remove()`.
schema.pre('remove', { query: true, document: false }, function() {
  console.log('Removing!');
});
```

Notes on findAndUpdate() and Query Middleware

Pre and post `save()` hooks are **not** executed on `update()`, `findOneAndUpdate()`, etc. You can see a more detailed discussion why in this GitHub issue. Mongoose 4.0 introduced distinct hooks for these functions.

```
schema.pre('find', function() {
  console.log(this instanceof mongoose.Query); // true
  this.start = Date.now();
});

schema.post('find', function(result) {
  console.log(this instanceof mongoose.Query); // true
  // prints returned documents
  console.log('find() returned ' + JSON.stringify(result));
  // prints number of milliseconds the query took
  console.log('find() took ' + (Date.now() - this.start) + ' millis');
});
```

Query middleware differs from document middleware in a subtle but important way: in document middleware, `this` refers to the document being updated. In query middleware, mongoose doesn't necessarily have a reference to the document being updated, so `this` refers to the **query** object rather than the document being updated.

For instance, if you wanted to add an `updatedAt` timestamp to every `updateOne()` call, you would use the following pre hook.

```
schema.pre('updateOne', function() {
  this.set({ updatedAt: new Date() });
});
```

You **cannot** access the document being updated in `pre('updateOne')` or `pre('findOneAndUpdate')` query middleware. If you need to access the document that will be updated, you need to execute an explicit query for the document.

```
schema.pre('findOneAndUpdate', async function() {
  const docToUpdate = await this.model.findOne(this.getQuery());
  console.log(docToUpdate); // The document that `findOneAndUpdate()` will modify
});
```

However, if you define `pre('updateOne')` document middleware, `this` will be the document being updated. That's because `pre('updateOne')` document middleware hooks into `Document#updateOne()` rather than `Query#updateOne()`.

```
schema.pre('updateOne', { document: true, query: false }, function() {
  console.log('Updating');
});
const Model = mongoose.model('Test', schema);

const doc = new Model();
await doc.updateOne({ $set: { name: 'test' } }); // Prints "Updating"

// Doesn't print "Updating", because `Query#updateOne()` doesn't fire
// document middleware.
await Model.updateOne({}, { $set: { name: 'test' } });
```

Error Handling Middleware

New in 4.5.0

Middleware execution normally stops the first time a piece of middleware calls `next()` with an error. However, there is a special kind of post middleware called "error handling middleware" that executes specifically when an error occurs. Error handling middleware is useful for reporting errors and making error messages more readable.

Error handling middleware is defined as middleware that takes one extra parameter: the 'error' that occurred as the first parameter to the function. Error handling middleware can then transform the error however you want.

```
const schema = new Schema({
  name: {
    type: String,
    // Will trigger a MongoServerError with code 11000 when
    // you save a duplicate
    unique: true
  }
});

// Handler **must** take 3 parameters: the error that occurred, the document
// in question, and the `next()` function
schema.post('save', function(error, doc, next) {
  if (error.name === 'MongoServerError' && error.code === 11000) {
    next(new Error('There was a duplicate key error'));
  } else {
    next();
  }
});

// Will trigger the `post('save')` error handler
Person.create([{ name: 'Axl Rose' }, { name: 'Axl Rose' }]);
```

Error handling middleware also works with query middleware. You can also define a post `update()` hook that will catch MongoDB duplicate key errors.

```
// The same E11000 error can occur when you call `update()`
// This function **must** take 3 parameters. If you use the
// `passRawResult` function, this function **must** take 4
// parameters
schema.post('update', function(error, res, next) {
  if (error.name === 'MongoServerError' && error.code === 11000) {
    next(new Error('There was a duplicate key error'));
  } else {
    next(); // The `update()` call will still error out.
  }
});

const people = [{ name: 'Axl Rose' }, { name: 'Slash' }];
Person.create(people, function(error) {
  Person.update({ name: 'Slash' }, { $set: { name: 'Axl Rose' } }, function(error) {
    // `error.message` will be "There was a duplicate key error"
  });
});
```

Error handling middleware can transform an error, but it can't remove the error. Even if you call `next()` with no error as shown above, the function call will still error out.

Aggregation Hooks

You can also define hooks for the `Model.aggregate()` function. In aggregation middleware functions, `this` refers to the Mongoose `Aggregate` object. For example, suppose you're implementing soft deletes on a `Customer` model by adding an `isDeleted` property. To make sure `aggregate()` calls only look at customers that aren't soft deleted, you can use the below middleware to add a `$match` stage to the beginning of each aggregation pipeline.

```
customerSchema.pre('aggregate', function() {
  // Add a $match state to the beginning of each pipeline.
  this.pipeline().unshift({ $match: { isDeleted: { $ne: true } } });
});
```

The `Aggregate#pipeline()` function lets you access the MongoDB aggregation pipeline that Mongoose will send to the MongoDB server. It is useful for adding stages to the beginning of the pipeline from middleware.

Synchronous Hooks

Certain Mongoose hooks are synchronous, which means they do **not** support functions that return promises or receive a `next()` callback. Currently, only `init` hooks are synchronous, because the `init()` function is synchronous. Below is an example of using pre and post init hooks.

```
const schema = new Schema({ title: String, loadedAt: Date });

schema.pre('init', pojo => {
  assert.equal(pojo.constructor.name, 'Object'); // Plain object before init
});

const now = new Date();
schema.post('init', doc => {
  assert.ok(doc instanceof mongoose.Document); // Mongoose doc after init
  doc.loadedAt = now;
});

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

return Test.create({ title: 'Casino Royale' }).
  then(doc => Test.findById(doc)).
  then(doc => assert.equal(doc.loadedAt.valueOf(), now.valueOf()));
```

To report an error in an init hook, you must throw a **synchronous** error. Unlike all other middleware, init middleware does **not** handle promise rejections.

```
const schema = new Schema({ title: String });

const swallowedError = new Error('will not show');
// init hooks do **not** handle async errors or any sort of async behavior
schema.pre('init', () => Promise.reject(swallowedError));
schema.post('init', () => { throw Error('will show'); });

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

return Test.create({ title: 'Casino Royale' }).
  then(doc => Test.findById(doc)).
  catch(error => assert.equal(error.message, 'will show'));
```

Next Up

Now that we've covered middleware, let's take a look at Mongoose's approach to faking JOINs with its query population helper.