mongoose

API Docs

Index

Mongoose()

Mongoose constructor.

The exports object of the `mongoose` module is an instance of this class. Most apps will only use this one instance.

Mongoose.prototype.Aggregate()

The Mongoose Aggregate constructor

Mongoose.prototype.CastError()

Parameters

- type «String» The name of the type
- value «Any» The value that failed to cast
- path «String» The path `a.b.c` in the doc where this cast error occurred
- [reason] «Error» The original error that was thrown

The Mongoose CastError constructor

Mongoose.prototype.Collection()

The Mongoose Collection constructor

Mongoose.prototype.Connection()

The Mongoose Connection constructor

Mongoose.prototype.Decimal128

The Mongoose Decimal128 SchemaType. Used for declaring paths in your schema that should be 128-bit decimal floating points. Do not use this to create a new Decimal128 instance, use `mongoose.Types.Decimal128` instead.

Example:

```
const vehicleSchema = new Schema({ fuelLevel: mongoose.Decimal128 });
```

Mongoose.prototype.Document()

The Mongoose Document constructor.

Mongoose.prototype.DocumentProvider()

The Mongoose DocumentProvider constructor. Mongoose users should not have to use this directly

Mongoose.prototype.Error()

The MongooseError constructor.

Mongoose.prototype.Mixed

The Mongoose Mixed SchemaType. Used for declaring paths in your schema that Mongoose's change tracking, casting, and validation should ignore.

Example:

```
const schema = new Schema({ arbitrary: mongoose.Mixed });
```

Mongoose.prototype.Model()

The Mongoose Model constructor.

Mongoose.prototype.Mongoose()

The Mongoose constructor

The exports of the mongoose module is an instance of this class.

Example:

```
var mongoose = require('mongoose');
var mongoose2 = new mongoose.Mongoose();
```

Mongoose.prototype.ObjectId

The Mongoose ObjectId SchemaType. Used for declaring paths in your schema that should be MongoDB ObjectIds. Do not use this to create a new ObjectId instance, use `mongoose.Types.ObjectId` instead.

Example:

```
const childSchema = new Schema({ parentId: mongoose.ObjectId });
```

Mongoose.prototype.Promise

The Mongoose Promise constructor.

Mongoose.prototype.PromiseProvider()

Storage layer for mongoose promises

Mongoose.prototype.Query()

The Mongoose Query constructor.

Mongoose.prototype.STATES

Expose connection states for user-land

Mongoose.prototype.Schema()

The Mongoose Schema constructor

Example:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var CatSchema = new Schema(..);
```

Mongoose.prototype.SchemaType()

The Mongoose SchemaType constructor

Mongoose.prototype.SchemaTypes

The various Mongoose SchemaTypes.

Note:

Alias of mongoose.Schema.Types for backwards compatibility.

Mongoose.prototype.Types

The various Mongoose Types.

Example:

```
var mongoose = require('mongoose');
var array = mongoose.Types.Array;
```

Types:

- ObjectId
- Buffer
- SubDocument
- Array
- DocumentArray

Using this exposed access to the `ObjectId` type, we can construct ids on demand.

```
var ObjectId = mongoose.Types.ObjectId;
var id1 = new ObjectId;
```

Mongoose.prototype.VirtualType()

The Mongoose VirtualType constructor

Mongoose.prototype.connect()

Parameters

- uri(s) «String»
- [options] «Object» passed down to the MongoDB driver's `connect()` function, except for 4 mongoose-specific options explained below.
    - [options.dbName] «String» The name of the database we want to use. If not provided, use database name from connection string.
    - [options.user] «String» username for authentication, equivalent to `options.auth.user`. Maintained for backwards compatibility.
    - [options.pass] «String» password for authentication, equivalent to `options.auth.password`. Maintained for backwards compatibility.
    - [options.autoIndex=true] «Boolean» Mongoose-specific option. Set to false to disable automatic index creation for all models associated with this connection.
    - [options.bufferCommands=true] «Boolean» Mongoose specific option. Set to false to disable buffering on all models associated with this connection.
    - [options.useCreateIndex=true] «Boolean» Mongoose-specific option. If `true`, this connection will use `createIndex()` instead of `ensureIndex()` for automatic index builds via `Model.init()`.
    - [options.useFindAndModify=true] «Boolean» True by default. Set to `false` to make `findOneAndUpdate()` and `findOneAndRemove()` use native `findOneAndUpdate()` rather than `findAndModify()`.
    - [options.useNewUrlParser=false] «Boolean» False by default. Set to `true` to make all connections set the `useNewUrlParser` option by default.
- [callback] «Function»

Returns:

- «Promise» resolves to `this` if connection succeeded

Opens the default mongoose connection.

Example:

```
mongoose.connect('mongodb://user:pass@localhost:port/database');

// replica sets
var uri = 'mongodb://user:pass@localhost:port,anotherhost:port,yetanother:port/mydatabas
mongoose.connect(uri);

// with options
mongoose.connect(uri, options);

// optional callback that gets fired when initial connection completed
var uri = 'mongodb://nonexistent.domain:27000';
mongoose.connect(uri, function(error) {
  // if error is truthy, the initial connection failed.
})
```

Mongoose.prototype.connection

Returns:

- «Connection»

The default connection of the mongoose module.

Example:

```
var mongoose = require('mongoose');
mongoose.connect(...);
mongoose.connection.on('error', cb);
```

This is the connection used by default for every model created using mongoose.model.

Mongoose.prototype.createConnection()

Parameters

- [uri] «String» a mongodb:// URI
- [options] «Object» passed down to the MongoDB driver's `connect()` function, except for 4 mongoose-specific options explained below.
  - [options.user] «String» username for authentication, equivalent to `options.auth.user`. Maintained for backwards compatibility.
  - [options.pass] «String» password for authentication, equivalent to `options.auth.password`. Maintained for backwards compatibility.
  - [options.autoIndex=true] «Boolean» Mongoose-specific option. Set to false to disable automatic index creation for all models associated with this connection.
  - [options.bufferCommands=true] «Boolean» Mongoose specific option. Set to false to disable buffering on all models associated with this connection.

Returns:

- «Connection» the created Connection object. Connections are thenable, so you can do `await mongoose.createConnection()`

Creates a Connection instance.

Each `connection` instance maps to a single database. This method is helpful when mangaging multiple db connections.

Options passed take precedence over options included in connection strings.

Example:

```
// with mongodb:// URI
db = mongoose.createConnection('mongodb://user:pass@localhost:port/database');

// and options
var opts = { db: { native_parser: true }}
db = mongoose.createConnection('mongodb://user:pass@localhost:port/database', opts);

// replica sets
db = mongoose.createConnection('mongodb://user:pass@localhost:port,anotherhost:port,yeta

// and options
var opts = { replset: { strategy: 'ping', rs_name: 'testSet' }}
db = mongoose.createConnection('mongodb://user:pass@localhost:port,anotherhost:port,yeta

// and options
var opts = { server: { auto_reconnect: false }, user: 'username', pass: 'mypassword' }
db = mongoose.createConnection('localhost', 'database', port, opts)

// initialize now, connect later
db = mongoose.createConnection();
db.open('localhost', 'database', port, [opts]);
```

Mongoose.prototype.deleteModel()

Parameters

- name «String|RegExp» if string, the name of the model to remove. If regexp, removes all models whose name matches the regexp.

Returns:

- «Mongoose» this

Removes the model named `name` from the default connection, if it exists. You can use this function to clean up any models you created in your tests to prevent OverwriteModelErrors.

Equivalent to `mongoose.connection.deleteModel(name)`.

Example:

```
mongoose.model('User', new Schema({ name: String }));
console.log(mongoose.model('User')); // Model object
mongoose.deleteModel('User');
console.log(mongoose.model('User')); // undefined

// Usually useful in a Mocha `afterEach()` hook
```

```
afterEach(function() {
  mongoose.deleteModel(/.+/); // Delete every model
});
```

Mongoose.prototype.disconnect()

Parameters

- [callback] «Function» called after all connection close, or when first error occurred.

Returns:

- «Promise» resolves when all connections are closed, or rejects with the first error that occurred.

Runs `.close()` on all connections in parallel.

Mongoose.prototype.get()

Parameters

- key «String»

Gets mongoose options

Example:

```
mongoose.get('test') // returns the 'test' value
```

Mongoose.prototype.model()

Parameters

- name «String|Function» model name or class extending Model
- [schema] «Schema»
- [collection] «String» name (optional, inferred from model name)
- [skipInit] «Boolean» whether to skip initialization (defaults to false)

Returns:

- «Model»

Defines a model or retrieves it.

Models defined on the `mongoose` instance are available to all connection created by the same `mongoose` instance.

Example:

```
var mongoose = require('mongoose');

// define an Actor model with this mongoose instance
mongoose.model('Actor', new Schema({ name: String }));

// create a new connection
var conn = mongoose.createConnection(..);

// retrieve the Actor model
var Actor = conn.model('Actor');
```

When no `collection` argument is passed, Mongoose uses the model name. If you don't like this behavior, either pass a collection name, use `mongoose.pluralize()`, or set your schemas collection name option.

Example:

```
var schema = new Schema({ name: String }, { collection: 'actor' });

// or

schema.set('collection', 'actor');

// or

var collectionName = 'actor'
var M = mongoose.model('Actor', schema, collectionName)
```

Mongoose.prototype.modelNames()

Returns:

- «Array»

Returns an array of model names created on this instance of Mongoose.

Note:

Does not include names of models created using `connection.model()`.

Mongoose.prototype.mongo

The node-mongodb-native driver Mongoose uses.

Mongoose.prototype.mquery

The mquery query builder Mongoose uses.

Mongoose.prototype.now()

Mongoose uses this function to get the current time when setting timestamps. You may stub out this function using a tool like Sinon for testing.

Mongoose.prototype.plugin()

Parameters

- fn «Function» plugin callback
- [opts] «Object» optional options

Returns:

- «Mongoose» this

Declares a global plugin executed on all Schemas.

Equivalent to calling `.plugin(fn)` on each Schema you create.

Mongoose.prototype.pluralize()

Parameters

- [fn] «Function|null» overwrites the function used to pluralize collection names

Returns:

- «Function,null» the current function used to pluralize collection names, defaults to the legacy function from `mongoose-legacy-pluralize`.

Getter/setter around function for pluralizing collection names.

Mongoose.prototype.set()

Parameters

- key «String»
- value «String|Function|Boolean»

Sets mongoose options

Example:

```
mongoose.set('test', value) // sets the 'test' option to `value`

mongoose.set('debug', true) // enable logging collection methods + arguments to the cons

mongoose.set('debug', function(collectionName, methodName, arg1, arg2...) {}); // use cu
```

Currently supported options are

- 'debug': prints the operations mongoose sends to MongoDB to the console
- 'bufferCommands': enable/disable mongoose's buffering mechanism for all connections and models
- 'useCreateIndex': false by default. Set to `true` to make Mongoose's default index build use `createIndex()` instead of `ensureIndex()` to avoid deprecation warnings from the MongoDB driver.
- 'useFindAndModify': true by default. Set to `false` to make `findOneAndUpdate()` and `findOneAndRemove()` use native `findOneAndUpdate()` rather than `findAndModify()`.
- 'useNewUrlParser': false by default. Set to `true` to make all connections set the `useNewUrlParser` option by default
- 'cloneSchemas': false by default. Set to `true` to `clone()` all schemas before compiling into a model.
- 'applyPluginsToDiscriminators': false by default. Set to true to apply global plugins to discriminator schemas. This typically isn't necessary because plugins are applied to the base schema and discriminators copy all middleware, methods, statics, and properties from the base schema.

- 'objectIdGetter': true by default. Mongoose adds a getter to MongoDB ObjectId's called `_id` that returns `this` for convenience with populate. Set this to false to remove the getter.
- 'runValidators': false by default. Set to true to enable update validators for all validators by default.
- 'toObject': `{ transform: true, flattenDecimals: true }` by default. Overwrites default objects to `toObject()`
- 'toJSON': `{ transform: true, flattenDecimals: true }` by default. Overwrites default objects to `toJSON()`, for determining how Mongoose documents get serialized by `JSON.stringify()`
- 'strict': true by default, may be `false`, `true`, or `'throw'`. Sets the default strict mode for schemas.
- 'selectPopulatedPaths': true by default. Set to false to opt out of Mongoose adding all fields that you `populate()` to your `select()`. The schema-level option `selectPopulatedPaths` overwrites this one.

Mongoose.prototype.startSession()

Parameters

- [options] «Object» see the mongodb driver options
  - [options.causalConsistency=true] «Boolean» set to false to disable causal consistency
- [callback] «Function»

Returns:

- «Promise<ClientSession>» promise that resolves to a MongoDB driver `ClientSession`

Requires MongoDB >= 3.6.0. Starts a MongoDB session for benefits like causal consistency, retryable writes, and transactions.

Calling `mongoose.startSession()` is equivalent to calling `mongoose.connection.startSession()`. Sessions are scoped to a connection, so calling `mongoose.startSession()` starts a session on the default mongoose connection.

Mongoose.prototype.version

The Mongoose version

Example

```
console.log(mongoose.version); // '5.x.x'
```

Schema

Schema()

Parameters

- definition «Object»
- [options] «Object»

Schema constructor.

Example:

```
var child = new Schema({ name: String });
var schema = new Schema({ name: String, age: Number, children: [child] });
var Tree = mongoose.model('Tree', schema);

// setting schema options
new Schema({ name: String }, { _id: false, autoIndex: false })
```

Options:

- autoIndex: bool - defaults to null (which means use the connection's autoIndex option)
- autoCreate: bool - defaults to null (which means use the connection's autoCreate option)
- bufferCommands: bool - defaults to true
- capped: bool - defaults to false
- collection: string - no default
- id: bool - defaults to true
- _id: bool - defaults to true
- `minimize` : bool - controls document#toObject behavior when called manually - defaults to true
- read: string
- writeConcern: object - defaults to null, use to override the MongoDB server's default write concern settings
- shardKey: bool - defaults to `null`
- strict: bool - defaults to true
- toJSON - object - no default
- toObject - object - no default
- typeKey - string - defaults to 'type'
- useNestedStrict - boolean - defaults to false
- validateBeforeSave - bool - defaults to `true`
- versionKey: string - defaults to "__v"

- collation: object - defaults to null (which means use no collation)
- selectPopulatedPaths: boolean - defaults to `true`

Note:

When nesting schemas, (`children` in the example above), always declare the child schema first before passing it into its parent.

Schema.Types

The various built-in Mongoose Schema Types.

Example:

```
var mongoose = require('mongoose');
var ObjectId = mongoose.Schema.Types.ObjectId;
```

Types:

- String
- Number
- Boolean | Bool
- Array
- Buffer
- Date
- ObjectId | Oid
- Mixed

Using this exposed access to the `Mixed` SchemaType, we can use them in our schema.

```
var Mixed = mongoose.Schema.Types.Mixed;
new mongoose.Schema({ _user: Mixed })
```

Schema.indexTypes

The allowed index types

Schema.prototype.add()

Parameters

- obj «Object|Schema» plain object with paths to add, or another schema
- [prefix] «String» path to prefix the newly added paths with

Returns:

- «Schema» the Schema instance

Adds key path / schema type pairs to this schema.

Example:

```
const ToySchema = new Schema();
ToySchema.add({ name: 'string', color: 'string', price: 'number' });

const TurboManSchema = new Schema();
// You can also `add()` another schema and copy over all paths, virtuals,
// getters, setters, indexes, methods, and statics.
TurboManSchema.add(ToySchema).add({ year: Number });
```

Schema.prototype.childSchemas

Array of child schemas (from document arrays and single nested subdocs) and their corresponding compiled models. Each element of the array is an object with 2 properties: `schema` and `model`.

This property is typically only useful for plugin authors and advanced users. You do not need to interact with this property at all to use mongoose.

Schema.prototype.clone()

Returns:

- «Schema» the cloned schema

Returns a deep copy of the schema

Schema.prototype.eachPath()

Parameters

- fn «Function» callback function

Returns:

- «Schema» this

Iterates the schemas paths similar to Array#forEach.

The callback is passed the pathname and schemaType as arguments on each iteration.

Schema.prototype.get()

Parameters

- key «String» option name

Gets a schema option.

Schema.prototype.index()

Parameters

- fields «Object»
- [options] «Object» Options to pass to MongoDB driver's `createIndex()` function
    - [options.expires=null] «String» Mongoose-specific syntactic sugar, uses ms to convert `expires` option into seconds for the `expireAfterSeconds` in the above link.

Defines an index (most likely compound) for this schema.

Example

```
schema.index({ first: 1, last: -1 })
```

Schema.prototype.indexes()

Returns a list of indexes that this schema declares, via `schema.index()` or by `index: true` in a path's options.

Schema.prototype.loadClass()

Parameters

- model «Function»
- [virtualsOnly] «Boolean» if truthy, only pulls virtuals from the class, not methods or statics

Loads an ES6 class into a schema. Maps setters + getters, static methods, and instance methods to schema virtuals, statics, and methods.

Example:

```
const md5 = require('md5');
const userSchema = new Schema({ email: String });
class UserClass {
  // `gravatarImage` becomes a virtual
  get gravatarImage() {
    const hash = md5(this.email.toLowerCase());
    return `https://www.gravatar.com/avatar/${hash}`;
  }

  // `getProfileUrl()` becomes a document method
  getProfileUrl() {
    return `https://mysite.com/${this.email}`;
  }

  // `findByEmail()` becomes a static
  static findByEmail(email) {
    return this.findOne({ email });
  }
}

// `schema` will now have a `gravatarImage` virtual, a `getProfileUrl()` method,
// and a `findByEmail()` static
userSchema.loadClass(UserClass);
```

Schema.prototype.method()

Parameters

- method «String|Object» name
- [fn] «Function»

Adds an instance method to documents constructed from Models compiled from this schema.

Example

```
var schema = kittySchema = new Schema(..);

schema.method('meow', function () {
  console.log('meeeeeoooooooooooow');
})

var Kitty = mongoose.model('Kitty', schema);

var fizz = new Kitty;
fizz.meow(); // meeeeeoooooooooooow
```

If a hash of name/fn pairs is passed as the only argument, each name/fn pair will be added as methods.

```
schema.method({
    purr: function () {}
  , scratch: function () {}
});

// later
fizz.purr();
fizz.scratch();
```

NOTE: `Schema.method()` adds instance methods to the `Schema.methods` object. You can also add instance methods directly to the `Schema.methods` object as seen in the guide

Schema.prototype.obj

The original object passed to the schema constructor

Example:

```
var schema = new Schema({ a: String }).add({ b: String });
schema.obj; // { a: String }
```

Schema.prototype.path()

Parameters

- path «String»
- constructor «Object»

Gets/sets schema paths.

Sets a path (if arity 2) Gets a path (if arity 1)

Example

```
schema.path('name') // returns a SchemaType
schema.path('name', Number) // changes the schemaType of `name` to Number
```

Schema.prototype.pathType()

Parameters

- path «String»

Returns:

- «String»

Returns the pathType of `path` for this schema.

Given a path, returns whether it is a real, virtual, nested, or ad-hoc/undefined path.

Schema.prototype.plugin()

Parameters

- plugin «Function» callback
- [opts] «Object»

Registers a plugin for this schema.

Schema.prototype.post()

Parameters

- method «String|RegExp» or regular expression to match method name
- [options] «Object»
    - [options.document] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on document middleware.

    - [options.query] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on query middleware.

- fn «Function» callback

Defines a post hook for the document

```
var schema = new Schema(..);
schema.post('save', function (doc) {
  console.log('this fired after a document was saved');
});

schema.post('find', function(docs) {
  console.log('this fired after you ran a find query');
});

schema.post(/Many$/, function(res) {
  console.log('this fired after you ran `updateMany()` or `deleteMany()`');
});

var Model = mongoose.model('Model', schema);

var m = new Model(..);
m.save(function(err) {
  console.log('this fires after the `post` hook');
});

m.find(function(err, docs) {
  console.log('this fires after the post find hook');
});
```

Schema.prototype.pre()

Parameters

- method «String|RegExp» or regular expression to match method name
- [options] «Object»
    - [options.document] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on document middleware.
    - [options.query] «Boolean» If `name` is a hook for both document and query middleware, set to `true` to run on query middleware.
- callback «Function»

Defines a pre hook for the document.

Example

```
var toySchema = new Schema({ name: String, created: Date });

toySchema.pre('save', function(next) {
  if (!this.created) this.created = new Date;
  next();
});

toySchema.pre('validate', function(next) {
  if (this.name !== 'Woody') this.name = 'Woody';
  next();
});

// Equivalent to calling `pre()` on `find`, `findOne`, `findOneAndUpdate`.
toySchema.pre(/^find/, function(next) {
  console.log(this.getQuery());
});
```

Schema.prototype.queue()

Parameters

- name «String» name of the document method to call later
- args «Array» arguments to pass to the method

Adds a method call to the queue.

Schema.prototype.remove()

Parameters

- path «String|Array»

Returns:

- «Schema» the Schema instance

Removes the given `path` (or [`paths`]).

Schema.prototype.requiredPaths()

Parameters

- invalidate «Boolean» refresh the cache

Returns:

- «Array»

Returns an Array of path strings that are required by this schema.

Schema.prototype.set()

Parameters

- key «String» option name
- [value] «Object» if not passed, the current option value is returned

Sets/gets a schema option.

Example

```
schema.set('strict'); // 'true' by default
schema.set('strict', false); // Sets 'strict' to false
schema.set('strict'); // 'false'
```

Schema.prototype.static()

Parameters

- name «String|Object»
- [fn] «Function»

Adds static "class" methods to Models compiled from this schema.

Example

```
var schema = new Schema(..);
schema.static('findByName', function (name, callback) {
  return this.find({ name: name }, callback);
});

var Drink = mongoose.model('Drink', schema);
Drink.findByName('sanpellegrino', function (err, drinks) {
  //
});
```

If a hash of name/fn pairs is passed as the only argument, each name/fn pair will be added as statics.

Schema.prototype.virtual()

Parameters

- name «String»
- [options] «Object»

Returns:

- «VirtualType»

Creates a virtual type with the given name.

Schema.prototype.virtualpath()

Parameters

- name «String»

Returns:

- «VirtualType»

Returns the virtual type with the given `name`.

Schema.reserved

Reserved document keys.

Keys in this object are names that are rejected in schema declarations b/c they conflict with mongoose functionality. Using these key name will throw an error.

```
on, emit, _events, db, get, set, init, isNew, errors, schema, options, modelName, collec
```

NOTE: Use of these terms as method names is permitted, but play at your own risk, as they may be existing mongoose document methods you are stomping on.

```
var schema = new Schema(..);
 schema.methods.init = function () {} // potentially breaking
```

Connection

Connection()

Parameters

- base «Mongoose» a mongoose instance

Connection constructor

For practical reasons, a Connection equals a Db.

Connection.prototype.close()

Parameters

- [force] «Boolean» optional
- [callback] «Function» optional

Returns:

- «Connection» self

Closes the connection

Connection.prototype.collection()

Parameters

- name «String» of the collection
- [options] «Object» optional collection options

Returns:

- «Collection» collection instance

Retrieves a collection, creating it if not cached.

Not typically needed by applications. Just talk to your collection through your model.

Connection.prototype.collections

A hash of the collections associated with this connection

Connection.prototype.config

A hash of the global options that are associated with this connection

Connection.prototype.createCollection()

Parameters

- collection «string» The collection to create
- [options] «Object» see MongoDB driver docs

- [callback] «Function»

Returns:

- «Promise»

Helper for `createCollection()`. Will explicitly create the given collection with specified options. Used to create capped collections and views from mongoose.

Options are passed down without modification to the MongoDB driver's `createCollection()` function

Connection.prototype.db

The mongodb.Db instance, set when the connection is opened

Connection.prototype.deleteModel()

Parameters

- name «String|RegExp» if string, the name of the model to remove. If regexp, removes all models whose name matches the regexp.

Returns:

- «Connection» this

Removes the model named `name` from this connection, if it exists. You can use this function to clean up any models you created in your tests to prevent OverwriteModelErrors.

Example:

```
conn.model('User', new Schema({ name: String }));
console.log(conn.model('User')); // Model object
conn.deleteModel('User');
console.log(conn.model('User')); // undefined

// Usually useful in a Mocha `afterEach()` hook
afterEach(function() {
  conn.deleteModel(/.+/); // Delete every model
});
```

Connection.prototype.dropCollection()

Parameters

- collection «string» The collection to delete
- [callback] «Function»

Returns:

- «Promise»

Helper for `dropCollection()`. Will delete the given collection, including all documents and indexes.

Connection.prototype.dropDatabase()

Parameters

- [callback] «Function»

Returns:

- «Promise»

Helper for `dropDatabase()`. Deletes the given database, including all collections, documents, and indexes.

Connection.prototype.host

The host name portion of the URI. If multiple hosts, such as a replica set, this will contain the first host name in the URI

Example

```
mongoose.createConnection('mongodb://localhost:27017/mydb').host; // "localhost"
```

Connection.prototype.model()

Parameters

- name «String|Function» the model name or class extending Model

- [schema] «Schema» a schema. necessary when defining a model
- [collection] «String» name of mongodb collection (optional) if not given it will be induced from model name

Returns:

- «Model» The compiled model

Defines or retrieves a model.

```
var mongoose = require('mongoose');
var db = mongoose.createConnection(..);
db.model('Venue', new Schema(..));
var Ticket = db.model('Ticket', new Schema(..));
var Venue = db.model('Venue');
```

When no `collection` argument is passed, Mongoose produces a collection name by passing the model `name` to the utils.toCollectionName method. This method pluralizes the name. If you don't like this behavior, either pass a collection name or set your schemas collection name option.

Example:

```
var schema = new Schema({ name: String }, { collection: 'actor' });

// or

schema.set('collection', 'actor');

// or

var collectionName = 'actor'
var M = conn.model('Actor', schema, collectionName)
```

Connection.prototype.modelNames()

Returns:

- «Array»

Returns an array of model names created on this connection.

Connection.prototype.name

The name of the database this connection points to.

Example

```
mongoose.createConnection('mongodb://localhost:27017/mydb').name; // "mydb"
```

Connection.prototype.pass

The password specified in the URI

Example

```
mongoose.createConnection('mongodb://val:psw@localhost:27017/mydb').pass; // "psw"
```

Connection.prototype.port

The port portion of the URI. If multiple hosts, such as a replica set, this will contain the port from the first host name in the URI.

Example

```
mongoose.createConnection('mongodb://localhost:27017/mydb').port; // 27017
```

Connection.prototype.readyState

Connection ready state

- 0 = disconnected
- 1 = connected
- 2 = connecting
- 3 = disconnecting

Each state change emits its associated event name.

Example

```
conn.on('connected', callback);
conn.on('disconnected', callback);
```

Connection.prototype.startSession()

Parameters

- [options] «Object» see the mongodb driver options
  - [options.causalConsistency=true] «Boolean» set to false to disable causal consistency
- [callback] «Function»

Returns:

- «Promise<ClientSession>» promise that resolves to a MongoDB driver `ClientSession`

Requires MongoDB >= 3.6.0. Starts a MongoDB session for benefits like causal consistency, retryable writes, and transactions.

Example:

```
const session = await conn.startSession();
let doc = await Person.findOne({ name: 'Ned Stark' }, null, { session });
await doc.remove();
// `doc` will always be null, even if reading from a replica set
// secondary. Without causal consistency, it is possible to
// get a doc back from the below query if the query reads from a
// secondary that is experiencing replication lag.
doc = await Person.findOne({ name: 'Ned Stark' }, null, { session, readPreference: 'secc
```

Connection.prototype.useDb()

Parameters

- name «String» The database name

Returns:

- «Connection» New Connection Object

Switches to a different database using the same connection pool.

Returns a new connection object, with the new db.

Connection.prototype.user

The username specified in the URI

Example

```
mongoose.createConnection('mongodb://val:psw@localhost:27017/mydb').user; // "val"
```

Document

- Document.prototype.populate()
- Document.prototype.populated()
- Document.prototype.replaceOne()
- Document.prototype.save()
- Document.prototype.schema
- Document.prototype.set()
- Document.prototype.toJSON()
- Document.prototype.toObject()
- Document.prototype.toString()
- Document.prototype.unmarkModified()
- Document.prototype.update()
- Document.prototype.updateOne()
- Document.prototype.validate()
- Document.prototype.validateSync()

Document.prototype.$ignore()

Parameters

- path «String» the path to ignore

Don't run validation on this path or persist changes to this path.

Example:

```
doc.foo = null;
doc.$ignore('foo');
doc.save(); // changes to foo will not be persisted and validators won't be run
```

Document.prototype.$isDefault()

Parameters

- [path] «String»

Returns:

- «Boolean»

Checks if a path is set to its default.

Example

```
MyModel = mongoose.model('test', { name: { type: String, default: 'Val '} });
var m = new MyModel();
m.$isDefault('name'); // true
```

Document.prototype.$isDeleted()

Parameters

- [val] «Boolean» optional, overrides whether mongoose thinks the doc is deleted

Returns:

- «Boolean» whether mongoose thinks this doc is deleted.

Getter/setter, determines whether the document was removed or not.

Example:

```
product.remove(function (err, product) {
  product.isDeleted(); // true
  product.remove(); // no-op, doesn't send anything to the db

  product.isDeleted(false);
  product.isDeleted(); // false
  product.remove(); // will execute a remove against the db
})
```

Document.prototype.$markValid()

Parameters

- path «String» the field to mark as valid

Marks a path as valid, removing existing validation errors.

Document.prototype.$session()

Parameters

- [session] «ClientSession» overwrite the current session

Returns:

- «ClientSession»

Getter/setter around the session associated with this document. Used to automatically set `session` if you `save()` a doc that you got from a query with an associated session.

Example:

```
const session = MyModel.startSession();
const doc = await MyModel.findOne().session(session);
doc.$session() === session; // true
doc.$session(null);
doc.$session() === null; // true
```

If this is a top-level document, setting the session propagates to all child docs.

Document.prototype.$set()

Parameters

- path «String|Object» path or object of key/vals to set
- val «Any» the value to set
- [type] «Schema|String|Number|Buffer|*» optionally specify a type for "on-the-fly" attributes
- [options] «Object» optionally specify options that modify the behavior of the set

Alias for `set()`, used internally to avoid conflicts

Document.prototype.depopulate()

Parameters

- path «String»

Returns:

- «Document» this

Takes a populated field and returns it to its unpopulated state.

Example:

```
Model.findOne().populate('author').exec(function (err, doc) {
  console.log(doc.author.name); // Dr.Seuss
  console.log(doc.depopulate('author'));
  console.log(doc.author); // '5144cf8050f071d979c118a7'
})
```

If the path was not populated, this is a no-op.

Document.prototype.equals()

Parameters

- doc «Document» a document to compare

Returns:

- «Boolean»

Returns true if the Document stores the same data as doc.

Documents are considered equal when they have matching `_id` s, unless neither document has an `_id` , in which case this function falls back to using `deepEqual()` .

Document.prototype.errors

Hash containing current validation errors.

Document.prototype.execPopulate()

Parameters

- [callback] «Function» optional callback. If specified, a promise will **not** be returned

Returns:

- «Promise» promise that resolves to the document when population is done

Explicitly executes population and returns a promise. Useful for ES2015 integration.

Example:

```
var promise = doc.
  populate('company').
  populate({
    path: 'notes',
    match: /airline/,
    select: 'text',
    model: 'modelName'
    options: opts
  }).
  execPopulate();

// summary
doc.execPopulate().then(resolve, reject);
```

Document.prototype.get()

Parameters

- path «String»
- [type] «Schema|String|Number|Buffer|*» optionally specify a type for on-the-fly attributes

Returns the value of a path.

Example

```
// path
doc.get('age') // 47

// dynamic casting to a string
doc.get('age', String) // "47"
```

Document.prototype.id

The string version of this documents _id.

Note:

This getter exists on all documents by default. The getter can be disabled by setting the `id` option of its `Schema` to false at construction time.

```
new Schema({ name: String }, { id: false });
```

Document.prototype.init()

Parameters

- doc «Object» document returned by mongo

Initializes the document without setters or marking anything modified.

Called internally after a document is returned from mongodb. Normally, you do **not** need to call this function on your own.

This function triggers `init` middleware. Note that `init` hooks are synchronous.

Document.prototype.inspect()

Helper for console.log

Document.prototype.invalidate()

Parameters

- path «String» the field to invalidate
- errorMsg «String|Error» the error which states the reason `path` was invalid
- value «Object|String|Number|any» optional invalid value
- [kind] «String» optional `kind` property for the error

Returns:

- «ValidationError» the current ValidationError, with all currently invalidated paths

Marks a path as invalid, causing validation to fail.

The `errorMsg` argument will become the message of the `ValidationError`.

The `value` argument (if passed) will be available through the `ValidationError.value` property.

```
doc.invalidate('size', 'must be less than 20', 14);

doc.validate(function (err) {
  console.log(err)
  // prints
  { message: 'Validation failed',
```

```
    name: 'ValidationError',
    errors:
     { size:
        { message: 'must be less than 20',
          name: 'ValidatorError',
          path: 'size',
          type: 'user defined',
          value: 14 } } }
})
```

Document.prototype.isDirectModified()

Parameters

- path «String»

Returns:

- «Boolean»

Returns true if `path` was directly set and modified, else false.

Example

```
doc.set('documents.0.title', 'changed');
doc.isDirectModified('documents.0.title') // true
doc.isDirectModified('documents') // false
```

Document.prototype.isDirectSelected()

Parameters

- path «String»

Returns:

- «Boolean»

Checks if `path` was explicitly selected. If no projection, always returns true.

Example

```
Thing.findOne().select('nested.name').exec(function (err, doc) {
   doc.isDirectSelected('nested.name') // true
   doc.isDirectSelected('nested.otherName') // false
```

```
    doc.isDirectSelected('nested')  // false
})
```

Document.prototype.isInit()

Parameters

- path «String»

Returns:

- «Boolean»

Checks if `path` was initialized.

Document.prototype.isModified()

Parameters

- [path] «String» optional

Returns:

- «Boolean»

Returns true if this document was modified, else false.

If `path` is given, checks if a path or any full path containing `path` as part of its path chain has been modified.

Example

```
doc.set('documents.0.title', 'changed');
doc.isModified()                      // true
doc.isModified('documents')           // true
doc.isModified('documents.0.title')   // true
doc.isModified('documents otherProp') // true
doc.isDirectModified('documents')     // false
```

Document.prototype.isNew

Boolean flag specifying if the document is new.

Document.prototype.isSelected()

Parameters

- path «String»

Returns:

- «Boolean»

Checks if `path` was selected in the source query which initialized this document.

Example

```
Thing.findOne().select('name').exec(function (err, doc) {
   doc.isSelected('name') // true
   doc.isSelected('age')  // false
})
```

Document.prototype.markModified()

Parameters

- path «String» the path to mark modified
- [scope] «Document» the scope to run validators with

Marks the path as having pending changes to write to the db.

Very helpful when using Mixed types.

Example:

```
doc.mixed.type = 'changed';
doc.markModified('mixed.type');
doc.save() // changes to mixed.type are now persisted
```

Document.prototype.modifiedPaths()

Parameters

- [options] «Object»

- [options.includeChildren=false] «Boolean» if true, returns children of modified paths as well. For example, if false, the list of modified paths for `doc.colors = { primary: 'blue' };` will **not** contain `colors.primary`. If true, `modifiedPaths()` will return an array that contains `colors.primary`.

Returns:

- «Array»

Returns the list of paths that have been modified.

Document.prototype.populate()

Parameters

- [path] «String|Object» The path to populate or an options object
- [callback] «Function» When passed, population is invoked

Returns:

- «Document» this

Populates document references, executing the `callback` when complete. If you want to use promises instead, use this function with `execPopulate()`

Example:

```
doc
.populate('company')
.populate({
  path: 'notes',
  match: /airline/,
  select: 'text',
  model: 'modelName'
  options: opts
}, function (err, user) {
  assert(doc._id === user._id) // the document itself is passed
})

// summary
doc.populate(path)                  // not executed
doc.populate(options);              // not executed
doc.populate(path, callback)        // executed

doc.populate(options, callback);    // executed
doc.populate(callback);             // executed
doc.populate(options).execPopulate() // executed, returns promise
```

NOTE:

Population does not occur unless a `callback` is passed *or* you explicitly call `execPopulate()`. Passing the same path a second time will overwrite the previous path options. See Model.populate() for explaination of options.

Document.prototype.populated()

Parameters

- path «String»

Returns:

- «Array,ObjectId,Number,Buffer,String,undefined»

Gets _id(s) used during population of the given `path`.

Example:

```
Model.findOne().populate('author').exec(function (err, doc) {
  console.log(doc.author.name)         // Dr.Seuss
  console.log(doc.populated('author')) // '5144cf8050f071d979c118a7'
})
```

If the path was not populated, undefined is returned.

Document.prototype.replaceOne()

Parameters

- doc «Object»
- options «Object»
- callback «Function»

Returns:

- «Query»

Sends a replaceOne command with this document `_id` as the query selector.

Valid options:

- same as in Model.replaceOne

Document.prototype.save()

Parameters

- [options] «Object» options optional options
  - [options.safe] «Object» (DEPRECATED) overrides schema's safe option
  - [options.validateBeforeSave] «Boolean» set to false to save without validating.
- [fn] «Function» optional callback

Returns:

- «Promise,undefined» Returns undefined if used with callback or a Promise otherwise.

Saves this document.

Example:

```
product.sold = Date.now();
product.save(function (err, product) {
  if (err) ..
})
```

The callback will receive two parameters

1. `err` if an error occurred
2. `product` which is the saved `product`

As an extra measure of flow control, save will return a Promise.

Example:

```
product.save().then(function(product) {
   ...
});
```

Document.prototype.schema

The documents schema.

Document.prototype.set()

Parameters

- path «String|Object» path or object of key/vals to set
- val «Any» the value to set
- [type] «Schema|String|Number|Buffer|*» optionally specify a type for "on-the-fly" attributes
- [options] «Object» optionally specify options that modify the behavior of the set

Sets the value of a path, or many paths.

Example:

```
// path, value
doc.set(path, value)

// object
doc.set({
    path  : value
  , path2 : {
       path  : value
    }
})

// on-the-fly cast to number
doc.set(path, value, Number)

// on-the-fly cast to string
doc.set(path, value, String)

// changing strict mode behavior
doc.set(path, value, { strict: false });
```

Document.prototype.toJSON()

Parameters

- options «Object»

Returns:

- «Object»

The return value of this method is used in calls to JSON.stringify(doc).

This method accepts the same options as Document#toObject. To apply the options to every document of your schema by default, set your schemas `toJSON` option to the same argument.

```
schema.set('toJSON', { virtuals: true })
```

See schema options for details.

Document.prototype.toObject()

Parameters

- [options] «Object»
  - [options.getters=false] «Boolean» if true, apply all getters, including virtuals
  - [options.virtuals=false] «Boolean» if true, apply virtuals. Use `{ getters: true, virtuals: false }` to just apply getters, not virtuals
  - [options.minimize=true] «Boolean» if true, omit any empty objects from the output
  - [options.transform=null] «Function|null» if set, mongoose will call this function to allow you to transform the returned object
  - [options.depopulate=false] «Boolean» if true, replace any conventionally populated paths with the original id in the output. Has no affect on virtual populated paths.
  - [options.versionKey=true] «Boolean» if false, exclude the version key (`__v` by default) from the output

Returns:

- «Object» js object

Converts this document into a plain javascript object, ready for storage in MongoDB.

Buffers are converted to instances of mongodb.Binary for proper storage.

Options:

- `getters` apply all getters (path and virtual getters), defaults to false
- `virtuals` apply virtual getters (can override `getters` option), defaults to false
- `minimize` remove empty objects (defaults to true)
- `transform` a transform function to apply to the resulting document before returning
- `depopulate` depopulate any populated paths, replacing them with their original refs (defaults to false)
- `versionKey` whether to include the version key (defaults to true)

Getters/Virtuals

Example of only applying path getters

```
doc.toObject({ getters: true, virtuals: false })
```

Example of only applying virtual getters

```
doc.toObject({ virtuals: true })
```

Example of applying both path and virtual getters

```
doc.toObject({ getters: true })
```

To apply these options to every document of your schema by default, set your schemas `toObject` option to the same argument.

```
schema.set('toObject', { virtuals: true })
```

Transform

We may need to perform a transformation of the resulting object based on some criteria, say to remove some sensitive information or return a custom object. In this case we set the optional `transform` function.

Transform functions receive three arguments

```
function (doc, ret, options) {}
```

- `doc` The mongoose document which is being converted
- `ret` The plain object representation which has been converted
- `options` The options in use (either schema options or the options passed inline)

Example

```
// specify the transform schema option
if (!schema.options.toObject) schema.options.toObject = {};
schema.options.toObject.transform = function (doc, ret, options) {
  // remove the _id of every document before returning the result
  delete ret._id;
  return ret;
}

// without the transformation in the schema
doc.toObject(); // { _id: 'anId', name: 'Wreck-it Ralph' }

// with the transformation
doc.toObject(); // { name: 'Wreck-it Ralph' }
```

With transformations we can do a lot more than remove properties. We can even return completely new customized objects:

```
if (!schema.options.toObject) schema.options.toObject = {};
schema.options.toObject.transform = function (doc, ret, options) {
  return { movie: ret.name }
}

// without the transformation in the schema
doc.toObject(); // { _id: 'anId', name: 'Wreck-it Ralph' }

// with the transformation
doc.toObject(); // { movie: 'Wreck-it Ralph' }
```

Note: if a transform function returns `undefined`, the return value will be ignored.

Transformations may also be applied inline, overridding any transform set in the options:

```
function xform (doc, ret, options) {
  return { inline: ret.name, custom: true }
}

// pass the transform as an inline option
doc.toObject({ transform: xform }); // { inline: 'Wreck-it Ralph', custom: true }
```

If you want to skip transformations, use `transform: false`:

```
if (!schema.options.toObject) schema.options.toObject = {};
schema.options.toObject.hide = '_id';
schema.options.toObject.transform = function (doc, ret, options) {
  if (options.hide) {
    options.hide.split(' ').forEach(function (prop) {
      delete ret[prop];
    });
  }
  return ret;
}

var doc = new Doc({ _id: 'anId', secret: 47, name: 'Wreck-it Ralph' });
doc.toObject();                                        // { secret: 47, name: 'Wreck-it
doc.toObject({ hide: 'secret _id', transform: false });// { _id: 'anId', secret: 47, nam
doc.toObject({ hide: 'secret _id', transform: true }); // { name: 'Wreck-it Ralph' }
```

Transforms are applied *only to the document and are not applied to sub-documents*.

Transforms, like all of these options, are also available for `toJSON`.

See schema options for some more details.

During save, no custom options are applied to the document before being sent to the database.

Document.prototype.toString()

Helper for console.log

Document.prototype.unmarkModified()

Parameters

- path «String» the path to unmark modified

Clears the modified state on the specified path.

Example:

```
doc.foo = 'bar';
doc.unmarkModified('foo');
doc.save(); // changes to foo will not be persisted
```

Document.prototype.update()

Parameters

- doc «Object»
- options «Object»
- callback «Function»

Returns:

- «Query»

Sends an update command with this document `_id` as the query selector.

Example:

```
weirdCar.update({$inc: {wheels:1}}, { w: 1 }, callback);
```

Valid options:

- same as in Model.update

Document.prototype.updateOne()

Parameters

- doc «Object»
- options «Object»
- callback «Function»

Returns:

- «Query»

Sends an updateOne command with this document `_id` as the query selector.

Example:

```
weirdCar.updateOne({$inc: {wheels:1}}, { w: 1 }, callback);
```

Valid options:

- same as in Model.updateOne

Document.prototype.validate()

Parameters

- optional «Object» options internal options
- callback «Function» optional callback called after validation completes, passing an error if one occurred

Returns:

- «Promise» Promise

Executes registered validation rules for this document.

Note:

This method is called `pre` save and if a validation rule is violated, save is aborted and the error is returned to your `callback`.

Example:

```
doc.validate(function (err) {
  if (err) handleError(err);
  else // validation passed
});
```

Document.prototype.validateSync()

Parameters

- pathsToValidate «Array|string» only validate the given paths

Returns:

- «MongooseError,undefined» MongooseError if there are errors during validation, or undefined if there is no error.

Executes registered validation rules (skipping asynchronous validators) for this document.

Note:

This method is useful if you need synchronous validation.

Example:

```
var err = doc.validateSync();
if ( err ){
  handleError( err );
} else {
  // validation passed
}
```

Model

- Model()
- Model.aggregate()
- Model.bulkWrite()
- Model.count()
- Model.countDocuments()

- Model.create()
- Model.createCollection()
- Model.createIndexes()
- Model.deleteMany()
- Model.deleteOne()
- Model.discriminator()
- Model.distinct()
- Model.ensureIndexes()
- Model.estimatedDocumentCount()
- Model.find()
- Model.findById()
- Model.findByIdAndDelete()
- Model.findByIdAndRemove()
- Model.findByIdAndUpdate()
- Model.findOne()
- Model.findOneAndDelete()
- Model.findOneAndRemove()
- Model.findOneAndUpdate()
- Model.geoSearch()
- Model.hydrate()
- Model.init()
- Model.insertMany()
- Model.listIndexes()
- Model.mapReduce()
- Model.populate()
- Model.prototype.$where
- Model.prototype.$where()
- Model.prototype.base
- Model.prototype.baseModelName
- Model.prototype.collection
- Model.prototype.db
- Model.prototype.discriminators
- Model.prototype.increment()
- Model.prototype.model()
- Model.prototype.modelName

- Model.prototype.remove()
- Model.prototype.save()
- Model.prototype.schema
- Model.prototypedelete
- Model.remove()
- Model.replaceOne()
- Model.startSession()
- Model.syncIndexes()
- Model.translateAliases()
- Model.update()
- Model.updateMany()
- Model.updateOne()
- Model.watch()
- Model.where()

Model()

Parameters

- doc «Object» values for initial set
- optional «[fields]» object containing the fields that were selected in the query which returned this document. You do **not** need to set this parameter to ensure Mongoose handles your query projetion.

A Model is a class that's your primary tool for interacting with MongoDB. An instance of a Model is called a Document.

In Mongoose, the term "Model" refers to subclasses of the `mongoose.Model` class. You should not use the `mongoose.Model` class directly. The `mongoose.model()` and `connection.model()` functions create subclasses of `mongoose.Model` as shown below.

Example:

```
// `UserModel` is a "Model", a subclass of `mongoose.Model`.
const UserModel = mongoose.model('User', new Schema({ name: String }));

// You can use a Model to create new documents using `new`:
const userDoc = new UserModel({ name: 'Foo' });
await userDoc.save();

// You also use a model to create queries:
const userFromDb = await UserModel.findOne({ name: 'Foo' });
```

Model.aggregate()

Parameters

- [pipeline] «Array» aggregation pipeline as an array of objects
- [callback] «Function»

Returns:

- «Aggregate»

Performs aggregations on the models collection.

If a `callback` is passed, the `aggregate` is executed and a `Promise` is returned. If a callback is not passed, the `aggregate` itself is returned.

This function triggers the following middleware.

- `aggregate()`

Example:

```
// Find the max balance of all accounts
Users.aggregate([
  { $group: { _id: null, maxBalance: { $max: '$balance' }}},
  { $project: { _id: 0, maxBalance: 1 }}
]).
then(function (res) {
  console.log(res); // [ { maxBalance: 98000 } ]
});

// Or use the aggregation pipeline builder.
Users.aggregate().
  group({ _id: null, maxBalance: { $max: '$balance' } }).
  project('-id maxBalance').
  exec(function (err, res) {
    if (err) return handleError(err);
    console.log(res); // [ { maxBalance: 98 } ]
  });
```

NOTE:

- Arguments are not cast to the model's schema because `$project` operators allow redefining the "shape" of the documents at any stage of the pipeline, which may leave documents in an incompatible format.

- The documents returned are plain javascript objects, not mongoose documents (since any shape of document can be returned).
- Requires MongoDB >= 2.1

Model.bulkWrite()

Parameters

- ops «Array»
- [options] «Object»
- [callback] «Function» callback `function(error, bulkWriteOpResult) {}`

Returns:

- «Promise» resolves to a `BulkWriteOpResult` if the operation succeeds

Sends multiple `insertOne`, `updateOne`, `updateMany`, `replaceOne`, `deleteOne`, and/or `deleteMany` operations to the MongoDB server in one command. This is faster than sending multiple independent operations (like) if you use `create()`) because with `bulkWrite()` there is only one round trip to MongoDB.

Mongoose will perform casting on all operations you provide.

This function does **not** trigger any middleware, not `save()` nor `update()`. If you need to trigger `save()` middleware for every document use `create()` instead.

Example:

```
Character.bulkWrite([
  {
    insertOne: {
      document: {
        name: 'Eddard Stark',
        title: 'Warden of the North'

      }
    }
  },
  {
    updateOne: {
      filter: { name: 'Eddard Stark' },
      // If you were using the MongoDB driver directly, you'd need to do
      // `update: { $set: { title: ... } }` but mongoose adds $set for
      // you.
      update: { title: 'Hand of the King' }
    }
  },
  {
```

```
    deleteOne: {
      {
        filter: { name: 'Eddard Stark' }
      }
    }
  }
]).then(res => {
 // Prints "1 1 1"
 console.log(res.insertedCount, res.modifiedCount, res.deletedCount);
});
```

Model.count()

Parameters

- filter «Object»
- [callback] «Function»

Returns:

- «Query»

Counts number of documents that match `filter` in a database collection.

This method is deprecated. If you want to count the number of documents in a collection, e.g. `count({})`, use the `estimatedDocumentCount()` function instead. Otherwise, use the `countDocuments()` function instead.

Example:

```
Adventure.count({ type: 'jungle' }, function (err, count) {
  if (err) ..
  console.log('there are %d jungle adventures', count);
});
```

Model.countDocuments()

Parameters

- filter «Object»
- [callback] «Function»

Returns:

- «Query»

Counts number of documents matching `filter` in a database collection.

Example:

```
Adventure.countDocuments({ type: 'jungle' }, function (err, count) {
  console.log('there are %d jungle adventures', count);
});
```

If you want to count all documents in a large collection, use the `estimatedDocumentCount()` function instead. If you call `countDocuments({})`, MongoDB will always execute a full collection scan and **not** use any indexes.

The `countDocuments()` function is similar to `count()`, but there are a few operators that `countDocuments()` does not support. Below are the operators that `count()` supports but `countDocuments()` does not, and the suggested replacement:

- `$where`: `$expr`
- `$near`: `$geoWithin` with `$center`
- `$nearSphere`: `$geoWithin` with `$centerSphere`

Model.create()

Parameters

- docs «Array|Object» Documents to insert, as a spread or array
- [options] «Object» Options passed down to `save()`. To specify `options`, `docs` **must** be an array, not a spread.
- [callback] «Function» callback

Returns:

- «Promise»

Shortcut for saving one or more documents to the database. `MyModel.create(docs)` does `new MyModel(doc).save()` for every doc in docs.

This function triggers the following middleware.

- `save()`

Example:

```
// pass a spread of docs and a callback
Candy.create({ type: 'jelly bean' }, { type: 'snickers' }, function (err, jellybean, sni
```

```
  if (err) // ...
});

// pass an array of docs
var array = [{ type: 'jelly bean' }, { type: 'snickers' }];
Candy.create(array, function (err, candies) {
  if (err) // ...

  var jellybean = candies[0];
  var snickers = candies[1];
  // ...
});

// callback is optional; use the returned promise if you like:
var promise = Candy.create({ type: 'jawbreaker' });
promise.then(function (jawbreaker) {
  // ...
})
```

Model.createCollection()

Parameters

- [options] «Object» see MongoDB driver docs
- [callback] «Function»

Create the collection for this model. By default, if no indexes are specified, mongoose will not create the collection for the model until any documents are created. Use this method to create the collection explicitly.

Note 1: You may need to call this before starting a transaction See https://docs.mongodb.com/manual/core/transactions/#transactions-and-operations

Note 2: You don't have to call this if your schema contains index or unique field. In that case, just use `Model.init()`

Example:

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

User.createCollection().then(function(collection) {
  console.log('Collection is created!');
});
```

Model.createIndexes()

Parameters

- [options] «Object» internal options
- [cb] «Function» optional callback

Returns:

- «Promise»

Similar to `ensureIndexes()`, except for it uses the `createIndex` function.

Model.deleteMany()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Deletes all of the documents that match `conditions` from the collection. Behaves like `remove()`, but deletes all documents that match `conditions` regardless of the `single` option.

Example:

```
Character.deleteMany({ name: /Stark/, age: { $gte: 18 } }, function (err) {});
```

Note:

Like `Model.remove()`, this function does **not** trigger `pre('remove')` or `post('remove')` hooks.

Model.deleteOne()

Parameters

- conditions «Object»

- [callback] «Function»

Returns:

- «Query»

Deletes the first document that matches `conditions` from the collection. Behaves like `remove()`, but deletes at most one document regardless of the `single` option.

Example:

```
Character.deleteOne({ name: 'Eddard Stark' }, function (err) {});
```

Note:

Like `Model.remove()`, this function does **not** trigger `pre('remove')` or `post('remove')` hooks.

Model.discriminator()

Parameters

- name «String» discriminator model name
- schema «Schema» discriminator model schema
- value «String» the string stored in the `discriminatorKey` property

Adds a discriminator type.

Example:

```
function BaseSchema() {
  Schema.apply(this, arguments);

  this.add({
    name: String,
    createdAt: Date
  });
}
util.inherits(BaseSchema, Schema);

var PersonSchema = new BaseSchema();
var BossSchema = new BaseSchema({ department: String });


var Person = mongoose.model('Person', PersonSchema);
var Boss = Person.discriminator('Boss', BossSchema);
new Boss().__t; // "Boss". `__t` is the default `discriminatorKey`
```

```
var employeeSchema = new Schema({ boss: ObjectId });
var Employee = Person.discriminator('Employee', employeeSchema, 'staff');
new Employee().__t; // "staff" because of 3rd argument above
```

Model.distinct()

Parameters

- field «String»
- [conditions] «Object» optional
- [callback] «Function»

Returns:

- «Query»

Creates a Query for a `distinct` operation.

Passing a `callback` immediately executes the query.

Example

```
Link.distinct('url', { clicks: {$gt: 100}}, function (err, result) {
  if (err) return handleError(err);

  assert(Array.isArray(result));
  console.log('unique urls with more than 100 clicks', result);
})

var query = Link.distinct('url');
query.exec(callback);
```

Model.ensureIndexes()

Parameters

- [options] «Object» internal options
- [cb] «Function» optional callback

Returns:

- «Promise»

Sends `createIndex` commands to mongo for each index declared in the schema. The `createIndex` commands are sent in series.

Example:

```
Event.ensureIndexes(function (err) {
  if (err) return handleError(err);
});
```

After completion, an `index` event is emitted on this `Model` passing an error if one occurred.

Example:

```
var eventSchema = new Schema({ thing: { type: 'string', unique: true }})
var Event = mongoose.model('Event', eventSchema);

Event.on('index', function (err) {
  if (err) console.error(err); // error occurred during index creation
})
```

NOTE: It is not recommended that you run this in production. Index creation may impact database performance depending on your load. Use with caution.

Model.estimatedDocumentCount()

Parameters

- [options] «Object»
- [callback] «Function»

Returns:

- «Query»

Estimates the number of documents in the MongoDB collection. Faster than using `countDocuments()` for large collections because `estimatedDocumentCount()` uses collection metadata rather than scanning the entire collection.

Example:

```
const numAdventures = Adventure.estimatedDocumentCount();
```

Model.find()

Parameters

- conditions «Object»
- [projection] «Object|String» optional fields to return, see `Query.prototype.select()`
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Finds documents

The `conditions` are cast to their respective SchemaTypes before the command is sent.

Examples:

```
// named john and at least 18
MyModel.find({ name: 'john', age: { $gte: 18 }});

// executes immediately, passing results to callback
MyModel.find({ name: 'john', age: { $gte: 18 }}, function (err, docs) {});

// name LIKE john and only selecting the "name" and "friends" fields, executing immediat
MyModel.find({ name: /john/i }, 'name friends', function (err, docs) { })

// passing options
MyModel.find({ name: /john/i }, null, { skip: 10 })

// passing options and executing immediately
MyModel.find({ name: /john/i }, null, { skip: 10 }, function (err, docs) {});

// executing a query explicitly
var query = MyModel.find({ name: /john/i }, null, { skip: 10 })
query.exec(function (err, docs) {});

// using the promise returned from executing a query
var query = MyModel.find({ name: /john/i }, null, { skip: 10 });
var promise = query.exec();
promise.addBack(function (err, docs) {});
```

Model.findById()

Parameters

- id «Object|String|Number» value of `_id` to query by
- [projection] «Object|String» optional fields to return, see `Query.prototype.select()`
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Finds a single document by its _id field. `findById(id)` is almost* equivalent to `findOne({ _id: id })`. If you want to query by a document's `_id`, use `findById()` instead of `findOne()`.

The `id` is cast based on the Schema before sending the command.

This function triggers the following middleware.

- `findOne()`

* Except for how it treats `undefined`. If you use `findOne()`, you'll see that `findOne(undefined)` and `findOne({ _id: undefined })` are equivalent to `findOne({})` and return arbitrary documents. However, mongoose translates `findById(undefined)` into `findOne({ _id: null })`.

Example:

```
// find adventure by id and execute immediately
Adventure.findById(id, function (err, adventure) {});

// same as above
Adventure.findById(id).exec(callback);

// select only the adventures name and length
Adventure.findById(id, 'name length', function (err, adventure) {});

// same as above
Adventure.findById(id, 'name length').exec(callback);

// include all properties except for `length`
Adventure.findById(id, '-length').exec(function (err, adventure) {});

// passing options (in this case return the raw js objects, not mongoose documents by pa
Adventure.findById(id, 'name', { lean: true }, function (err, doc) {});

// same as above
Adventure.findById(id, 'name').lean().exec(function (err, doc) {});
```

Model.findByIdAndDelete()

Parameters

- id «Object|Number|String» value of `_id` to query by
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Issue a MongoDB `findOneAndDelete()` command by a document's _id field. In other words, `findByIdAndDelete(id)` is a shorthand for `findOneAndDelete({ _id: id })`.

This function triggers the following middleware.

- `findOneAndDelete()`

Model.findByIdAndRemove()

Parameters

- id «Object|Number|String» value of `_id` to query by
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Issue a mongodb findAndModify remove command by a document's _id field. `findByIdAndRemove(id, ...)` is equivalent to `findOneAndRemove({ _id: id }, ...)`.

Finds a matching document, removes it, passing the found document (if any) to the callback.

Executes immediately if `callback` is passed, else a `Query` object is returned.

This function triggers the following middleware.

- `findOneAndRemove()`

Options:

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `select`: sets the document fields to return
- `rawResult`: if true, returns the raw result from the MongoDB driver

- `strict` : overwrites the schema's strict mode option for this update

Examples:

```
A.findByIdAndRemove(id, options, callback) // executes
A.findByIdAndRemove(id, options)  // return Query
A.findByIdAndRemove(id, callback) // executes
A.findByIdAndRemove(id) // returns Query
A.findByIdAndRemove()           // returns Query
```

Model.findByIdAndUpdate()

Parameters

- id «Object|Number|String» value of `_id` to query by
- [update] «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()`.
- [callback] «Function»

Returns:

- «Query»

Issues a mongodb findAndModify update command by a document's _id field. `findByIdAndUpdate(id, ...)` is equivalent to `findOneAndUpdate({ _id: id }, ...)`.

Finds a matching document, updates it according to the `update` arg, passing any `options`, and returns the found document (if any) to the callback. The query executes immediately if `callback` is passed else a Query object is returned.

This function triggers the following middleware.

- `findOneAndUpdate()`

Options:

- `new` : bool - true to return the modified document rather than the original. defaults to false
- `upsert` : bool - creates the object if it doesn't exist. defaults to false.
- `runValidators` : if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` : if this and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created. This option only works on

MongoDB >= 2.4 because it relies on MongoDB's `$setOnInsert` operator.

- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `select` : sets the document fields to return
- `rawResult` : if true, returns the raw result from the MongoDB driver
- `strict` : overwrites the schema's strict mode option for this update

Examples:

```
A.findByIdAndUpdate(id, update, options, callback) // executes
A.findByIdAndUpdate(id, update, options)  // returns Query
A.findByIdAndUpdate(id, update, callback) // executes
A.findByIdAndUpdate(id, update)           // returns Query
A.findByIdAndUpdate()                     // returns Query
```

Note:

All top level update keys which are not `atomic` operation names are treated as set operations:

Example:

```
Model.findByIdAndUpdate(id, { name: 'jason bourne' }, options, callback)

// is sent as
Model.findByIdAndUpdate(id, { $set: { name: 'jason bourne' }}, options, callback)
```

This helps prevent accidentally overwriting your document with `{ name: 'jason bourne' }`.

Note:

Values are cast to their appropriate types when using the findAndModify helpers. However, the below are not executed by default.

- defaults. Use the `setDefaultsOnInsert` option to override.

`findAndModify` helpers support limited validation. You can enable these by setting the `runValidators` options, respectively.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.findOne()

Parameters

- [conditions] «Object»
- [projection] «Object|String» optional fields to return, see `Query.prototype.select()`
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Finds one document.

The `conditions` are cast to their respective SchemaTypes before the command is sent.

Note: `conditions` is optional, and if `conditions` is null or undefined, mongoose will send an empty `findOne` command to MongoDB, which will return an arbitrary document. If you're querying by `_id`, use `findById()` instead.

Example:

```
// find one iphone adventures - iphone adventures??
Adventure.findOne({ type: 'iphone' }, function (err, adventure) {});

// same as above
Adventure.findOne({ type: 'iphone' }).exec(function (err, adventure) {});

// select only the adventures name
Adventure.findOne({ type: 'iphone' }, 'name', function (err, adventure) {});

// same as above
Adventure.findOne({ type: 'iphone' }, 'name').exec(function (err, adventure) {});

// specify options, in this case lean
Adventure.findOne({ type: 'iphone' }, 'name', { lean: true }, callback);

// same as above
Adventure.findOne({ type: 'iphone' }, 'name', { lean: true }).exec(callback);

// chaining findOne queries (same as above)
Adventure.findOne({ type: 'iphone' }).select('name').lean().exec(callback);
```

Model.findOneAndDelete()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Issue a MongoDB `findOneAndDelete()` command.

Finds a matching document, removes it, and passes the found document (if any) to the callback.

Executes immediately if `callback` is passed else a Query object is returned.

This function triggers the following middleware.

- `findOneAndDelete()`

This function differs slightly from `Model.findOneAndRemove()` in that `findOneAndRemove()` becomes a MongoDB `findAndModify()` command, as opposed to a `findOneAndDelete()` command. For most mongoose use cases, this distinction is purely pedantic. You should use `findOneAndDelete()` unless you have a good reason not to.

Options:

- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS` : puts a time limit on the query - requires mongodb >= 2.6.0
- `select` : sets the document fields to return
- `projection` : like select, it determines which fields to return, ex. `{ projection: { _id: 0 } }`
- `rawResult` : if true, returns the raw result from the MongoDB driver
- `strict` : overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndDelete(conditions, options, callback) // executes
A.findOneAndDelete(conditions, options)  // return Query
A.findOneAndDelete(conditions, callback) // executes
A.findOneAndDelete(conditions) // returns Query
A.findOneAndDelete()           // returns Query
```

Values are cast to their appropriate types when using the findAndModify helpers. However, the below are not executed by default.

- defaults. Use the `setDefaultsOnInsert` option to override.

`findAndModify` helpers support limited validation. You can enable these by setting the `runValidators` options, respectively.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.findOneAndRemove()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Issue a mongodb findAndModify remove command.

Finds a matching document, removes it, passing the found document (if any) to the callback.

Executes immediately if `callback` is passed else a Query object is returned.

This function triggers the following middleware.

- `findOneAndRemove()`

Options:

- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS` : puts a time limit on the query - requires mongodb >= 2.6.0
- `select` : sets the document fields to return
- `projection` : like select, it determines which fields to return, ex. `{ projection: { _id: 0 } }`
- `rawResult` : if true, returns the raw result from the MongoDB driver
- `strict` : overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndRemove(conditions, options, callback) // executes
A.findOneAndRemove(conditions, options)  // return Query
A.findOneAndRemove(conditions, callback) // executes
A.findOneAndRemove(conditions) // returns Query
A.findOneAndRemove()           // returns Query
```

Values are cast to their appropriate types when using the findAndModify helpers. However, the below are not executed by default.

- defaults. Use the `setDefaultsOnInsert` option to override.

`findAndModify` helpers support limited validation. You can enable these by setting the `runValidators` options, respectively.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.findOneAndUpdate()

Parameters

- [conditions] «Object»
- [update] «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()`.
- [callback] «Function»

Returns:

- «Query»

Issues a mongodb findAndModify update command.

Finds a matching document, updates it according to the `update` arg, passing any `options`, and returns the found document (if any) to the callback. The query executes immediately if `callback` is passed else a Query object is returned.

Options:

- `new` : bool - if true, return the modified document rather than the original. defaults to false (changed in 4.0)
- `upsert` : bool - creates the object if it doesn't exist. defaults to false.
- `fields` : {Object|String} - Field selection. Equivalent to `.select(fields).findOneAndUpdate()`
- `maxTimeMS` : puts a time limit on the query - requires mongodb >= 2.6.0
- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `runValidators` : if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` : if this and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created. This option only works on MongoDB >= 2.4 because it relies on MongoDB's `$setOnInsert` operator.
- `rawResult` : if true, returns the raw result from the MongoDB driver
- `strict` : overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndUpdate(conditions, update, options, callback) // executes
A.findOneAndUpdate(conditions, update, options)  // returns Query
A.findOneAndUpdate(conditions, update, callback) // executes
A.findOneAndUpdate(conditions, update)           // returns Query
A.findOneAndUpdate()                             // returns Query
```

Note:

All top level update keys which are not `atomic` operation names are treated as set operations:

Example:

```
var query = { name: 'borne' };
Model.findOneAndUpdate(query, { name: 'jason bourne' }, options, callback)

// is sent as
Model.findOneAndUpdate(query, { $set: { name: 'jason bourne' }}, options, callback)
```

This helps prevent accidentally overwriting your document with `{ name: 'jason bourne' }`.

Note:

Values are cast to their appropriate types when using the findAndModify helpers. However, the below are not executed by default.

- defaults. Use the `setDefaultsOnInsert` option to override.

`findAndModify` helpers support limited validation. You can enable these by setting the `runValidators` options, respectively.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.geoSearch()

Parameters

- conditions «Object» an object that specifies the match condition (required)
- options «Object» for the geoSearch, some (near, maxDistance) are required
  - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()`.
- [callback] «Function» optional callback

Returns:

- «Promise»

Implements `$geoSearch` functionality for Mongoose

This function does not trigger any middleware

Example:

```
var options = { near: [10, 10], maxDistance: 5 };
Locations.geoSearch({ type : "house" }, options, function(err, res) {
  console.log(res);
});
```

Options:

- `near` {Array} x,y point to search for
- `maxDistance` {Number} the maximum distance from the point near that a result can be
- `limit` {Number} The maximum number of results to return
- `lean` {Boolean} return the raw object instead of the Mongoose Model

Model.hydrate()

Parameters

- obj «Object»

Returns:

- «Model» document instance

Shortcut for creating a new Document from existing raw data, pre-saved in the DB. The document returned has no paths marked as modified initially.

Example:

```
// hydrate previous data into a Mongoose document
var mongooseCandy = Candy.hydrate({ _id: '54108337212ffb6d459f854c', type: 'jelly bean'
```

Model.init()

Parameters

- [callback] «Function»

This function is responsible for building indexes, unless `autoIndex` is turned off.

Mongoose calls this function automatically when a model is created using `mongoose.model()` or * `connection.model()`, so you don't need to call it. This function is also idempotent, so you may call it to get back a promise that will resolve when your indexes are finished building as an alternative to `MyModel.on('index')`

Example:

```
var eventSchema = new Schema({ thing: { type: 'string', unique: true }})
// This calls `Event.init()` implicitly, so you don't need to call

// `Event.init()` on your own.
var Event = mongoose.model('Event', eventSchema);

Event.init().then(function(Event) {
  // You can also use `Event.on('index')` if you prefer event emitters
  // over promises.
  console.log('Indexes are done building!');
});
```

Model.insertMany()

Parameters

- doc(s) «Array|Object|*»
- [options] «Object» see the mongodb driver options
- [options.ordered «Boolean» = true] if true, will fail fast on the first error encountered. If false, will insert all the documents it can and report errors later. An `insertMany()` with `ordered = false` is called an "unordered" `insertMany()`.
- [options.rawResult «Boolean» = false] if false, the returned promise resolves to the documents that passed mongoose document validation. If `true`, will return the raw result from the MongoDB driver with a `mongoose` property that contains `validationErrors` if this is an unordered `insertMany`.
- [callback] «Function» callback

Returns:

- «Promise»

Shortcut for validating an array of documents and inserting them into MongoDB if they're all valid. This function is faster than `.create()` because it only sends one operation to the server, rather than one for each document.

Mongoose always validates each document **before** sending `insertMany` to MongoDB. So if one document has a validation error, no documents will be saved, unless you set the `ordered` option to false.

This function does **not** trigger save middleware.

This function triggers the following middleware.

- `insertMany()`

Example:

```
var arr = [{ name: 'Star Wars' }, { name: 'The Empire Strikes Back' }];
Movies.insertMany(arr, function(error, docs) {});
```

Model.listIndexes()

Parameters

- [cb] «Function» optional callback

Returns:

- «Promise,undefined» Returns `undefined` if callback is specified, returns a promise if no callback.

Lists the indexes currently defined in MongoDB. This may or may not be the same as the indexes defined in your schema depending on whether you use the `autoIndex` option and if you build indexes manually.

Model.mapReduce()

Parameters

- o «Object» an object specifying map-reduce options
- [callback] «Function» optional callback

Returns:

- «Promise»

Executes a mapReduce command.

`o` is an object specifying all mapReduce options as well as the map and reduce functions. All options are delegated to the driver implementation. See node-mongodb-native mapReduce() documentation for more detail about options.

This function does not trigger any middleware.

Example:

```
var o = {};
// `map()` and `reduce()` are run on the MongoDB server, not Node.js,
// these functions are converted to strings
o.map = function () { emit(this.name, 1) };
o.reduce = function (k, vals) { return vals.length };
User.mapReduce(o, function (err, results) {
  console.log(results)
})
```

Other options:

- `query` {Object} query filter object.
- `sort` {Object} sort input objects using this key
- `limit` {Number} max number of documents
- `keeptemp` {Boolean, default:false} keep temporary data
- `finalize` {Function} finalize function

- `scope` {Object} scope variables exposed to map/reduce/finalize during execution
- `jsMode` {Boolean, default:false} it is possible to make the execution stay in JS. Provided in MongoDB > 2.0.X
- `verbose` {Boolean, default:false} provide statistics on job execution time.
- `readPreference` {String}
- `out*` {Object, default: {inline:1}} sets the output target for the map reduce job.

*** out options:**

- `{inline:1}` the results are returned in an array
- `{replace: 'collectionName'}` add the results to collectionName: the results replace the collection
- `{reduce: 'collectionName'}` add the results to collectionName: if dups are detected, uses the reducer / finalize functions
- `{merge: 'collectionName'}` add the results to collectionName: if dups exist the new docs overwrite the old

If `options.out` is set to `replace`, `merge`, or `reduce`, a Model instance is returned that can be used for further querying. Queries run against this model are all executed with the `lean` option; meaning only the js object is returned and no Mongoose magic is applied (getters, setters, etc).

Example:

```
var o = {};
// You can also define `map()` and `reduce()` as strings if your
// linter complains about `emit()` not being defined
o.map = 'function () { emit(this.name, 1) }';
o.reduce = 'function (k, vals) { return vals.length }';
o.out = { replace: 'createdCollectionNameForResults' }
o.verbose = true;

User.mapReduce(o, function (err, model, stats) {
  console.log('map reduce took %d ms', stats.processtime)
  model.find().where('value').gt(10).exec(function (err, docs) {
    console.log(docs);
  });
})

// `mapReduce()` returns a promise. However, ES6 promises can only
// resolve to exactly one value,
o.resolveToObject = true;
var promise = User.mapReduce(o);
promise.then(function (res) {
  var model = res.model;
  var stats = res.stats;
  console.log('map reduce took %d ms', stats.processtime)
  return model.find().where('value').gt(10).exec();
}).then(function (docs) {
```

```
    console.log(docs);
}).then(null, handleError).end()
```

Model.populate()

Parameters

- docs «Document|Array» Either a single document or array of documents to populate.
- options «Object» A hash of key/val (path, options) used for population.
  - [options.retainNullValues=false] «boolean» by default, Mongoose removes null and undefined values from populated arrays. Use this option to make `populate()` retain `null` and `undefined` array entries.
  - [options.getters=false] «boolean» if true, Mongoose will call any getters defined on the `localField`. By default, Mongoose gets the raw value of `localField`. For example, you would need to set this option to `true` if you wanted to add a `lowercase` getter to your `localField`.
- [callback(err,doc)] «Function» Optional callback, executed upon completion. Receives `err` and the `doc(s)`.

Returns:

- «Promise»

Populates document references.

Available top-level options:

- path: space delimited path(s) to populate
- select: optional fields to select
- match: optional query conditions to match
- model: optional name of the model to use for population
- options: optional query options like sort, limit, etc
- justOne: optional boolean, if true Mongoose will always set `path` to an array. Inferred from schema by default.

Examples:

```
// populates a single object
User.findById(id, function (err, user) {
  var opts = [
      { path: 'company', match: { x: 1 }, select: 'name' }
    , { path: 'notes', options: { limit: 10 }, model: 'override' }
  ]
```

```
  User.populate(user, opts, function (err, user) {
    console.log(user);
  });
});

// populates an array of objects
User.find(match, function (err, users) {
  var opts = [{ path: 'company', match: { x: 1 }, select: 'name' }]

  var promise = User.populate(users, opts);
  promise.then(console.log).end();
})

// imagine a Weapon model exists with two saved documents:
//   { _id: 389, name: 'whip' }
//   { _id: 8921, name: 'boomerang' }
// and this schema:
// new Schema({
//   name: String,
//   weapon: { type: ObjectId, ref: 'Weapon' }
// });

var user = { name: 'Indiana Jones', weapon: 389 }
Weapon.populate(user, { path: 'weapon', model: 'Weapon' }, function (err, user) {
  console.log(user.weapon.name) // whip
})

// populate many plain objects
var users = [{ name: 'Indiana Jones', weapon: 389 }]
users.push({ name: 'Batman', weapon: 8921 })
Weapon.populate(users, { path: 'weapon' }, function (err, users) {
  users.forEach(function (user) {
    console.log('%s uses a %s', users.name, user.weapon.name)
    // Indiana Jones uses a whip
    // Batman uses a boomerang
  });

});
// Note that we didn't need to specify the Weapon model because
// it is in the schema's ref
```

Model.prototype.$where

Additional properties to attach to the query when calling `save()` and `isNew` is false.

Model.prototype.$where()

Parameters

- argument «String|Function» is a javascript string or anonymous function

Returns:

- «Query»

Creates a `Query` and specifies a `$where` condition.

Sometimes you need to query for things in mongodb using a JavaScript expression. You can do so via `find({ $where: javascript })`, or you can use the mongoose shortcut method $where via a Query chain or from your mongoose Model.

```
Blog.$where('this.username.indexOf("val") !== -1').exec(function (err, docs) {});
```

Model.prototype.base

Base Mongoose instance the model uses.

Model.prototype.baseModelName

If this is a discriminator model, `baseModelName` is the name of the base model.

Model.prototype.collection

Collection the model uses.

Model.prototype.db

Connection the model uses.

Model.prototype.discriminators

Registered discriminators for this model.

Model.prototype.increment()

Signal that we desire an increment of this documents version.

Example:

```
Model.findById(id, function (err, doc) {
  doc.increment();
  doc.save(function (err) { .. })
})
```

Model.prototype.model()

Parameters

- name «String» model name

Returns another Model instance.

Example:

```
var doc = new Tank;
doc.model('User').findById(id, callback);
```

Model.prototype.modelName

The name of the model

Model.prototype.remove()

Parameters

- [fn] «function(err|product)» optional callback

Returns:

- «Promise» Promise

Removes this document from the db.

Example:

```
product.remove(function (err, product) {
  if (err) return handleError(err);
  Product.findById(product._id, function (err, product) {
    console.log(product) // null
  })
})
```

As an extra measure of flow control, remove will return a Promise (bound to `fn` if passed) so it could be chained, or hooked to recieve errors

Example:

```
product.remove().then(function (product) {
   ...
}).catch(function (err) {
   assert.ok(err)
})
```

Model.prototype.save()

Parameters

- [options] «Object» options optional options
  - [options.safe] «Object» (DEPRECATED) overrides schema's safe option. Use the `w` option instead.
  - [options.validateBeforeSave] «Boolean» set to false to save without validating.
  - [options.w] «Number|String» set the write concern. Overrides the schema-level `writeConcern` option
  - [options.j] «Boolean» set to true for MongoDB to wait until this `save()` has been journaled before resolving the returned promise. Overrides the schema-level `writeConcern` option
  - [options.wtimeout] «Number» sets a timeout for the write concern. Overrides the schema-level `writeConcern` option.
- [fn] «Function» optional callback

Returns:

- «Promise,undefined» Returns undefined if used with callback or a Promise otherwise.

Saves this document.

Example:

```
product.sold = Date.now();
product.save(function (err, product) {
  if (err) ..
})
```

The callback will receive two parameters

1. `err` if an error occurred
2. `product` which is the saved `product`

As an extra measure of flow control, save will return a Promise.

Example:

```
product.save().then(function(product) {
   ...
});
```

Model.prototype.schema

Schema the model uses.

Model.prototypedelete

Alias for remove

Model.remove()

Parameters

- conditions «Object»
- [callback] «Function»

Returns:

- «Query»

Removes all documents that match `conditions` from the collection. To remove just the first document that matches `conditions`, set the `single` option to true.

Example:

```
Character.remove({ name: 'Eddard Stark' }, function (err) {});
```

Note:

This method sends a remove command directly to MongoDB, no Mongoose documents are involved. Because no Mongoose documents are involved, *no middleware (hooks) are executed*.

Model.replaceOne()

Parameters

- conditions «Object»
- doc «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Same as `update()`, except MongoDB replace the existing document with the given document (no atomic operators like `$set`).

This function triggers the following middleware.

- `replaceOne()`

Model.startSession()

Parameters

- [options] «Object» see the mongodb driver options
  - [options.causalConsistency=true] «Boolean» set to false to disable causal consistency
- [callback] «Function»

Returns:

- «Promise<ClientSession>» promise that resolves to a MongoDB driver `ClientSession`

Requires MongoDB >= 3.6.0. Starts a MongoDB session for benefits like causal consistency, retryable writes, and transactions.

Calling `MyModel.startSession()` is equivalent to calling `MyModel.db.startSession()`.

This function does not trigger any middleware.

Example:

```
const session = await Person.startSession();
let doc = await Person.findOne({ name: 'Ned Stark' }, null, { session });
await doc.remove();
// `doc` will always be null, even if reading from a replica set
// secondary. Without causal consistency, it is possible to
// get a doc back from the below query if the query reads from a
// secondary that is experiencing replication lag.
doc = await Person.findOne({ name: 'Ned Stark' }, null, { session, readPreference: 'secc
```

Model.syncIndexes()

Parameters

- [options] «Object» options to pass to `ensureIndexes()`
- [callback] «Function» optional callback

Returns:

- «Promise,undefined» Returns `undefined` if callback is specified, returns a promise if no callback.

Makes the indexes in MongoDB match the indexes defined in this model's schema. This function will drop any indexes that are not defined in the model's schema except the `_id` index, and build any indexes that are in your schema but not in MongoDB.

See the introductory blog post for more information.

Example:

```
const schema = new Schema({ name: { type: String, unique: true } });
const Customer = mongoose.model('Customer', schema);
await Customer.createIndex({ age: 1 }); // Index is not in schema
// Will drop the 'age' index and create an index on `name`
await Customer.syncIndexes();
```

Model.translateAliases()

Parameters

- raw «Object» fields/conditions that may contain aliased keys

Returns:

- «Object» the translated 'pure' fields/conditions

Translate any aliases fields/conditions so the final query or document object is pure

Example:

```
Character
  .find(Character.translateAliases({
    '名': 'Eddard Stark' // Alias for 'name'
  })
  .exec(function(err, characters) {})
```

Note:

Only translate arguments of object type anything else is returned raw

Model.update()

Parameters

- conditions «Object»
- doc «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Updates one document in the database without returning it.

This function triggers the following middleware.

- `update()`

Examples:

```
MyModel.update({ age: { $gt: 18 } }, { oldEnough: true }, fn);
MyModel.update({ name: 'Tobi' }, { ferret: true }, { multi: true }, function (err, raw)
```

```
  if (err) return handleError(err);
  console.log('The raw response from Mongo was ', raw);
});
```

Valid options:

- `safe` (boolean) safe mode (defaults to value set in schema (true))
- `upsert` (boolean) whether to create the doc if it doesn't match (false)
- `multi` (boolean) whether multiple documents should be updated (false)
- `runValidators` : if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` : if this and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created. This option only works on MongoDB >= 2.4 because it relies on MongoDB's `$setOnInsert` operator.
- `strict` (boolean) overrides the `strict` option for this update
- `overwrite` (boolean) disables update-only mode, allowing you to overwrite the doc (false)

All `update` values are cast to their appropriate SchemaTypes before being sent.

The `callback` function receives `(err, rawResponse)` .

- `err` is the error if any occurred
- `rawResponse` is the full response from Mongo

Note:

All top level keys which are not `atomic` operation names are treated as set operations:

Example:

```
var query = { name: 'borne' };
Model.update(query, { name: 'jason bourne' }, options, callback)

// is sent as
Model.update(query, { $set: { name: 'jason bourne' }}, options, callback)
// if overwrite option is false. If overwrite is true, sent without the $set wrapper.
```

This helps prevent accidentally overwriting all documents in your collection with `{ name: 'jason bourne' }` .

Note:

Be careful to not use an existing model instance for the update clause (this won't work and can cause weird behavior like infinite loops). Also, ensure that the update clause does not have an _id property, which causes Mongo to return a "Mod on _id not allowed" error.

Note:

Although values are casted to their appropriate types when using update, the following are *not* applied:

- defaults
- setters
- validators
- middleware

If you need those features, use the traditional approach of first retrieving the document.

```
Model.findOne({ name: 'borne' }, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
})
```

Model.updateMany()

Parameters

- conditions «Object»
- doc «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Same as `update()`, except MongoDB will update *all* documents that match `criteria` (as opposed to just the first one) regardless of the value of the `multi` option.

Note updateMany will *not* fire update middleware. Use `pre('updateMany')` and `post('updateMany')` instead.

This function triggers the following middleware.

- `updateMany()`

Model.updateOne()

Parameters

- conditions «Object»
- doc «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Same as `update()`, except it does not support the `multi` or `overwrite` options.

- MongoDB will update *only* the first document that matches `criteria` regardless of the value of the `multi` option.
- Use `replaceOne()` if you want to overwrite an entire document rather than using atomic operators like `$set`.

This function triggers the following middleware.

- `updateOne()`

Model.watch()

Parameters

- [pipeline] «Array»
- [options] «Object» see the mongodb driver options

Returns:

- «ChangeStream» mongoose-specific change stream wrapper, inherits from EventEmitter

Requires a replica set running MongoDB >= 3.6.0. Watches the underlying collection for changes using MongoDB change streams.

This function does **not** trigger any middleware. In particular, it does **not** trigger aggregate middleware.

The ChangeStream object is an event emitter that emits the following events

- 'change': A change occurred, see below example
- 'error': An unrecoverable error occurred. In particular, change streams currently error out if they lose connection to the replica set primary. Follow this GitHub issue for updates.

- 'end': Emitted if the underlying stream is closed
- 'close': Emitted if the underlying stream is closed

Example:

```
const doc = await Person.create({ name: 'Ned Stark' });
const changeStream = Person.watch().on('change', change => console.log(change));
// Will print from the above `console.log()`:
// { _id: { _data: ... },
//   operationType: 'delete',
//   ns: { db: 'mydb', coll: 'Person' },
//   documentKey: { _id: 5a51b125c5500f5aa094c7bd } }
await doc.remove();
```

Model.where()

Parameters

- path «String»
- [val] «Object» optional value

Returns:

- «Query»

Creates a Query, applies the passed conditions, and returns the Query.

For example, instead of writing:

```
User.find({age: {$gte: 21, $lte: 65}}, callback);
```

we can instead write:

```
User.where('age').gte(21).lte(65).exec(callback);
```

Since the Query class also supports `where` you can continue chaining

```
User
.where('age').gte(21).lte(65)
.where('name', /^b/i)
... etc
```

Query

- Query.prototype.getOptions()
- Query.prototype.getPopulatedPaths()
- Query.prototype.getQuery()
- Query.prototype.getUpdate()
- Query.prototype.gt()
- Query.prototype.gte()
- Query.prototype.hint()
- Query.prototype.in()
- Query.prototype.intersects()
- Query.prototype.j()
- Query.prototype.lean()
- Query.prototype.limit()
- Query.prototype.lt()
- Query.prototype.lte()
- Query.prototype.map()
- Query.prototype.maxDistance()
- Query.prototype.maxScan()
- Query.prototype.maxscan()
- Query.prototype.merge()
- Query.prototype.merge()
- Query.prototype.mod()
- Query.prototype.mongooseOptions()
- Query.prototype.ne()
- Query.prototype.near()
- Query.prototype.nearSphere()
- Query.prototype.nin()
- Query.prototype.nor()
- Query.prototype.or()
- Query.prototype.orFail()
- Query.prototype.polygon()
- Query.prototype.populate()
- Query.prototype.read()
- Query.prototype.readConcern()
- Query.prototype.regex()
- Query.prototype.remove()

- Query.prototype.replaceOne()
- Query.prototype.select()
- Query.prototype.selected()
- Query.prototype.selectedExclusively()
- Query.prototype.selectedInclusively()
- Query.prototype.session()
- Query.prototype.set()
- Query.prototype.setOptions()
- Query.prototype.setQuery()
- Query.prototype.setUpdate()
- Query.prototype.size()
- Query.prototype.skip()
- Query.prototype.slaveOk()
- Query.prototype.slice()
- Query.prototype.snapshot()
- Query.prototype.sort()
- Query.prototype.tailable()
- Query.prototype.then()
- Query.prototype.toConstructor()
- Query.prototype.update()
- Query.prototype.updateMany()
- Query.prototype.updateOne()
- Query.prototype.use$geoWithin
- Query.prototype.w()
- Query.prototype.where()
- Query.prototype.within()
- Query.prototype.wtimeout()

Query()

Parameters

- [options] «Object»
- [model] «Object»
- [conditions] «Object»

- [collection] «Object» Mongoose collection

Query constructor used for building queries. You do not need to instantiate a `Query` directly. Instead use Model functions like `Model.find()`.

Example:

```
const query = MyModel.find(); // `query` is an instance of `Query`
query.setOptions({ lean : true });
query.collection(MyModel.collection);
query.where('age').gte(21).exec(callback);

// You can instantiate a query directly. There is no need to do
// this unless you're an advanced user with a very good reason to.
const query = new mongoose.Query();
```

Query.prototype.$where()

Parameters

- js «String|Function» javascript string or function

Returns:

- «Query» this

Specifies a javascript function or expression to pass to MongoDBs query system.

Example

```
query.$where('this.comments.length === 10 || this.name.length === 5')

// or

query.$where(function () {
  return this.comments.length === 10 || this.name.length === 5;
})
```

NOTE:

Only use `$where` when you have a condition that cannot be met using other MongoDB operators like `$lt`. **Be sure to read about all of its caveats before using.**

Query.prototype.Symbol.asyncIterator()

Returns an asyncIterator for use with `for/await/of` loops This function *only* works for `find()` queries. You do not need to call this function explicitly, the JavaScript runtime will call it for you.

Example

```
for await (const doc of Model.aggregate([{ $sort: { name: 1 } }])) {
  console.log(doc.name);
}
```

Node.js 10.x supports async iterators natively without any flags. You can enable async iterators in Node.js 8.x using the `--harmony_async_iteration` flag.

Note: This function is not if `Symbol.asyncIterator` is undefined. If `Symbol.asyncIterator` is undefined, that means your Node.js version does not support async iterators.

Query.prototype.all()

Parameters

- [path] «String»
- val «Number»

Specifies an $all query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.and()

Parameters

- array «Array» array of conditions

Returns:

- «Query» this

Specifies arguments for a `$and` condition.

Example

```
query.and([{ color: 'green' }, { status: 'ok' }])
```

Query.prototype.batchSize()

Parameters

- val «Number»

Specifies the batchSize option.

Example

```
query.batchSize(100)
```

Note

Cannot be used with `distinct()`

Query.prototype.box()

Parameters

- val «Object»
- Upper «[Array]» Right Coords

Returns:

- «Query» this

Specifies a $box condition

Example

```
var lowerLeft = [40.73083, -73.99756]
var upperRight= [40.741404,  -73.988135]

query.where('loc').within().box(lowerLeft, upperRight)
query.box({ ll : lowerLeft, ur : upperRight })
```

Query.prototype.cast()

Parameters

- [model] «Model» the model to cast to. If not set, defaults to `this.model`
- [obj] «Object»

Returns:

- «Object»

Casts this query to the schema of `model`

Note

If `obj` is present, it is cast instead of this query.

Query.prototype.catch()

Parameters

- [reject] «Function»

Returns:

- «Promise»

Executes the query returning a `Promise` which will be resolved with either the doc(s) or rejected with the error. Like `.then()`, but only takes a rejection handler.

Query.prototype.center()

DEPRECATED Alias for circle

Deprecated. Use circle instead.

Query.prototype.centerSphere()

Parameters

- [path] «String»
- val «Object»

Returns:

- «Query» this

DEPRECATED Specifies a $centerSphere condition

Deprecated. Use circle instead.

Example

```
var area = { center: [50, 50], radius: 10 };
query.where('loc').within().centerSphere(area);
```

Query.prototype.circle()

Parameters

- [path] «String»
- area «Object»

Returns:

- «Query» this

Specifies a $center or $centerSphere condition.

Example

```
var area = { center: [50, 50], radius: 10, unique: true }
query.where('loc').within().circle(area)
// alternatively
query.circle('loc', area);

// spherical calculations
var area = { center: [50, 50], radius: 10, unique: true, spherical: true }
query.where('loc').within().circle(area)
// alternatively
query.circle('loc', area);
```

New in 3.7.0

Query.prototype.collation()

Parameters

- value «Object»

Returns:

- «Query» this

Adds a collation to this op (MongoDB 3.4 and up)

Query.prototype.comment()

Parameters

- val «Number»

Specifies the `comment` option.

Example

```
query.comment('login query')
```

Note

Cannot be used with `distinct()`

Query.prototype.count()

Parameters

- [filter] «Object» count documents that match this object
- [callback] «Function» optional params are (error, count)

Returns:

- «Query» this

Specifies this query as a `count` query.

This method is deprecated. If you want to count the number of documents in a collection, e.g. `count({})`, use the `estimatedDocumentCount()` function instead. Otherwise, use the `countDocuments()` function instead.

Passing a `callback` executes the query.

This function triggers the following middleware.

- `count()`

Example:

```
var countQuery = model.where({ 'color': 'black' }).count();

query.count({ color: 'black' }).count(callback)

query.count({ color: 'black' }, callback)

query.where('color', 'black').count(function (err, count) {
  if (err) return handleError(err);
  console.log('there are %d kittens', count);
})
```

Query.prototype.countDocuments()

Parameters

- [filter] «Object» mongodb selector
- [callback] «Function» optional params are (error, count)

Returns:

- «Query» this

Specifies this query as a `countDocuments()` query. Behaves like `count()`, except it always does a full collection scan when passed an empty filter `{}`.

There are also minor differences in how `countDocuments()` handles `$where` and a couple geospatial operators. versus `count()`.

Passing a `callback` executes the query.

This function triggers the following middleware.

- `countDocuments()`

Example:

```
const countQuery = model.where({ 'color': 'black' }).countDocuments();

query.countDocuments({ color: 'black' }).count(callback);

query.countDocuments({ color: 'black' }, callback);

query.where('color', 'black').countDocuments(function(err, count) {
  if (err) return handleError(err);
  console.log('there are %d kittens', count);
});
```

The `countDocuments()` function is similar to `count()`, but there are a few operators that `countDocuments()` does not support. Below are the operators that `count()` supports but `countDocuments()` does not, and the suggested replacement:

- `$where`: `$expr`
- `$near`: `$geoWithin` with `$center`
- `$nearSphere`: `$geoWithin` with `$centerSphere`

Query.prototype.cursor()

Parameters

- [options] «Object»

Returns:

- «QueryCursor»

Returns a wrapper around a mongodb driver cursor. A QueryCursor exposes a Streams3 interface, as well as a `.next()` function.

The `.cursor()` function triggers pre find hooks, but **not** post find hooks.

Example

```
// There are 2 ways to use a cursor. First, as a stream:
Thing.
  find({ name: /^hello/ }).
  cursor().
  on('data', function(doc) { console.log(doc); }).
  on('end', function() { console.log('Done!'); });

// Or you can use `.next()` to manually get the next doc in the stream.
// `.next()` returns a promise, so you can use promises or callbacks.
var cursor = Thing.find({ name: /^hello/ }).cursor();
cursor.next(function(error, doc) {
  console.log(doc);
});

// Because `.next()` returns a promise, you can use co
// to easily iterate through all documents without loading them

// all into memory.
co(function*() {
  const cursor = Thing.find({ name: /^hello/ }).cursor();
  for (let doc = yield cursor.next(); doc != null; doc = yield cursor.next()) {
    console.log(doc);
  }
});
```

Valid options

- `transform` : optional function which accepts a mongoose document. The return value of the function will be emitted on `data` and returned by `.next()` .

Query.prototype.deleteMany()

Parameters

- [filter] «Object|Query» mongodb selector
- [callback] «Function» optional params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as a `deleteMany()` operation. Works like remove, except it deletes *every* document that matches `criteria` in the collection, regardless of the value of `single` .

This function does not trigger any middleware

Example

```
Character.deleteMany({ name: /Stark/, age: { $gte: 18 } }, callback)
Character.deleteMany({ name: /Stark/, age: { $gte: 18 } }).then(next)
```

Query.prototype.deleteOne()

Parameters

- [filter] «Object|Query» mongodb selector
- [callback] «Function» optional params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as a `deleteOne()` operation. Works like remove, except it deletes at most one document regardless of the `single` option.

This function does not trigger any middleware.

Example

```
Character.deleteOne({ name: 'Eddard Stark' }, callback)
Character.deleteOne({ name: 'Eddard Stark' }).then(next)
```

Query.prototype.distinct()

Parameters

- [field] «String»
- [filter] «Object|Query»
- [callback] «Function» optional params are (error, arr)

Returns:

- «Query» this

Declares or executes a distict() operation.

Passing a `callback` executes the query.

This function does not trigger any middleware.

Example

```
distinct(field, conditions, callback)
distinct(field, conditions)
distinct(field, callback)
distinct(field)
distinct(callback)
distinct()
```

Query.prototype.elemMatch()

Parameters

- path «String|Object|Function»
- criteria «Object|Function»

Returns:

- «Query» this

Specifies an `$elemMatch` condition

Example

```
query.elemMatch('comment', { author: 'autobot', votes: {$gte: 5}})

query.where('comment').elemMatch({ author: 'autobot', votes: {$gte: 5}})

query.elemMatch('comment', function (elem) {
  elem.where('author').equals('autobot');
  elem.where('votes').gte(5);
})

query.where('comment').elemMatch(function (elem) {
  elem.where({ author: 'autobot' });
  elem.where('votes').gte(5);
})
```

Query.prototype.equals()

Parameters

- val «Object»

Returns:

- «Query» this

Specifies the complementary comparison value for paths specified with `where()`

Example

```
User.where('age').equals(49);

// is the same as

User.where('age', 49);
```

Query.prototype.error()

Parameters

- err «Error|null» if set, `exec()` will fail fast before sending the query to MongoDB

Returns:

- «Query» this

Gets/sets the error flag on this query. If this flag is not null or undefined, the `exec()` promise will reject without executing.

Example:

```
Query().error(); // Get current error value
Query().error(null); // Unset the current error
Query().error(new Error('test')); // `exec()` will resolve with test
Schema.pre('find', function() {
  if (!this.getQuery().userId) {
    this.error(new Error('Not allowed to query without setting userId'));
  }
});
```

Note that query casting runs **after** hooks, so cast errors will override custom errors.

Example:

```
var TestSchema = new Schema({ num: Number });
var TestModel = db.model('Test', TestSchema);
TestModel.find({ num: 'not a number' }).error(new Error('woops')).exec(function(error) {
  // `error` will be a cast error because `num` failed to cast
});
```

Query.prototype.estimatedDocumentCount()

Parameters

- [options] «Object» passed transparently to the MongoDB driver
- [callback] «Function» optional params are (error, count)

Returns:

- «Query» this

Specifies this query as a `estimatedDocumentCount()` query. Faster than using `countDocuments()` for large collections because `estimatedDocumentCount()` uses collection metadata rather than scanning the entire collection.

`estimatedDocumentCount()` does **not** accept a filter. `Model.find({ foo: bar }).estimatedDocumentCount()` is equivalent to `Model.find().estimatedDocumentCount()`

This function triggers the following middleware.

- `estimatedDocumentCount()`

Example:

```
await Model.find().estimatedDocumentCount();
```

Query.prototype.exec()

Parameters

- [operation] «String|Function»
- [callback] «Function» optional params depend on the function being called

Returns:

- «Promise»

Executes the query

Examples:

```
var promise = query.exec();
var promise = query.exec('update');

query.exec(callback);
query.exec('find', callback);
```

Query.prototype.exists()

Parameters

- [path] «String»
- val «Number»

Returns:

- «Query» this

Specifies an `$exists` condition

Example

```
// { name: { $exists: true }}
Thing.where('name').exists()
Thing.where('name').exists(true)
```

```
Thing.find().exists('name')

// { name: { $exists: false }}
Thing.where('name').exists(false);
Thing.find().exists('name', false);
```

Query.prototype.explain()

Parameters

- [verbose] «String» The verbosity mode. Either 'queryPlanner', 'executionStats', or 'allPlansExecution'. The default is 'queryPlanner'

Returns:

- «Query» this

Sets the `explain` option, which makes this query return detailed execution stats instead of the actual query result. This method is useful for determining what index your queries use.

Calling `query.explain(v)` is equivalent to `query.setOption({ explain: v })`

Example:

```
const query = new Query();
const res = await query.find({ a: 1 }).explain('queryPlanner');
console.log(res);
```

Query.prototype.find()

Parameters

- [filter] «Object» mongodb selector. If not specified, returns all documents.
- [callback] «Function»

Returns:

- «Query» this

Find all documents that match `selector`. The result will be an array of documents.

If there are too many documents in the result to fit in memory, use `Query.prototype.cursor()`

Example

```
// Using async/await
const arr = await Movie.find({ year: { $gte: 1980, $lte: 1989 } });

// Using callbacks
Movie.find({ year: { $gte: 1980, $lte: 1989 } }, function(err, arr) {});
```

Query.prototype.findOne()

Parameters

- [filter] «Object» mongodb selector
- [projection] «Object» optional fields to return
- [options] «Object» see `setOptions()`
- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Declares the query a findOne operation. When executed, the first found document is passed to the callback.

Passing a `callback` executes the query. The result of the query is a single document.

- *Note:* `conditions` is optional, and if `conditions` is null or undefined, mongoose will send an empty `findOne` command to MongoDB, which will return an arbitrary document. If you're querying by `_id`, use `Model.findById()` instead.

This function triggers the following middleware.

- `findOne()`

Example

```
var query  = Kitten.where({ color: 'white' });
query.findOne(function (err, kitten) {
  if (err) return handleError(err);
  if (kitten) {
    // doc may be null if no document matched
  }
});
```

Query.prototype.findOneAndDelete()

Parameters

- [conditions] «Object»
- [options] «Object»
  - [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Issues a MongoDB findOneAndDelete command.

Finds a matching document, removes it, and passes the found document (if any) to the callback. Executes immediately if `callback` is passed.

This function triggers the following middleware.

- `findOneAndDelete()`

This function differs slightly from `Model.findOneAndRemove()` in that `findOneAndRemove()` becomes a MongoDB `findAndModify()` command, as opposed to a `findOneAndDelete()` command. For most mongoose use cases, this distinction is purely pedantic. You should use `findOneAndDelete()` unless you have a good reason not to.

Available options

- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS` : puts a time limit on the query - requires mongodb >= 2.6.0
- `rawResult` : if true, resolves to the raw result from the MongoDB driver

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if
}
```

Examples

```
A.where().findOneAndDelete(conditions, options, callback) // executes
A.where().findOneAndDelete(conditions, options)  // return Query
A.where().findOneAndDelete(conditions, callback) // executes
```

```
A.where().findOneAndDelete(conditions) // returns Query
A.where().findOneAndDelete(callback)   // executes
A.where().findOneAndDelete()           // returns Query
```

Query.prototype.findOneAndRemove()

Parameters

- [conditions] «Object»
- [options] «Object»
  - [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
- [callback] «Function» optional params are (error, document)

Returns:

- «Query» this

Issues a mongodb findAndModify remove command.

Finds a matching document, removes it, passing the found document (if any) to the callback. Executes immediately if `callback` is passed.

This function triggers the following middleware.

- `findOneAndRemove()`

Available options

- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS` : puts a time limit on the query - requires mongodb >= 2.6.0
- `rawResult` : if true, resolves to the raw result from the MongoDB driver

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if
}
```

Examples

```
A.where().findOneAndRemove(conditions, options, callback) // executes
A.where().findOneAndRemove(conditions, options)  // return Query
A.where().findOneAndRemove(conditions, callback) // executes
A.where().findOneAndRemove(conditions) // returns Query
A.where().findOneAndRemove(callback)   // executes
A.where().findOneAndRemove()           // returns Query
```

Query.prototype.findOneAndUpdate()

Parameters

- [query] «Object|Query»
- [doc] «Object»
- [options] «Object»
  - [options.rawResult] «Boolean» if true, returns the raw result from the MongoDB driver
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
  - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()`.
- [callback] «Function» optional params are (error, doc), *unless* `rawResult` is used, in which case params are (error, writeOpResult)

Returns:

- «Query» this

Issues a mongodb findAndModify update command.

Finds a matching document, updates it according to the `update` arg, passing any `options`, and returns the found document (if any) to the callback. The query executes immediately if `callback` is passed.

This function triggers the following middleware.

- `findOneAndUpdate()`

Available options

- `new`: bool - if true, return the modified document rather than the original. defaults to false (changed in 4.0)
- `upsert`: bool - creates the object if it doesn't exist. defaults to false.

- `fields` : {Object|String} - Field selection. Equivalent to `.select(fields).findOneAndUpdate()`
- `sort` : if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS` : puts a time limit on the query - requires mongodb >= 2.6.0
- `runValidators` : if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` : if this and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created. This option only works on MongoDB >= 2.4 because it relies on MongoDB's `$setOnInsert` operator.
- `rawResult` : if true, returns the raw result from the MongoDB driver
- `context` (string) if set to 'query' and `runValidators` is on, `this` will refer to the query in custom validator functions that update validation runs. Does nothing if `runValidators` is false.

Callback Signature

```
function(error, doc) {
  // error: any errors that occurred
  // doc: the document before updates are applied if `new: false`, or after updates if
}
```

Examples

```
query.findOneAndUpdate(conditions, update, options, callback) // executes
query.findOneAndUpdate(conditions, update, options)  // returns Query
query.findOneAndUpdate(conditions, update, callback) // executes
query.findOneAndUpdate(conditions, update)           // returns Query
query.findOneAndUpdate(update, callback)             // returns Query
query.findOneAndUpdate(update)                       // returns Query
query.findOneAndUpdate(callback)                     // executes
query.findOneAndUpdate()                             // returns Query
```

Query.prototype.geometry()

Parameters

- object «Object» Must contain a `type` property which is a String and a `coordinates` property which is an Array. See the examples.

Returns:

- «Query» this

Specifies a `$geometry` condition

Example

```
var polyA = [[[ 10, 20 ], [ 10, 40 ], [ 30, 40 ], [ 30, 20 ]]]
query.where('loc').within().geometry({ type: 'Polygon', coordinates: polyA })

// or
var polyB = [[ 0, 0 ], [ 1, 1 ]]
query.where('loc').within().geometry({ type: 'LineString', coordinates: polyB })

// or
var polyC = [ 0, 0 ]
query.where('loc').within().geometry({ type: 'Point', coordinates: polyC })

// or
query.where('loc').intersects().geometry({ type: 'Point', coordinates: polyC })
```

The argument is assigned to the most recent path passed to `where()`.

NOTE:

`geometry()` **must** come after either `intersects()` or `within()`.

The `object` argument must contain `type` and `coordinates` properties. - type {String} - coordinates {Array}

Query.prototype.getOptions()

Returns:

- «Object» the options

Gets query options.

Example:

```
var query = new Query();
query.limit(10);
query.setOptions({ maxTimeMS: 1000 })
query.getOptions(); // { limit: 10, maxTimeMS: 1000 }
```

Query.prototype.getPopulatedPaths()

Returns:

- «Array» an array of strings representing populated paths

Gets a list of paths to be populated by this query

Example:

```
bookSchema.pre('findOne', function() {
  let keys = this.getPopulatedPaths(); // ['author']
})
...
Book.findOne({}).populate('author')
```

Query.prototype.getQuery()

Returns:

- «Object» current query conditions

Returns the current query conditions as a JSON object.

Example:

```
var query = new Query();
query.find({ a: 1 }).where('b').gt(2);
query.getQuery(); // { a: 1, b: { $gt: 2 } }
```

Query.prototype.getUpdate()

Returns:

- «Object» current update operations

Returns the current update operations as a JSON object.

Example:

```
var query = new Query();
query.update({}, { $set: { a: 5 } });
query.getUpdate(); // { $set: { a: 5 } }
```

Query.prototype.gt()

Parameters

- [path] «String»
- val «Number»

Specifies a $gt query condition.

When called with one argument, the most recent path passed to `where()` is used.

Example

```
Thing.find().where('age').gt(21)

// or
Thing.find().gt('age', 21)
```

Query.prototype.gte()

Parameters

- [path] «String»
- val «Number»

Specifies a $gte query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.hint()

Parameters

- val «Object» a hint object

Returns:

- «Query» this

Sets query hints.

Example

```
query.hint({ indexA: 1, indexB: -1})
```

Note

Cannot be used with `distinct()`

Query.prototype.in()

Parameters

- [path] «String»
- val «Number»

Specifies an $in query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.intersects()

Parameters

- [arg] «Object»

Returns:

- «Query» this

Declares an intersects query for `geometry()`.

Example

```
query.where('path').intersects().geometry({
    type: 'LineString'
  , coordinates: [[180.0, 11.0], [180, 9.0]]
})

query.where('path').intersects({
    type: 'LineString'
  , coordinates: [[180.0, 11.0], [180, 9.0]]
})
```

NOTE:

MUST be used after `where()`.

NOTE:

In Mongoose 3.7, `intersects` changed from a getter to a function. If you need the old syntax, use this.

Query.prototype.j()

Parameters

- val «boolean»

Returns:

- «Query» this

Requests acknowledgement that this operation has been persisted to MongoDB's on-disk journal.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern.j` option

Example:

```
await mongoose.model('Person').deleteOne({ name: 'Ned Stark' }).j(true);
```

Query.prototype.lean()

Parameters

- bool «Boolean|Object» defaults to true

Returns:

- «Query» this

Sets the lean option.

Documents returned from queries with the `lean` option enabled are plain javascript objects, not MongooseDocuments. They have no `save` method, getters/setters or other Mongoose magic applied.

Example:

```
new Query().lean() // true
new Query().lean(true)
new Query().lean(false)

Model.find().lean().exec(function (err, docs) {
  docs[0] instanceof mongoose.Document // false
});
```

This is a great option in high-performance read-only scenarios, especially when combined with stream.

Query.prototype.limit()

Parameters

- val «Number»

Specifies the maximum number of documents the query will return.

Example

```
query.limit(20)
```

Note

Cannot be used with `distinct()`

Query.prototype.lt()

Parameters

- [path] «String»
- val «Number»

Specifies a $lt query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.lte()

Parameters

- [path] «String»
- val «Number»

Specifies a $lte query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.map()

Query.prototype.maxDistance()

Parameters

- [path] «String»
- val «Number»

Specifies a $maxDistance query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.maxScan()

Parameters

- val «Number»

Specifies the maxScan option.

Example

```
query.maxScan(100)
```

Note

Cannot be used with `distinct()`

Query.prototype.maxscan()

DEPRECATED Alias of `maxScan`

Query.prototype.merge()

Parameters

- source «Query|Object»

Returns:

- «Query» this

Merges another Query or conditions object into this one.

When a Query is passed, conditions, field selection and options are merged.

Query.prototype.merge()

Parameters

- source «Query|Object»

Returns:

- «Query» this

Merges another Query or conditions object into this one.

When a Query is passed, conditions, field selection and options are merged.

New in 3.7.0

Query.prototype.mod()

Parameters

- [path] «String»
- val «Array» must be of length 2, first element is `divisor`, 2nd element is `remainder`.

Returns:

- «Query» this

Specifies a `$mod` condition, filters documents for documents whose `path` property is a number that is equal to `remainder` modulo `divisor`.

Example

```
// All find products whose inventory is odd
Product.find().mod('inventory', [2, 1]);
Product.find().where('inventory').mod([2, 1]);
// This syntax is a little strange, but supported.
Product.find().where('inventory').mod(2, 1);
```

Query.prototype.mongooseOptions()

Parameters

- options «Object» if specified, overwrites the current options

Returns:

- «Object» the options

Getter/setter around the current mongoose-specific options for this query Below are the current Mongoose-specific options.

- `populate`: an array representing what paths will be populated. Should have one entry for each call to `Query.prototype.populate()`
- `lean`: if truthy, Mongoose will not hydrate any documents that are returned from this query. See `Query.prototype.lean()` for more information.
- `strict`: controls how Mongoose handles keys that aren't in the schema for updates. This option is `true` by default, which means Mongoose will silently strip any paths in the update that aren't in the schema. See the `strict` mode docs for more information.

- `strictQuery` : controls how Mongoose handles keys that aren't in the schema for the query `filter` . This option is `false` by default for backwards compatibility, which means Mongoose will allow `Model.find({ foo: 'bar' })` even if `foo` is not in the schema. See the `strictQuery` docs for more information.
- `useFindAndModify` : used to work around the `findAndModify()` deprecation warning
- `omitUndefined` : delete any properties whose value is `undefined` when casting an update. In other words, if this is set, Mongoose will delete `baz` from the update in `Model.updateOne({}, { foo: 'bar', baz: undefined })` before sending the update to the server.
- `nearSphere` : use `$nearSphere` instead of `near()` . See the `Query.prototype.nearSphere()` docs

Mongoose maintains a separate object for internal options because Mongoose sends `Query.prototype.options` to the MongoDB server, and the above options are not relevant for the MongoDB server.

Query.prototype.ne()

Parameters

- [path] «String»
- val «Number»

Specifies a $ne query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.near()

Parameters

- [path] «String»
- val «Object»

Returns:

- «Query» this

Specifies a `$near` or `$nearSphere` condition

These operators return documents sorted by distance.

Example

```
query.where('loc').near({ center: [10, 10] });
query.where('loc').near({ center: [10, 10], maxDistance: 5 });
query.where('loc').near({ center: [10, 10], maxDistance: 5, spherical: true });
query.near('loc', { center: [10, 10], maxDistance: 5 });
```

Query.prototype.nearSphere()

DEPRECATED Specifies a `$nearSphere` condition

Example

```
query.where('loc').nearSphere({ center: [10, 10], maxDistance: 5 });
```

Deprecated. Use `query.near()` instead with the `spherical` option set to `true`.

Example

```
query.where('loc').near({ center: [10, 10], spherical: true });
```

Query.prototype.nin()

Parameters

- [path] «String»
- val «Number»

Specifies an $nin query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.nor()

Parameters

- array «Array» array of conditions

Returns:

- «Query» this

Specifies arguments for a `$nor` condition.

Example

```
query.nor([{ color: 'green' }, { status: 'ok' }])
```

Query.prototype.or()

Parameters

- array «Array» array of conditions

Returns:

- «Query» this

Specifies arguments for an `$or` condition.

Example

```
query.or([{ color: 'red' }, { status: 'emergency' }])
```

Query.prototype.orFail()

Parameters

- [err] «Function|Error» optional error to throw if no docs match `filter`

Returns:

- «Query» this

Make this query throw an error if no documents match the given `filter`. This is handy for integrating with async/await, because `orFail()` saves you an extra `if` statement to check if no document was found.

Example:

```
// Throws if no doc returned
await Model.findOne({ foo: 'bar' }).orFail();

// Throws if no document was updated
await Model.updateOne({ foo: 'bar' }, { name: 'test' }).orFail();
```

```
// Throws "No docs found!" error if no docs match `{ foo: 'bar' }`
await Model.find({ foo: 'bar' }).orFail(new Error('No docs found!'));

// Throws "Not found" error if no document was found
await Model.findOneAndUpdate({ foo: 'bar' }, { name: 'test' }).
  orFail(() => Error('Not found'));
```

Query.prototype.polygon()

Parameters

- [path] «String|Array»
    - [coordinatePairs...] «Array|Object»

Returns:

- «Query» this

Specifies a $polygon condition

Example

```
query.where('loc').within().polygon([10,20], [13, 25], [7,15])
query.polygon('loc', [10,20], [13, 25], [7,15])
```

Query.prototype.populate()

Parameters

- path «Object|String» either the path to populate or an object specifying all parameters
- [select] «Object|String» Field selection for the population query
- [model] «Model» The model you wish to use for population. If not specified, populate will look up the model by the name in the Schema's `ref` field.
- [match] «Object» Conditions for the population query
- [options] «Object» Options for the population query (sort, etc)

Returns:

- «Query» this

Specifies paths which should be populated with other documents.

Example:

```
Kitten.findOne().populate('owner').exec(function (err, kitten) {
  console.log(kitten.owner.name) // Max
})

Kitten.find().populate({
    path: 'owner'
  , select: 'name'
  , match: { color: 'black' }
  , options: { sort: { name: -1 }}
}).exec(function (err, kittens) {
  console.log(kittens[0].owner.name) // Zoopa
})

// alternatively
Kitten.find().populate('owner', 'name', null, {sort: { name: -1 }}).exec(function (err,
  console.log(kittens[0].owner.name) // Zoopa
})
```

Paths are populated after the query executes and a response is received. A separate query is then executed for each path specified for population. After a response for each query has also been returned, the results are passed to the callback.

Query.prototype.read()

Parameters

- pref «String» one of the listed preference options or aliases
- [tags] «Array» optional tags for this query

Returns:

- «Query» this

Determines the MongoDB nodes from which to read.

Preferences:

```
primary - (default) Read from primary only. Operations will produce an error if primary
secondary           Read from secondary if available, otherwise error.
primaryPreferred    Read from primary if available, otherwise a secondary.
secondaryPreferred  Read from a secondary if available, otherwise read from the primary
nearest             All operations read from among the nearest candidates, but unlike c
```

Aliases

```
p   primary
pp  primaryPreferred
s   secondary
sp  secondaryPreferred
n   nearest
```

Example:

```
new Query().read('primary')
new Query().read('p')  // same as primary

new Query().read('primaryPreferred')
new Query().read('pp') // same as primaryPreferred

new Query().read('secondary')
new Query().read('s')  // same as secondary

new Query().read('secondaryPreferred')
new Query().read('sp') // same as secondaryPreferred

new Query().read('nearest')
new Query().read('n')  // same as nearest

// read from secondaries with matching tags
new Query().read('s', [{ dc:'sf', s: 1 },{ dc:'ma', s: 2 }])
```

Read more about how to use read preferrences here and here.

Query.prototype.readConcern()

Parameters

- level «String» one of the listed read concern level or their aliases

Returns:

- «Query» this

Sets the readConcern option for the query.

Example:

```
new Query().readConcern('local')
new Query().readConcern('l')  // same as local

new Query().readConcern('available')
new Query().readConcern('a')  // same as available
```

```
new Query().readConcern('majority')
new Query().readConcern('m')  // same as majority

new Query().readConcern('linearizable')
new Query().readConcern('lz') // same as linearizable

new Query().readConcern('snapshot')
new Query().readConcern('s')  // same as snapshot
```

Read Concern Level:

```
local         MongoDB 3.2+ The query returns from the instance with no guarantee guarant
available     MongoDB 3.6+ The query returns from the instance with no guarantee guarant
majority      MongoDB 3.2+ The query returns the data that has been acknowledged by a ma
linearizable  MongoDB 3.4+ The query returns data that reflects all successful majority-
snapshot      MongoDB 4.0+ Only available for operations within multi-document transacti
```

Aliases

```
l   local
a   available
m   majority
lz  linearizable
s   snapshot
```

Read more about how to use read concern here.

Query.prototype.regex()

Parameters

- [path] «String»
- val «String|RegExp»

Specifies a $regex query condition.

When called with one argument, the most recent path passed to `where()` is used.

Query.prototype.remove()

Parameters

- [filter] «Object|Query» mongodb selector

- [callback] «Function» optional params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as a remove() operation.

This function does not trigger any middleware

Example

```
Model.remove({ artist: 'Anne Murray' }, callback)
```

Note

The operation is only executed when a callback is passed. To force execution without a callback, you must first call `remove()` and then execute it by using the `exec()` method.

```
// not executed
var query = Model.find().remove({ name: 'Anne Murray' })

// executed
query.remove({ name: 'Anne Murray' }, callback)
query.remove({ name: 'Anne Murray' }).remove(callback)

// executed without a callback
query.exec()

// summary
query.remove(conds, fn); // executes
query.remove(conds)
query.remove(fn) // executes
query.remove()
```

Query.prototype.replaceOne()

Parameters

- [criteria] «Object»
- [doc] «Object» the update command
- [options] «Object»
- [callback] «Function» optional params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as a replaceOne() operation. Same as `update()`, except MongoDB will replace the existing document and will not accept any atomic operators (`$set`, etc.)

Note replaceOne will *not* fire update middleware. Use `pre('replaceOne')` and `post('replaceOne')` instead.

This function triggers the following middleware.

- `replaceOne()`

Query.prototype.select()

Parameters

- arg «Object|String»

Returns:

- «Query» this

Specifies which document fields to include or exclude (also known as the query "projection")

When using string syntax, prefixing a path with `-` will flag that path as excluded. When a path does not have the `-` prefix, it is included. Lastly, if a path is prefixed with `+`, it forces inclusion of the path, which is useful for paths excluded at the schema level.

A projection *must* be either inclusive or exclusive. In other words, you must either list the fields to include (which excludes all others), or list the fields to exclude (which implies all other fields are included). The `_id` field is the only exception because MongoDB includes it by default.

Example

```
// include a and b, exclude other fields
query.select('a b');

// exclude c and d, include other fields
query.select('-c -d');

// Use `+` to override schema-level `select: false` without making the
// projection inclusive.
const schema = new Schema({
  foo: { type: String, select: false },
  bar: String
});
// ...
```

```
query.select('+foo'); // Override foo's `select: false` without excluding `bar`

// or you may use object notation, useful when
// you have keys already prefixed with a "-"
query.select({ a: 1, b: 1 });
query.select({ c: 0, d: 0 });
```

Query.prototype.selected()

Returns:

- «Boolean»

Determines if field selection has been made.

Query.prototype.selectedExclusively()

Returns:

- «Boolean»

Determines if exclusive field selection has been made.

```
query.selectedExclusively() // false
query.select('-name')
query.selectedExclusively() // true
query.selectedInclusively() // false
```

Query.prototype.selectedInclusively()

Returns:

- «Boolean»

Determines if inclusive field selection has been made.

```
query.selectedInclusively() // false
query.select('name')
query.selectedInclusively() // true
```

Query.prototype.session()

Parameters

- [session] «ClientSession» from `await conn.startSession()`

Returns:

- «Query» this

Sets the MongoDB session associated with this query. Sessions are how you mark a query as part of a transaction.

Calling `session(null)` removes the session from this query.

Example:

```
const s = await mongoose.startSession();
await mongoose.model('Person').findOne({ name: 'Axl Rose' }).session(s);
```

Query.prototype.set()

Parameters

- path «String|Object» path or object of key/value pairs to set
- [val] «Any» the value to set

Returns:

- «Query» this

Adds a `$set` to this query's update without changing the operation. This is useful for query middleware so you can add an update regardless of whether you use `updateOne()`, `updateMany()`, `findOneAndUpdate()`, etc.

Example:

```
// Updates `{ $set: { updatedAt: new Date() } }`
new Query().updateOne({}, {}).set('updatedAt', new Date());
new Query().updateMany({}, {}).set({ updatedAt: new Date() });
```

Query.prototype.setOptions()

Parameters

- options «Object»

Returns:

- «Query» this

Sets query options. Some options only make sense for certain operations.

Options:

The following options are only for `find()`:

- tailable
- sort
- limit
- skip
- maxscan
- batchSize
- comment
- snapshot
- readPreference
- hint

The following options are only for write operations: `update()`, `updateOne()`, `updateMany()`, `replaceOne()`, `findOneAndUpdate()`, and `findByIdAndUpdate()`:

- upsert
- writeConcern
- timestamps: If `timestamps` is set in the schema, set this option to `false` to skip timestamps for that particular update. Has no effect if `timestamps` is not enabled in the schema options.

The following options are only for `find()`, `findOne()`, `findById()`, `findOneAndUpdate()`, and `findByIdAndUpdate()`:

- lean

The following options are only for all operations **except** `update()`, `updateOne()`, `updateMany()`, `remove()`, `deleteOne()`, and `deleteMany()`:

- maxTimeMS

The following options are for all operations

- collation
- session

Query.prototype.setQuery()

Parameters

- new «Object» query conditions

Returns:

- «undefined»

Sets the query conditions to the provided JSON object.

Example:

```
var query = new Query();
query.find({ a: 1 })
query.setQuery({ a: 2 });
query.getQuery(); // { a: 2 }
```

Query.prototype.setUpdate()

Parameters

- new «Object» update operation

Returns:

- «undefined»

Sets the current update operation to new value.

Example:

```
var query = new Query();
query.update({}, { $set: { a: 5 } });
query.setUpdate({ $set: { b: 6 } });
query.getUpdate(); // { $set: { b: 6 } }
```

Query.prototype.size()

Parameters

- [path] «String»

- val «Number»

Specifies a $size query condition.

When called with one argument, the most recent path passed to `where()` is used.

Example

```
MyModel.where('tags').size(0).exec(function (err, docs) {
  if (err) return handleError(err);

  assert(Array.isArray(docs));
  console.log('documents with 0 tags', docs);
})
```

Query.prototype.skip()

Parameters

- val «Number»

Specifies the number of documents to skip.

Example

```
query.skip(100).limit(20)
```

Note

Cannot be used with `distinct()`

Query.prototype.slaveOk()

Parameters

- v «Boolean» defaults to true

Returns:

- «Query» this

DEPRECATED Sets the slaveOk option.

Deprecated in MongoDB 2.2 in favor of read preferences.

Example:

```
query.slaveOk() // true
query.slaveOk(true)
query.slaveOk(false)
```

Query.prototype.slice()

Parameters

- [path] «String»
- val «Number» number/range of elements to slice

Returns:

- «Query» this

Specifies a $slice projection for an array.

Example

```
query.slice('comments', 5)
query.slice('comments', -5)
query.slice('comments', [10, 5])
query.where('comments').slice(5)
query.where('comments').slice([-10, 5])
```

Query.prototype.snapshot()

Returns:

- «Query» this

Specifies this query as a `snapshot` query.

Example

```
query.snapshot() // true
query.snapshot(true)
query.snapshot(false)
```

Note

Cannot be used with `distinct()`

Query.prototype.sort()

Parameters

- arg «Object|String»

Returns:

- «Query» this

Sets the sort order

If an object is passed, values allowed are `asc`, `desc`, `ascending`, `descending`, `1`, and `-1`.

If a string is passed, it must be a space delimited list of path names. The sort order of each path is ascending unless the path name is prefixed with `-` which will be treated as descending.

Example

```
// sort by "field" ascending and "test" descending
query.sort({ field: 'asc', test: -1 });

// equivalent
query.sort('field -test');
```

Note

Cannot be used with `distinct()`

Query.prototype.tailable()

Parameters

- bool «Boolean» defaults to true
- [opts] «Object» options to set
    - [opts.numberOfRetries] «Number» if cursor is exhausted, retry this many times before giving up
    - [opts.tailableRetryInterval] «Number» if cursor is exhausted, wait this many milliseconds before retrying

Sets the tailable option (for use with capped collections).

Example

```
query.tailable() // true
query.tailable(true)
query.tailable(false)
```

Note

Cannot be used with `distinct()`

Query.prototype.then()

Parameters

- [resolve] «Function»
- [reject] «Function»

Returns:

- «Promise»

Executes the query returning a `Promise` which will be resolved with either the doc(s) or rejected with the error.

Query.prototype.toConstructor()

Returns:

- «Query» subclass-of-Query

Converts this query to a customized, reusable query constructor with all arguments and options retained.

Example

```
// Create a query for adventure movies and read from the primary
// node in the replica-set unless it is down, in which case we'll
// read from a secondary node.
var query = Movie.find({ tags: 'adventure' }).read('primaryPreferred');
```

```
// create a custom Query constructor based off these settings
var Adventure = query.toConstructor();

// Adventure is now a subclass of mongoose.Query and works the same way but with the
// default query parameters and options set.
Adventure().exec(callback)

// further narrow down our query results while still using the previous settings
Adventure().where({ name: /^Life/ }).exec(callback);

// since Adventure is a stand-alone constructor we can also add our own
// helper methods and getters without impacting global queries
Adventure.prototype.startsWith = function (prefix) {
  this.where({ name: new RegExp('^' + prefix) })
  return this;
}
Object.defineProperty(Adventure.prototype, 'highlyRated', {
  get: function () {
    this.where({ rating: { $gt: 4.5 }});
    return this;
  }
})
Adventure().highlyRated.startsWith('Life').exec(callback)
```

New in 3.7.3

Query.prototype.update()

Parameters

- [criteria] «Object»
- [doc] «Object» the update command
- [options] «Object»
  - [options.multipleCastError] «Boolean» by default, mongoose only returns the first error that occurred in casting the query. Turn on this option to aggregate all the cast errors.
- [callback] «Function» optional, params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as an update() operation.

All paths passed that are not $atomic operations will become $set ops.

This function triggers the following middleware.

- `update()`

Example

```
Model.where({ _id: id }).update({ title: 'words' })

// becomes

Model.where({ _id: id }).update({ $set: { title: 'words' }})
```

Valid options:

- `upsert` (boolean) whether to create the doc if it doesn't match (false)
- `multi` (boolean) whether multiple documents should be updated (false)
- `runValidators` : if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` : if this and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created. This option only works on MongoDB >= 2.4 because it relies on MongoDB's `$setOnInsert` operator.
- `strict` (boolean) overrides the `strict` option for this update
- `overwrite` (boolean) disables update-only mode, allowing you to overwrite the doc (false)
- `context` (string) if set to 'query' and `runValidators` is on, `this` will refer to the query in custom validator functions that update validation runs. Does nothing if `runValidators` is false.
- `read`
- `writeConcern`

Note

Passing an empty object `{}` as the doc will result in a no-op unless the `overwrite` option is passed. Without the `overwrite` option set, the update operation will be ignored and the callback executed without sending the command to MongoDB so as to prevent accidently overwritting documents in the collection.

Note

The operation is only executed when a callback is passed. To force execution without a callback, we must first call update() and then execute it by using the `exec()` method.

```
var q = Model.where({ _id: id });
q.update({ $set: { name: 'bob' }}).update(); // not executed

q.update({ $set: { name: 'bob' }}).exec(); // executed

// keys that are not $atomic ops become $set.
```

```
// keys that are not $atomic ops become $set.
// this executes the same command as the previous example.
q.update({ name: 'bob' }).exec();

// overwriting with empty docs
var q = Model.where({ _id: id }).setOptions({ overwrite: true })
q.update({ }, callback); // executes

// multi update with overwrite to empty doc
var q = Model.where({ _id: id });
q.setOptions({ multi: true, overwrite: true })
q.update({ });
q.update(callback); // executed

// multi updates
Model.where()
     .update({ name: /^match/ }, { $set: { arr: [] }}, { multi: true }, callback)

// more multi updates
Model.where()
     .setOptions({ multi: true })
     .update({ $set: { arr: [] }}, callback)

// single update by default
Model.where({ email: 'address@example.com' })
     .update({ $inc: { counter: 1 }}, callback)
```

API summary

```
update(criteria, doc, options, cb) // executes
update(criteria, doc, options)
update(criteria, doc, cb) // executes
update(criteria, doc)
update(doc, cb) // executes
update(doc)
update(cb) // executes
update(true) // executes
update()
```

Query.prototype.updateMany()

Parameters

- [criteria] «Object»
- [doc] «Object» the update command
- [options] «Object»
- [callback] «Function» optional params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as an updateMany() operation. Same as `update()`, except MongoDB will update *all* documents that match `criteria` (as opposed to just the first one) regardless of the value of the `multi` option.

Note updateMany will *not* fire update middleware. Use `pre('updateMany')` and `post('updateMany')` instead.

This function triggers the following middleware.

- `updateMany()`

Query.prototype.updateOne()

Parameters

- [criteria] «Object»
- [doc] «Object» the update command
- [options] «Object»
- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query» this

Declare and/or execute this query as an updateOne() operation. Same as `update()`, except it does not support the `multi` or `overwrite` options.

- MongoDB will update *only* the first document that matches `criteria` regardless of the value of the `multi` option.
- Use `replaceOne()` if you want to overwrite an entire document rather than using atomic operators like `$set`.

Note updateOne will *not* fire update middleware. Use `pre('updateOne')` and `post('updateOne')` instead.

This function triggers the following middleware.

- `updateOne()`

Query.prototype.use$geoWithin

Flag to opt out of using `$geoWithin`.

```
mongoose.Query.use$geoWithin = false;
```

MongoDB 2.4 deprecated the use of `$within`, replacing it with `$geoWithin`. Mongoose uses `$geoWithin` by default (which is 100% backward compatible with $within). If you are running an older version of MongoDB, set this flag to `false` so your `within()` queries continue to work.

Query.prototype.w()

Parameters

- val «String|number» 0 for fire-and-forget, 1 for acknowledged by one server, 'majority' for majority of the replica set, or any of the more advanced options.

Returns:

- «Query» this

Sets the specified number of `mongod` servers, or tag set of `mongod` servers, that must acknowledge this write before this write is considered successful.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern.w` option

Example:

```
// The 'majority' option means the `deleteOne()` promise won't resolve
// until the `deleteOne()` has propagated to the majority of the replica set
await mongoose.model('Person').
  deleteOne({ name: 'Ned Stark' }).
  w('majority');
```

Query.prototype.where()

Parameters

- [path] «String|Object»
- [val] «any»

Returns:

- «Query» this

Specifies a `path` for use with chaining.

Example

```
// instead of writing:
User.find({age: {$gte: 21, $lte: 65}}, callback);

// we can instead write:
User.where('age').gte(21).lte(65);

// passing query conditions is permitted
User.find().where({ name: 'vonderful' })

// chaining
User
.where('age').gte(21).lte(65)
.where('name', /^vonderful/i)
.where('friends').slice(10)
.exec(callback)
```

Query.prototype.within()

Returns:

- «Query» this

Defines a `$within` or `$geoWithin` argument for geo-spatial queries.

Example

```
query.where(path).within().box()
query.where(path).within().circle()
query.where(path).within().geometry()

query.where('loc').within({ center: [50,50], radius: 10, unique: true, spherical: true }
```

```
query.where('loc').within({ box: [[40.73, -73.9], [40.7, -73.988]] });
query.where('loc').within({ polygon: [[],[],[],[]] });

query.where('loc').within([], [], []) // polygon
query.where('loc').within([], []) // box
query.where('loc').within({ type: 'LineString', coordinates: [...] }); // geometry
```

MUST be used after `where()`.

NOTE:

As of Mongoose 3.7, `$geoWithin` is always used for queries. To change this behavior, see Query.use$geoWithin.

NOTE:

In Mongoose 3.7, `within` changed from a getter to a function. If you need the old syntax, use this.

Query.prototype.wtimeout()

Parameters

- ms «number» number of milliseconds to wait

Returns:

- «Query» this

If `w > 1`, the maximum amount of time to wait for this write to propagate through the replica set before this operation fails. The default is `0`, which means no timeout.

This option is only valid for operations that write to the database

- `deleteOne()`
- `deleteMany()`
- `findOneAndDelete()`
- `findOneAndUpdate()`
- `remove()`
- `update()`
- `updateOne()`
- `updateMany()`

Defaults to the schema's `writeConcern.wtimeout` option

Example:

```
// The `deleteOne()` promise won't resolve until this `deleteOne()` has
// propagated to at least `w = 2` members of the replica set. If it takes
// longer than 1 second, this `deleteOne()` will fail.
await mongoose.model('Person').
  deleteOne({ name: 'Ned Stark' }).
  w(2).
  wtimeout(1000);
```

QueryCursor

- QueryCursor()
- QueryCursor.prototype.addCursorFlag()
- QueryCursor.prototype.close()
- QueryCursor.prototype.eachAsync()
- QueryCursor.prototype.map()
- QueryCursor.prototype.next()

QueryCursor()

Parameters

- query «Query»
- options «Object» query options passed to `.find()`

A QueryCursor is a concurrency primitive for processing query results one document at a time. A QueryCursor fulfills the Node.js streams3 API, in addition to several other mechanisms for loading documents from MongoDB one at a time.

QueryCursors execute the model's pre find hooks, but **not** the model's post find hooks.

Unless you're an advanced user, do **not** instantiate this class directly. Use `Query#cursor()` instead.

QueryCursor.prototype.addCursorFlag()

Parameters

- flag «String»
- value «Boolean»

Returns:

- «AggregationCursor» this

Adds a cursor flag. Useful for setting the `noCursorTimeout` and `tailable` flags.

QueryCursor.prototype.close()

Parameters

- callback «Function»

Returns:

- «Promise»

Marks this cursor as closed. Will stop streaming and subsequent calls to `next()` will error.

QueryCursor.prototype.eachAsync()

Parameters

- fn «Function»
- [options] «Object»
    - [options.parallel] «Number» the number of promises to execute in parallel. Defaults to 1.
- [callback] «Function» executed when all docs have been processed

Returns:

- «Promise»

Execute `fn` for every document in the cursor. If `fn` returns a promise, will wait for the promise to resolve before iterating on to the next one. Returns a promise that resolves when done.

QueryCursor.prototype.map()

Parameters

- fn «Function»

Returns:

- «QueryCursor»

Registers a transform function which subsequently maps documents retrieved via the streams interface or `.next()`

Example

```
// Map documents returned by `data` events
Thing.
  find({ name: /^hello/ }).
  cursor().
  map(function (doc) {
   doc.foo = "bar";
   return doc;
  })
  on('data', function(doc) { console.log(doc.foo); });

// Or map documents returned by `.next()`
var cursor = Thing.find({ name: /^hello/ }).
  cursor().
  map(function (doc) {
    doc.foo = "bar";
    return doc;
  });
cursor.next(function(error, doc) {
  console.log(doc.foo);
});
```

QueryCursor.prototype.next()

Parameters

- callback «Function»

Returns:

- «Promise»

Get the next document from this cursor. Will return `null` when there are no documents left.

Aggregate

- Aggregate.prototype.sortByCount()
- Aggregate.prototype.then()
- Aggregate.prototype.unwind()

Aggregate()

Parameters

- [pipeline] «Array» aggregation pipeline as an array of objects

Aggregate constructor used for building aggregation pipelines. Do not instantiate this class directly, use Model.aggregate() instead.

Example:

```
const aggregate = Model.aggregate([
  { $project: { a: 1, b: 1 } },
  { $skip: 5 }
]);

Model.
  aggregate([{ $match: { age: { $gte: 21 }}}]).
  unwind('tags').
  exec(callback);
```

Note:

- The documents returned are plain javascript objects, not mongoose documents (since any shape of document can be returned).
- Mongoose does **not** cast pipeline stages. The below will **not** work unless `_id` is a string in the database

```
  new Aggregate([{ $match: { _id: '000000000000000000000000a' } }]);
  // Do this instead to cast to an ObjectId
  new Aggregate([{ $match: { _id: mongoose.Types.ObjectId('000000000000000000000000a') } }]);
```

Aggregate.prototype.Symbol.asyncIterator()

Returns an asyncIterator for use with `for/await/of` loops This function *only* works for `find()` queries. You do not need to call this function explicitly, the JavaScript runtime will call it for you.

Example

```
for await (const doc of Model.find().sort({ name: 1 })) {
  console.log(doc.name);
}
```

Node.js 10.x supports async iterators natively without any flags. You can enable async iterators in Node.js 8.x using the `--harmony_async_iteration` flag.

Note: This function is not if `Symbol.asyncIterator` is undefined. If `Symbol.asyncIterator` is undefined, that means your Node.js version does not support async iterators.

Aggregate.prototype.addCursorFlag()

Parameters

- flag «String»
- value «Boolean»

Returns:

- «Aggregate» this

Sets an option on this aggregation. This function will be deprecated in a future release. Use the `cursor()`, `collation()`, etc. helpers to set individual options, or access `agg.options` directly.

Note that MongoDB aggregations do **not** support the `noCursorTimeout` flag, if you try setting that flag with this function you will get a "unrecognized field 'noCursorTimeout'" error.

Aggregate.prototype.addFields()

Parameters

- arg «Object» field specification

Returns:

- «Aggregate»

Appends a new $addFields operator to this aggregate pipeline. Requires MongoDB v3.4+ to work

Examples:

```
// adding new fields based on existing fields
aggregate.addFields({
    newField: '$b.nested'
  , plusTen: { $add: ['$val', 10]}
  , sub: {
       name: '$a'
    }
})

// etc
aggregate.addFields({ salary_k: { $divide: [ "$salary", 1000 ] } });
```

Aggregate.prototype.allowDiskUse()

Parameters

- value «Boolean» Should tell server it can use hard drive to store data during aggregation.
- [tags] «Array» optional tags for this query

Sets the allowDiskUse option for the aggregation query (ignored for < 2.6.0)

Example:

```
await Model.aggregate([{ $match: { foo: 'bar' } }]).allowDiskUse(true);
```

Aggregate.prototype.append()

Parameters

- ops «Object» operator(s) to append

Returns:

- «Aggregate»

Appends new operators to this aggregate pipeline

Examples:

```
aggregate.append({ $project: { field: 1 }}, { $limit: 2 });

// or pass an array
```

```
var pipeline = [{ $match: { daw: 'Logic Audio X' }} ];
aggregate.append(pipeline);
```

Aggregate.prototype.collation()

Parameters

- collation «Object» options

Returns:

- «Aggregate» this

Adds a collation

Example:

```
Model.aggregate(..).collation({ locale: 'en_US', strength: 1 }).exec();
```

Aggregate.prototype.count()

Parameters

- the «String» name of the count field

Returns:

- «Aggregate»

Appends a new $count operator to this aggregate pipeline.

Examples:

```
aggregate.count("userCount");
```

Aggregate.prototype.cursor()

Parameters

- options «Object»

- options.batchSize «Number» set the cursor batch size
    - [options.useMongooseAggCursor] «Boolean» use experimental mongoose-specific aggregation cursor (for `eachAsync()` and other query cursor semantics)

Returns:

- «Aggregate» this

Sets the cursor option option for the aggregation query (ignored for < 2.6.0). Note the different syntax below: .exec() returns a cursor object, and no callback is necessary.

Example:

```
var cursor = Model.aggregate(..).cursor({ batchSize: 1000 }).exec();
cursor.each(function(error, doc) {
  // use doc
});
```

Aggregate.prototype.exec()

Parameters

- [callback] «Function»

Returns:

- «Promise»

Executes the aggregate pipeline on the currently bound Model.

Example:

```
aggregate.exec(callback);

// Because a promise is returned, the `callback` is optional.
var promise = aggregate.exec();
promise.then(..);
```

Aggregate.prototype.explain()

Parameters

- callback «Function»

Returns:

- «Promise»

Execute the aggregation with explain

Example:

```
Model.aggregate(..).explain(callback)
```

Aggregate.prototype.facet()

Parameters

- facet «Object» options

Returns:

- «Aggregate» this

Combines multiple aggregation pipelines.

Example:

```
Model.aggregate(...)
 .facet({
   books: [{ groupBy: '$author' }],
   price: [{ $bucketAuto: { groupBy: '$price', buckets: 2 } }]
 })
 .exec();

// Output: { books: [...], price: [{...}, {...}] }
```

Aggregate.prototype.graphLookup()

Parameters

- options «Object» to $graphLookup as described in the above link

Returns:

- «Aggregate»

Appends new custom $graphLookup operator(s) to this aggregate pipeline, performing a recursive search on a collection.

Note that graphLookup can only consume at most 100MB of memory, and does not allow disk use even if `{ allowDiskUse: true }` is specified.

Examples:

```
// Suppose we have a collection of courses, where a document might look like `{ _id: 0,
 aggregate.graphLookup({ from: 'courses', startWith: '$prerequisite', connectFromField:
```

Aggregate.prototype.group()

Parameters

- arg «Object» $group operator contents

Returns:

- «Aggregate»

Appends a new custom $group operator to this aggregate pipeline.

Examples:

```
aggregate.group({ _id: "$department" });
```

Aggregate.prototype.hint()

Parameters

- value «Object|String» a hint object or the index name

Sets the hint option for the aggregation query (ignored for < 3.6.0)

Example:

```
Model.aggregate(..).hint({ qty: 1, category: 1 } }).exec(callback)
```

Aggregate.prototype.limit()

Parameters

- num «Number» maximum number of records to pass to the next stage

Returns:

- «Aggregate»

Appends a new $limit operator to this aggregate pipeline.

Examples:

```
aggregate.limit(10);
```

Aggregate.prototype.lookup()

Parameters

- options «Object» to $lookup as described in the above link

Returns:

- «Aggregate»

Appends new custom $lookup operator(s) to this aggregate pipeline.

Examples:

```
aggregate.lookup({ from: 'users', localField: 'userId', foreignField: '_id', as: 'users'
```

Aggregate.prototype.match()

Parameters

- arg «Object» $match operator contents

Returns:

- «Aggregate»

Appends a new custom $match operator to this aggregate pipeline.

Examples:

```
aggregate.match({ department: { $in: [ "sales", "engineering" ] } });
```

Aggregate.prototype.model()

Parameters

- model «Model» the model to which the aggregate is to be bound

Returns:

- «Aggregate»

Binds this aggregate to a model.

Aggregate.prototype.near()

Parameters

- arg «Object»

Returns:

- «Aggregate»

Appends a new $geoNear operator to this aggregate pipeline.

NOTE:

MUST be used as the first operator in the pipeline.

Examples:

```
aggregate.near({
  near: [40.724, -73.997],
  distanceField: "dist.calculated", // required
  maxDistance: 0.008,
  query: { type: "public" },
  includeLocs: "dist.location",
  uniqueDocs: true,
  num: 5
});
```

Aggregate.prototype.option()

Parameters

- options «Object» keys to merge into current options
- number «[options.maxTimeMS]» limits the time this aggregation will run, see MongoDB docs on `maxTimeMS`
- boolean «[options.allowDiskUse]» if true, the MongoDB server will use the hard drive to store data during this aggregation
- object «[options.collation]» see `Aggregate.prototype.collation()`
- ClientSession «[options.session]» see `Aggregate.prototype.session()`

Returns:

- «Aggregate» this

Lets you set arbitrary options, for middleware or plugins.

Example:

```
var agg = Model.aggregate(..).option({ allowDiskUse: true }); // Set the `allowDiskUse`
agg.options; // `{ allowDiskUse: true }`
```

Aggregate.prototype.options

Contains options passed down to the aggregate command.

Supported options are

- `readPreference`
- `cursor`
- `explain`
- `allowDiskUse`
- `maxTimeMS`
- `bypassDocumentValidation`
- `raw`
- `promoteLongs`
- `promoteValues`
- `promoteBuffers`
- `collation`

- comment
- session

Aggregate.prototype.pipeline()

Returns:

- «Array»

Returns the current pipeline

Example:

```
MyModel.aggregate().match({ test: 1 }).pipeline(); // [{ $match: { test: 1 } }]
```

Aggregate.prototype.project()

Parameters

- arg «Object|String» field specification

Returns:

- «Aggregate»

Appends a new $project operator to this aggregate pipeline.

Mongoose query selection syntax is also supported.

Examples:

```
// include a, include b, exclude _id
aggregate.project("a b -_id");

// or you may use object notation, useful when
// you have keys already prefixed with a "-"
aggregate.project({a: 1, b: 1, _id: 0});

// reshaping documents
aggregate.project({
    newField: '$b.nested'
  , plusTen: { $add: ['$val', 10]}
  , sub: {
       name: '$a'
    }
```

```
})

// etc
aggregate.project({ salary_k: { $divide: [ "$salary", 1000 ] } });
```

Aggregate.prototype.read()

Parameters

- pref «String» one of the listed preference options or their aliases
- [tags] «Array» optional tags for this query

Returns:

- «Aggregate» this

Sets the readPreference option for the aggregation query.

Example:

```
Model.aggregate(..).read('primaryPreferred').exec(callback)
```

Aggregate.prototype.readConcern()

Parameters

- level «String» one of the listed read concern level or their aliases

Returns:

- «Aggregate» this

Sets the readConcern level for the aggregation query.

Example:

```
Model.aggregate(..).readConcern('majority').exec(callback)
```

Aggregate.prototype.redact()

Parameters

- expression «Object» redact options or conditional expression
- [thenExpr] «String|Object» true case for the condition
- [elseExpr] «String|Object» false case for the condition

Returns:

- «Aggregate» this

Appends a new $redact operator to this aggregate pipeline.

If 3 arguments are supplied, Mongoose will wrap them with if-then-else of $cond operator respectively If `thenExpr` or `elseExpr` is string, make sure it starts with $$, like `$$DESCEND`, `$$PRUNE` or `$$KEEP`.

Example:

```
Model.aggregate(...)
 .redact({
   $cond: {
     if: { $eq: [ '$level', 5 ] },
     then: '$$PRUNE',
     else: '$$DESCEND'
   }
 })
 .exec();

// $redact often comes with $cond operator, you can also use the following syntax provid
Model.aggregate(...)
 .redact({ $eq: [ '$level', 5 ] }, '$$PRUNE', '$$DESCEND')
 .exec();
```

Aggregate.prototype.replaceRoot()

Parameters

- the «String|Object» field or document which will become the new root document

Returns:

- «Aggregate»

Appends a new $replaceRoot operator to this aggregate pipeline.

Note that the `$replaceRoot` operator requires field strings to start with '$'. If you are passing in a string Mongoose will prepend '$' if the specified field doesn't start '$'. If you are passing

in an object the strings in your expression will not be altered.

Examples:

```
aggregate.replaceRoot("user");

aggregate.replaceRoot({ x: { $concat: ['$this', '$that'] } });
```

Aggregate.prototype.sample()

Parameters

- size «Number» number of random documents to pick

Returns:

- «Aggregate»

Appends new custom $sample operator(s) to this aggregate pipeline.

Examples:

```
aggregate.sample(3); // Add a pipeline that picks 3 random documents
```

Aggregate.prototype.session()

Parameters

- session «ClientSession»

Sets the session for this aggregation. Useful for transactions.

Example:

```
const session = await Model.startSession();
await Model.aggregate(..).session(session);
```

Aggregate.prototype.skip()

Parameters

- num «Number» number of records to skip before next stage

Returns:

- «Aggregate»

Appends a new $skip operator to this aggregate pipeline.

Examples:

```
aggregate.skip(10);
```

Aggregate.prototype.sort()

Parameters

- arg «Object|String»

Returns:

- «Aggregate» this

Appends a new $sort operator to this aggregate pipeline.

If an object is passed, values allowed are `asc`, `desc`, `ascending`, `descending`, `1`, and `-1`.

If a string is passed, it must be a space delimited list of path names. The sort order of each path is ascending unless the path name is prefixed with `-` which will be treated as descending.

Examples:

```
// these are equivalent
aggregate.sort({ field: 'asc', test: -1 });
aggregate.sort('field -test');
```

Aggregate.prototype.sortByCount()

Parameters

- arg «Object|String»

Returns:

- «Aggregate» this

Appends a new $sortByCount operator to this aggregate pipeline. Accepts either a string field name or a pipeline object.

Note that the `$sortByCount` operator requires the new root to start with '$'. Mongoose will prepend '$' if the specified field name doesn't start with '$'.

Examples:

```
aggregate.sortByCount('users');
aggregate.sortByCount({ $mergeObjects: [ "$employee", "$business" ] })
```

Aggregate.prototype.then()

Parameters

- [resolve] «Function» successCallback
- [reject] «Function» errorCallback

Returns:

- «Promise»

Provides promise for aggregate.

Example:

```
Model.aggregate(..).then(successCallback, errorCallback);
```

Aggregate.prototype.unwind()

Parameters

- fields «String» the field(s) to unwind

Returns:

- «Aggregate»

Appends new custom $unwind operator(s) to this aggregate pipeline.

Note that the `$unwind` operator requires the path name to start with '$'. Mongoose will prepend '$' if the specified field doesn't start '$'.

Examples:

```
aggregate.unwind("tags");
aggregate.unwind("a", "b", "c");
```

AggregationCursor

AggregationCursor()

Parameters

- agg «Aggregate»
- options «Object»

An AggregationCursor is a concurrency primitive for processing aggregation results one document at a time. It is analogous to QueryCursor.

An AggregationCursor fulfills the Node.js streams3 API, in addition to several other mechanisms for loading documents from MongoDB one at a time.

Creating an AggregationCursor executes the model's pre aggregate hooks, but **not** the model's post aggregate hooks.

Unless you're an advanced user, do **not** instantiate this class directly. Use `Aggregate#cursor()` instead.

AggregationCursor.prototype.addCursorFlag()

Parameters

- flag «String»
- value «Boolean»

Returns:

- «AggregationCursor» this

Adds a cursor flag. Useful for setting the `noCursorTimeout` and `tailable` flags.

AggregationCursor.prototype.close()

Parameters

- callback «Function»

Returns:

- «Promise»

Marks this cursor as closed. Will stop streaming and subsequent calls to `next()` will error.

AggregationCursor.prototype.eachAsync()

Parameters

- fn «Function»
- [options] «Object»
    - [options.parallel] «Number» the number of promises to execute in parallel. Defaults to 1.
- [callback] «Function» executed when all docs have been processed

Returns:

- «Promise»

Execute `fn` for every document in the cursor. If `fn` returns a promise, will wait for the promise to resolve before iterating on to the next one. Returns a promise that resolves when done.

AggregationCursor.prototype.map()

Parameters

- fn «Function»

Returns:

- «AggregationCursor»

Registers a transform function which subsequently maps documents retrieved via the streams interface or `.next()`

Example

```
// Map documents returned by `data` events
Thing.
  find({ name: /^hello/ }).
  cursor().
  map(function (doc) {
   doc.foo = "bar";
   return doc;
  })
  on('data', function(doc) { console.log(doc.foo); });

// Or map documents returned by `.next()`
var cursor = Thing.find({ name: /^hello/ }).
  cursor().
  map(function (doc) {
    doc.foo = "bar";
    return doc;
  });
cursor.next(function(error, doc) {
  console.log(doc.foo);
});
```

AggregationCursor.prototype.next()

Parameters

- callback «Function»

Returns:

- «Promise»

Get the next document from this cursor. Will return `null` when there are no documents left.

Schematype

SchemaType()

Parameters

- path «String»
- [options] «Object»
- [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.default()

Parameters

- val «Function|any» the default value

Returns:

- «defaultValue»

Sets a default value for this SchemaType.

Example:

```
var schema = new Schema({ n: { type: Number, default: 10 })
var M = db.model('M', schema)
var 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:
var schema = new Schema({ aNumber: { type: Number, default: 4.815162342 }})
var M = db.model('M', schema)
var m = new M;
console.log(m.aNumber) // 4.815162342

// default unique objects for Mixed types:
var 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:
var schema = new Schema({ mixed: Schema.Types.Mixed });
schema.path('mixed').default({});
var M = db.model('M', schema);
var m1 = new M;

m1.mixed.added = 1;
console.log(m1.mixed); // { added: 1 }
var 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
var s = new Schema({ born: { type: Date, get: dob })

// or by retreiving its SchemaType
var 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);
}


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

var 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, schematype) {
  if (schematype.options.required) {
    return schematype.path + ' is required';
  } else {
    return schematype.path + ' is not';
  }
}

var VirusSchema = new Schema({
```

```
  name: { type: String, required: true, get: inspector },
  taxonomy: { type: String, get: inspector }
})

var 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.index()

Parameters

- options «Object|Boolean|String»

Returns:

- «SchemaType» this

Declares the index options for this schematype.

Example:

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

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

// or with custom error message

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

// or with a function

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

// or with a function and a custom message
var 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

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

// with custom error messaging
```

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

// or make a path conditionally required based on a function
var isOver18 = function() { return this.age >= 18; };
Schema.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
var s = new Schema({ name: { type: String, set: capitalize }});

// or with the SchemaType
var 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();
}

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

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

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

// or
var 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@
```

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

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

var Virus = db.model('Virus', VirusSchema);
var 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:

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

SchemaType.prototype.text()

Parameters

- bool «Boolean»

Returns:

- «SchemaType» this

Declares a full text index.

Example:

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

SchemaType.prototype.unique()

Parameters

- bool «Boolean»

Returns:

- «SchemaType» this

Declares an unique index.

Example:

```
var s = new Schema({ name: { type: String, unique: true }});
Schema.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

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

// adding many validators at a time

var many = [
    { validator: validator, msg: 'uh oh' }
  , { validator: anotherValidator, msg: 'failed' }
```

```
]
new Schema({ name: { type: String, validate: many }});

// or utilizing SchemaType methods directly:

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

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.

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

var Product = conn.model('Product', yourSchema);
var 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);
```

Virtualtype

- VirtualType()
- VirtualType.prototype._applyDefaultGetters()
- VirtualType.prototype.applyGetters()
- VirtualType.prototype.applySetters()
- VirtualType.prototype.get()
- VirtualType.prototype.set()

VirtualType()

Parameters

- options «Object»
  - [options.ref] «string|function» if `ref` is not nullish, this becomes a populated virtual

- [options.localField] «string|function» the local field to populate on if this is a populated virtual.
- [options.foreignField] «string|function» the foreign field to populate on if this is a populated virtual.
- [options.justOne=false] «boolean» by default, a populated virtual is an array. If you set `justOne`, the populated virtual will be a single doc or `null`.
- [options.getters=false] «boolean» if you set this to `true`, Mongoose will call any custom getters you defined on this virtual

VirtualType constructor

This is what mongoose uses to define virtual attributes via `Schema.prototype.virtual`.

Example:

```
const fullname = schema.virtual('fullname');
fullname instanceof mongoose.VirtualType // true
```

VirtualType.prototype._applyDefaultGetters()

Parameters

- fn «Function»

Returns:

- «VirtualType» this

If no getters/getters, add a default

VirtualType.prototype.applyGetters()

Parameters

- value «Object»
- scope «Object»

Returns:

- «any» the value after applying all getters

Applies getters to `value` using optional `scope`.

VirtualType.prototype.applySetters()

Parameters

- value «Object»
- scope «Object»

Returns:

- «any» the value after applying all setters

Applies setters to `value` using optional `scope`.

VirtualType.prototype.get()

Parameters

- fn «Function»

Returns:

- «VirtualType» this

Defines a getter.

Example:

```
var virtual = schema.virtual('fullname');
virtual.get(function () {
  return this.name.first + ' ' + this.name.last;
});
```

VirtualType.prototype.set()

Parameters

- fn «Function»

Returns:

- «VirtualType» this

Defines a setter.

Example:

```
var virtual = schema.virtual('fullname');
virtual.set(function (v) {
  var parts = v.split(' ');
  this.name.first = parts[0];
  this.name.last = parts[1];
});
```

Error

- MongooseError.CastError
- MongooseError.DivergentArrayError
- MongooseError.DocumentNotFoundError
- MongooseError.MissingSchemaError
- MongooseError.OverwriteModelError
- MongooseError.ParallelSaveError
- MongooseError.ValidationError
- MongooseError.ValidatorError
- MongooseError.VersionError
- MongooseError.messages

MongooseError.CastError

An instance of this error class will be returned when mongoose failed to cast a value.

MongooseError.DivergentArrayError

An instance of this error will be returned if you used an array projection and then modified the array in an unsafe way.

MongooseError.DocumentNotFoundError

An instance of this error class will be returned when `save()` fails because the underlying document was not found. The constructor takes one parameter, the conditions that mongoose passed to `update()` when trying to update the document.

MongooseError.MissingSchemaError

Thrown when you try to access a model that has not been registered yet

MongooseError.OverwriteModelError

Thrown when a model with the given name was already registered on the connection. See the FAQ about `OverwriteModelError`.

MongooseError.ParallelSaveError

An instance of this error class will be returned when you call `save()` multiple times on the same document in parallel. See the FAQ for more information.

MongooseError.ValidationError

An instance of this error class will be returned when validation failed.

MongooseError.ValidatorError

A `ValidationError` has a hash of `errors` that contain individual `ValidatorError` instances

MongooseError.VersionError

An instance of this error class will be returned when you call `save()` after the document in the database was changed in a potentially unsafe way. See the `versionKey` option for more

information.

MongooseError.messages

The default built-in validator error messages.

Array

MongooseArray.prototype.$pop()

Pops the array atomically at most one time per document `save()`.

NOTE:

Calling this mulitple times on an array before saving sends the same command as calling it once. This update is implemented using the MongoDB $pop method which enforces this

restriction.

```
doc.array = [1,2,3];

 var popped = doc.array.$pop();
 console.log(popped); // 3
 console.log(doc.array); // [1,2]

 // no affect
 popped = doc.array.$pop();
 console.log(doc.array); // [1,2]

 doc.save(function (err) {
   if (err) return handleError(err);

   // we saved, now $pop works again
   popped = doc.array.$pop();
   console.log(popped); // 2
   console.log(doc.array); // [1]
 })
```

MongooseArray.prototype.$shift()

Atomically shifts the array at most one time per document `save()`.

NOTE:

Calling this mulitple times on an array before saving sends the same command as calling it once. This update is implemented using the MongoDB $pop method which enforces this restriction.

```
doc.array = [1,2,3];

 var shifted = doc.array.$shift();
 console.log(shifted); // 1
 console.log(doc.array); // [2,3]

 // no affect
 shifted = doc.array.$shift();
 console.log(doc.array); // [2,3]

 doc.save(function (err) {
   if (err) return handleError(err);

   // we saved, now $shift works again
   shifted = doc.array.$shift();
   console.log(shifted ); // 2
```

```
  console.log(doc.array); // [3]
})
```

MongooseArray.prototype.addToSet()

Parameters

- [args...] «any»

Returns:

- «Array» the values that were added

Adds values to the array if not already present.

Example:

```
console.log(doc.array) // [2,3,4]
var added = doc.array.addToSet(4,5);
console.log(doc.array) // [2,3,4,5]
console.log(added)     // [5]
```

MongooseArray.prototype.indexOf()

Parameters

- obj «Object» the item to look for

Returns:

- «Number»

Return the index of `obj` or `-1` if not found.

MongooseArray.prototype.inspect()

Helper for console.log

MongooseArray.prototype.nonAtomicPush()

Parameters

- [args...] «any»

Pushes items to the array non-atomically.

NOTE:

marks the entire array as modified, which if saved, will store it as a `$set` operation, potentially overwritting any changes that happen between when you retrieved the object and when you save it.

MongooseArray.prototype.pop()

Wraps `Array#pop` with proper change tracking.

Note:

marks the entire array as modified which will pass the entire thing to $set potentially overwritting any changes that happen between when you retrieved the object and when you save it.

MongooseArray.prototype.pull()

Parameters

- [args...] «any»

Pulls items from the array atomically. Equality is determined by casting the provided value to an embedded document and comparing using the `Document.equals()` function.

Examples:

```
doc.array.pull(ObjectId)
doc.array.pull({ _id: 'someId' })
doc.array.pull(36)
doc.array.pull('tag 1', 'tag 2')
```

To remove a document from a subdocument array we may pass an object with a matching `_id`.

```
doc.subdocs.push({ _id: 4815162342 })
doc.subdocs.pull({ _id: 4815162342 }) // removed
```

Or we may passing the _id directly and let mongoose take care of it.

```
doc.subdocs.push({ _id: 4815162342 })
doc.subdocs.pull(4815162342); // works
```

The first pull call will result in a atomic operation on the database, if pull is called repeatedly without saving the document, a $set operation is used on the complete array instead, overwriting possible changes that happened on the database in the meantime.

MongooseArray.prototype.push()

Parameters

- [args...] «Object»

Wraps `Array#push` with proper change tracking.

MongooseArray.prototype.remove()

Alias of pull

MongooseArray.prototype.set()

Returns:

- «Array» this

Sets the casted `val` at index `i` and marks the array modified.

Example:

```
// given documents based on the following
var Doc = mongoose.model('Doc', new Schema({ array: [Number] }));

var doc = new Doc({ array: [2,3,4] })

console.log(doc.array) // [2,3,4]
```

```
doc.array.set(1,"5");
console.log(doc.array); // [2,5,4] // properly cast to number
doc.save() // the change is saved

// VS not using array#set
doc.array[1] = "5";
console.log(doc.array); // [2,"5",4] // no casting
doc.save() // change is not saved
```

MongooseArray.prototype.shift()

Wraps `Array#shift` with proper change tracking.

Example:

```
doc.array = [2,3];
var res = doc.array.shift();
console.log(res) // 2
console.log(doc.array) // [3]
```

Note:

marks the entire array as modified, which if saved, will store it as a `$set` operation, potentially overwritting any changes that happen between when you retrieved the object and when you save it.

MongooseArray.prototype.sort()

Wraps `Array#sort` with proper change tracking.

NOTE:

marks the entire array as modified, which if saved, will store it as a `$set` operation, potentially overwritting any changes that happen between when you retrieved the object and when you save it.

MongooseArray.prototype.splice()

Wraps `Array#splice` with proper change tracking and casting.

Note:

marks the entire array as modified, which if saved, will store it as a `$set` operation, potentially overwritting any changes that happen between when you retrieved the object and when you save it.

MongooseArray.prototype.toObject()

Parameters

- options «Object»

Returns:

- «Array»

Returns a native js Array.

MongooseArray.prototype.unshift()

Wraps `Array#unshift` with proper change tracking.

Note:

marks the entire array as modified, which if saved, will store it as a `$set` operation, potentially overwritting any changes that happen between when you retrieved the object and when you save it.