Queries

A `Query` is what is returned when calling many `Model` methods. These `Query` objects provide a chaining api to specify search terms, cursor options, hints, and other behavior.

Query#where

Lets you specify query terms with sugar.

```
query.where(path [, val])
```

`path` is a valid document path. `val` is optional. Its useful to omit `val` when used in conjunction with other query methods. For example:

```
query
.where('name', 'Space Ghost')
.where('age').gte(21).lte(65)
.exec(callback)
```

In this case, `gte()` and `lte()` operate on the previous path if not explicitly passed. The above query results in the following query expression:

```
{ name: 'Space Ghost', age: { $gte: 21, $lte: 65 }}
```

Query#$where

Specifies the javascript function to run on MongoDB on each document scanned. See the MongoDB docs for details.

```
Model.find().$where(fn)
```

`fn` can be either a function or a string.

Query#or

Specifies the $or operator.

```
query.or(array);
```

`array` is an array of expressions.

```
query.or([{ color: 'blue' }, { color: 'red' }]);
```

Query#nor

Specifies the $nor operator.

```
query.nor(array);
```

`array` is an array of expressions.

```
query.nor([{ color: 'daffodil yellow' }, { color: 'atomic tangerine' }]);
```

Query#equals

Assigns `val` to the last path used in `where()`.

```
query.where(path).equals(val)

// same as
```

```
query.where(path, val);
```

Query#gt

Specifies a greater than expression.

```
query.gt(path, val);
```

`gt` is also one of the methods with extra chaining sugar: when only one argument is passed, it uses the path used the last call to `where()`.
Example:

```
Model.where('age').gt(21)
```

Results in:

```
{ age: { $gt: 21 }}
```

Query#gte

Specifies a greater than or equal to expression.

```
query.gte(path, val);
```

`gte` is also one of the methods with extra chaining sugar: when only one argument is passed, it uses the path used the last call to `where()`.
Example:

```
Model.where('age').gte(21)
```

Results in:

```
{ age: { $gte: 21 }}
```

Query#lt

Specifies a less than expression.

```
query.lt(path, val);
```

`lt` is also one of the methods with extra chaining sugar: when only one argument is passed, it uses the path used the last call to `where()`.
Example:

```
Model.where('age').lt(21)
```

Results in:

```
{ age: { $lt: 21 }}
```

Query#lte

Specifies a less than or equal to expression.

```
query.lte(path, val);
```

`lte` is also one of the methods with extra chaining sugar: when only one argument is passed, it uses the path used the last call to `where()`.
Example:

```
Model.where('age').lte(21)
```

Results in:

```
{ age: { $lte: 21 }}
```

Query#ne

Specifies the $ne operator.

```
query.ne(path, val);
```

These methods have extra sugar in that when only one argument is passed, the path in the last call to `where()` is used. Example:

```
query.where('_id').ne('4ecf92f31993a52c58e07f6a')
```

results in

```
{ _id: { $ne: ObjectId('4ecf92f31993a52c58e07f6a') }}
```

Query#in

Specifies the `$in` operator.

```
query.in(path, array)
```

This method has extra sugar in that when only one argument is passed, the path in the last call to `where()` is used.

```
query.where('tags').in(['game', 'fun', 'holiday'])
```

results in

```
{ tags: { $in: ['game', 'fun', 'holiday'] }}
```

Query#nin

Specifies the `$nin` operator.

```
query.nin(path, array)
```

This method has extra sugar in that when only one argument is passed, the path in the last call to `where()` is used.

```
query.where('games').nin(['boring', 'lame'])
```

results in

```
{ games: { $nin: ['boring', 'lame'] }}
```

Query#all

Specifies the `$all` operator.

```
query.all(path, array)
```

This method has extra sugar in that when only one argument is passed, the path in the last call to `where()` is used.

```
query.where('games').all(['fun', 'exhausting'])
```

results in

```
{ games: { $all: ['fun', 'exhausting'] }}
```

Query#regex

Specifies the `$regex` operator.

```
query.regex('name.first', /^a/i)
```

This method has extra sugar in that when only one argument is passed, the path in the last call to `where()` is used.

```
query.where('name.first').regex(/^a/i)
```

results in

```
{ 'name.first': { $regex: /^a/i }}
```

Query#size

Specifies the $size operator.

```
query.size(path, integer)
```

This method has extra sugar in that when only one argument is passed, the path in the last call to `where()` is used.

```
query.size('comments', 2)
query.where('comments').size(2)
```

both result in

```
{ comments: { $size: 2 }}
```

Query#mod

Specifies the $mod operator.

```
var array = [10, 1]
query.mod(path, array)

query.mod(path, 10, 1)

query.where(path).mod(10, 1)
```

all result in

```
{ path: { $mod: [10, 1] }}
```

Query#exists

Specifies the $exists operator.

```
query.exists(path, Boolean)
```

These methods have extra sugar in that when only one argument is passed, the path from the last call to `where()` is used.
Given the following query,

```
var query = Model.find();
```

the following queries

```
query.exists('occupation');
query.exists('occupation', true);
query.where('occupation').exists();
```

all result in

```
{ occupation: { $exists: true }}
```

Another example:

```
query.exists('occupation', false)
query.where('occupation').exists(false);
```

result in

```
{ occupation: { $exists: false }}
```

Query#elemMatch

Specifies the $elemMatch operator.

```
query.elemMatch(path, criteria)
query.where(path).elemMatch(criteria)
```

Functions can also be passed so you can further use query sugar to declare the expression:

```
query.where(path).elemMatch(function)
query.elemMatch(path, function)
```

In this case a `query` is passed as the only argument into the function.

```
query.where('comments').elemMatch(function (elem) {
  elem.where('author', 'bnoguchi')
  elem.where('votes').gte(5);
});
```

Results in

```
{ comments: { $elemMatch: { author: 'bnoguchi', votes: { $gte: 5 }}}}
```

Query#within

```
query.within.box
query.within.center
```

todo

Query#box

todo

Query#center

todo

Query#centerSphere

todo

Query#near

Specifies the `$near` operator.

```
var array = [10, 1]
query.near(path, array)

query.near(path, 10, 1)

query.where(path).$near(10, 1)
```

all result in

```
{ path: { $near: [10, 1] }}
```

Query#maxDistance

Specifies the `$maxDistance` operator.

```
query.where('checkin').near([40, -72]).maxDistance(1);
```

results in

```
{ checkin: { $near: [40, -72], $maxDistance: 1 }}
```

Query#select

Specifies which subset of fields you want to return.

```
query.select('title name'})
// only _id, title, and name fields will be populated in your docs
```

You can also use object syntax:

```
query.select({ age: 0, contact: 0 })
// return everything but age and contact
```

Query#only

deprecated

Specifies a subset of fields to return. This is like `select()` but this option only specifies which fields you want returned.

```
query.only('title name')
query.only('title', 'name')
query.only(['title', 'name'])
```

Query#exclude

deprecated

Specifies a subset of fields to return. This is like `select()` but this option only specifies which fields you DO NOT want returned.

```
query.exclude('title name')
query.exclude('title', 'name')
query.exclude(['title', 'name'])
```

Query#slice

Retrieve a sub-range of elements in an array with the $slice method.

```
query.slice(path, val)
```

`val` can be a Number:

```
query.where('tags').slice(5) // last 5 tags
query.where('tags').slice(-5) // first 5 tags
```

or an Array.

```
query.where('tags').slice([20, 10]) // skip 20, limit 10
query.where('tags').slice([-20, 10]) // 20 from the end, limit 10
```

`slice` is also one of the methods with extra chaining sugar: when only one argument is passed, it uses the path used the last call to `where()`. Example:

```
query.slice('tags', -5)
```

is the same as

```
query.where('tags').slice(-5)
```

Query#populate

Specifies the paths to be populated. See in-depth docs here.

Sorting

Query#sort

Sets the sort path and direction.

```
query.sort('path', 1)
query.sort('path', -1)
query.sort('path', 1, 'another.path', -1)
```

Query#asc

deprecated

Sorting sugar.

```
query.asc('path' [, paths])
```

Each string `path` argument will be added to the sort ascending clause.

```
query.asc('title', 'name');
```

Is the same as

```
query.sort('title', 1).sort('name', 1);
```

Query#desc

deprecated
Sorting sugar.

```
query.desc('path' [, paths])
```

Each string `path` argument will be added to the sort descending clause.

```
query.desc('title', 'name');
```

Is the same as

```
query.sort('title', -1, 'name', -1);
```

Options

Query#limit

The limit method specifies the max number of documents to return.

```
query.limit(20).skip(10)
```

Query#skip

The skip method specifies at which document the database should begin returning results.

```
query.skip(10).limit(20)
```

Query#maxscan

Limits the number of documents to scan.
query.maxscan(Number)

Query#snapshot

The snapshot method indicates the use of snapshot mode for the query.

```
query.snapshot(Boolean)
```

Query#batchSize

Sets the numer of documents to return per database query. For example, if we were qeurying for 10000 docs and streaming them to the client, we may want to limit the number of documents retrieved per cursor iteration to reduce memory consuption (all docs are held in memory during iteration). Setting `batchSize` to, say, 100, would mean that the cursor would be pulling only 100 documents at a time from MongoDB.

```
query.batchSize(Number)
```

Query#slaveOk

Sets the slaveOk option or `true` with no arguments.

```
query.slaveOk(Boolean)
query.slaveOk() // equivalent to query.slaveOk(true)
```

Query#tailable

Sets the tailable option or `true` with no arguments.

```
query.tailable(Boolean)
```

```
query.tailable() // equivalent to query.tailable(true)
```

Tailable queries can only be used on capped collections, can only return documents in their natural order, and never use indexes. Unless the cursor dies, a tailable QueryStream will remain open and receive documents as they are inserted into the collection, much like the Unix `tail -f` command.

```
var stream = Model.find().tailable().stream();

stream.on('data', function (doc) {
    // do stuff
});
```

Query#hint

Specifies the hint option for MongoDB.

```
query.hint(indexName)
```

If your schema has an index like

```
Thing.index({ name: 1, title: 1 })
```

and you wanted to tell MongoDB to use that index for your query (in the off chance that MongoDB was not able to figure our that it should use it) you can give MongoDB a hint like so:

```
query.hint({ name: 1, title: 1}).exec(callback)
```

Query#comment

Sets the comment option.

```
query.comment(String)
```

Executing

Query#find

```
query.find(criteria [, callback])
```

Query#findOne

Sends the findOne command to MongoDB.

```
query.findOne([callback])
```

Query#count

Sends the count command to MongoDB.

```
query.count(callback)
```

Query#distinct

Casts `field` and sends a distinct command to MongoDB.

```
query.distinct(field, callback)
```

Query#update

Casts the `doc` according to the model Schema and sends an update command to MongoDB.

```
query.update(doc, callback)
```

All paths passed that are not $atomic operations will become $set ops so we retain backwards compatibility.

```
query.update({..}, { title: 'remove words' }, ...)
```

becomes

```
query.update({..}, { $set: { title: 'remove words' }}, ...)
```

Query#remove

Casts the query, then sends the remove command to MongoDB.

```
query.remove(callback)
```

Query#exec

Executes the query passing the results to the optional `callback`.

```
query.exec([callback])
```

Typically used in chaining scenarios:

```
Model.findOne().where('points').$gt(1000).exec(function (err, doc) {
  if (err) ...
});
```

Query#each

deprecated

A streaming cursor interface.

2.4.0 introduces the query.stream() method which is a more 'node-like' way of streaming records and provides better cursor management. It is recommended that you use query.stream in place of query.each.

```
query.each(callback);
```

The `callback` is called repeatedly for each document found as its streamed. If an error occurs streaming stops.

```
query.each(function (err, user) {
  if (err) return res.end("aww, received an error. all done.");
  if (user) {
    res.write(user.name + '\n')
  } else {
    res.end("reached end of cursor. all done.");
  }
});
```

A third parameter may also be used in the callback which allows you to iterate the cursor manually.

```
query.each(function (err, user, next) {
  if (err) return res.end("aww, received an error. all done.");
  if (user) {
    res.write(user.name + '\n')
    doSomethingAsync(next);
  } else {
    res.end("reached end of cursor. all done.");
  }
});
```

Query#stream

Returns a QueryStream for the query.

```
Model.find({}).stream().pipe(writeStream)
```

See the QueryStream docs for details.

Google GROUPS

Email: Subscribe