Collections

Meteor stores data in *collections*. To get started, declare a collection with `new Mongo.Collection`.

{% apibox “Mongo.Collection” %}

Calling this function is analogous to declaring a model in a traditional ORM (Object-Relation Mapper)-centric framework. It sets up a *collection* (a storage space for records, or “documents”) that can be used to store a particular type of information, like users, posts, scores, todo items, or whatever matters to your application. Each document is a EJSON object. It includes an `_id` property whose value is unique in the collection, which Meteor will set when you first create the document.

```
// Common code on client and server declares a DDP-managed Mongo collection.
const Chatrooms = new Mongo.Collection('chatrooms');
const Messages = new Mongo.Collection('messages');
```

The function returns an object with methods to `insert` documents in the collection, `update` their properties, and `remove` them, and to `find` the documents in the collection that match arbitrary criteria. The way these methods work is compatible with the popular Mongo database API. The same database API works on both the client and the server (see below).

```
// Return an array of my messages.
const myMessages = Messages.find({ userId: Meteor.userId() }).fetch();

// Create a new message.
Messages.insert({ text: 'Hello, world!' });

// Mark my first message as important.
Messages.update(myMessages[0]._id, { $set: { important: true } });
```

If you pass a `name` when you create the collection, then you are declaring a persistent collection — one that is stored on the server and seen by all users. Client code and server code can both access the same collection using the same API.

Specifically, when you pass a `name`, here’s what happens:

- On the server (if you do not specify a `connection`), a collection with that name is created on a backend Mongo server. When you call methods on that collection on the server, they translate directly into normal Mongo operations (after checking that they match your access control rules).
- On the client (and on the server if you specify a `connection`), a Minimongo instance is created. Minimongo is essentially an in-memory, non-persistent implementation of Mongo in pure JavaScript. It serves as a local cache that stores just the subset of the database that this client is working with. Queries (`find`) on these collections are served directly out of this cache, without talking to the server.
- When you write to the database on the client (`insert`, `update`, `remove`), the command is executed locally immediately, and, simultaneously, it's sent to the server and executed there too. This happens via stubs, because writes are implemented as methods.

  When, on the server, you write to a collection which has a specified `connection` to another server, it sends the corresponding method to the other server and receives the changed values back from it over DDP. Unlike on the client, it does not execute the write locally first.

If you pass a name to a client-only collection, it will not be synchronized with the server and you need to populate the collection "manually" using the low-level publication interface (`added/changed/removed`). See `added` for more information.

If you pass `null` as the `name`, then you're creating a local collection. It's not synchronized anywhere; it's just a local scratchpad that supports Mongo-style `find`, `insert`, `update`, and `remove` operations. (On both the client and the server, this scratchpad is implemented using Minimongo.)

By default, Meteor automatically publishes every document in your collection to each connected client. To turn this behavior off, remove the `autopublish` package, in your terminal:

```
meteor remove autopublish
```

and instead call `Meteor.publish` to specify which parts of your collection should be published to which users.

```
// Create a collection called `Posts` and put a document in it. The document
// will be immediately visible in the local copy of the collection. It will be
// written to the server-side database a fraction of a second later, and a
// fraction of a second after that, it will be synchronized down to any other
// clients that are subscribed to a query that includes it (see
// `Meteor.subscribe` and `autopublish`).
const Posts = new Mongo.Collection('posts');
Posts.insert({ title: 'Hello world', body: 'First post' });
```

```
// Changes are visible immediately-no waiting for a round trip to the server.
assert(Posts.find().count() === 1);

// Create a temporary, local collection. It works just like any other collection
// but it doesn't send changes to the server, and it can't receive any data from
// subscriptions.
const Scratchpad = new Mongo.Collection;

for (let i = 0; i < 10; i += 1) {
  Scratchpad.insert({ number: i * 2 });
}

assert(Scratchpad.find({ number: { $lt: 9 } }).count() === 5);
```

Generally, you'll assign `Mongo.Collection` objects in your app to global variables. You can only create one `Mongo.Collection` object for each underlying Mongo collection.

If you specify a `transform` option to the `Collection` or any of its retrieval methods, documents are passed through the `transform` function before being returned or passed to callbacks. This allows you to add methods or otherwise modify the contents of your collection from their database representation. You can also specify `transform` on a particular `find`, `findOne`, `allow`, or `deny` call. Transform functions must return an object and they may not change the value of the document's `_id` field (though it's OK to leave it out).

```
// An animal class that takes a document in its constructor.
class Animal {
  constructor(doc) {
    _.extend(this, doc);
  }

  makeNoise() {
    console.log(this.sound);
  }
}

// Define a collection that uses `Animal` as its document.
const Animals = new Mongo.Collection('animals', {
  transform: (doc) => new Animal(doc)
});

// Create an animal and call its `makeNoise` method.
Animals.insert({ name: 'raptor', sound: 'roar' });
Animals.findOne({ name: 'raptor' }).makeNoise(); // Prints 'roar'
```

`transform` functions are not called reactively. If you want to add a dynamically changing attribute to an object, do it with a function that computes the value at the time it's called, not by computing the attribute at `transform` time.

{% pullquote warning %} In this release, Minimongo has some limitations:

- `$pull` in modifiers can only accept certain kinds of selectors.
- `findAndModify`, aggregate functions, and map/reduce aren't supported.

All of these will be addressed in a future release. For full Minimongo release notes, see packages/minimongo/NOTES in the repository. {% endpullquote %}

{% pullquote warning %} Minimongo doesn't currently have indexes. It's rare for this to be an issue, since it's unusual for a client to have enough data that an index is worthwhile. {% endpullquote %}

Read more about collections and how to use them in the Collections article in the Meteor Guide.

{% apibox "Mongo.Collection#find" %}

`find` returns a cursor. It does not immediately access the database or return documents. Cursors provide `fetch` to return all matching documents, `map` and `forEach` to iterate over all matching documents, and `observe` and `observeChanges` to register callbacks when the set of matching documents changes. Cursors also implement ES2015's iteration protocols.

{% pullquote warning %} Collection cursors are not query snapshots. If the database changes between calling `Collection.find` and fetching the results of the cursor, or while fetching results from the cursor, those changes may or may not appear in the result set. {% endpullquote %}

Cursors are a reactive data source. On the client, the first time you retrieve a cursor's documents with `fetch`, `map`, or `forEach` inside a reactive computation (eg, a template or `autorun`), Meteor will register a dependency on the underlying data. Any change to the collection that changes the documents in a cursor will trigger a recomputation. To disable this behavior, pass `{reactive: false}` as an option to `find`.

Note that when `fields` are specified, only changes to the included fields will trigger callbacks in `observe`, `observeChanges` and invalidations in reactive computations using this cursor. Careful use of `fields` allows for more fine-grained reactivity for computations that don't depend on an entire document.

On the client, there will be a period of time between when the page loads and when the published data arrives from the server during which your client-side collections will be empty.

{% apibox "Mongo.Collection#findOne" %}

Equivalent to `find(selector, options).fetch()[0]` with `options.limit = 1`.

{% apibox "Mongo.Collection#insert" %}

Add a document to the collection. A document is just an object, and its fields can contain any combination of EJSON-compatible datatypes (arrays, objects, numbers, strings, `null`, true, and false).

`insert` will generate a unique ID for the object you pass, insert it in the database, and return the ID. When `insert` is called from untrusted client code, it will be allowed only if passes any applicable `allow` and `deny` rules.

On the server, if you don't provide a callback, then `insert` blocks until the database acknowledges the write, or throws an exception if something went wrong. If you do provide a callback, `insert` still returns the ID immediately. Once the insert completes (or fails), the callback is called with error and result arguments. In an error case, `result` is undefined. If the insert is successful, `error` is undefined and `result` is the new document ID.

On the client, `insert` never blocks. If you do not provide a callback and the insert fails on the server, then Meteor will log a warning to the console. If you provide a callback, Meteor will call that function with `error` and `result` arguments. In an error case, `result` is undefined. If the insert is successful, `error` is undefined and `result` is the new document ID.

Example:

```
const groceriesId = Lists.insert({ name: 'Groceries' });

Items.insert({ list: groceriesId, name: 'Watercress' });
Items.insert({ list: groceriesId, name: 'Persimmons' });
```

{% apibox “Mongo.Collection#update” %}

Modify documents that match `selector` according to `modifier` (see modifier documentation).

The behavior of `update` differs depending on whether it is called by trusted or untrusted code. Trusted code includes server code and method code. Untrusted code includes client-side code such as event handlers and a browser's JavaScript console.

- Trusted code can modify multiple documents at once by setting `multi` to true, and can use an arbitrary Mongo selector to find the documents to modify. It bypasses any access control rules set up by `allow` and `deny`. The number of affected documents will be returned from the `update` call if you don't pass a callback.

- Untrusted code can only modify a single document at once, specified by its `_id`. The modification is allowed only after checking any applicable `allow` and `deny` rules. The number of affected documents will be returned to the callback. Untrusted code cannot perform upserts, except in insecure mode.

On the server, if you don't provide a callback, then `update` blocks until the database acknowledges the write, or throws an exception if something went wrong. If you do provide a callback, update returns immediately. Once the update completes, the callback is called with a single error argument in the case of failure, or a second argument indicating the number of affected documents if the update was successful.

On the client, update never blocks. If you do not provide a callback and the update fails on the server, then Meteor will log a warning to the console. If you provide a callback, Meteor will call that function with an error argument if there was an error, or a second argument indicating the number of affected documents if the update was successful.

Client example:

```
// When the 'give points' button in the admin dashboard is pressed, give 5
// points to the current player. The new score will be immediately visible on
// everyone's screens.
Template.adminDashboard.events({
  'click .give-points'() {
    Players.update(Session.get('currentPlayer'), {
      $inc: { score: 5 }
    });
  }
});
```

Server example:

```
// Give the 'Winner' badge to each user with a score greater than 10. If they
// are logged in and their badge list is visible on the screen, it will update
// automatically as they watch.
Meteor.methods({
  declareWinners() {
    Players.update({ score: { $gt: 10 } }, {
      $addToSet: { badges: 'Winner' }
    }, { multi: true });
  }
});
```

You can use update to perform a Mongo upsert by setting the upsert option to true. You can also use the upsert method to perform an upsert that returns the _id of the document that was inserted (if there was one) in addition to the number of affected documents.

{% apibox “Mongo.Collection#upsert” %}

Modify documents that match selector according to modifier, or insert a document if no documents were modified. upsert is the same as calling update with the upsert option set to true, except that the return value of upsert is an object that contain the keys numberAffected and insertedId. (update returns only the number of affected documents.)

{% apibox “Mongo.Collection#remove” %}

Find all of the documents that match `selector` and delete them from the collection.

The behavior of `remove` differs depending on whether it is called by trusted or untrusted code. Trusted code includes server code and method code. Untrusted code includes client-side code such as event handlers and a browser’s JavaScript console.

- Trusted code can use an arbitrary Mongo selector to find the documents to remove, and can remove more than one document at once by passing a selector that matches multiple documents. It bypasses any access control rules set up by `allow` and `deny`. The number of removed documents will be returned from `remove` if you don’t pass a callback.

  As a safety measure, if `selector` is omitted (or is `undefined`), no documents will be removed. Set `selector` to `{}` if you really want to remove all documents from your collection.

- Untrusted code can only remove a single document at a time, specified by its `_id`. The document is removed only after checking any applicable `allow` and `deny` rules. The number of removed documents will be returned to the callback.

On the server, if you don’t provide a callback, then `remove` blocks until the database acknowledges the write and then returns the number of removed documents, or throws an exception if something went wrong. If you do provide a callback, `remove` returns immediately. Once the remove completes, the callback is called with a single error argument in the case of failure, or a second argument indicating the number of removed documents if the remove was successful.

On the client, `remove` never blocks. If you do not provide a callback and the remove fails on the server, then Meteor will log a warning to the console. If you provide a callback, Meteor will call that function with an error argument if there was an error, or a second argument indicating the number of removed documents if the remove was successful.

Example (client):

```
// When the 'remove' button is clicked on a chat message, delete that message.
Template.chat.events({
  'click .remove'() {
    Messages.remove(this._id);
  }
});
```

Example (server):

```
// When the server starts, clear the log and delete all players with a karma of
// less than -2.
```

```
Meteor.startup(() => {
  if (Meteor.isServer) {
    Logs.remove({});
    Players.remove({ karma: { $lt: -2 } });
  }
});
```

{% apibox "Mongo.Collection#createIndex" %}

For efficient and performant queries you will sometimes need to define indexes other than the default `_id` field. You should add indexes to fields (or combinations of fields) you use to lookup documents in a collection. This is where `createIndex` comes into play. It takes in 2 objects. First is the key and index type specification (which field and how they should be indexed) and second are options like the index name. For details on how indexes work read the MongoDB documentation.

> Note that indexes only apply to server and MongoDB collection. They are not implemented for Minimongo at this time.

Example defining a simple index on Players collection in Meteor:

```
Players.createIndex({ userId: 1 }, { name: 'user reference on players' });
```

{% apibox "Mongo.Collection#allow" %}

{% pullquote warning %} While `allow` and `deny` make it easy to get started building an app, it's harder than it seems to write secure `allow` and `deny` rules. We recommend that developers avoid `allow` and `deny`, and switch directly to custom methods once they are ready to remove `insecure` mode from their app. See the Meteor Guide on security for more details. {% endpullquote %}

When a client calls `insert`, `update`, or `remove` on a collection, the collection's `allow` and `deny` callbacks are called on the server to determine if the write should be allowed. If at least one `allow` callback allows the write, and no `deny` callbacks deny the write, then the write is allowed to proceed.

These checks are run only when a client tries to write to the database directly, for example by calling `update` from inside an event handler. Server code is trusted and isn't subject to `allow` and `deny` restrictions. That includes methods that are called with `Meteor.call` — they are expected to do their own access checking rather than relying on `allow` and `deny`.

You can call `allow` as many times as you like, and each call can include any combination of `insert`, `update`, and `remove` functions. The functions should return `true` if they think the operation should be allowed. Otherwise they should return `false`, or nothing at all (`undefined`). In that case Meteor will continue searching through any other `allow` rules on the collection.

The available callbacks are:

{% dtdd name:"insert(userId, doc)" %} The user `userId` wants to insert the document `doc` into the collection. Return `true` if this should be allowed.

doc will contain the _id field if one was explicitly set by the client, or if there is an active transform. You can use this to prevent users from specifying arbitrary _id fields. {% enddtdd %}

{% dtdd name:"update(userId, doc, fieldNames, modifier)" %}

The user userId wants to update a document doc. (doc is the current version of the document from the database, without the proposed update.) Return true to permit the change.

fieldNames is an array of the (top-level) fields in doc that the client wants to modify, for example ['name', 'score'].

modifier is the raw Mongo modifier that the client wants to execute; for example, { $set: { 'name.first': 'Alice' }, $inc: { score: 1 } }.

Only Mongo modifiers are supported (operations like $set and $push). If the user tries to replace the entire document rather than use $-modifiers, the request will be denied without checking the allow functions.

{% enddtdd %}

{% dtdd name:"remove(userId, doc)" %}

The user userId wants to remove doc from the database. Return true to permit this.

{% enddtdd %}

When calling update or remove Meteor will by default fetch the entire document doc from the database. If you have large documents you may wish to fetch only the fields that are actually used by your functions. Accomplish this by setting fetch to an array of field names to retrieve.

Example:

```
// Create a collection where users can only modify documents that they own.
// Ownership is tracked by an `owner` field on each document. All documents must
// be owned by the user that created them and ownership can't be changed. Only a
// document's owner is allowed to delete it, and the `locked` attribute can be
// set on a document to prevent its accidental deletion.
const Posts = new Mongo.Collection('posts');

Posts.allow({
  insert(userId, doc) {
    // The user must be logged in and the document must be owned by the user.
    return userId && doc.owner === userId;
  },

  update(userId, doc, fields, modifier) {
    // Can only change your own documents.
    return doc.owner === userId;
  },

  remove(userId, doc) {
    // Can only remove your own documents.
    return doc.owner === userId;
  },

  fetch: ['owner']
});

Posts.deny({
  update(userId, doc, fields, modifier) {
    // Can't change owners.
    return _.contains(fields, 'owner');
  },

  remove(userId, doc) {
    // Can't remove locked documents.
    return doc.locked;
  },

  fetch: ['locked'] // No need to fetch `owner`
});
```

If you never set up any `allow` rules on a collection then all client writes to the collection will be denied, and it will only be possible to write to the collection from server-side code. In this case you will have to create a method for each possible write that clients are allowed to do. You'll then call these methods with `Meteor.call` rather than having the clients call `insert`, `update`, and `remove` directly on the collection.

Meteor also has a special "insecure mode" for quickly prototyping new applications. In insecure mode, if you haven't set up any `allow` or `deny` rules on a collection, then all users have full write access to the collection. This is the only effect of insecure mode. If you call `allow` or `deny` at all on a collection, even `Posts.allow({})`, then access is checked just like normal on that collection. **New Meteor projects start in insecure mode by default.** To turn it off just run in your terminal:

```
meteor remove insecure
```

{% apibox "Mongo.Collection#deny" %}

{% pullquote warning %} While `allow` and `deny` make it easy to get started building an app, it's harder than it seems to write secure `allow` and `deny` rules. We recommend that developers avoid `allow` and `deny`, and switch directly to custom methods once they are ready to remove `insecure` mode from their app. See the Meteor Guide on security for more details. {% endpullquote %}

This works just like `allow`, except it lets you make sure that certain writes are definitely denied, even if there is an `allow` rule that says that they should be permitted.

When a client tries to write to a collection, the Meteor server first checks the collection's `deny` rules. If none of them return true then it checks the collection's `allow` rules. Meteor allows the write only if no `deny` rules return `true` and at least one `allow` rule returns `true`.

{% apibox "Mongo.Collection#rawCollection" %}

The methods (like `update` or `insert`) you call on the resulting *raw* collection return promises and can be used outside of a Fiber.

{% apibox "Mongo.Collection#rawDatabase" %}

Cursors

To create a cursor, use `find`. To access the documents in a cursor, use `forEach`, `map`, `fetch`, or ES2015's iteration protocols.

{% apibox "Mongo.Cursor#forEach" %}

This interface is compatible with Array.forEach.

When called from a reactive computation, `forEach` registers dependencies on the matching documents.

Examples:

```
// Print the titles of the five top-scoring posts.
const topPosts = Posts.find({}, { sort: { score: -1 }, limit: 5 });
let count = 0;

topPosts.forEach((post) => {
  console.log(`Title of post ${count}: ${post.title}`);
  count += 1;
});
```

{% apibox "Mongo.Cursor#map" %}

This interface is compatible with Array.map.

When called from a reactive computation, `map` registers dependencies on the matching documents.

On the server, if `callback` yields, other calls to `callback` may occur while the first call is waiting. If strict sequential execution is necessary, use `forEach` instead.

{% apibox "Mongo.Cursor#fetch" %}

When called from a reactive computation, `fetch` registers dependencies on the matching documents.

{% apibox “Mongo.Cursor#count” %}

Unlike the other functions, `count` registers a dependency only on the number of matching documents. (Updates that just change or reorder the documents in the result set will not trigger a recomputation.)

{% apibox “Mongo.Cursor#observe” %}

Establishes a *live query* that invokes callbacks when the result of the query changes. The callbacks receive the entire contents of the document that was affected, as well as its old contents, if applicable. If you only need to receive the fields that changed, see `observeChanges`.

`callbacks` may have the following functions as properties:

added(document) or

addedAt(document, atIndex, before)

A new document `document` entered the result set. The new document appears at position `atIndex`. It is immediately before the document whose `_id` is `before`. `before` will be `null` if the new document is at the end of the results.

changed(newDocument, oldDocument) or

changedAt(newDocument, oldDocument, atIndex)

The contents of a document were previously `oldDocument` and are now `newDocument`. The position of the changed document is `atIndex`.

removed(oldDocument) or

removedAt(oldDocument, atIndex)

The document `oldDocument` is no longer in the result set. It used to be at position `atIndex`.

{% dtdd name:“movedTo(document, fromIndex, toIndex, before)” %} A document changed its position in the result set, from `fromIndex` to `toIndex` (which is before the document with id `before`). Its current contents is `document`. {% enddtdd %}

Use `added`, `changed`, and `removed` when you don’t care about the order of the documents in the result set. They are more efficient than `addedAt`, `changedAt`, and `removedAt`.

Before `observe` returns, `added` (or `addedAt`) will be called zero or more times to deliver the initial results of the query.

`observe` returns a live query handle, which is an object with a `stop` method. Call `stop` with no arguments to stop calling the callback functions and tear down the query. **The query will run forever until you call this.** If `observe` is called from a `Tracker.autorun` computation, it is automatically stopped when the computation is rerun or stopped. (If the cursor was created with the option `reactive` set to false, it will only deliver the initial results and will not call any further callbacks; it is not necessary to call `stop` on the handle.)

{% apibox "Mongo.Cursor#observeChanges" %}

Establishes a *live query* that invokes callbacks when the result of the query changes. In contrast to `observe`, `observeChanges` provides only the difference between the old and new result set, not the entire contents of the document that changed.

`callbacks` may have the following functions as properties:

added(id, fields) or

addedBefore(id, fields, before)

A new document entered the result set. It has the `id` and `fields` specified. `fields` contains all fields of the document excluding the `_id` field. The new document is before the document identified by `before`, or at the end if `before` is `null`.

{% dtdd name:"changed(id, fields)" %} The document identified by `id` has changed. `fields` contains the changed fields with their new values. If a field was removed from the document then it will be present in `fields` with a value of `undefined`. {% enddtdd %}

{% dtdd name:"movedBefore(id, before)" %} The document identified by `id` changed its position in the ordered result set, and now appears before the document identified by `before`. {% enddtdd %}

{% dtdd name:"removed(id)" %} The document identified by `id` was removed from the result set. {% enddtdd %}

`observeChanges` is significantly more efficient if you do not use `addedBefore` or `movedBefore`.

Before `observeChanges` returns, `added` (or `addedBefore`) will be called zero or more times to deliver the initial results of the query.

`observeChanges` returns a live query handle, which is an object with a `stop` method. Call `stop` with no arguments to stop calling the callback functions and tear down the query. **The query will run forever until you call this.** If `observeChanges` is called from a `Tracker.autorun` computation, it is automatically stopped when the computation is rerun or stopped. (If the cursor was created with the option `reactive` set to false, it will only deliver the initial results and will not call any further callbacks; it is not necessary to call `stop` on the handle.)

> Unlike `observe`, `observeChanges` does not provide absolute position information (that is, `atIndex` positions rather than `before` positions.) This is for efficiency.

Example:

```
// Keep track of how many administrators are online.
let count = 0;
const cursor = Users.find({ admin: true, onlineNow: true });

const handle = cursor.observeChanges({
  added(id, user) {
    count += 1;
    console.log(`${user.name} brings the total to ${count} admins.`);
  },

  removed() {
    count -= 1;
    console.log(`Lost one. We're now down to ${count} admins.`);
  }
});

// After five seconds, stop keeping the count.
setTimeout(() => handle.stop(), 5000);
```

{% apibox “Mongo.ObjectID” %}

`Mongo.ObjectID` follows the same API as the Node MongoDB driver `ObjectID` class. Note that you must use the `equals` method (or `EJSON.equals`) to compare them; the `===` operator will not work. If you are writing generic code that needs to deal with `_id` fields that may be either strings or `ObjectID`s, use `EJSON.equals` instead of `===` to compare them.

> `ObjectID` values created by Meteor will not have meaningful answers to their `getTimestamp` method, since Meteor currently constructs them fully randomly.

Mongo-Style Selectors

The simplest selectors are just a string or `Mongo.ObjectID`. These selectors match the document with that value in its `_id` field.

A slightly more complex form of selector is an object containing a set of keys that must match in a document:

```
// Matches all documents where `deleted` is false.
{ deleted: false }

// Matches all documents where the `name` and `cognomen` are as given.
{ name: 'Rhialto', cognomen: 'the Marvelous' }

// Matches every document.
{}
```

But they can also contain more complicated tests:

```
// Matches documents where `age` is greater than 18.
{ age: { $gt: 18 } }

// Matches documents where `tags` is an array containing 'popular'.
{ tags: 'popular' }

// Matches documents where `fruit` is one of three possibilities.
{ fruit: { $in: ['peach', 'plum', 'pear'] } }
```

See the complete documentation.

Mongo-Style Modifiers

A modifier is an object that describes how to update a document in place by changing some of its fields. Some examples:

```
// Set the `admin` property on the document to true.
{ $set: { admin: true } }

// Add 2 to the `votes` property and add 'Traz' to the end of the `supporters`
// array.
{ $inc: { votes: 2 }, $push: { supporters: 'Traz' } }
```

But if a modifier doesn't contain any $-operators, then it is instead interpreted as a literal document, and completely replaces whatever was previously in the database. (Literal document modifiers are not currently supported by validated updates.)

```
// Find the document with ID '123' and completely replace it.
Users.update({ _id: '123' }, { name: 'Alice', friends: ['Bob'] });
```

See the full list of modifiers.

Sort Specifiers

Sorts may be specified using your choice of several syntaxes:

```
// All of these do the same thing (sort in ascending order by key `a`, breaking
// ties in descending order of key `b`).
[['a', 'asc'], ['b', 'desc']]
['a', ['b', 'desc']]
{ a: 1, b: -1 }

// Sorted by `createdAt` descending.
Users.find({}, { sort: { createdAt: -1 } });

// Sorted by `createdAt` descending and by `name` ascending.
Users.find({}, { sort: [['createdAt', 'desc'], ['name', 'asc']] });
```

The last form will only work if your JavaScript implementation preserves the order of keys in objects. Most do, most of the time, but it's up to you to be sure.

For local collections you can pass a comparator function which receives two document objects, and returns -1 if the first document comes first in order, 1 if the second document comes first, or 0 if neither document comes before the other. This is a Minimongo extension to MongoDB.

Field Specifiers

Queries can specify a particular set of fields to include or exclude from the result object.

To exclude specific fields from the result objects, the field specifier is a dictionary whose keys are field names and whose values are `0`. All unspecified fields are included.

```
Users.find({}, { fields: { password: 0, hash: 0 } });
```

To include only specific fields in the result documents, use `1` as the value. The `_id` field is still included in the result.

```
Users.find({}, { fields: { firstname: 1, lastname: 1 } });
```

With one exception, it is not possible to mix inclusion and exclusion styles: the keys must either be all 1 or all 0. The exception is that you may specify `_id: 0` in an inclusion specifier, which will leave `_id` out of the result object as well. However, such field specifiers can not be used with `observeChanges`, `observe`, cursors returned from a publish function, or cursors used in `{% raw %}{{#each}}{% endraw %}` in a template. They may be used with `fetch`, `findOne`, `forEach`, and `map`.

Field operators such as `$` and `$elemMatch` are not available on the client side yet.

A more advanced example:

```
Users.insert({
  alterEgos: [
    { name: 'Kira', alliance: 'murderer' },
    { name: 'L', alliance: 'police' }
  ],
  name: 'Yagami Light'
});

Users.findOne({}, { fields: { 'alterEgos.name': 1, _id: 0 } });
// Returns { alterEgos: [{ name: 'Kira' }, { name: 'L' }] }
```

See the MongoDB docs for details of the nested field rules and array behavior.

Connecting to your database

When developing your application, Meteor starts a local MongoDB instance and automatically connects to it. In production, you must specify a `MONGO_URL` environment variable pointing at your database in the standard mongo connection string format.

> You can also set MONGO_URL in development if you want to connect to a different MongoDB instance.

If you want to use oplog tailing for livequeries, you should also set MONGO_OPLOG_URL (generally you'll need a special user with oplog access, but the detail can differ depending on how you host your MongoDB. Read more here).

> As of Meteor 1.4, you must ensure you set the replicaSet parameter on your METEOR_OPLOG_URL

Mongo Connection Options

MongoDB provides many connection options, usually the default works but in some cases you may want to pass additional options. You can do it in two ways:

Meteor settings

You can use your Meteor settings file to set the options in a property called options inside packages > mongo, these values will be provided as options for MongoDB in the connect method.

> this option was introduced in Meteor 1.10.2

For example, you may want to specify a certificate for your TLS connection (see the options here) then you could use these options:

```
"packages": {
  "mongo": {
    "options": {
      "tls": true,
      "tlsCAFileAsset": "certificate.pem"
    }
  }
}
```

Meteor will convert relative paths to absolute paths if the option name (key) ends with Asset, for this to work properly you need to place the files in the private folder in the root of your project. In the example Mongo connection would receive this:

```
"packages": {
  "mongo": {
    "options": {
      "tls": true,
      "tlsCAFile": "/absolute/path/certificate.pem"
    }
  }
}
```

See that the final option name (key) does not contain `Asset` in the end as expected by MongoDB.

This configuration is necessary in some MongoDB host providers to avoid this error `MongoNetworkError: failed to connect to server [sg-meteorappdb-32194.servers.mongodirector.com:27017] on first connect [Error: self signed certificate`.

Another way to avoid this error is to allow invalid certificates with this option:

```
"packages": {
  "mongo": {
    "options": {
      "tlsAllowInvalidCertificates": true
    }
  }
}
```

You can pass any MongoDB valid option, these are just examples using certificates configurations.

Mongo.setConnectionOptions

You can also call `Mongo.setConnectionOptions` to set the connection options but you need to call it before any other package using Mongo connections is initialized so you need to add this code in a package and add it above the other packages, like accounts-base in your `.meteor/packages` file.

> this option was introduced in Meteor 1.4