After reading this guide, you'll know:

1. The different types of MongoDB collections in Meteor, and how to use them.
2. How to define a schema for a collection to control its content.
3. What to consider when defining your collection's schema.
4. How to enforce the schema when writing to a collection.
5. How to carefully change the schema of your collection.
6. How to deal with associations between records.

MongoDB collections in Meteor

At its core, a web application offers its users a view into, and a way to modify, a persistent set of data. Whether managing a list of todos, or ordering a car to pick you up, you are interacting with a permanent but constantly changing data layer.

In Meteor, that data layer is typically stored in MongoDB. A set of related data in MongoDB is referred to as a "collection". In Meteor you access MongoDB through collections, making them the primary persistence mechanism for your app data.

However, collections are a lot more than a way to save and retrieve data. They also provide the core of the interactive, connected user experience that users expect from the best applications. Meteor makes this user experience easy to implement.

In this article, we'll look closely at how collections work in various places in the framework, and how to get the most out of them.

Server-side collections

When you create a collection on the server:

```
Todos = new Mongo.Collection('todos');
```

You are creating a collection within MongoDB, and an interface to that collection to be used on the server. It's a fairly straightforward layer on top of the underlying Node MongoDB driver, but with a synchronous API:

```
// This line won't complete until the insert is done
Todos.insert({_id: 'my-todo'});
// So this line will return something
const todo = Todos.findOne({_id: 'my-todo'});
// Look ma, no callbacks!
console.log(todo);
```

Client-side collections

On the client, when you write the same line:

```
Todos = new Mongo.Collection('todos');
```

It does something totally different!

On the client, there is no direct connection to the MongoDB database, and in fact a synchronous API to it is not possible (nor probably what you want). Instead, on the client, a collection is a client side *cache* of the database. This is

achieved thanks to the Minimongo library---an in-memory, all JS, implementation of the MongoDB API.

```
// This line is changing an in-memory Minimongo data structure
Todos.insert({_id: 'my-todo'});
// And this line is querying it
const todo = Todos.findOne({_id: 'my-todo'});
// So this happens right away!
console.log(todo);
```

The way that you move data from the server (and MongoDB-backed) collection into the client (in-memory) collection is the subject of the data loading article. Generally speaking, you *subscribe* to a *publication*, which pushes data from the server to the client. Usually, you can assume that the client contains an up-to-date copy of some subset of the full MongoDB collection.

To write data back to the server, you use a *Method*, the subject of the methods article.

Local collections

There is a third way to use a collection in Meteor. On the client or server, if you create a collection in one of these two ways:

```
SelectedTodos = new Mongo.Collection(null);
SelectedTodos = new Mongo.Collection('selectedtodos', {connection: null});
```

This creates a *local collection*. This is a Minimongo collection that has no database connection (ordinarily a collection would either be directly connected to the database on the server, or via a subscription on the client).

A local collection is a convenient way to use the full power of the Minimongo library for in-memory storage. For instance, you might use it instead of a simple array if you need to execute complex queries over your data. Or you may want to take advantage of its *reactivity* on the client to drive some UI in a way that feels natural in Meteor.

Defining a schema

Although MongoDB is a schema-less database, which allows maximum flexibility in data structuring, it is generally good practice to use a schema to constrain the contents of your collection to conform to a known format. If you don't, then you tend to end up needing to write defensive code to check and confirm the structure of your data as it *comes out* of the database, instead of when it *goes into* the database. As in most things, you tend to *read data more often than you write it*, and so it's usually easier, and less buggy to use a schema when writing.

In Meteor, the pre-eminent schema package is the npm simpl-schema package. It's an expressive, MongoDB based schema that's used to insert and update documents. Another alternative is jagi:astronomy which is a full Object Model (OM) layer offering schema definition, server/client side validators, object methods and event handlers.

Let's assume that we have a `Lists` collection. To define a schema for this collection using `simpl-schema`, you can create a new instance of the `SimpleSchema` class and attach it to the `Lists` object:

```
import SimpleSchema from 'simpl-schema';

Lists.schema = new SimpleSchema({
  name: {type: String},
  incompleteCount: {type: Number, defaultValue: 0},
```

```
  userId: {type: String, regEx: SimpleSchema.RegEx.Id, optional: true}
});
```

This example from the Todos app defines a schema with a few simple rules:

2. We specify that the `name` field of a list is required and must be a string.
3. We specify the `incompleteCount` is a number, which on insertion is set to `0` if not otherwise specified.
4. We specify that the `userId`, which is optional, must be a string that looks like the ID of a user document.

We attach the schema to the namespace of `Lists` directly, which allows us to check objects against this schema directly whenever we want, such as in a form or Method. In the next section we'll see how to use this schema automatically when writing to the collection.

You can see that with relatively little code we've managed to restrict the format of a list significantly. You can read more about more complex things that can be done with schemas in the Simple Schema docs.

Validating against a schema

Now we have a schema, how do we use it?

It's pretty straightforward to validate a document with a schema. We can write:

```
const list = {
  name: 'My list',
  incompleteCount: 3
};

Lists.schema.validate(list);
```

In this case, as the list is valid according to the schema, the `validate()` line will run without problems. If however, we wrote:

```
const list = {
  name: 'My list',
  incompleteCount: 3,
  madeUpField: 'this should not be here'
};

Lists.schema.validate(list);
```

Then the `validate()` call will throw a `ValidationError` which contains details about what is wrong with the `list` document.

The `ValidationError`

What is a `ValidationError`? It's a special error that is used in Meteor to indicate a user-input based error in modifying a collection. Typically, the details on a `ValidationError` are used to mark up a form with information about what inputs don't match the schema. In the methods article, we'll see more about how this works.

Designing your data schema

Now that you are familiar with the basic API of Simple Schema, it's worth considering a few of the constraints of the Meteor data system that can influence the design of your data schema. Although generally speaking you can build a Meteor data schema much like any MongoDB data schema, there are some important details to keep in mind.

The most important consideration is related to the way DDP, Meteor's data loading protocol, communicates documents over the wire. The key thing to realize is that DDP sends changes to documents at the level of top-level document *fields*. What this means is that if you have large and complex subfields on a document that change often, DDP can send unnecessary changes over the wire.

For instance, in "pure" MongoDB you might design the schema so that each list document had a field called `todos` which was an array of todo items:

```
Lists.schema = new SimpleSchema({
  name: {type: String},
  todos: {type: [Object]}
});
```

The issue with this schema is that due to the DDP behavior just mentioned, each change to *any* todo item in a list will require sending the *entire* set of todos for that list over the network. This is because DDP has no concept of "change the `text` field of the 3rd item in the field called `todos` ". It can only "change the field called `todos` to a totally new array".

Denormalization and multiple collections

The implication of the above is that we need to create more collections to contain sub-documents. In the case of the Todos application, we need both a `Lists` collection and a `Todos` collection to contain each list's todo items. Consequently we need to do some things that you'd typically associate with a SQL database, like using foreign keys (`todo.listId`) to associate one document with another.

In Meteor, it's often less of a problem doing this than it would be in a typical MongoDB application, as it's easy to publish overlapping sets of documents (we might need one set of users to render one screen of our app, and an intersecting set for another), which may stay on the client as we move around the application. So in that scenario there is an advantage to separating the subdocuments from the parent.

However, given that MongoDB prior to version 3.2 doesn't support queries over multiple collections ("joins"), we typically end up having to denormalize some data back onto the parent collection. Denormalization is the practice of storing the same piece of information in the database multiple times (as opposed to a non-redundant "normal" form). MongoDB is a database where denormalizing is encouraged, and thus optimized for this practice.

In the case of the Todos application, as we want to display the number of unfinished todos next to each list, we need to denormalize `list.incompleteTodoCount` . This is an inconvenience but typically reasonably easy to do as we'll see in the section on abstracting denormalizers below.

Another denormalization that this architecture sometimes requires can be from the parent document onto sub-documents. For instance, in Todos, as we enforce privacy of the todo lists via the `list.userId` attribute, but we publish the todos separately, it might make sense to denormalize `todo.userId` also. To do this, we'd need to be careful to take the `userId` from the list when creating the todo, and updating all relevant todos whenever a list's `userId` changed.

Designing for the future

An application, especially a web application, is rarely finished, and it's useful to consider potential future changes when designing your data schema. As in most things, it's rarely a good idea to add fields before you actually need

them (often what you anticipate doesn't actually end up happening, after all).

However, it's a good idea to think ahead to how the schema may change over time. For instance, you may have a list of strings on a document (perhaps a set of tags). Although it's tempting to leave them as a subfield on the document (assuming they don't change much), if there's a good chance that they'll end up becoming more complicated in the future (perhaps tags will have a creator, or subtags later on?), then it might be easier in the long run to make a separate collection from the beginning.

The amount of foresight you bake into your schema design will depend on your app's individual constraints, and will need to be a judgement call on your part.

Using schemas on write

Although there are a variety of ways that you can run data through a Simple Schema before sending it to your collection (for instance you could check a schema in every method call), the simplest and most reliable is to use the `aldeed:collection2` package to run every mutator (`insert/update/upsert` call) through the schema.

To do so, we use `attachSchema()` :

```
Lists.attachSchema(Lists.schema);
```

What this means is that now every time we call `Lists.insert()` , `Lists.update()` , `Lists.upsert()` , first our document or modifier will be automatically checked against the schema (in subtly different ways depending on the exact mutator).

`defaultValue` and data cleaning

One thing that Collection2 does is "clean" the data before sending it to the database. This includes but is not limited to:

1. Coercing types - converting strings to numbers
2. Removing attributes not in the schema
3. Assigning default values based on the `defaultValue` in the schema definition

However, sometimes it's useful to do more complex initialization to documents before inserting them into collections. For instance, in the Todos app, we want to set the name of new lists to be `List X` where `X` is the next available unique letter.

To do so, we can subclass `Mongo.Collection` and write our own `insert()` method:

```
class ListsCollection extends Mongo.Collection {
  insert(list, callback) {
    if (!list.name) {
      let nextLetter = 'A';
      list.name = `List ${nextLetter}`;

      while (!!this.findOne({name: list.name})) {
        // not going to be too smart here, can't go past Z
        nextLetter = String.fromCharCode(nextLetter.charCodeAt(0) + 1);
        list.name = `List ${nextLetter}`;
      }
    }
```

```
    // Call the original `insert` method, which will validate
    // against the schema
    return super.insert(list, callback);
  }
}

Lists = new ListsCollection('lists');
```

Hooks on insert/update/remove

The technique above can also be used to provide a location to "hook" extra functionality into the collection. For instance, when removing a list, we *always* want to remove all of its todos at the same time.

We can use a subclass for this case as well, overriding the `remove()` method:

```
class ListsCollection extends Mongo.Collection {
  // ...
  remove(selector, callback) {
    Package.todos.Todos.remove({listId: selector});
    return super.remove(selector, callback);
  }
}
```

This technique has a few disadvantages:

1. Mutators can get very long when you want to hook in multiple times.
2. Sometimes a single piece of functionality can be spread over multiple mutators.
3. It can be a challenge to write a hook in a completely general way (that covers every possible selector and modifier), and it may not be necessary for your application (because perhaps you only ever call that mutator in one way).

A way to deal with points 1. and 2. is to separate out the set of hooks into their own module, and use the mutator as a point to call out to that module in a sensible way. We'll see an example of that below.

Point 3. can usually be resolved by placing the hook in the *Method* that calls the mutator, rather than the hook itself. Although this is an imperfect compromise (as we need to be careful if we ever add another Method that calls that mutator in the future), it is better than writing a bunch of code that is never actually called (which is guaranteed to not work!), or giving the impression that your hook is more general that it actually is.

Abstracting denormalizers

Denormalization may need to happen on various mutators of several collections. Therefore, it's sensible to define the denormalization logic in one place, and hook it into each mutator with one line of code. The advantage of this approach is that the denormalization logic is one place rather than spread over many files, but you can still examine the code for each collection and fully understand what happens on each update.

In the Todos example app, we build a `incompleteCountDenormalizer` to abstract the counting of incomplete todos on the lists. This code needs to run whenever a todo item is inserted, updated (checked or unchecked), or removed. The code looks like:

```
const incompleteCountDenormalizer = {
  _updateList(listId) {
```

```
    // Recalculate the correct incomplete count direct from MongoDB
    const incompleteCount = Todos.find({
      listId,
      checked: false
    }).count();

    Lists.update(listId, {$set: {incompleteCount}});
  },
  afterInsertTodo(todo) {
    this._updateList(todo.listId);
  },
  afterUpdateTodo(selector, modifier) {
    // We only support very limited operations on todos
    check(modifier, {$set: Object});

    // We can only deal with $set modifiers, but that's all we do in this app
    if (_.has(modifier.$set, 'checked')) {
      Todos.find(selector, {fields: {listId: 1}}).forEach(todo => {
        this._updateList(todo.listId);
      });
    }
  },
  // Here we need to take the list of todos being removed, selected *before* the
update
  // because otherwise we can't figure out the relevant list id(s) (if the todo has
been deleted)
  afterRemoveTodos(todos) {
    todos.forEach(todo => this._updateList(todo.listId));
  }
};
```

We are then able to wire in the denormalizer into the mutations of the `Todos` collection like so:

```
class TodosCollection extends Mongo.Collection {
  insert(doc, callback) {
    doc.createdAt = doc.createdAt || new Date();
    const result = super.insert(doc, callback);
    incompleteCountDenormalizer.afterInsertTodo(doc);
    return result;
  }
}
```

Note that we only handled the mutators we actually use in the application---we don't deal with all possible ways the todo count on a list could change. For example, if you changed the `listId` on a todo item, it would need to change the `incompleteCount` of *two* lists. However, since our application doesn't do this, we don't handle it in the denormalizer.

Dealing with every possible MongoDB operator is difficult to get right, as MongoDB has a rich modifier language. Instead we focus on dealing with the modifiers we know we'll see in our app. If this gets too tricky, then moving the hooks for the logic into the Methods that actually make the relevant modifications could be sensible (although you need to be diligent to ensure you do it in *all* the relevant places, both now and as the app changes in the future).

It could make sense for packages to exist to completely abstract some common denormalization techniques and actually attempt to deal with all possible modifications. If you write such a package, please let us know!

Migrating to a new schema

As we discussed above, trying to predict all future requirements of your data schema ahead of time is impossible. Inevitably, as a project matures, there will come a time when you need to change the schema of the database. You need to be careful about how you make the migration to the new schema to make sure your app works smoothly during and after the migration.

Writing migrations

A useful package for writing migrations is `percolate:migrations`, which provides a nice framework for switching between different versions of your schema.

Suppose, as an example, that we wanted to add a `list.todoCount` field, and ensure that it was set for all existing lists. Then we might write the following in server-only code (e.g. `/server/migrations.js`):

```
Migrations.add({
  version: 1,
  up() {
    Lists.find({todoCount: {$exists: false}}).forEach(list => {
      const todoCount = Todos.find({listId: list._id}).count();
      Lists.update(list._id, {$set: {todoCount}});
    });
  },
  down() {
    Lists.update({}, {$unset: {todoCount: true}}, {multi: true});
  }
});
```

This migration, which is sequenced to be the first migration to run over the database, will, when called, bring each list up to date with the current todo count.

To find out more about the API of the Migrations package, refer to its documentation.

Bulk changes

If your migration needs to change a lot of data, and especially if you need to stop your app server while it's running, it may be a good idea to use a MongoDB Bulk Operation.

The advantage of a bulk operation is that it only requires a single round trip to MongoDB for the write, which usually means it is a *lot* faster. The downside is that if your migration is complex (which it usually is if you can't do an `.update(.., .., {multi: true})`), it can take a significant amount of time to prepare the bulk update.

What this means is if users are accessing the site whilst the update is being prepared, it will likely go out of service! Also, a bulk update will lock the entire collection while it is being applied, which can cause a significant blip in your user experience if it takes a while. For these reason, you often need to stop your server and let your users know you are performing maintenance while the update is happening.

We could write our above migration like so (note that you must be on MongoDB 2.6 or later for the bulk update operations to exist). We can access the native MongoDB API via `Collection#rawCollection()`:

```
Migrations.add({
  version: 1,
  up() {
    // This is how to get access to the raw MongoDB node collection that the Meteor
server collection wraps
    const batch = Lists.rawCollection().initializeUnorderedBulkOp();

    //Mongo throws an error if we execute a batch operation without actual
operations, e.g. when Lists was empty.
    let hasUpdates = false;
    Lists.find({todoCount: {$exists: false}}).forEach(list => {
      const todoCount = Todos.find({listId: list._id}).count();
      // We have to use pure MongoDB syntax here, thus the `{_id: X}`
      batch.find({_id: list._id}).updateOne({$set: {todoCount}});
      hasUpdates = true;
    });

    if(hasUpdates){
      // We need to wrap the async function to get a synchronous API that migrations
expects
      const execute = Meteor.wrapAsync(batch.execute, batch);
      return execute();
    }

    return true;
  },
  down() {
    Lists.update({}, {$unset: {todoCount: true}}, {multi: true});
  }
});
```

Note that we could make this migration faster by using an Aggregation to gather the initial set of todo counts.

Running migrations

To run a migration against your development database, it's easiest to use the Meteor shell:

```
// After running `meteor shell` on the command line:
Migrations.migrateTo('latest');
```

If the migration logs anything to the console, you'll see it in the terminal window that is running the Meteor server.

To run a migration against your production database, run your app locally in production mode (with production settings and environment variables, including database settings), and use the Meteor shell in the same way. What this does is run the `up()` function of all outstanding migrations, against your production database. In our case, it should ensure all lists have a `todoCount` field set.

A good way to do the above is to spin up a virtual machine close to your database that has Meteor installed and SSH access (a special EC2 instance that you start and stop for the purpose is a reasonable option), and running the command after shelling into it. That way any latencies between your machine and the database will be eliminated, but you still can be very careful about how the migration is run.

Note that you should always make a database backup before running any migration!

Breaking schema changes

Sometimes when we change the schema of an application, we do so in a breaking way -- so that the old schema doesn't work properly with the new code base. For instance, if we had some UI code that heavily relied on all lists having a `todoCount` set, there would be a period, before the migration runs, in which the UI of our app would be broken after we deployed.

The simple way to work around the problem is to take the application down for the period in between deployment and completing the migration. This is far from ideal, especially considering some migrations can take hours to run (although using Bulk Updates probably helps a lot here).

A better approach is a multi-stage deployment. The basic idea is that:

1. Deploy a version of your application that can handle both the old and the new schema. In our case, it'd be code that doesn't expect the `todoCount` to be there, but which correctly updates it when new todos are created.
2. Run the migration. At this point you should be confident that all lists have a `todoCount`.
3. Deploy the new code that relies on the new schema and no longer knows how to deal with the old schema. Now we are safe to rely on `list.todoCount` in our UI.

Another thing to be aware of, especially with such multi-stage deploys, is that being prepared to rollback is important! For this reason, the migrations package allows you to specify a `down()` function and call `Migrations.migrateTo(x)` to migrate *back* to version `x`.

So if we wanted to reverse our migration above, we'd run

```
// The "0" migration is the unmigrated (before the first migration) state
Migrations.migrateTo(0);
```

If you find you need to roll your code version back, you'll need to be careful about the data, and step carefully through your deployment steps in reverse.

Caveats

Some aspects of the migration strategy outlined above are possibly not the most ideal way to do things (although perhaps appropriate in many situations). Here are some other things to be aware of:

1. Usually it is better to not rely on your application code in migrations (because the application will change over time, and the migrations should not). For instance, having your migrations pass through your Collection2 collections (and thus check schemas, set autovalues etc) is likely to break them over time as your schemas change over time.

One way to avoid this problem is to not run old migrations on your database. This is a little bit limiting but can be made to work.

2. Running the migration on your local machine will probably make it take a lot longer as your machine isn't as close to the production database as it could be.

Deploying a special "migration application" to the same hardware as your real application is probably the best way to solve the above issues. It'd be amazing if such an application kept track of which migrations ran when, with logs and provided a UI to examine and run them. Perhaps a boilerplate application to do so could be built (if you do so, please let us know and we'll link to it here!).

Associations between collections

As we discussed earlier, it's very common in Meteor applications to have associations between documents in different collections. Consequently, it's also very common to need to write queries fetching related documents once you have a document you are interested in (for instance all the todos that are in a single list).

To make this easier, we can attach functions to the prototype of the documents that belong to a given collection, to give us "methods" on the documents (in the object oriented sense). We can then use these methods to create new queries to find related documents.

To make things even easier, we can use the `cultofcoders:grapher` package to associate collections and fetch their relations. For example:

```
// Configure how collections relate to each other
Todos.addLinks({
  list: {
    type: 'one',
    field: 'listId',
    collection: Lists
  }
});

Lists.addLinks({
  todos: {
    collection: Todos,
    inversedBy: 'list' // This represents the name of the link we defined in Todos
  }
});
```

This allows us to properly fetch a list along with its todos:

```
// With Grapher you must always specify the fields you want
const listsAndTodos = Lists.createQuery({
  name: 1,
  todos: {
      text: 1
  }
}).fetch();
```

`listsAndTodos` will look like this:

```
[
  {
    name: 'My List',
    todos: [
      {text: 'Do something'}
    ],
  }
]
```

Grapher supports isomorphic queries (reactive and non-reactive), has built-in security features, works with many types of relationships, and more. Refer to the Grapher documentation for more details.

Collection helpers

We can use the `dburles:collection-helpers` package to easily attach such methods (or "helpers") to documents. For instance:

```
Lists.helpers({
  // A list is considered to be private if it has a userId set
  isPrivate() {
    return !!this.userId;
  }
});
```

Once we've attached this helper to the `Lists` collection, every time we fetch a list from the database (on the client or server), it will have a `.isPrivate()` function available:

```
const list = Lists.findOne();
if (list.isPrivate()) {
  console.log('The first list is private!');
}
```

Association helpers

Now we can attach helpers to documents, we can define a helper that fetches related documents

```
Lists.helpers({
  todos() {
    return Todos.find({listId: this._id}, {sort: {createdAt: -1}});
  }
});
```

Now we can find all the todos for a list:

```
const list = Lists.findOne();
console.log(`The first list has ${list.todos().count()} todos`);
```