March 18, 2014

In Meteor 0.7, we reworked how Meteor gets real-time database updates from MongoDB.

The key method that Meteor adds to the MongoDB API which makes it into a real-time database API is `cursor.observeChanges`. `observeChanges` run an arbitrary MongoDB query, returns its results via `added` callbacks, and then **continues to watch the query** and calls additional callbacks as its results change.

Most users don't use `observeChanges` directly, but whenever you return a cursor from a publish function, Meteor calls `observeChanges` on that cursor and provides it with callbacks which publish the query's changes to clients.

Previous versions of Meteor only had one strategy for implementing `observeChanges`: the "poll-and-diff" algorithm, implemented by the `PollingObserveDriver` class. This approach re-runs the query frequently and calculates the difference between each set of results. This code is simple and has historically contained very few bugs. But the cost of the `PollingObserveDriver` is proportional to the poll frequency and to the size of each query result, and the time from database change to callback invocation depends on whether the write originated in the same Meteor server process (very fast) or in another process (up to 10 seconds).

Starting with Meteor 0.7.0, Meteor can use an additional strategy to implemnt `observeChanges`: **oplog tailing**, implemented by the `OplogObserveDriver` class. Meteor now knows how to read the MongoDB "operations log" --- a special collection that records all the write operations as they are applied to your database. This means changes to the database can be instantly noticed and reflected in Meteor, whether they originated from Meteor or from an external database client. Oplog tailing has different performance characteristics than "poll-and-diff" which are superior in many cases.

`OplogObserveDriver` needs to understand the meaning of MongoDB selectors, field specifiers, modifiers, and sort specifiers at a much deeper level than `PollingObserveDriver`. This is because it actually needs to understand how write operations that it sees in the oplog interact with queries, instead of just relying on the MongoDB server to repeatedly execute the query. To deal with these structures, `OplogObserveDriver` uses Meteor's implementation of the MongoDB query engine, Minimongo, which Meteor also uses as its client-side local database cache.

As of Meteor 0.7.2, we use `OplogObserveDriver` for most queries. There are a few types of queries that still use `PollingObserveDriver`:

- Selectors containing geospatial operators, the `$where` operator, or any operator not supported by Minimongo (such as `$text`)
- Queries specifying the `skip` option
- Queries specifying the `limit` option without a `sort` specifier or with a sort based on `$natural` order
- Field specifiers and sort specifiers with `$` operators such as `$slice` or `$natural`
- Calls to `observeChanges` using "ordered" callbacks `addedBefore` and `movedBefore`. (The implicit call to `observeChanges` which occurs when you return a cursor from a publish function does not use the ordered callbacks.)

Oplog tailing is automatically enabled in development mode with `meteor run`, and can be enabled in production with the `MONGO_OPLOG_URL` environment variable.

`OplogObserveDriver` in production

To use oplog tailing in your production Meteor app, your MongoDB servers must be configured as a replica set; a single-`mongod` database has no oplog. Your cluster may not use Mongo sharding.

Once you have set up your replica set, you need to provide read access to the oplog to a user; we recommend creating a special user (say, `oplogger`) that is used only for this purpose. Note that the oplog is shared between **all databases** served by your replica set of `mongod` processes; if you are sharing your cluster with unrelated apps, the `oplogger` user will be able to see **all changes to all databases** in your cluster. You will need Mongo administrator credentials to create this user.

Log in to the `admin` database with the Mongo shell using your administrator credentials. (You must connect to the current primary in your replica set; if the prompt says `SECONDARY` instead of `PRIMARY`, type `db.isMaster()` in the `mongo` shell and try again connecting to the server listed under `primary`.)

```
$ mongo -u YourExistingAdminUserName -p YourExistingAdminPassword mongo-server-
1.example.com/admin
```

Now run the following command to make an `oplogger` user with the ability to read collections in the `local` database.

If you are using Mongo 2.6 (though see the "Note for Mongo 2.6 users" below if you are running Meteor 1.0.3.2 or earlier):

```
cluster:PRIMARY> db.createUser({user: "oplogger", pwd: "PasswordForOplogger", roles:
[{role: "read", db: "local"}]})
```

If you are using Mongo 2.4:

```
cluster:PRIMARY> db.addUser({user: "oplogger", pwd: "PasswordForOplogger", roles: [],
otherDBRoles: {local: ["read"]}})
```

Then, when running your bundled Meteor app, set the `MONGO_OPLOG_URL` environment variable:

```
MONGO_OPLOG_URL=mongodb://oplogger:PasswordForOplogger@mongo-server-
1.example.com,mongo-server-2.example.com,mongo-server-3.example.com/local?
authSource=admin&replicaSet=replicaSetName
```

(You can find the name of your replica set by running `rs.config()._id` in the mongo console).

(You may be used to running `db.createUser` (or `db.addUser`) inside the actual database that you want the new user to be able to access (in this case, `local`), instead of running it in `admin` and using the `authSource` flag to specify that you want to authenticate against `admin`. However, this doesn't work with the special case of the `local` database. Mongo 2.6 specifically prevents you from creating users in the `local` database, and while Mongo 2.4 would let you do it, you would find that you need to run `db.addUser` separately against each database replica (and risking ending up with different passwords on each), because the `local` database is not itself replicated across servers.)

(Note for Mongo 2.6 users: releases of Meteor prior to 1.0.4 were not tested with Mongo 2.6, and you will need to set up a custom user-defined role that can access the `system.replset` collection in order to use these versions of Meteor with Mongo 2.6. See [this comment by @rwillmer] for instructions. Meteor 1.0.4 does not require special permissions on `system.replset` and the simpler `createUser` call above will work.)

`OplogObserveDriver` is currently not supported for apps deployed with `meteor deploy`. Galaxy (the in-progress replacement for the `meteor deploy` servers which currently hosts some of Meteor Development Group's sites) will support `OplogObserveDriver`.

`OplogObserveDriver` during development

Starting with Meteor 0.7.0, whenever you run your local development server, the `meteor` tool will set up your local development database as a single-member replica set and automatically enable `OplogObserveDriver`.

Our goal is for `OplogObserveDriver` to be indistinguishable from `PollingObserveDriver` (except that it should notice database changes faster). However, if you do find a bug in `OplogObserveDriver` that affects your app's development, then (after reporting it), you can disable `OplogObserveDriver` as a workaround.

There are several ways to do so. For a specific query, you can specify the undocumented option `_disableOplog` to the find call:

```
Meteor.publish("comments", function (postId) {
  return Comments.find({post: postId}, {_disableOplog: true, fields: {secret: 0}});
});
```

Or to entirely disable `OplogObserveDriver` for your app, run `$ meteor add disable-oplog`. (And please let us know why!)

When running Meteor's internal tests with `meteor test-packages`, pass the `--disable-oplog` flag to disable `OplogObserveDriver`. (We do this during our pre-release QA process, to ensure that our tests pass with both observe drivers.)

How to tell if your queries are using `OplogObserveDriver`

For now, we only have a crude way to tell how many `observeChanges` calls are using `OplogObserveDriver`, and not which calls they are.

This uses the `facts` package, an internal Meteor package that exposes real-time metrics for the current Meteor server. In your app, run `$ meteor add facts`, and add the `{{> serverFacts}}` template to your app. If you are using the `autopublish` package, Meteor will automatically publish all metrics to all users. If you are not using `autopublish`, you will have to tell Meteor which users can see your metrics by calling `Facts.setUserIdFilter` in server code; for example:

```
Facts.setUserIdFilter(function (userId) {
  var user = Meteor.users.findOne(userId);
  return user && user.admin;
});
```

(When running your app locally, `Facts.setUserIdFilter(function () { return true; });` may be good enough!)

Now look at your app. The `facts` template will render a variety of metrics; the ones we're looking for are `observe-drivers-oplog` and `observe-drivers-polling` in the `mongo-livedata` section. If `observe-drivers-polling` is zero or not rendered at all, then all of your `observeChanges` calls are using `OplogObserveDriver`!