



(<http://richsilv.github.io/>)



## Neuron Mint Garden


(<http://richsilv.github.io/>)

*Observations on Meteor, running, the world, and anything  
else I can be bothered to write about.*

# *Understanding Meteor's Low-Level Publications API*

BY RICHARD SILVERTON ([HTTP://RICHSILV.GITHUB.IO/ABOUT/](http://richsilv.github.io/about/))

 AUGUST 01, 2014     6 COMMENTS

 PERMALINK ([HTTP://RICHSILV.GITHUB.IO/METEOR/METEOR-LOW-LEVEL-PUBLICATIONS/](http://richsilv.github.io/meteor/meteor-low-level-publications/))

## Motivation

As increasing numbers of developers are discovering, Meteor (<https://www.meteor.com/>) is an incredibly powerful, feature-rich platform with which to create web apps. By obviating many of the traditional frustrations involved in rolling out a production app, like user authentication or synchronisation of data on client and server, Meteor makes it remarkably easy to progress from idea to prototype to fully-functional product in no time at all. However, the developer can become dazzled by such power, and there's the danger that he or she can end up producing complex, multi-page applications without fully understanding one of the most fundamental components of the platform - the Pub/Sub framework. This is certainly an accurate description of my personal history with Meteor, and a situation I have recently put right, as I describe below.

## Meteor.publish

The first remark I should make is that the canonical demonstration of Meteor's low-level publish API not only exists, but it's almost the first thing ([http://docs.meteor.com/#meteor\\_publish](http://docs.meteor.com/#meteor_publish)) to appear in the official documentation. I can only assume that this is part of the problem - the `counts-by-room` example is relatively subtle and benefits from some understanding of DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) (the protocol developed by Meteor specifically for client-server communication) - which will probably leave many readers skipping over it on their first visit in their enthusiasm to get to the shiny stuff below. Which, it turns out, is a mistake, at least judging by the number of recent (<http://stackoverflow.com/questions/25033436/is-it-possible-to-publish-subscribe-to-a-remote-api-instead-of-a-collection-in-m>) questions (<http://stackoverflow.com/questions/25086631/meteor-reactive-publishes-subscribes>) on (<http://stackoverflow.com/questions/25079984/meteor-publish-method>) SO (<http://stackoverflow.com/questions/25045783/meteor-ddp-how-to-get-notified-when-a-new-document-is-added-to-a-collection/25069279#25069279>) which can be resolved with a good understanding of this topic.

### Two Publication Patterns within a single Method

Anybody who has some familiarity with Meteor will be aware of the first of the available publication patterns, in which the publish function uses the familiar collection API to return a collection cursor. Those with knowledge of DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) will also be aware that the cursor object itself cannot actually be communicated via this protocol, so returning a cursor is really a way of describing the documents (current and future) which the app designer wants to make available on the client. Meteor's internals then take care of the actual transmission of these objects via DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>), as well as continuing to observe the cursor for changes and sending the `ready` message after initial transmission, which is used by the `onReady` callback and `ready` methods on the client side, and further utilised in iron-router's `wait` method and `waitOn` hooks.

Here's the basic setup to which I'll be referring, which involves a

Understanding Meteor's Low-Level Publishing API | <https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>

test collection being populated with a random integer in the range [0, 1000) every ten seconds. There's also an example of the familiar cursor-based pub-sub pattern:

```
TestData = new Meteor.Collection('testdata');

if (Meteor.isServer) {
  Meteor.publish('cursorPub', function(filter) {
    return TestData.find(filter || {});
  });

  Meteor.startup(function () {
    Meteor.setInterval(function() {
      TestData.insert({number: Math.floor(Math.random() * 1000)});
    }, 10000);
  });
}

if (Meteor.isClient) {
  Session.set('filter', {});

  Deps.autorun(function(c) {
    mySub = Meteor.subscribe('cursorPub', Session.get('filter'));
  });
}
```

Note that the subscription on the client side is contained within a `Deps.autorun` block and depends on a reactive `Session` variable. This means that we can change the subscription filter simply by changing the value of the session variable, and Meteor is clever enough to manage the resubscription (including doing nothing if the filter hasn't actually changed).

We can use the Websockets filter in the Network inspector within Chrome Dev Tools to see exactly how this pub/sub example translates into DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) messages sent and received by the client:

Name	Data	Length	Time
websocket	["msg","added","collection":"testdata","id":"c4L7Eket9fzJXE5h","fields":{"number":"7233"}]	109	3:16:08 PM
	["msg","ready","subs":{"uH5jBo9fMWhc9vt"}]	57	3:16:04 PM
	["msg","added","collection":"testdata","id":"FoMgT8PNma84kNJ","fields":{"number":"327"}]	109	3:16:04 PM
	["msg","added","collection":"testdata","id":"ahZ7Eymz3SrlB5xW","fields":{"number":"954"}]	109	3:16:04 PM
	["msg","ready","subs":{"Q6P8pPkSD46PLBE"}]	57	3:16:04 PM
	["msg","added","collection":"meteor_autoupdate_clientVersions","id":"01f82aef0d4f165476b7d82fa60d311277d239","fields":{"current":true}]	158	3:16:04 PM
	["msg","connected","session":{"dEWZHB47PL2mmSpV"}]	62	3:16:04 PM
	["server_id","0"]	26	3:16:04 PM
	["msg","sub","id":"uH5jBo9fMWhc9vt","name":"cursorPub","params":{"id":"0"}]	89	3:16:04 PM
	["msg","sub","id":"Q6P8pPkSD46PLBE","name":"meteor_autoupdate_clientVersions","params":{"id":"0"}]	110	3:16:04 PM
	["msg","connect","version":"pre2","support":{"pre2":{"pre1":true}}]	78	3:16:04 PM
	o	1	3:16:04 PM

Reading from the bottom up, and ignoring messages relating to the `meteor_autoupdate_clientVersions` subscription, which is a meteor internal, we can see the following:

1. A `connect` message sent by the client to the server.
2. A `subscribe` message for the `cursorPub` publication, coming from the client, with an attached subscription id.
3. The `connected` message returned by the server.
4. Two `added` messages returned with a `collection` field of `testdata` so that the client knows where to store these documents, and the (in this case rather limited) document contents.
5. A `ready` message sent by the server indicating that initial data on this subscription has all been sent - note that the `subs` field has the same id as the `subscribe` message which was sent by the client in step (2).
6. A further `added` message as the `setInterval` block on the server runs again and adds a new document to the collection. Meteor is automatically observing the cursor which was returned by the `Meteor.publish` block and sending any changes to subscribing clients. Note that the timestamp on this message is several seconds after the others, confirming that this was a document added after the connection had been made and the initial data synchronised.

## Distributed Data Protocol

The second way to use the Pub/Sub model is really just an API to exactly this DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) flow from the server side, relating to a specific connection and subscription request. What this means is that the named publication function will be run once for each incoming subscription on that name, but rather than returning a cursor and leaving it for Meteor to generate the required DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) messages, it allows us to send customised DDP

Understanding Meteor's <https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) messages to suit the requirements of the application. Here's an example which does exactly the same as the one above:

```
Meteor.publish('ddpPub', function(filter) {
  var self = this;

  var subHandle = TestData.find(filter || {}).observeChanges
    added: function (id, fields) {
      self.added("testdata", id, fields);
    },
    changed: function(id, fields) {
      self.changed("testdata", id, fields);
    },
    removed: function (id) {
      self.removed("testdata", id);
    }
  });

  self.ready();

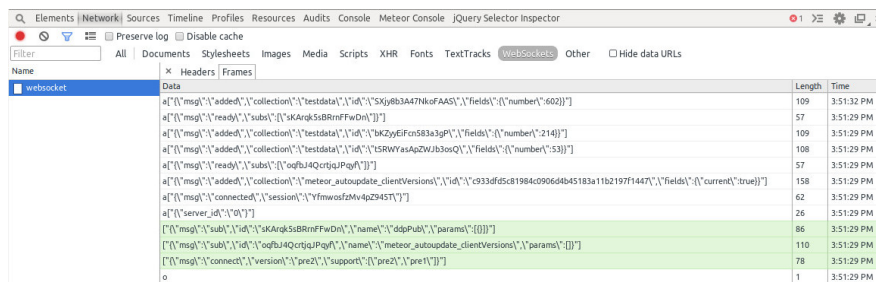
  self.onStop(function () {
    subHandle.stop();
  });
});
```

So what exactly are we doing here?

1. Storing a reference to the publish function's context in `self`, as we'll need to use it inside the functions contained within `observeChanges`, where the value of `this` will be different.
2. Setting up an observer to monitor the `TestData` collection (appropriately filtered). In this simple example, we are simply passing any changes to the returned document set on the server (i.e. when documents are `added`, `changed` or `removed`) to the subscribing client verbatim, by instructing the publish function to send an appropriate DDP (<https://github.com/meteor/meteor/blob/master/packages/livedata/DDP.md>) message with exactly the same details down to the client. This is what the `self.added`,

- Note that these functions will run immediately in a synchronous manner for that specific subscriber, so all the existing documents will fire the `added` hook before any further processing is done.
- Thus, by the time we reach `self.ready()` we can be confident that all the existing documents have already been sent with `self.added` calls, and it's safe to tell the client that their subscription is ready to use. It's very important that we make this call at some stage, otherwise anything that hooks into the subscription's `ready` method will never be fired.
- Finally, we have to make sure we `stop` the observer when the subscription is closed (at which point the `onStop` callback is fired). If we don't do this then the server will continue to observe the document set until it is restarted, even with no subscriber to send changes to, and server resource utilisation will increase inexorably as clients connect and reconnect. This would not be a good outcome!

When we inspect the result in Chrome Dev Tools, the result is uncannily similar:



Name	Headers	Frames	Data	Length	Time
websocket			[{"type": "added", "collection": "testdata", "id": "5Xjy6B3A47Nk0FAAS", "fields": {"number": 602}}]	109	3:51:32 PM
			[{"type": "ready", "subs": "tkAqK5dB8mFfwDn"}]	57	3:51:29 PM
			[{"type": "added", "collection": "testdata", "id": "bk2zyEifCn583a3pP", "fields": {"number": 214}}]	109	3:51:29 PM
			[{"type": "added", "collection": "testdata", "id": "tSRWYasAp2WJb3ocQ", "fields": {"number": 53}}]	108	3:51:29 PM
			[{"type": "ready", "subs": "oqfBj4QcrtjqPqf"}]	57	3:51:29 PM
			[{"type": "added", "collection": "meteor_autoupdate_clientVersions", "id": "c933dfc581984c99064b45183a11b2197f1447", "fields": {"current": true}}]	158	3:51:29 PM
			[{"type": "connected", "session": "YfmwofzHv4p2945T"}]	62	3:51:29 PM
			[{"type": "server_id", "id": "0"}]	26	3:51:29 PM
			[{"type": "sub", "id": "tkAqK5dB8mFfwDn", "name": "ddpPub", "params": {}}]	86	3:51:29 PM
			[{"type": "sub", "id": "oqfBj4QcrtjqPqf", "name": "meteor_autoupdate_clientVersions", "params": {}}]	110	3:51:29 PM
			[{"type": "connect", "version": "pre2", "support": {"pre2": {"pre1": {}}}]	78	3:51:29 PM
			o	1	3:51:29 PM

## Something Rather More Useful

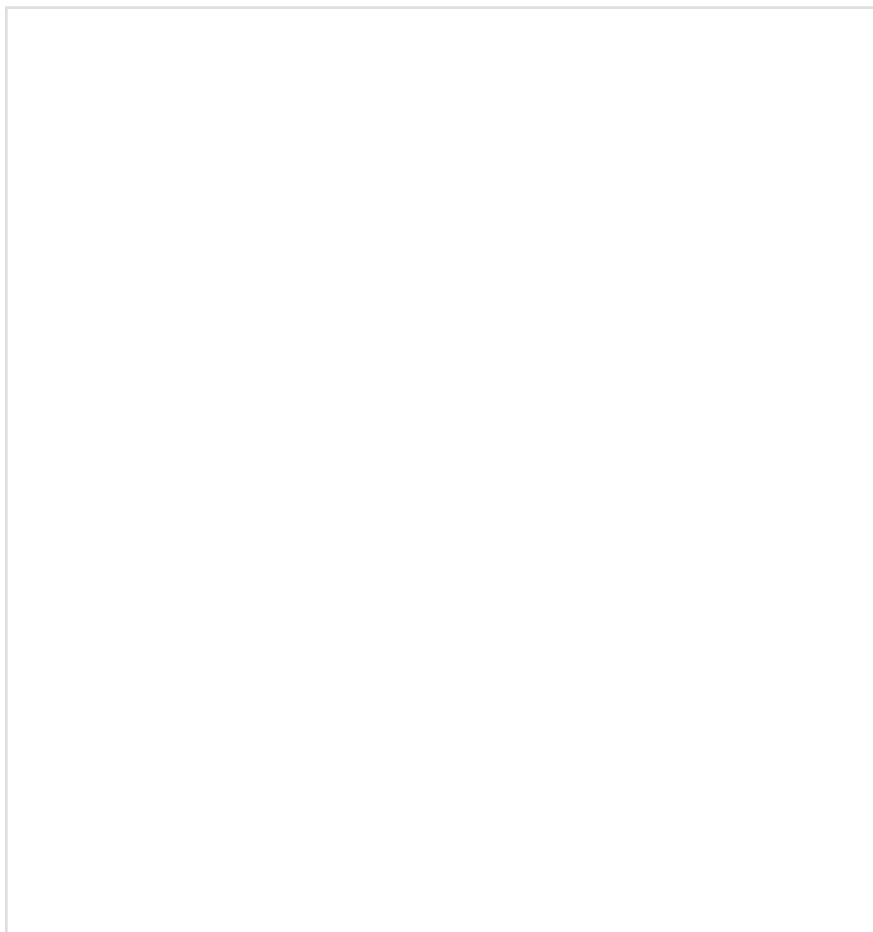
At this stage, it would be reasonable to ask what the point of all this was, since we've just recreated exactly the same series of messages using significantly more code. But the fact that we now have a way of intermediating the conversation between client and server databases gives us a huge amount of power. This is utilised very cleverly in the `counts-by-room` example

([http://docs.meteor.com/#meteor\\_publish](http://docs.meteor.com/#meteor_publish)) provided within the Meteor docs, but I'm going to take a slightly different tack.

## Existing Documents versus New Additions

### Pattern 1: A Document Counter

One way of confirming that the original document set had arrived intact would be to send the number of documents `added` before the `ready` call as an additional document itself. This would best be done using another collection to avoid the need to filter it out of the data set that you're actually interested in communicating.



```
CollectionCount = new Meteor.Collection('collectioncount');

Meteor.publish('ddpPub', function(filter) { // WHILST THIS
  var self = this,
      ready = false,
      count = 0;

  var subHandle = TestData.find(filter || {}).observeChanges
    added: function (id, fields) {
      if (!ready)
        count ++;
      self.added("testdata", id, fields);
    },
    changed: function(id, fields) {
      self.changed("testdata", id, fields);
    },
    removed: function (id) {
      self.removed("testdata", id);
    }
  });

  self.added("collectioncount", Random.id(), {Collection: "

  self.ready();
  ready = true;

  self.onStop(function () {
    subHandle.stop();
  });
});
```

This publish function will also populate the `CollectionCount` collection (which only needs to be constructed on the client) with an object that contains the number of documents in the existing set. You can then delay mission critical logic on the client until `TestData.find().count()` is equal to `CollectionCount.findOne({Collection: "testdata"}).Count`. Note that the actual logic will need to be slightly more involved to account for the period in which `CollectionCount.findOne({Collection: "testdata"})` returns nothing as the publish function hasn't yet sent the corresponding



## Pattern 2: An Additional Field

A similar scenario is one in which we don't need to know the exact number of documents in the collection when we subscribe, but we do need to know which they are. This can be solved as follows:

```
Meteor.publish('ddpPub', function(filter) {
  var self = this,
      ready = false;

  var subHandle = TestData.find(filter || {}).observeChanges({
    added: function (id, fields) {
      if (!ready)
        fields.existing = true;
      self.added("testdata", id, fields);
    },
    changed: function(id, fields) {
      self.changed("testdata", id, fields);
    },
    removed: function (id) {
      self.removed("testdata", id);
    }
  });

  self.ready();
  ready = true;

  self.onStop(function () {
    subHandle.stop();
  });
});
```

Now if we query the collection on the client immediately after we've received the `ready` message, we will notice that

`TestData.find().count` is equal to

`TestData.find({existing: true}).count()`. However, once

additional documents are added on the client side, this will cease to be the case as these will no longer have the `existing` property, allowing us to identify that these are genuinely newly-added documents from a global perspective.

The two examples above could be combined to solve a problem in which data was being rapidly generated and the client needed to be sure that, not only did it have the right number of documents, but those were exactly the documents in the existing set.

Alternatively, in some use cases, we might not want to send the existing documents over the wire at all, in which case we would simply not execute the `self.added` call if `!ready` evaluated to true.

## Conclusion

Hopefully, this has shed some light on the power and flexibility of the low-level Publications API, which I believe is frequently ignored by Meteor enthusiasts who may have grown too accustomed to the convenience of the higher-level cursor-based API. I'm sure there are hundreds of interesting and more elaborate use-cases, involving selective updates, periodic removals, record extension with myriad extra data and so on, and I look forward to seeing them.

## A Note on Subscriptions

Finally, as I pointed out earlier, I have chosen to put my subscription inside a `Deps.autorun` block on the client side, to allow automatic resubscription when the filter is amended. This provides other benefits from a reactivity perspective:

- If you have a Template helper which has a dependency on the readiness of a subscription, this allows you to resubscribe and maintain reactivity. What this means is that `mySub.ready()` in the example above will always reactively supply the state of the current subscription, even though the actual subscription object has been stopped and replaced by a new one. This is very convenient!
- In contrast, if you *don't* put your subscription in a `Deps.autorun` block like this, your reactivity can break when you resubscribe. This means that if you were using `mySub.ready()` in a helper function (or another reactive context) and then resubscribe with `mySub = Meteor.subscribe('testdata', newFilter);` your helper will not rerun. For this reason I would always recommend putting your subscriptions in `Deps.autorun` blocks and having them update by changing the value of Session variables, or another reactive data source.



 Recommend 2

Sort by Best ▾

Join the discussion...

**Caio Ribeiro Pereira** • a year ago

Nice post!

I'll add it to the newsletter UDGWebDev Weekly  
<http://weekly.udgwebdev.com>

1 ^ | ▾ • Reply • Share &gt;

**Baek Edmond** • 18 days ago

Fantastic post!

Now, I will make an effort to control low-level APIs.  
I am curious of that Tracker.autorun's behaviour is same  
as Deps.autorun's.

^ | ▾ • Reply • Share &gt;

**Jay** • a month ago

This is a great post. Currently, I need to aggregate the  
results of a query on the client. The way I'm doing it now  
is to observe the query's added/changed/remove  
callbacks on the client, populating null collections; this  
allows data aggregation while maintaining access to the  
original collection (storing the data on server isn't an  
option due to the nature of the data.)

Instead, using the techniques you describe I might be  
able to compute the data on the server, and then just  
send over the documents to the client in a client-side  
collection (trying to avoid explicit method calls)

^ | ▾ • Reply • Share &gt;

**dcsan** • 3 months ago

excellent article. would be good to know a bit more about  
how to use <http://docs.meteor.com/##/full/...> added to  
make server side joins like "virtual collections"

^ | ▾ • Reply • Share &gt;

**Allen Eubank** • 3 months ago

8 months old and this article is still awesome! Thanks for  
the write up. I was definitely one of those users that  
skimmed over the pub/sub parts of Meteor. I was too  
excited about reactivity. DDP is going to be one of the  
main focuses of my studies now. It's still like magic to

Next article (<http://richsilv.github.io/running/biomechanics-of-heart-rate-training/>)

---

© 2015 Richard Silverton. Powered by Jekyll (<http://jekyllrb.com>) using the So Simple Theme (<http://mademistakes.com/so-simple/>).



(<https://twitter.com/richsilvo>)



(<https://www.facebook.com/richard.silverton.5>)



(<https://plus.google.com/u/0/117608652502810550836>)



(<https://www.linkedin.com/profile/view?id=15501651>)



(<http://stackoverflow.com/users/2068581/richsilv>)



(<https://github.com/richsilv>)