meteor: common mistakes

David Weldon

Over the past year I've spent countless hours reviewing code and answering questions both on stackoverflow and at the monthly SF devshop. There are certain patterns of mistakes and misunderstandings that most meteor developers (myself included) make on their journey toward learning the framework. In this post I'll catalog some of the most frequent occurrences in the hopes that you can avoid being bitten by at least one of them.

Profile editing

Do you have any code that looks like this?

```
if (Meteor.user().profile.isAdmin)
  // do important admin things
```

If so, I've got some bad news for you. Any user can open up a console and run:

```
Meteor.users.update(Meteor.userId(), {$set: {'profile.isAdmin': true}});
```

Surprise! User profiles are editable by default even if `insecure` has been removed. To prevent this, just add the following deny rule:

```
Meteor.users.deny({
  update: function() {
    return true;
  }
});
```

Published secrets

Let's say you have a publisher for all of the users in a group:

```
Meteor.publish('groupUsers', function(groupId) {
  check(groupId, String);
  var group = Groups.findOne(groupId);
  var selector = {_id: {$in: group.members}};
  return Meteor.users.find(selector);
});
```

What's wrong with this (apart from the fact that it isn't reactive)?

Unless you specify which fields are to be returned, mongo will return **all** of them, which means that

clients can see login tokens, encrypted passwords, and anything else stored in your user documents.

You must specify which fields you want when publishing user data in order to prevent leaking your user's secrets. The corrected function could look something like this:

```
Meteor.publish('groupUsers', function(groupId) {
  check(groupId, String);
  var group = Groups.findOne(groupId);
  var selector = {_id: {$in: group.members}};
  var options = {fields: {username: 1}};
  return Meteor.users.find(selector, options);
});
```

Variables as keys

Building dynamic selectors is extremely common when creating a rich user interface. Imagine a user being able to search a collection of animals by a key (species, size, region, etc.) and a value from a text input. Your code could look something like:

```
var key = 'species';
var value = 'elephant';

var selector = {key: value};
Animals.find(selector);
```

But there is a subtle JavaScript gotcha that will prevent this `find` from returning the right results:

> Using a variable identifier as a key in an object literal will substitute the identifier and not the value.

`selector` is actually evaluated as `{key: 'elephant'}` and not `{species: 'elephant'}` as intended. The only way to fix this is to initialize `selector` as an empty object and then use bracket notation to set the key:

```
var key = 'species';
var value = 'elephant';

var selector = {};
selector[key] = value;
Animals.find(selector);
```

Subscriptions don't block

Many aspects of the framework seem like magic. So much so that it may cause you to forget how web browsers work. Take this simple example:

```
Meteor.subscribe('posts');
var post = Posts.findOne();
```

The idea that `post` will be `undefined` is the root cause of roughly one in twenty meteor questions on stackoverflow.

`subscribe` works like a garden hose - you turn it on and, after a while, things come out the other end. Activating the subscription does not block execution of the browser, therefore an immediate `find` on the collection won't return any data.

`subscribe` does, however, have an optional callback which you can use like so:

```
Meteor.subscribe('posts', function() {
  console.log(Posts.find().count());
});
```

Alternatively, you can call `ready` on the subscription handle:

```
var handle = Meteor.subscribe('posts');

Tracker.autorun(function() {
  if (handle.ready())
    console.log(Posts.find().count());
});
```

In more complex apps, your subscription activation code and your data consumption code are often very separate, which reduces the value of the above techniques. Fortunately there are two common ways of tackling the problem:

1. If you use iron router, you can wait on subscriptions, prior to rendering the dependent templates.
2. You can use guards to check if the data exists within your template code.

I'd recommend using a combination of both approaches in your applications depending on your UI requirements. Waiting on a subscription will force a delay before rendering the page, whereas guards will progressively render data as it's received on the client.

Corollary: In meteor, the majority of "Cannot read property of undefined"errors are caused by an incorrect assumption about the existence of subscribed data.

Overworked helpers

Helpers act like a rosetta stone for your templates. They can read things like collection documents, session data, and the current template context,then mix it all up and spit out a nicely formatted result.

Notice that I didn't say anything about calling methods, changing state, or posting messages on twitter. This is because helpers are intended to be synchronous translators which are free of side effects.

Because helpers are reactive, developers are often tempted to use them as a shortcut to link changes together. To illustrate this point, here's an example helper which formats a post comment:

```
Template.postComment.helpers({
  message: function() {
    var comment = Comments.findOne(this.commentId);

    if (Session.get('isExcited')) {
      return comment.text.toUpperCase() + '!';
    } else {
```

```
      return comment.text;
    }
  }
});
```

Now let's suppose we need to animate the message whenever the comment text changes. We could add some jQuery to our helper, but that makes a big assumption: that the helper only reruns under one condition. What if the session variable changes? What if some other property of the comment gets updated? What if changes in the parent cause `postComment` to be re-rendered?

It's safe to assume your helpers will run many more times than you expect them to. For this reason, its never a good idea to use them beyond their intended purpose. Instead you should use tools like autorun and observeChanges to solve these problems.

Note that if your helper needs the result of an asynchronous operation (like a method call) see the answers to this question.

Sorted publish

When you publish documents to the client, they are merged with other documents from the same collection and rearranged into an in-memory data store called minimongo. The key word being **rearranged**.

Many new meteor developers have a mental model of published data as existing in an ordered list. This leads to questions like: "I published my data in sorted order, so why doesn't it appear that way on the client?" That's expected. There's one simple rule to follow:

> If you need your documents to be ordered on the client, sort them on the client.

Sorting in a publish function isn't usually necessary unless the result of the sort changes which documents are sent (e.g. you are using a `limit`).

You may, however, want to retain the server-side sort in cases where the data transmission time is significant. Imagine publishing several hundred blog posts but initially showing only the most recent ten. In this case, having the most recent documents arrive on the client first would help minimize the number of template renderings. [1]

Data attributes

Remember that hack where you used to stuff all of your application state into the DOM using data attributes? I constantly see code from new users that looks like this:

```
<template name="nametag">
  <div data-name="{{name}}">{{name}}</div>
</template>

Template.nametag.events({
  click: function(e) {
```

```
    console.log($(e.currentTarget).data().name);
  }
});
```

Meteor is here to stop the madness. Because helpers, event handlers, and templates share the same context, the same code should be written as:

```
<template name="nametag">
  <div>{{name}}</div>
</template>

Template.nametag.events({
  click: function(e) {
    console.log(this.name);
  }
});
```

Caveat: Data attributes may be necessary if you are using a jQuery plugin which requires them.

---

[1] Thanks to Josh Owens for pointing out this exception.

David Weldon – *Co-Founder and CTO at Edthena*

December 02, 2014

Read Next: meteor: How we use Kadira at Edthena

- Blog Home
- Archive
- Bio
- RSS

You can also find me on GitHub. Get in touch via Email.