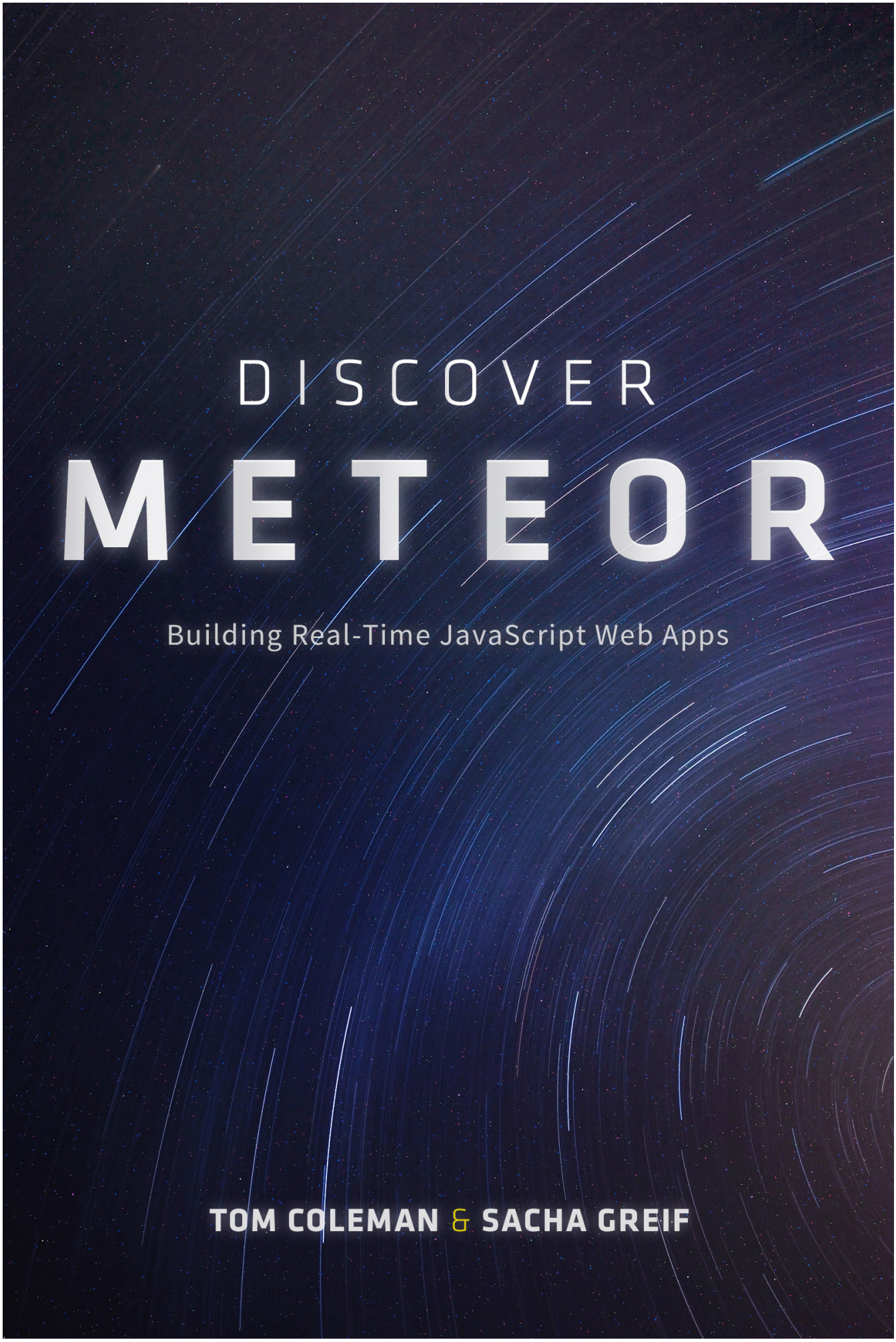

DISCOVER
METEOR
Building Real-Time JavaScript Web Apps
TOM COLEMAN & SACHA GREIF

DISCOVER **METEOR**

Building Real-Time JavaScript Web Apps

Tom Coleman & Sacha Greif

wwww.discovermeteor.com

In this chapter, you will:

- Learn about Meteor's templating language, Handlebars.
- Create your first three templates.
- Learn how Meteor managers work.
- Get a basic prototype working with static data.

Our First Template

To ease into Meteor development, we'll adopt an outside-in approach. In other words we'll build a “dumb” HTML/JavaScript outer shell first, and then hook it up to our app's inner workings later on.

This means that in this chapter we'll only concern ourselves with what's happening inside the `/client` directory.

Let's create a new file named `main.html` inside our `/client` directory, and fill it with the following code:

```
<head>
  <title>Microscope</title>
</head>
<body>
  <div class="container">
  <header class="navbar">
    <div class="navbar-inner">
      <a class="brand" href="/">Microscope</a>
    </div>
  </header>
  <div id="main" class="row-fluid">
    {{> postsList}}
  </div>
  </div>
</body>
```

client/main.html

This will be our main app template. As you can see it's all HTML except for a single `{{> postsList}}` tag, which is an insertion point for the `postsList` template as we'll soon see. For now, let's create a couple more templates.

Meteor Templates

At its core, a social news site is comprised of posts organized in lists, and that's exactly how we'll organize our templates.

Let's create a `/views` directory inside `/client`. This will be where we put all our templates, and to keep things tidy we'll also create `/posts` inside `/views` just for our post-related templates.

Finding Files

Meteor is great at finding files. No matter where you put your code in the `/client` directory, Meteor will find it and compile it properly. This means you never need to manually write include paths for JavaScript or CSS files.

It also means you could very well put all your files in the same directory, or even all your code in the same file. But since Meteor will compile everything to a single minified file anyway, we'd rather keep things well-organized and use a cleaner file structure.

We're finally ready to create our second template. Inside `client/views/posts`, create `posts_list.html`:

```
<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}
  </div>
</template>
```

client/views/posts/posts_list.html

And `post_item.html`:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
    </div>
  </div>
</template>
```

client/views/posts/post_item.html

Note the `name="postsList"` attribute of the template element. This is the name that will be used by Meteor to keep track of what template goes where.

It's time to introduce Meteor's templating system, **Handlebars**. Handlebars is simply HTML, with the addition of three things: *partials*, *expressions* and *block helpers*.

Partials use the `{{> templateName}}` syntax, and simply tell Meteor to replace the partial with the template of the same name (in our case `postItem`).

Expressions such as `{{title}}` either call a property of the current object, or the return value of a template helper as defined in the current template's manager (more on this later).

Finally, *block helpers* are special tags that control the flow of the template, such as `{{#each}}…{{/each}}` or `{{#if}}…{{/if}}`.

Going Further

You can refer to the **official Handlebars site** or **this handy tutorial** if you'd like to learn more about Handlebars.

Armed with this knowledge, we can easily understand what's going on here.

First, in the `postsList` template, we're iterating over a `posts` object with the `{{#each}}…{{/each}}` block helper. Then, for each iteration we're including the `postItem` template.

Where is this `posts` object coming from? Good question. It's actually a template helper, and we'll define it when we look at template managers.

The `postItem` template itself is fairly straightforward. It only uses three expressions: `{{url}}` and `{{title}}` both return the document's properties, and `{{domain}}` calls a template helper.

We've mentioned “template helpers” a lot throughout this chapter without really explaining what they do. But in order to fix this, we must first talk about managers.

Template Managers

Up to now we've been dealing with Handlebars, which is little more than HTML with a few tags sprinkled in. Unlike other languages like PHP (or even regular HTML pages, which can include JavaScript), Meteor keeps templates and their logic separated, and these templates don't do much by themselves.

In order to come to life, a template needs a **manager**. You can think of the manager as the chef that takes raw ingredients (your data) and prepares them, before handing out the finished dish to the waiter (the template) who then presents it to you.

In other words, while the template's role is limited to displaying or looping over variables, the manager is the one who actually does the heavy lifting by assigning a value to each variable.

Managers?

When we asked around to see what other Meteor developers called template managers, half said “controllers”, and half said “those files where I put my JavaScript code”.

Managers aren't really controllers (at least, not in the sense of MVC controllers) and “TFWIPMJSC” isn't really that catchy, so we rejected both propositions.

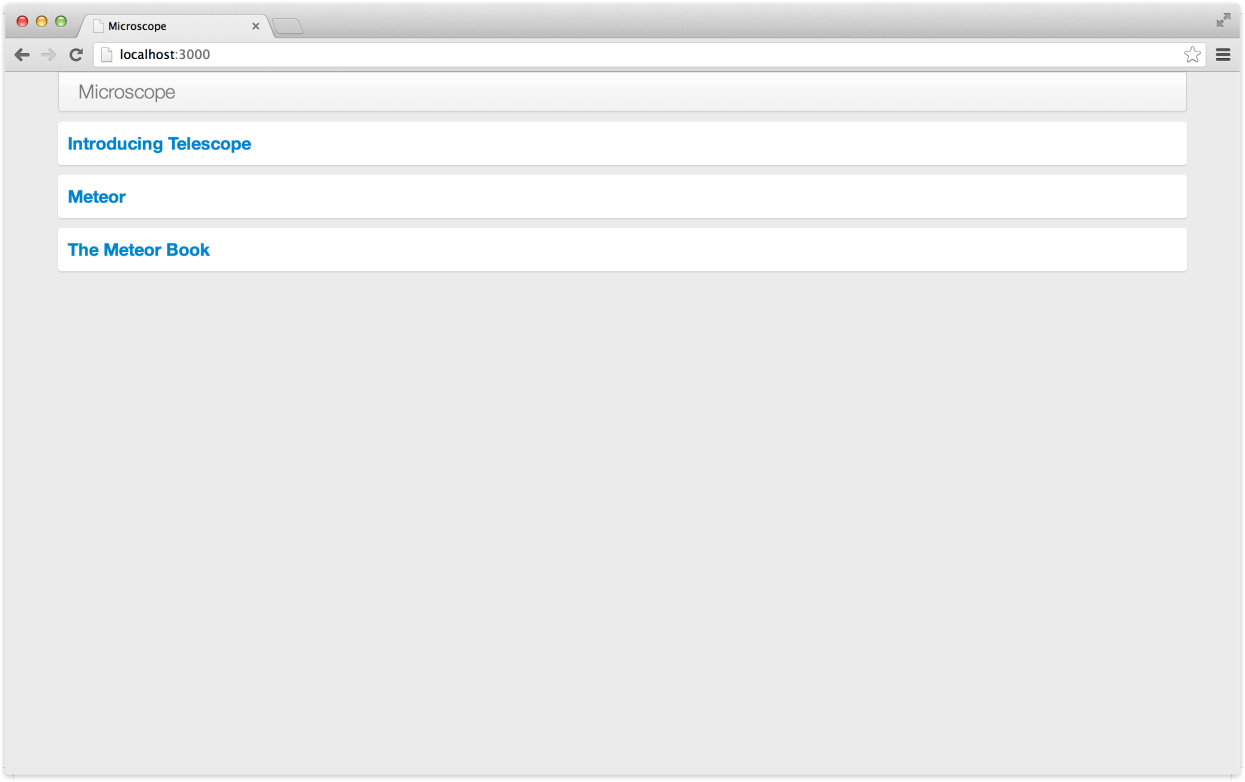
Since we still wanted a way to indicate what we were talking about, we came up with the term “manager” as a handy shortcut that didn't have any pre-existing meaning as far as web frameworks are concerned.

To keep things simple, we'll adopt the convention of naming the manager after the template, except with a **.js** extension. So let's create `posts_list.js` inside `/client/views/posts` right away and start building our first manager:

```
var postsData = [
  {
    title: 'Introducing Telescope',
    author: 'Sacha Greif',
    url: 'http://sachagreif.com/introducing-telescope/'
  },
  {
    title: 'Meteor',
    author: 'Tom Coleman',
    url: 'http://meteor.com'
  },
  {
    title: 'The Meteor Book',
    author: 'Tom Coleman',
    url: 'http://themeteorbook.com'
  }
];
Template.postsList.helpers({
  posts: postsData
});
```

client/views/posts/posts_list.js

If you've done it right, you should now be seeing something similar to this in your browser:



Our first templates with static data

Commit 3-1

Added basic posts list template and static data.

View on GitHub | Launch Instance

We're doing two things here. First we're setting up some dummy prototype data in the `postsData` array. That data would normally come from the database, but since we haven't seen how to do that yet (wait for the next chapter) we're "cheating" by using static data.

Second, we're using Meteor's `Template.myTemplate.helpers()` function to define a template helper called `posts` that simply returns out `postsData` array.

Defining the `posts` helper means it is now available for our template to use:

```
<template name="postsList">
  <div class="posts">
    {{#each posts}}
      {{> postItem}}
    {{/each}}
  </div>
</template>
```

client/views/posts/posts_list.html

So our template will be able to iterate over our `postsData` array, and send each object contained within to the `postItem` template.

The Value of "this"

We'll now create the `post_item.js` manager:

```
Template.postItem.helpers({
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  }
});
```

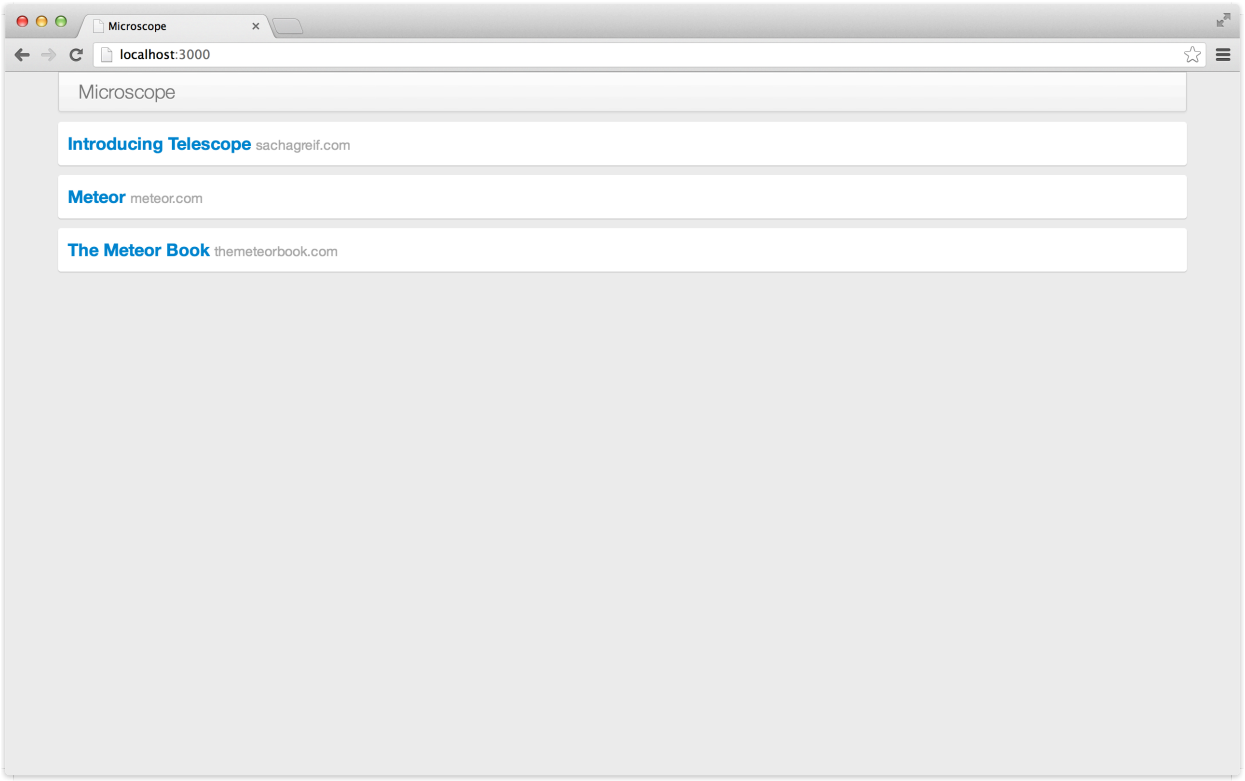
client/views/posts/post_item.js

Commit 3-2

Setup a domain helper on the postItem.

View on GitHub | Launch Instance

This time our `domain` helper's value is not an array, but an anonymous function. This pattern is much more common (and more useful) compared to our previous simplified dummy data examples.



Displaying domains for each links.

The `domain` helper takes a URL and returns its domain via a bit of JavaScript magic. But where does it take that url from

in the first place?

To answer that question we need to go back to our `posts_list.html` template. The `{{#each}}` block helper not only iterates over our array, it also **sets the value of `this` inside the block to the iterated object**.

This means that between both `{{#each}}` tags, each post is assigned to `this` successively, and that extends all the way inside the included template's manager (`post_item.js`).

We now understand why `this.url` returns the current post's URL. And moreover, if we use `{{title}}` and `{{url}}` inside our `post_item.html` template, Meteor knows that we mean `this.title` and `this.url` and returns the correct values.

If you've followed along correctly, you should be seeing a list of posts in your browser. That list is just static data, so it doesn't take advantage of Meteor's real-time features just yet. We'll show you how to change that in the next chapter!

Hot Code Reload

You might have noticed that you didn't even need to manually reload your browser window whenever you changed a file.

This is because Meteor tracks all the files within your project directory, and automatically refreshes your browser for you whenever it detects a modification to one of them.

Meteor's hot code reload is pretty smart, even preserving the state of your app in between two refreshes!