



DISCOVER **METEOR**

Building Real-Time JavaScript Web Apps

TOM COLEMAN & SACHA GREIF

DISCOVER **METEOR**

Building Real-Time JavaScript Web Apps

Version 1.3 (updated May 31, 2016)

Tom Coleman & Sacha Greif

Cover photo credit: **Perseid Hunting** by Darren Blackburn, licensed under a Creative Commons Attribution 2.0 Generic license.

www.discovermeteor.com

Do a little mental experiment for me. Imagine you're opening the same folder in two different windows on your computer.

Now click inside one of the two windows and delete a file. Did the file disappear from the other window as well?

You don't need to actually do these steps to know that it did. When we modify something on our local filesystems, the change is applied everywhere without the need for refreshes or callbacks. It just happens.

However, let's think about how the same scenario would play out on the web. For example, let's say you opened the same WordPress site admin in two browser windows and then created a new post in one of them. Unlike on the desktop, no matter how long you wait, the other window won't reflect the change unless you refresh it.

Over the years, we've gotten used to the idea that a website is something that you only communicate with in short, separate bursts.

But Meteor is part of a new wave of frameworks and technologies that are looking to challenge the status quo by making the web real-time and reactive.

What is Meteor?

Meteor is a platform built on top of Node.js for building real-time web apps. It's what sits between your app's database and its user interface and makes sure that both are kept in sync.

Since it's built on Node.js, Meteor uses JavaScript on both the client and on the server. What's more, Meteor is also able to share code between both environments.

The result of all this is a platform that manages to be very powerful and very simple by abstracting away many of the usual hassles and pitfalls of web app development.

Why Meteor?

So why should you spend your time learning Meteor rather than another web framework? Leaving aside all the various features of Meteor, we believe it boils down to one thing: Meteor is easy to learn.

More so than any other framework, Meteor makes it possible to get a real-time web app up and running on the web in a matter of hours. And if you've ever done front-end development before, you'll already be familiar with JavaScript and won't even need to learn a new language.

Meteor might be the ideal framework for your needs, or then again it might not. But since you can get started over the course of a few evenings or a week-end, why not try it and find out for yourself?

Why This Book?

For the past couple years, we've been working on numerous Meteor projects, spanning the range from web to mobile apps, and from commercial to open-source projects.

We learned a ton, but it wasn't always easy to find the answers to our questions. We had to piece things together from many different sources, and in many cases even invent our own solutions. So with this book, we wanted to share all these lessons, and create a simple step-by-step guide that will walk you through building a full-fledged Meteor app from scratch.

The app we'll be building is a simplified version of a social news site like [Hacker News](#) or [Reddit](#), which we'll call Microscope (by analogy with its big brother, Meteor open-source app [Telescope](#)). While building it, we'll address all the different elements that go into building a Meteor app, such as user accounts, Meteor collections, routing, and more.

Who Is This Book For?

One of our goals while writing the book was to keep things approachable and easy to understand. So, you should be able to follow along even if you have no experience with Meteor, Node.js, MVC frameworks, or even server-side coding in general.

On the other hand, we do assume familiarity with basic JavaScript syntax and concepts. But if you've ever hacked together some jQuery code or played around with the browser's developer console, you should be OK.

If you're not that comfortable with JavaScript yet, we suggest checking out our [JavaScript primer for Meteor](#) before getting started with the book.

About the Authors

In case you're wondering who we are and why you should trust us, here is a little more background on both of us.

Tom Coleman founded [Percolate Studio](#), a web development shop with a focus on quality and user experience, was one of the creators of the [Atmosphere](#) package repository, and is also one of the brains behind many other Meteor open-source projects. He now works full-time for the Meteor Development Group.

Sacha Greif has worked with startups such as [Hipmunk](#) and [RubyMotion](#) as a product and web designer. He's the creator of [Telescope](#) and [Sidebar](#) (which is based on Telescope).

Chapters & Sidebars

We wanted this book to be useful both for the novice Meteor user and the advanced programmer, so we split the chapters into two categories: regular chapters (numbered 1 through 14) and sidebars (.5 numbers).

Regular chapters will walk you through building the app, and will try to get you operational as soon as possible by explaining the most important steps without bogging you down with too much detail.

On the other hand, sidebars will go deeper into Meteor's intricacies, and will help you get a better understanding of what's really going on behind the scenes.

So if you're a beginner, feel free to skip the sidebars on your first read, and come back to them later

on once you've played around with Meteor.

Git Commits

There's nothing worse than following along in a programming book and suddenly realizing your code has gotten out of sync with the examples and that nothing works like it should anymore.

To prevent this, we've set up [a GitHub repository for Microscope](#), and we'll also provide direct links to git commits every few code changes. Here's an example of what that will look like:

Commit 11-2

Display notifications in the header.

[View on GitHub](#)[Launch Instance](#)

But note that just because we provide these commits doesn't mean you should just go from one `git checkout` to the next. You will learn much better if you take the time to manually type out your app's code!

A Few Other Resources

If you ever want to learn more about a particular aspect of Meteor, the [official Meteor documentation](#) and [Meteor Guide](#) are the best places to start.

We also recommend [Stack Overflow](#) for troubleshooting and questions, and the [#meteor IRC channel](#) if you need live help.

Do I Need Git?

While being familiar with Git version control is not strictly necessary to follow along with this book, we strongly recommend it.

If you want to get up to speed, we recommend Nick Farina's **Git Is Simpler Than You Think**.

If you're a Git novice, we also recommend the **GitHub** app (Mac OS), which lets you clone and manage repos without using the command line, or **SourceTree** (Mac OS & Windows), both of which are free.

Getting in Touch

- If you'd like to get in touch with us, you can email us at **hello@discovermeteor.com**.
- Additionally, if you find a typo or another mistake in the book's contents, you can let us know by **submitting a bug in this GitHub repo**.
- If you have a problem with Microscope's code, you can **submit a bug in Microscope's repository**.
- Finally, for every other question you can also just leave us a comment in this app's side panel.

First impressions are important, and Meteor’s install process should be relatively painless. In most cases, you’ll be up and running in less than five minutes.

To begin with, if you’re on Mac OS or Linux you can install Meteor by opening a terminal window and typing:

```
curl https://install.meteor.com | sh
```

If you’re running Windows, refer to the [install instructions](#) on the Meteor site.

This will install the `meteor` executable onto your system and have you ready to use Meteor.

Not Installing Meteor

If you can’t (or don’t want to) install Meteor locally, we recommend checking out [Nitrous.io](#).

Nitrous.io is a service that lets you run apps and edit their code right in your browser, and we’ve written [a short guide](#) to help you get set up.

You can simply follow that guide up to (and including) the “Installing Meteor” section, and then follow along with the book again starting from the “Creating a Simple App” section of this chapter.

Creating a Simple App

Now that we have installed Meteor, let’s create an app. To do this, we use Meteor’s command line tool `meteor`:


```
meteor create microscope
```

This command will download Meteor, and set up a basic, ready to use Meteor project for you. When it's done, you should see a directory, `microscope/`, containing the following directories and files:

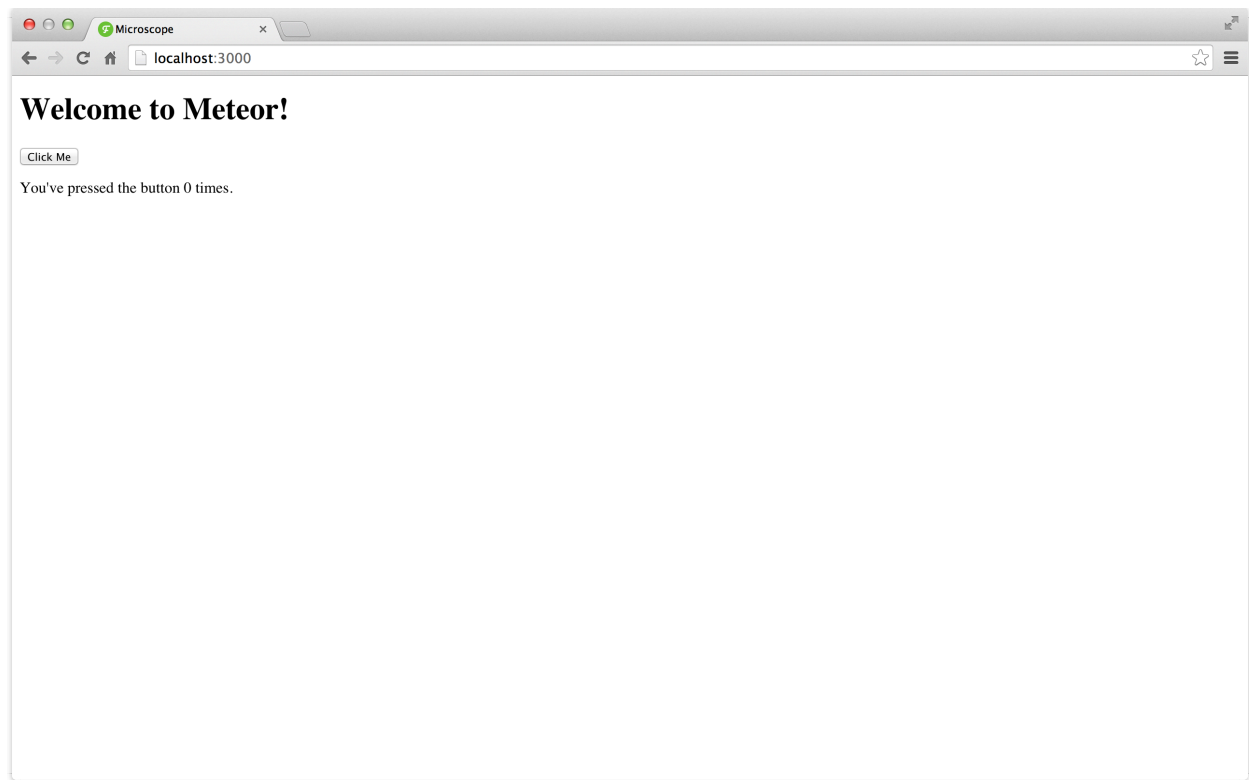
```
.meteor
client
| main.css
| main.html
| main.js
server
| main.js
.gitignore
package.json
```

The app that Meteor has created for you is a simple boilerplate application demonstrating a few simple patterns.

Even though our app doesn't do much, we can still run it. To run the app, go back to your terminal and type:

```
cd microscope
meteor
```

Now point your browser to `http://localhost:3000/` (or the equivalent `http://0.0.0.0:3000/`) and you should see something like this:



Meteor's Hello World.

Commit 2-1

Created basic microscope project.

[View on GitHub](#)

[Launch Instance](#)

Congratulations! You've got your first Meteor app running. By the way, to stop the app all you need to do is bring up the terminal tab where the app is running, and press `ctrl+c`.

Also note that if you're using Git, this is a good time to initialize your repo with `git init`.

Bye Bye Meteorite

There was a time where Meteor relied on an external package manager called Meteorite. Since Meteor version 0.9.0, Meteorite is not needed anymore since its features have been assimilated into Meteor itself.

So if you encounter any references to Meteorite's `mrt` command line utility while browsing Meteor-related material, you can safely replace them by the usual `meteor`.

Adding a Package

We will now use Meteor's package system to add the **Bootstrap** framework to our project.

This is no different from adding Bootstrap the usual way by manually including its CSS and JavaScript files, except that we rely on the package maintainer to keep everything up to date for us.

The `bootstrap` package is maintained by the `twbs` user, which gives us the full name of the package, `twbs:bootstrap`.

```
meteor add twbs:bootstrap
```

Note that we're adding Bootstrap **3**. Some of the screenshots in this book were taken with an older version of Microscope running Bootstrap **2**, which means they might look slightly different.

We'll also add the **Session** package, which will come in handy later:

```
meteor add session
```

Although Session is a first-party Meteor package, it's not enabled by default in new Meteor apps so we need to add it manually.

This is a good time to check which packages we're running. It turns out that in addition to the one we just added, Meteor also comes with a few packages enabled out of the box. You can see the list by opening the `packages` file inside the hidden `.meteor` directory:

```
# Meteor packages used by this project, one per line.
# Check this file (and the other files in this directory) into your repository.
#
# 'meteor add' and 'meteor remove' will edit this file for you,
# but you can also edit it by hand.

meteor-base           # Packages every Meteor app needs to have
mobile-experience     # Packages for a great mobile UX
mongo                 # The database Meteor supports right now
blaze-html-templates  # Compile .html files into Meteor Blaze views
reactive-var          # Reactive variable for tracker
jquery                # Helpful client-side library
tracker               # Meteor's client-side reactive programming library

standard-minifier-css # CSS minifier run for production mode
standard-minifier-js  # JS minifier run for production mode
es5-shim              # ECMAScript 5 compatibility for older browsers.
ecmascript            # Enable ECMAScript2015+ syntax in app code

autopublish           # Publish all data to the clients (for prototyping)
insecure              # Allow all DB writes from clients (for prototyping)

session
twbs:bootstrap
```

This might seem a bit overwhelming, but you don't need to worry about every single package right now. Just verify that our `twbs:bootstrap` package was successfully added at the end of your package list.

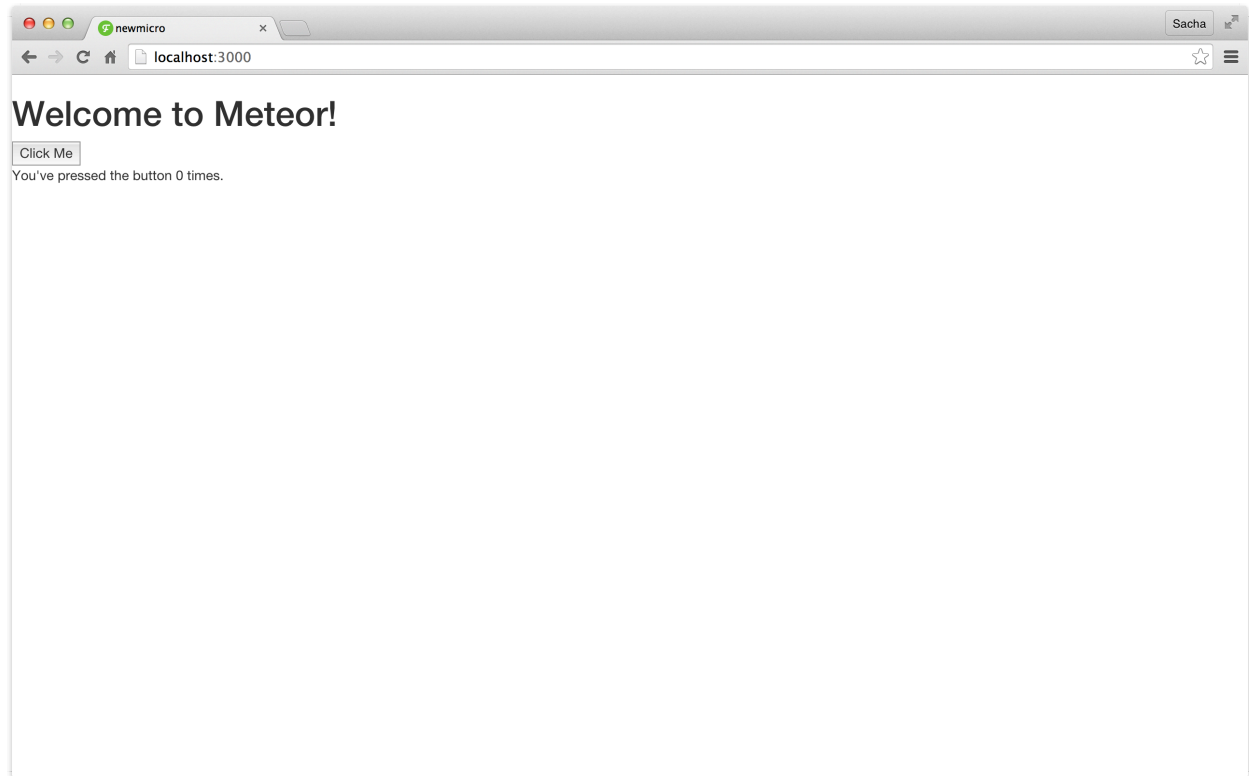
Also, note that unlike `twbs:bootstrap`, the other packages don't have an `author:` part in their name, signaling the fact that these are official, core Meteor packages.

Commit 2-2

Added bootstrap package.

[View on GitHub](#)[Launch Instance](#)

As soon as you've added the Bootstrap package you should notice a change in our bare-bones app:



With Bootstrap.

Unlike the “traditional” way of including external assets, we haven’t had to link up any CSS or JavaScript files, because Meteor takes care of all that for us! That’s just one of the many advantages of Meteor packages.

NPM Packages

While we’re on the subject of packages, you’ll notice we also have a `package.json` file in our repository:

```
{
  "name": "microscope",
  "private": true,
  "scripts": {
    "start": "meteor run"
  },
  "dependencies": {
    "meteor-node-stubs": "~0.2.0"
  }
}
```

This file is used to tell NPM (Node's package manager) what to do, just like the `.meteor/packages` file tells Meteor which packages to run. One key difference between both packages system though is that unlike Meteor, NPM doesn't automatically parse an app's `package.json` file. So you'll have to do it manually every time you add or remove an NPM package.

Let's do it right now with:

```
npm install
```

A Note on Packages

When speaking about packages in the context of Meteor, it pays to be specific. Meteor uses five basic types of packages:

- The `meteor-base` package stands alone: it contains **Meteor's core components**, and a Meteor app can't run without it.
- **First-party packages** come bundled with Meteor. Some, such as `mongo` or `session`, are included with new Meteor apps by default but can be removed if you don't need them, while others (such as `check` or `http`) need to be added explicitly.
- **Local packages** are specific to your app, and live in its local `/packages` directory.
- **Atmosphere packages** are custom, third-party packages that have been published to **Atmosphere**, Meteor's online package repository. They all follow the `author:package` naming convention.
- Finally, **NPM packages** are Node.js packages. They can't be included in your Meteor package list, but instead are listed in your app's `package.json` file.

The File Structure of a Meteor App

Before we begin coding, we must set up our project properly. To ensure we have a clean build, open up the `client` and `server` directory and delete the contents of `client/main.html`, `client/main.js`. Then go ahead and delete `server/main.js` entirely.

Next, create two new root directories inside `/microscope`: `/public`, and `/lib`.

Don't worry if all this breaks the app for now, we'll start filling in these files in the next chapter.

We should mention that some of these directories are special. When it comes to running code, Meteor has a few rules:

- Code in the `/server` directory only runs on the server.
- Code in the `/client` directory only runs on the client.

- Everything else runs on both the client and server.
- Your static assets (fonts, images, etc.) go in the `/public` directory.

And it's also useful to know how Meteor decides in which order to load your files:

- Files in `/lib` are loaded *before* anything else.
- Any `main.*` file is loaded *after* everything else.
- Everything else loads in alphabetical order based on the file name.

Note that although Meteor has these rules, it doesn't really force you to use any predefined file structure for your app if you don't want to. So the structure we suggest is just our way of doing things, not a rule set in stone.

We encourage you to check out the [official Meteor docs](#) if you want more details on this.

Is Meteor MVC?

If you're coming to Meteor from other frameworks such as Ruby on Rails, you might be wondering if Meteor apps adopt the MVC (Model View Controller) pattern.

The short answer is no. Unlike Rails, Meteor doesn't impose any predefined structure to your app. So in this book we'll simply lay out code in the way that makes the most sense to us, without worrying too much about acronyms.

No public?

OK, we lied. We don't actually need the `public/` directory for the simple reason that Microscope doesn't use any static assets! But, since most other Meteor apps are going to include at least a couple images, we thought it was important to cover it too.

By the way, you might also notice a hidden `.meteor` directory. This is where Meteor stores its own code, and modifying things in there is usually a very bad idea. In fact, you don't really ever need to look in this directory at all. The only exceptions to this are the `.meteor/packages` and

`.meteor/release` files, which are respectively used to list your smart packages and the version of Meteor to use. When you add packages and change Meteor releases, it can be helpful to check the changes to these files.

Underscores vs CamelCase

The only thing we'll say about the age-old underscore (`my_variable`) vs camelCase (`myVariable`) debate is that it doesn't really matter which one you pick as long as you stick to it.

In this book, we're using camelCase because it's the usual JavaScript way of doing things (after all, it's JavaScript, not `java_script`!).

The only exceptions to this rule are file names, which will use underscores (`my_file.js`), and CSS classes, which use hyphens (`.my-class`). The reason for this is that in the filesystem, underscores are most common, while the CSS syntax itself already uses hyphens (`font-family` , `text-align` , etc.).

Taking Care of CSS

This book is not about CSS. So to avoid slowing you down with styling details, we've decided to make the whole stylesheet available from the start, so you don't need to worry about it ever again.

CSS automatically gets loaded and minified by Meteor, so unlike other static assets it goes into `/client` , not `/public` . Go ahead and create a `client/stylesheet/` directory now, and put this `style.css` file inside it:

```
.grid-block, .main, .post, .comments li, .comment-form {  
  background: #fff;  
  border-radius: 3px;  
  padding: 10px;  
  margin-bottom: 10px;  
  -webkit-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);  
  -moz-box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15);  
  box-shadow: 0 1px 1px rgba(0, 0, 0, 0.15); }
```

```
body {  
  background: #eee;  
  color: #666666; }  
  
#main {  
  position: relative;  
}  
  
.page {  
  position: absolute;  
  top: 0px;  
  width: 100%;  
}  
  
.navbar {  
  margin-bottom: 10px; }  
/* line 32, ../sass/style.scss */  
.navbar .navbar-inner {  
  border-radius: 0px 0px 3px 3px; }  
  
#spinner {  
  height: 300px; }  
  
.post {  
  /* For modern browsers */  
  /* For IE 6/7 (trigger hasLayout) */  
  *zoom: 1;  
  position: relative;  
  opacity: 1; }  
  .post:before, .post:after {  
    content: "";  
    display: table; }  
  .post:after {  
    clear: both; }  
  .post.invisible {  
    opacity: 0; }  
  .post.instant {  
    -webkit-transition: none;  
    -moz-transition: none;  
    -o-transition: none;  
    transition: none; }  
  .post.animate {  
    -webkit-transition: all 300ms 0ms;  
    -moz-transition: all 300ms 0ms ease-in;  
    -o-transition: all 300ms 0ms ease-in;  
    transition: all 300ms 0ms ease-in; }  
  .post .upvote {  
    display: block;  
    margin: 7px 12px 0 0;  
    float: left; }  
  .post .post-content {  
    float: left; }
```

```
.post .post-content h3 {
  margin: 0;
  line-height: 1.4;
  font-size: 18px; }
.post .post-content h3 a {
  display: inline-block;
  margin-right: 5px; }
.post .post-content h3 span {
  font-weight: normal;
  font-size: 14px;
  display: inline-block;
  color: #aaaaaa; }
.post .post-content p {
  margin: 0; }
.post .discuss {
  display: block;
  float: right;
  margin-top: 7px; }

.comments {
  list-style-type: none;
  margin: 0; }
.comments li h4 {
  font-size: 16px;
  margin: 0; }
.comments li h4 .date {
  font-size: 12px;
  font-weight: normal; }
.comments li h4 a {
  font-size: 12px; }
.comments li p:last-child {
  margin-bottom: 0; }

.dropdown-menu span {
  display: block;
  padding: 3px 20px;
  clear: both;
  line-height: 20px;
  color: #bbb;
  white-space: nowrap; }

.load-more {
  display: block;
  border-radius: 3px;
  background: rgba(0, 0, 0, 0.05);
  text-align: center;
  height: 60px;
  line-height: 60px;
  margin-bottom: 10px; }
.load-more:hover {
  text-decoration: none; }
```

```
background: rgba(0, 0, 0, 0.1); }

.posts .spinner-container{
  position: relative;
  height: 100px;
}

.jumbotron{
  text-align: center;
}
.jumbotron h2{
  font-size: 60px;
  font-weight: 100;
}

@-webkit-keyframes fadeOut {
  0% {opacity: 0;}
  10% {opacity: 1;}
  90% {opacity: 1;}
  100% {opacity: 0;}
}

@keyframes fadeOut {
  0% {opacity: 0;}
  10% {opacity: 1;}
  90% {opacity: 1;}
  100% {opacity: 0;}
}

.errors{
  position: fixed;
  z-index: 10000;
  padding: 10px;
  top: 0px;
  left: 0px;
  right: 0px;
  bottom: 0px;
  pointer-events: none;
}

.alert {
  animation: fadeOut 2700ms ease-in 0s 1 forwards;
  -webkit-animation: fadeOut 2700ms ease-in 0s 1 forwards;
  -moz-animation: fadeOut 2700ms ease-in 0s 1 forwards;
  width: 250px;
  float: right;
  clear: both;
  margin-bottom: 5px;
  pointer-events: auto;
}
```


Commit 2-3

Re-arranged file structure.

[View on GitHub](#)[Launch Instance](#)

A Note on CoffeeScript

In this book we'll be writing in pure JavaScript. But if you prefer CoffeeScript, Meteor has you covered. Simply add the CoffeeScript package and you'll be good to go:

```
meteor add coffeescript
```

Some people like to work quietly on a project until it's perfect, while others can't wait to show the world as soon as possible.

If you're the first kind of person and would rather develop locally for now, feel free to skip this chapter. On the other hand, if you'd rather take the time to learn how to deploy your Meteor app online, we've got you covered.

We will be learning how to deploy a Meteor app in few different ways. Feel free to use each of them at any stage of your development process, whether you're working on Microscope or any other Meteor app. Let's get started!

Introducing Sidebars

This is a **sidebar** chapter. Sidebars take a deeper look at more general Meteor topics independently of the rest of the book.

So if you'd rather go on with building Microscope, you can safely skip it for now and come back to it later.

Deploying On Heroku

A fast and easy (and free!) way to deploy your Meteor app is to deploy to **Heroku** using the **Horse buildpack**.

See the **readme** for a step-by-step explanation.

Deploying On Galaxy

Galaxy is the Meteor Development Group's official hosting platform, and it's built in right into the Meteor command line tool via the `meteor deploy` command.

Galaxy doesn't offer a free plan, but it's a great option once you're ready to push your app live.

Note that the Meteor Development Group used to provide a free hosting service (usually referred to as just "Meteor hosting") but as of March 2016, that free tier sadly doesn't exist anymore.

Deploying with MupX

Finally, if you'd rather deploy on your own servers, **MupX** is a handy Docker deploy script that will set up and deploy your app on any Digital Ocean, AWS, etc. box.

These three ways of deploying Meteor apps should be enough for most use cases. Of course, we know some of you would prefer to be in complete control and set up their Meteor server from scratch. But that's a topic for another day... or maybe another book!

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.

If you haven't done so already, 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 navbar-default" role="navigation">
      <div class="navbar-header">
        <a class="navbar-brand" href="/">Microscope</a>
      </div>
    </header>
    <div id="main">
      {{> 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}}` template inclusion tag, which is an insertion point for the upcoming `postsList` template. For now, let's create a couple more templates.

Meteor Templates

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

Let's create a `/templates` directory inside `/client`. This will be where we put all our templates, and to keep things tidy we'll also create `/posts` inside `/templates` 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/templates/posts`, create `posts_list.html`:

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

`client/templates/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/templates/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 (note that the name of the actual *file* is not relevant).

It's time to introduce Meteor's templating system, **Spacebars**. Spacebars is simply HTML, with the addition of three things: *inclusions* (also sometimes known as “partials”), *expressions* and *block helpers*.

Inclusions use the `{{> templateName}}` syntax, and simply tell Meteor to replace the inclusion 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 [Spacebars documentation](#) if you'd like to learn more about Spacebars.

Armed with this knowledge, we can start to 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 you can think of it as a placeholder for a dynamic value.

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.

Template Helpers

Up to now we've been dealing with Spacebars, 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 **helpers**. You can think of these helpers as the cooks that take raw ingredients (your data) and prepare them, before handing out the finished dish (the templates) to the waiter, who then presents it to you.

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

Controllers?

It might be tempting to think of the file containing all of a template's helpers as a controller of sorts. But that can be ambiguous, as controllers (at least in the MVC sense) usually have a slightly different role.

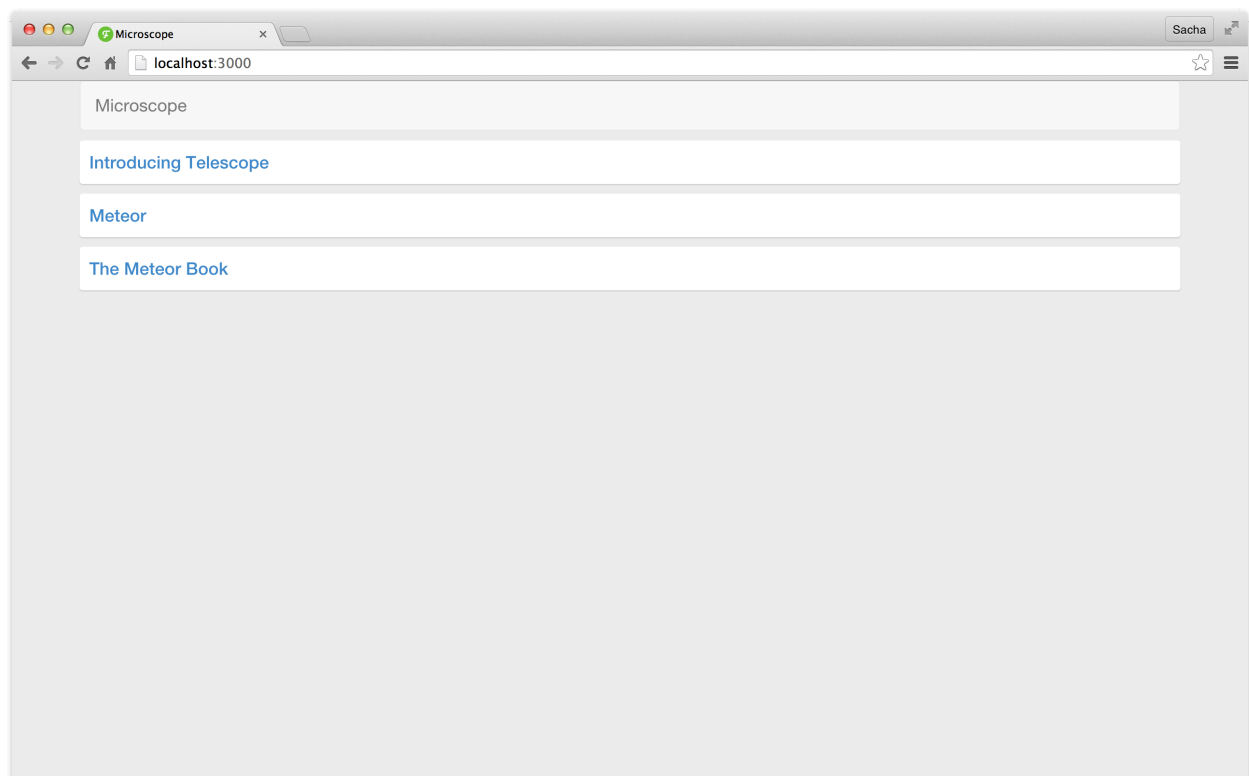
So we decided to stay away from that terminology, and simply refer to "the template's helpers" or "the template's logic" when talking about a template's companion JavaScript code.

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

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

`client/templates/posts/posts_list.js`

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



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.postsList.helpers()` function to create a template helper called `posts` that returns the `postsData` array we just defined above.

If you remember, we are using that `posts` helper in our `postsList` template:

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

client/templates/posts/posts_list.html

Defining the `posts` helper means it is now available for our template to use, so our template will be able to iterate over our `postsData` array and pass each object contained within to the `postItem` template.

Commit 3-1

Added basic posts list template and static data.

[View on GitHub](#)[Launch Instance](#)

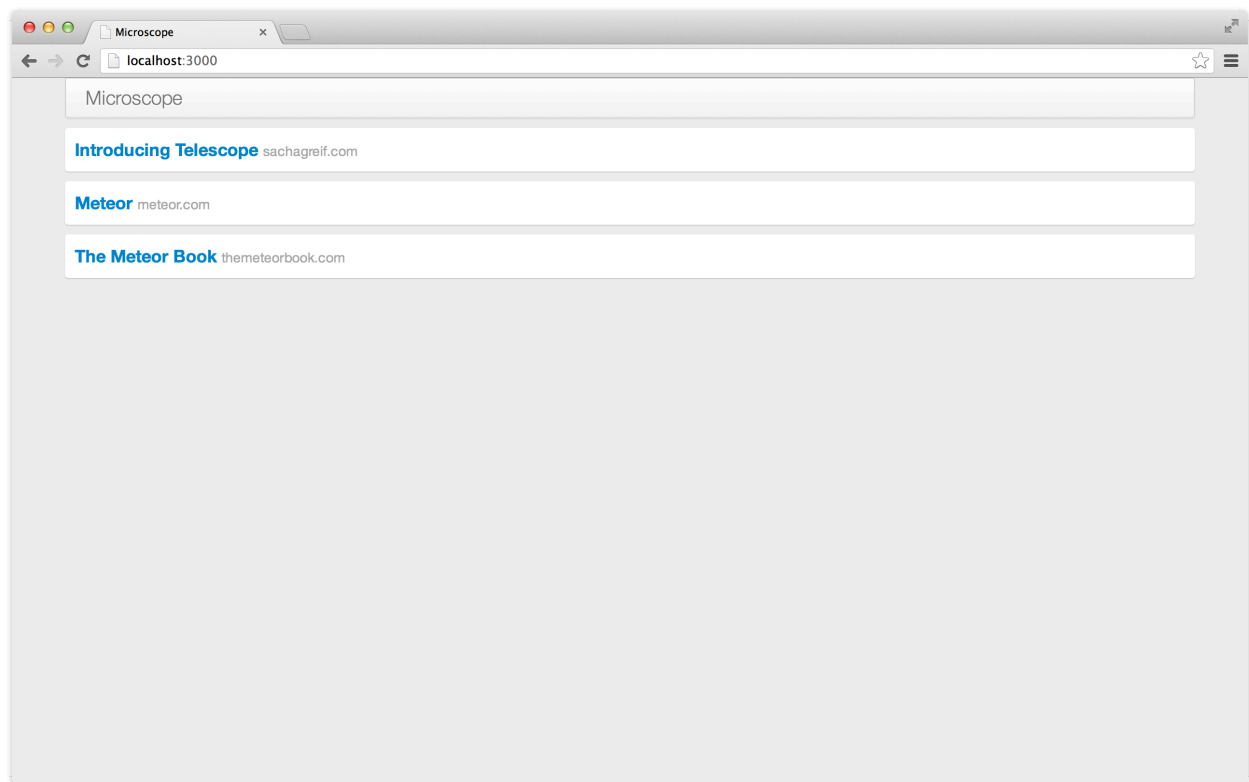
The `domain` Helper

Similarly, we'll now create `post_item.js` to hold the `postItem` template's logic:

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

client/templates/posts/post_item.js

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 example.



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.

Commit 3-2

Setup a `domain` helper on the `postItem`.

[View on GitHub](#)[Launch Instance](#)

JavaScript Magic

Although this is not specific to Meteor, here's a quick explanation of the above bit of "JavaScript magic". First, we're creating an empty anchor (`a`) HTML element and storing it in memory.

We then set its `href` attribute to be equal to the current post's URL (as we've just seen, in a helper `this` is the object currently being acted upon).

Finally, we take advantage of that `a` element's special `hostname` property to get back the link's domain name without the rest of the URL.

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!

GitHub is a social repository for open-source projects based around the **Git** version control system, and its primary function is to make it easy to share code and collaborate on projects. But it's also a great learning tool. In this sidebar, we'll quickly go over a few ways you can use GitHub to follow along with *Discover Meteor*.

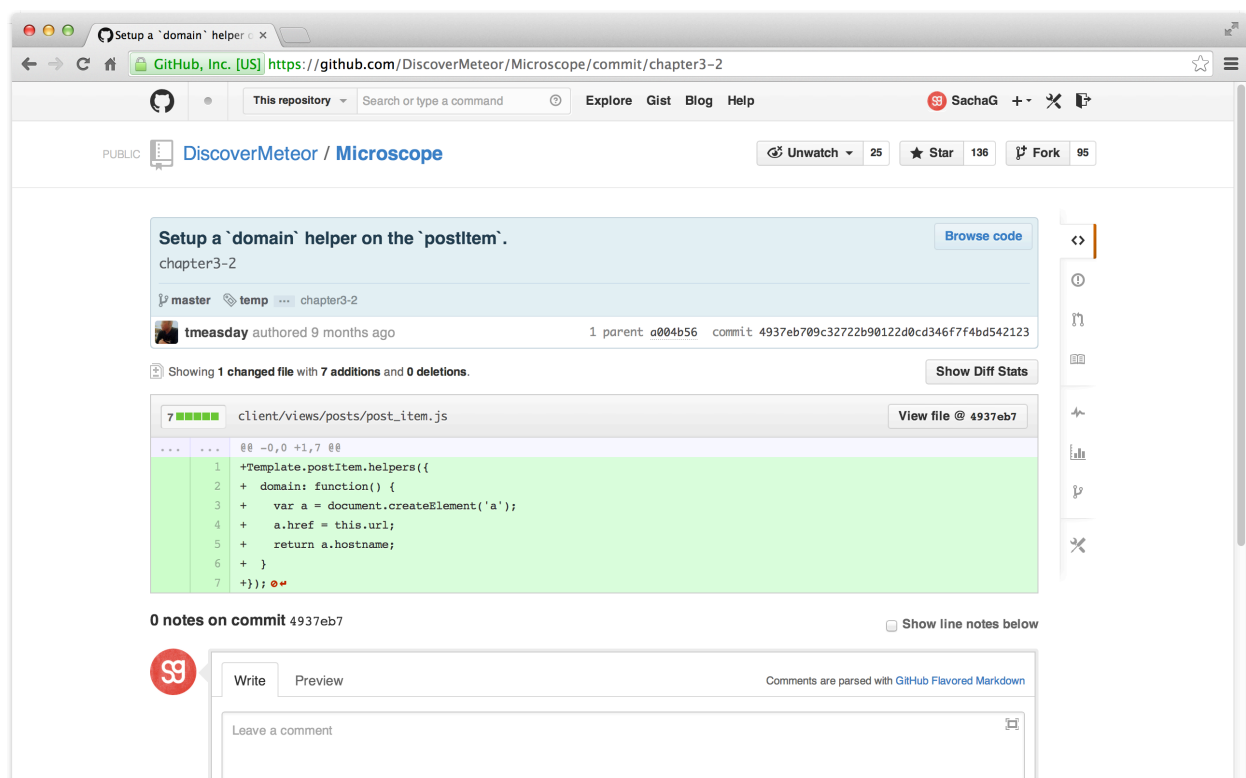
This sidebar assumes you're not that familiar with Git and GitHub. If you're already comfortable with both, feel free to skip on to the next chapter!

Being Committed

The basic working block of a git repository is a *commit*. You can think of a commit as a snapshot of your codebase's state at a given moment in time.

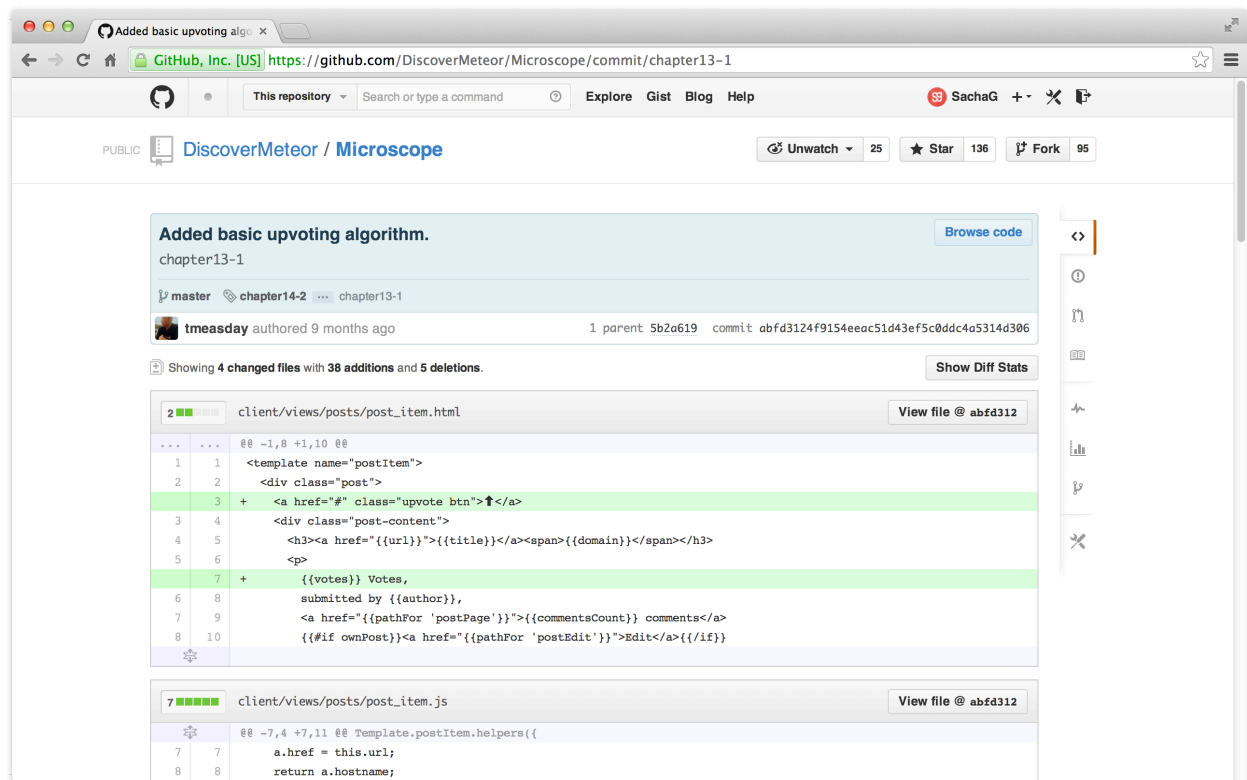
Instead of simply giving you the finished code for Microscope, we've taken these snapshots every step of the way, and you can see all of them online on GitHub.

For example, this is what **the last commit of the previous chapter** looks like:



What you see here is the “diff” (for “difference”) of the `post_item.js` file, in other words the changes introduced by this commit. In this case, we created the `post_item.js` file from scratch, so all its contents are highlighted in green.

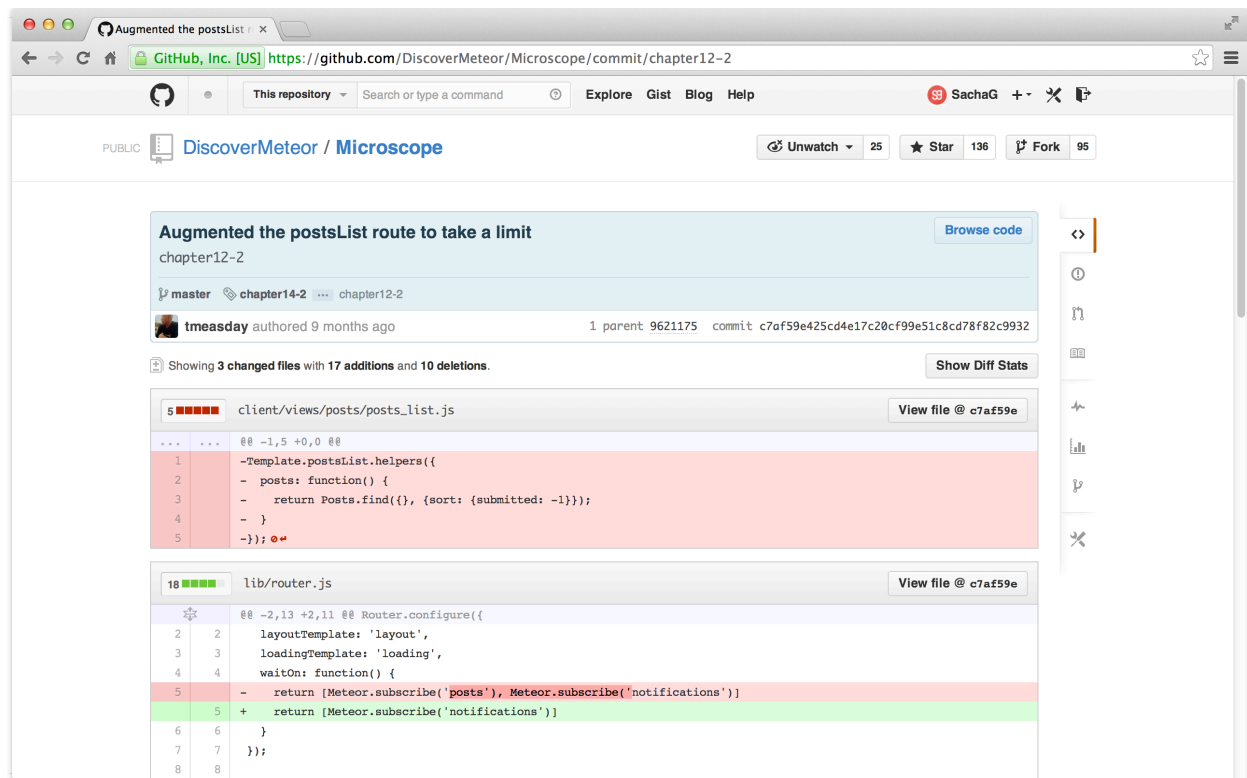
Let’s compare with an example from **later on in the book**:



Modifying code.

This time, only the modified lines are highlighted in green.

And of course, sometimes you’re not adding or modifying lines of code, but **deleting them**:



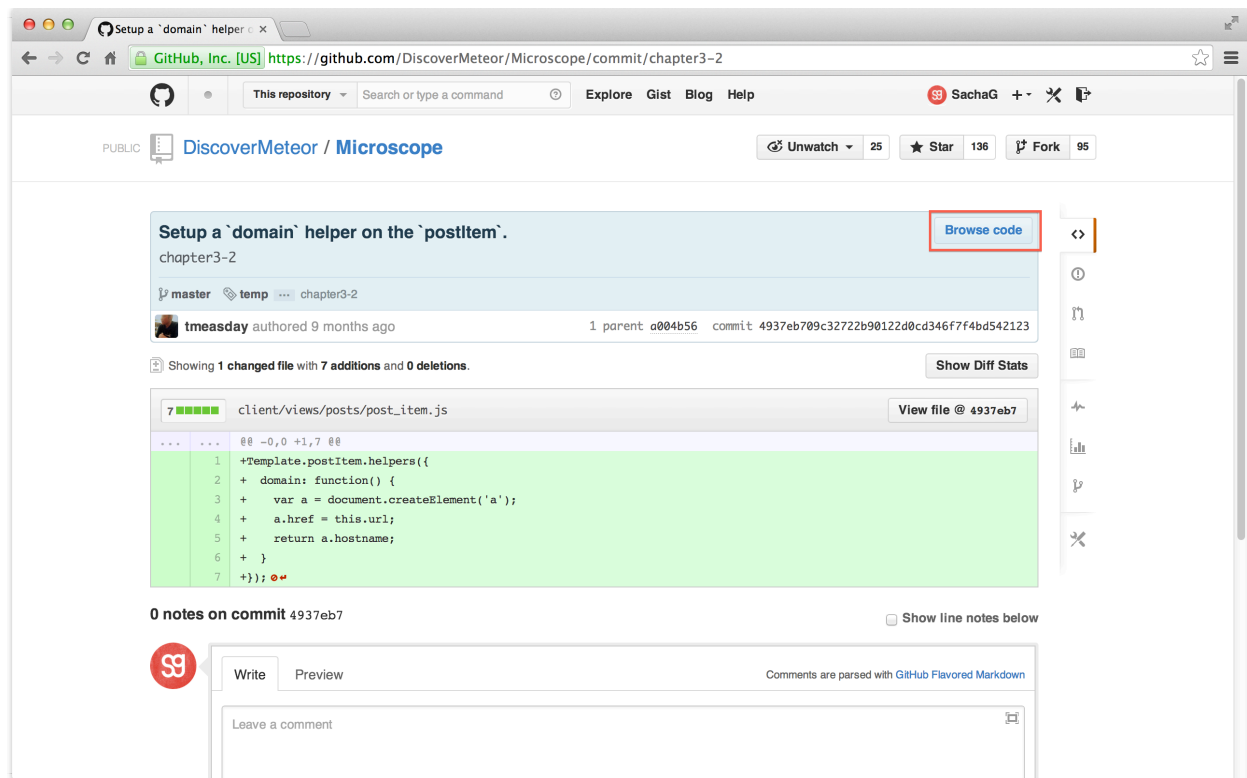
Deleting code.

So we've seen the first use of GitHub: seeing what's changed at a glance.

Browsing A Commit's Code

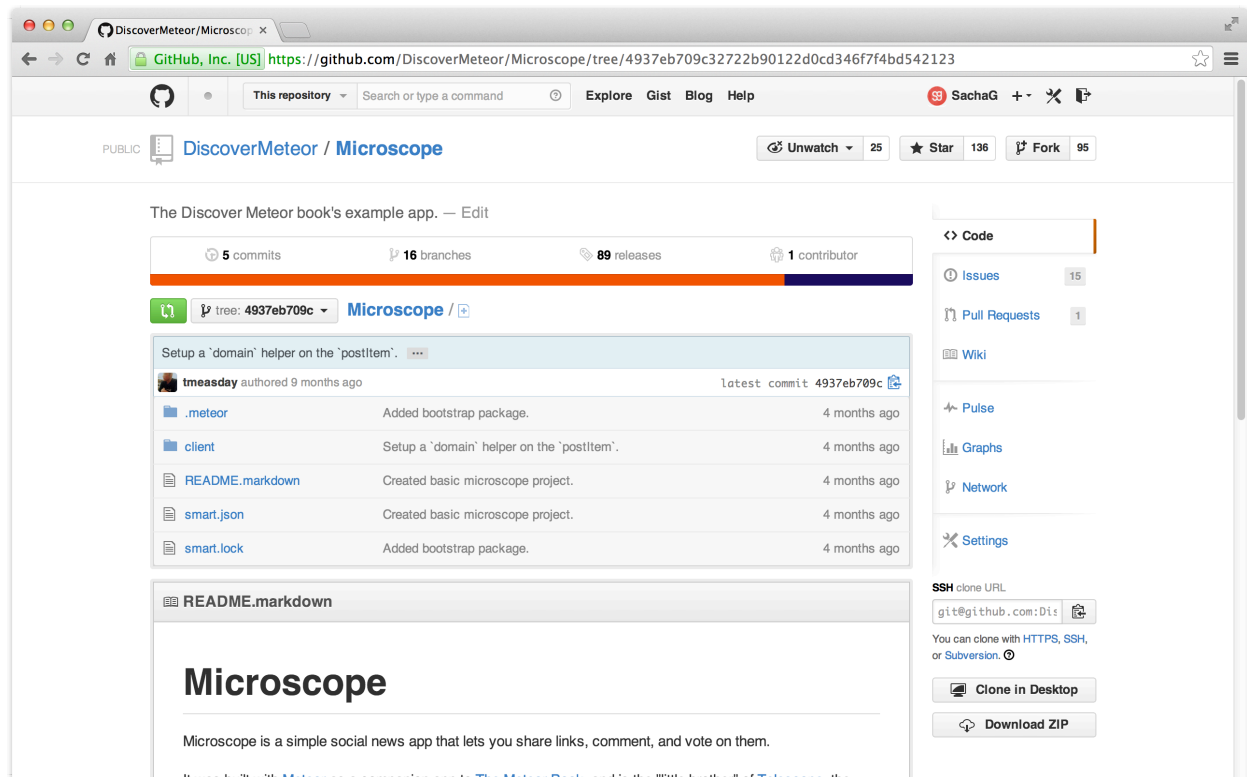
Git's commit view shows us the changes included in this commit, but sometimes you might want to look at files that *haven't* changed, just to make sure what their code is supposed to look like at this stage of the process.

Once again GitHub comes through for us. When you're on a commit page, click the **Browse code** button:



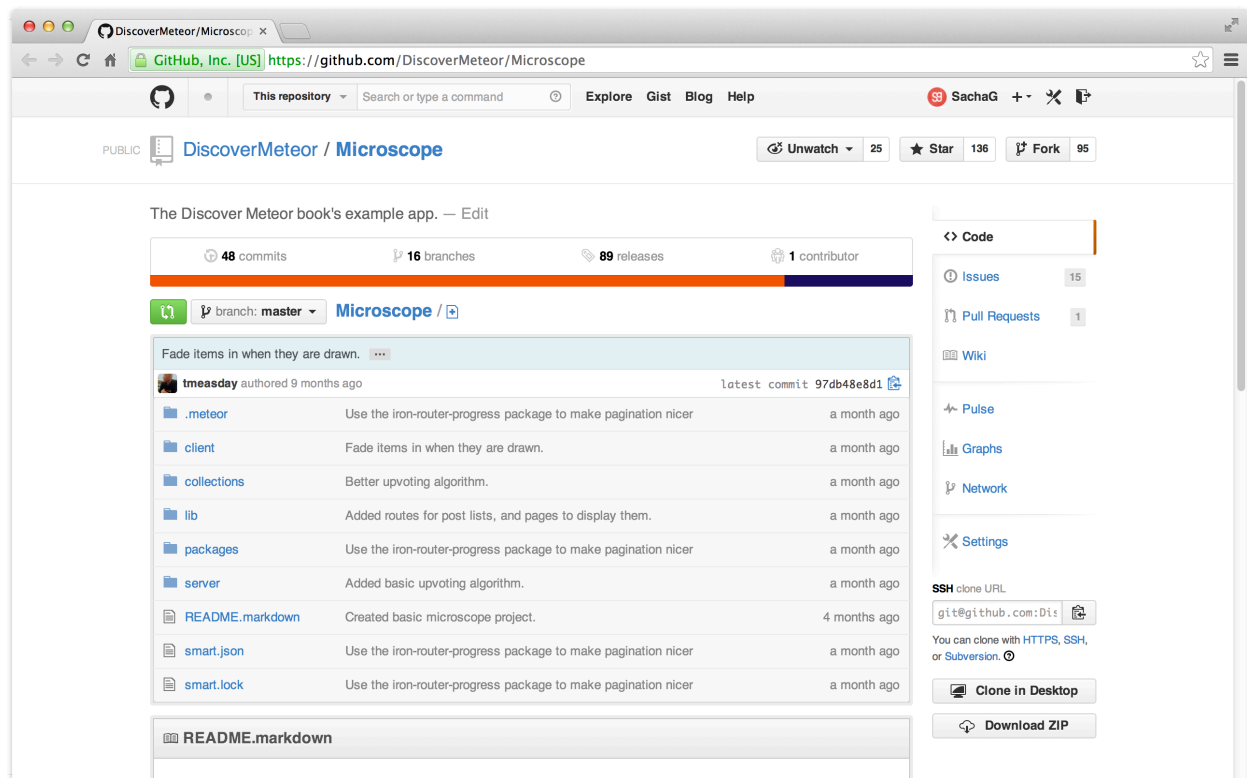
The Browse code button.

You'll now have access to the repo *as it stands at a specific commit*:



The repository at commit 3-2.

GitHub doesn't give us a lot of visual clues that we're looking at a commit, but you can compare with the "normal" master view and see at a glance that the file structure is different:



The repository at commit 14-2.

Accessing A Commit Locally

We've just seen how to browse a commit's entire code online on GitHub. But what if you want to do the same thing locally? For example, you might want to run the app locally at a specific commit to see how it's supposed to behave at this point in the process.

To do this, we'll take our first steps (well, in this book at least) with the `git` command line utility. For starters, **make sure you have Git installed**. Then **clone** (in other words, download a copy locally) the Microscope repository with:

```
git clone https://github.com/DiscoverMeteor/Microscope.git github_microscope
```

That `github_microscope` at the end is simply the name of the local directory you'll be cloning the app into. Assuming you already have a pre-existing `microscope` directory, just pick any different

name (it doesn't need to have the same name as the GitHub repo).

Let's `cd` into the repository so that we can start using the `git` command line utility:

```
cd github_microscope
```

Now when we cloned the repository from GitHub, we downloaded *all* the code of the app, which means we're looking at the code for the last ever commit.

Thankfully, there is a way to go back in time and “check out” a specific commit without affecting the other ones. Let's try it out:

```
git checkout chapter3-1
```

```
Note: checking out 'chapter3-1'.
```

You are **in** 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make **in** this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may **do** so (now or later) by using `-b` with the checkout `command` again. Example:

```
git checkout -b new_branch_name
```

```
HEAD is now at a004b56... Added basic posts list template and static data.
```

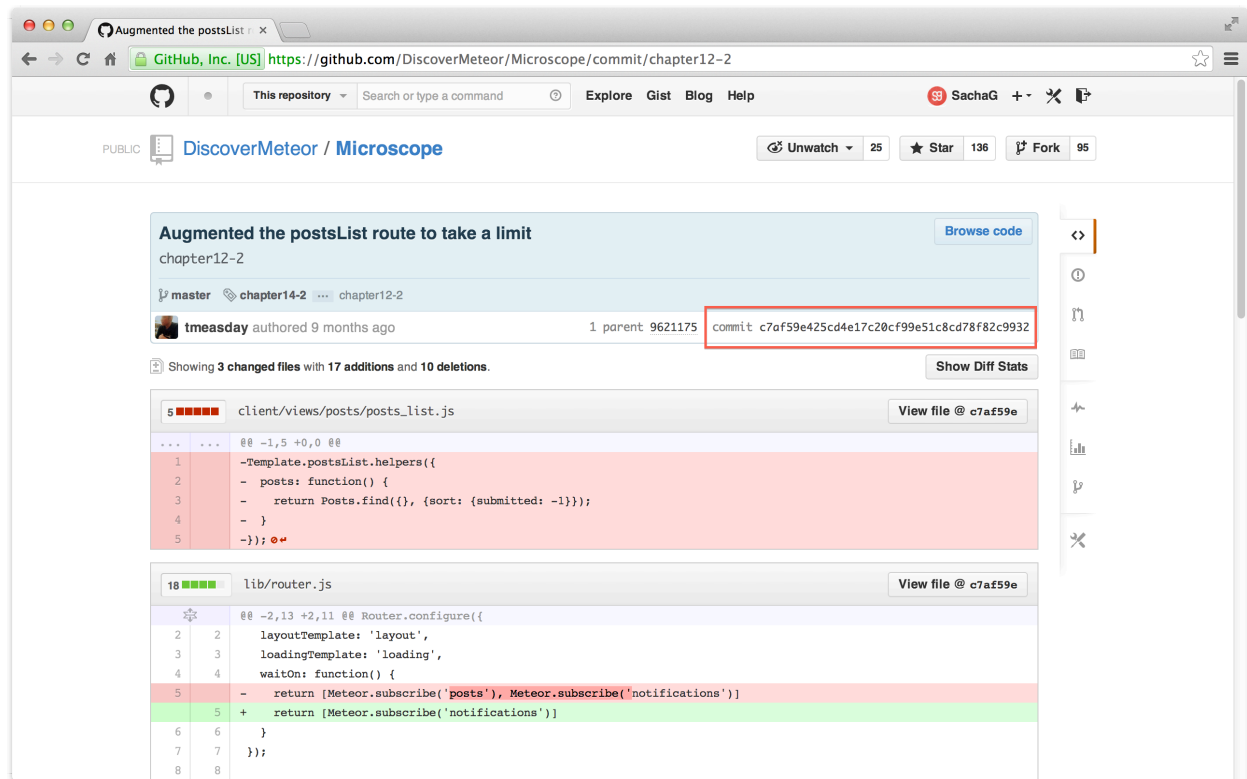
Git informs us that we are in “detached HEAD” state, which means that as far as Git is concerned, we can observe past commits but we can't modify them. You can think of it as a wizard inspecting the past through a crystal ball.

(Note that Git also has commands that let you *change* past commits. This would be more like a time traveller going back in time and possibly stepping on a butterfly, but it's outside the scope of this brief introduction.)

The reason why you were able to simply type `chapter3-1` is that we've pre-tagged all of Microscope's commits with the correct chapter marker. If this weren't the case, you'd need to first

find out the commit's **hash**, or unique identifier.

Once again, GitHub makes our life easier. You can find a commit's hash in the bottom right corner of the blue commit header box, as shown here:



Finding a commit hash.

So let's try it with the hash instead of a tag (if this specific hash doesn't work, feel free to get another one from GitHub):

```
git checkout b1280aa8affdb9f4ca5dab5f84d0f9878fc2f67d
Previous HEAD position was a004b56... Added basic posts list template and stati
c data.
HEAD is now at c7af59e... Augmented the postsList route to take a limit
```

And finally, what if we want to stop looking into our magic crystal ball and come back to the present? We tell Git that we want to check out the **master** branch:

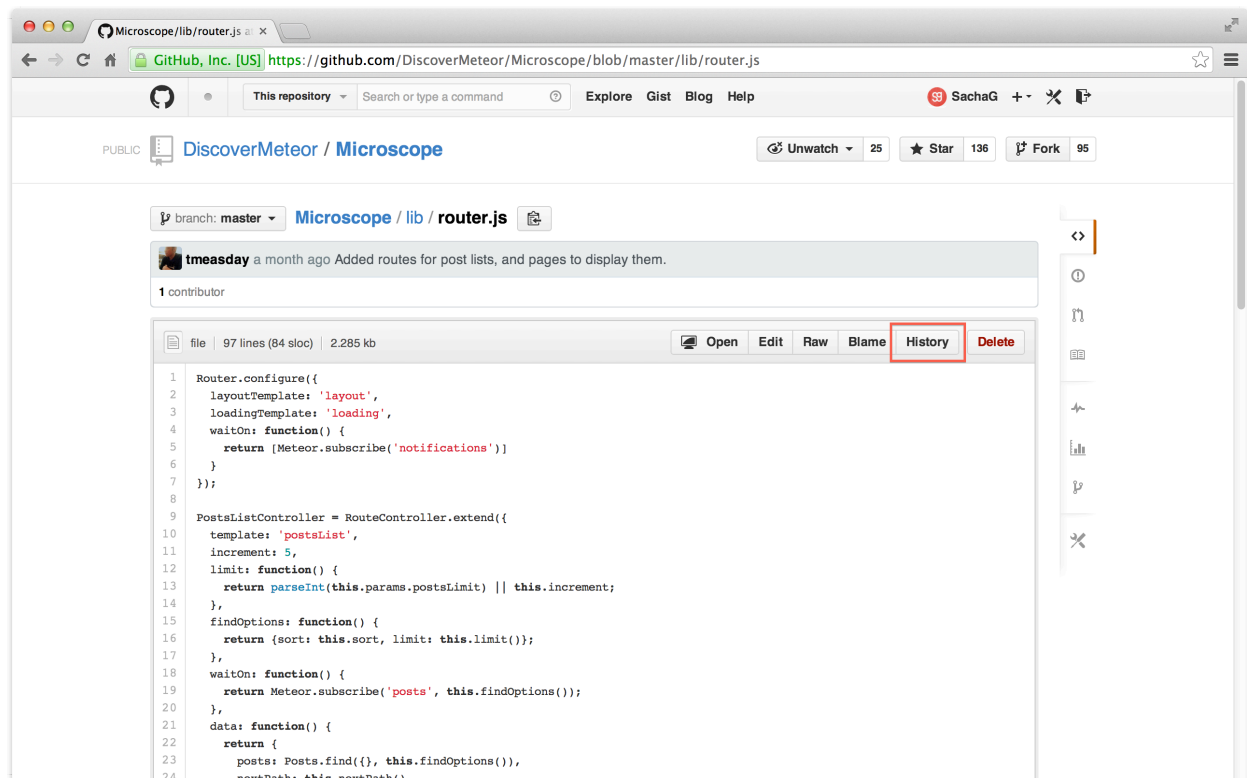
```
git checkout master
```

Note that you can also run the app with the `meteor` command at any point in the process, even when in “detached HEAD” state. You might need to run a quick `meteor update` first if Meteor complains about missing packages, since package code is not included in Microscope’s Git repo.

Historical Perspective

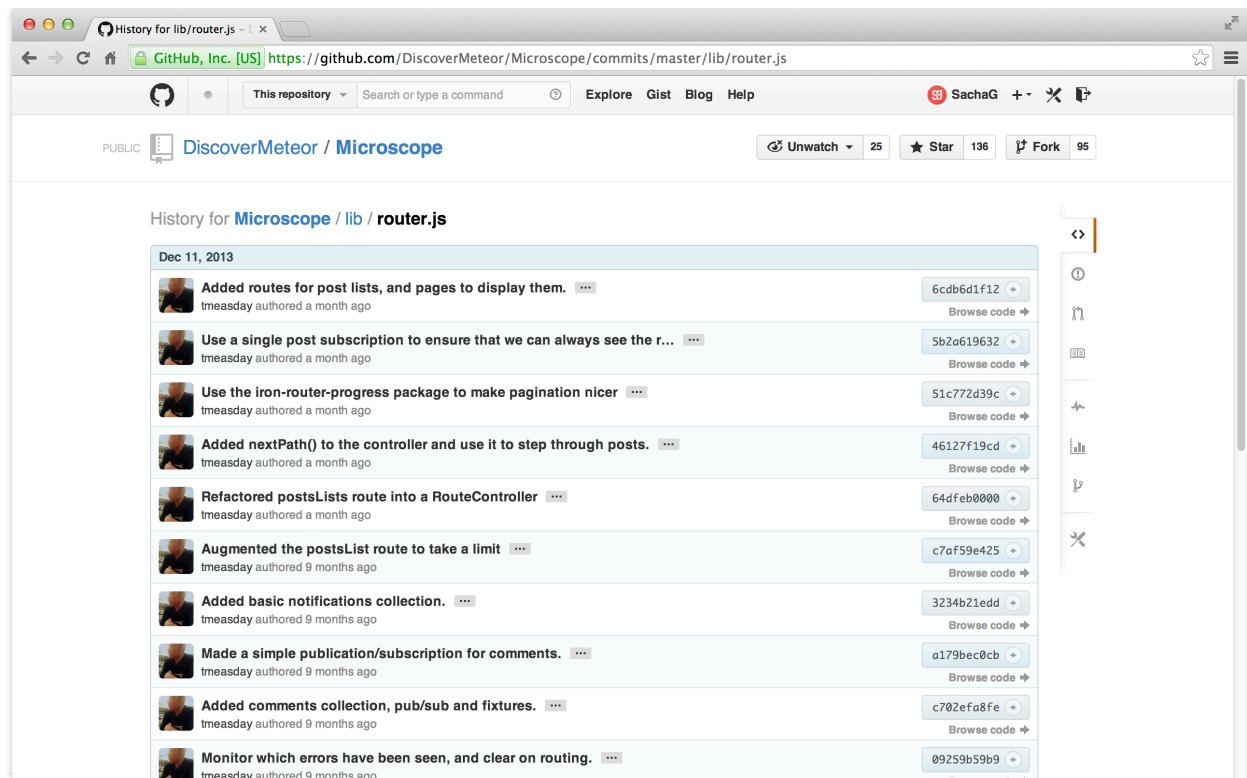
Here’s another common scenario: you’re looking at a file and notice some changes you hadn’t seen before. The thing is, you can’t remember *when* the file changed. You could just look at each commit one by one until you find the right one, but there’s an easier way thanks to GitHub’s **History** feature.

First, access one of your repository’s files on GitHub, then locate the “History” button:



GitHub's History button.

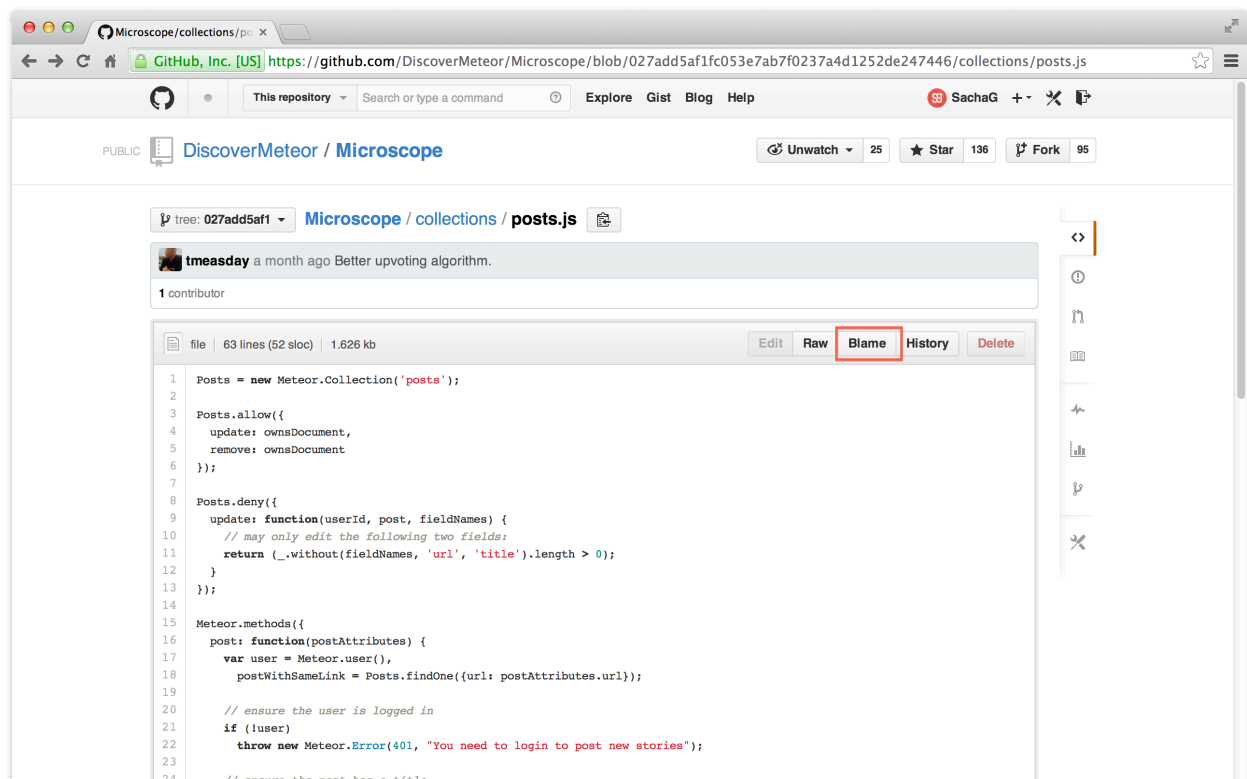
You now have a neat list of all the commits that affected this particular file:



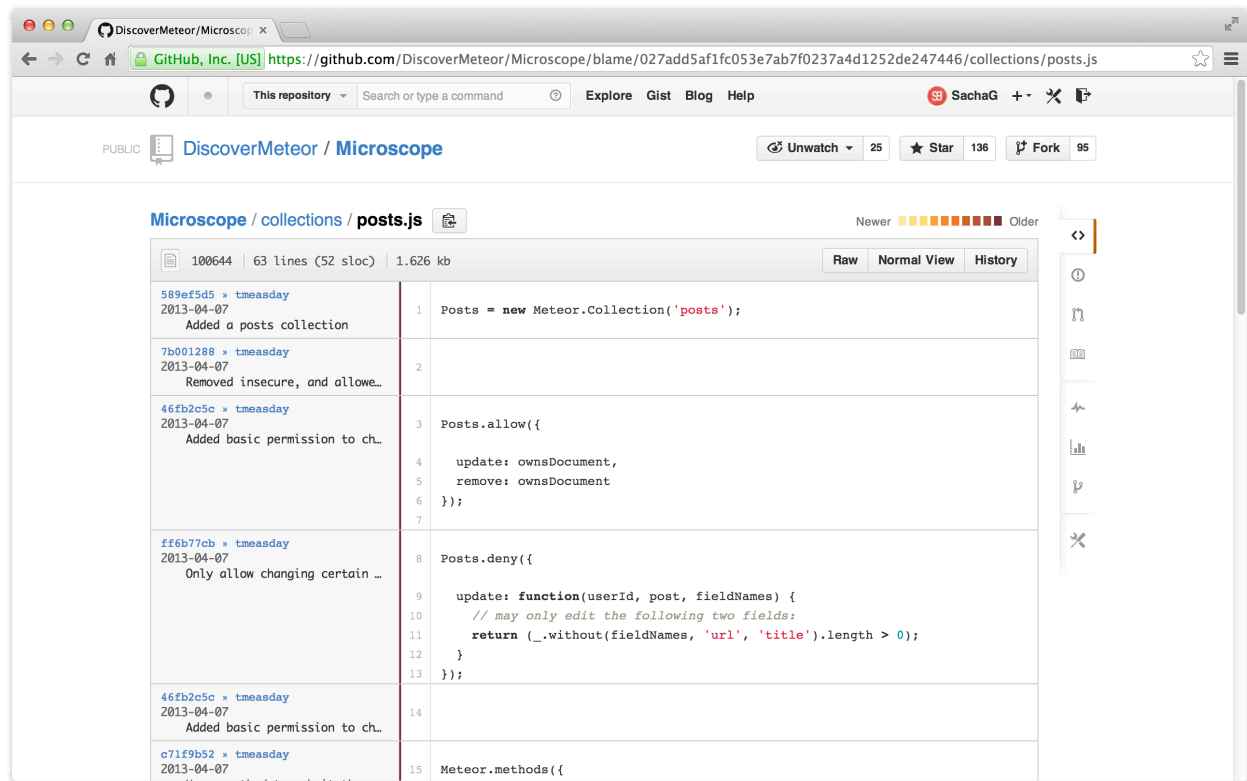
Displaying a file's history.

The Blame Game

To wrap things up, let's take a look at **Blame**:



This neat view shows us line by line who modified a file, and in which commit (in other words, who's to blame when things aren't working anymore):



Now Git is a fairly complex tool – and so is GitHub –, so we can't hope to cover everything in a single chapter. In fact, we've barely scratched the surface of what is possible with these tools. But hopefully, even that tiny bit will prove helpful as you follow along the rest of the book.

In chapter one, we spoke about the core feature of Meteor, the automatic synchronisation of data between client and server.

In this chapter, we'll take a closer look at how that works, and observe the operation of the key piece of technology that enables this, the Meteor **Collection**.

A collection is a special data structure that takes care of storing your data in the permanent, server-side MongoDB database, and then synchronising it with each connected user's browser in real-time.

We want our posts to be permanent and shared between users, so we'll start by creating a collection called `Posts` to store them in.

Collections are pretty central to any app, so to make sure they are always defined first we'll put them inside the `lib` directory. So if you haven't done so already, create a `collections/` folder inside `lib`, and then a `posts.js` file inside it. Then add:

```
Posts = new Mongo.Collection('posts');
```

`lib/collections/posts.js`

Commit 4-1

Added a posts collection

[View on GitHub](#)[Launch Instance](#)

To Var Or Not To Var?

In Meteor, the `var` keyword limits the scope of an object to the current file. Here, we want to make the `Posts` collection available to our whole app, which is why we're *not* using the `var` keyword.

Storing Data

Web apps have three basic ways of storing data at their disposal, each filling a different role:

- **The browser's memory:** things like JavaScript variables are stored in the browser's memory, which means they're not *permanent*: they're local to the current browser tab, and will disappear as soon as you close it.
- **The browser's storage:** browsers can also store data more permanently using cookies or **Local Storage**. Although this data will persist from session to session, it's *local* to the current user (but available across tabs) and can't be easily shared with other users.
- **The server-side database:** the best place for permanent data that you also want to make available to more than one user is in a good old database (MongoDB being the default solution for Meteor apps).

Meteor makes use of all three, and will sometimes synchronize data from one place to another (as we'll soon see). That being said, the database remains the “canonical” data source that contains the master copy of your data.

Client & Server

Code inside folders that are not `client/` or `server/` will run in *both* contexts. So the `Posts` collection is available to both client and server. However, what the collection does in each environment can be pretty different.

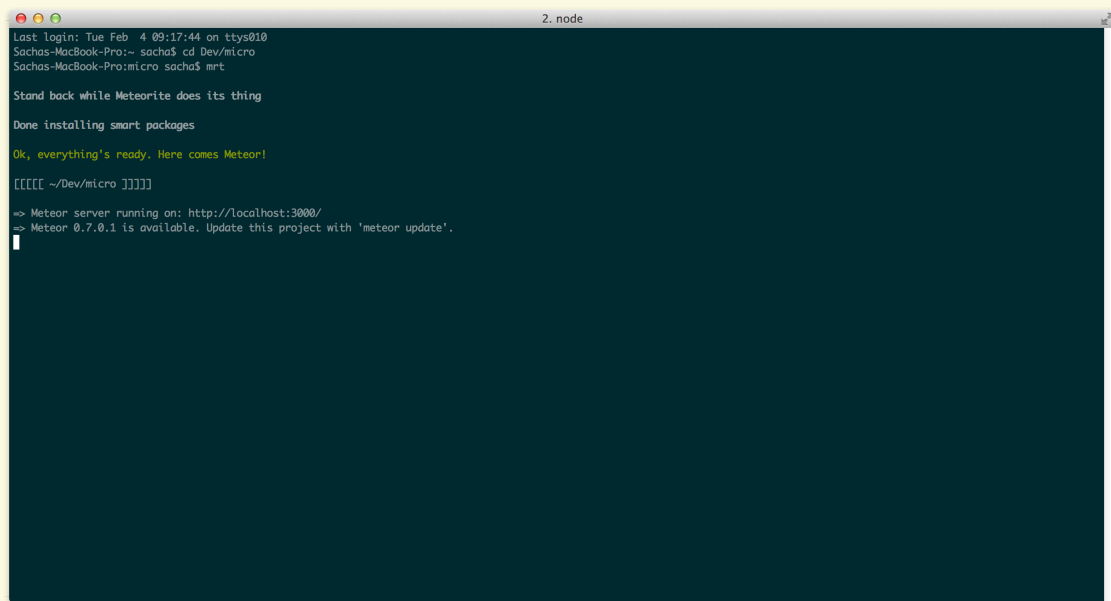
On the server, the collection has the job of talking to the MongoDB database, and reading and writing any changes. In this sense, it can be compared to a standard database library.

On the client however, the collection is a copy of a *subset* of the real, canonical collection. The client-side collection is constantly and (mostly) transparently kept up to date with that subset in real-time.

Console vs Console vs Console

In this chapter, we'll start making use of the **browser console**, which is not to be confused with the **terminal**, the **Meteor shell**, or the **Mongo shell**. Here's a quick primer on each of them.

Terminal



```
2. node
Last login: Tue Feb  4 09:17:44 on ttys010
Sachas-MacBook-Pro:~ sachas$ cd Dev/micro
Sachas-MacBook-Pro:micro sachas$ mrt

Stand back while Meteorite does its thing

Done installing smart packages

Ok, everything's ready. Here comes Meteor!

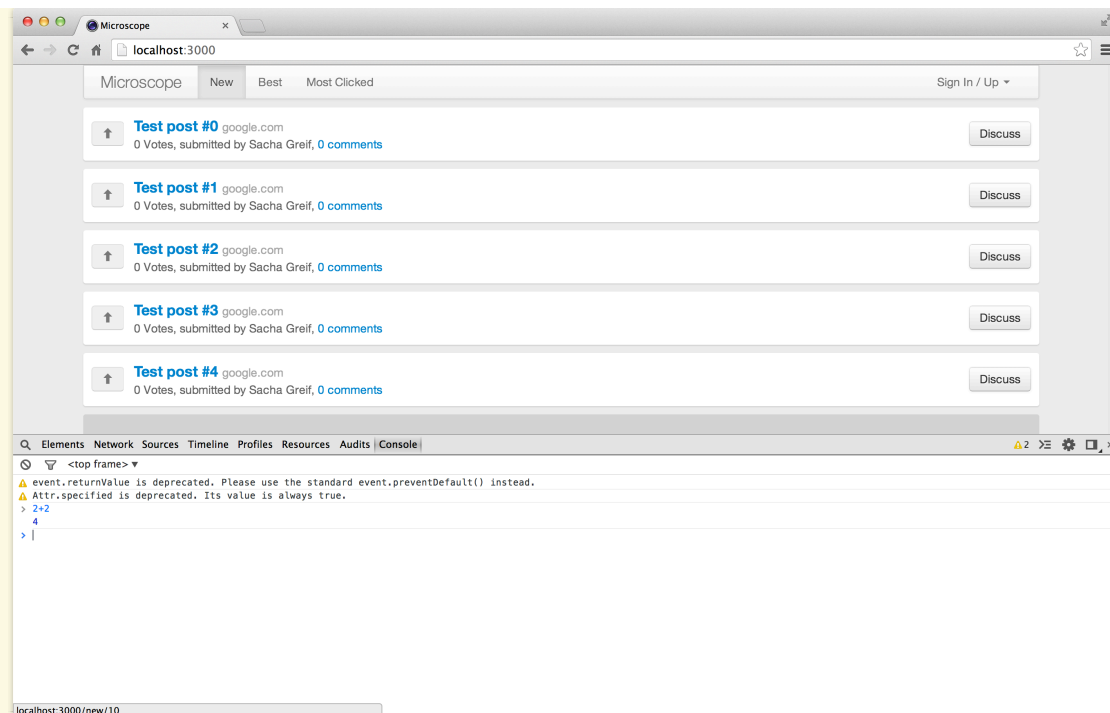
[[[[[ ~/Dev/micro ]]]]]

=> Meteor server running on: http://localhost:3000/
=> Meteor 0.7.0.1 is available. Update this project with 'meteor update'.
█
```

The Terminal

- Called from your operating system.
- **Server-side** `console.log()` calls output here.
- Prompt: `$`.
- Also known as: Shell, Bash

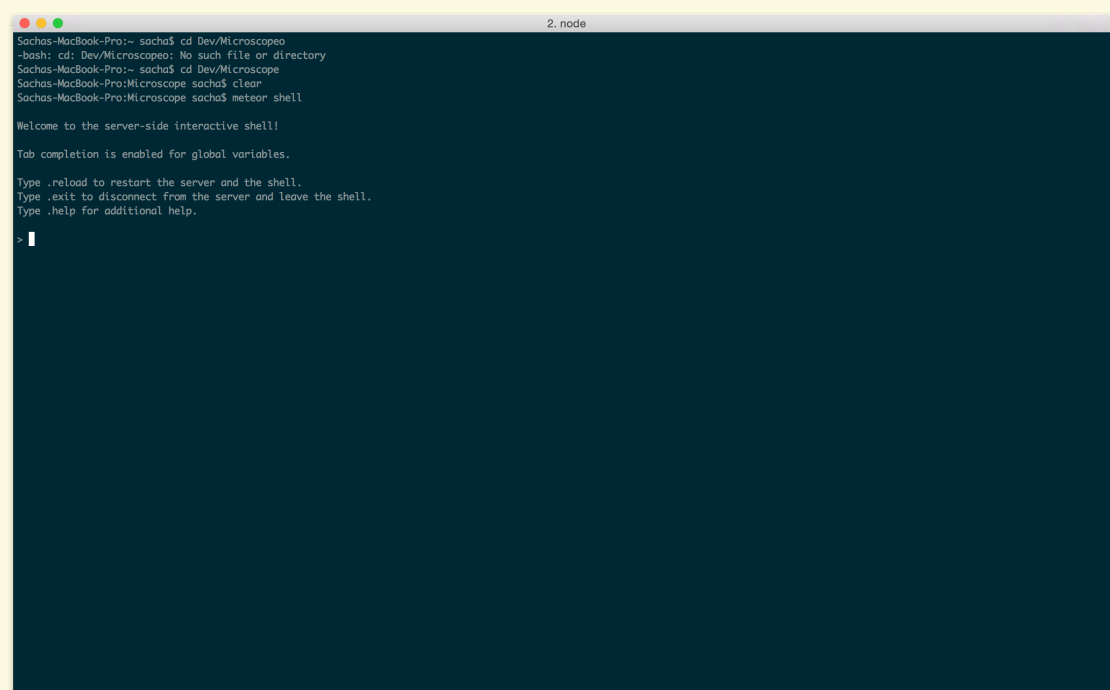
Browser Console



The Browser Console

- Called from inside the browser, executes JavaScript code.
- **Client-side** `console.log()` calls output here.
- Prompt: `>`.
- Also known as: JavaScript Console, DevTools Console

Meteor Shell



The Meteor Shell

- Called from the Terminal with `meteor shell`.
- Gives you direct access to your app's server-side code.
- Prompt: `>`.

Mongo Shell

```

node
node
2. node
Sachas-MacBook-Pro:~ sachas$ cd Dev/micro
Sachas-MacBook-Pro:micro sachas$ mrt mongo

Stand back while Meteorite does its thing

Done installing smart packages

Ok, everything's ready. Here comes Meteor!

MongoDB shell version: 2.4.4
connecting to: 127.0.0.1:3002/meteor
> db.posts.find()
{ "title": "Introducing Telescope", "userId": "3w4cWap35gyt2wb7M", "author": "Sacha Greif", "url": "http://sachagreif.com/introducing-telescope/", "submitted": 1391453182226, "commentsCount": 2, "upvoters": [ ], "votes": 0, "_id": "pSNipNMW4jdNEFvQio" }
{ "title": "Meteor", "userId": "b43n74hbK858CtY4j", "author": "Tom Coleman", "url": "http://meteor.com", "submitted": 1391442382226, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "3FesumbiJZda8nHB2" }
{ "title": "The Meteor Book", "userId": "b43n74hbK858CtY4j", "author": "Tom Coleman", "url": "http://themeteorbook.com", "submitted": 1391435182226, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "xJhuDhMLYuZhzESLa" }
{ "title": "Test post #0", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-0", "submitted": 1391478382227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "r29HT8hefJptaasY" }
{ "title": "Test post #1", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-1", "submitted": 1391474782227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "ZmbaE2L6CmsmNtnou" }
{ "title": "Test post #2", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-2", "submitted": 1391471182227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "NkEmnBRtpsaGHaFLY" }
{ "title": "Test post #3", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-3", "submitted": 1391467582227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "HHGETf6a2pRdMvsbk" }
{ "title": "Test post #4", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-4", "submitted": 1391463982227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "TmWBNQ6rF8nCX08da" }
{ "title": "Test post #5", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-5", "submitted": 1391460382227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "T9EKnxZ3aGgSap2EM" }
{ "title": "Test post #6", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-6", "submitted": 1391456782227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "iTKpRWyy8Mg4NBjWJX" }
{ "title": "Test post #7", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-7", "submitted": 1391453182227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "FKRKYQ9pDxfuLEfzS" }
{ "title": "Test post #8", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-8", "submitted": 1391449582227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "rQWhxzyCyHjy3iHNS" }
{ "title": "Test post #9", "author": "Sacha Greif", "userId": "3w4cWap35gyt2wb7M", "url": "http://google.com/?q=test-9", "submitted": 1391445982227, "commentsCount": 0, "upvoters": [ ], "votes": 0, "_id": "cFFPvrvsRtLap2q2M" }
>

```

The Mongo Shell

- Called from the Terminal with `meteor mongo`.
- Gives you direct access to your app's database.
- Prompt: `>`.
- Also know as: Mongo Console

Note that in each case, you're not supposed to type the prompt character (`$`, `>`, or `>`) as part of the command. And you can assume that any line *not* beginning with the prompt is the output of the preceding command.

Server-Side Collections

Going back to the server, the collection acts as an API into your Mongo database. In your server-side code, this allows you to write Mongo commands like `Posts.insert()` or `Posts.update()`, and they will make changes to the `posts` collection stored inside Mongo.

To look inside the Mongo database, open up a second terminal window (while `meteor` is still

running in your first), and go to your app's directory. Then, run the command `meteor mongo` to initiate a Mongo shell, into which you can type standard Mongo commands (and as usual, you can quit it with the `ctrl+c` keyboard shortcut). For example, let's insert a new post:

```
meteor mongo

> db.posts.insert({title: "A new post"});

> db.posts.find();
{ "_id": ObjectId("..."), "title" : "A new post" };
```

The Mongo Shell

Mongo on Meteor.com

Note that when hosting your app on `*.meteor.com`, you can also access your deployed app's Mongo shell with `meteor mongo myApp`.

And while we're at it, you can also get your app's logs by typing `meteor logs myApp`.

Mongo's syntax is familiar, as it uses a JavaScript interface. We won't be doing any further data manipulation in the Mongo shell, but we might take a peek inside from time to time just to make sure what's in there.

Client-Side Collections

Collections get more interesting client-side. When you declare `Posts = new Mongo.Collection('posts');` on the client, what you are creating is a *local, in-browser cache* of the real Mongo collection. When we talk about a client-side collection being a “cache”, we mean it in the sense that it contains a *subset* of your data, and offers very *quick* access to this data.

It's important to understand these points as it's fundamental to the way Meteor works. In general, a client side collection consists of a subset of all the documents stored in the Mongo collection (after all, we generally don't want to send our *whole* database to the client).

Secondly, those documents are stored *in browser memory*, which means that accessing them is basically instantaneous. So there are no slow trips to the server or the database to fetch the data when you call `Posts.find()` on the client, as the data is already pre-loaded.

Introducing MiniMongo

Meteor's client-side Mongo implementation is called MiniMongo. It's not a perfect implementation yet, and you may encounter occasional Mongo features that don't work in MiniMongo. Nevertheless, all the features we cover in this book work similarly in both Mongo and MiniMongo.

Client-Server Communication

The key piece of all this is how the client-side collection synchronizes its data with the server-side collection of the same name (`'posts'` in our case).

Rather than explaining this in detail, let's just watch what happens.

Start by opening up two browser windows, and accessing the browser console in each one. Then, open up the Mongo shell on the command line.

At this point, we should be able to find the single document we created earlier in all three contexts (note that our app's *user interface* is still displaying our previous three dummy posts. Just ignore them for now).

```
> db.posts.find();  
{title: "A new post", _id: ObjectId("...")};
```

The Mongo Shell

```
> Posts.findOne();  
{title: "A new post", _id: LocalCollection._ObjectID};
```

First browser console

Let's create a new post. In one of the browser windows, run an insert command:

```
> Posts.find().count();  
1  
> Posts.insert({title: "A second post"});  
'xxx'  
> Posts.find().count();  
2
```

First browser console

Unsurprisingly, the post made it into the local collection. Now let's check Mongo:

```
> db.posts.find();  
{title: "A new post", _id: ObjectId("..")};  
{title: "A second post", _id: 'yyy'};
```

The Mongo Shell

As you can see, the post made it all the way back to the Mongo database, without us writing a single line of code to hook our client up to the server (well, strictly speaking, we did write a *single* line of code: `new Mongo.Collection('posts')`). But that's not all!

Bring up the second browser window and enter this in the browser console:

```
> Posts.find().count();  
2
```

Second browser console

The post is there too! Even though we never refreshed or even interacted with the second browser, and we certainly didn't write any code to push updates out. It all happened magically – and instantly too, although this will become more obvious later.

What happened is that our server-side collection was informed by a client collection of a new post, and took on the task of distributing that post into the Mongo database and back out to all the other connected `post` collections.

Fetching posts on the browser console isn't that useful. We will soon learn how to wire this data into our templates, and in the process turn our simple HTML prototype into a functioning realtime web application.

Populating the Database

Looking at the contents of our Collections on the browser console is one thing, but what we'd really like to do is display the data, and the changes to that data, on the screen. In doing so we'll turn our app from a simple web *page* displaying static data, to a real-time web *application* with dynamic, changing data.

The first thing we'll do is put some data into the database. We'll do so with a fixture file that loads a set of structured data into the `Posts` collection when the server first starts up.

First, let's make sure there's nothing in the database. We'll use `meteor reset`, which erases your database and resets your project. Of course, you'll want to be very careful with this command once you start working on real-world projects.

Stop the Meteor server (by pressing `ctrl-c`) and then, on the command line, run:

```
meteor reset
```

The reset command completely clears out the Mongo database. It's a useful command in development, where there's a strong possibility of our database falling into an inconsistent state.

Let's start our Meteor app again:

```
meteor
```

Now that the database is empty, we can add the following code that will load up three posts whenever the server starts, as long as the `Posts` collection is empty:

```
if (Posts.find().count() === 0) {  
  Posts.insert({  
    title: 'Introducing Telescope',  
    url: 'http://sachagreif.com/introducing-telescope/'  
  });  
  
  Posts.insert({  
    title: 'Meteor',  
    url: 'http://meteor.com'  
  });  
  
  Posts.insert({  
    title: 'The Meteor Book',  
    url: 'http://themetorbook.com'  
  });  
}
```

```
server/fixtures.js
```

Commit 4-2

Added data to the posts
collection.

[View on GitHub](#)[Launch Instance](#)

We've placed this file in the `server/` directory, so it will never get loaded on any user's browser. The code will run immediately when the server starts, and make `insert` calls on the database to add three simple posts in our `Posts` collection.

Now run your server again with `meteor`, and these three posts will get loaded into the database.

Dynamic Data

If we open up a browser console, we'll see all three posts loaded up into MiniMongo:

```
> Posts.find().fetch();
```

Browser console

To get these posts into rendered HTML, we'll use our friend the template helper.

In Chapter 3 we saw how Meteor allows us to bind a *data context* to our Spacebars templates to build HTML views of simple data structures. We can bind in our collection data in the exact same way. We'll just replace our static `postsData` JavaScript object by a dynamic collection.

Speaking of which, feel free to delete the `postsData` code at this point. Here's what `posts_list.js` should now look like:

```
Template.postsList.helpers({  
  posts: function() {  
    return Posts.find();  
  }  
});
```

client/templates/posts/posts_list.js

Commit 4-3

Wired collection into `postsList` template.

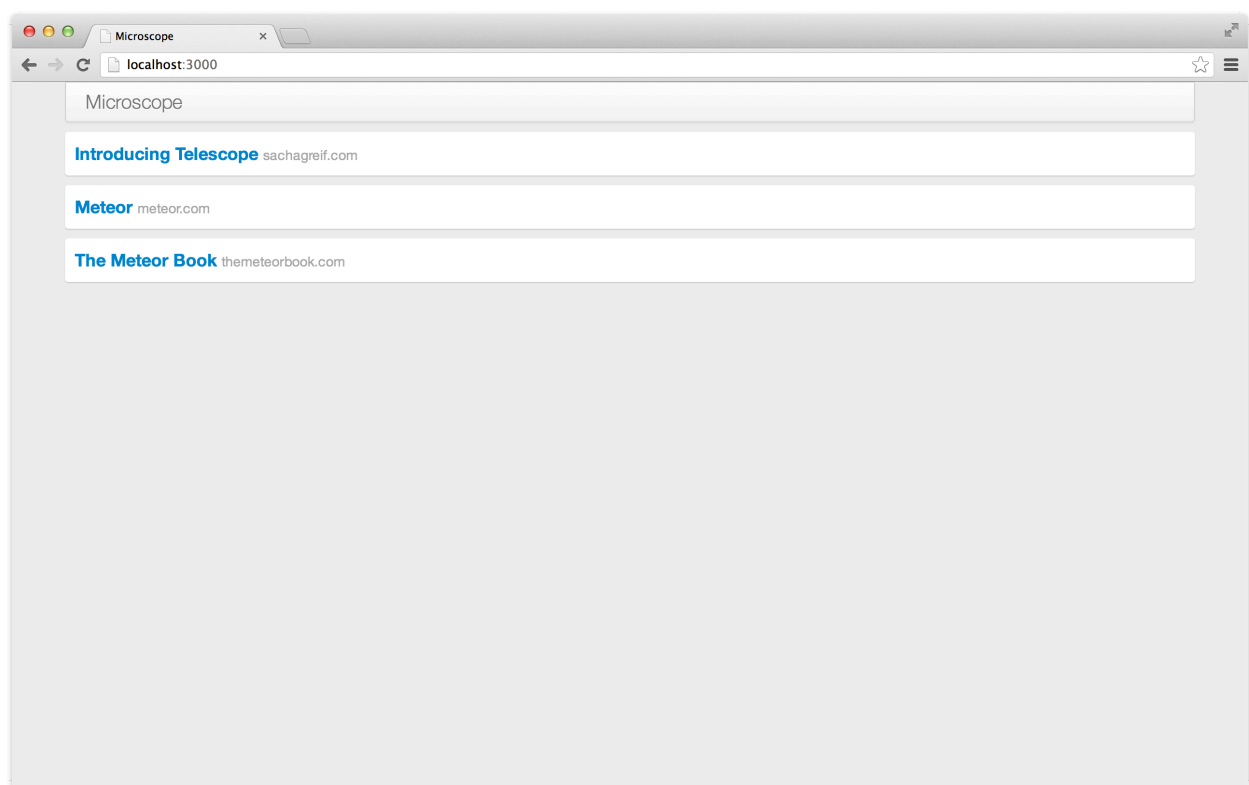
[View on GitHub](#)[Launch Instance](#)

Find & Fetch

In Meteor, `find()` returns a *cursor*, which is a **reactive data source**. When we want to log its contents, we can then use `fetch()` on that cursor to transform it into an array.

Within an app, Meteor is smart enough to know how to iterate over cursors without having to explicitly convert them into arrays first. This is why you won't see `fetch()` that often in actual Meteor code (and why we didn't use it in the above example).

Rather than pulling a list of posts as a static array from a variable, we're now returning a cursor to our `posts` helper (although things won't look very different since we're still using the same data):



Using live data

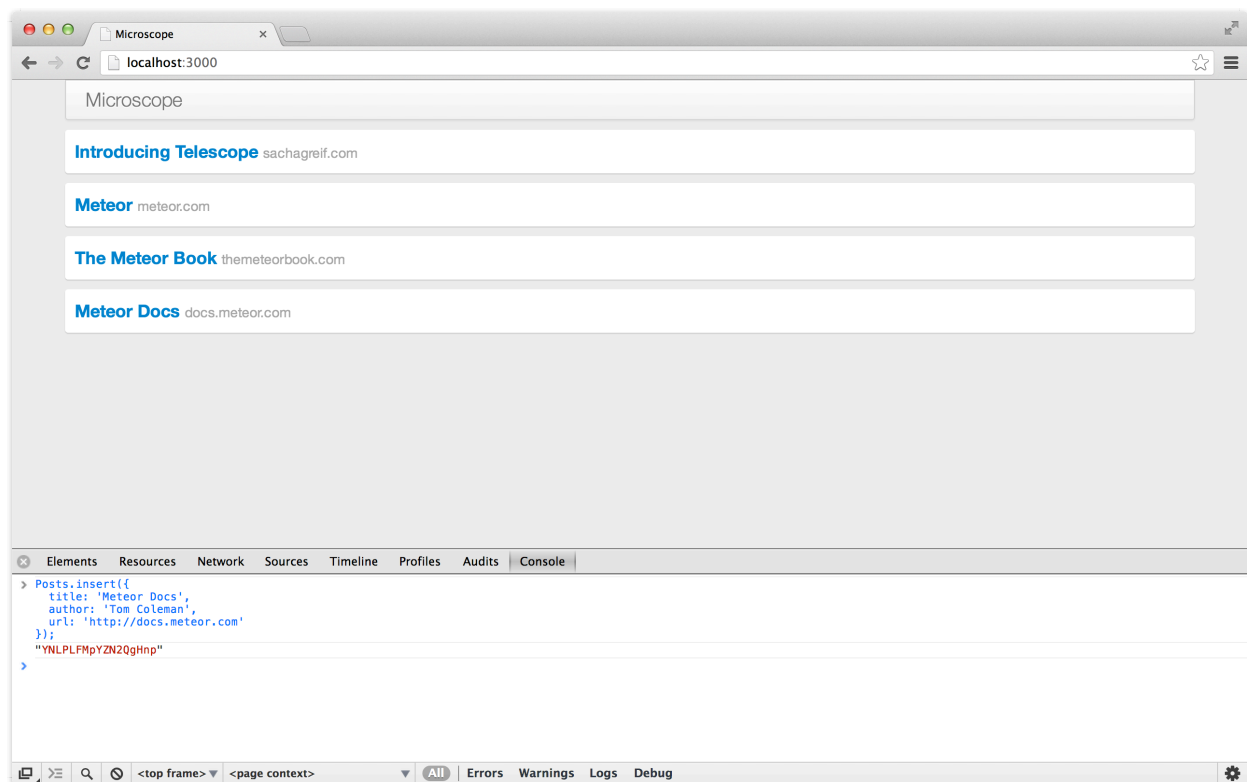
Our `{{#each}}` helper has iterated over all of our `Posts` and displayed them on the screen. The server-side collection pulled the posts from Mongo, passed them over the wire to our client-side collection, and our Spacebars helper passed them into the template.

Now, we'll take this one step further; let's add another post via the console:

```
> Posts.insert({
  title: 'Meteor Docs',
  author: 'Tom Coleman',
  url: 'http://docs.meteor.com'
});
```

Browser console

Look back at the browser – you should see this:



Adding posts via the console

You have just seen reactivity in action for the first time. When we told Spacebars to iterate over the `Posts.find()` cursor, it knew how to observe that cursor for changes, and patch the HTML in the simplest way to display the correct data on screen.

Inspecting DOM Changes

In this case, the simplest change possible was to add another `<div class="post">...</div>`. If you want to make sure this is really what happened, open the DOM inspector and select the `<div>` corresponding to one of the existing posts.

Now, in the JavaScript console, insert another post. When you tab back to the inspector, you'll see an extra `<div>`, corresponding to the new post, but you will still have the *same* existing `<div>` selected. This is a useful way to tell when elements have been re-rendered and when they have been left alone.

Connecting Collections: Publications and Subscriptions

So far, we've had the `autopublish` package enabled, which is not intended for production applications. As its name indicates, this package simply says that each collection should be shared in its entirety to each connected client. This isn't what we really want, so let's turn it off.

Open a new terminal window, and type:

```
meteor remove autopublish
```

This has an instant effect. If you look in your browser now, you'll see that all our posts have disappeared! This is because we were relying on `autopublish` to make sure our client-side collection of posts was a mirror of all the posts in the database.

Eventually we'll need to make sure we're only transferring the posts that the user actually needs to see (taking into account things like pagination). But for now, we'll just setup `Posts` to be published in its entirety.

To do so, we create a `publish()` function that returns a cursor referencing all posts:

```
Meteor.publish('posts', function() {  
  return Posts.find();  
});
```

server/publications.js

In the client, we need to *subscribe* to the publication. We'll just add the following line to `main.js`:

```
Meteor.subscribe('posts');
```

client/main.js

Commit 4-4

Removed `autopublish` and set up a basic publication.

[View on GitHub](#)[Launch Instance](#)

If we check the browser again, our posts are back. Phew!

Conclusion

So what have we achieved? Well, although we don't have a user interface yet, what we have now is a functional web application. We could deploy this application to the Internet, and (using the browser console) start posting new stories and see them appear in other user's browsers all over the world.

Publications and subscriptions are one of the most fundamental and important concepts in Meteor, but can be hard to wrap your head around when you're just getting started.

This has led to a lot of misunderstandings, such as the belief that Meteor is insecure, or that Meteor apps can't deal with large amounts of data.

A big part of the reason people find these concepts a bit confusing initially is the “magic” that Meteor does for us. Although this magic is ultimately very useful, it can obscure what's really going on behind the scenes (as magic tends to do). So let's strip away the layers of magic to try and understand what's happening.

The Olden Days

But first, let's take a look back at the good old days of 2011 when Meteor wasn't yet around. Let's say you're building a simple Rails app. When a user hits your site, the client (i.e. your browser) sends a request to your app, which is living on the server.

The app's first job is to figure out what data the user needs to see. This could be page 12 of search results, Mary's user profile information, Bob's 20 latest tweets, and so on. You can basically think of it as a bookstore clerk browsing through the aisles for the book you asked for.

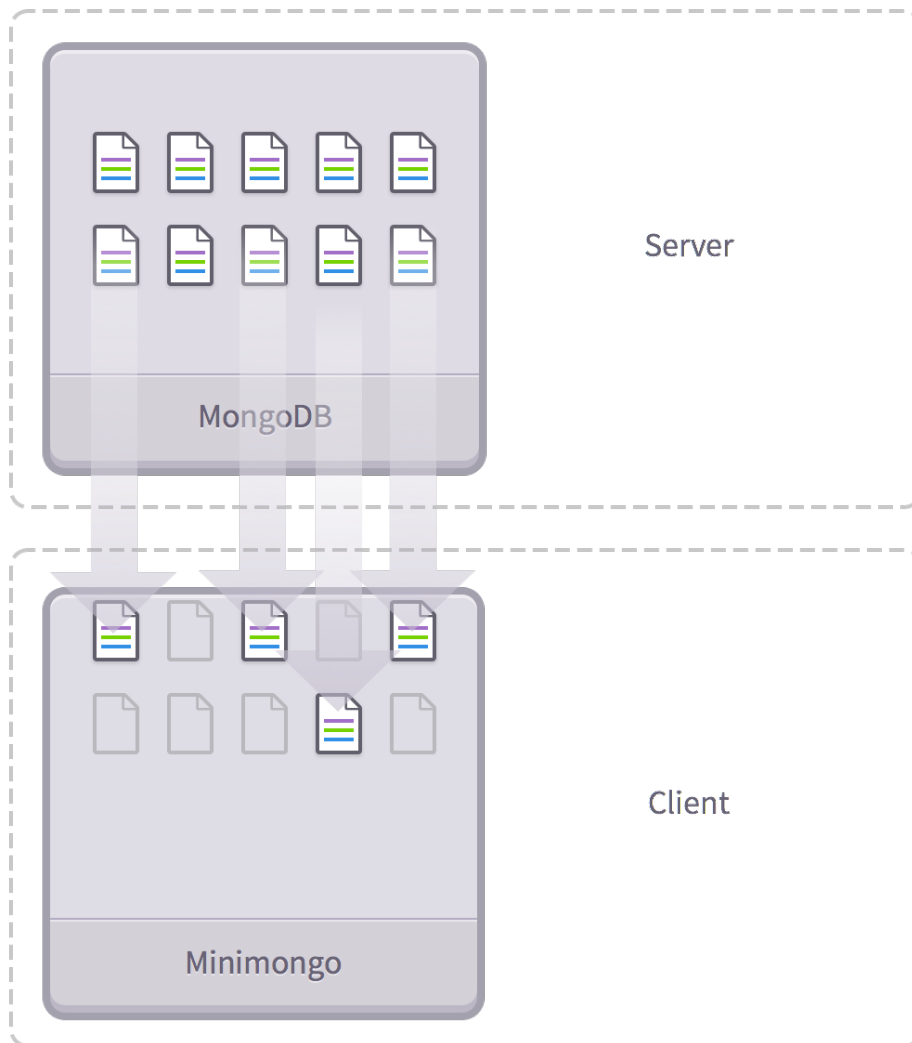
Once the right data has been selected, the app's second job is translating that data into nice, human-readable HTML (or JSON in the case of an API).

In the bookstore metaphor, that would be wrapping up the book you just bought and putting it in a nice bag. This is the “View” part of the famous Model-View-Controller model.

Finally, the app takes that HTML code and sends it over to the browser. The app's job is done, and now that the whole thing is out of its virtual hands it can just kick back with a beer while waiting for the next request.

The Meteor Way

Let's review what makes Meteor so special in comparison. As we've seen, the key innovation of Meteor is that where a Rails app only lives **on the server**, a Meteor app also includes a client-side component that will run **on the client** (the browser).



Pushing a subset of the database to the client.

This is like a store clerk who not only finds the right book for you, but also follows you home to read it to you at night (which we'll admit does sound a bit creepy).

This architecture lets Meteor do many cool things, chief among them what Meteor calls **database everywhere**. Simply put, Meteor will take a subset of your database and *copy it to the client*.

This has two big implications: first, instead of sending HTML code to the client, a Meteor app will send **the actual, raw data** and let the client deal with it (**data on the wire**). Second, you'll be able to **access and even modify that data instantaneously** without having to wait for a round-trip to the server (**latency compensation**).

Publishing

An app's database can contain tens of thousands of documents, some of which might even be private or sensitive. So we obviously shouldn't just mirror our whole database on the client, for security and scalability reasons.

So we'll need a way to tell Meteor which **subset** of data can be sent to the client, and we'll accomplish this through a **publication**.

Let's go back and use Microscope as an example. Here are all of our app's posts sitting in the database:



All the posts contained in our database.

Although that feature admittedly does not actually exist in Microscope, we'll imagine that some of

our posts have been flagged for abusive language. Although we want to keep them in our database, they should not be made available to users (i.e. sent to a client).

Our first task will be telling Meteor what data we *do* want to send to the client. We'll tell Meteor we only want to **publish** unflagged posts:



Excluding flagged posts.

Here's the corresponding code, which would reside on the server:

```
// on the server
Meteor.publish('posts', function() {
  return Posts.find({flagged: false});
});
```

This ensures there is **no possible way** that a client will be able to access a flagged post. This is exactly how you'd make a Meteor app secure: just ensure you're only publishing data you want the current client to have access to.

By the way, note that the code snippets contained in this chapter are only provided to illustrate the

concepts covered here, and shouldn't be taken literally as part of Microscope's codebase.

DDP

Fundamentally, you can think of the publication/subscription system as a funnel that transfers data from a server-side (source) collection to a client-side (target) collection.

The protocol that is spoken over that funnel is called **DDP** (which stands for Distributed Data Protocol). To learn more about DDP, you can watch [this talk from The Real-time Conference](#) by Matt DeBergalis (one of the founders of Meteor), or [this screencast](#) by Chris Mather that walks you through this concept in a little more detail.

Subscribing

Even though we want to make any non-flagged post available to clients, we can't just send thousands of posts at once. We need a way for clients to specify which subset of that data they need at any particular moment, and that's exactly where **subscriptions** come in.

Any data you subscribe to will be **mirrored** on the client thanks to Minimongo, Meteor's client-side implementation of MongoDB.

For example, let's say we're currently browsing Bob Smith's profile page, and only want to display *his* posts.



Subscribing to Bob's posts will mirror them on the client.

First, we would amend our publication to take a parameter:

```
// on the server
Meteor.publish('posts', function(author) {
  return Posts.find({flagged: false, author: author});
});
```

And we would then define that parameter when we *subscribe* to that publication in our app's client-side code:

```
// on the client
Meteor.subscribe('posts', 'bob-smith');
```

This is how you make a Meteor app scalable client-side: instead of subscribing to *all* available data, just pick and choose the parts that you currently need. This way, you'll avoid overloading the browser's memory no matter how big your server-side database is.

Finding

Now Bob's posts happen to be spread across multiple categories (for example: "JavaScript", "Ruby", and "Python"). Maybe we still want to load all of Bob's posts in memory, but we only want to display those from the "JavaScript" category right now. This is where "finding" comes in.



Selecting a subset of documents on the client.

Just like we did on the server, we'll use the `Posts.find()` function to select a subset of our data:

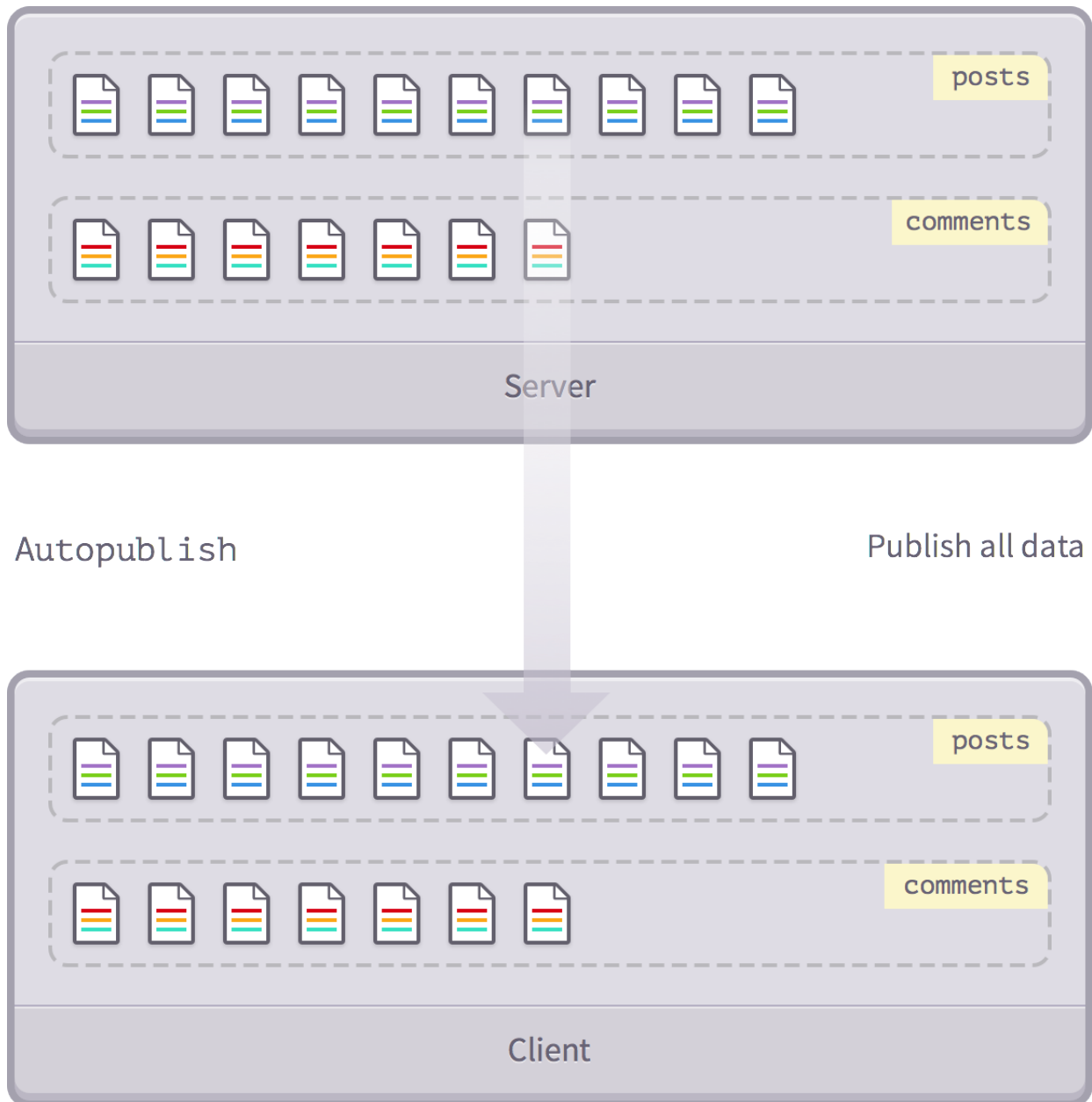
```
// on the client
Template.posts.helpers({
  posts: function(){
    return Posts.find({author: 'bob-smith', category: 'JavaScript'});
  }
});
```

Now that we have a good grasp of what role publications and subscriptions play, let's dig in deeper and review a few common implementation patterns.

Autopublish

If you create a Meteor project from scratch (i.e using `meteor create`), it will automatically have the `autopublish` package enabled. As a starting point, let's talk about what that does exactly.

The goal of `autopublish` is to make it very easy to get started coding your Meteor app, and it does this by automatically mirroring *all data* from the server on the client, thus taking care of publications and subscriptions for you.



Autopublish

How does this work? Suppose you have a collection called `'posts'` on the server. Then

`autopublish` will automatically send every post that it finds in the Mongo posts collection into a collection called `'posts'` on the client (assuming there is one).

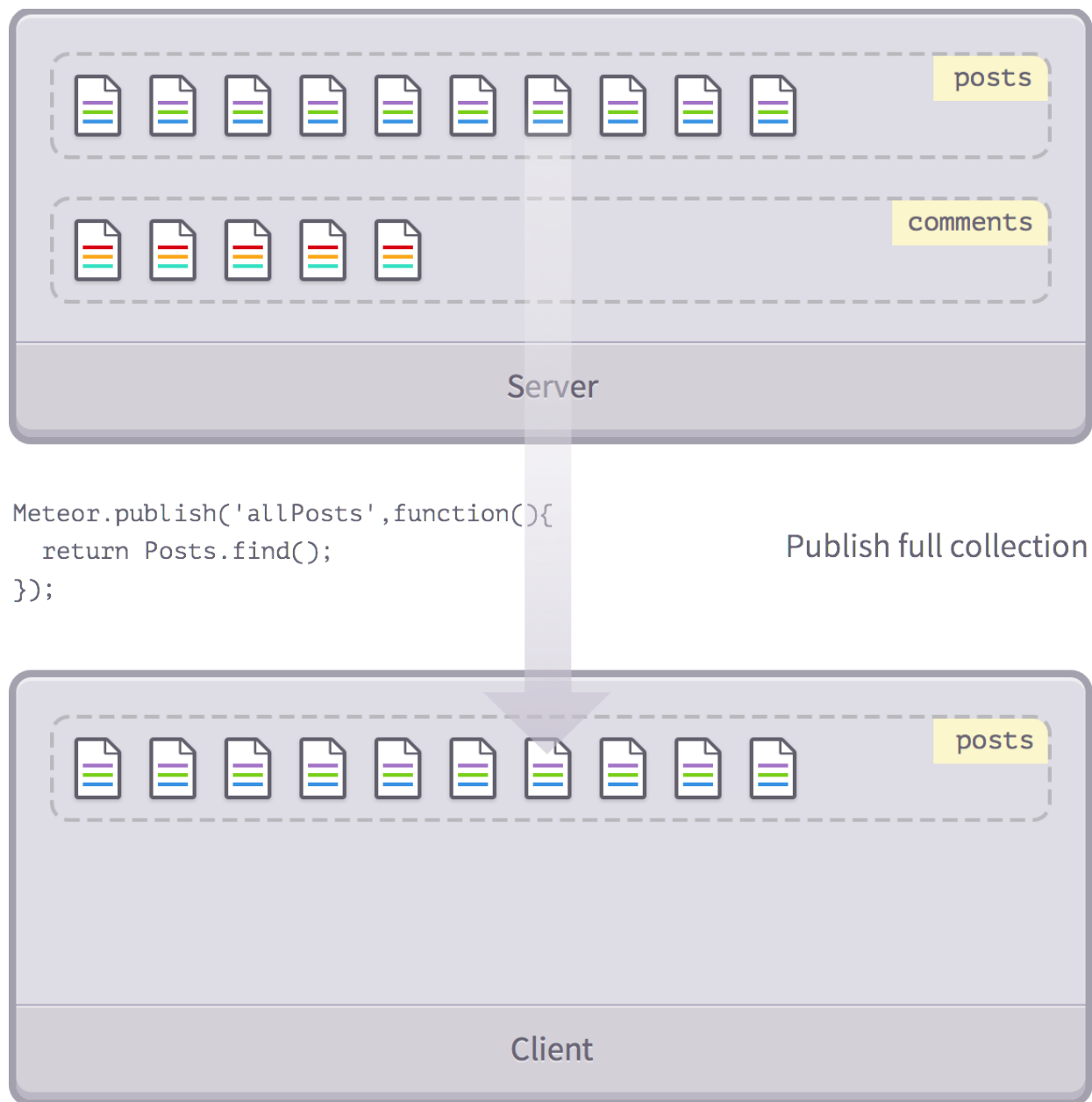
So if you are using `autopublish`, you don't need to think about publications. Data is ubiquitous, and things are simple. Of course, there are obvious problems with having a complete copy of your app's database cached on every user's machine.

For this reason, `autopublish` is only appropriate when you are starting out, and haven't yet thought about publications.

Publishing Full Collections

Once you remove `autopublish`, you'll quickly realize that all your data has vanished from the client. An easy way to get it back is to simply duplicate what `autopublish` does, and publish a collection in its entirety. For example:

```
Meteor.publish('allPosts', function(){  
  return Posts.find();  
});
```

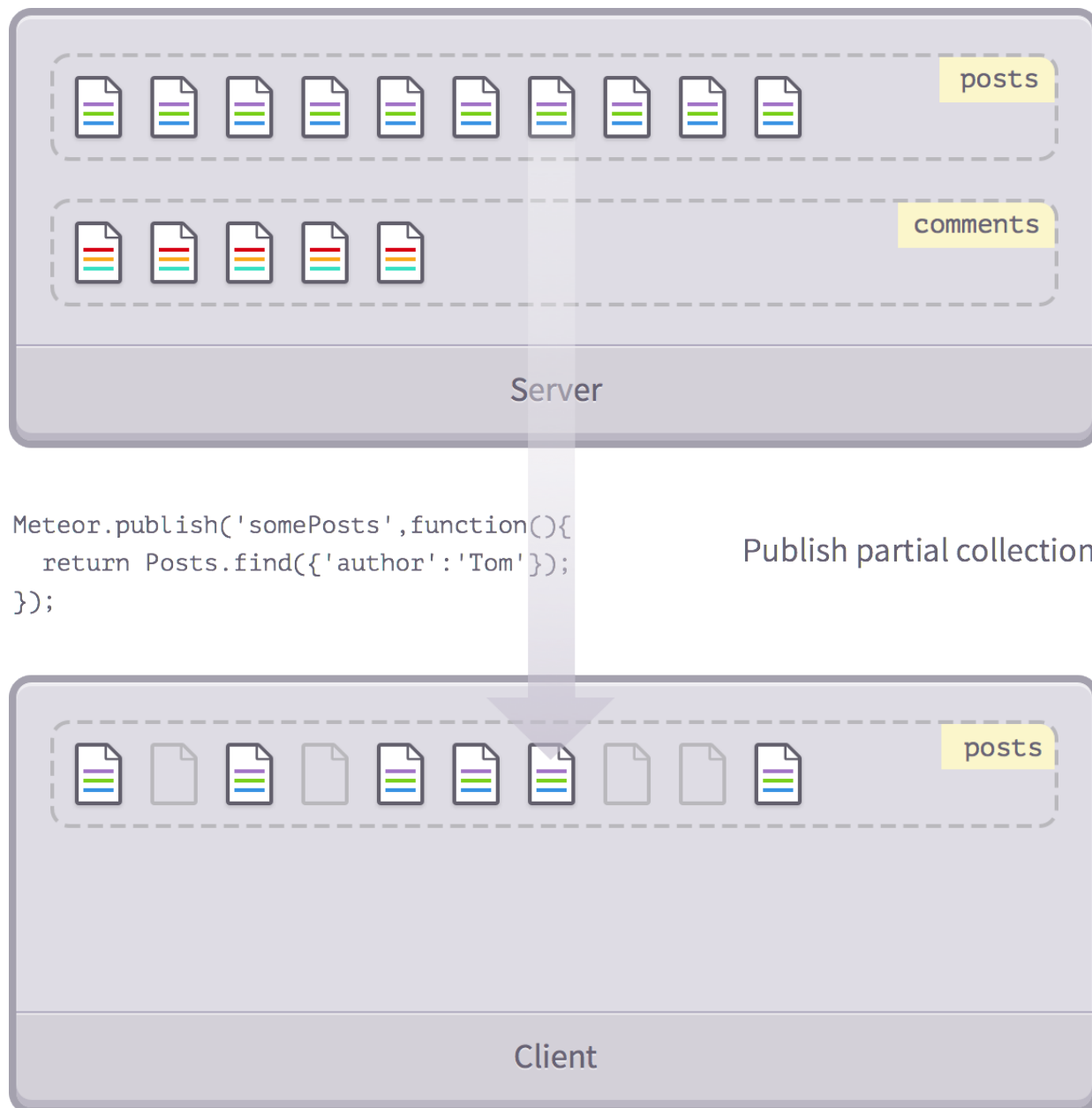
Publishing a full collection

We're still publishing full collections, but at least we now have control over which collections we publish or not. In this case, we're publishing the **Posts** collection but not **Comments**.

Publishing Partial Collections

The next level of control is publishing only *part* of a collection. For example only the posts that belong to a certain author:

```
Meteor.publish('somePosts', function(){  
  return Posts.find({'author':'Tom'});  
});
```



Publishing a partial collection

Behind The Scenes

If you've read the **Meteor publication documentation**, you were perhaps overwhelmed by talk of using `added()` and `ready()` to set attributes of records on the client, and struggled to square that with the Meteor apps that you've seen that never use those methods.

The reason is that Meteor provides a very important convenience: the `_publishCursor()` method. You've never seen that used either? Perhaps not directly, but if you return a **cursor** (i.e. `Posts.find({'author': 'Tom'})`) in a publish function, that's exactly what Meteor is using.

When Meteor sees that the `somePosts` publication has returned a cursor, it calls `_publishCursor()` to – you guessed it – publish that cursor automatically.

Here's what `_publishCursor()` does:

- It checks the name of the server-side collection.
- It pulls all matching documents from the cursor and sends it into a client-side collection *of the same name*. (It uses `.added()` to do this).
- Whenever a document is added, removed or changed, it sends those changes down to the client-side collection. (It uses `.observe()` on the cursor and `.added()`, `.changed()` and `removed()` to do this).

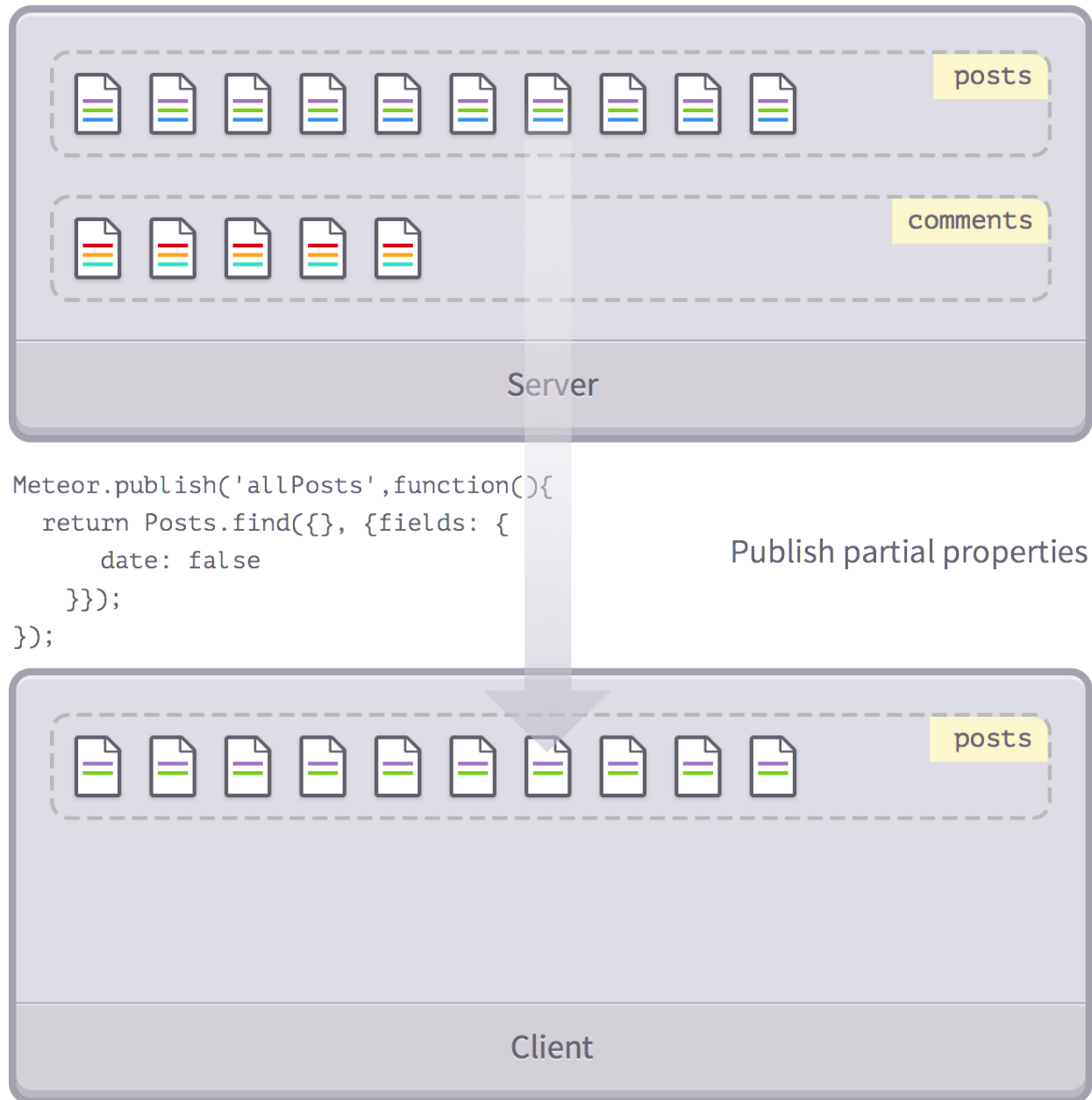
So in the example above, we are able to make sure that the user only has the posts that they are interested in (the ones written by Tom) available to them in their client side cache.

Publishing Partial Properties

We've seen how to only publish some of our posts, but we can keep slicing thinner! Let's see how to only publish specific *properties*.

Just like before, we'll use `find()` to return a cursor, but this time we'll exclude certain fields:

```
Meteor.publish('allPosts', function(){  
  return Posts.find({}, {fields: {  
    date: false  
  }});  
});
```



Publishing partial properties

Of course, we can also combine both techniques. For example, if we wanted to return all posts by Tom while leaving aside their dates, we would write:

```
Meteor.publish('allPosts', function(){
  return Posts.find({'author':'Tom'}, {fields: {
    date: false
  }});
});
```

Summing Up

So we've seen how to go from publishing every property of all documents of every collection (with `autopublish`) to publishing only *some* properties of *some* documents of *some* collections.

This covers the basics of what you can do with Meteor publications, and these simple techniques should take care of the vast majority of use cases.

Sometimes, you'll need to go further by combining, linking, or merging publications. We will cover these in a later chapter!

Now that we have a list of posts (which will eventually be user-submitted), we need an individual post page where our users will be able to discuss each post.

We'd like these pages to be accessible via a *permalink*, a URL of the form

`http://myapp.com/posts/xyz` (where `xyz` is a MongoDB `_id` identifier) that is unique to each post.

This means we'll need some kind of *routing* to look at what's inside the browser's URL bar and display the right content accordingly.

Adding the Iron Router Package

Iron Router is a routing package that was conceived specifically for Meteor apps.

Not only does it help with routing (setting up paths), but it can also take care of filters (assigning actions to some of these paths) and even manage subscriptions (control which path has access to what data). (Note: Iron Router was developed in part by *Discover Meteor* co-author Tom Coleman.)

First, let's install the package from Atmosphere:

```
meteor add iron:router
```

Terminal

This command downloads and installs the Iron Router package into our app, ready to use. Note that you might sometimes need to restart your Meteor app (with `ctrl+c` to kill the process, then `meteor` to start it again) before a package can be used.

Router Vocabulary

We'll be touching on a lot of different features of the router in this chapter. If you have some experience with a framework such as Rails, you'll already be familiar with most of these concepts. But if not, here's a quick glossary to bring you up to speed:

- **Routes:** A route is the basic building block of routing. It's basically the set of instructions that tell the app where to go and what to do when it encounters a URL.
- **Paths:** A path is a URL within your app. It can be static (`/terms_of_service`) or dynamic (`/posts/xyz`), and even include query parameters (`/search?keyword=meteor`).
- **Segments:** The different parts of a path, delimited by forward slashes (`/`).
- **Hooks:** Hooks are actions that you'd like to perform before, after, or even during the routing process. A typical example would be checking if the user has the proper rights before displaying a page.
- **Filters:** Filters are simply hooks that you define globally for one or more routes.
- **Route Templates:** Each route needs to point to a template. If you don't specify one, the router will look for a template with the same name as the route by default.
- **Layouts:** You can think of layouts as a "frame" for your content. They contain all the HTML code that wraps the current template, and will remain the same even if the template itself changes.
- **Controllers:** Sometimes, you'll realize that a lot of your templates are reusing the same parameters. Rather than duplicate your code, you can let all these routes inherit from a single *routing controller* which will contain all the common routing logic.

For more information about Iron Router, check out [the full documentation on GitHub](#).

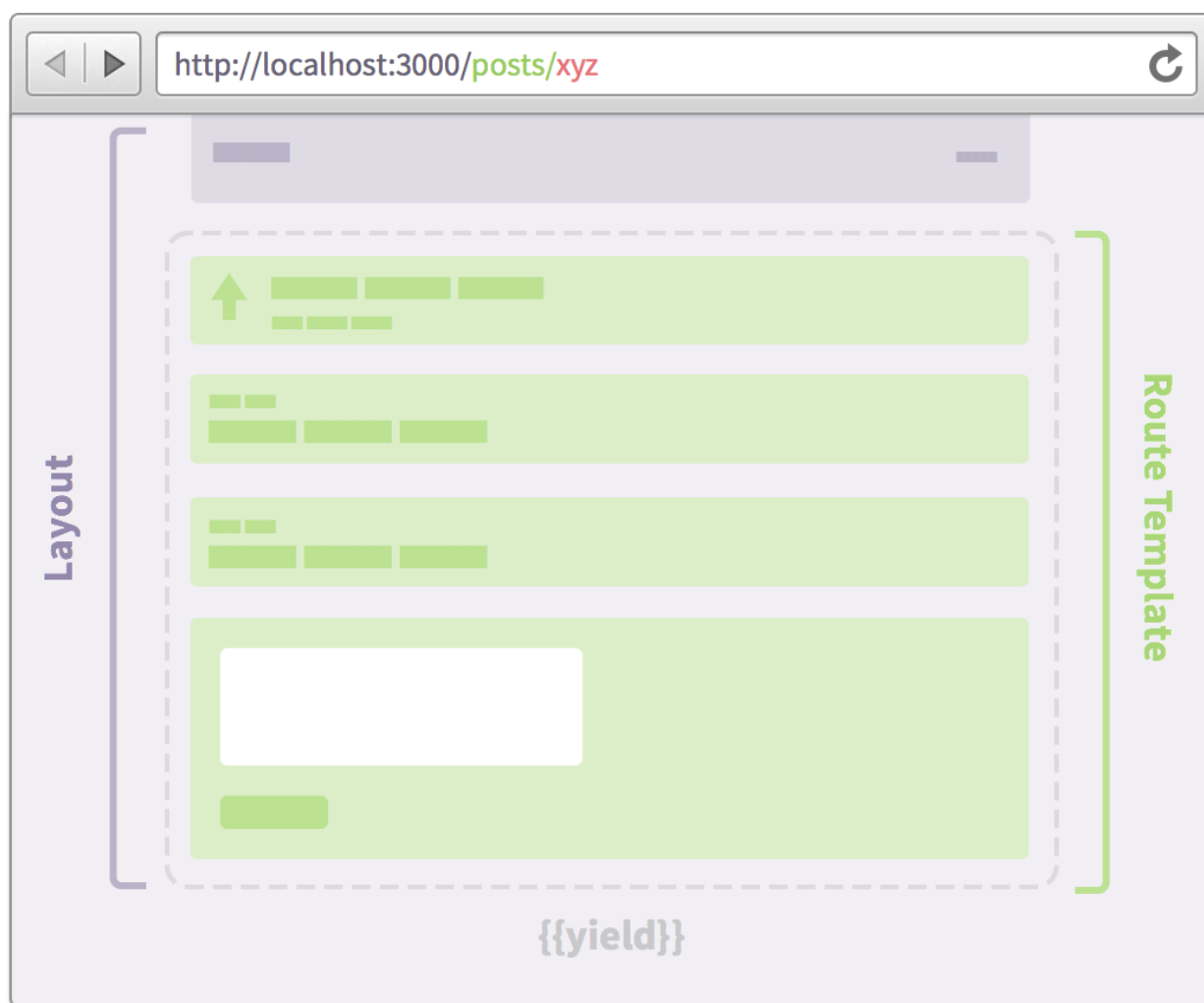
Routing: Mapping URLs To Templates

So far, we've built our layout using hard-coded template includes (such as `{{>postsList}}`). So although the content of our app can change, the page's basic structure is always the same: a header, with a list of posts below it.

Iron Router lets us break out of this mold by taking over what renders inside the HTML `<body>`

tag. So we won't define that tag's content ourselves, as we would with a regular HTML page. Instead, we will point the router to a special layout template that contains a `{{> yield}}` template helper.

This `{{> yield}}` helper will define a special dynamic zone that will automatically render whichever template corresponds to the current route (as a convention, we'll designate this special template as the "route template" from now on):



Layouts and templates.

We'll start by creating our layout and adding the `{{> yield}}` helper. First, we'll remove our HTML `<body>` tag from `main.html`, and move its contents to their own template, `layout.html` (which we'll place inside a new `client/templates/application` directory).

Iron Router will take care of embedding our layout into the stripped-down `main.html` template for us, which now looks like this:


```
<head>
  <title>Microscope</title>
</head>
```

client/main.html

While the newly created `layout.html` will now contain the app's outer layout:

```
<template name="layout">
  <div class="container">
    <header class="navbar navbar-default" role="navigation">
      <div class="navbar-header">
        <a class="navbar-brand" href="/">Microscope</a>
      </div>
    </header>
    <div id="main">
      {{> yield}}
    </div>
  </div>
</template>
```

client/templates/application/layout.html

You'll notice we've replaced the inclusion of the `postsList` template with a call to `yield` helper.

After this change, our browser tab will show the default Iron Router help page. This is because we haven't told the router what to do with the `/` URL yet, so it simply serves up an empty template.

To begin, we can regain our old behavior by mapping the root `/` URL to the `postsList` template. We'll create a new `router.js` file inside the `/lib` directory at our project's root:

```
Router.configure({
  layoutTemplate: 'layout'
});

Router.route('/', {name: 'postsList'});
```

lib/router.js

We've done two important things. First, we've told the router to use the `layout` template we just created as the default layout for all routes.

Second, we've defined a new route named `postsList` and mapped it to the root `/` path.

The `/lib` folder

Anything you put inside the `/lib` folder is guaranteed to load first before anything else in your app (with the possible exception of smart packages). This makes it a great place to put any helper code that needs to be available at all times.

A bit of warning though: note that since the `/lib` folder is neither inside `/client` or `/server`, this means its contents will be available to both environments.

Named Routes

Let's clear up a bit of ambiguity here. We named our route `postsList`, but we also have a *template* called `postsList`. So what's going on here?

By default, Iron Router will look for a template with the same name as the route name. In fact, it will even infer the name from the *path* you provide. Although it wouldn't work in this particular case (since our path is `/`), Iron Router would've found the correct template if we had used `http://localhost:3000/postsList` as our path.

You may be wondering why we even need to name our routes in the first place. Naming routes lets us use a few Iron Router features that make it easier to build links inside our app. The most useful one is the `{{pathFor}}` Spacebars helper, which returns the URL path component of any route.

We want our main home link to point us back to the posts list, so instead of specifying a static `/` URL, we can also use the Spacebars helper. The end result will be the same, but this gives us more flexibility since the helper will always output the right URL even if we later change the route's path in the router.

```
<header class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <a class="navbar-brand" href="{{pathFor 'postsList'}}">Microscope</a>
  </div>
</header>

//...
```

client/templates/application/layout.html

Commit 5-1

Very basic routing.

[View on GitHub](#)[Launch Instance](#)

Waiting On Data

If you deploy the current version of the app (or launch the web instance using the link above), you'll notice that the list appears empty for a few moments before the posts appear. This is because when the page first loads, there are no posts to display until the `posts` subscription is done grabbing the post data from the server.

It would be a much better user experience to provide some visual feedback that something is happening, and that the user should wait a moment.

Luckily, Iron Router gives us an easy way to do just that: we can ask it to *wait on* the subscription.

We start by moving our `posts` subscription from `main.js` to the router:

```
Router.configure({
  layoutTemplate: 'layout',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
```

lib/router.js

What we are saying here is that for every route on the site (we only have one right now, but soon we'll have more!), we want to subscribe to the `posts` subscription.

The key difference between this and what we had before (when the subscription was in `main.js`, which should now be empty and can be removed), is that now Iron Router knows when the route is “ready” – that is, when the route has the data it needs to render.

Get A Load Of This

Knowing when the `postsList` route is ready doesn't do us much good if we're just going to display an empty template anyway. Thankfully, Iron Router comes with a built-in way to delay showing a template until the route calling it is ready, and show a `loading` template instead:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
```

lib/router.js

Note that since we're defining our `waitOn` function globally at the router level, this sequence will only happen once when a user first accesses your app. After that, the data will already be loaded in the browser's memory and the router won't need to wait for it again.

The final piece of the puzzle is the actual loading template. We'll use the `spin` package to create a nice animated loading spinner. Add it with `meteor add sachaspin`, and then create the `loading` template as follows in the `client/templates/includes` directory:

```
<template name="loading">
  {{>spinner}}
</template>
```

`client/templates/includes/loading.html`

Note that `{{>spinner}}` is a partial contained in the `spin` package. Even though this partial comes from “outside” our app, we can include it just like any other template.

It's usually a good idea to wait on your subscriptions, not just for the user experience, but also because it means you can safely assume that data will always be available from within a template. This eliminates the need to deal with templates being rendered before their underlying data is available, which often requires tricky workarounds.

Commit 5-2

Wait on the post subscription.

[View on GitHub](#)[Launch Instance](#)

A First Glance At Reactivity

Reactivity is a core part of Meteor, and although we've yet to really touch on it, our loading template gives us a first glance at this concept.

Redirecting to a loading template if data isn't loaded yet is all well and good, but how does the router know when to redirect the user *back* to the right page once the data comes through?

For now, let's just say that this is exactly where reactivity comes in, and leave it at this. But don't worry, you'll learn more about it very soon!

Routing To A Specific Post

Now that we've seen how to route to the `postsList` template, let's set up a route to display the details of a single post.

There's just one catch: we can't go ahead and define one route per post, since there might be hundreds of them. So we'll need to set up a single *dynamic* route, and make that route display any post we want.

To start with, we'll create a new template that simply renders the same post template that we used earlier in the list of posts.

```
<template name="postPage">
  <div class="post-page page">
    {{> postItem}}
  </div>
</template>
```

client/templates/posts/post_page.html

We'll add more elements to this template later on (such as comments), but for now it'll simply serve as a shell for our `{{> postItem}}` include.

We are going to create another named route, this time mapping URL paths of the form `/posts/<ID>` to the `postPage` template:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
Router.route('/posts/:_id', {
  name: 'postPage'
});
```

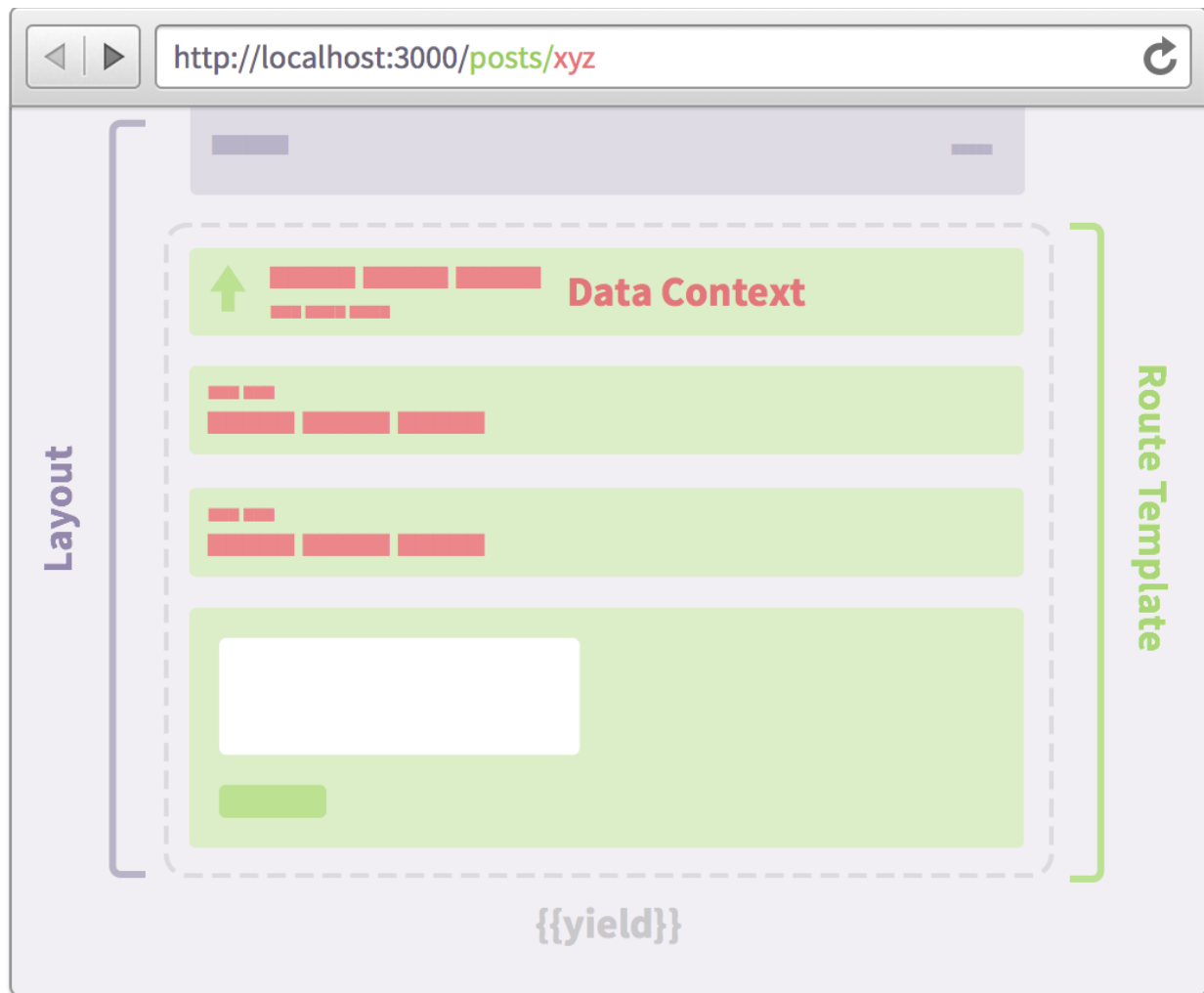
lib/router.js

The special `:_id` syntax tells the router two things: first, to match any route of the form `/posts/xyz/`, where “xyz” can be anything at all. Second, to put whatever it finds in this “xyz” spot inside an `_id` property in the router’s `params` array.

Note that we’re only using `_id` for convenience’s sake here. The router has no way of knowing if you’re passing it an actual `_id`, or just some random string of characters.

We’re now routing to the correct template, but we’re still missing something: the router knows the `_id` of the post we’d like to display, but the template still has no clue. So how do we bridge that gap?

Thankfully, the router has a clever built-in solution: it lets you specify a template’s **data context**. You can think of the data context as the filling inside a delicious cake made of templates and layouts. Simply put, it’s what you fill up your template with:



The data context.

In our case, we can get the proper data context by looking for our post based on the `_id` we got from the URL:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});
Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});
```

lib/router.js

So every time a user accesses this route, we'll find the appropriate post and pass it to the template. Remember that `findOne` returns a single post that matches a query, and that providing just an `id` as an argument is a shorthand for `{_id: id}`.

Within the `data` function for a route, `this` corresponds to the currently matched route, and we can use `this.params` to access the named parts of the route (which we indicated by prefixing them with `:` inside our `path`).

More About Data Contexts

By setting a template's *data context*, you can control the value of `this` inside template helpers.

This is usually done implicitly with the `{{#each}}` iterator, which automatically sets the data context of each iteration to the item currently being iterated on:

```
{{#each widgets}}  
  {{> widgetItem}}  
{{/each}}
```

But we can also do it explicitly using `{{#with}}`, which simply says “take this object, and apply the following template to it”. For example, we can write:

```
{{#with myWidget}}  
  {{> widgetPage}}  
{{/with}}
```

It turns out you can achieve the same result by passing the context as an *argument* to the template call. So the previous block of code can be rewritten as:

```
{{> widgetPage myWidget}}
```

For an in-depth exploration of data contexts we suggest [reading our blog post](#) on the topic.

Using a Dynamic Named Route Helper

Finally, we'll create a new "Discuss" button that will link to our individual post page. Again, we could do something like ``, but using a route helper is just more reliable.

We've named the post route `postPage`, so we can use a `{{pathFor 'postPage'}}` helper:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  </div>
</template>
```

client/templates/posts/post_item.html

Commit 5-3

Routing to a single post page.

[View on GitHub](#)[Launch Instance](#)

But wait, how exactly does the router know where to get the `xyz` part in `/posts/xyz`? After all, we're not passing it any `_id`.

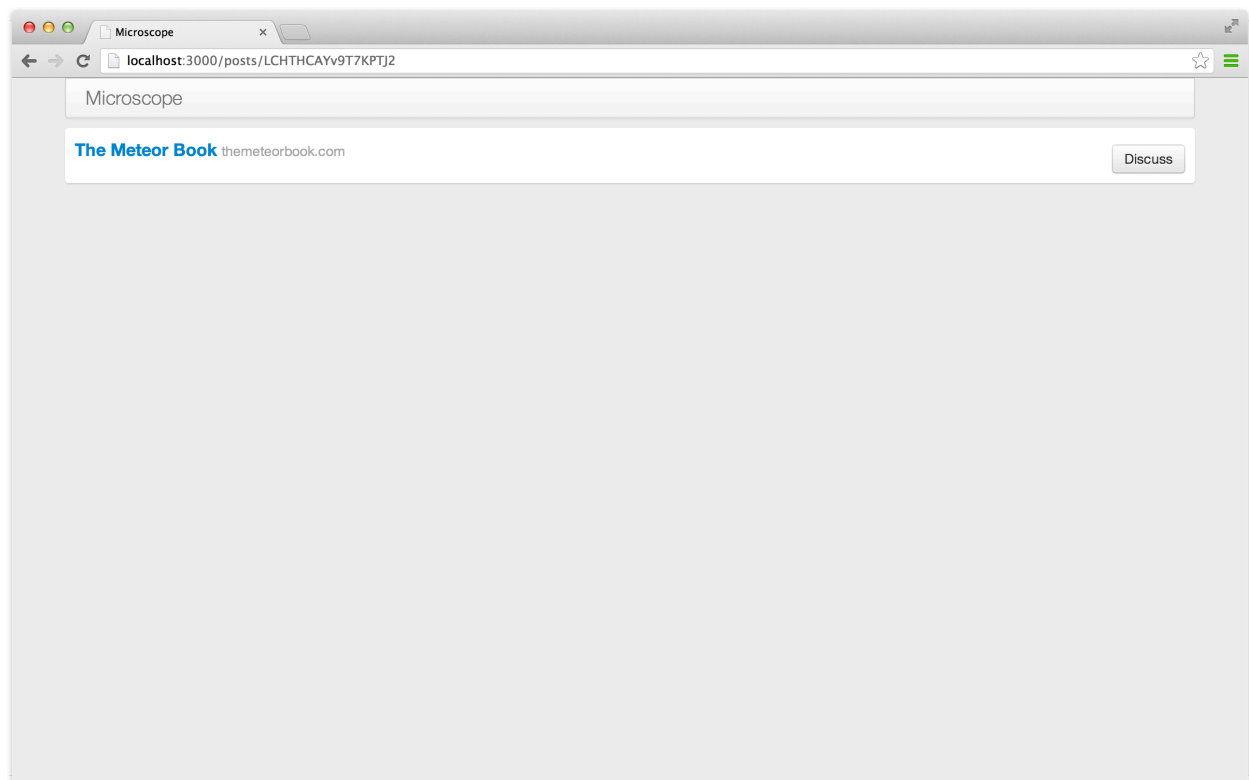
It turns out that Iron Router is smart enough to figure it out by itself. We're telling the router to use the `postPage` route, and the router knows that this route requires an `_id` of some kind (since that's how we defined our `path`).

So the router will look for this `_id` in the most logical place available: the data context of the `{{pathFor 'postPage'}}` helper, in other words `this`. And it so happens that our `this`

corresponds to a post, which (surprise!) does possess an `_id` property.

Alternatively, you can also explicitly tell the router where you'd like it to look for the `_id` property, by passing a second argument to the helper (i.e. `{{pathFor 'postPage' someOtherPost}}`). A practical use of this pattern would be getting the link to the previous or next posts in a list, for example.

To see if it works correctly, browse to the post list and click on one of the 'Discuss' links. You should see something like this:



A single post page.

HTML5 pushState

One thing to realize is that these URL changes are happening using **HTML5 pushState**.

The Router picks up clicks on URLs that are internal to the site, and prevents the browser from browsing away from the app, instead just making the necessary changes to the app's state.

If everything is working correctly the page should change instantaneously. In fact, sometimes things change so fast that some kind of page transition might be needed. This is outside of the scope of this chapter, but an interesting topic nonetheless.

Post Not Found

Let's not forget that routing works both ways: it can change the URL when we visit a page, but it can also display a new page when we change *the URL*. So we need to figure out what happens if somebody enters the *wrong* URL.

Thankfully, Iron Router takes care of this for us through the `notFoundTemplate` option.

First, we'll set up a new template to show a simple 404 error message:

```
<template name="notFound">
  <div class="not-found page jumbotron">
    <h2>404</h2>
    <p>Sorry, we couldn't find a page at this address.</p>
  </div>
</template>
```

```
client/templates/application/not_found.html
```

Then, we'll simply point Iron Router to this template:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

//...
```

lib/router.js

To test out your new error page, you can try accessing a random URL like

`http://localhost:3000/nothing-here` .

But wait, what if someone enters a URL of the form `http://localhost:3000/posts/xyz` , where `xyz` is *not* a valid post `_id` ? This is still a valid route, just not one that points to any data.

Thankfully, Iron Router is smart enough to figure this out if we just add a special `dataNotFound` hook at the end of `router.js` :

```
//...

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
```

lib/router.js

This tells Iron Router to show the “not found” page not just for invalid routes but also for the `postPage` route, whenever the `data` function returns a “falsy” (i.e. `null` , `false` , `undefined` , or empty) object.

Commit 5-4

Added not found template.

[View on GitHub](#)

[Launch Instance](#)

Why “Iron”?

You might be wondering about the story behind the name “Iron Router”. According to Iron Router author Chris Mather, it comes from the fact that meteors are composed primarily of iron.

Meteor is a reactive framework. What this means is that as data changes, things in your application change without you having to explicitly do anything.

We've already seen this in action in how our templates change as the data and the route changes.

We'll dive deeper into how this works in later chapters, but for now, we'd like to introduce some basic reactive features that are extremely useful in general apps.

The Meteor Session

Right now in Microscope, the current state of the user's application is completely contained in the URL that they are looking at (and the database).

But in many cases, you'll need to store some ephemeral state that is only relevant to the current user's version of the application (for example, whether an element is shown or hidden). The Session is a convenient way to do this.

The Session is a global reactive data store. It's global in the sense of a global singleton object: there's one session, and it's accessible everywhere. Global variables are usually seen as a bad thing, but in this case the session can be used as a central communication bus for different parts of the application.

Changing the Session

The Session is available anywhere on the client as the `Session` object. To set a session value, you can call:

```
Session.set('pageTitle', 'A different title');
```

Browser console

You can read the data back out again with `Session.get('mySessionProperty');`. This is a reactive data source, which means that if you were to put it in a helper, you would see the helper's output change reactively as the Session variable is changed.

To try this, add the following code to the layout template:

```
<header class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <a class="navbar-brand" href="{{pathFor 'postsList'}}">{{pageTitle}}</a>
  </div>
</header>
```

client/templates/application/layout.html

```
Template.layout.helpers({
  pageTitle: function() { return Session.get('pageTitle'); }
});
```

client/templates/application/layout.js

A Note About Sidebar Code

Note that code featured in sidebar chapters is not part of the main flow of the book. So either create a new branch now (if you're using Git), or else make sure to revert your changes at the end of this chapter.

Meteor's automatic reload (known as the "hot code reload" or HCR) preserves Session variables, so we should now see "A different title" displayed in the nav bar. If not, just type the previous `Session.set()` command again.

Moreover if we change the value once more (again in the browser console), we should see yet another title displayed:


```
> Session.set('pageTitle', 'A brand new title');
```

Browser console

The Session is globally available, so such changes can be made anywhere in the application. This gives us a lot of power, but can also be a trap if used too much.

By the way, it's important to point out that the Session object is *not* shared between users, or even between browser tabs. That's why if you open your app in a new tab now, you'll be faced with a blank site title.

Identical Changes

If you modify a Session variable with `Session.set()` but set it to an identical value, Meteor is smart enough to bypass the reactive chain, and avoid unnecessary function calls.

Introducing Autorun

We've looked at an example of a reactive data source, and watched it in action inside a template helper. But while some contexts in Meteor (such as template helpers) are inherently reactive, the majority of a Meteor's app code is still plain old non-reactive JavaScript.

Let's suppose we have the following code snippet somewhere in our app:

```
helloWorld = function() {  
  alert(Session.get('message'));  
}
```

Even though we're calling a Session variable, the *context* in which it's called is not reactive, meaning that we won't get new `alert`s every time we change the variable.

This is where **Autorun** comes in. As the name implies, the code inside an `autorun` block will automatically run and keep running each and every time the reactive data sources used inside it change.

Try typing this into the browser console:

```
> Tracker.autorun( function() { console.log('Value is: ' + Session.get('pageTitle')); } );  
Value is: A brand new title
```

Browser console

As you might expect, the block of code provided inside the `autorun` runs once, outputting its data to the console. Now, let's try changing the title:

```
> Session.set('pageTitle', 'Yet another value');  
Value is: Yet another value
```

Browser console

Magic! As the session value changed, the `autorun` knew it had to run its contents all over again, re-outputting the new value to the console.

So going back to our previous example, if we want to trigger a new alert every time our Session variable changes, all we need to do is wrap our code in an `autorun` block:

```
Tracker.autorun(function() {  
  alert(Session.get('message'));  
});
```

As we've just seen, `autorun` can be very useful to track reactive data sources and react imperatively to them.

Hot Code Reload

During our development of Microscope, we've been taking advantage of one of Meteor's time-saving features: hot code reload (HCR). Whenever we save one of our source code files, Meteor detects the changes and transparently restarts the running Meteor server, informing each client to reload the page.

This is similar to an automatic reload of the page, but with an important difference.

To find out what that is, start by resetting the session variable we've been using:

```
> Session.set('pageTitle', 'A brand new title');  
> Session.get('pageTitle');  
'A brand new title'
```

Browser console

If we were to reload our browser window manually, our Session variables would naturally be lost (since this would create a new session). On the other hand, if we trigger a hot code reload (for example, by saving one of our source files) the page will reload, but the session variable will still be set. Try it now!

```
> Session.get('pageTitle');  
'A brand new title'
```

Browser console

So if we're using session variables to keep track of exactly what the user is doing, the HCR should be almost transparent to the user, as it will preserve the value of all session variables. This enables us to deploy new production versions of our Meteor application with the confidence that our users will be minimally disrupted.

Consider this for a moment. If we can manage to keep all of our state in the URL and the session,

we can transparently change the *running source code* of each client's application underneath them with minimal disruption.

Let's now check what happens when we refresh the page manually:

```
> Session.get('pageTitle');  
null
```

Browser console

When we reloaded the page, we lost the session. On an HCR, Meteor saves the session to local storage in your browser and loads it in again after the reload. However, the alternate behaviour on explicit reload makes sense: if a user reloads the page, it's as if they've browsed to the same URL again, and they should be reset to the starting state that any user would see when they visit that URL.

The important lessons in all this are:

1. Always store user state in the Session or the URL so that users are minimally disrupted when a hot code reload happens.
2. Store any state that you want to be shareable between users *within the URL itself*.

This concludes our exploration of the Session, one of Meteor's most handy features. Now don't forget to revert any changes to your code before moving on to the next chapter.

So far, we've managed to create and display some static fixture data in a sensible fashion and wire it together into a simple prototype.

We've even seen how our UI is responsive to changes in the data, and inserted or changed data appears immediately. Still, our site is hamstrung by the fact that we can't enter data. In fact, we don't even have users yet!

Let's see how we can fix that.

Accounts: users made simple

In most web frameworks, adding user accounts is a familiar drag. Sure, you have to do it on almost every project, but it's never as easy as it could be. What's more, as soon as you have to deal with OAuth or other 3rd party authentication schemes, things tend to get ugly fast.

Luckily, Meteor has you covered. Thanks to the way Meteor packages can contribute code on both the server (JavaScript) and client (JavaScript, HTML, and CSS) side, we can get an accounts system almost for free.

We could just use Meteor's built-in UI for accounts (with `meteor add accounts-ui`) but since we've built our whole app with Bootstrap, we'll use the `ian:accounts-ui-bootstrap-3` package instead (don't worry, the only difference is the styling). On the command line, we type:

```
meteor add ian:accounts-ui-bootstrap-3
meteor add accounts-password
```

Terminal

Those two commands make the special accounts templates available to us, and we can include them in our site using the `{{> loginButtons}}` helper. A handy tip: you can control on which side your log-in dropdown shows up using the `align` attribute (for example: `{{> loginButtons`

```
align="right"}} ).
```

We'll add the buttons to our header. And since that header is starting to grow larger, let's give it more room in its own template (we'll put it in `client/templates/includes/`). We're also using some extra markup and classes **as specified by Bootstrap** to make sure everything looks nice:

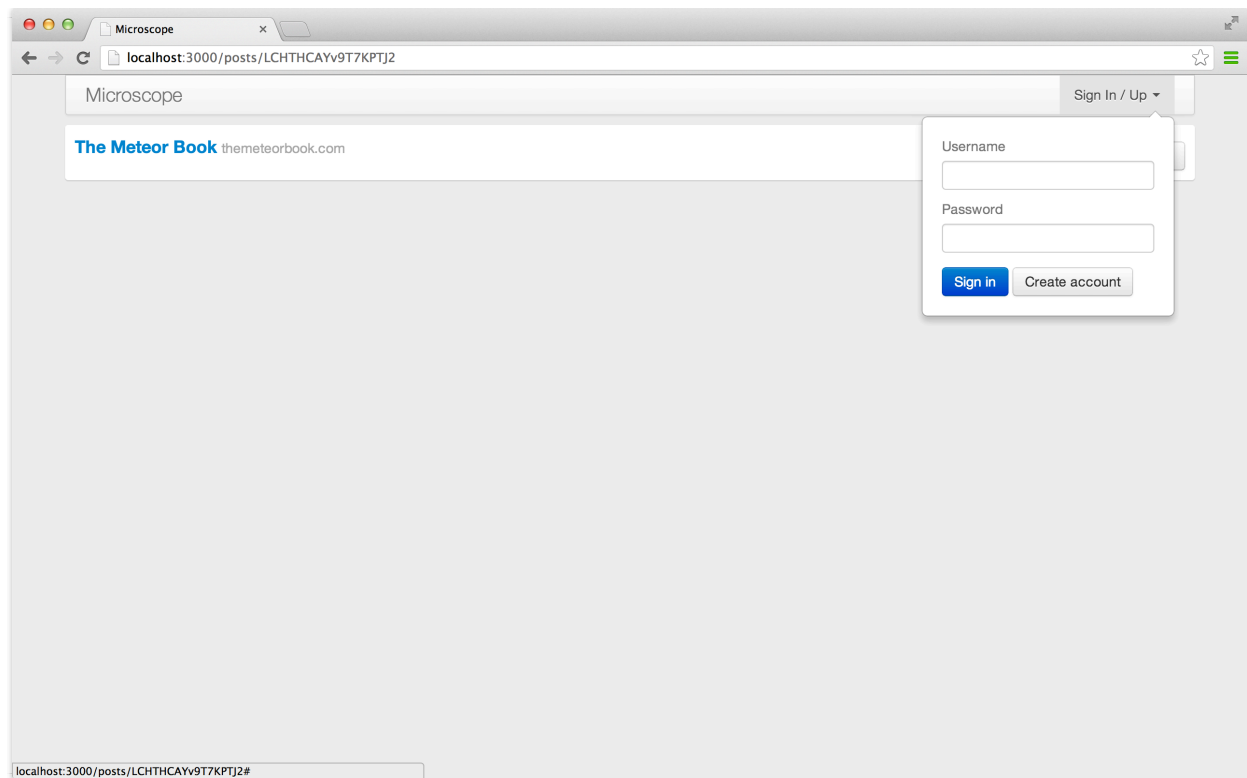
```
<template name="layout">
  <div class="container">
    {{> header}}
    <div id="main">
      {{> yield}}
    </div>
  </div>
</template>
```

`client/templates/application/layout.html`

```
<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'postsList'}}">Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>
```

`client/templates/includes/header.html`

Now, when we browse to our app, we see the accounts login buttons in the top right hand corner of our site.



Meteor's built-in accounts UI

We can use these to sign up, log in, request a change of password, and everything else that a simple site needs for password-based accounts.

To tell our accounts system that we want users to log-in via a username, we simply add an `Accounts.ui` config block in a new `config.js` file inside `client/helpers/`:

```
Accounts.ui.config({  
  passwordSignupFields: 'USERNAME_ONLY'  
});
```

`client/helpers/config.js`

Commit 6-1

Added accounts and added template to the header

[View on GitHub](#)

[Launch Instance](#)

Creating Our First User

Go ahead and sign up for an account: the “Sign in” button will change to show your username. This confirms that a user account has been created for you. But where is that user account data coming from?

By adding the `accounts` package, Meteor has created a special new collection, which can be accessed at `Meteor.users`. To see it, open your browser console and type:

```
> Meteor.users.findOne();
```

Browser console

The console should return an object representing your user object; if you take a look, you can see that your username is in there, as well as an `_id` that uniquely identifies you. Note that you can also get the currently logged-in user with `Meteor.user()`.

Now log out and sign up again with a different username. `Meteor.user()` should now return a second user. But wait, let's run:

```
> Meteor.users.find().count();
```

```
1
```

Browser console

The console returns 1. Hold on, shouldn't that be 2? Has the first user been deleted? If you try logging in as that first user again, you'll see that's not the case.

Let's make sure and check in the canonical data-store, the Mongo database. We'll log into Mongo (`meteor mongo` in your terminal) and check:


```
> db.users.count()  
2
```

Mongo console

There are definitely two users. So why can we only see a single one at a time in the browser?

A Mystery Publication!

If you think back to Chapter 4, you might remember that by turning off `autopublish`, we stopped collections from automatically sending all the data from the server into each connected client's local version of the collection. We needed to create a publication and subscription pair to channel the data across.

Yet we never set up any kind of user publication. So how come we can even see any user data at all?

The answer is that the accounts package actually does “auto-publish” the currently logged in user's basic account details no matter what. If it didn't, then that user could never log in to the site!

The accounts package only publishes the *current* user though. This explains why one user can't see another's account details.

So the publication is only publishing one user object per logged-in user (and none when you are not logged in).

What's more, documents in our user collection don't seem to contain the same fields on the server and on the client. In Mongo, a user has a lot of data in it. To see it, just go back to your Mongo terminal and type:

```
> db.users.find()
{
  "_id": "H5kKyxtbkLhmPgtqs",
  "createdAt": ISODate("2015-02-10T08:26:48.196Z"),
  "profile": {},
  "services": {
    "password": {
      "bcrypt": "$2a$10$yGPYwo3/53IHsdffdwe766roZviT03YBGltJ0UG"
    },
    "resume": {
      "loginTokens": [{
        "when": ISODate("2015-02-10T08:26:48.203Z"),
        "hashedToken": "npXGH7Rmkuxcv098wzz+qR0/jHl0EAGWr0D9Zp0w="
      }]
    }
  },
  "username": "sacha"
}
```

Mongo console

On the other hand, in the browser the user object is much more pared down, as you can see by typing the equivalent command:

```
> Meteor.users.findOne();
Object { _id: "kYdBd9hr3fWPGPcii", username: "tmeasday" }
```

Browser console

This example shows us how a local collection can be a *secure subset* of the real database. The logged-in user only sees enough of the real dataset to get the job done (in this case, signing in). This is a useful pattern to learn from, as you'll see later on.

That doesn't mean you can't make more user data public if you want to. You can refer to the [Meteor docs](#) to see how to optionally publish more fields in the `Meteor.users` collection.

If collections are Meteor's core feature, then **reactivity** is the shell that makes that core useful.

Collections radically transform the way your application deals with data changes. Rather than having to check for data changes manually (e.g. through an AJAX call) and then patch those changes into your HTML, data changes can instead come in at any time and get applied to your user interface seamlessly by Meteor.

Take a moment to think it through: behind the scenes, Meteor is able to change *any* part of your user interface when an underlying collection is updated.

The *imperative* way to do this would be to use `.observe()`, a cursor function that fires callbacks when documents matching that cursor change. We could then make changes to the DOM (the rendered HTML of our webpage) through those callbacks. The resulting code would look something like this:

```
Posts.find().observe({
  added: function(post) {
    // when 'added' callback fires, add HTML element
    $('ul').append('<li id="' + post._id + '">' + post.title + '</li>');
  },
  changed: function(post) {
    // when 'changed' callback fires, modify HTML element's text
    $('ul li#' + post._id).text(post.title);
  },
  removed: function(post) {
    // when 'removed' callback fires, remove HTML element
    $('ul li#' + post._id).remove();
  }
});
```

You can probably already see how such code is going to get complex pretty quickly. Imagine dealing with changes to *each attribute* of the post, and having to change complex HTML within the post's ``. Not to mention all the complicated edge cases that can come out when we start relying on multiple sources of information that can all change in realtime.

When *Should* We Use `observe()` ?

Using the above pattern is sometimes necessary, especially when dealing with third-party widgets. For example, let's imagine we want to add or remove pins on a map in real time based on Collection data (say, to show the locations of currently logged in users).

In such cases, you'll need to use `observe()` callbacks in order to get the map to "talk" with the Meteor collection and know how to react to data changes. For example, you would rely on the `added` and `removed` callbacks to call the map API's own `dropPin()` or `removePin()` methods.

A Declarative Approach

Meteor provides us with a better way: reactivity, which is at its core a **declarative** approach. Being declarative lets us define the relationship between objects once and know they'll be kept in sync, instead of having to specify behaviors for every possible change.

This is a powerful concept, because a realtime system has many inputs that can all change at unpredictable times. By declaratively stating how we render HTML based on whatever reactive data sources we care about, Meteor can take care of the job of monitoring those sources and transparently take on the messy job of keeping the user interface up to date.

All this to say that instead of thinking about `observe` callbacks, Meteor lets us write:

```
<template name="postsList">
  <ul>
    {{#each posts}}
      <li>{{title}}</li>
    {{/each}}
  </ul>
</template>
```

And then get our list of posts with:

```
Template.postsList.helpers({  
  posts: function() {  
    return Posts.find();  
  }  
});
```

Behind the scenes, Meteor is wiring up `observe()` callbacks for us, and re-drawing the relevant sections of HTML when the reactive data changes.

Dependency Tracking in Meteor: Computations

While Meteor is a real-time, reactive framework, not *all* of the code inside a Meteor app is reactive. If this were the case, your whole app would re-run every time anything changed. Instead, reactivity is limited to specific areas of your code, and we call these areas **computations**.

In other words, a computation is a block of code that runs every time one of the reactive data sources it depends on changes. If you have a reactive data source (for example, a Session variable) and would like to respond reactively to it, you'll need to set up a computation for it.

Note that you usually don't need to do this explicitly because Meteor already gives each template and helper it renders its own special computation (meaning that you can be sure your templates will reactively reflect their source data).

Every reactive data source tracks all the computations that are using it so that it can let them know when its own value changes. To do so, it calls the `invalidate()` function on the computation.

Computations are generally set up to simply re-evaluate their contents on invalidation, and this is what happens to the template computations (although template computations also do some magic to try and redraw the page more efficiently). Although you can have more control on what your computation does on invalidation if you need to, in practice this is almost always the behavior you'll be using.

Setting Up a Computation

Now that we understand the theory behind computations, actually setting one up will make a lot more sense. We can use the `Tracker.autorun` function to enclose a block of code in a computation and make it reactive:

```
Meteor.startup(function() {  
  Tracker.autorun(function() {  
    console.log('There are ' + Posts.find().count() + ' posts');  
  });  
});
```

Note that we need to wrap the `Tracker` block inside a `Meteor.startup()` block to ensure that it only runs once Meteor has finished loading the `Posts` collection.

Behind the scenes, `autorun` then creates a computation, and wires it up to re-evaluate whenever the data sources it depends on change. We've set up a very simple computation that simply logs the number of posts to the console. Since `Posts.find()` is a reactive data source, it will take care of telling the computation to re-evaluate every time the number of posts changes.

```
> Posts.insert({title: 'New Post'});  
There are 4 posts.
```

The net result of all this is that we can write code that uses reactive data in a very natural way, knowing that behind the scenes the dependency system will take care of re-running it at just the right times.

We've seen how easy it is to create posts via the console, using the `Posts.insert` database call, but we can't expect our users to open the console to create a new post.

Eventually, we'll need to build some kind of user interface to let our users post new stories to our app.

Building The New Post Page

We begin by defining a route for our new page:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
```

lib/router.js

Adding A Link To The Header

With that route defined, we can now add a link to our submit page in our header:

```

<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'postsList'}}">Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav">
        <li><a href="{{pathFor 'postSubmit'}}">Submit Post</a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>

```

client/templates/includes/header.html

Setting up our route means that if a user browses to the `/submit` URL, Meteor will display the `postSubmit` template. So let's write that template:

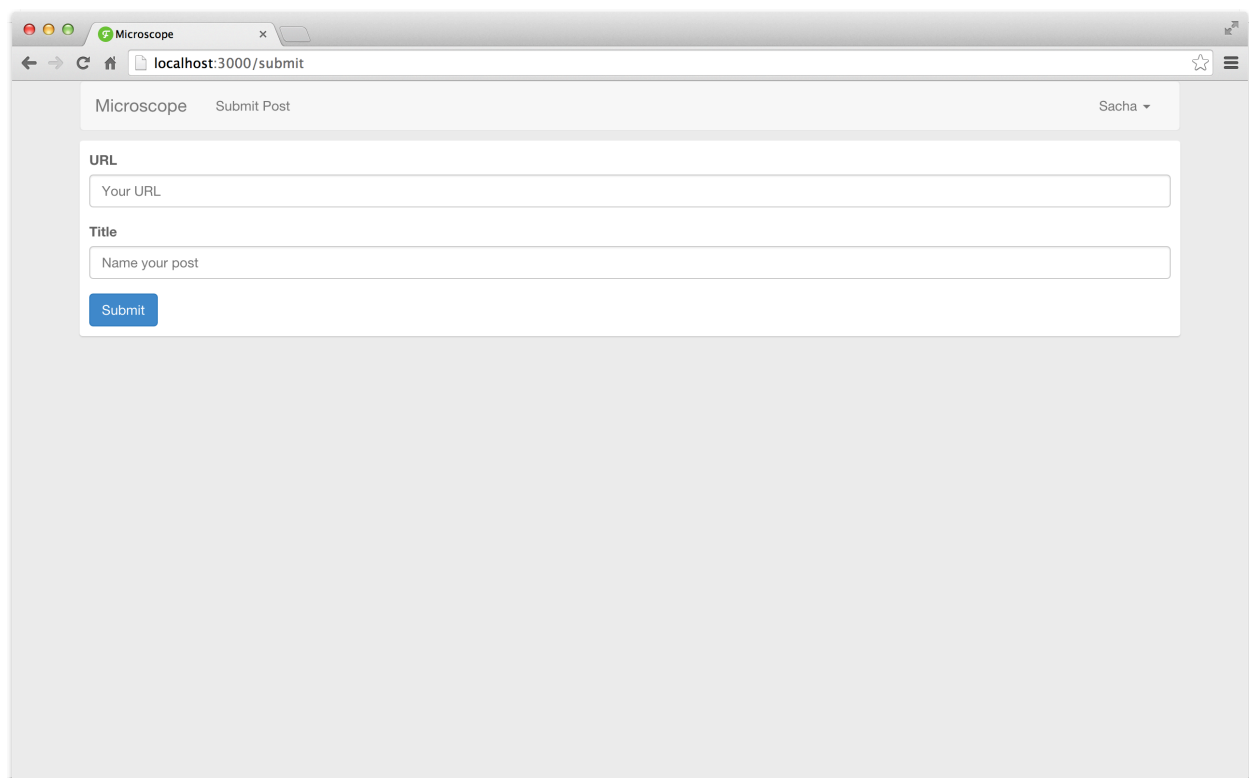

```

<template name="postSubmit">
  <form class="main form page">
    <div class="form-group">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="" placeholder="Your URL
" class="form-control"/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="" placeholder="Name
your post" class="form-control"/>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary"/>
  </form>
</template>

```

client/templates/posts/post_submit.html

Note: that's a lot of markup, but it simply comes from using Twitter Bootstrap. While only the form elements are essential, the extra markup will help make our app look a little bit nicer. It should now look similar to this:



This is a simple form. We don't need to worry about an action for it, as we'll be intercepting submit events on the form and updating data via JavaScript. (It doesn't make sense to provide a non-JS fallback when you consider that a Meteor app is completely non-functional with JavaScript disabled).

Creating Posts

Let's bind an event handler to the form `submit` event. It's best to use the `submit` event (rather than say a `click` event on the button), as that will cover all possible ways of submitting (such as hitting enter for instance).

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    post._id = Posts.insert(post);
    Router.go('postPage', post);
  }
});
```

client/templates/posts/post_submit.js

Commit 7-1

Added a submit post page and linked to it in the header.

[View on GitHub](#)[Launch Instance](#)

This function uses **jQuery** to parse out the values of our various form fields, and populate a new post object from the results. We need to ensure we `preventDefault` on the `event` argument to

our handler to make sure the browser doesn't go ahead and try to submit the form.

Finally, we can route to our new post's page. The `insert()` function on a collection returns the generated `_id` for the object that has been inserted into the database, which the Router's `go()` function will use to construct a URL for us to browse to.

The net result is the user hits submit, a post is created, and the user is instantly taken to the discussion page for that new post.

Adding Some Security

Creating posts is all very well, but we don't want to let any random visitor do it: we want them to have to be logged in to do so. Of course, we can start by hiding the new post form from logged out users. Still, a user could conceivably create a post in the browser console without being logged in, and we can't have that.

Thankfully data security is baked right into Meteor collections; it's just that it's turned off by default when you create a new project. This enables you to get started easily and start building out your app while leaving the boring stuff for later.

Our app no longer needs these training wheels, so let's take them off! We'll remove the `insecure` package:

```
meteor remove insecure
```

Terminal

After doing so, you'll notice that the post form no longer works properly. This is because without the `insecure` package, client-side inserts into the posts collection *are no longer allowed*.

We need to either set some explicit rules telling Meteor when it's OK for a client to insert posts, or else do our post insertions server-side.

Allowing Post Inserts

To begin with, we'll show how to allow client-side post inserts in order to get our form working again. As it turns out, we'll eventually settle on a different technique, but for now, the following will get things working again easily enough:

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  insert: function(userId, doc) {
    // only allow posting if you are logged in
    return !! userId;
  }
});
```

lib/collections/posts.js

Commit 7-2

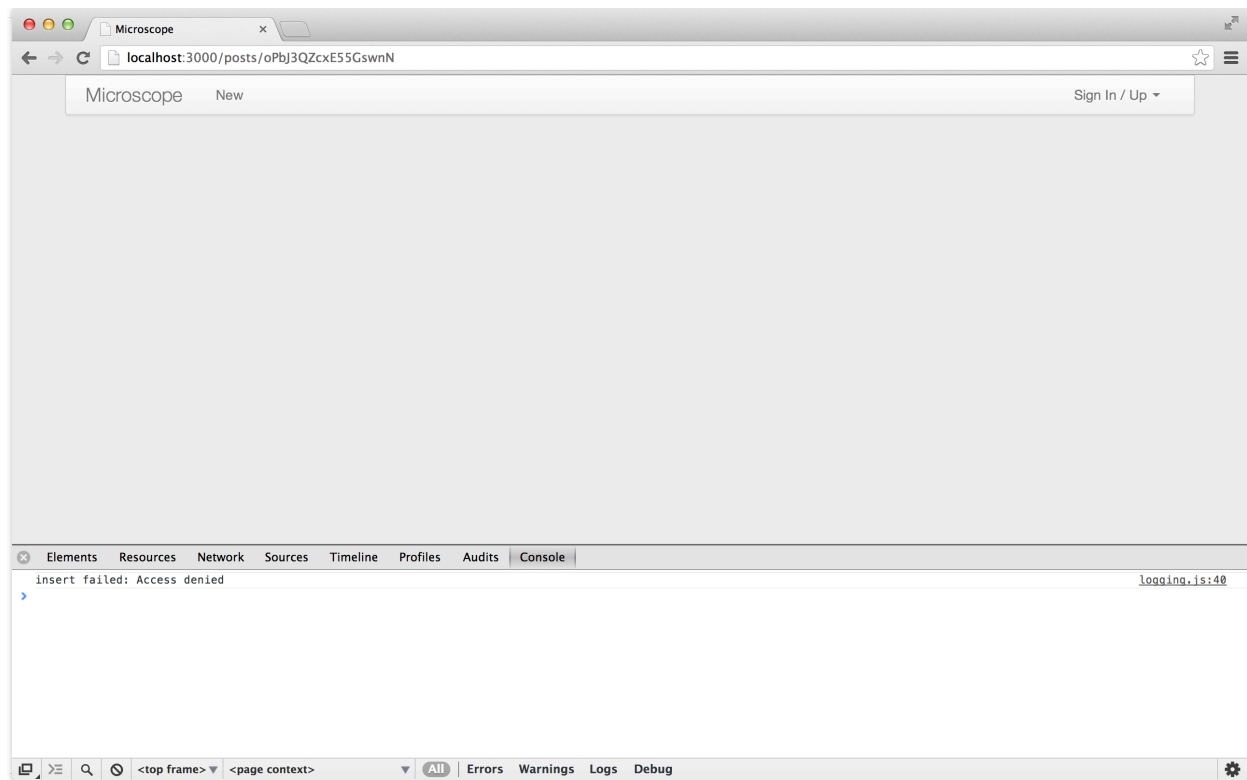
Removed insecure, and allowed certain writes to posts.

[View on GitHub](#)[Launch Instance](#)

We call `Posts.allow`, which tells Meteor “this is a set of circumstances under which clients are allowed to do things to the `Posts` collection”. In this case, we are saying “clients are allowed to insert posts as long as they have a `userId`”.

The `userId` of the user doing the modification is passed to the `allow` and `deny` calls (or returns `null` if no user is logged in), which is almost always useful. And as user accounts are tied into the core of Meteor, we can rely on `userId` always being correct.

We've managed to ensure that you need to be logged in to create a post. Try logging out and creating a post; you should see this in your console:



Insert failed: Access denied

However, we still have to deal with a couple of issues:

- Logged out users can still reach the create post form.
- The post is not tied to the user in any way (and there's no code on the server to enforce this).
- Multiple posts can be created that point to the same URL.

Let's fix these problems.

Securing Access To The New Post Form

Let's start by preventing logged out users from seeing the post submit form. We'll do that at the router level, by defining a *route hook*.

A hook intercepts the routing process and potentially changes the action that the router takes. You can think of it as a security guard that checks your credentials before letting you in (or turning you away).

What we need to do is check if the user is logged in, and if they're not render the `accessDenied`

template instead of the expected `postSubmit` template (we then stop the router from doing anything else). So let's modify `router.js` like so:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

var requireLogin = function() {
  if (!Meteor.user()) {
    this.render('accessDenied');
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

lib/router.js

We also create the template for the access denied page:

```
<template name="accessDenied">
  <div class="access-denied page jumbotron">
    <h2>Access Denied</h2>
    <p>You can't get here! Please log in.</p>
  </div>
</template>
```

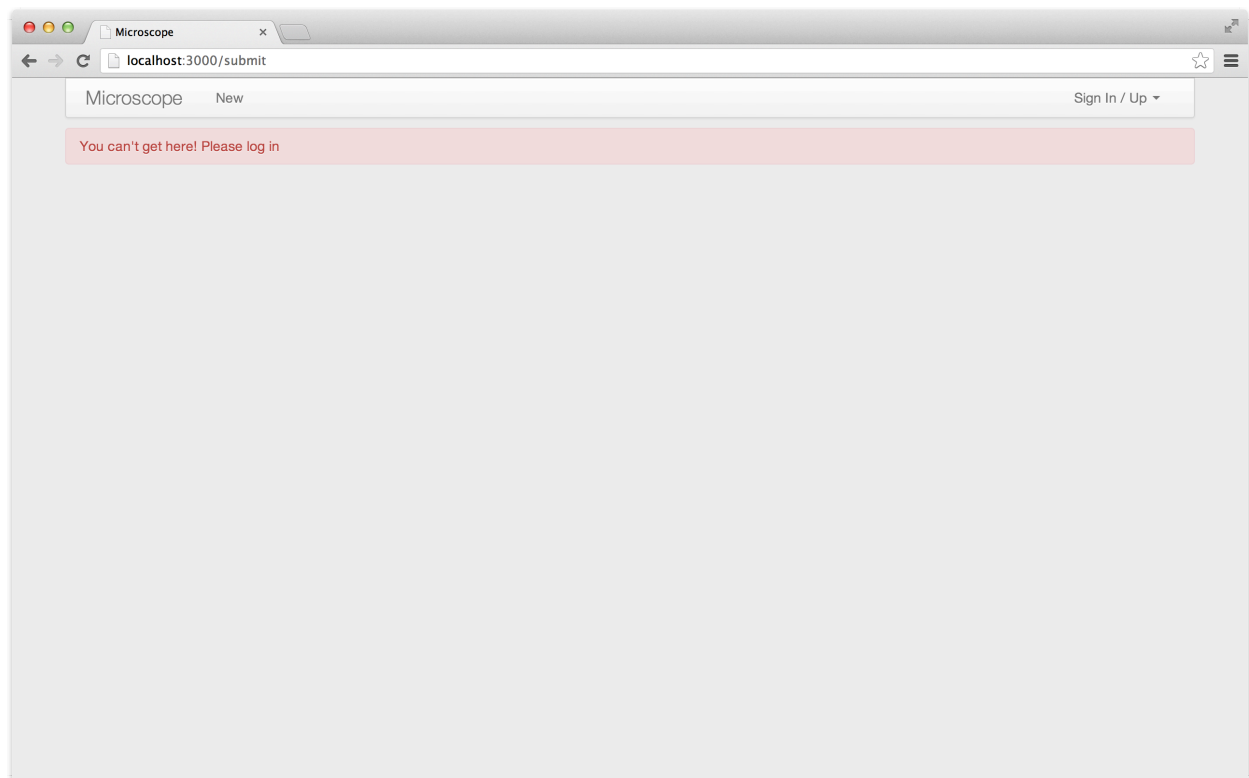
client/templates/includes/access_denied.html

Commit 7-3

Denied access to new posts page when not logged in.

[View on GitHub](#)[Launch Instance](#)

If you now head to `http://localhost:3000/submit/` without being logged in, you should see a message similar to this one:



The access denied template

The nice thing about routing hooks is that they too are *reactive*. This means we don't need to think about setting up callbacks when the user logs in: when the log-in state of the user changes, the Router's page template instantly changes from `accessDenied` to `postSubmit` without us having to write any explicit code to handle it (and by the way, this even works across browser tabs).

Log in, then try refreshing the page. You might sometimes see the access denied template flash up for a brief moment before the post submission page appears. The reason for this is that Meteor begins rendering templates as soon as possible, before it has talked to the server and checked if the current user (stored in the browser's local storage) even exists.

To avoid this problem (which is a common class of problem that you'll see more of as you deal with the intricacies of latency between client and server), we'll just display a loading screen for the brief moment that we are waiting to see if the user has access or not.

After all at this stage we don't know if the user has the correct log-in credentials, and we can't show either the `accessDenied` or the `postSubmit` template until we do.

So we modify our hook to use our loading template while `Meteor.loggingIn()` is true:

```
//...

var requireLogin = function() {
  if (! Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

lib/router.js

Commit 7-4

Show a loading screen while waiting to login.

[View on GitHub](#)[Launch Instance](#)

Hiding the Link

The easiest way to prevent users from trying to reach this page by mistake when they are logged out is to hide the link from them. We can do this pretty easily:


```
//...
```

```
<ul class="nav navbar-nav">
  {{#if currentUser}}<li><a href="{{pathFor 'postSubmit'}}">Submit Post</a></li>
</ul>
```

```
//...
```

client/templates/includes/header.html

Commit 7-5

Only show submit post link if logged in.

[View on GitHub](#)[Launch Instance](#)

The `currentUser` helper is provided to us by the `accounts` package and is the Spacebars equivalent of `Meteor.user()`. Since it's reactive, the link will appear or disappear as you log in and out of the app.

Meteor Method: Better Abstraction and Security

We've managed to secure access to the new post page for logged out users, and deny such users from creating posts even if they cheat and use the console. Yet there are still a few more things we need to take care of:

- Timestamping the posts.
- Ensuring that the same URL can't be posted more than once.
- Adding details about the post author (ID, username, etc.).

You may be thinking we can do all of that in our `submit` event handler. Realistically, however, we would quickly run into a range of problems.

- For the timestamp, we'd have to rely on the user's computer's time being correct, which is

not always going to be the case.

- Clients won't know about *all* of the URLs ever posted to the site. They'll only know about the posts that they can currently see (we'll see how exactly this works later), so there's no way to enforce URL uniqueness client-side.
- Finally, although we *could* add the user details client-side, we wouldn't be enforcing its accuracy, which could open our app up to exploitation by people using the browser console.

For all these reasons, it's better to keep our event handlers simple and, if we are doing more than the most basic inserts or updates to collections, use a **Method**.

A Meteor Method is a server-side function that is *called* client-side. We aren't totally unfamiliar with them – in fact, behind the scenes, the `Collection`'s `insert`, `update` and `remove` functions are all Methods. Let's see how to create our own.

Let's go back to `post_submit.js`. Rather than inserting directly into the `Posts` collection, we'll call a Method named `postInsert`:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

The `Meteor.call` function calls a Method named by its first argument. You can provide

arguments to the call (in this case, the `post` object we constructed from the form), and finally attach a callback, which will execute when the server-side Method is done.

Meteor method callbacks always have two arguments, `error` and `result`. If for whatever reason the `error` argument exists, we'll alert the user (using `return` to abort the callback). If everything is working as it should, we'll redirect the user to the freshly created post's discussion page.

Methods & Security

It's often good practice to share a method's code on both client and server, in order to enable latency compensation (also known as optimistic updates – see next chapter).

You might think sharing server code on the client is a bad idea, but in fact there's no security risk, since even if the method's code was somehow tampered with on the client, the client part of the method will only ever affect the in-client copy of the database, not the actual, server-side database. Moreover, any discrepancies that may arise will get automatically corrected by Meteor from the “real”, server-side db.

Security Check

We'll take advantage of this opportunity to add some security to our method by using the `check` and `audit-argument-checks` packages. Let's start by adding the packages to our app with:

```
meteor add check
meteor add audit-argument-checks
```

The `check` package lets you check any JavaScript object against a predefined pattern, and `audit-argument-checks` helps you make sure you're using `check` in every single method.

In our case, we'll use them to check that the user calling the method is properly logged in (by making sure that `Meteor.userId()` is a `String`), and that the `postAttributes` object being passed as argument to the method contains `title` and `url` strings, so we don't end up entering

any random piece of data into our database.

So let's define the `postInsert` method in our `lib/collections/posts.js` file. We'll remove the `allow()` block from `posts.js` since Meteor Methods bypass them anyway.

We'll then `extend` the `postAttributes` object with three more properties: the user's `_id` and `username`, as well as the post's `submitted` timestamp.

We *could* have the client provide these properties to the method just like `title` and `url`, this would open us to a whole host of security issues: with a simple `Posts.insert()` entered in their browser console, any user would be able to assign a post to somebody else, or back-date it. This is a very common exploit, and one we take care to avoid by assigning sensitive properties on the server, and not on the client.

Once our `post` object is complete, we then insert the whole thing in our database and return the resulting `_id` to the client (in other words, the original caller of this method) in a JavaScript object.

```

Posts = new Mongo.Collection('posts');

Meteor.methods({
  postInsert: function(postAttributes) {
    check(Meteor.userId(), String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});

```

lib/collections/posts.js

Note that the `_.extend()` method is part of the **Underscore** library, and simply lets you “extend” one object with the properties of another.

Commit 7-6

Use a method to submit the post.

[View on GitHub](#)
[Launch Instance](#)

Bye Bye Allow/Deny

Meteor Methods are executed on the server, so Meteor assumes they can be trusted. As such, Meteor methods bypass any allow/deny callbacks.

If you want to run some code before every `insert`, `update`, or `remove` *even on the server*, we suggest checking out the **collection-hooks** package.

Preventing Duplicates

We'll make one more check before wrapping up our method. If a post with the same URL has already been created previously, we won't add the link a second time but instead redirect the user to this existing post.

```

Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});

```

lib/collections/posts.js

We're searching our database for any posts with the same URL. If any are found, we `return` that post's `_id` along with a `postExists: true` flag to let the client know about this special situation.

And since we're triggering a `return` call, the method stops at that point without executing the `insert` statement, thus elegantly preventing any duplicates.

All that's left is to use this new `postExists` information in our client-side event helper to show a warning message:

```

Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});

```

client/templates/posts/post_submit.js

Commit 7-7

Enforce post URL uniqueness.

[View on GitHub](#)

[Launch Instance](#)

Sorting Posts

Now that we have a submitted date on all our posts, it makes sense to ensure that they are sorted using this attribute. To do so, we can just use Mongo's `sort` operator, which expects an object consisting of the keys to sort by, and a sign indicating whether they are ascending or descending.


```
Template.postsList.helpers({  
  posts: function() {  
    return Posts.find({}, {sort: {submitted: -1}});  
  }  
});
```

client/templates/posts/posts_list.js

Commit 7-8

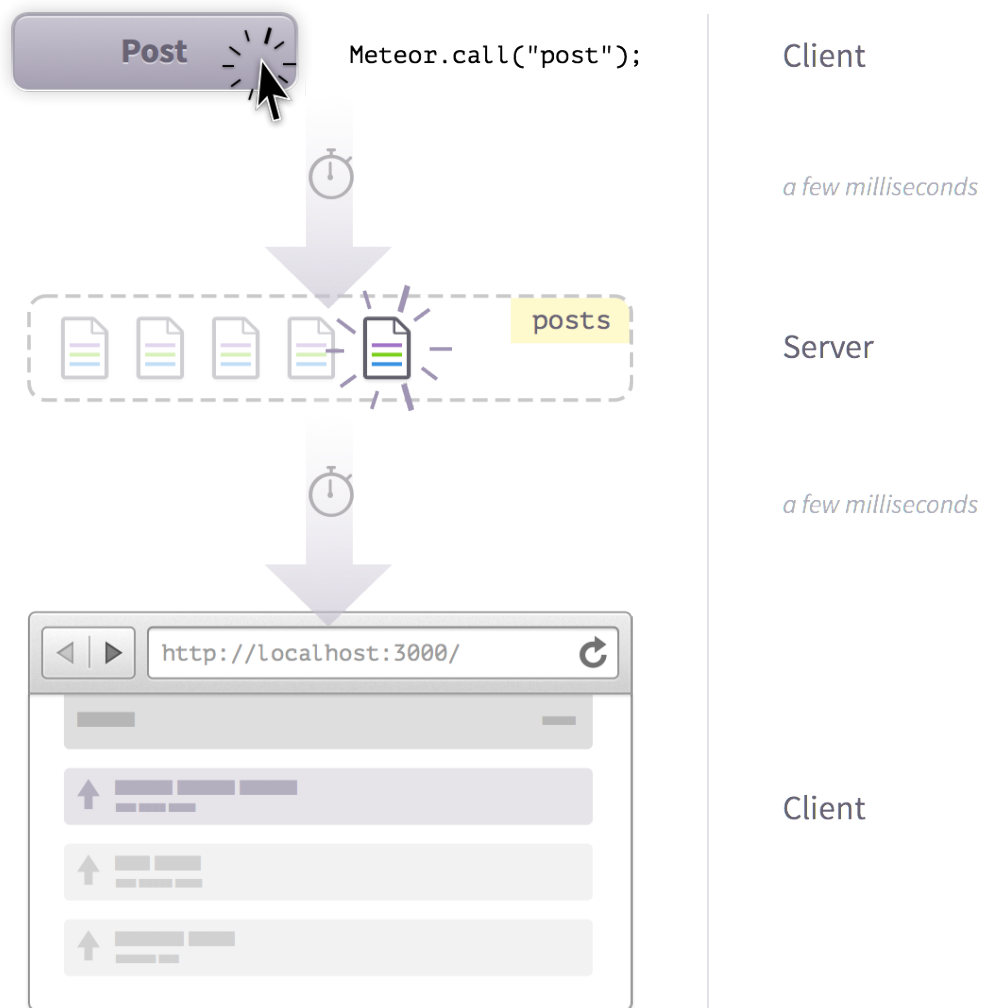
Sort posts by submitted timestamp.

[View on GitHub](#)[Launch Instance](#)

It took a bit of work, but we finally have a user interface to let users securely enter content in our app!

But any app that lets users create content also needs to give them a way to edit or delete it. That's what the next chapter will be all about.

In the last chapter, we introduced a new concept in the Meteor world: **Methods**.



Without latency compensation

A Meteor Method is a way of executing a series of commands on the server in a structured way. In our example, we used a Method because we wanted to make sure that new posts were tagged with their author's name and id as well as the current server time.

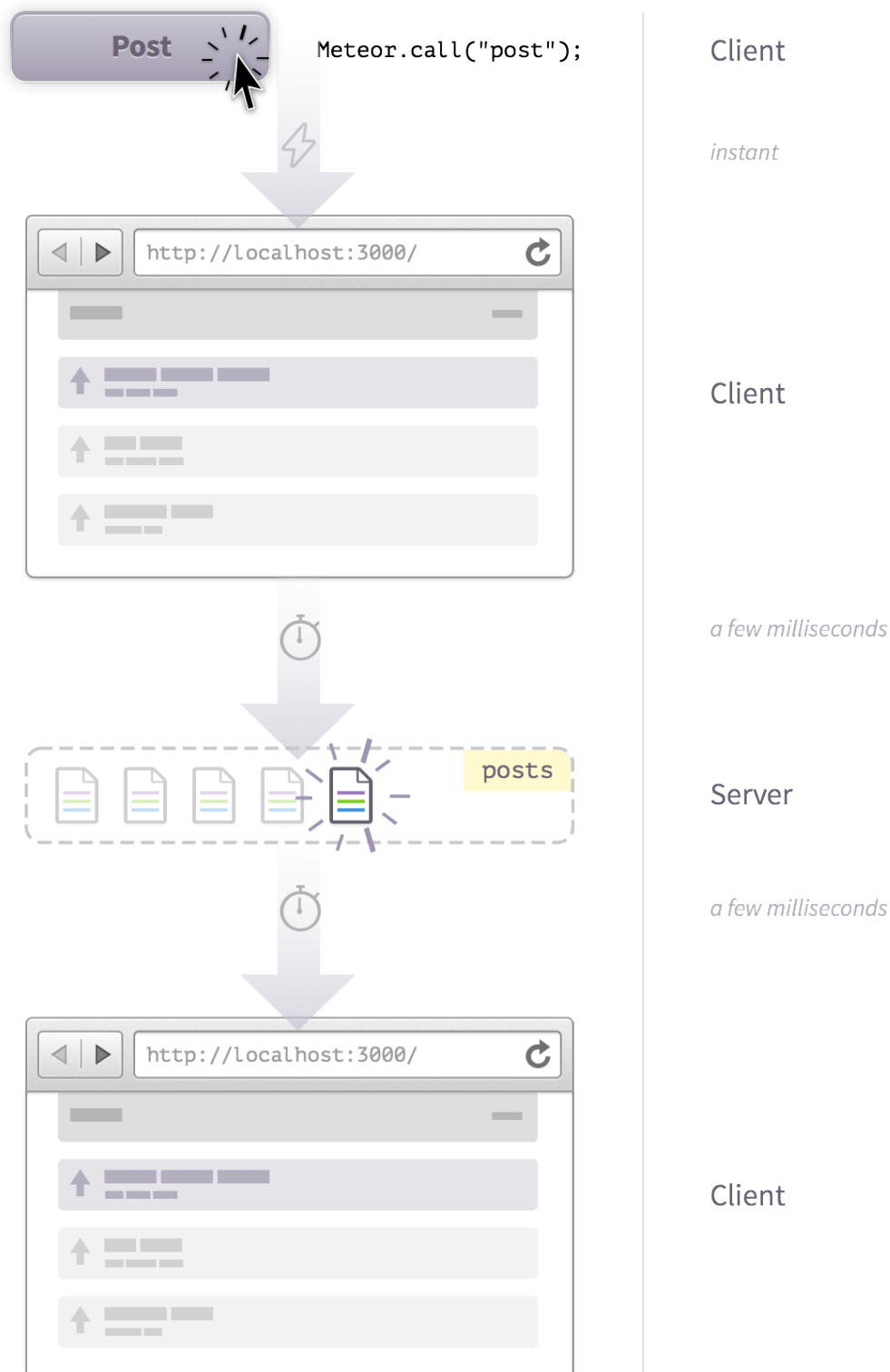
However, if Meteor executed Methods in the most basic way, we'd have a problem. Consider the following sequence of events (note: the timestamps are random values picked for illustrative purpose only):

- `+0ms`: The user clicks a submit button and the browser fires a Method call.

- +200ms: The server makes changes to the Mongo database.
- +500ms: The client receives these changes, and updates the UI to reflect them.

If this were the way Meteor operated, then there'd be a short lag between performing such actions and seeing the results (that lag being more or less noticeable depending on how close you were to the server). We can't have that in a modern web application!

Latency Compensation



To avoid this problem, Meteor introduces a concept called **Latency Compensation**. When we defined our `post` Method, we placed it within a file in the `collections/` directory. This means it is available to both the server *and the client* – and it will run on both at the same time!

When you make a Method call, the client sends off the call to the server, but also simultaneously *simulates* the action of the Method on its client collections. So our workflow now becomes:

- *+0ms*: The user clicks a submit button and the browser fires a Method call.
- *+0ms*: The client simulates the action of the Method call on the client collections and changes the UI to reflect this
- *+200ms*: The server makes changes to the Mongo database.
- *+500ms*: The client receives those changes and undoes its simulated changes, replacing them with the server's changes (which are generally the same). The UI changes to reflect this.

This results in the user seeing the changes instantly. When the server's response returns a few moments later, there may or may not be noticeable changes as the server's canonical documents come down the wire. One thing to learn from this is that we should try to make sure we simulate the real documents as closely as we can.

Observing Latency Compensation

We can make a little change to the `post` method call to see this in action. To do so, we'll use the handy `Meteor._sleepForMs()` function to delay the method call by five seconds, but (crucially) *only on the server*.

We'll use `isServer` to ask Meteor if the Method is currently being invoked on the client (as a “stub”) or on the server. A **stub** is the Method simulation that Meteor runs on the client in parallel, while the “real” Method is being run on the server.

So we'll ask Meteor if the code is being executed on the server. If so, we'll delay things by five seconds and add the string `(server)` at the end of our `post`'s title. If not, we'll add the string

(client):

```
Posts = new Mongo.Collection('posts');

Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    if (Meteor.isServer) {
      postAttributes.title += "(server)";
      // wait for 5 seconds
      Meteor._sleepForMs(5000);
    } else {
      postAttributes.title += "(client)";
    }

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});
```

collections/posts.js

If we were to stop here, the demonstration wouldn't be very conclusive. At this point, it just looks

like the post submit form is pausing for five seconds before redirecting you to the main post list, and not much else is happening.

To understand why, let's go back to the post submit event handler:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});
```

client/templates/posts/post_submit.js

We've placed our `Router.go()` routing call inside the method call's callback. Which means the form is waiting for that method to succeed before redirecting.

Now this would usually be the right course of action. After all, you can't redirect the user before you know if their post submission was valid or not, if only because it would be extremely confusing to be redirected once, and then be redirected again back to the original post submission page to correct your data all within a few seconds.

But for this example's sake, we want to see the results of our actions immediately. So we'll change the routing call to redirect to the `postsList` route (we can't route to the post because we don't

know its `_id` outside the method), take it out from the callback, and see what happens:

```
Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return alert(error.reason);

      // show this result but route anyway
      if (result.postExists)
        alert('This link has already been posted');
    });

    Router.go('postsList');
  }
});
```

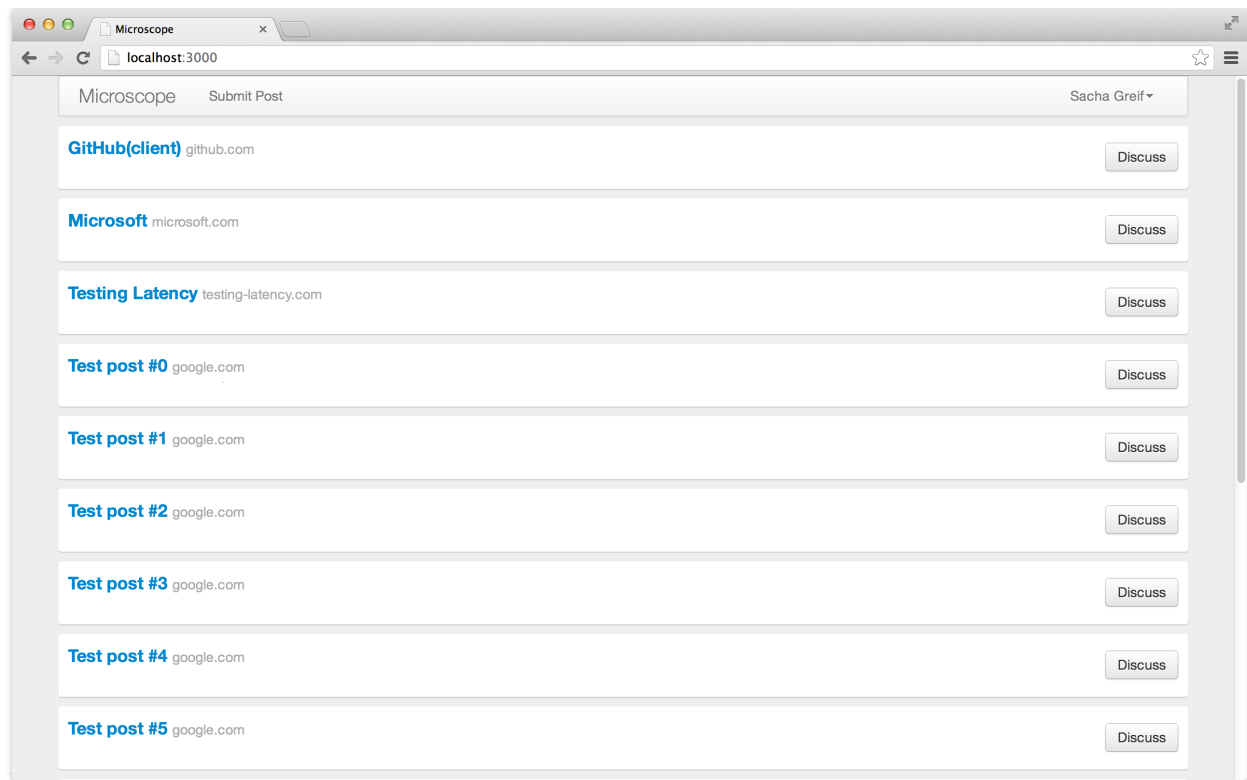
client/templates/posts/post_submit.js

Commit 7-5-1

Demonstrate the order that posts appear using a sleep.

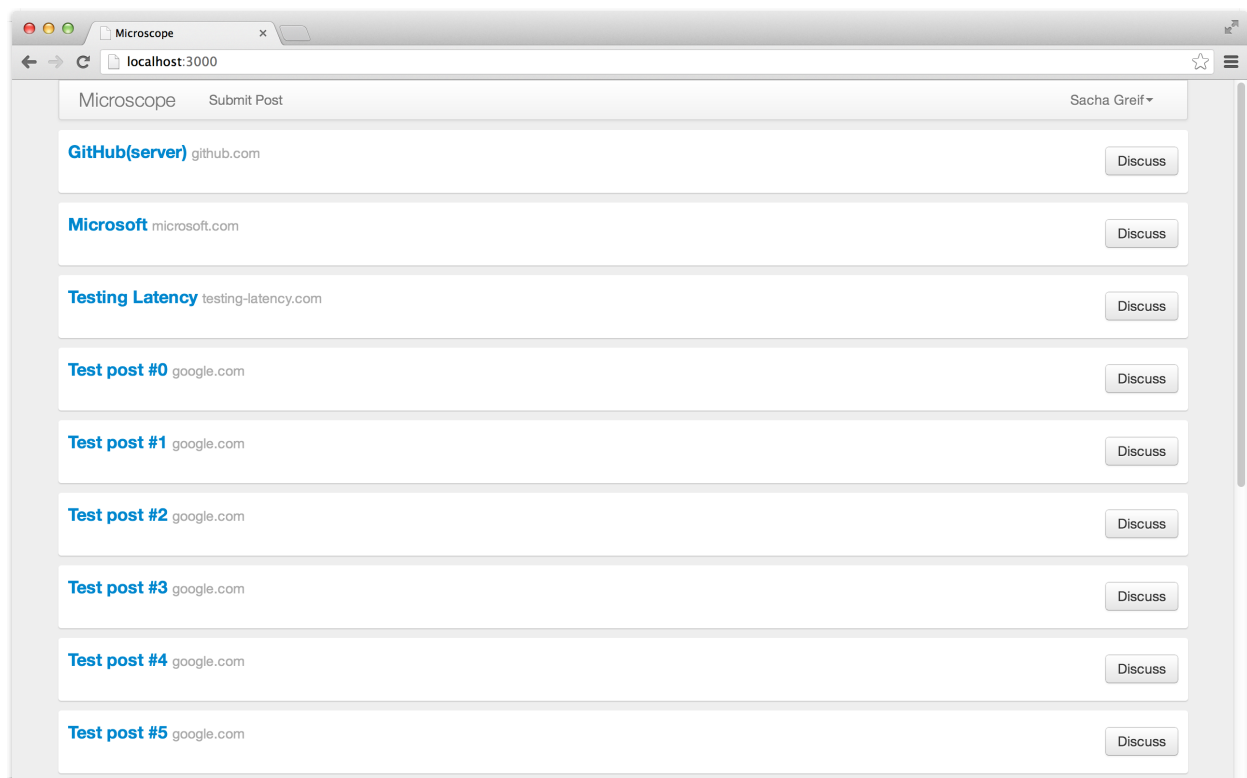
[View on GitHub](#)[Launch Instance](#)

If we create a post now, we see latency compensation clearly. First, a post is inserted with `(client)` in the title (the first post in the list, linking to GitHub):



Our post as first stored in the client collection

Then, five seconds later, it is cleanly replaced with the real document that was inserted by the server:



Our post once the client receives the update from the server collection

Client Collection Methods

You might think that Methods are complicated after this, but in fact they can be quite simple. We've actually seen three very simple Methods already: the collection mutation Methods, `insert`, `update` and `remove`.

When you define a server collection called `'posts'`, you are implicitly defining three Methods: `posts/insert`, `posts/update` and `posts/delete`. In other words, when you call `Posts.insert()` on your client collection, you are calling a latency compensated Method that does two things:

1. Checks to see if we can make the mutation by calling `allow` and `deny` callbacks (this doesn't need to happen in the simulation however).
2. Actually makes the modification to the underlying data store.

Methods Calling Methods

If you are keeping up, you might have just realized that our `post` Method is calling another Method (`posts/insert`) when we insert our post. How does this work?

When the simulation (client-side version of the Method) is being run, we run `insert`'s simulation (so we insert into our client collection), but we *do not* call the real, server-side `insert`, as we expect that the *server-side* version of `post` will do this.

Consequently, when the server-side `post` Method calls `insert` there's no need to worry about simulation, and the insertion goes ahead smoothly.

As before, don't forget to revert your changes before moving on to the next chapter.

Now that we can create posts, the next step is being able to edit and delete them. While the UI code to do so is fairly simple, this is a good time to talk about how Meteor manages user permissions.

Let's first hook up our router. We'll add a route to access the post edit page and set its data context:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() { return Meteor.subscribe('posts'); }
});

Router.route('/', {name: 'postsList'});

Router.route('/posts/:_id', {
  name: 'postPage',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/posts/:_id/edit', {
  name: 'postEdit',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

var requireLogin = function() {
  if (! Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});
```

The Post Edit Template

We can now focus on the template. Our `postEdit` template will be a fairly standard form:

```
<template name="postEdit">
  <form class="main form page">
    <div class="form-group">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="{{url}}" placeholder="Y
our URL" class="form-control"/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="{{title}}" placehol
der="Name your post" class="form-control"/>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary submit"/>
    <hr/>
    <a class="btn btn-danger delete" href="#">Delete post</a>
  </form>
</template>
```

client/templates/posts/post_edit.html

And here's the `post_edit.js` file that goes with it:

```

Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    }

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        alert(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },

  'click .delete': function(e) {

    if (confirm("Delete this post?")) {
      var currentPostId = this._id;
      Posts.remove(currentPostId);
      Router.go('postsList');
    }
  }
});

```

client/templates/posts/post_edit.js

By now most of that code should be familiar to you.

We have two template event callbacks: one for the form's `submit` event, and one for the delete link's `click` event.

The delete callback is extremely simple: suppress the default click event, then ask for confirmation. If you get it, obtain the current post ID from the Template's data context, delete it, and finally redirect the user to the homepage.

The update callback is a little longer, but not much more complicated. After suppressing the default event and getting the current post, we get the new form field values from the page and store them in a `postProperties` object.

We then pass this object to Meteor's `Collection.update()` Method using the `$set` operator (which replaces a set of specified fields while leaving the others untouched), and use a callback that either displays an error if the update failed, or sends the user back to the post's page if the update succeeded.

Adding Links

We should also add edit links to our posts so that users have a way to access the post edit page:

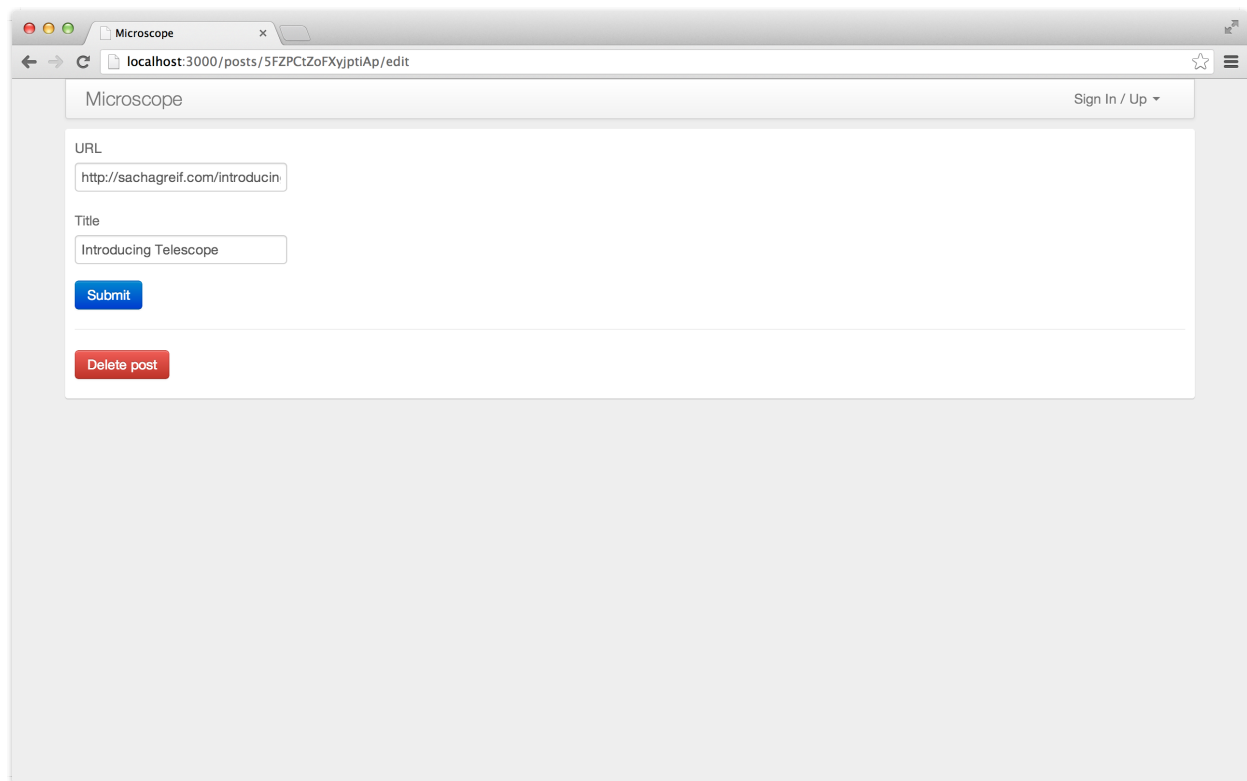
```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        submitted by {{author}}
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  </div>
</template>
```

client/templates/posts/post_item.html

Of course, we don't want to show you an edit link to somebody else's form. This is where the `ownPost` helper comes in:

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

client/templates/posts/post_item.js



Post edit form.

Commit 8-1

Added edit posts form.

[View on GitHub](#)

[Launch Instance](#)

Our post edit form is looking good, but you won't be able to actually edit anything right now.

What's going on?

Setting Up Permissions

Since we've previously removed the `insecure` package, all client-side modifications are currently being denied.

To fix this, we'll set up some permission rules. First, create a new `permissions.js` file inside `lib`. This makes sure our permissions logic loads first (and is available in both environments):

```
// check that the userId specified owns the documents
ownsDocument = function(userId, doc) {
  return doc && doc.userId === userId;
}
```

lib/permissions.js

In the [Creating Posts](#) chapter, we got rid of the `allow()` Methods because we were only inserting new posts via a server Method (which bypasses `allow()` anyway).

But now that we're editing and deleting posts from the client, let's go back to `posts.js` and add this `allow()` block:

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  update: function(userId, post) { return ownsDocument(userId, post); },
  remove: function(userId, post) { return ownsDocument(userId, post); }
});

//...
```

lib/collections/posts.js

Commit 8-2

Added basic permission to check the post's owner.

[View on GitHub](#)[Launch Instance](#)

Limiting Edits

Just because you can edit your own posts, doesn't mean you should be able to edit *every* property. For example, we don't want users to be able to create a post and then assign it to somebody else.

So we'll use Meteor's `deny()` callback to ensure users can only edit specific fields:

```
Posts = new Mongo.Collection('posts');

Posts.allow({
  update: function(userId, post) { return ownsDocument(userId, post); },
  remove: function(userId, post) { return ownsDocument(userId, post); }
});

Posts.deny({
  update: function(userId, post, fieldNames) {
    // may only edit the following two fields:
    return (_.without(fieldNames, 'url', 'title').length > 0);
  }
});

//...
```

lib/collections/posts.js

Commit 8-3

Only allow changing certain fields of posts.

[View on GitHub](#)[Launch Instance](#)

We're taking the `fieldNames` array that contains a list of the fields being modified, and using **Underscore's** `without()` Method to return a sub-array containing the fields that are *not* `url` or `title`.

If everything's normal, that array should be empty and its length should be 0. If someone is trying anything funky, that array's length will be 1 or more, and the callback will return `true` (thus denying the update).

You might have noticed that nowhere in our post editing code do we check for duplicate links. This means a user could submit a link and then edit it to change its URL to bypass that check. The solution to this issue would be to also use a Meteor method for the edit post form, but we'll leave this as an exercise to the reader.

Method Calls vs Client-side Data Manipulation

To create posts, we are using a `postInsert` Meteor Method, whereas to edit and delete them, we are calling `update` and `remove` directly on the client and limiting access via `allow` and `deny`.

While it's tempting to use `allow` and `deny` when things are relatively straightforward and do things directly from the client, the truth is that this approach is prone to security issues and in fact is no longer officially recommended.

So while we cover both techniques here, it's probably always better to use a Method, especially if you start needing to do things that should be outside the user's control (such as timestamping a new post or assigning it to the correct user).

Method calls are also useful in a few other scenarios:

- When you need to know or return values via callback rather than waiting for the reactivity and synchronization to propagate.
- For heavy database functions that would be too expensive to ship a large collection over.
- To summarize or aggregate data (e.g. count, average, sum).

Check out our blog for a more in-depth exploration of this topic.

Meteor's security system allows us to control database modification without having to define Methods every time we want to make changes.

Because we needed to do auxiliary tasks like decorating the post with extra properties and taking special action when the post's URL had already been posted, using a specific `post` Method made a lot of sense when creating a post.

On the other hand, we didn't really need to create new Methods for updating and deleting posts. We just needed to check if the user had permission to do these actions, and this was made easy by `allow` and `deny` callbacks.

Using these callbacks lets us be more declarative about database modifications, and say what kind of updates can be used. The fact that they integrate with the accounts system is an added bonus.

Deprecating Allow & Deny

We're keeping this chapter around for reference's sake, but it's important to note that the allow & deny approach is no longer officially recommended, mainly because it makes it very easy to introduce security failures in your code.

Meteor Methods are a much safer alternative.

Multiple callbacks

We can define as many `allow` callbacks as required. We just need *at least one* of them to return `true` for the given change that is happening. So when `Posts.insert` is called in a browser (no matter if it's from our app's client-side code or from the console), the server will in turn call whatever allowed-`insert` checks it can until it finds one that returns true. If it does not find any, it will not allow the insert, and will return a `403` error to the client.

Similarly, we can define one or more `deny` callbacks. If *any* of those callbacks return `true`, the change will be cancelled and a `403` will be returned. The logic of this means that for a successful `insert`, one or more `allow insert` callback as well as every `deny insert` callback will be executed.

<div> <div>Callback Execution Order</div> <div>↓</div> </div>	<pre> Posts.deny({ update: function(){...} }); Posts.deny({ update: function(){...} }); Posts.allow({ update: function(){...} }); Posts.allow({ update: function(){...} }); </pre>	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>
		<code>false</code>	<code>false</code>	<code>n/e</code>	<code>false</code>
		<code>true</code>	<code>false</code>	<code>n/e</code>	<code>false</code>
		<code>n/e</code>	<code>true</code>	<code>n/e</code>	<code>false</code>
Result		✓	✓	✗	✗

Note: n/e stands for Not Executed

In other words, Meteor moves down the callback list starting first with `deny`, then with `allow`, and executes every callback until one of them returns `true`.

A practical example of this pattern could be having two `allow()` callbacks, one that checks if a post belongs to the current user, and a second one that checks if the current user has admin rights. If the current user is an admin, this ensures they will be able to update any post, since at least one of those callbacks will return `true`.

Latency Compensation

Remember that database mutation Methods (such as `.update()`) are latency compensated, just like any other Method. So for instance, if you try to delete a post that does not belong to you via the browser console, you'll see the post briefly disappear as your local collection loses the document, but then re-appear as the server informs it that, no, in fact the document wasn't deleted.

Of course this behaviour is not a problem when triggered from the console (after all, if users are

going to try and mess with data on the console, it's not really your problem what happens in *their* browser). However, you need to make sure that this doesn't happen in your user interface. For instance, you need to take pains to ensure that you're not showing users delete buttons for documents that they're not allowed to delete.

Thankfully, since you can share permissions code between the client and server (for instance, you could write a library function `canDeletePost(user, post)` and put it in the shared `/lib` directory), doing so usually doesn't require too much extra code.

Server-side Permissions

Remember that the permission system only applies to database mutations initiated from the client. On the server, Meteor assumes that *all* operations are permitted.

This means that if you were to write a server-side `deletePost` Meteor Method that could be called from the client, anybody would be able to delete any post. So you probably don't want to do that unless you checked user permissions within that Method as well.

Merely using the browser's standard `alert()` dialog to warn the user when there's a problem with their submission is a bit dissatisfying, and it certainly doesn't make for great UX. We can do better.

Instead, let's build a more versatile error reporting mechanism that will do a better job of telling the user what's going on without breaking up the flow.

We are going to implement a simple system which displays new errors in the upper-right corner of the window, similar to popular Mac OS app **Growl**.

Introducing Local Collections

To start off, we need to create a collection to store our errors in. Given that the errors are only relevant to the current session and don't need to be persistent in any way, we are going to do something new, and create a *local collection*. What this means is that the `Errors` collection will only exist *in the browser*, and will make no attempt to synchronize back to the server.

To achieve this, we create the error inside the `client` directory (to make the collection client-only), with its MongoDB collection name set to `null` (since this collection's data will never be saved into the server-side database):

```
// Local (client-only) collection
Errors = new Mongo.Collection(null);
```

client/helpers/errors.js

Now that the collection has been created, we can add a `throwError` function which we'll call to add errors to it. We don't need to worry about `allow` or `deny` or any other security concerns, as this collection is "local" to the current user.

```
throwError = function(message) {  
  Errors.insert({message: message});  
};
```

client/helpers/errors.js

The advantage of using a local collection to store the errors is that, like all collections, it's reactive – meaning we can reactively display errors in the same way we display any other collection data.

Displaying Errors

We are going to insert errors at the top of our main layout:

```
<template name="layout">  
  <div class="container">  
    {{> header}}  
    {{> errors}}  
    <div id="main">  
      {{> yield}}  
    </div>  
  </div>  
</template>
```

client/templates/application/layout.html

Let's now create the `errors` and `error` templates in `errors.html` :

```

<template name="errors">
  <div class="errors">
    {{#each errors}}
      {{> error}}
    {{/each}}
  </div>
</template>

<template name="error">
  <div class="alert alert-danger" role="alert">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
    {{message}}
  </div>
</template>

```

client/templates/includes/errors.html

Twin Templates

You'll notice we're putting two templates in a single file. Up to now we've tried to adhere to a "one file, one template" convention, but as far as Meteor is concerned putting all our templates in a single file works just as well (although it would make for a very confusing `main.html`!).

In this case, since both error templates are fairly short, we'll make an exception and put them in the same file to make our repo a bit cleaner.

We just need to implement our template helper and we'll be good to go!

```

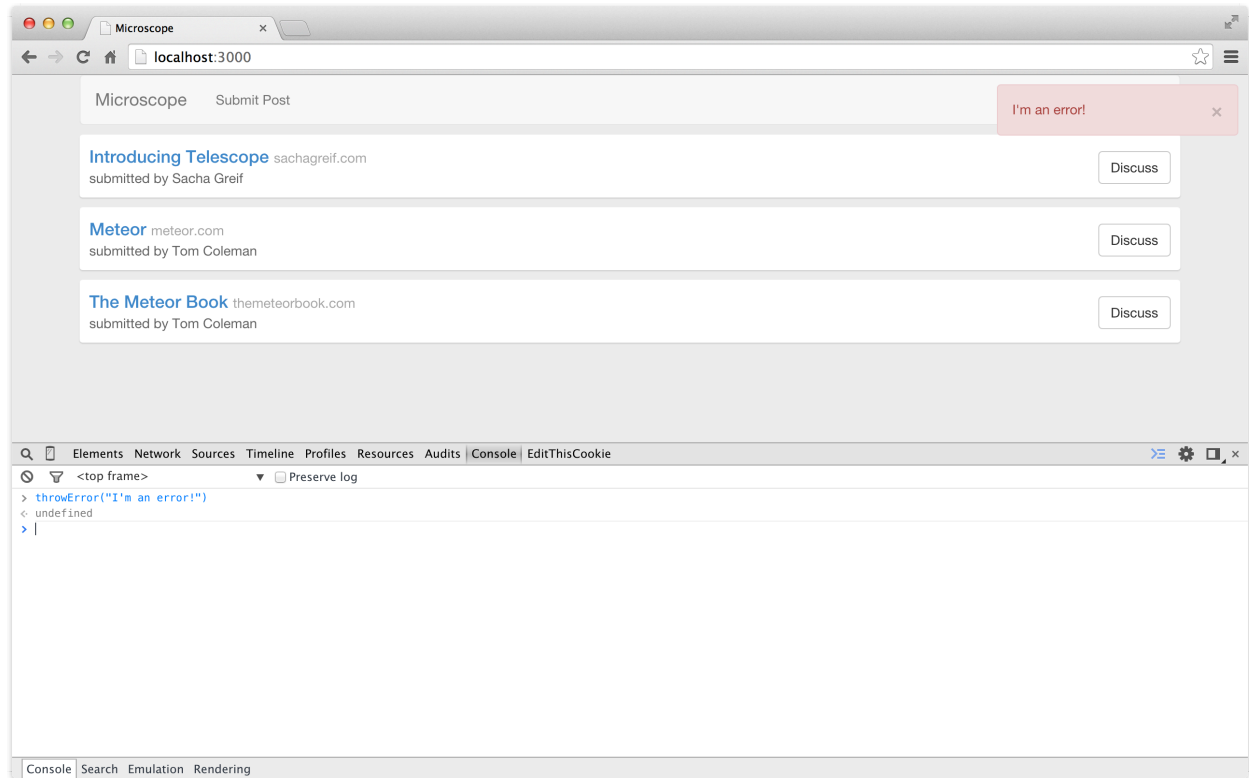
Template.errors.helpers({
  errors: function() {
    return Errors.find();
  }
});

```

client/templates/includes/errors.js

You can already try testing out our new error messages manually. Just open the browser console and type:

```
throwError("I'm an error!");
```



Testing error messages.

Commit 9-1

Basic error reporting.

[View on GitHub](#)[Launch Instance](#)

Two Kinds of Errors

At this point it's important to make a distinction between “app-level” errors and “code-level” errors.

App-level errors are generally user-triggered, and users are in turn able to act upon them. These include things like validation errors, permission errors, “not found” errors, and so on. These are the kind of errors you want to show to the user in order to help them fix whatever problem they've just encountered.

Code-level errors, on the other kind, are unexpectedly triggered by actual bugs in your code, and you probably *don't* want to surface them to users directly, but instead track them with some kind of third-party error-tracking service (such as [Kadira](#)).

In this chapter, we'll focus on dealing with the first type of error, not on catching bugs.

Creating errors

We now know how to display errors, but we still need to trigger one before we'll see anything. We've actually already implemented a good error scenario: our duplicate post warning. We'll simply replace the `alert` calls in the `postSubmit` event helper with the new `throwError` function we just set up:

```

Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return throwError(error.reason);

      // show this result but route anyway
      if (result.postExists)
        throwError('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});

```

client/templates/posts/post_submit.js

While we're at it, we'll do the same thing for the `postEdit` event helper:

```

Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    }

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        throwError(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },
  //...
});

```

client/templates/posts/post_edit.js

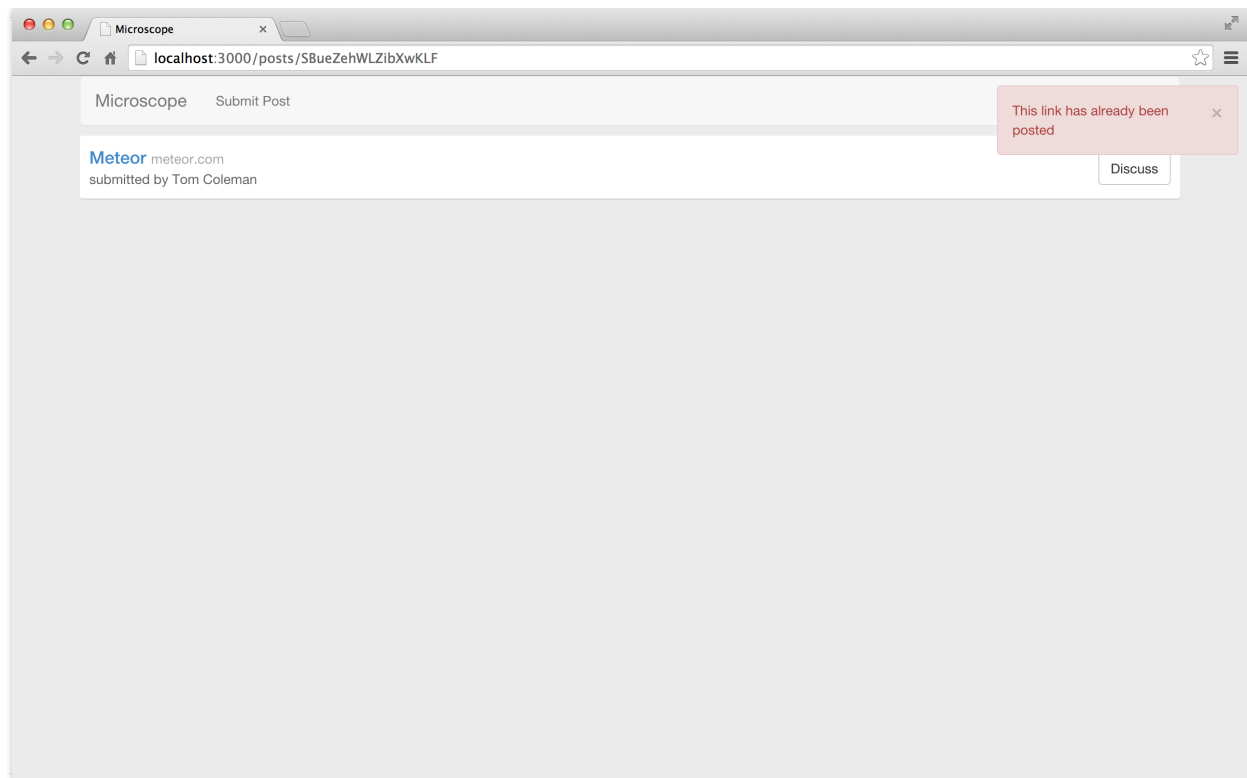
Commit 9-2

Actually use the error reporting.

[View on GitHub](#)

[Launch Instance](#)

Give it a try: try creating a post and entering the URL `http://meteor.com`. As this URL is already attached to a post in the fixtures, you should see:



Triggering an error

Clearing Errors

You'll notice the error messages are disappearing by themselves after a few seconds. This is actually due to a bit of CSS magic included in the stylesheet we added all the way back at the beginning of this book:

```
@keyframes fadeOut {  
  0% {opacity: 0;}  
  10% {opacity: 1;}  
  90% {opacity: 1;}  
  100% {opacity: 0;}  
}  
  
//...  
  
.alert {  
  animation: fadeOut 2700ms ease-in 0s 1 forwards;  
  //...  
}
```

client/stylesheet/style.css

We're defining a `fadeOut` CSS animation that specifies four keyframes for the opacity property (at 0%, 10%, 90%, and 100% of the total animation duration), and applying this animation to the `.alert` class.

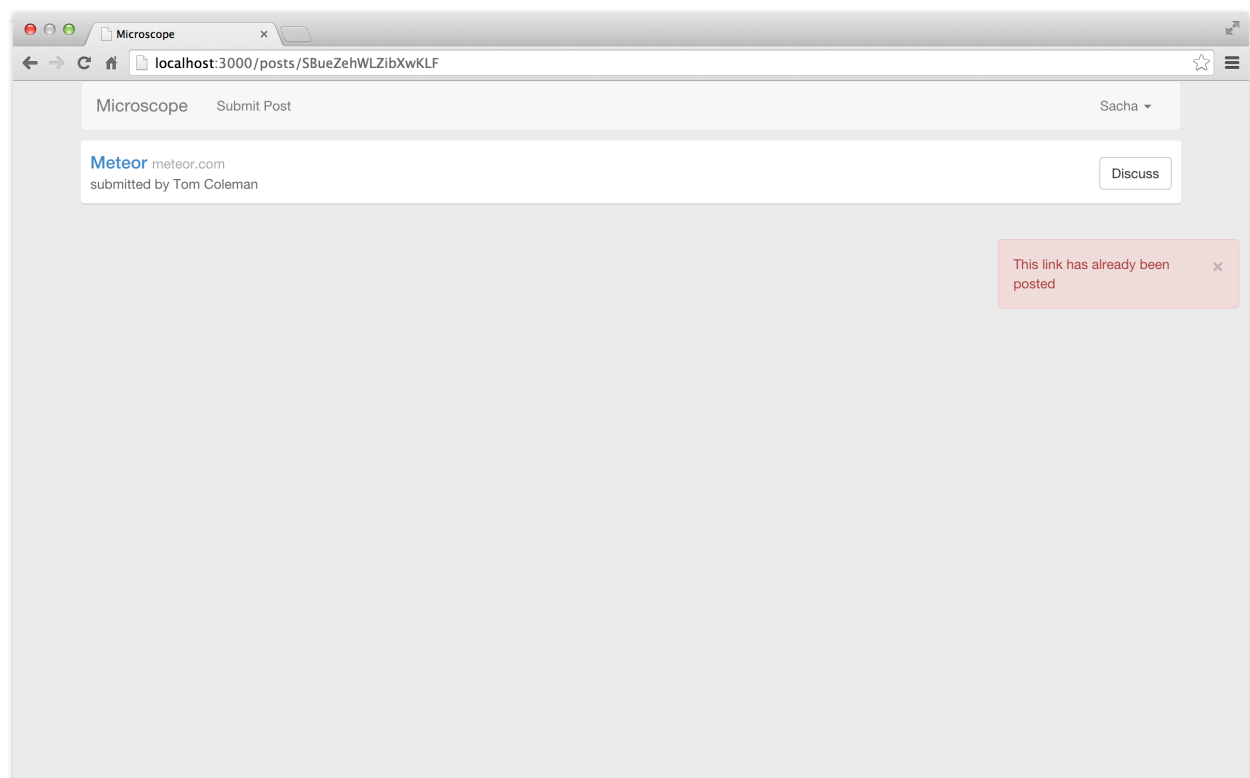
The animation will run for 2700 milliseconds total, use the `ease-in` timing equation, run with a delay of 0 seconds, run only once, and finally stay at the last keyframe once it's done running.

Animations vs Animations

You might be wondering why we're using CSS-based animations (which are pre-determined and outside of our app's control), instead of animations controlled by Meteor itself.

While Meteor does provide support for inserting animations, we wanted this chapter to be focused on errors. So we'll use "dumb" CSS animations for now and we'll leave the fancy stuff for the Animations chapter.

This works, but if you were to trigger multiple errors (by submitting the same link three times, for example) you'd notice that they're getting stacked on top of one another:



This is because while the `.alert` elements are disappearing *visually*, they're still present in the DOM. We need to fix this.

This is exactly the kind of situation where Meteor shines. Since the `Errors` collection is reactive, all we need to do to get rid of these old errors is remove them from the collection!

We'll use `Meteor.setTimeout` to specify a callback function to be executed after the timeout (in this case, 3000 milliseconds) expires.

```
Template.errors.helpers({
  errors: function() {
    return Errors.find();
  }
});

Template.error.onRendered(function() {
  var error = this.data;
  Meteor.setTimeout(function () {
    Errors.remove(error._id);
  }, 3000);
});
```

client/templates/includes/errors.js

Commit 9-3

Clear errors after 3 seconds.

[View on GitHub](#)

[Launch Instance](#)

The `onRendered` callback triggers once our template has been rendered in the browser. Inside the callback, `this` refers to the current template instance, and `this.data` lets us access the data of the object that is currently being rendered (in our case, an error).

So far we haven't enforced any kind of validation on our form. At the minimum, we'll want users to provide both a URL and a title for their new post. So let's make sure they do that.

We'll do two things to point out any missing fields: first, we'll give a special `has-error` CSS class to the parent `div` of any problematic form field. Second, we'll display a helpful error message just below the field.

To get started, let's prep our `postSubmit` template to accept these new helpers:

```
<template name="postSubmit">
  <form class="main form page">
    <div class="form-group {{errorClass 'url'}}">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="" placeholder="Your URL
" class="form-control"/>
        <span class="help-block">{{errorMessage 'url'}}</span>
      </div>
    </div>
    <div class="form-group {{errorClass 'title'}}">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="" placeholder="Name
your post" class="form-control"/>
        <span class="help-block">{{errorMessage 'title'}}</span>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary"/>
  </form>
</template>
```

client/templates/posts/post_submit.html

Note that we're passing parameters (`url` and `title` respectively) to each helper. This lets us reuse the same helper both times, modifying its behavior based on the parameter.

Now for the fun part: making these helpers actually do something.

We'll use the **Session** to store a `postSubmitErrors` object containing any potential error message. As the user interacts with the form, this object will change, which in turn will reactively update the form's markup and contents.

First, we'll initialize the object whenever the `postSubmit` template is created. This ensures that the user won't see old error messages left over from a previous visit to this page.

We then define our two template helpers. They both look at the `field` property of `Session.get('postSubmitErrors')` (where `field` is either `url` or `title` depending on where we're calling the helper from).

While `errorMessage` simply returns the message itself, `errorClass` checks for the *presence* of a message and returns `has-error` if such a message exists.

```
Template.postSubmit.onCreated(function() {
  Session.set('postSubmitErrors', {});
});

Template.postSubmit.helpers({
  errorMessage: function(field) {
    return Session.get('postSubmitErrors')[field];
  },
  errorClass: function(field) {
    return !!Session.get('postSubmitErrors')[field] ? 'has-error' : '';
  }
});

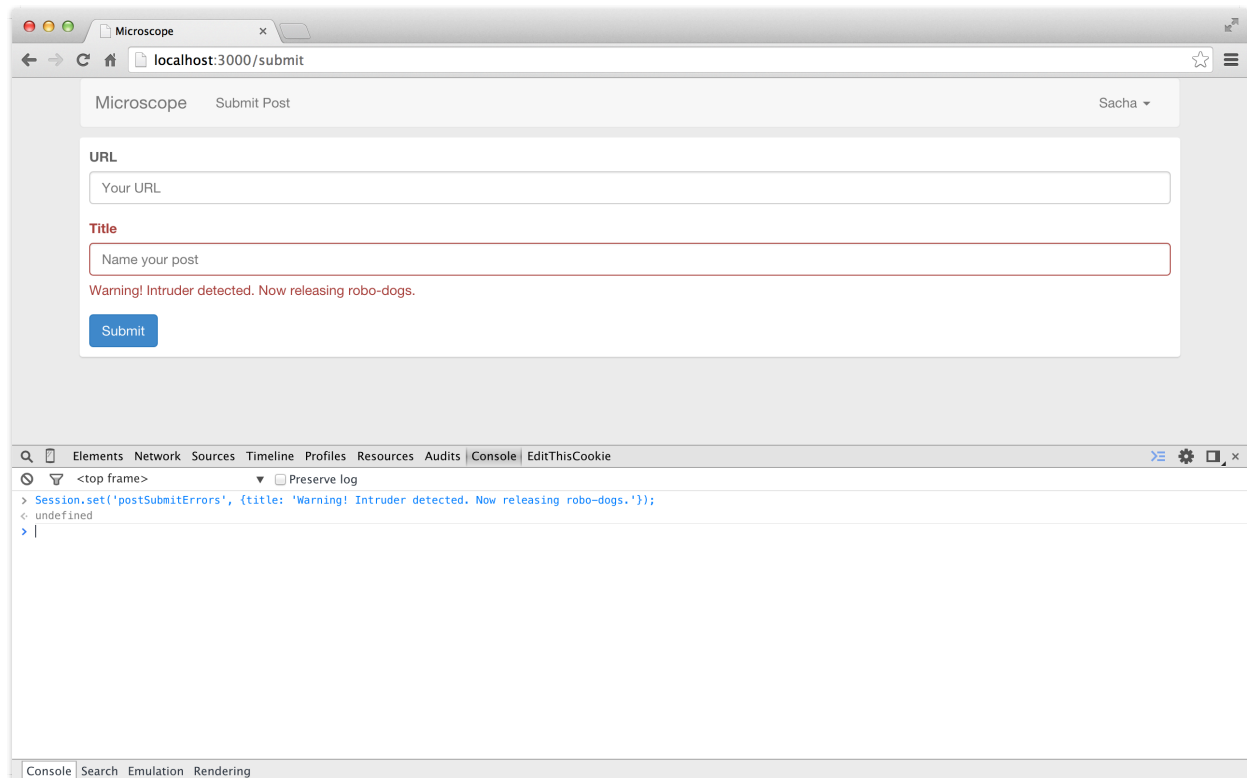
//...
```

client/templates/posts/post_submit.js

You can test that our helpers are working properly by opening the browser console and typing the following line of code:

```
Session.set('postSubmitErrors', {title: 'Warning! Intruder detected. Now releasing robo-dogs.'});
```

Browser console



Red alert! Red alert!

The next step is actually hooking up that `postSubmitErrors` Session object to the form.

Before doing so, we'll create a new `validatePost` function in `posts.js` that looks at a `post` object, and returns an `errors` object containing any relevant errors (namely, whether the `title` or `url` fields are missing):

```
//...

validatePost = function (post) {
  var errors = {};

  if (!post.title)
    errors.title = "Please fill in a headline";

  if (!post.url)
    errors.url = "Please fill in a URL";

  return errors;
}

//...
```

lib/collections/posts.js

We'll call this function from the `postSubmit` event helper:

```

Template.postSubmit.events({
  'submit form': function(e) {
    e.preventDefault();

    var post = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    };

    var errors = validatePost(post);
    if (errors.title || errors.url)
      return Session.set('postSubmitErrors', errors);

    Meteor.call('postInsert', post, function(error, result) {
      // display the error to the user and abort
      if (error)
        return throwError(error.reason);

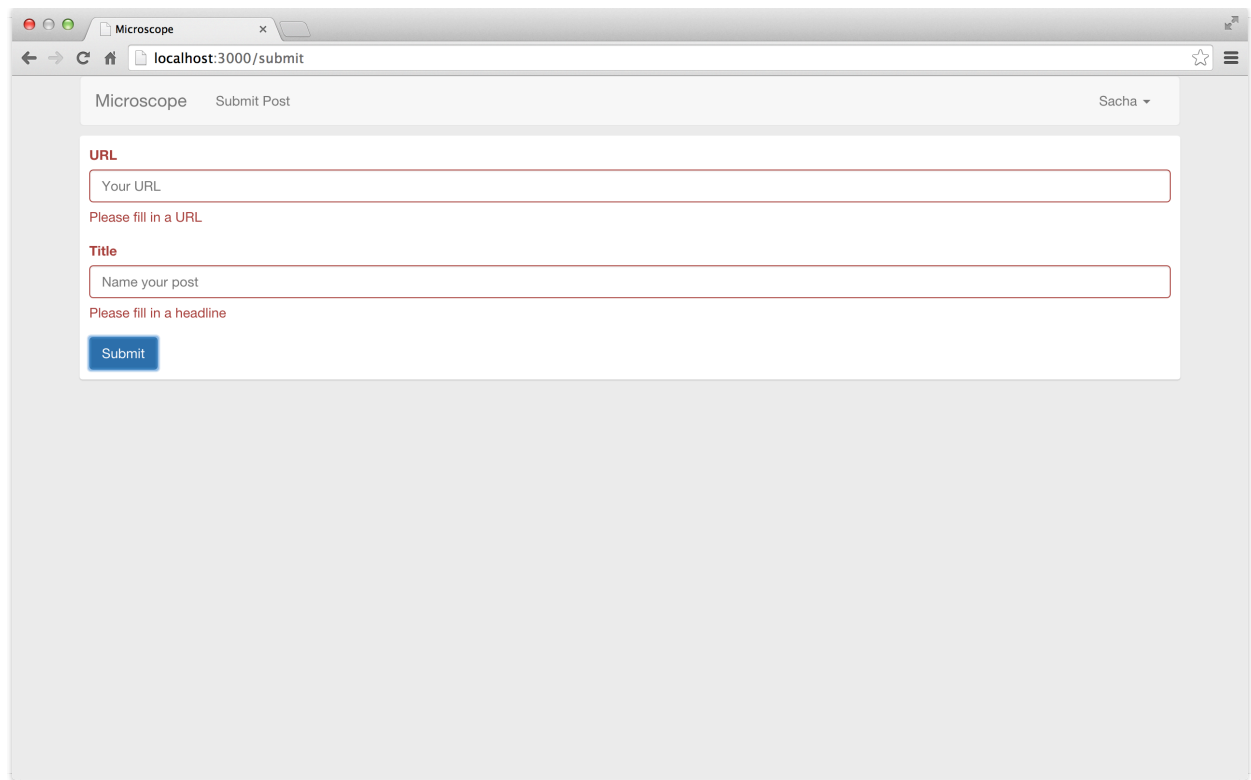
      // show this result but route anyway
      if (result.postExists)
        throwError('This link has already been posted');

      Router.go('postPage', {_id: result._id});
    });
  }
});

```

client/templates/posts/post_submit.js

Notice that we're using `return` to abort the execution of the helper if any errors are present, not because we want to actually return this value anywhere.



Caught red-handed.

Server-side Validation

We're not quite done though. We're validating the presence of a URL and title on the *client*, but what about the *server*? After all, someone could still try to enter an empty post by manually calling the `postInsert` method through the browser console.

Even though we don't need to display any error messages on the server, we can still make use of that same `validatePost` function. Except this time we'll call it from within the `postInsert` *method* too, and not just the event helper:

```

Meteor.methods({
  postInsert: function(postAttributes) {
    check(this.userId, String);
    check(postAttributes, {
      title: String,
      url: String
    });

    var errors = validatePost(postAttributes);
    if (errors.title || errors.url)
      throw new Meteor.Error('invalid-post', "You must set a title and URL for
your post");

    var postWithSameLink = Posts.findOne({url: postAttributes.url});
    if (postWithSameLink) {
      return {
        postExists: true,
        _id: postWithSameLink._id
      }
    }

    var user = Meteor.user();
    var post = _.extend(postAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    var postId = Posts.insert(post);

    return {
      _id: postId
    };
  }
});

```

lib/collections/posts.js

Again, users should normally never have to see this “You must set a title and URL for your post” message. It will only ever show up if someone wants to bypass the user interface we painstakingly put together, and uses the console directly instead.

To test this out, open up the browser console and try entering a post with no URL:

```
Meteor.call('postInsert', {url: '', title: 'No URL here!'});
```

If we've done our job properly, you'll get back a scary bunch of code along with the message "You must set a title and URL for your post".

Commit 9-4

Validate post contents on submission.

[View on GitHub](#)[Launch Instance](#)

Edition Validation

To round things up, we'll also apply the same validation to our post *edit* form. The code will look pretty similar. First, the template:

```

<template name="postEdit">
  <form class="main form page">
    <div class="form-group {{errorClass 'url'}}">
      <label class="control-label" for="url">URL</label>
      <div class="controls">
        <input name="url" id="url" type="text" value="{{url}}" placeholder="Your URL" class="form-control"/>
        <span class="help-block">{{errorMessage 'url'}}</span>
      </div>
    </div>
    <div class="form-group {{errorClass 'title'}}">
      <label class="control-label" for="title">Title</label>
      <div class="controls">
        <input name="title" id="title" type="text" value="{{title}}" placeholder="Name your post" class="form-control"/>
        <span class="help-block">{{errorMessage 'title'}}</span>
      </div>
    </div>
    <input type="submit" value="Submit" class="btn btn-primary submit"/>
    <hr/>
    <a class="btn btn-danger delete" href="#">Delete post</a>
  </form>
</template>

```

client/templates/posts/post_edit.html

Then the template helpers:


```

Template.postEdit.onCreate(function() {
  Session.set('postEditErrors', {});
});

Template.postEdit.helpers({
  errorMessage: function(field) {
    return Session.get('postEditErrors')[field];
  },
  errorClass: function (field) {
    return !!Session.get('postEditErrors')[field] ? 'has-error' : '';
  }
});

Template.postEdit.events({
  'submit form': function(e) {
    e.preventDefault();

    var currentPostId = this._id;

    var postProperties = {
      url: $(e.target).find('[name=url]').val(),
      title: $(e.target).find('[name=title]').val()
    }

    var errors = validatePost(postProperties);
    if (errors.title || errors.url)
      return Session.set('postEditErrors', errors);

    Posts.update(currentPostId, {$set: postProperties}, function(error) {
      if (error) {
        // display the error to the user
        throwError(error.reason);
      } else {
        Router.go('postPage', {_id: currentPostId});
      }
    });
  },

  'click .delete': function(e) {
    e.preventDefault();

    if (confirm("Delete this post?")) {
      var currentPostId = this._id;
      Posts.remove(currentPostId);
      Router.go('postsList');
    }
  }
});

```

```
client/templates/posts/post_edit.js
```

Just like we did for the post submit form, we'll also want to validate our posts on the server. Except that you'll remember we're not using a method to edit posts, but an `update` call directly from the client.

This means we'll have to add a new `deny` callback instead:

```
//...

Posts.deny({
  update: function(userId, post, fieldNames, modifier) {
    var errors = validatePost(modifier.$set);
    return errors.title || errors.url;
  }
});

//...
```

```
lib/collections/posts.js
```

Note that the `post` argument refers to the *existing* post. In this case, we want to validate the *update*, which is why we're calling `validatePost` on the contents of the `modifier`'s `$set` property (as in `Posts.update({$set: {title: ..., url: ...}})`).

This works because `modifier.$set` contains the same two `title` and `url` property as the whole `post` object would. Of course, it does mean that any partial update affecting only `title` or only `url` will fail, but in practice that shouldn't be an issue.

You might notice this is our second `deny` callback. When adding multiple `deny` callbacks, the operation will fail if any one of them returns `true`. In this case, this means the `update` will only succeed if it's only targeting the `title` and `url` fields, as well as if neither one of these fields are empty.

Commit 9-5

Validate post contents when editing.

[View on GitHub](#)[Launch Instance](#)

We've built a re-usable pattern with our errors work, so why not package it up into a smart package and share it with the rest of the Meteor community?

To get started, we need to make sure we have a Meteor Developer account. You can go claim yours at meteor.com, but there's a good chance you already did so when you signed up for the book! In either case, you should figure out what your username is, as we'll make heavy use of it in this chapter.

We'll use the username `tmeasday` in this chapter – you can substitute your own in for it.

First we need to create some structure for our package to reside in. We can use the `meteor create --package tmeasday:errors` command to do so. Note that Meteor has created a folder named `packages/tmeasday:errors/`, with some files inside. We'll start by editing `package.js`, the file that informs Meteor of how the package should be used, and which objects or functions it needs to export.

```
Package.describe({
  name: "tmeasday:errors",
  summary: "A pattern to display application errors to the user",
  version: "1.0.0",
  documentation: null
});

Package.onUse(function (api, where) {
  api.versionsFrom('0.9.0');

  api.use(['minimongo', 'mongo-livedata', 'templating'], 'client');

  api.addFiles(['errors.js', 'errors_list.html', 'errors_list.js'], 'client');

  if (api.export)
    api.export('Errors');
});
```

`packages/tmeasday:errors/package.js`

When developing a package for real-world use, it's good practice to fill the `Package.describe` block's `git` section with your repo's Git URL (such as `https://github.com/tmeasday/meteor-errors.git`). This way users can read the source code, and (assuming you are using GitHub) your package's readme will appear on Atmosphere.

Let's add three files to the package. (We can remove the boilerplate that Meteor put down) We can pull these files from Microscope without much change except for some proper namespacing and a slightly cleaner API:

```
Errors = {  
  // Local (client-only) collection  
  collection: new Mongo.Collection(null),  
  
  throw: function(message) {  
    Errors.collection.insert({message: message, seen: false})  
  }  
};
```

packages/tmeasday:errors/errors.js

```
<template name="meteorErrors">  
  <div class="errors">  
    {{#each errors}}  
      {{> meteorError}}  
    {{/each}}  
  </div>  
</template>  
  
<template name="meteorError">  
  <div class="alert alert-danger" role="alert">  
    <button type="button" class="close" data-dismiss="alert">&times;</button>  
    {{message}}  
  </div>  
</template>
```

packages/tmeasday:errors/errors_list.html

```
Template.meteorErrors.helpers({
  errors: function() {
    return Errors.collection.find();
  }
});

Template.meteorError.onRendered(function() {
  var error = this.data;
  Meteor.setTimeout(function () {
    Errors.collection.remove(error._id);
  }, 3000);
});
```

packages/tmeasday:errors/errors_list.js

Testing the package out with Microscope

We will now test things locally with Microscope to ensure our changed code works. To link the package into our project, we run `meteor add tmeasday:errors`. Then, we need to delete the existing files that have been made redundant by the new package:

```
rm client/helpers/errors.js
rm client/templates/includes/errors.html
rm client/templates/includes/errors.js
```

removing old files on the bash console

One other thing we need to do is to make some minor updates to use the correct API:

```
{{> header}}
{{> meteorErrors}}
```

client/templates/application/layout.html

```
//...

Meteor.call('postInsert', post, function(error, result) {
  // display the error to the user and abort
  if (error)
    return Errors.throw(error.reason);

  // show this result but route anyway
  if (result.postExists)
    Errors.throw('This link has already been posted');

  Router.go('postPage', {_id: result._id});
});

//...
```

client/templates/posts/post_submit.js

```
//...

Posts.update(currentPostId, {$set: postProperties}, function(error) {
  if (error) {
    // display the error to the user
    Errors.throw(error.reason);
  } else {
    Router.go('postPage', {_id: currentPostId});
  }
});

//...
```

client/templates/posts/post_edit.js

Commit 9-5-1

Created basic errors package and linked it in.

[View on GitHub](#)[Launch Instance](#)

Once these changes have been made, we should get our original pre-package behaviour back.

Writing Tests

The first step in developing a package is testing it against an application, but the next is to write a test suite that properly tests the package's behaviour. Meteor itself comes with Tinytest (a built in package tester), which makes it easy to run such tests and maintain peace of mind when sharing our package with others.

Let's create a test file that uses Tinytest to run some tests against the errors codebase:

```
Tinytest.add("Errors - collection", function(test) {
  test.equal(Errors.collection.find({}).count(), 0);

  Errors.throw('A new error!');
  test.equal(Errors.collection.find({}).count(), 1);

  Errors.collection.remove({});
});

Tinytest.addAsync("Errors - template", function(test, done) {
  Errors.throw('A new error!');
  test.equal(Errors.collection.find({}).count(), 1);

  // render the template
  Blaze.insert(Blaze.render(Template.meteorErrors), document.body);

  Meteor.setTimeout(function() {
    test.equal(Errors.collection.find({}).count(), 0);
    done();
  }, 3500);
});
```

packages/tmeasday:errors/errors_tests.js

In these tests we're checking the basic `Meteor.Errors` functions work, as well as double checking that the `onRendered` code in the template is still functioning.

We won't cover the specifics of writing Meteor package tests here (as the API is not yet finalized and highly in flux), but hopefully it's fairly self explanatory how it works.

To tell Meteor how to run the tests in `package.js`, use the following code:


```
Package.onTest(function(api) {  
  api.use('tmeasday:errors', 'client');  
  api.use(['templating', 'tinytest', 'test-helpers'], 'client');  
  
  api.addFiles('errors_tests.js', 'client');  
});
```

packages/tmeasday:errors/package.js

Commit 9-5-2

Added tests to the package.

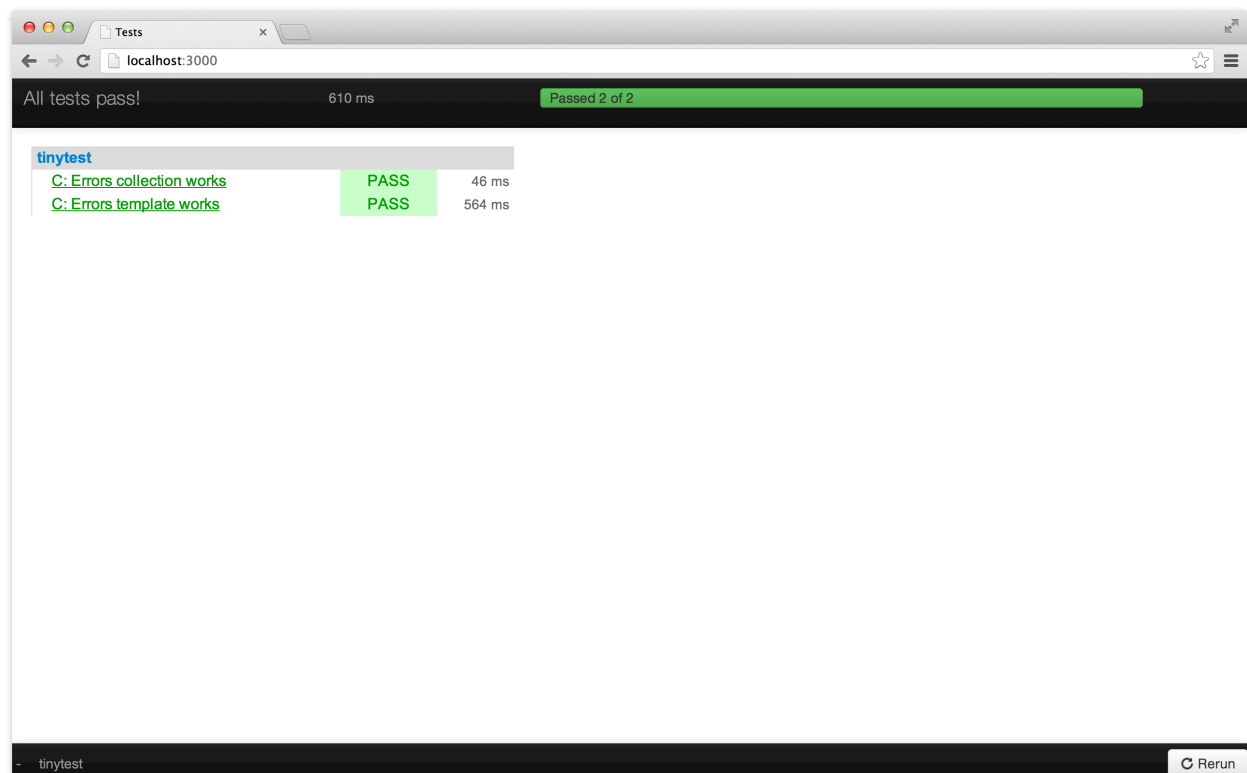
[View on GitHub](#)

[Launch Instance](#)

Then we can run the tests with:

```
meteor test-packages tmeasday:errors
```

Terminal



Releasing the package

Now, we want to release the package and make it available to the world. We do this by pushing it to Meteor's package server, and getting it onto Atmosphere.

Luckily, it's very easy. We just `cd` into the package's directory, and run `meteor publish --create`:

```
cd packages/tmeasday:errors
meteor publish --create
```

Terminal

Now that the package is released, we can now delete it from the project and then add it back in directly:

```
rm -r packages/errors
meteor add tmeasday:errors
```

Terminal (run from the top level of the app)

Commit 9-5-4

Removed package from development tree.

[View on GitHub](#)[Launch Instance](#)

Now we should see Meteor download our package for the very first time. Well done!

As usual with sidebar chapters, make sure you revert your changes before moving on (or else be sure to account for them when following along the rest of the book).

The goal of a social news site is to create a community of users, and it will be hard to do that without providing a way for people to talk to each other. So in this chapter, let's add comments!

We'll begin by creating a new collection to store comments in, and adding some basic fixture data into that collection.

```
Comments = new Mongo.Collection('comments');
```

lib/collections/comments.js

```
// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // create two users
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachaId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sacha = Meteor.users.findOne(sachaId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sacha._id,
    author: sacha.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: new Date(now - 7 * 3600 * 1000)
  });

  Comments.insert({
    postId: telescopeId,
    userId: tom._id,
    author: tom.profile.name,
    submitted: new Date(now - 5 * 3600 * 1000),
    body: 'Interesting project Sacha, can I get involved?'
  });
}
```

```

Comments.insert({
  postId: telescopeId,
  userId: sach._id,
  author: sach.profile.name,
  submitted: new Date(now - 3 * 3600 * 1000),
  body: 'You sure can Tom!'
});

Posts.insert({
  title: 'Meteor',
  userId: tom._id,
  author: tom.profile.name,
  url: 'http://meteor.com',
  submitted: new Date(now - 10 * 3600 * 1000)
});

Posts.insert({
  title: 'The Meteor Book',
  userId: tom._id,
  author: tom.profile.name,
  url: 'http://themetorbook.com',
  submitted: new Date(now - 12 * 3600 * 1000)
});
}

```

server/fixtures.js

Let's not forget to publish and subscribe to our new collection:

```

Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function() {
  return Comments.find();
});

```

server/publications.js

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('posts'), Meteor.subscribe('comments')];
  }
});
```

lib/router.js

Commit 10-1

Added comments collection, pub/sub and fixtures.

[View on GitHub](#)[Launch Instance](#)

Note that to trigger this fixture code, you'll need to `meteor reset` to clear your database. After resetting, don't forget to create a new account and log back in!

First, we created a couple of (completely fake) users, inserting them into the database and using their `ids` to select them out of the database afterwards. Then we added a comment for each user on the first post, linking the comment to the post (with `postId`), and the user (with `userId`). We also added a submission date and body to each comment, along with `author`, a denormalized field.

Also, we augmented our router to wait on an *array* containing both the comments and the posts subscriptions.

Displaying comments

It's all very well putting comments into the database, but we also need to show them on the discussion page. Hopefully this process should be familiar to you by now, and you have an idea of the steps involved:

```

<template name="postPage">
  <div class="post-page page">
    {{> postItem}}

    <ul class="comments">
      {{#each comments}}
        {{> commentItem}}
      {{/each}}
    </ul>
  </div>
</template>

```

client/templates/posts/post_page.html

```

Template.postPage.helpers({
  comments: function() {
    return Comments.find({postId: this._id});
  }
});

```

client/templates/posts/post_page.js

We put the `{{#each comments}}` block inside the post template, so `this` is a post within the `comments` helper. To find the relevant comments, we check those that are linked to that post via the `postId` attribute.

Given what we've learned about helpers and Spacebars, rendering a comment is fairly straightforward. We'll create a new `comments` directory inside `templates` to store all our comment information, and a new `commentItem` template inside that:

```

<template name="commentItem">
  <li>
    <h4>
      <span class="author">{{author}}</span>
      <span class="date">on {{submittedText}}</span>
    </h4>
    <p>{{body}}</p>
  </li>
</template>

```

```
client/templates/comments/comment_item.html
```

Let's set up a quick template helper to format our `submitted` date to a more user-friendly format:

```
Template.commentItem.helpers({
  submittedText: function() {
    return this.submitted.toString();
  }
});
```

```
client/templates/comments/comment_item.js
```

Then, we'll show the number of comments on each post:

```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        submitted by {{author}},
        <a href="{{pathFor 'postPage'}}">{{commentsCount}} comments</a>
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  </div>
</template>
```

```
client/templates/posts/post_item.html
```

And add the `commentsCount` helper to `post_item.js` :


```

Template.postItem.helpers({
  ownPost: function() {
    return this.userId === Meteor.userId();
  },
  domain: function() {
    var a = document.createElement('a');
    a.href = this.url;
    return a.hostname;
  },
  commentsCount: function() {
    return Comments.find({postId: this._id}).count();
  }
});

```

client/templates/posts/post_item.js

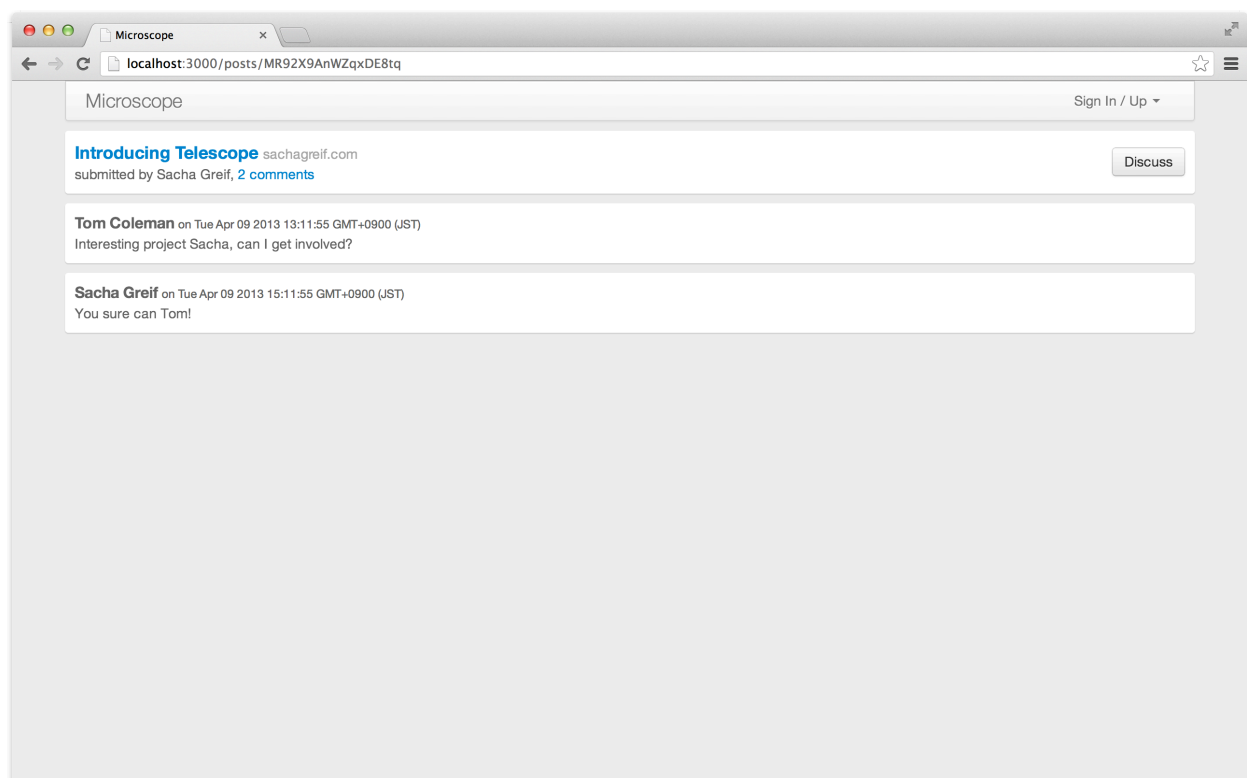
Commit 10-2

Display comments on `postPage`.

[View on GitHub](#)

[Launch Instance](#)

You should now be able to display our fixture comments and see something like this:



Submitting Comments

Let's add a way for our users to create new comments. The process we'll follow will be pretty similar to how we've already allowed users to create new posts.

We'll start by adding a submit box at the bottom of each post:

```
<template name="postPage">
  <div class="post-page page">
    {{> postItem}}

    <ul class="comments">
      {{#each comments}}
        {{> commentItem}}
      {{/each}}
    </ul>

    {{#if currentUser}}
      {{> commentSubmit}}
    {{else}}
      <p>Please log in to leave a comment.</p>
    {{/if}}
  </div>
</template>
```

client/templates/posts/post_page.html

And then create the comment form template:

```
<template name="commentSubmit">
  <form name="comment" class="comment-form form">
    <div class="form-group {{errorClass 'body'}}">
      <div class="controls">
        <label for="body">Comment on this post</label>
        <textarea name="body" id="body" class="form-control" rows="3"></tex
        tarea>
        <span class="help-block">{{errorMessage 'body'}}</span>
      </div>
    </div>
    <button type="submit" class="btn btn-primary">Add Comment</button>
  </form>
</template>
```

client/templates/comments/comment_submit.html

To submit our comments, we call a `comment` Method in `comment_submit.js` that operates in a similar way to what we did for submitting posts:

```

Template.commentSubmit.onCreated(function() {
  Session.set('commentSubmitErrors', {});
});

Template.commentSubmit.helpers({
  errorMessage: function(field) {
    return Session.get('commentSubmitErrors')[field];
  },
  errorClass: function (field) {
    return !!Session.get('commentSubmitErrors')[field] ? 'has-error' : '';
  }
});

Template.commentSubmit.events({
  'submit form': function(e, template) {
    e.preventDefault();

    var $body = $(e.target).find('[name=body]');
    var comment = {
      body: $body.val(),
      postId: template.data._id
    };

    var errors = {};
    if (! comment.body) {
      errors.body = "Please write some content";
      return Session.set('commentSubmitErrors', errors);
    }

    Meteor.call('commentInsert', comment, function(error, commentId) {
      if (error){
        throwError(error.reason);
      } else {
        $body.val('');
      }
    });
  }
});

```

client/templates/comments/comment_submit.js

Just like we previously set up a `post` server-side Meteor Method, we'll set up a `comment` Meteor Method to create our comments, check that everything is legit, and finally insert the new comment into the comments collection.

```

Comments = new Mongo.Collection('comments');

Meteor.methods({
  commentInsert: function(commentAttributes) {
    check(this.userId, String);
    check(commentAttributes, {
      postId: String,
      body: String
    });

    var user = Meteor.user();
    var post = Posts.findOne(commentAttributes.postId);

    if (!post)
      throw new Meteor.Error('invalid-comment', 'You must comment on a post');

    comment = _.extend(commentAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    return Comments.insert(comment);
  }
});

```

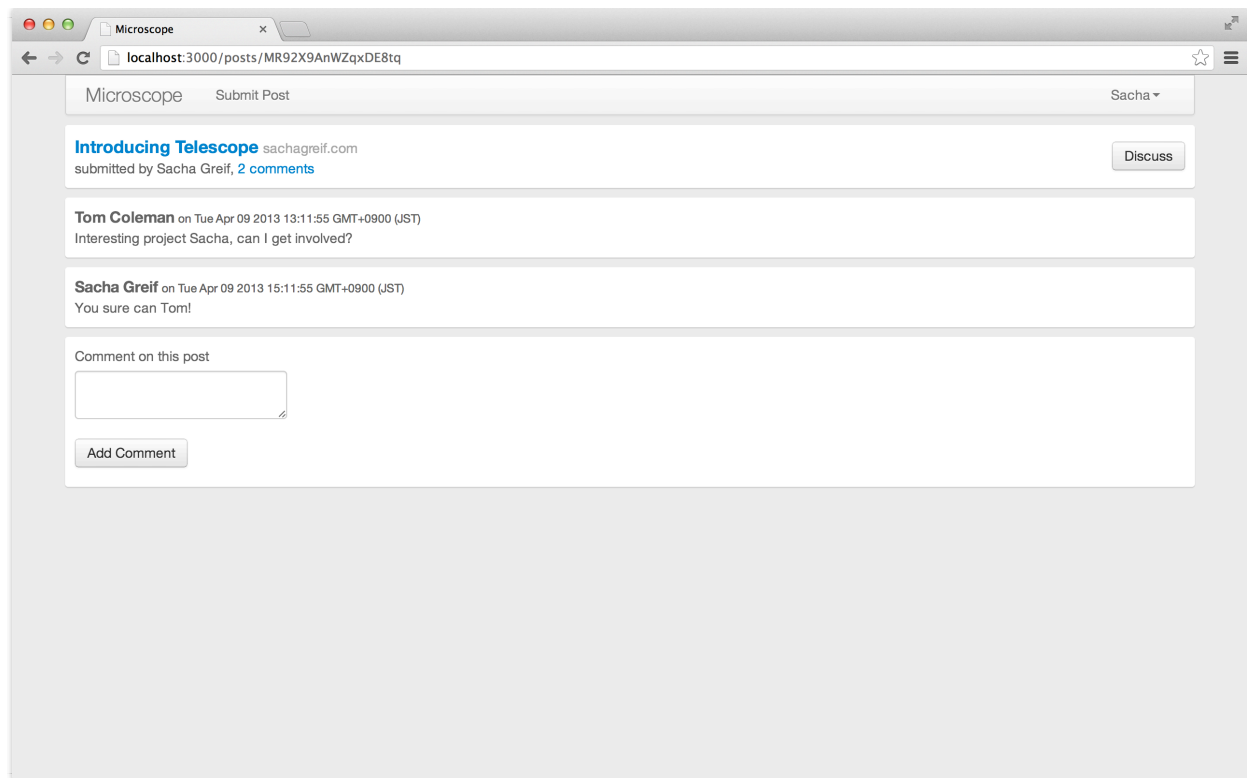
lib/collections/comments.js

Commit 10-3

Created a form to submit comments.

[View on GitHub](#)
[Launch Instance](#)

This is not doing anything too fancy, just checking that the user is logged in, that the comment has a body, and that it's linked to a post.



The comment submit form

Controlling the Comments Subscription

As things stand, we are publishing all comments across all posts to all connected clients. That seems a little wasteful. After all, we're only actually using a small subset of this data at any given time. So let's improve our publication and subscription to control exactly which comments are published.

If we think about it, the only time we need to subscribe to our `comments` publication is when a user accesses a post's individual page, and we only need to load the subset of comments related to that particular post.

The first step will be changing the way we subscribe to comments. Up to now, we've been subscribing at the *router* level, which means we load all our data once when the router is initialized.

But we now want our subscription to depend on a path parameter, and that parameter can obviously change at any point. So we'll need to move our subscription code from the *router* level to the *route* level.

This has another consequence: instead of loading our data when we initialize our app, we'll now be loading it whenever we hit our *route*. This means that you'll now get loading times while browsing within the app, but it's an unavoidable downside unless you intend to front-load the entirety of your data set forever.

First, we'll stop preloading all comments in the `configure` block by removing `Meteor.subscribe('comments')` (in other words, going back to what we had previously):

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return Meteor.subscribe('posts');
  }
});
```

lib/router.js

And we'll add a new *route*-level `waitOn` function just for the `postPage` route:

```
//...

Router.route('/posts/:_id', {
  name: 'postPage',
  waitOn: function() {
    return Meteor.subscribe('comments', this.params._id);
  },
  data: function() { return Posts.findOne(this.params._id); }
});

//...
```

lib/router.js

We're passing `this.params._id` as an argument to the subscription. So let's use that new information to make sure we restrict our data set to comments belonging to the current post:

```
Meteor.publish('posts', function() {  
  return Posts.find();  
});  
  
Meteor.publish('comments', function(postId) {  
  check(postId, String);  
  return Comments.find({postId: postId});  
});
```

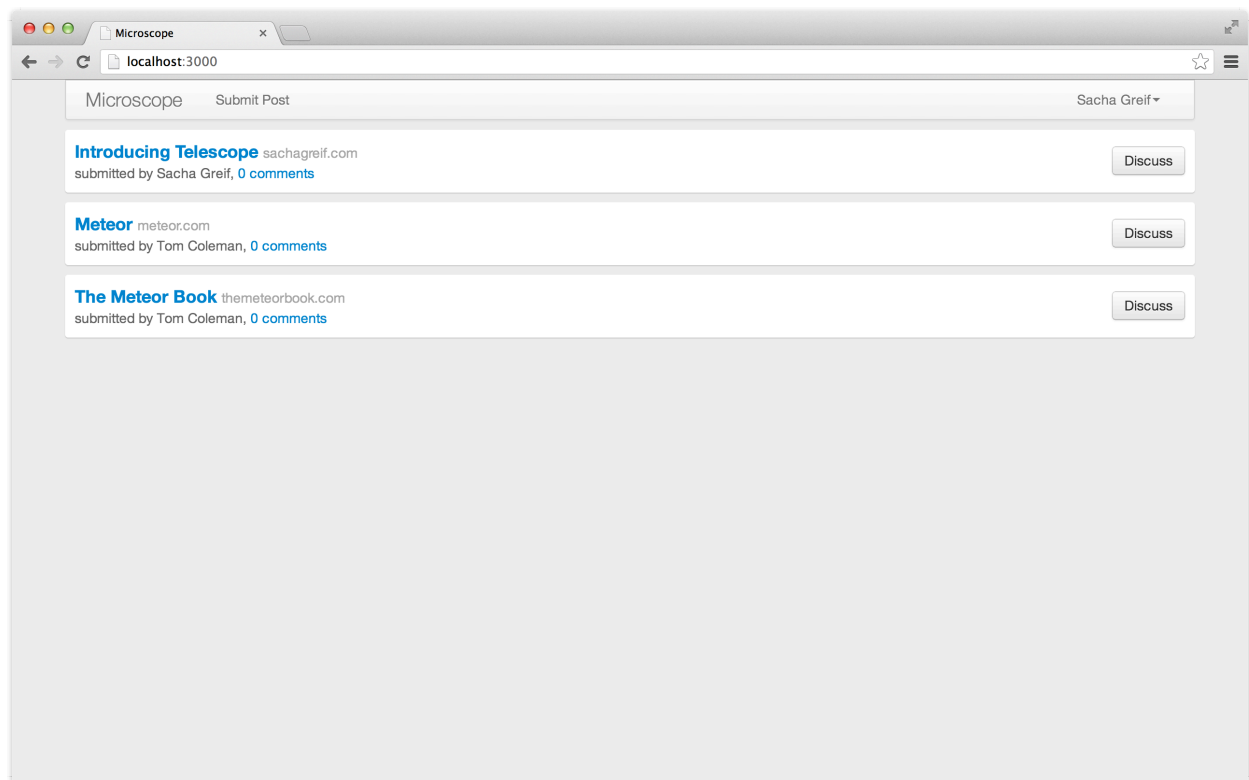
server/publications.js

Commit 10-4

Made a simple publication/subscription for comments.

[View on GitHub](#)[Launch Instance](#)

There's only one problem: when we return to the homepage, it claims that all our posts have 0 comments:



Our comments are gone!

Counting Comments

The reason for this will quickly become clear: we only load comments on the `postPage` route, so when we call `Comments.find({postId: this._id})` in the `commentsCount` helper, Meteor can't find the necessary client-side data to provide us with a result.

The best way to deal with this is to *denormalize* the number of comments onto the post (if you're not sure what that means don't worry, the next sidebar has got you covered!). Although as we'll see, there's a minor addition of complexity in our code, the performance benefit we gain from not having to publish *all* the comments to display the post list is worth it.

We'll achieve this by adding a `commentsCount` property to the `post` data structure. To begin with, we update our post fixtures (and `meteor reset` to reload them – don't forget to recreate your user account after):

```
// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();

  // create two users
  var tomId = Meteor.users.insert({
    profile: { name: 'Tom Coleman' }
  });
  var tom = Meteor.users.findOne(tomId);
  var sachId = Meteor.users.insert({
    profile: { name: 'Sacha Greif' }
  });
  var sach = Meteor.users.findOne(sachId);

  var telescopeId = Posts.insert({
    title: 'Introducing Telescope',
    userId: sach._id,
    author: sach.profile.name,
    url: 'http://sachagreif.com/introducing-telescope/',
    submitted: new Date(now - 7 * 3600 * 1000),
    commentsCount: 2
  });

  Comments.insert({
    postId: telescopeId,
    userId: tom._id,
    author: tom.profile.name,
```

```

    submitted: new Date(now - 5 * 3600 * 1000),
    body: 'Interesting project Sacha, can I get involved?'
  });

Comments.insert({
  postId: telescopeId,
  userId: sacha._id,
  author: sacha.profile.name,
  submitted: new Date(now - 3 * 3600 * 1000),
  body: 'You sure can Tom!'
});

Posts.insert({
  title: 'Meteor',
  userId: tom._id,
  author: tom.profile.name,
  url: 'http://meteor.com',
  submitted: new Date(now - 10 * 3600 * 1000),
  commentsCount: 0
});

Posts.insert({
  title: 'The Meteor Book',
  userId: tom._id,
  author: tom.profile.name,
  url: 'http://themetorbook.com',
  submitted: new Date(now - 12 * 3600 * 1000),
  commentsCount: 0
});
}

```

server/fixtures.js

As usual when updating the fixtures file, you'll have to `meteor reset` your database to make sure it runs again.

Then, we make sure that all new posts start with 0 comments:

```
//...

var post = _.extend(postAttributes, {
  userId: user._id,
  author: user.username,
  submitted: new Date(),
  commentsCount: 0
});

var postId = Posts.insert(post);

//...
```

lib/collections/posts.js

And then we update the relevant `commentsCount` when we make a new comment using Mongo's `$inc` operator (which increments a numeric field by one):

```
//...

comment = _.extend(commentAttributes, {
  userId: user._id,
  author: user.username,
  submitted: new Date()
});

// update the post with the number of comments
Posts.update(comment.postId, {$inc: {commentsCount: 1}});

return Comments.insert(comment);

//...
```

lib/collections/comments.js

Finally, we can just simply remove the `commentsCount` helper from `client/templates/posts/post_item.js`, as the field is now directly available on the post.

Commit 10-5

Denormalized the number of comments into the post.

[View on GitHub](#)[Launch Instance](#)

Now that users can talk to each other, it would be a shame if they missed out on new comments. And what do you know, the next chapter will show you how to implement notifications to prevent just this!

Denormalizing data means not storing that data in a “normal” form. In other words, denormalization means having multiple copies of the same piece of data hanging about.

In the last chapter, we denormalized the count of the number of comments into the post object to avoid having to load all the comments all the time. In a data modelling sense this is redundant, as we could instead just count the correct set of comments at any time to figure out that value (leaving out performance considerations).

Denormalizing often means extra work for the developer. In our example, every time we add or remove a comment we’ll also need to remember to update the relevant post to ensure that the `commentsCount` field stays accurate. This is exactly why relational databases such as MySQL frown upon this approach.

However, the normal approach also has its drawbacks: without a `commentsCount` property, we’d need to send *all* comments down the wire at all times just to be able to count them, which is what we were doing in the beginning. Denormalizing lets us avoid this entirely.

A Special Publication

It *would* be possible to create a special publication that only sends down the comment counts that we are interested in (i.e. comment counts of posts that we can currently see, via aggregate queries on the server).

But it’s worth considering if the complexity of such publication code would not outweigh the difficulties created by denormalizing...

Of course, such considerations are application-specific: if you are writing code where data integrity is of paramount importance, then avoiding data inconsistencies is far more important and of a higher priority to you than performance gains.

If you are experienced with Mongo, you might have been surprised to see that we created a second collection just for comments: why not just embed the comments in a list within the post document?

It turns out that many of the tools Meteor gives us work a lot better when operating at the collection level. For example:

1. The `{{#each}}` helper is very efficient when iterating over a cursor (the result of `collection.find()`). The same is not true when it iterates over an array of objects within a larger document.
2. `allow` and `deny` operate at the document level, and thus make it easy to ensure that any modifications of individual comments are correct in a way that would be more complex if we operated at a post level.
3. DDP operates at the level of top-level attributes of a document—this would mean if `comments` was a property of a `post`, every time a comment was created on a post, the server would send the entire updated comment list of that post out to each connected client.
4. Publications and subscriptions are a lot easier to control at the level of documents. For example, if we wanted to paginate comments on a post we would find it difficult to do so unless comments were in their own collection.

Mongo suggests embedding documents in order to reduce the number of expensive queries to fetch documents. However, this is less of an issue when we take into account Meteor's architecture: most of the time we are querying comments on the *client*, where database access is essentially free.

The Downsides of Denormalization

There's a good argument to be made that you *shouldn't* denormalize your data. For a good look at the case against denormalization, we recommend [Why You Should Never Use MongoDB](#) by Sarah Mei.

Now that users can comment on each other's posts, it'd be good to let them know that a conversation has begun.

To do so, we'll notify the post's owner that there's been a comment on their post, and provide them with a link to view that comment.

This is the kind of feature where Meteor really shines: because Meteor is realtime by default, we'll be displaying those notifications *instantly*. We don't need to wait for the user to refresh the page or check in any way, we can simply pop new notifications up without ever writing any special code.

Creating Notifications

We'll create a notification when someone comments on your posts. In the future, notifications could be extended to cover many other scenarios, but for now this will be enough to keep users informed of what's going on.

Let's create our `Notifications` collection, as well as a `createCommentNotification` function that will insert a matching notification for each new comment on one of your own posts.

Since we'll be updating notifications from the client, we need to make sure our `allow` call is bulletproof. So we'll check that:

- The user making the `update` call owns the notification being modified.
- The user is only trying to update a single field.
- That single field is the `read` property of our notifications.


```

Notifications = new Mongo.Collection('notifications');

Notifications.allow({
  update: function(userId, doc, fieldNames) {
    return ownsDocument(userId, doc) &&
      fieldNames.length === 1 && fieldNames[0] === 'read';
  }
});

createCommentNotification = function(comment) {
  var post = Posts.findOne(comment.postId);
  if (comment.userId !== post.userId) {
    Notifications.insert({
      userId: post.userId,
      postId: post._id,
      commentId: comment._id,
      commenterName: comment.author,
      read: false
    });
  }
};

```

lib/collections/notifications.js

Just like posts or comments, this `Notifications` collection will be shared by both client and server. As we need to update notifications once a user has seen them, we also enable updates, ensuring as usual that we restrict update permissions to a user's own data.

We've also created a simple function that looks at the post that the user is commenting on, discovers who should be notified from there, and inserts a new notification.

We are already creating comments in a server-side Method, so we can just augment that Method to call our function. We'll replace `return Comments.insert(comment);` by `comment._id = Comments.insert(comment)` in order to save the `_id` of the newly created comment in a variable, then call our `createCommentNotification` function:

```

Comments = new Mongo.Collection('comments');

Meteor.methods({
  commentInsert: function(commentAttributes) {

    //...

    comment = _.extend(commentAttributes, {
      userId: user._id,
      author: user.username,
      submitted: new Date()
    });

    // update the post with the number of comments
    Posts.update(comment.postId, {$inc: {commentsCount: 1}});

    // create the comment, save the id
    comment._id = Comments.insert(comment);

    // now create a notification, informing the user that there's been a commen
    t
    createCommentNotification(comment);

    return comment._id;
  }
});

```

lib/collections/comments.js

Let's also publish the notifications:

```

Meteor.publish('posts', function() {
  return Posts.find();
});

Meteor.publish('comments', function(postId) {
  check(postId, String);
  return Comments.find({postId: postId});
});

Meteor.publish('notifications', function() {
  return Notifications.find();
});

```

server/publications.js

And subscribe on the client:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('posts'), Meteor.subscribe('notifications')]
  }
});
```

lib/router.js

Commit 11-1

Added basic notifications collection.

[View on GitHub](#)[Launch Instance](#)

Displaying Notifications

Now we can go ahead and add a list of notifications to the header.

```

<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'postsList'}}">Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav">
        {{#if currentUser}}
          <li>
            <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
          </li>
          <li class="dropdown">
            {{> notifications}}
          </li>
        {{/if}}
      </ul>
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>

```

client/templates/includes/header.html

And create the `notifications` and `notificationItem` templates (they'll share a single `notifications.html` file):

```

<template name="notifications">
  <a href="#" class="dropdown-toggle" data-toggle="dropdown">
    Notifications
    {{#if notificationCount}}
      <span class="badge badge-inverse">{{notificationCount}}</span>
    {{/if}}
    <b class="caret"></b>
  </a>
  <ul class="notification dropdown-menu">
    {{#if notificationCount}}
      {{#each notifications}}
        {{> notificationItem}}
      {{/each}}
    {{else}}
      <li><span>No Notifications</span></li>
    {{/if}}
  </ul>
</template>

<template name="notificationItem">
  <li>
    <a href="{{notificationPostPath}}">
      <strong>{{commenterName}}</strong> commented on your post
    </a>
  </li>
</template>

```

client/templates/notifications/notifications.html

We can see that the plan is for each notification to contain a link to the post that was commented on, and the name of the user that commented on it.

Next, we need to make sure we select the right list of notifications in our helper, and update the notifications as “read” when the user clicks on the link to which they point.

```

Template.notifications.helpers({
  notifications: function() {
    return Notifications.find({userId: Meteor.userId(), read: false});
  },
  notificationCount: function(){
    return Notifications.find({userId: Meteor.userId(), read: false}).count();
  }
});

Template.notificationItem.helpers({
  notificationPostPath: function() {
    return Router.routes.postPage.path({_id: this.postId});
  }
});

Template.notificationItem.events({
  'click a': function() {
    Notifications.update(this._id, {$set: {read: true}});
  }
});

```

client/templates/notifications/notifications.js

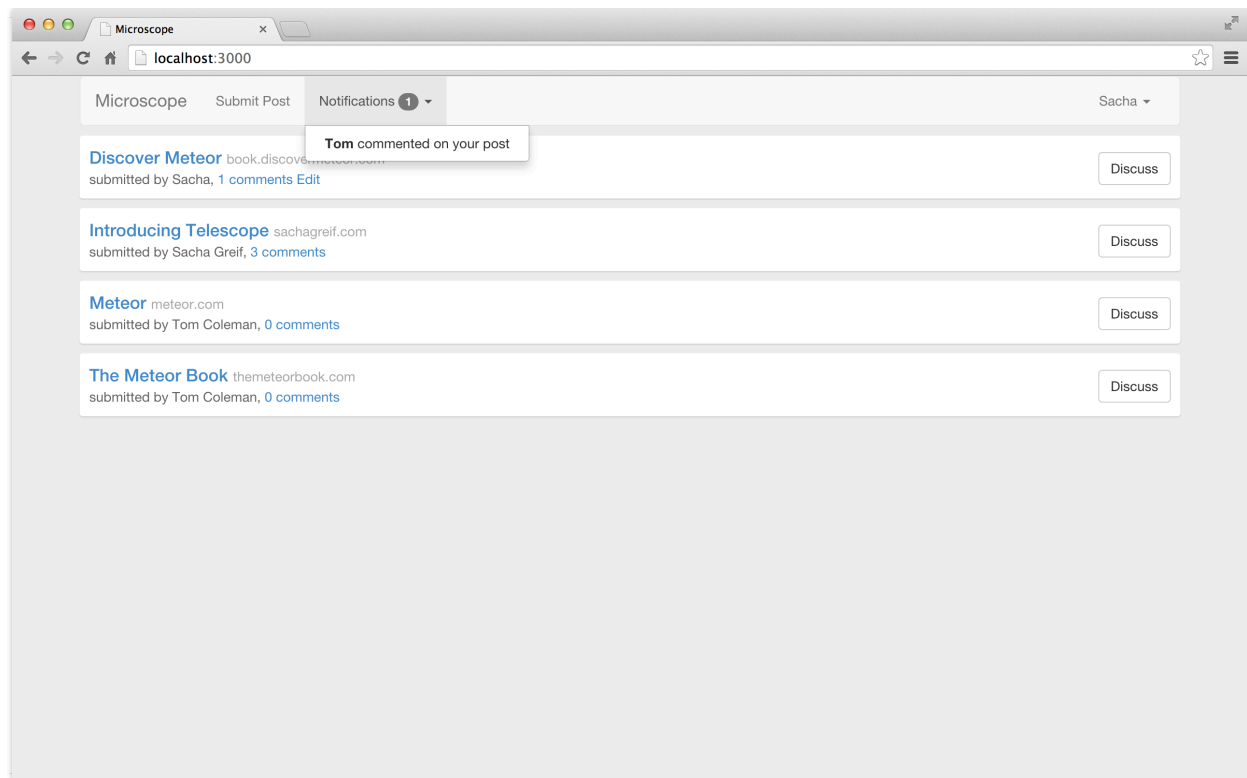
Commit 11-2

Display notifications in the header.

[View on GitHub](#)
[Launch Instance](#)

You may think that the notifications are not too different from the errors, and it's true that their structure is very similar. There is one key difference though: we've created a proper client-server synchronised collection. This means that our notifications are *persistent* and, as long as we use the same user account, will exist across browser refreshes and different devices.

Give it a try: open up a second browser (let's say Firefox), create a new user account, and comment on a post that you've created with your main account (which you've left open in Chrome). You should see something like this:



Displaying notifications.

Controlling access to notifications

Notifications are working well. However there's just a small problem: our notifications are public.

If you still have your second browser open, try running the following code in the browser console:

```
> Notifications.find().count();  
1
```

Browser console

This new user (the one that *commented*) shouldn't have any notifications. The notification they can see in the `Notifications` collection actually belongs to our *original* user.

Aside from potential privacy issues, we simply can't afford to have every user's notifications loaded in every other user's browser. On a big enough site, this could overload the browser's available memory and start causing serious performance problems.

We solve this issue with **publications**. We can use our publications to specify precisely which part of our collection we want to share with each browser.

To accomplish this, we need to return a different cursor in our publication than

`Notifications.find()`. Namely, we want to return a cursor that corresponds to the current user's notifications.

Doing so is straightforward enough, as a `publish` function has the current user's `_id` available at `this.userId`:

```
Meteor.publish('notifications', function() {  
  return Notifications.find({userId: this.userId, read: false});  
});
```

server/publications.js

Commit 11-3

Only sync notifications that are relevant to the user.

[View on GitHub](#)[Launch Instance](#)

Now if we check in our two browser windows, we should see two different notifications collections:

```
> Notifications.find().count();  
1
```

Browser console (user 1)

```
> Notifications.find().count();  
0
```

Browser console (user 2)

In fact, the list of Notifications should even change as you log in and out of the app. This is because publications automatically re-publish whenever the user account changes.

Our app is becoming more and more functional, and as more users join and start posting links we run the risk of ending up with a never-ending homepage. We'll address this in the next chapter by implementing pagination.

It's rare to need to write dependency tracking code yourself, but it's certainly useful to understand it to trace the way that the flow of dependency resolution works.

Imagine we wanted to track how many of the current user's Facebook friends have "liked" each post on Microscope. Let's assume we've already worked out the details of how to authenticate the user with Facebook, make the appropriate API calls, and parse the relevant data. We now have an asynchronous client-side function that returns the number of likes, `getFacebookLikeCount(user, url, callback)`.

The important thing to remember about such a function is that it is very much *non-reactive* and non-realtime. It will make an HTTP request to Facebook, retrieve some data, and make it available to the application in an asynchronous callback, but the function won't re-run by itself when that count changes over at Facebook, and our UI won't change when the underlying data does.

To fix this, we can start by using `setInterval` to call our function every few seconds:

```
currentLikeCount = 0;
Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId).url,
      function(err, count) {
        if (!err)
          currentLikeCount = count;
      });
  }
}, 5 * 1000);
```

Any time we check that `currentLikeCount` variable, we can expect to get the correct number with a five seconds margin of error. We can now use this variable in a helper like so:

```
Template.postItem.likeCount = function() {  
  return currentLikeCount;  
}
```

However, nothing yet tells our template to re-draw when `currentLikeCount` changes. Although the variable is now pseudo-realtime in that it changes by itself, it's not *reactive* so it still can't quite communicate properly with the rest of the Meteor ecosystem.

Tracking Reactivity: Computations

Meteor's reactivity is mediated by *dependencies*, data structures that track a set of computations.

As we saw in the earlier reactivity sidebar, a computation is a section of code that uses reactive data. In our case, there's a computation that's been implicitly created for the `postItem` template, and every helper on that template's manager has it's own computation as well.

You can think of the computation as the section of code that “cares” about the reactive data. When the data changes, it will be this computation that is informed (via `invalidate()`), and it's the computation that decides whether something needs to be done.

Turning a Variable Into a Reactive Function

To turn our `currentLikeCount` variable into a reactive data source, we need to track all of the computations that use it in a dependency. This requires changing it from a variable into a function (which will return a value):

```

var _currentLikeCount = 0;
var _currentLikeCountListeners = new Tracker.Dependency();

currentLikeCount = function() {
  _currentLikeCountListeners.depend();
  return _currentLikeCount;
}

Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId),
      function(err, count) {
        if (!err && count !== _currentLikeCount) {
          _currentLikeCount = count;
          _currentLikeCountListeners.changed();
        }
      });
  }
}, 5 * 1000);

```

What we've done is setup a `_currentLikeCountListeners` dependency, which tracks all the computations within which `currentLikeCount()` has been used. When the value of `_currentLikeCount` changes, we call the `changed()` function on that dependency, which invalidates all the tracked computations.

These computations can then go ahead and deal with the change on a case-by-case basis.

If that seemed like a lot of boilerplate for a simple reactive data source, you're right, and Meteor provides some built in tools to make it a bit easier (just like you don't have to use computations directly, you usually just use autoruns). There's a platform package called `reactive-var` that does exactly what our `currentLikeCount()` function is doing. If we add it:

```
meteor add reactive-var
```

The we can use it to simplify our code a bit:

```

var currentLikeCount = new ReactiveVar();

Meteor.setInterval(function() {
  var postId;
  if (Meteor.user() && postId = Session.get('currentPostId')) {
    getFacebookLikeCount(Meteor.user(), Posts.find(postId),
      function(err, count) {
        if (!err) {
          currentLikeCount.set(count);
        }
      });
  }
}, 5 * 1000);

```

Now to use it, we would call `currentLikeCount.get()` in our helper and it would work as before. There's also another platform package `reactive-dict`, which provides a reactive key-value store (almost exactly like the `Session`), which can be useful too.

Comparing Tracker to Angular

Angular is a client-side only reactive rendering library, developed by the good folks at Google. It's illustrative to compare Meteor's approach to dependency tracking to Angular's, as the approaches are quite different.

We've seen that Meteor's model uses blocks of code called computations. These computations are tracked by special "reactive" data sources (functions) that take care of invalidating them when appropriate. So the data source *explicitly* informs all of its dependencies when they need to call `invalidate()`. Note that although this is generally when data has changed, the data source could potentially also decide to trigger invalidation for other reasons.

Additionally, although computations usually just re-run when invalidated, you can set them up to behave any way you want. All this gives us a high level of control over reactivity.

In Angular, reactivity is mediated by the `scope` object. A scope can be thought of as plain JavaScript object with a couple of special methods.

When you want to reactively depend on a value in a scope, you call `scope.$watch`, providing the

expression that you are interested in (i.e. which parts of the scope you care about) and a listener function that will run every time that expression changes. So you explicitly state exactly what you want to do every time the value of the expression changes.

Going back to our Facebook example, we would write:

```
$rootScope.$watch('currentLikeCount', function(likeCount) {  
  console.log('Current like count is ' + likeCount);  
});
```

Of course, just like you rarely set up computations in Meteor, you don't often call `$watch` explicitly in Angular as `ng-model` directives and `{{expressions}}` automatically set up watches that then take care of re-rendering on change.

When such a reactive value has changed, `scope.$apply()` must then be called. This re-evaluates every watcher of the scope, but only calls the listener function of watchers whose expression's value has *changed*.

So `scope.$apply()` is similar to `dependency.changed()`, except that it acts at the level of the scope, rather than giving you the control to say precisely which listeners should be re-evaluated. That being said, this slight lack of control gives Angular the ability to be very smart and efficient in the way it determines precisely which listeners need to be re-evaluated.

With Angular, our `getFacebookLikeCount()` function code would've looked something like this:

```
Meteor.setInterval(function() {  
  getFacebookLikeCount(Meteor.user(), Posts.find(postId),  
    function(err, count) {  
      if (!err) {  
        $rootScope.currentLikeCount = count;  
        $rootScope.$apply();  
      }  
    }  
  });  
}, 5 * 1000);
```

Admittedly, Meteor takes care of most of the heavy lifting for us and lets us benefit from reactivity without much work on our part. But hopefully, learning about these patterns will prove helpful if you ever need to push things further.

Things are looking great with Microscope, and we can expect a hit reception when it's released to the world.

So we should probably think a little about the performance implication of the number of new posts that will be entered into the site as it takes off!

We've spoken before about how a client-side collection should contain a subset of the data on the server, and we've even managed to achieve this for our notification and comments collections.

At present though, we are still publishing all of our posts in one go, to all connected users. Eventually, if thousands of links are posted, this will become problematic. To solve this, we need to paginate our posts.

Adding More Posts

First, in our fixture data, let's load up enough posts so that pagination actually makes sense:


```

// Fixture data
if (Posts.find().count() === 0) {

  //...

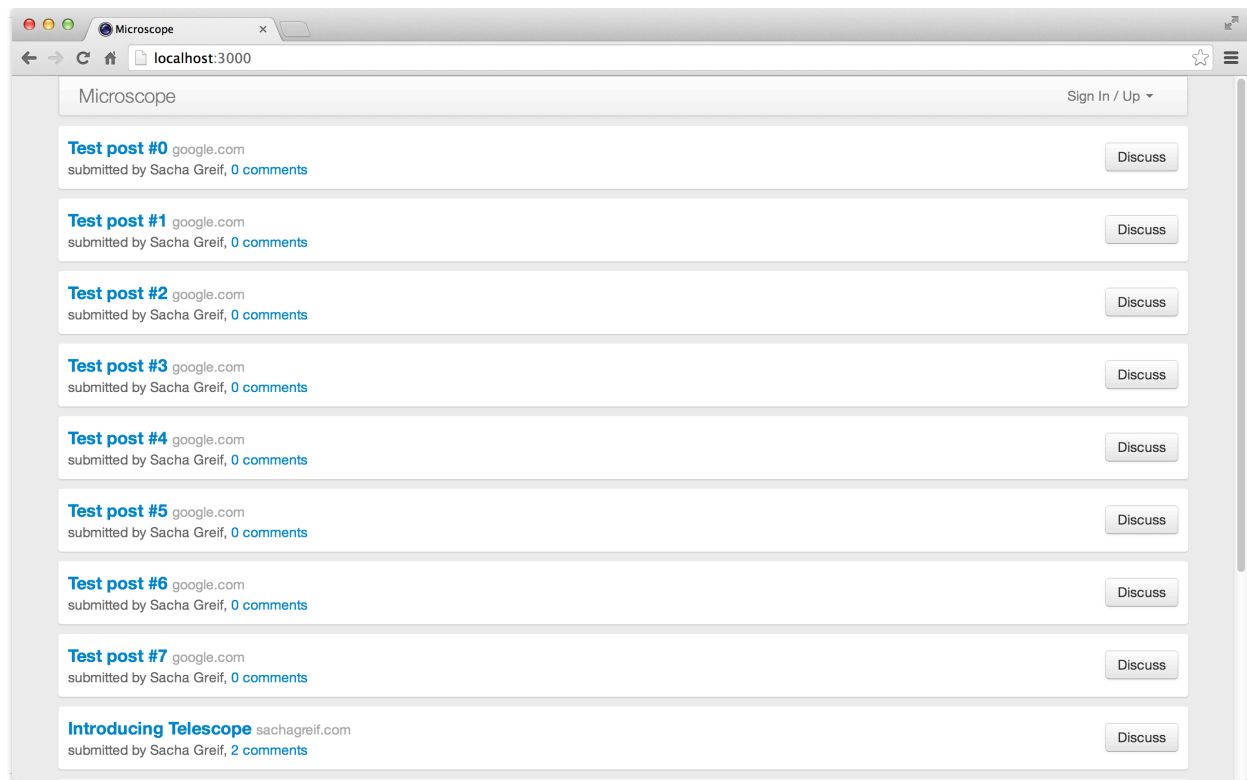
  Posts.insert({
    title: 'The Meteor Book',
    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themetorbook.com',
    submitted: new Date(now - 12 * 3600 * 1000),
    commentsCount: 0
  });

  for (var i = 0; i < 10; i++) {
    Posts.insert({
      title: 'Test post #' + i,
      author: sacha.profile.name,
      userId: sacha._id,
      url: 'http://google.com/?q=test-' + i,
      submitted: new Date(now - i * 3600 * 1000),
      commentsCount: 0
    });
  }
}

```

server/fixtures.js

After running `meteor reset` and starting your app again, you should now get something like this:



Displaying dummy data.

Commit 12-1

Added enough posts that pagination is necessary.

[View on GitHub](#)[Launch Instance](#)

Infinite Pagination

We'll be implementing an "infinite" style pagination. What we mean by that is that we'll first show, say, 10 posts on the screen, with a "load more" link pinned at the bottom. Clicking this link will add 10 more posts to the lists, and so on *ad infinitum*. This means we can control our entire pagination system with a single parameter representing the number of posts to display onscreen.

Now we'll need a way to tell the server about this parameter so that it knows how many posts to send up to the client. It so happens that we're already subscribing to the `posts` publication in the router, so we'll take advantage of this and let the router handle our pagination as well.

The easiest way to set this up is simply to make the posts limit parameter part of the path, giving us

URLs of the form `http://localhost:3000/25`. An added bonus of using the URL over other methods is that if you are currently displaying 25 posts and happen to reload the browser window by mistake, you'll still be seeing 25 posts once the page loads again.

In order to do this properly, we'll need to change the way we subscribe to posts. Just like we previously did in the *Comments* chapter, we'll need to move our subscription code from the *router* level to the *route* level.

This might all be a lot to take in at once, but it will become clearer with the code.

First, we'll stop subscribing to the `posts` publication in the `Router.configure()` block. Just delete `Meteor.subscribe('posts')`, leaving only the `notifications` subscription:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('notifications')]
  }
});
```

lib/router.js

We'll then add a `postsLimit` parameter to the route's path. Adding a `?` after the parameter name means that it's optional. So our route will not only match `http://localhost:3000/50`, but also plain old `http://localhost:3000`.

```
//...

Router.route('/:postsLimit?', {
  name: 'postsList',
});

//...
```

lib/router.js

It's important to note that a path of the form `/:parameter?` will match every possible path. Since each route will be parsed successively to see if it matches the current path, we need to make sure we organize our routes in order of decreasing specificity.

In other words, routes that target more specific routes like `/posts/:_id` should come first, and our `postsList` route should be moved **to the bottom** of the routes group since it pretty much matches everything,

It's now time to tackle the tough problem of subscribing and finding the right data. We need to deal with the case where the `postsLimit` parameter isn't present, so we'll assign it a default value. We'll use "5" to really give us enough room to play around with pagination.

```
//...

Router.route('/:postsLimit?', {
  name: 'postsList',
  waitOn: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
  }
});

//...
```

lib/router.js

You'll notice we're now passing a JavaScript object (`{sort: {submitted: -1}, limit: postsLimit}`) along with the name of our `posts` publication. This object will serve as the `options` parameter for the server side `Posts.find()` statement. Let's switch over to our server-side code to implement this:

```
Meteor.publish('posts', function(options) {
  check(options, {
    sort: Object,
    limit: Number
  });
  return Posts.find({}, options);
});

Meteor.publish('comments', function(postId) {
  check(postId, String);
  return Comments.find({postId: postId});
});

Meteor.publish('notifications', function() {
  return Notifications.find({userId: this.userId});
});
```

server/publications.js

Passing Parameters

Our publications code is in effect telling the server it can trust any JavaScript object sent by the client (in our case, `{limit: postsLimit}`) to serve as the `find()` statement's `options`. This makes it possible for users to submit any options they'd like via the browser console.

In our case, this is relatively harmless, since all a user could do is reorder posts differently, or change the limit (which is what we want to enable in the first place). Although a real-world app would probably need to limit the limit!

Thankfully, by using `check()` we know users can't sneak extra options in (such as `fields`, which in some cases might expose private data on documents).

Still, a more secure pattern could be passing the individual parameters themselves instead of the whole object, to make sure you stay in control of your data:

```
Meteor.publish('posts', function(sort, limit) {  
  return Posts.find({}, {sort: sort, limit: limit});  
});
```

Now that we're subscribing at the route level, it would also make sense to set the data context in the same place. We'll deviate a bit from our previous pattern and make the `data` function return a JavaScript object instead of simply returning a cursor. This lets us create a *named* data context, which we'll call `posts`.

What this means is simply that instead of being implicitly available as `this` inside the template, our data context will be available at `posts`. Apart from this small element, the code should feel familiar:

```
//...

Router.route('/:postsLimit?', {
  name: 'postsList',
  waitOn: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
  },
  data: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return {
      posts: Posts.find({}, {sort: {submitted: -1}, limit: limit})
    };
  }
});

//...
```

lib/router.js

And since we're setting the data context at the route level, we can now safely get rid of the `posts` template helper inside the `posts_list.js` file and just delete the contents of that file.

We named our data context `posts` (the same name as the helper), so we don't even need to touch our `postsList` template!

Let's recap. Here's what our new and improved `router.js` code should look like:

```
Router.configure({
  layoutTemplate: 'layout',
  loadingTemplate: 'loading',
  notFoundTemplate: 'notFound',
  waitOn: function() {
    return [Meteor.subscribe('notifications')]
  }
});

Router.route('/posts/:_id', {
  name: 'postPage',
  waitOn: function() {
    return Meteor.subscribe('comments', this.params._id);
  },
  data: function() { return Posts.findOne(this.params._id); }
```

```

});

Router.route('/posts/:_id/edit', {
  name: 'postEdit',
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/submit', {name: 'postSubmit'});

Router.route('/:postsLimit?', {
  name: 'postsList',
  waitOn: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return Meteor.subscribe('posts', {sort: {submitted: -1}, limit: limit});
  },
  data: function() {
    var limit = parseInt(this.params.postsLimit) || 5;
    return {
      posts: Posts.find({}, {sort: {submitted: -1}, limit: limit})
    };
  }
});

var requireLogin = function() {
  if (! Meteor.user()) {
    if (Meteor.loggingIn()) {
      this.render(this.loadingTemplate);
    } else {
      this.render('accessDenied');
    }
  } else {
    this.next();
  }
}

Router.onBeforeAction('dataNotFound', {only: 'postPage'});
Router.onBeforeAction(requireLogin, {only: 'postSubmit'});

```

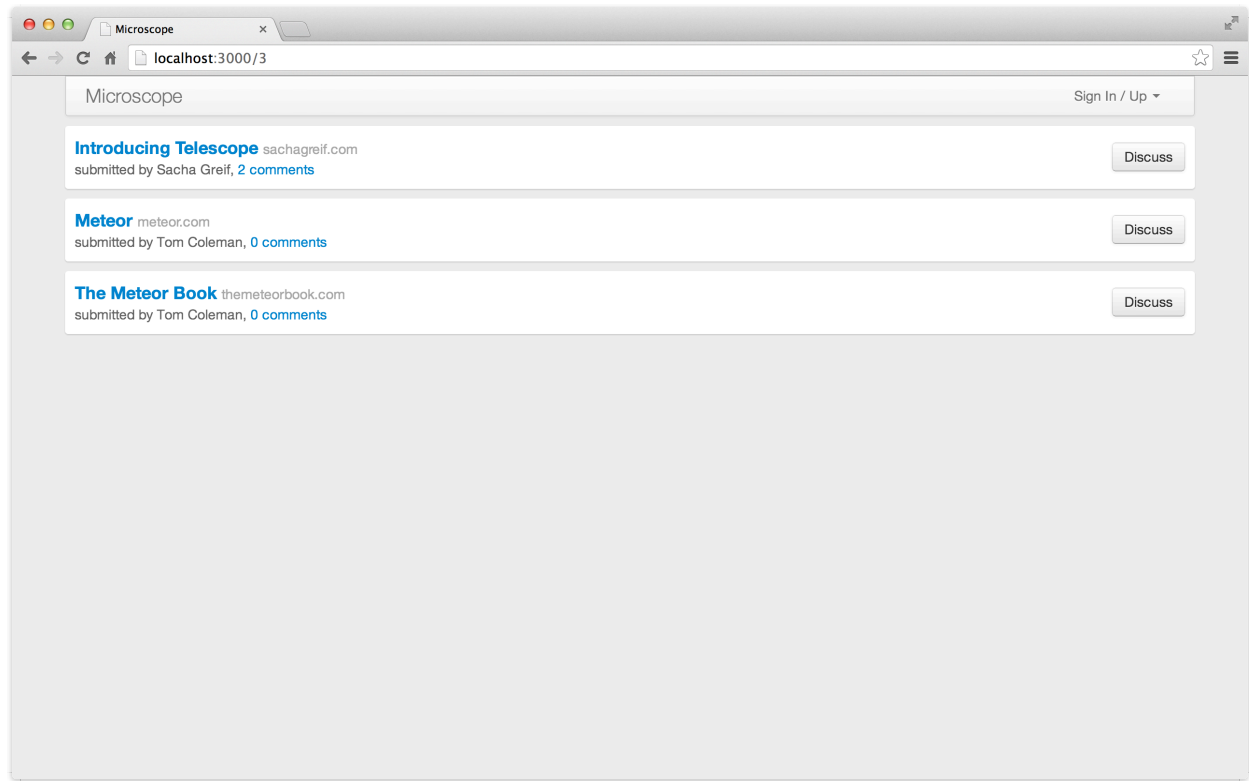
lib/router.js

Commit 12-2

Augmented the postsList route to take a limit.

[View on GitHub](#)
[Launch Instance](#)

Let's give our brand new pagination system a try. We now have the ability to display an arbitrary number of posts on the homepage simply by changing the URL parameter. For example, try accessing `http://localhost:3000/3`. You should now see something like this:



Controlling the number of posts on the homepage.

Why Not Pages?

Why are we using an “infinite pagination” approach instead of showing successive pages with 10 posts each, like what Google does for search results? This is actually due to the real-time paradigm embraced by Meteor.

Let’s imagine we are paginating our `Posts` collection using the Google results pagination pattern, and that we’re currently on page 2, which shows posts 10 to 20. What happens if another users deletes any of the previous 10 posts?

Since our app is real-time, our dataset would change. Post 10 would now become post 9, and drop out of our view, while post 11 would now be in range. The end result would be that the user would suddenly see their posts change for no apparent reason!

Even if we tolerated this UX quirk, traditional pagination is also hard to implement for technical reasons.

Let’s go back to our previous example. We’re publishing posts 10 to 20 from the `Posts` collection, but how would you find those posts on the client? You can’t pick posts 10 to 20, as there are only ten posts altogether in the client-side data set.

One solution would simply be to publish those 10 posts on the server, and then do a `Posts.find()` client-side to pick up *all* published posts.

This works if you only have a single subscription. But what if you start to have more than one post subscription, as we’ll do soon?

Let’s say one subscription asks for posts 10 to 20, and another one for posts 30 to 40. You now have 20 posts loaded client-side in total, with no way of knowing which ones belong to which subscription.

For all these reasons, traditional pagination just doesn’t make much sense when working with Meteor.

Creating a Route Controller

You might have noticed that we’re repeating the `var limit =`

`parseInt(this.params.postsLimit) || 5;` line twice. Plus, hard-coding the number “5” isn’t exactly ideal. This is not the end of the world, but since it’s always better to follow the DRY (Don’t Repeat Yourself) principle if you can, let’s see how we can refactor things a bit.

We’ll introduce a new aspect of Iron Router, *Route Controllers*. A route controller is simply a way to group routing features together in a nifty reusable package that any route can inherit from. Right now we’ll only use it for a single route, but you’ll see in the next chapter how this feature will come in handy.

```
//...

PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.postsLimit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  data: function() {
    return {posts: Posts.find({}, this.findOptions())};
  }
});

//...

Router.route('/:postsLimit?', {
  name: 'postsList'
});

//...
```

lib/router.js

Let’s go through this step by step. First, we’re creating our controller by extending `RouteController`. We then set the `template` property just like we did before, and then a new `increment` property.

We then define a new `postsLimit` function which will return the current limit, and a `findOptions` function which will return an options object. This might seem like an extra step, but we'll make use of it later on.

Next, we define our `waitOn` and `data` functions just like before, except they're now making use of our new `findOptions` function.

Because our controller is called the `PostsListController` and our route is named `postsList`, Iron Router will automatically use the controller. So we just need to remove the `waitOn` and `data` from our route definition (as the controller is now handling them). If we needed to use a controller with a different name, we could have used to `controller` option (we'll see an example of this in the next chapter).

Commit 12-3

Refactored postsLists route into a RouteController.

[View on GitHub](#)[Launch Instance](#)

Adding A Load More Link

We have a working pagination, and our code is looking good. There's just one problem: there's no way to actually *use* that pagination except by changing the URL manually. This definitely doesn't make for great user experience, so let's get to work on fixing this.

What we want to do is simple enough. We'll add a "load more" button at the bottom of our posts list, which will increment the number of posts currently displayed by 5 every time it's clicked. So if I'm currently on the URL `http://localhost:3000/5`, clicking "load more" should bring me to `http://localhost:3000/10`. If you've made it this far in the book, we trust you can handle a little arithmetic!

As before, we'll add our pagination logic in our route. Remember when we explicitly named our data context rather than just use an anonymous cursor? Well, there's no rule that says the `data`

function can only pass cursors, so we'll use the same technique to generate the URL of our “load more” button.

```
//...

PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.postsLimit()};
  },
  waitOn: function() {
    return Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.postsLimit();
    var nextPath = this.route.path({postsLimit: this.postsLimit() + this.increment});
    return {
      posts: this.posts(),
      nextPath: hasMore ? nextPath : null
    };
  }
});

//...
```

lib/router.js

Let's take a deeper look at this bit of router magic. Remember that the `postsList` route (which will inherit from the `PostsListController` controller we're currently working on) takes a `postsLimit` parameter.

So when we feed `{postsLimit: this.postsLimit() + this.increment}` to `this.route.path()`, we're telling the `postsList` route to build its own path using that JavaScript object as data context.

In other words, this is exactly the same thing as using the `{{pathFor 'postsList'}}` Spacebars helper, except we're replacing the implicit `this` by our own custom-made data context.

We're taking that path and adding it to the data context for our template, but *only* if there are more posts to display. The way we do that is a bit tricky.

We know that `this.limit()` returns the current number of posts we'd like to show, which can either be the value in the current URL, or our default value (5) if the URL doesn't contain any parameter.

On the other hand, `this.posts` refers to the current cursor, so `this.posts().count()` refers to the number of posts that are actually in the cursor.

So what we're saying here is that if we ask for `n` posts and we get `n` back, we'll keep showing the "load more" button. But if we ask for `n` and we get *less than* `n` back, then it means we've hit the limit and we should stop showing that button.

That being said, our system fails in one case: when the number of items in our database is *exactly* `n`. If that happens, the client will ask for `n` posts and get `n` posts back and keep showing the "load more" button, unaware that there are no more items left.

Sadly, there are no simple workarounds to this problem, so for now we'll have to settle with this less-than-perfect implementation.

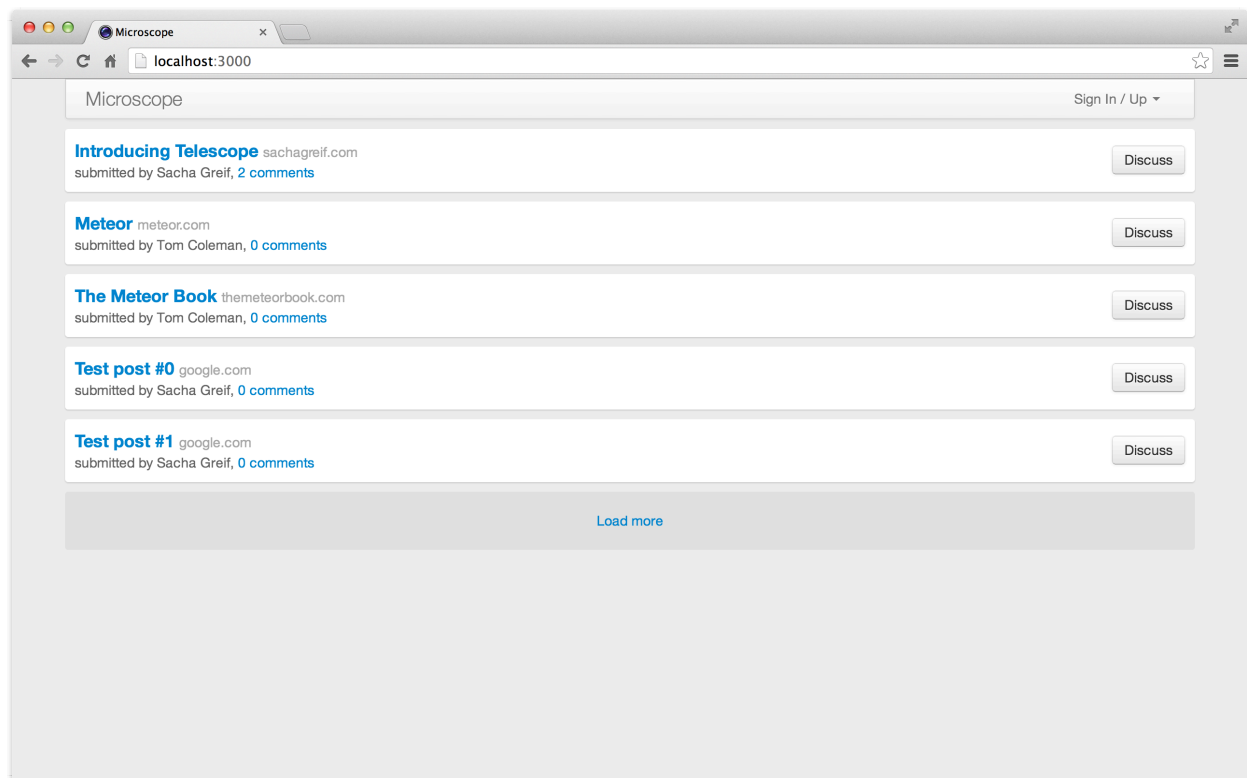
All that's left to do is to add the "load more" link at the bottom of our posts list, making sure to only show it if we actually have more posts to load:

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

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}">Load more</a>
    {{/if}}
  </div>
</template>
```

client/templates/posts/posts_list.html

Here's what your post list should now look like:



The "load more" button.

Commit 12-4

Added `nextPath()` to the controller and use it to step thr...

[View on GitHub](#)[Launch Instance](#)

A Better User Experience

Our pagination is now working properly, but it suffers from an annoying quirk: every time we click “load more” and the router asks for more posts, the Iron Router’s `waitOn` feature sends us to the `loading` template while we wait for the new data to come in. The result is that we’re sent back to the top of the page every time, and need to scroll all the way back down to resume our browsing.

So first, we’ll have to tell Iron Router not to `waitOn` the subscription after all. Instead, we’ll define our subscriptions in a `subscriptions` hook.

Note that we’re not *returning* this subscriptions in the hook. Returning it (which is how the `subscriptions` hook is usually employed) would trigger the global loading hook, and that’s exactly what we want to avoid in the first place. Instead we’re simply using the `subscriptions` hook as a convenient place to define our subscription, similar to using an `onBeforeAction` hook.

We’re also passing a `ready` variable referring to `this.postsSub.ready` as part of our data context. This will let us tell the template when the post subscription is done loading.


```
//...
```

```
PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: {submitted: -1}, limit: this.postsLimit()};
  },
  subscriptions: function() {
    this.postsSub = Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.postsLimit();
    var nextPath = this.route.path({postsLimit: this.postsLimit() + this.increment});
    return {
      posts: this.posts(),
      ready: this.postsSub.ready,
      nextPath: hasMore ? nextPath : null
    };
  }
});
```

```
//...
```

lib/router.js

We'll then check this `ready` variable in the template to show a spinner at the bottom of the post list while we are loading a new set of posts:

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

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}">Load more</a>
    {{else}}
      {{#unless ready}}
        {{> spinner}}
      {{/unless}}
    {{/if}}
  </div>
</template>
```

client/templates/posts/posts_list.html

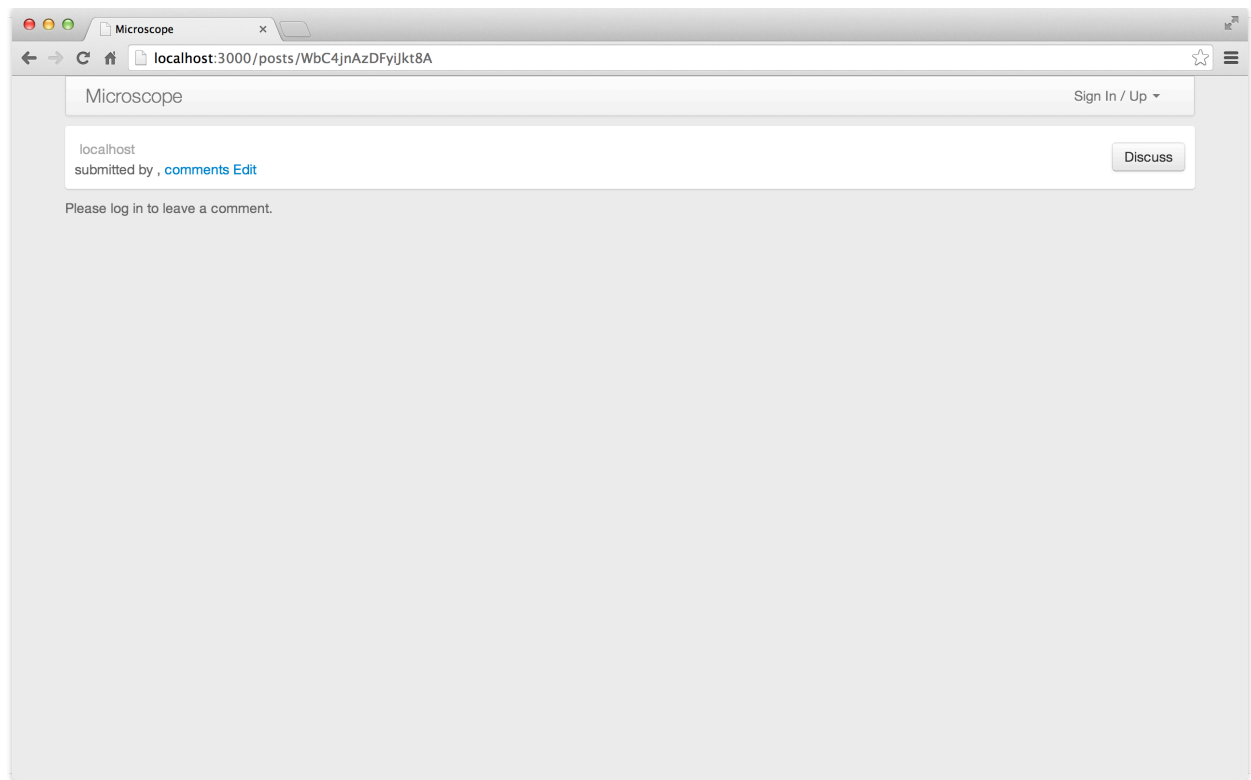
Commit 12-5

Add a spinner to make pagination nicer.

[View on GitHub](#)[Launch Instance](#)

Accessing Any Post

We're currently loading the five newest post by default, but what happens once someone browses to a post's individual page?



An empty template.

If you try it, you'll be faced with a "not found" error. This makes sense: we've told the router to subscribe to the `posts` publication when loading the `postsList` route, but we haven't told it what to do about the `postPage` route.

But so far, all we know how to do is subscribe to a list of the `n` latest posts. How do we ask the server for a single specific post? We'll let you in on a little secret here: you can have more than one publication for each collection!

So to get our missing posts back, we'll make a new, separate `singlePost` publication that only publishes one post, identified by `_id`.

```

Meteor.publish('posts', function(options) {
  return Posts.find({}, options);
});

Meteor.publish('singlePost', function(id) {
  check(id, String)
  return Posts.find(id);
});

//...

```

server/publications.js

Now, let's subscribe to the right posts client-side. We were already subscribing to the `comments` publication on the `postPage` route's `waitOn` function, so we can simply add the subscription to `singlePost` in there. And let's not forget to also add our subscription to the `postEdit` route, since it also needs the same data:

```

//...

Router.route('/posts/:_id', {
  name: 'postPage',
  waitOn: function() {
    return [
      Meteor.subscribe('singlePost', this.params._id),
      Meteor.subscribe('comments', this.params._id)
    ];
  },
  data: function() { return Posts.findOne(this.params._id); }
});

Router.route('/posts/:_id/edit', {
  name: 'postEdit',
  waitOn: function() {
    return Meteor.subscribe('singlePost', this.params._id);
  },
  data: function() { return Posts.findOne(this.params._id); }
});

//...

```

lib/router.js

Commit 12-6

Use a single post subscription to ensure that we can alwa...

[View on GitHub](#)[Launch Instance](#)

With pagination done, our app no longer suffers from scaling problems, and users are sure to contribute even more links than before. So wouldn't it be nice to have a way to somehow rank those links? Wouldn't you know it, this is precisely the topic of the next chapter!

Now that our site is getting more popular, finding the best links is quickly going to get tricky. What we need is some kind of ranking system to order our posts by.

We could build a complex ranking system with karma, time-based decay of points, and many other things (most of which are implemented in **Telescope**, Microscope's big brother). But for our app, we'll keep things simple and just rate posts by the number of votes they've received.

Let's start by giving users a way to vote on posts.

Data Model

We'll store a list of upvoters on each post so we can know whether to show the upvote button to users, as well as to prevent people from voting twice.

Data Privacy & Publications

We'll be publishing these lists of upvoters to all users, which will also automatically make that data publicly accessible via the browser console.

This is the kind of data privacy problem that can arise from the way collections work. For example, do we want people to be able to find out who has voted for their posts? In our case making that information publicly available won't really have any consequences, but it's important to at least acknowledge the issue.

We'll also denormalize the total number of upvoters on a post to make it easier to retrieve that figure. So we'll be adding two attributes to our posts, `upvoters` and `votes`. Let's start by adding them to our fixtures file:

```
// Fixture data
if (Posts.find().count() === 0) {
  var now = new Date().getTime();
```

```
// create two users
var tomId = Meteor.users.insert({
  profile: { name: 'Tom Coleman' }
});
var tom = Meteor.users.findOne(tomId);
var sachId = Meteor.users.insert({
  profile: { name: 'Sacha Greif' }
});
var sach = Meteor.users.findOne(sachId);

var telescopeId = Posts.insert({
  title: 'Introducing Telescope',
  userId: sach._id,
  author: sach.profile.name,
  url: 'http://sachagreif.com/introducing-telescope/',
  submitted: new Date(now - 7 * 3600 * 1000),
  commentsCount: 2,
  upvoters: [],
  votes: 0
});

Comments.insert({
  postId: telescopeId,
  userId: tom._id,
  author: tom.profile.name,
  submitted: new Date(now - 5 * 3600 * 1000),
  body: 'Interesting project Sacha, can I get involved?'
});

Comments.insert({
  postId: telescopeId,
  userId: sach._id,
  author: sach.profile.name,
  submitted: new Date(now - 3 * 3600 * 1000),
  body: 'You sure can Tom!'
});

Posts.insert({
  title: 'Meteor',
  userId: tom._id,
  author: tom.profile.name,
  url: 'http://meteor.com',
  submitted: new Date(now - 10 * 3600 * 1000),
  commentsCount: 0,
  upvoters: [],
  votes: 0
});

Posts.insert({
  title: 'The Meteor Book',
```

```

    userId: tom._id,
    author: tom.profile.name,
    url: 'http://themeteteorbook.com',
    submitted: new Date(now - 12 * 3600 * 1000),
    commentsCount: 0,
    upvoters: [],
    votes: 0
  });

  for (var i = 0; i < 10; i++) {
    Posts.insert({
      title: 'Test post #' + i,
      author: sacha.profile.name,
      userId: sacha._id,
      url: 'http://google.com/?q=test-' + i,
      submitted: new Date(now - i * 3600 * 1000 + 1),
      commentsCount: 0,
      upvoters: [],
      votes: 0
    });
  }
}

```

server/fixtures.js

As usual, stop your app, run `meteor reset`, restart your app, and create a new user account. Let's then also make sure these two properties are initialized when posts are created:


```
//...

var postWithSameLink = Posts.findOne({url: postAttributes.url});
if (postWithSameLink) {
  return {
    postExists: true,
    _id: postWithSameLink._id
  }
}

var user = Meteor.user();
var post = _.extend(postAttributes, {
  userId: user._id,
  author: user.username,
  submitted: new Date(),
  commentsCount: 0,
  upvoters: [],
  votes: 0
});

var postId = Posts.insert(post);

return {
  _id: postId
};

//...
```

collections/posts.js

Voting Templates

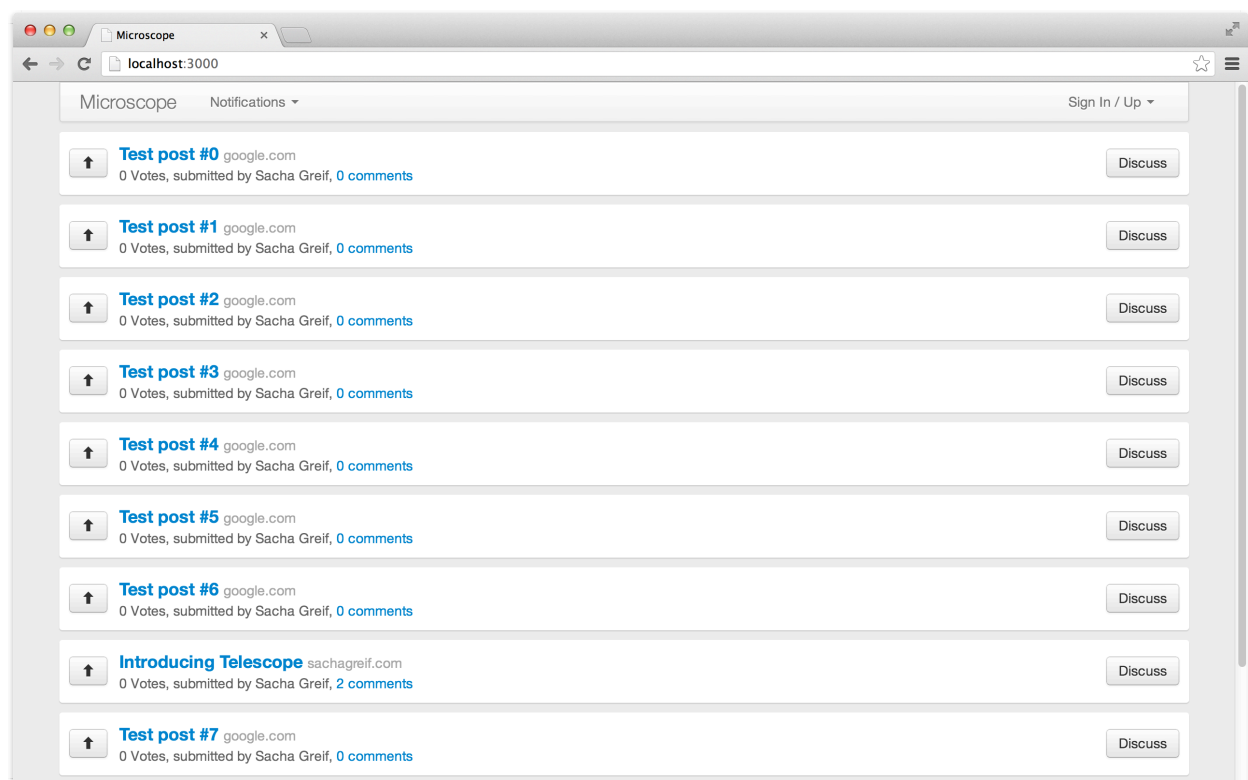
First off, we'll add an upvote button to our post partial and show the upvote count in the post's metadata:

```

<template name="postItem">
  <div class="post">
    <a href="#" class="upvote btn btn-default">↑</a>
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        {{votes}} Votes,
        submitted by {{author}},
        <a href="{{pathFor 'postPage'}}">{{commentsCount}} comments</a>
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor 'postPage'}}" class="discuss btn btn-default">Discuss</a>
  >
</div>
</template>

```

client/templates/posts/post_item.html



The upvote button

Next, we'll call a server upvote Method when the user clicks on the button:

```
//...

Template.postItem.events({
  'click .upvote': function(e) {
    e.preventDefault();
    Meteor.call('upvote', this._id);
  }
});
```

client/templates/posts/post_item.js

Finally, we'll go back to our `lib/collections/posts.js` file and add a Meteor server-side Method that will upvote posts:

```
//...

Meteor.methods({
  postInsert: function(postAttributes) {
    //...
  },

  upvote: function(postId) {
    check(this.userId, String);
    check(postId, String);

    var post = Posts.findOne(postId);
    if (!post)
      throw new Meteor.Error('invalid', 'Post not found');

    if (_.include(post.upvoters, this.userId))
      throw new Meteor.Error('invalid', 'Already upvoted this post');

    Posts.update(post._id, {
      $addToSet: {upvoters: this.userId},
      $inc: {votes: 1}
    });
  }
});

//...
```

lib/collections/posts.js

Commit 13-1

Added basic upvoting algorithm.

[View on GitHub](#)[Launch Instance](#)

This method is fairly straightforward. We do some defensive checks to ensure that the user is logged in and that the post really exists. Then we double check that the user hasn't already voted for the post, and if they haven't we increment the vote's total score and add the user to the set of upvoters.

This final step is interesting, as we've used a couple of special Mongo operators. There are many more to learn, but these two are extremely helpful: `$addToSet` adds an item to an array property as long as it doesn't already exist, and `$inc` simply increments an integer field.

User Interface Tweaks

If the user is not logged in, or has already upvoted a post, they won't be able to vote. To reflect this in our UI, we'll use a helper to conditionally add a `disabled` CSS class to the upvote button.

```
<template name="postItem">
  <div class="post">
    <a href="#" class="upvote btn btn-default {{upvotedClass}}">↑</a>
    <div class="post-content">
      //...
    </div>
  </template>
```

client/templates/posts/post_item.html

```

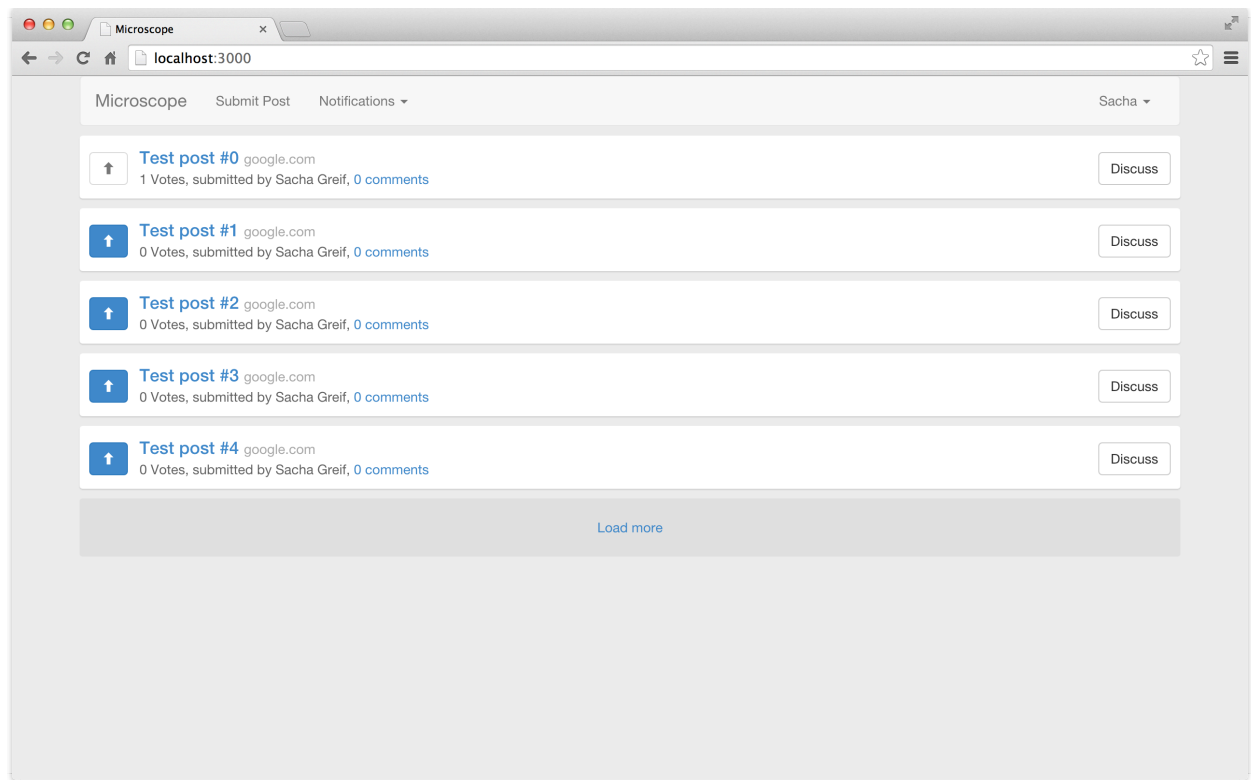
Template.postItem.helpers({
  ownPost: function() {
    //...
  },
  domain: function() {
    //...
  },
  upvotedClass: function() {
    var userId = Meteor.userId();
    if (userId && !_.include(this.upvoters, userId)) {
      return 'btn-primary upvotable';
    } else {
      return 'disabled';
    }
  }
});

Template.postItem.events({
  'click .upvotable': function(e) {
    e.preventDefault();
    Meteor.call('upvote', this._id);
  }
});

```

client/templates/posts/post_item.js

We're changing our class from `.upvote` to `.upvotable`, so don't forget to change the click event handler too.



Greying out upvote buttons.

Commit 13-2

Grey out upvote link when not logged in / already voted.

[View on GitHub](#)[Launch Instance](#)

Next, you may notice that posts with a single vote are labelled “1 votes”, so let’s take the time to pluralize those labels properly. Pluralization can be a complicated process, but for now we’ll do it in a fairly simplistic way. We’ll make a general Spacebars helper that we can use anywhere:

```
Template.registerHelper('pluralize', function(n, thing) {  
  // fairly stupid pluralizer  
  if (n === 1) {  
    return '1 ' + thing;  
  } else {  
    return n + ' ' + thing + 's';  
  }  
});
```

client/helpers/handlebars.js

The helpers we've created before have been tied to the template they apply to. But by using `Template.registerHelper`, we've created a *global* helper that can be used within any template:

```
<template name="postItem">

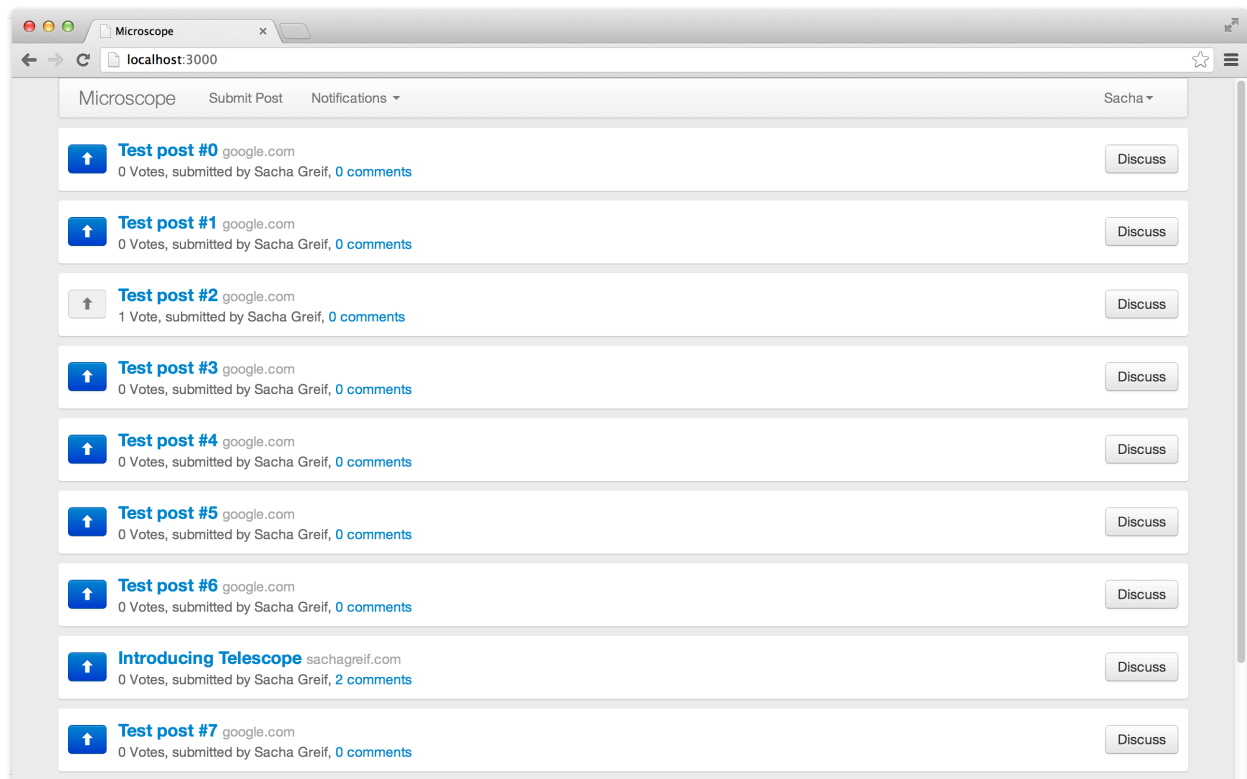
//...

<p>
  {{pluralize votes "Vote"}} ,
  submitted by {{author}},
  <a href="{{pathFor 'postPage'}}">{{pluralize commentsCount "comment"}}</a>
  {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
</p>

//...

</template>
```

client/templates/posts/post_item.html



Perfecting Proper Pluralization (now say that 10 times)

Commit 13-3

Added pluralize helper to format text better.

[View on GitHub](#)[Launch Instance](#)

We should now see “1 vote”.

Smarter Voting Algorithm

Our upvoting code is looking good, but we can still do better. In the upvote Method, we make two calls to Mongo: one to grab the post, then another to update it.

There are two issues with this. Firstly, it’s somewhat inefficient to go to the database twice. But more importantly, it introduces a race condition. We are following the following algorithm:

1. Grab the post from the database.
2. Check if the user has voted.
3. If not, do a vote by the user.

What if the same user voted for the post again in between steps 1 and 3? Our current code opens the door to the user being able to vote for the same post twice. Thankfully, Mongo allows us to be smarter and combine steps 1-3 into a single Mongo command:


```
//...

Meteor.methods({
  postInsert: function(postAttributes) {
    //...
  },

  upvote: function(postId) {
    check(this.userId, String);
    check(postId, String);

    var affected = Posts.update({
      _id: postId,
      upvoters: {$ne: this.userId}
    }, {
      $addToSet: {upvoters: this.userId},
      $inc: {votes: 1}
    });

    if (! affected)
      throw new Meteor.Error('invalid', "You weren't able to upvote that post");
  }
});

//...
```

collections/posts.js

Commit 13-4

Better upvoting algorithm.

[View on GitHub](#)

[Launch Instance](#)

What we are saying is “find all the posts with this `id` that this user hasn’t yet voted for, and update them in this way”. If the user *hasn’t* yet voted, it will of course find the post with that `id`. On the other hand if the user *has* voted, then the query will match no documents, and consequently nothing will happen.

Latency Compensation

Let's say you tried to cheat and send one of your posts to the top of the list by tweaking its number of votes:

```
> Posts.update(postId, {$set: {votes: 10000}});
```

Browser console

(Where `postId` is the id of one of your posts)

This brazen attempt at gaming the system would be caught by our `deny()` callback (in `collections/posts.js`, remember?) and immediately negated.

But if you look carefully, you might be able to see latency compensation in action. It may be quick, but the post will briefly jump to the top of the list before shooting back into position.

What's happened? In your local `Posts` collection, the `update` was applied without incident. This happens instantly, so the post shot to the top of the list. Meanwhile, on the server, the `update` was being denied. So some time later (measured in the milliseconds if you are running Meteor on your own machine), the server returned an error, telling the local collection to revert itself.

The end result: while waiting for the server to respond, the user interface can't help but trust the local collection. As soon as the server comes back and denies the modification, the user interfaces adapts to reflect that.

Ranking the Front Page Posts

Now that we have a score for each post based on the number of votes, let's display a list of the best posts. To do so, we'll see how to manage two separate subscriptions against the post collection, and make our `postsList` template a bit more general.

To start off, we'll want to have *two* subscriptions, one for each sort order. The trick here is that both subscriptions will subscribe to the *same* `posts` publication, only with different arguments!

We'll also create two new routes called `newPosts` and `bestPosts`, accessible at the URLs `/new` and `/best` respectively (along with `/new/5` and `/best/5` for our pagination, of course).

To do this, we'll *extend* our `PostsListController` into two distinct `NewPostsListController` and `BestPostsListController` controllers. This will let us re-use the exact same route options for both the `home` and `newPosts` routes, by giving us a single `NewPostsListController` to inherit from. And additionally, it's just a nice illustration of how flexible Iron Router can be.

So let's replace the `{submitted: -1}` sort property in `PostsListController` by `this.sort`, which will be provided by `NewPostsListController` and `BestPostsListController`:

```
//...

PostsListController = RouteController.extend({
  template: 'postsList',
  increment: 5,
  postsLimit: function() {
    return parseInt(this.params.postsLimit) || this.increment;
  },
  findOptions: function() {
    return {sort: this.sort, limit: this.postsLimit()};
  },
  subscriptions: function() {
    this.postsSub = Meteor.subscribe('posts', this.findOptions());
  },
  posts: function() {
    return Posts.find({}, this.findOptions());
  },
  data: function() {
    var hasMore = this.posts().count() === this.postsLimit();
    return {
      posts: this.posts(),
      ready: this.postsSub.ready,
      nextPath: hasMore ? this.nextPath() : null
    };
  }
});

NewPostsController = PostsListController.extend({
  sort: {submitted: -1, _id: -1},
  nextPath: function() {
    return Router.routes.newPosts.path({postsLimit: this.postsLimit() + this.in
crement})
  }
});
```

```

});

BestPostsController = PostsListController.extend({
  sort: {votes: -1, submitted: -1, _id: -1},
  nextPath: function() {
    return Router.routes.bestPosts.path({postsLimit: this.postsLimit() + this.increment});
  }
});

Router.route('/', {
  name: 'home',
  controller: NewPostsController
});

Router.route('/new/:postsLimit?', {name: 'newPosts'});

Router.route('/best/:postsLimit?', {name: 'bestPosts'});

```

lib/router.js

Note that now that we have more than one route, we're taking the `nextPath` logic out of `PostsListController` and into `NewPostsController` and `BestPostsController`, since the path will be different in either case.

Additionally, when we sort by `votes`, we have a subsequent sorts by submitted timestamp and then `_id` to ensure that the ordering is completely specified.

With our new controllers in place, we can now safely get rid of the previous `postsList` route. Just delete the following code:

```

Router.route('/:postsLimit?', {
  name: 'postsList'
})

```

lib/router.js

We'll also add links in the header:

```

<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'home'}}">Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav">
        <li>
          <a href="{{pathFor 'newPosts'}}">New</a>
        </li>
        <li>
          <a href="{{pathFor 'bestPosts'}}">Best</a>
        </li>
        {{#if currentUser}}
        <li>
          <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
        </li>
        <li class="dropdown">
          {{> notifications}}
        </li>
        {{/if}}
      </ul>
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>

```

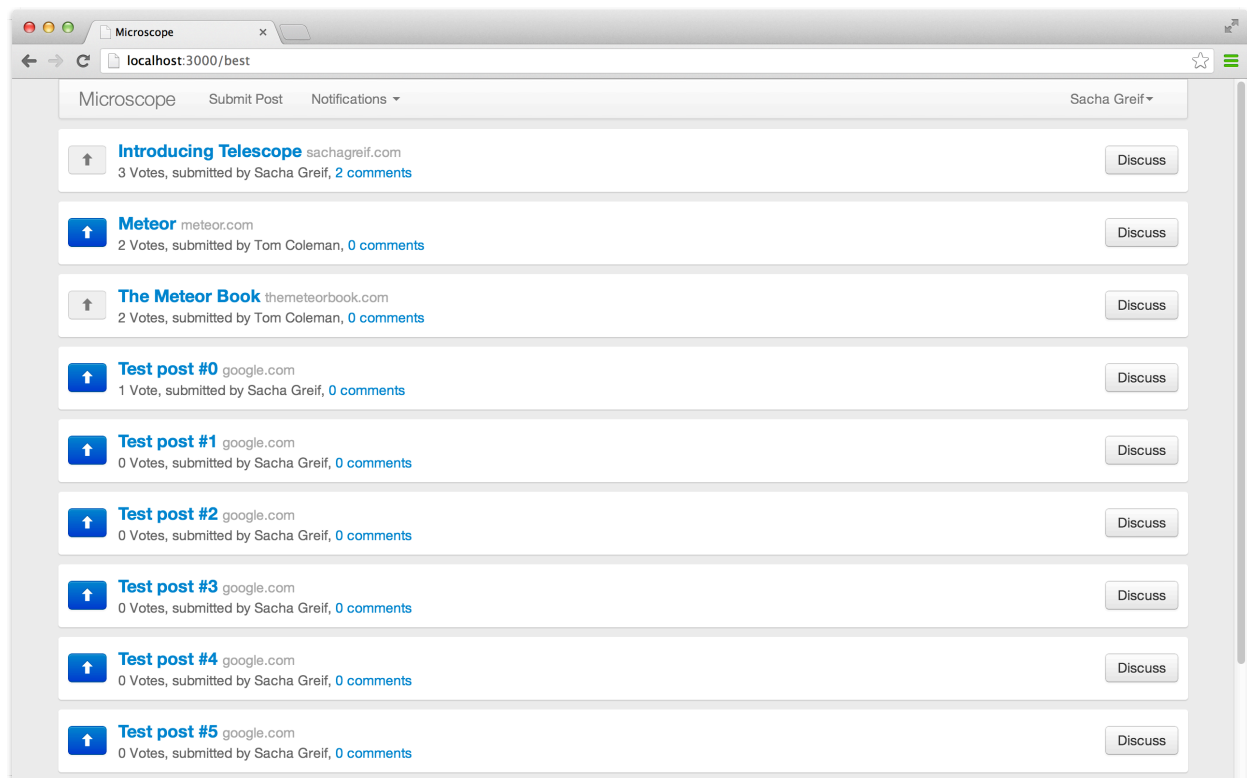
client/templates/includes/header.html

And finally, we also need to update our post deleting event handler:

```
'click .delete': function(e) {  
  e.preventDefault();  
  
  if (confirm("Delete this post?")) {  
    var currentPostId = this._id;  
    Posts.remove(currentPostId);  
    Router.go('home');  
  }  
}
```

client/templates/posts/post_edit.js

With all this done, we now gain a best posts list:



Ranking by points

Commit 13-5

Added routes for post lists, and pages to display them.

[View on GitHub](#)

[Launch Instance](#)

A Better Header

Now that we have two post list pages, it can be hard to know just which list you're currently viewing. So let's revisit our header to make it more obvious. We'll create a `header.js` manager and create a helper that uses the current path and one or more named routes to set an active class on our navigation items:

The reason why we want to support multiple named routes is that both our `home` and `newPosts` routes (which correspond to the `/` and `/new` URLs respectively) bring up the same template. Meaning that our `activeRouteClass` should be smart enough to make the `` tag active in both cases.

```
<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navigation">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{pathFor 'home'}}">Microscope</a>
    </div>
    <div class="collapse navbar-collapse" id="navigation">
      <ul class="nav navbar-nav">
        <li class="{{activeRouteClass 'home' 'newPosts'}}">
          <a href="{{pathFor 'newPosts'}}">New</a>
        </li>
        <li class="{{activeRouteClass 'bestPosts'}}">
          <a href="{{pathFor 'bestPosts'}}">Best</a>
        </li>
        {{#if currentUser}}
          <li class="{{activeRouteClass 'postSubmit'}}">
            <a href="{{pathFor 'postSubmit'}}">Submit Post</a>
          </li>
          <li class="dropdown">
            {{> notifications}}
          </li>
        {{/if}}
      </ul>
      <ul class="nav navbar-nav navbar-right">
        {{> loginButtons}}
      </ul>
    </div>
  </nav>
</template>
```

client/templates/includes/header.html


```

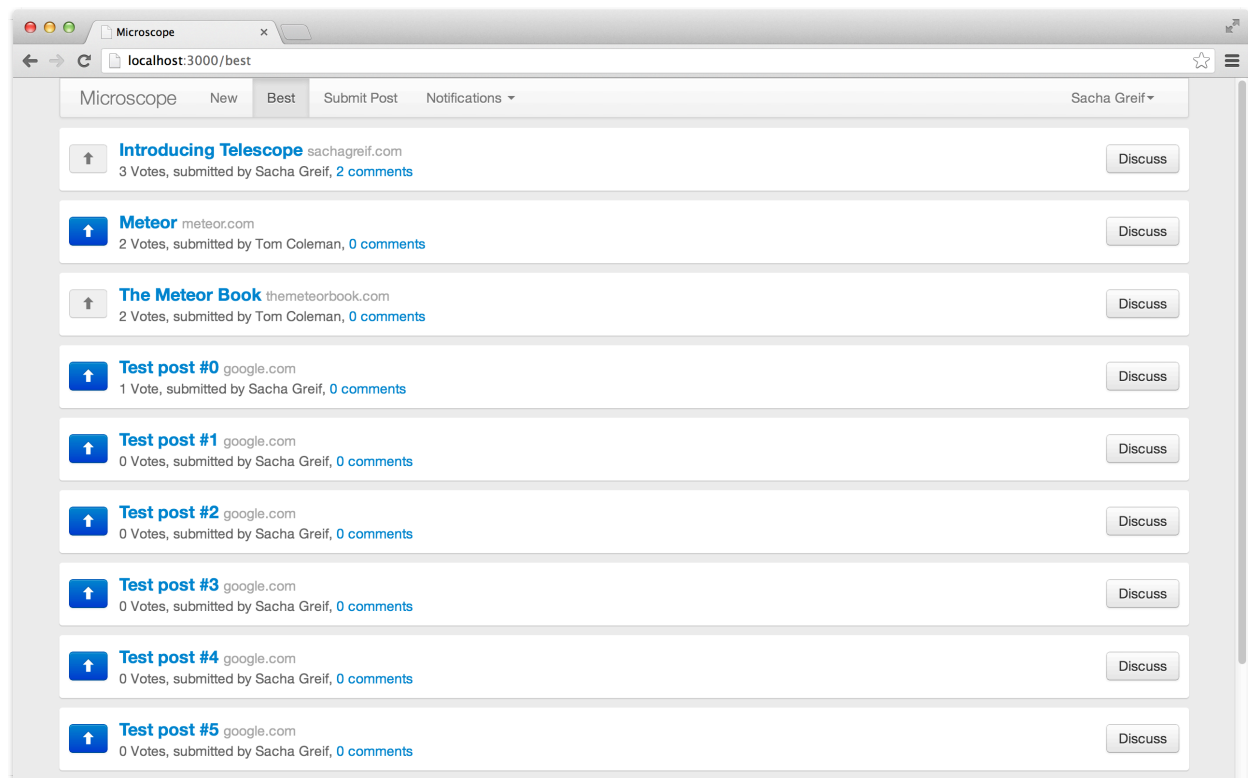
Template.header.helpers({
  activeRouteClass: function(/* route names */) {
    var args = Array.prototype.slice.call(arguments, 0);
    args.pop();

    var active = _.any(args, function(name) {
      return Router.current() && Router.current().route.getName() === name
    });

    return active && 'active';
  }
});

```

client/templates/includes/header.js



Showing the active page

Helper Arguments

We haven't used that specific pattern up to now, but just like any other Spacebars tags, template helper tags can take arguments.

And while you can of course pass specific named arguments to your function, you can also pass an unspecified number of anonymous parameters and retrieve them by calling the `arguments` object inside a function.

In this last case, you will probably want to convert the `arguments` object to a regular JavaScript array and then call `pop()` on it to get rid of the hash added at the end by Spacebars.

For each navigation item, the `activeRouteClass` helper takes a list of route names, and then uses Underscore's `any()` helper to see if any of the routes pass the test (i.e. their corresponding URL being equal to the current path).

If any of the routes do match up with the current path, `any()` will return `true`. Finally, we're taking advantage of the `boolean && string` JavaScript pattern where `false && myString` returns `false`, but `true && myString` returns `myString`.

Commit 13-6

Added active classes to the header.

[View on GitHub](#)[Launch Instance](#)

Now that users can vote on posts in real-time, you will see items jumping up and down the homepage as their ranking change. but wouldn't it be nice if there was a way to smooth out all this with a few well-timed animations?

By now you should have a good grasp of how publications and subscriptions interact. So let's get rid of the training wheels and examine a few more advanced scenarios.

Publishing a Collection Multiple Times

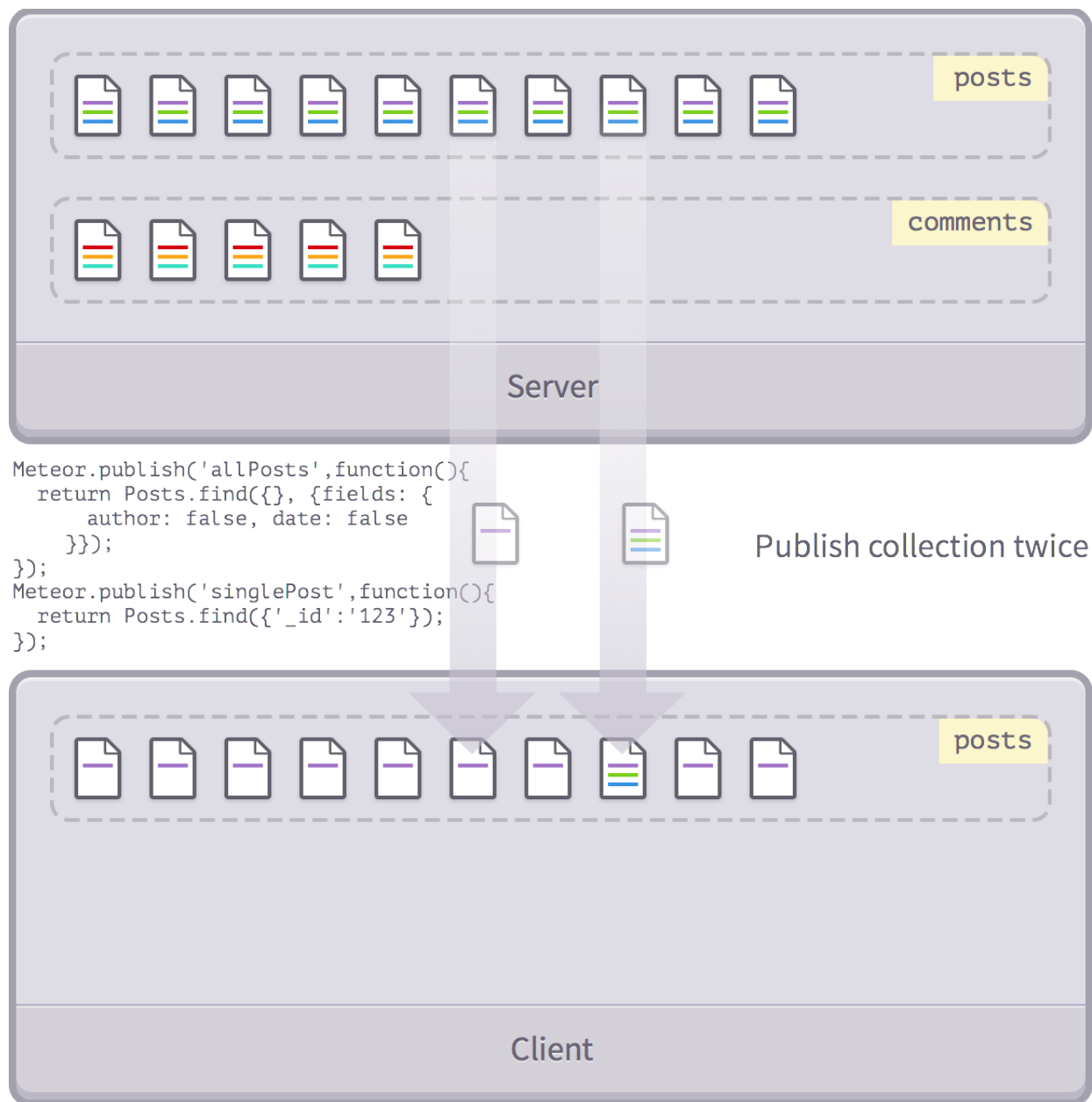
In **our first sidebar about publications**, we saw some of the more common publication and subscription patterns, and we learned how the `_publishCursor` function made them very easy to implement for our own sites.

First, let's recall what `_publishCursor` does for us exactly: it takes all the documents that match a given cursor, and pushes them down into the client-side collection *of the same name*. Notice that the name of the *publication* is in no way involved.

This means we can have *more than one publication* linking the client and server versions of any collection.

We've already encountered this pattern in our **pagination chapter**, when we published a paginated subset of all the posts in addition to the currently displayed post.

Another similar use case is to publish an *overview* of a large set of documents, as well as the full details of a single item:



Publishing a collection twice

```
Meteor.publish('allPosts', function() {
  return Posts.find({}, {fields: {title: true, author: true}});
});

Meteor.publish('postDetail', function(postId) {
  return Posts.find(postId);
});
```

Now when the client subscribes to those two publications, its **'posts'** collection gets populated from two sources: a list of titles and author's names from the first subscription, and the full details of a post from the second.

You may realize that the post published by `postDetail` is also being published by `allPosts` (although with only a subset of its properties). However, Meteor takes care of the overlap by merging the fields and ensuring there is no duplicate post.

This is great, because now when we render the list of post summaries, we are dealing with data objects that have just enough data for us to show what we need. However, when we render out the page for a single post, we have everything we need to show it. Of course, we need to take care on the client to not expect all fields to be available on all posts in this case – this is a common gotcha!

It should be noted that you're not limited to varying document properties. You could very well publish the same properties in both publications, but order items differently.

```
Meteor.publish('newPosts', function(limit) {
  return Posts.find({}, {sort: {submitted: -1}, limit: limit});
});

Meteor.publish('bestPosts', function(limit) {
  return Posts.find({}, {sort: {votes: -1, submitted: -1}, limit: limit});
});
```

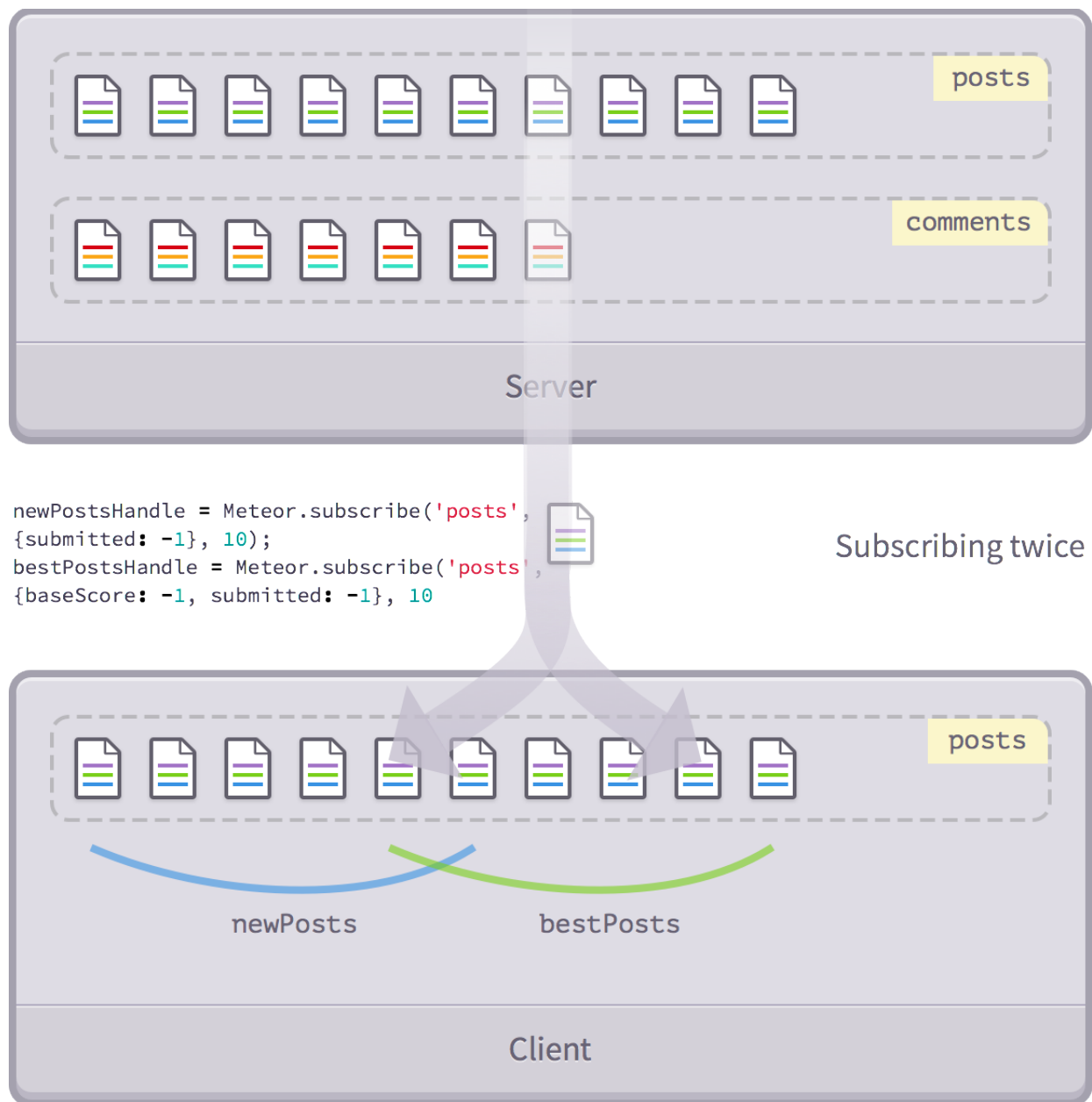
server/publications.js

Subscribing to a Publication Multiple Times

We've just seen how you can publish a single collection more than once. It turns out you can accomplish a very similar result with another pattern: creating a single publication, but *subscribing* to it multiple times.

In Microscope, we subscribe to the `posts` publication multiple times, but Iron Router sets up and tears down each subscription for us. Yet there's no reason why we couldn't subscribe multiple times *simultaneously*.

For example, let's say we wanted to load both the newest and best posts in memory at the same time:



Subscribing twice to one publication

We're setting up a single publication:

```
Meteor.publish('posts', function(options) {  
  return Posts.find({}, options);  
});
```

And we then subscribe to this publication multiple times. In fact this is more or less exactly what we're doing in Microscope:

```
Meteor.subscribe('posts', {submitted: -1, limit: 10});  
Meteor.subscribe('posts', {baseScore: -1, submitted: -1, limit: 10});
```

So what's happening here exactly? Each browser is opening up *two* different subscriptions, each connecting to the *same* publication on the server.

Each subscription provides different arguments to that publication, but fundamentally, each time a (different) set of documents is being plucked from the `posts` collection and sent down the wire to the client-side collection.

You can even subscribe to the same publication twice with *the same arguments*! It's hard to think of many scenarios where that would be useful, but the flexibility might be useful one day!

Multiple Collections in a Single Subscription

Unlike more traditional relational databses like MySQL which make use of *joins*, NoSQL databases like Mongo are all about *denormalizing* and *embedding*. Let's see how that works in the context of Meteor.

Let's look at a concrete example. We've added comments to our posts, and so far, we've been happy to only publish the comments on the single post that the user is looking at.

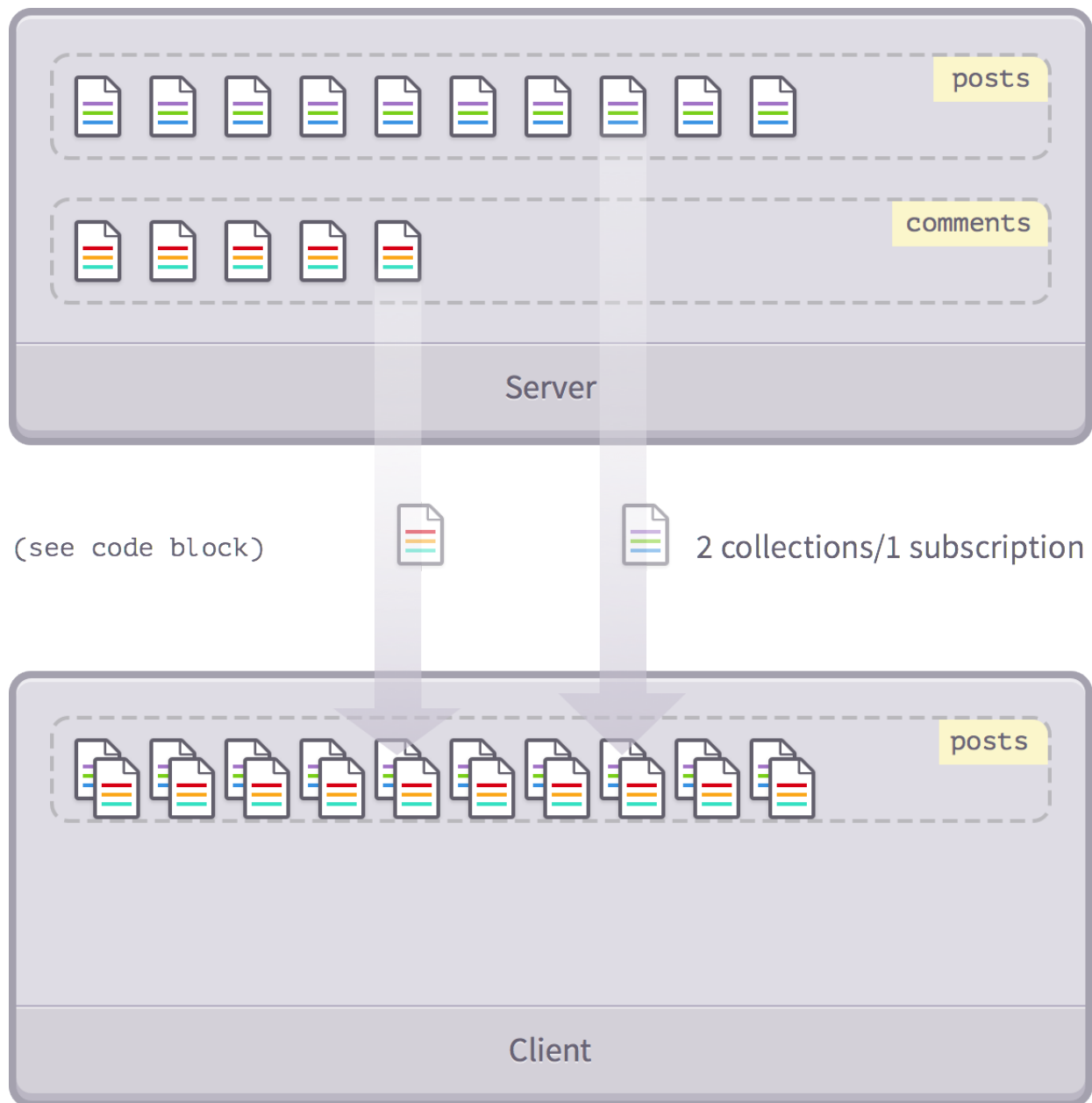
However, suppose we wanted to show comments on *all* the posts on the front page (keeping in mind that these posts will change as we paginate through them). This use case presents a good reason for embedding comments in posts, and in fact is what pushed us to denormalize comment *counts*.

Of course we could always just embed comments in posts, getting rid of the `Comments` collection altogether. But like we previously saw in the *Denormalization* chapter, by doing so we would be losing some of the extra benefits of working with separate collections.

But it turns out there's a trick involving subscriptions that makes it possible to embed our comments while preserving separate collections.

Let's suppose that along with our front-page list of posts, we want to subscribe to a list of the top 2 comments for each post.

It would be difficult to accomplish this with an independent comments publication, especially if the list of posts was limited in some way (say, the 10 most recent). We'd have to write a publication that looked something like this:



Two collections in one subscription

```
Meteor.publish('topComments', function(topPostIds) {  
  return Comments.find({postId: {$in: topPostIds}});  
});
```


This would be a problem from a performance standpoint, as the publication would need to get torn down and re-established each time the list of `topPostIds` changed.

There is a way around this though. We just use the fact that we can not only have more than one *publication* per *collection*, but we can also have more than one *collection* per *publication*:

```
Meteor.publish('topPosts', function(limit) {
  var sub = this, commentHandles = [], postHandle = null;

  // send over the top two comments attached to a single post
  function publishPostComments(postId) {
    var commentsCursor = Comments.find({postId: postId}, {limit: 2});
    commentHandles[postId] =
      Mongo.Collection._publishCursor(commentsCursor, sub, 'comments');
  }

  postHandle = Posts.find({}, {limit: limit}).observeChanges({
    added: function(id, post) {
      publishPostComments(id);
      sub.added('posts', id, post);
    },
    changed: function(id, fields) {
      sub.changed('posts', id, fields);
    },
    removed: function(id) {
      // stop observing changes on the post's comments
      commentHandles[id] && commentHandles[id].stop();
      // delete the post
      sub.removed('posts', id);
    }
  });

  sub.ready();

  // make sure we clean everything up (note `_publishCursor`
  // does this for us with the comment observers)
  sub.onStop(function() { postHandle.stop(); });
});
```

Note that we aren't returning anything in this publication, as we manually send messages to the `sub` ourselves (via `.added()` and friends). So we don't need to ask `_publishCursor` to do it for us by returning a cursor.

Now, every time we publish a post we also automatically publish the top two comments attached to it. And all with a single subscription call!

Although Meteor doesn't make this approach very straightforward yet, you can also look into the `publish-with-relations` package on Atmosphere, which aims to make this pattern easier to use.

Linking different collections

What else can our newfound knowledge of the flexibility of subscriptions give us? Well, if we don't use `_publishCursor`, we don't need to follow the constraint that the source collection on the server needs to have the same name as the target collection on the client.



One reason why we would want to do this is *Single Table Inheritance*.

Suppose that we wanted to reference various types of objects from our posts, each of which stored common fields but also differed slightly in content. For example, we could be building a Tumblr-like blogging engine where each post possesses the usual ID, timestamp, and title; but in addition can also feature an image, video, link, or just text.

We could store all these objects in a single `'resources'` collection, using a `type` attribute to indicate which sort of object they are. (`video`, `image`, `link`, etc.).

And although we'd have a single `Resources` collection on the server, we could transform that single collection into multiple `Videos`, `Images`, etc. collections on the client with the following bit of magic:

```
Meteor.publish('videos', function() {
  var sub = this;

  var videosCursor = Resources.find({type: 'video'});
  Mongo.Collection._publishCursor(videosCursor, sub, 'videos');

  // _publishCursor doesn't call this for us in case we do this more than once.
  sub.ready();
});
```

We are telling `_publishCursor` to publish our videos (just like returning) the cursor would do, but rather than publish to the `resources` collection on the client, instead we are publishing from `'resources'` to `'videos'`.

Another similar idea is to use publish to a client side collection where there's *no server side collection at all!* For instance, you might grab the data from a 3rd party service, and publish them into a client-side collection.

Thanks to the flexibility of the publish API, the possibilities are endless.

We now have real-time voting, scoring, and ranking. However, this leads to a jarring, erratic user experience as posts jump around on the homepage. We'll use animations to smooth this over.

Introducing `_uihooks`

`_uihooks` is a relatively new, and as yet undocumented feature of Blaze. As its name indicates, it gives access to hooks that can be triggered whenever elements are inserted, removed, or animated.

The full list of hooks is as follow:

- `insertElement` : called whenever a new element is inserted.
- `moveElement` : called when an element changes position.
- `removeElement` : called when an element is removed.

Once defined, these hooks will *replace* Meteor's default behavior. In other words, instead of inserting, moving, or removing elements, Meteor will now use whatever behavior we specify – and it'll be up to us to make sure this behavior actually works!

Meteor & the DOM

Before we can start the fun part (making things move around), we need to understand how Meteor interacts with the DOM (Document Object Model – the collection of HTML elements that make up a page's contents).

The crucial point to keep in mind is that elements in the DOM cannot really be “moved”; however, they can be deleted and created (note that this is a limitation of the DOM itself, not of Meteor). So to give the illusion of elements A and B switching place, Meteor will actually delete element B and insert a brand new copy (B') before element A.

This makes animation a little tricky, as we can't just animate B to move it to a new position,

because B will be gone as soon as Meteor re-renders the page (which happens instantly thanks to reactivity). Don't worry though, we'll find a way.

The Soviet Runner

But first, a story.

The year was 1980, at the height of the Cold War. The Olympics were being held in Moscow, and the Soviets were determined to win the 100 meters dash at any cost. So a group of brilliant Soviet scientists equipped one of their athletes with a teleporter, and as soon as the gunshot was heard the runner disappeared in a flash, and was instantly reinserted into the space-time continuum right at the finish line.

Thankfully, race officials noticed the infraction immediately, and the athlete had no choice but to teleport back to the starting blocks, before being allowed to participate in the race by running like everybody else.

My historical sources aren't that reliable, so you should take that story with a grain of salt. But do try and keep the "Soviet runner with a teleporter" analogy in mind as we go through this chapter.

Breaking It Down


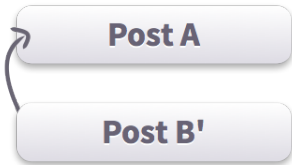
When Meteor receives an update and reactively modifies the DOM, our post will be instantly teleported to its final position, just like the soviet runner. But whether at the Olympics or in our app, we can't just have stuff teleporting around. So we'll teleport the element back to the "starting blocks" and make it "run" (in other words, animate it) up to the finish line.

So to switch posts A and B (positioned in positions p1 and p2, respectively), we would go through the following steps:

1. Delete B
2. Create B' before A in the DOM
3. Teleport B' to p2

4. Teleport A to p1
5. Animate A to p2
6. Animate B' to p1

The following diagram explains these steps in more detail:

Step	User Interface	DOM
Step 0 Start		<pre><div id="postA">...</div> <div id="postB">...</div></pre>
Step 1 Delete Post B		<pre><div id="postA">...</div></pre>
Step 2 Create the new Post B'		<pre><div id="postB">...</div> <div id="postA">...</div></pre>
Step 3 Move B' to p2		<pre><div id="postB">...</div> <div id="postA">...</div></pre>
Step 4 Move A to p1		<pre><div id="postB">...</div> <div id="postA">...</div></pre>
Step 5 Animate A to p2		<pre><div id="postB">...</div> <div id="postA">...</div></pre>
Step 6 Animate B' to p1		<pre><div id="postB">...</div> <div id="postA">...</div></pre>

Switching two posts

Again, in steps 3 and 4 we're not *animating* A and B' to their positions but "teleporting" them instantly. Since this is instantaneous, it will give the illusion that B was never deleted, and properly position both elements to be animated back to their new position.

By default, Meteor takes care of steps 1 & 2, and reimplementing them ourselves will be easy enough. And in steps 5 and 6 all we're doing is moving the elements to their proper spot. So the only part we really need to worry about is steps 3 and 4, i.e. sending the elements to the animation's starting point.

CSS Positioning

To animate the posts being reordered around the page, we'll have to venture into CSS territory. A quick review of CSS positioning might be in order.

Elements on a page use **static** positioning by default. Statically positioned elements just fit within the flow of the page, and their coordinates on the screen cannot be changed or animated.

Relative positioning on the other hand means that the element also fits in the flow of the page, but can be positioned *relative to its original position*.

Absolute positioning goes one step further and lets you give the element specific x/y coordinates relative to the **document** or **the first absolute or relative-positioned parent element**.

We'll use relative positioning to animate our posts. We've already taken care of the CSS for you, but if you needed to do it yourself all you would do is add this code to your stylesheet:

```
.post{
  position:relative;
}
.post.animate{
  transition:all 300ms 0ms ease-in;
}
```

client/stylesheets/style.css

Note that we only animate posts that have the CSS class `.animate`. This makes it possible to add and remove that class to control when animations should or shouldn't occur.

This makes steps 5 and 6 quite easy: all we need to do is reset `top` to `0px` (its default value) and our posts will slide back to their “normal” position.

So basically, our only challenge is figuring where to animate them *from* (steps 3 and 4) relative to their new position. In other words, how much to offset them. But that's not very hard either: the correct offset is simply a post's previous position minus its new one.

Implementing `_uihooks`

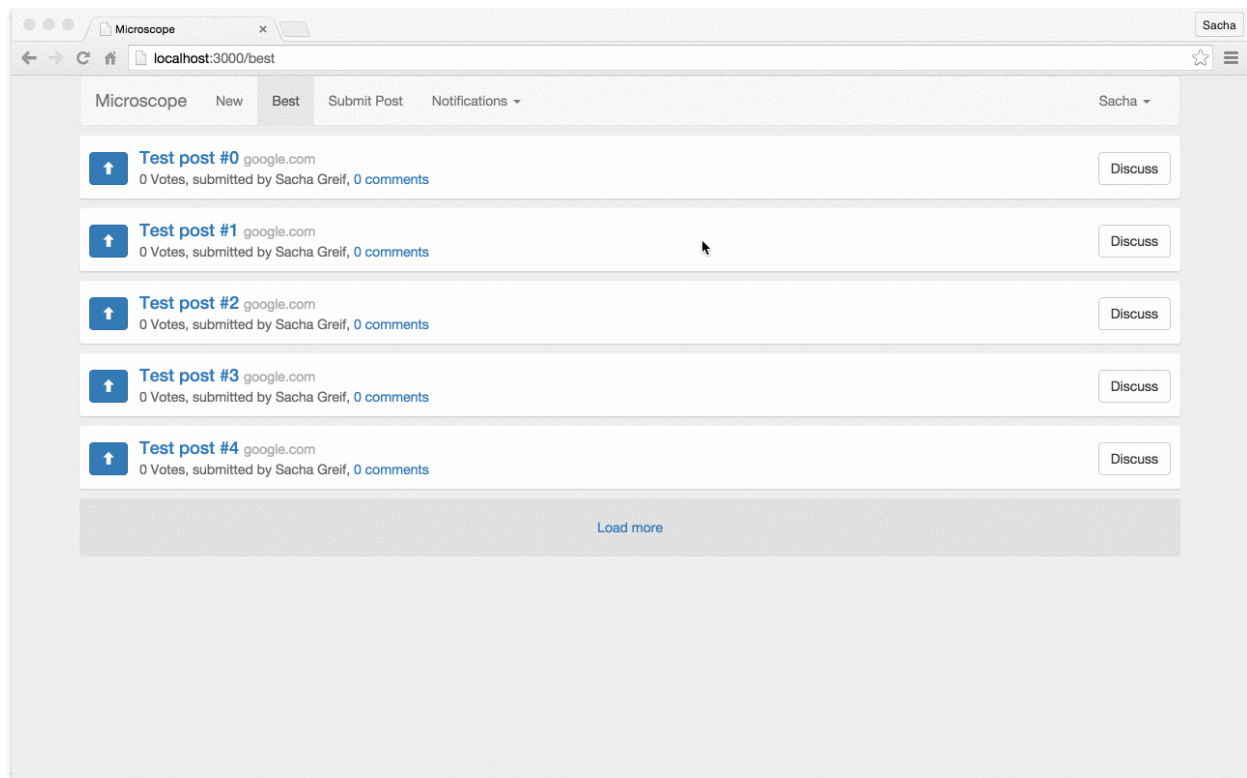
Now that we understand the various factors at play in animating a list of items, we're ready to start implementing the animation. We'll first need to wrap our list of posts in a new `.wrapper` container element:

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

    {{#if nextPath}}
      <a class="load-more" href="{{nextPath}}">Load more</a>
    {{else}}
      {{#unless ready}}
        {{> spinner}}
      {{/unless}}
    {{/if}}
  </div>
</template>
```

client/templates/posts/posts_list.html

Before we do anything else, let's review our posts' current behavior, *without* animations:



The non-animated post list.

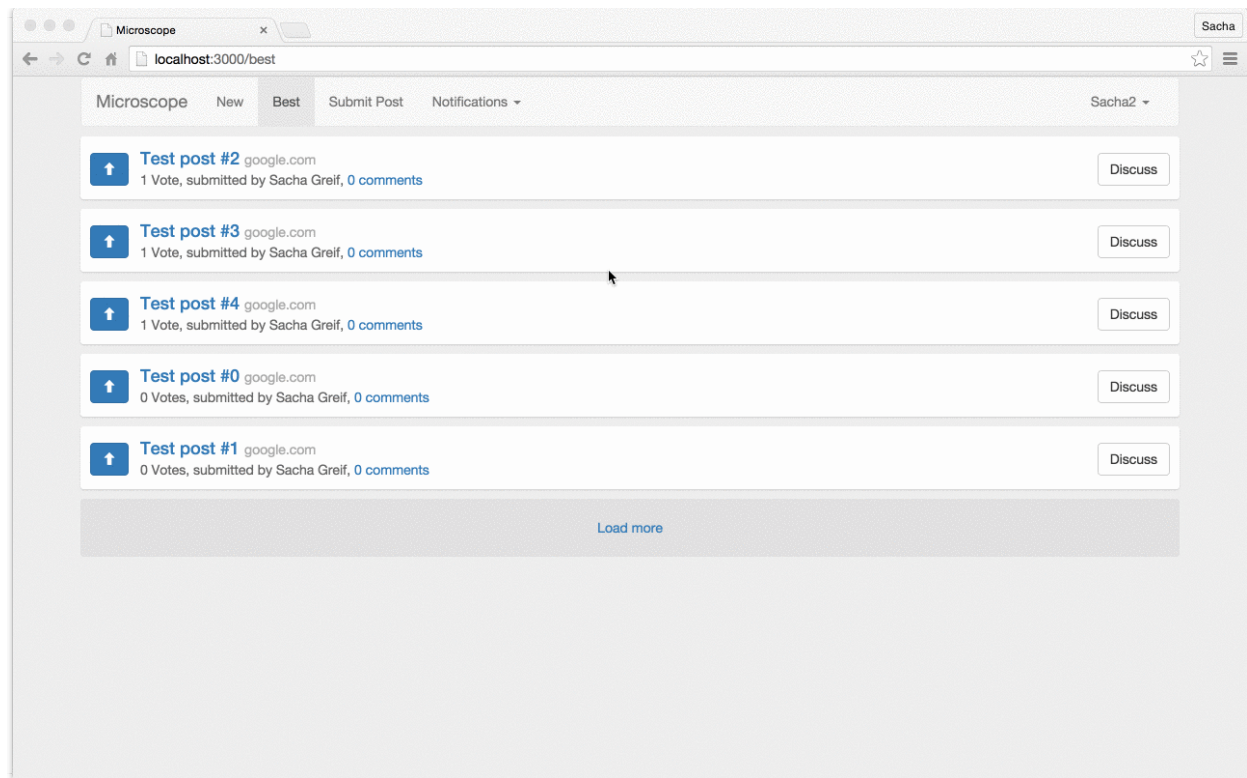
Let's bring in `_uihooks`. We'll select that `.wrapper` div inside the template's `onRendered` callback, and define a `moveElement` hook.

```
Template.postsList.onRendered(function () {  
  this.find('.wrapper')._uihooks = {  
    moveElement: function (node, next) {  
      // do nothing for now  
    }  
  }  
});
```

client/templates/posts/posts_list.js

The `moveElement` function we just defined will be called whenever an element's position changes *instead* of Blaze's default behavior. And since the function is empty, this means *nothing will happen*.

Go ahead and try it: open up the "Best" view and upvote a few posts: the ordering won't change until you force a rerender (either by reloading the page or switching routes).



An empty `moveElement` callback: nothing happens

We've ascertained that `_uihooks` works. Now let's make it animate!

Animating Posts Reordering

The `moveElement` hook takes two arguments: `node` and `next`.

- `node` is the element currently being moved to a new position in the DOM.
- `next` is the element right *after* the new position that `node` is being moved to.

Knowing this, we can work out the following animation process (feel free to refer back to the “Soviet Runner” example if you need to refresh your memory). When a new position change is detected, we'll:

1. Insert `node` before `next` (in other words, the default behavior that will happen if we don't specify any `moveElement` hook at all).
2. Move `node` back to its original position.
3. Nudge every element between `node` and `next` to make room for `node`.

4. Animate all elements back to their new default position.

We'll do all this through the magic of **jQuery**, by far the best DOM manipulation library out there. jQuery in general is out of the scope of this book, but let's quickly go over the handful of jQuery methods we'll use:

- `$()` : wrap any DOM element with the jQuery method to make it a jQuery object.
- `offset()` : retrieve the current position of an element relative to *the document*, and returns an object containing `top` and `left` properties.
- `outerHeight()` : get the “outer” height (including padding and optionally margin) of an element.
- `nextUntil(selector)` : get all elements after the target element up to (but not including) the element matched by `selector` .
- `insertBefore(selector)` : insert an element before the one matched by `selector` .
- `removeClass(class)` : remove the `class` CSS class, if present on the element.
- `css(propertyName, propertyValue)` : set the `propertyName` CSS property to `propertyValue` .
- `height()` : get an element's height.
- `addClass(class)` : add the `class` CSS class to an element.

```

Template.postsList.onRendered(function () {
  this.find('.wrapper')._uihooks = {
    moveElement: function (node, next) {
      var $node = $(node), $next = $(next);
      var oldTop = $node.offset().top;
      var height = $node.outerHeight(true);

      // find all the elements between next and node
      var $inBetween = $next.nextUntil(node);
      if ($inBetween.length === 0)
        $inBetween = $node.nextUntil(next);

      // now put node in place
      $node.insertBefore(next);

      // measure new top
      var newTop = $node.offset().top;

      // move node *back* to where it was before
      $node
        .removeClass('animate')
        .css('top', oldTop - newTop);

      // push every other element down (or up) to put them back
      $inBetween
        .removeClass('animate')
        .css('top', oldTop < newTop ? height : -1 * height);

      // force a redraw
      $node.offset();

      // reset everything to 0, animated
      $node.addClass('animate').css('top', 0);
      $inBetween.addClass('animate').css('top', 0);
    }
  }
});

```

client/templates/posts/posts_list.js

A few notes:

- We calculate `$node`'s height so we know by how much to offset the `$inBetween` elements. We use `outerHeight(true)` so that margin and padding are factored in the calculation.

- We don't know if `next` comes after or before `node` when going down the DOM. So we check both configurations when defining `$inBetween`.
- In order to switch between “teleporting” and “animating” elements, we're simply toggling the `animate` CSS class on and off (the actual animation being defined in the app's CSS code).
- Since we're using relative positioning, we can always reset any element's `top` property back to 0 to bring it back to where it's supposed to be.

Forcing The Redraw

You're probably wondering about that `$node.offset()` line. Why are we asking for `$node`'s position if we're not going to do anything with it?

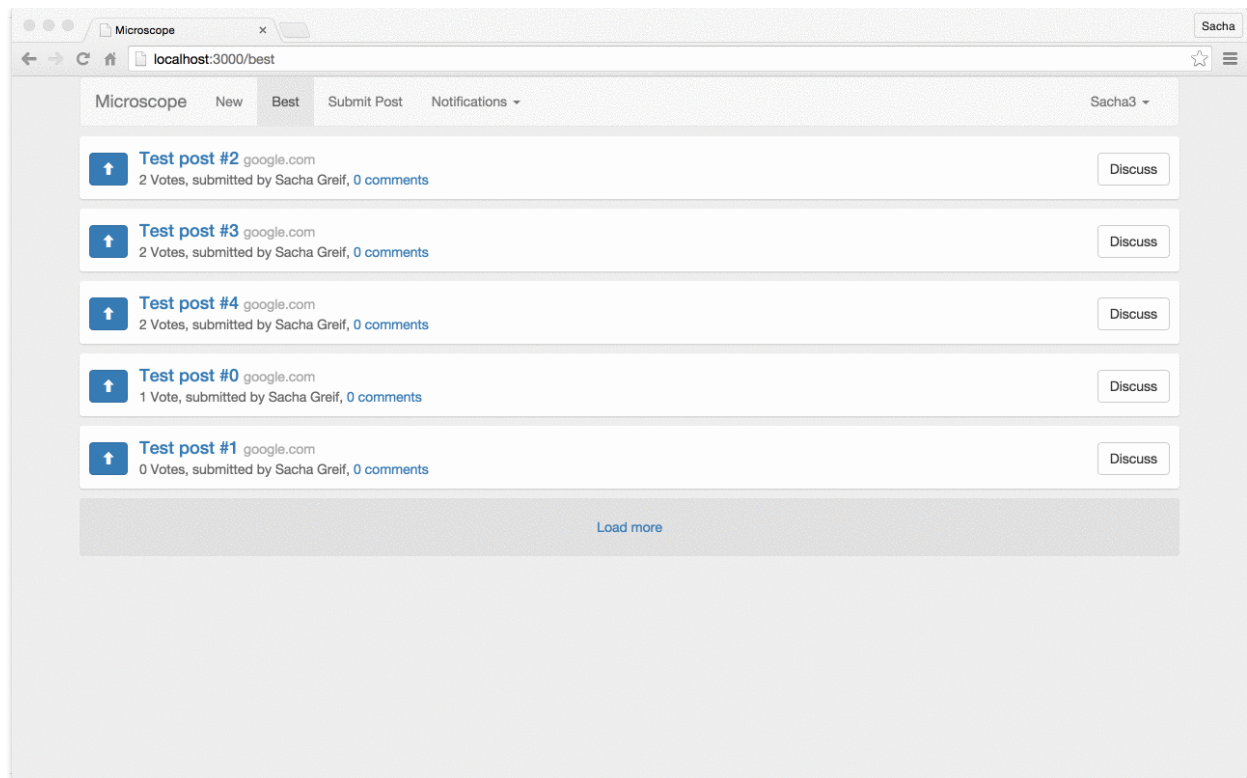
Think of it this way: if you told a perfectly logical android to run north for 5 kilometers, and then once that's done run back to its starting point, it would probably deduce that since it will end up in the same place it might as well save its energy and not run at all.

So in order to ensure that our android runs during the entire 10 kilometers, we'll ask it to measure its coordinates at the 5k mark before turning around.

The browser works in a similar way: if we just gave both the `css('top', oldTop - newTop)` and `css('top', 0)` instructions simultaneously, the new coordinates would simply replace the old ones and nothing would happen. If we want to actually see our animation, we need to force the browser to redraw the element after the first position change.

And a simple way to force that redraw is asking the browser to check the element's `offset` – it can't know what that is until it's drawn the element again.

Let's give it a spin. Go back to the “Best” view and start upvoting: you should now see our posts glide up and down with ballet-like grace!



Animated reordering

Commit 14-1

Added post reordering animation.

[View on GitHub](#)[Launch Instance](#)

Can't Fade Me

Now that we've taken care of the tricky reordering sequence, animating posts being inserted and removed will be a piece of cake!

First, we'll fade in new posts (note that for simplicity's sake, we're using JavaScript animations this time):


```

Template.postsList.onRendered(function () {
  this.find('.wrapper')._uihooks = {
    insertElement: function (node, next) {
      $(node)
        .hide()
        .insertBefore(next)
        .fadeIn();
    },
    moveElement: function (node, next) {
      //...
    }
  }
});

```

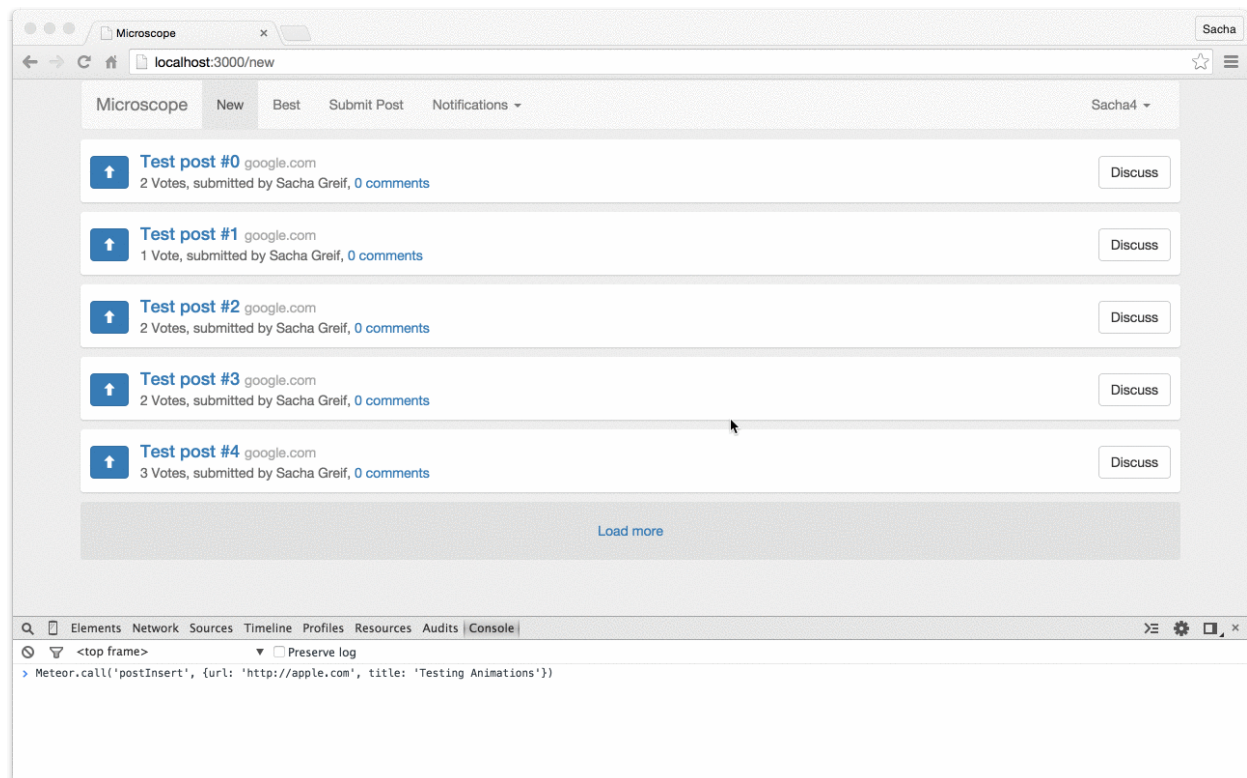
client/templates/posts/posts_list.js

To get a clear picture of the result, we can test out new animation by inserting a post via the console with:

```

Meteor.call('postInsert', {url: 'http://apple.com', title: 'Testing Animations'})

```



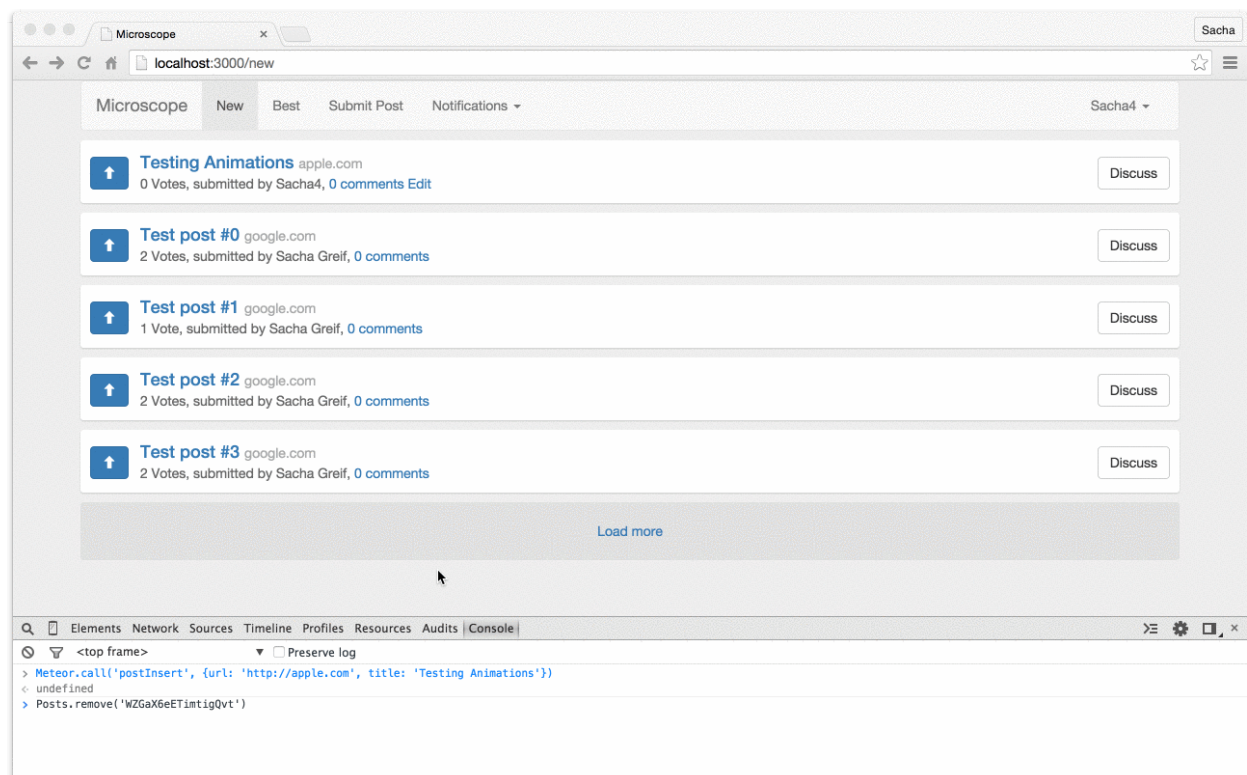
Fading in new posts

And then we'll fade out deleted posts:

```
Template.postsList.onRendered(function () {  
  this.find('.wrapper')._uihooks = {  
    insertElement: function (node, next) {  
      $(node)  
        .hide()  
        .insertBefore(next)  
        .fadeIn();  
    },  
    moveElement: function (node, next) {  
      //...  
    },  
    removeElement: function (node) {  
      $(node).fadeOut(function() {  
        $(this).remove();  
      });  
    }  
  }  
});
```

client/templates/posts/posts_list.js

Again, just delete a post via the console (using `Posts.remove('somePostId')`) to see the effect in action.



Commit 14-2

Fade items in when they are drawn.

[View on GitHub](#)[Launch Instance](#)

Page Transitions

So far we've animated elements *within* a page. But what if we wanted to add animated transitions *in-between* pages?

Page transitions are Iron Router's job. You click a link, and the content of the `{{> yield}}` helper in `layout.html` are automatically replaced.

It turns out that just like we overrode Blaze's default behavior for our post list, we can do the same thing for that `{{> yield}}` to add a fade transition in between routes!

If we want to fade pages in and out, we'll have to make sure they're displayed one on top of the other. We do that by using `position:absolute` on the `.page` container div that wraps every page template.

We don't want our pages to be absolutely positioned relative to the window though, since they would overlap the app's header. So we give `position:relative` to the containing `#main` div so that the `.page` div's `position:absolute` takes its origin from `#main`.

To save time, we've already added the necessary CSS code to `style.css`:

```
//...

#main{
  position: relative;
}
.page{
  position: absolute;
  top: 0px;
  width: 100%;
}

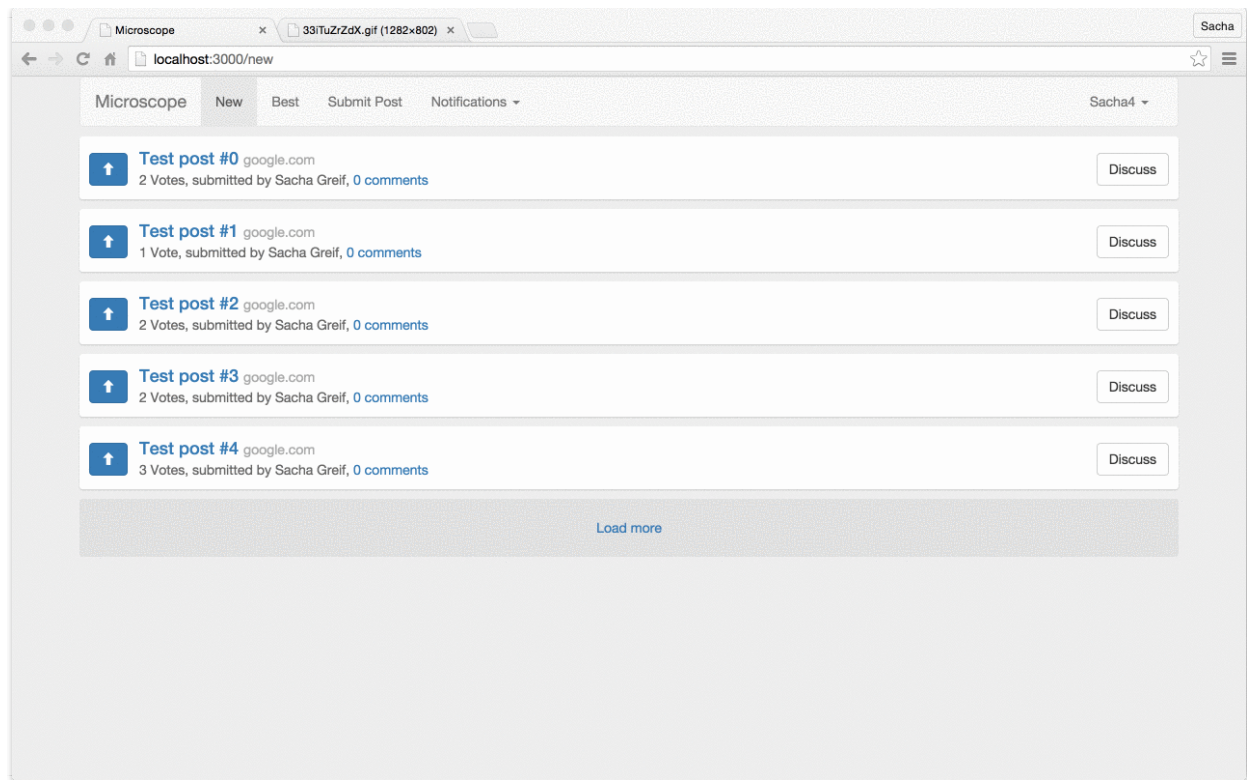
//...
```

client/stylesheets/style.css

It's time to add the page fade code. It should look familiar, since it's the exact same code we already used for the post insertion and removal:

```
Template.layout.onRendered(function() {
  this.find('#main')._uihooks = {
    insertElement: function(node, next) {
      $(node)
        .hide()
        .insertBefore(next)
        .fadeIn();
    },
    removeElement: function(node) {
      $(node).fadeOut(function() {
        $(this).remove();
      });
    }
  };
});
```

client/templates/application/layout.js



Transitioning in-between pages with a fade

Commit 14-3

Transition between pages by fading.

[View on GitHub](#)[Launch Instance](#)

We've just seen a few patterns for animating elements within your Meteor app. While this is not an exhaustive list by any means, hopefully it will provide a foundation on which to build more elaborate transitions.

Hopefully, the past chapters have given you a good overview of what's involved in building a Meteor app. So where do you go from now?

Extra Chapters

First of all, if you haven't already done so you can purchase the **Full** or **Premium** editions to unlock access to our extra chapters. These chapters will walk you through real-world scenarios such as building an API for your app, integrating with third-party services, and migrating your data.

Meteor Manual

In addition to the official **documentation**, the **Meteor Manual** digs deeper into specific topics such as Tracker and Blaze.

Evented Mind

If you'd like to take a deeper dive into the intricacies of Meteor, we also strongly recommend checking out Chris Mather's **Evented Mind**, a video learning platform with over 50 individual Meteor videos (and new videos being added every week).

MeteorHacks

One of the best ways to keep up with Meteor is to subscribe to Arunoda Susiripala's **MeteorHacks**' weekly newsletter. The MeteorHacks blog is also a great source for advanced Meteor tips.

Atmosphere

Atmosphere, Meteor's official package repository, is another great place to learn more: you can discover new packages and take a look at their code to see what patterns people are using.

Meteorpedia

Meteorpedia is a wiki for all things Meteor. And of course, it's built with Meteor!

BulletProof Meteor

Another initiative from MeteorHacks' Arunoda, **BulletProof Meteor** will walk you through a handful of performance-focused Meteor lessons, letting you test your knowledge through a Q&A format each step of the way.

The Meteor Podcast

Josh and Ben record **the Meteor Podcast** every week, and it's another great way to keep up with what's going on in the Meteor community.

Other Resources

Stephan Hochhaus has compiled a pretty exhaustive list of **Meteor resources**.

Manuel Schoebel's blog is another nice source of Meteor posts.

Getting Help

If you run into a stumbling block, the best place to ask is **Stack Overflow**. Make sure you tag your question with the `meteor` tag.

Community

Finally, the best way to stay up to date with Meteor is to be active in the community. We recommend signing up for the Meteor **mailing list**, joining the official **Meteor forums**, and staying up to date with Meteor news community **Crater.io**.

Client

When we talk about the Client, we are referring to code running in the user's *web browser*, whether that is a traditional browser like Firefox or Safari, or something as complex as a `UIWebView` in a native iPhone application.

Collection

A Meteor Collection is the data store that automatically synchronizes between client and server. Collections have a name (such as `posts`), and usually exist both on client and server. Although they behave differently, they have a common API based on Mongo's API.

Computation

A computation is a block of code that runs every time one of the reactive data sources that it depends on changes. If you have a reactive data source (for example, a `Session` variable) and would like to respond reactively to it, you'll need set up a computation for it.

Cursor

A cursor is the result of running a query on a Mongo collection. On the client side, a cursor isn't just an array of results, but a *reactive* object that can be observed as objects in the relevant collection are added, removed and updated.

DDP

DDP is Meteor's Distributed Data Protocol, the wire protocol used to synchronize collections and make Method calls. DDP is intended as a generic protocol, which takes the place of HTTP for realtime applications that are data heavy.

Tracker

Tracker is Meteor's reactivity system. Tracker is used behind the scenes to keep HTML

automatically sync with the underlying data model.

Document

Mongo is a document-based data-store, so the objects that come out of collections are called “documents”. They are plain JavaScript objects (although they can’t contain functions) with a single special property, the `_id`, which Meteor uses to track their properties over DDP.

Helpers

When a template needs to render something more complex than a document property it can call a helper, a function that is used to aid rendering.

Latency Compensation

Is a technique to allow simulation of Method calls on the client, to avoid laginess while waiting for the server to respond.

Meteor Development Group (MDG)

The actual company developing Meteor, by opposition to the framework itself.

Method

A Meteor Method is a remote procedure call from the client to the server, with some special logic to keep track of collection changes and allow Latency Compensation.

MiniMongo

The client-side collection is an in-memory data store offering a Mongo-like API. The library that supports this behaviour is called “MiniMongo”, to indicate it’s a smaller version of Mongo that runs completely in memory.

Package

A Meteor package can consist of JavaScript code to run on the server, JavaScript code to run on the

client, instructions on how to process resources (such as SASS to CSS), and resources to be processed.

A package is like a super-powered library. Meteor comes with an extensive set of core packages, and there's also **Atmosphere**, which is a collection of community supplied third party packages.

Publication

A publication is a named set of data that is customized for each user that subscribes to it. You set up a publication on the server.

Server

The Meteor server is a HTTP and DDP server run via Node.js. It consists of all the Meteor libraries as well as your server-side JavaScript code. When you start your Meteor server, it connects to a Mongo database (which it starts itself in development).

Session

The Session in Meteor refers to a client-side reactive data source that's used by your application to track the state that the user's in.

Subscription

A subscription is a connection to a publication for a specific client. The subscription is code that runs in the browser that talks to a publication on the server and keeps the data in sync.

Template

A template is a method of generating HTML in JavaScript. By default, Meteor supports Spacebars, a logic-less templating system, although there are plans to support more in the future.

Template Data Context

When a template renders, it refers to a JavaScript object that provides specific data for this particular rendering. Usually such objects are plain-old-JavaScript-objects (POJOs), often

documents from a collection, although they can be more complicated and have functions available on them.

May 31, 2015 1.3

- Changed starting file structure for Meteor 1.3 in Getting Started chapter.
- Added paragraph about adding Session package in Getting Started chapter.
- Updated Deploying chapter.
- Added note about method security in Creating Posts chapter.
- Clarified note about allow/deny vs methods in Editing Posts chapter.
- Also added note about allow/deny vs methods in Allow & Deny sidebar.

October 21, 2015 1.10

- Updated Sacha & Tom's bios in Introduction chapter.
- Updated list of default packages in Getting Started chapter.
- Removed the `underscore` package in Getting Started chapter and change commit 2-2's commit message.
- Rewrote the "Note on Packages" note in Getting Started chapter.
- Added note about getting new hash in Using GitHub chapter.
- Added note about sidebar code snippets in Publications & Subscriptions chapter.
- Added note about adding `check` and `audit-argument-checks` packages in Creating Posts chapter.
- Fixed code highlighting in "Adding A Link To The Header" section of Creating Posts chapter.
- Added note about not trusting the client with `userId`, `author`, and `submitted` properties in Creating posts chapter.
- Replaced `UI.insert` and `UI.render` by `Template.insert` and `Template.render` in Creating a Meteor Package chapter.
- Replaced `rendered` by `onRendered` in Creating a Meteor Package chapter.

- Changed `percolatestudio:percolatestudio-migrations` to `percolate:migrations` in Migrations chapter.
- Fixed code snippets for commit 9-5-1 in Creating a Meteor Package chapter.
- Fixed code highlighting for commit 13-5 in Voting chapter.
- Changed `post` to `postInsert` in Voting chapter.
- Added `$in` to “Multiple Collections in a Single Subscription” code snippet in Advanced Publications chapter.
- Added link to Meteor forums in Going Further chapter.
- Changed `on_use` to `onUse` and `add_files` to `addFiles` in Intercom chapter.

April 15, 2015 1.9

- Updated Introduction & Getting Started sections for Windows.
- Changed `rendered` to `onRendered` and `created` to `onCreated`.
- Explained package names in Getting Started chapter.
- Added note about default Iron Router help page in Routing chapter.
- Fixed `audit-argument-checks` link in Creating Posts chapter.
- Fixed file paths in Comments chapter.
- Wrong `limit()` changed to `postsLimit()` in Pagination chapter.
- Changed `UI.registerHelper` to `Template.registerHelper` in Voting chapter.
- Added a note about `.animate` CSS class in Animations chapter.
- Fixed file paths in Animations chapter.

February 10, 2015 1.8

- Rewrote animation chapter to use `_uihooks`.
- Wrapped every page in a `.page` div.
- Used the official `twbs:bootstrap` Bootstrap package.
- Added `.page` CSS to `style.css`.

- Used `Template.registerHelper` instead of `UI.registerHelper`.
- Removed Deploying On Modulus section (now referencing their docs instead).
- Updated `db.users.find()` result in “Adding Users” chapter.
- Added a note about the Meteor shell in the “Collections” chapter.

December 5, 2014 1.7.2

- Adding paragraph about `subscriptions` in Pagination chapter.
- Various typo fixes.
- Various code fixes.

November 10, 2014 1.7.1

Various fixes.

- Fix code highlighting in Voting chapter.
- Change “router” to “route” in Pagination chapter.
- Removed mentions of `Router.map()` in Comments and Pagination chapters.
- Linking to Bootstrap site in Adding Users chapter.
- Added BulletProof Meteor to Going Further chapter.

October 28, 2014 1.7

Updating the book for Iron Router 1.0.

- Defining routes with new path-first syntax.
- Use `subscriptions` instead of `onBeforeAction` in posts controller.
- Use `this.next()` in `onBeforeAction` hook.
- Rename `XYZPostsListController` to `XYZPostsController`.

October 24, 2014 1.6.1

-
- Fixing a few typos.
 - Finishing switching `Meteor.Collection` to `Mongo.Collection`.
 - Updated introduction.
 - Added “Get A Load Of This” section in Routing chapter.

October 15, 2014 1.6

Updating the book for Meteor 1.0.

General Changes

- `collections` directory is now in `lib`.
- `views` directory is now named `templates`.
- Removed `$` from bash prompts.
- Now using Bootstrap 3.
- Being more consistent about using `//...` to denote skipped lines in code.

Getting Started

- Explained the advantages of Meteor packages over manually adding files.
- Explicitly adding `underscore` package.
- Updated “5 Types of Packages” section.
- Not creating `collections` directory anymore.
- Updated CSS code.

Templates

- Changed “partials” to “inclusions”.
- Not talking about “managers” anymore.

Collections

- Cut down Collections chapter intro.
- Changed `Meteor.Collection()` to `Mongo.Collection()` .
- Added “Storing Data” section.
- General edits and tweaks.

Routing

- Added “Post Not Found” section.
- General edits and improvements.

The Session

- Added reminder to revert code changes at the end of the chapter.

Adding Users

- Now using `ian:accounts-ui-bootstrap-3` package.

Reactivity

- Using `Trackers` instead of `Deps` .

Creating Posts

- Removed `message` field from posts.
- Added “Security Check” section.
- Added “Preventing Duplicates” section.
- Changed `post` to `postInsert` , updated `postInsert` method description.

Latency Compensation

- Updated code examples.
- Added more explanations.

Allow & Deny

- Remove “Using Deny as a Callback” section.

Errors

- Completely rewrote error handling code.
- Added “Seeking Validation” section.

Creating a Meteor Package

- Various edits and updates.

Comments

- Rename `comment` template to `commentItem`.
- Various edits.

Notifications

- Added “No Trespassers Allowed” note.

Advanced Reactivity

- Added section about `reactive-var` package.

Pagination

- Got rid of `iron-router-progress`.

Voting

- Various edits.

Advanced Publications

- Various edits.

Animations

This chapter is out of date. Update coming sometimes after 1.0.

Note: the following extra chapters are only included in the Full and Premium editions:

RSS Feeds & APIs

- Updated package syntax.
- Minor tweaks.

Using External APIs

- Minor edits.

Implementing Intercom

- Added `favorite_color` custom attribute.
- Various minor edits.

Migrations

- Minor edits.

October 3, 2014 1.5.1

- Fix quotes in comments chapter.
- Clarified Session chapter.
- Added link to the blog in chapter 8.
- Adding a note about reversing changes at the end of Session sidebar.
- Reworking section about the five types of packages.
- Changing “partial” to “inclusion” in Templates chapter.
- Added note about Animations chapter being out of date.

August 27, 2014 1.5

- Updated Pagination chapter.
- Fixed typos.
- Removed mentions of Meteorite throughout the book.
- Updated Creating A Package sidebar for Meteor 0.9.0.
- Now including changelog in book repo.
- Book version is now tracked in changelog, not in intro chapter.
- Added section about manual.meteor.com in Going Further chapter.

May 30, 2014 1.3.4

- Replaced Vocabulary chapter with **Going Further** chapter.
- Added new Vocabulary sidebar.

May 20, 2014 1.3.3

- Various typos and highlighting fixes.
- Small correction in **Errors** chapter.

May 5, 2014 1.3.2

Various typos fixes.

April 8, 2014 1.3.1

Finished 0.8.0 Update.

- **12 – Pagination:** Use `count()` instead of `fetch().length()`.
- **14 – Animations:** Rewrote the chapter to use a helper instead of the `rendered` callback.

March 31, 2014 1.3

Updated to support Meteor 0.8.0 and Blaze.

- **5 – Routing:** Routing changes to support IR 0.7.0:
 - `{{yield}}` becomes `{{> yield}}`
 - Explicitly add `loading` hook.
 - Use `onBeforeAction` rather than `before` .
- **6 – Adding Users:** Minor change for Blaze:
 - `{{loginButtons}}` becomes `{{> loginButtons}}`
- **7 – Creating Posts:**
 - HTML changes for stricter parser.
 - Update our `onBeforeAction` hook to use `pause()` rather than `this.stop()`
- **13 – Voting:** Small change to the `activeRouteClass` helper.

January 13, 2014 1.2

The first update of 2014 is a big one! First of all, you'll notice a beautiful, photo-based layout that makes each chapter stand out more and introduces a bit of variety in the book.

And on the content side, we've updated parts of the book and even written two whole new chapters:

New Chapters

- **[NEW!] 3.5 – Using GitHub:** New sidebar on how to use GitHub.
- **[NEW!] 17.5 – Migrations:** New sidebar about database migrations.

Updates

- **2.5 – Deploying:** Rewrote chapter from scratch to be more up to date.
- **4.5 – Publications and Subscriptions:** Merged in content from **Understanding Meteor Publications & Subscriptions**
- **15 – RSS Feeds & APIs:** Updated chapter to use Iron Router.
- **16 – Using External APIs:** Updated chapter to use Iron Router.

- **17 – Implementing Intercom:** Rewrote chapter to match the **Intercom package**.

December 1, 2013 1.1

Major Updates

- **5 – Routing:** Rewrote chapter from scratch to use **Iron Router**.
- **5.5 – The Session:** Added a section about Autorun.
- **10 – Comments:** Updated chapter to use IR.
- **12 – Pagination:** Rewrote chapter from scratch, now managing pagination with IR.
- **13 – Voting:** Updated the chapter to use IR, simplified the template structure.

Minor Updates

Minor updates include API changes between the old Router and Iron Router, file paths updates, and small rewordings.

- **6 – Adding Users**
- **7 – Creating Posts**
- **7.5 – Latency Compensation**
- **8 – Editing Posts**
- **9 – Errors**
- **11 – Notifications**
- **12 – Animations**

If you'd like to confirm what exactly has changed, we've created a **full diff of our Markdown source files [PDF]**.

October 4, 2013 1.02

- Various typo fixes

September 4, 2013 1.01

- Updated “Creating a Meteorite Package” chapter to Meteor 0.6.5
- Updated package syntax in Intercom and API extra chapters.

May 5, 2013 1.0

First version.