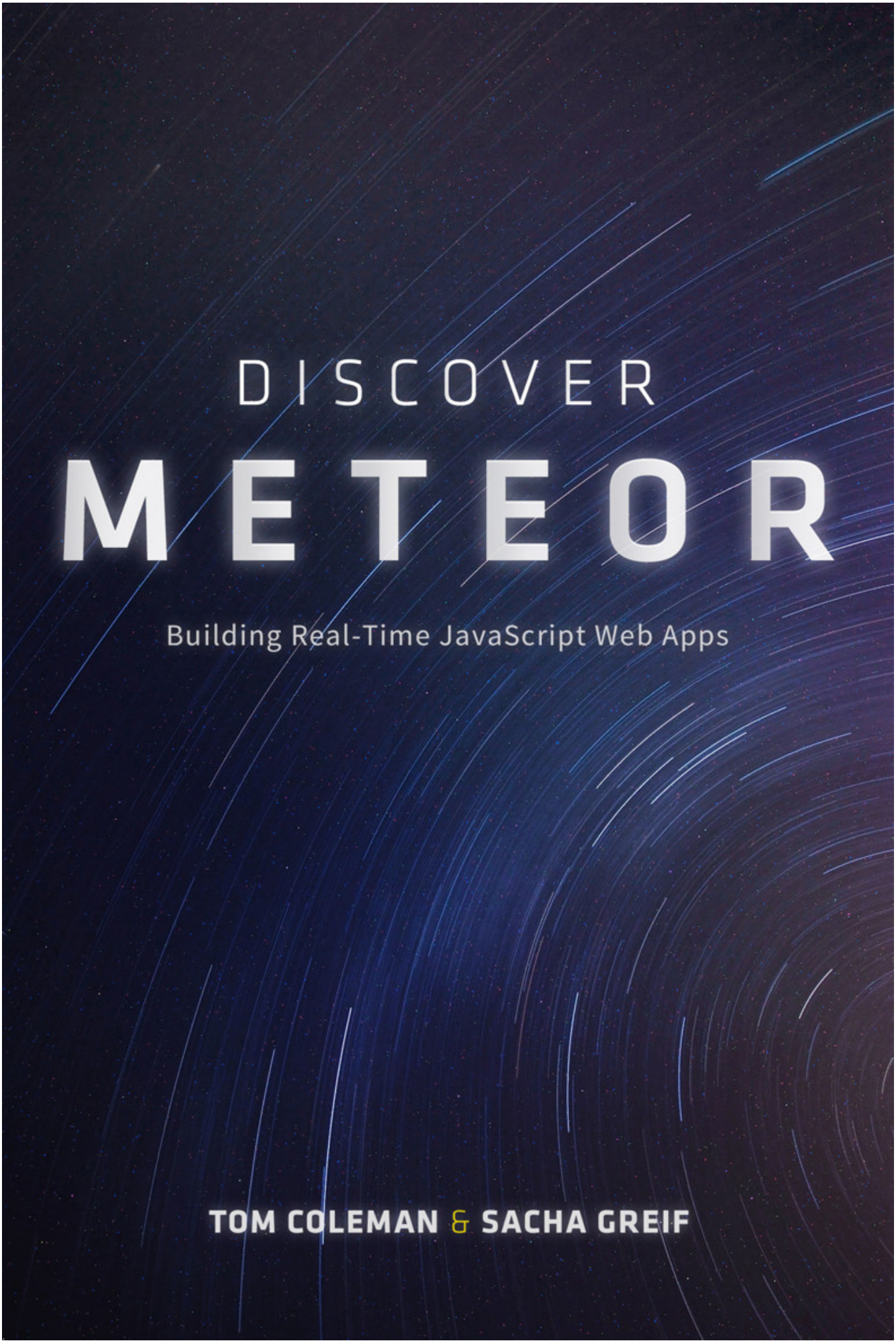

DISCOVER
METEOR
Building Real-Time JavaScript Web Apps
TOM COLEMAN & SACHA GREIF

DISCOVER **METEOR**

Building Real-Time JavaScript Web Apps

Version 1.7 (updated October 28, 2014)

Tom Coleman & Sacha Greif

www.discovermeteor.com

Table of Contents

Introduction 1

Getting Started 2

Deployment SIDEBAR 2.5

Templates 3

Using Git & GitHub SIDEBAR 3.5

Collections 4

Publications and Subscriptions SIDEBAR 4.5

Routing 5

The Session SIDEBAR 5.5

Adding Users 6

Reactivity SIDEBAR 6.5

Creating Posts 7

Latency Compensation SIDEBAR 7.5

Editing Posts 8

Introduction

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.

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 is one part of **Percolate Studio**, a web development shop with a focus on quality and user experience. He's one of the maintainers of the **Atmosphere** package repository, and is also one of the brains behind many other Meteor open-source projects (such as **Iron Router**).

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), and is also the founder of **Folyo**.

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.

Commits & Live Instances

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. Additionally, each commit also links to a live instance of the app at this particular commit, so you can compare it with your local copy. 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** is the best place 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 for Mac** app, which lets you clone and manage repos without using the command line.

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, we can install Meteor by opening a terminal window and typing:

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

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:

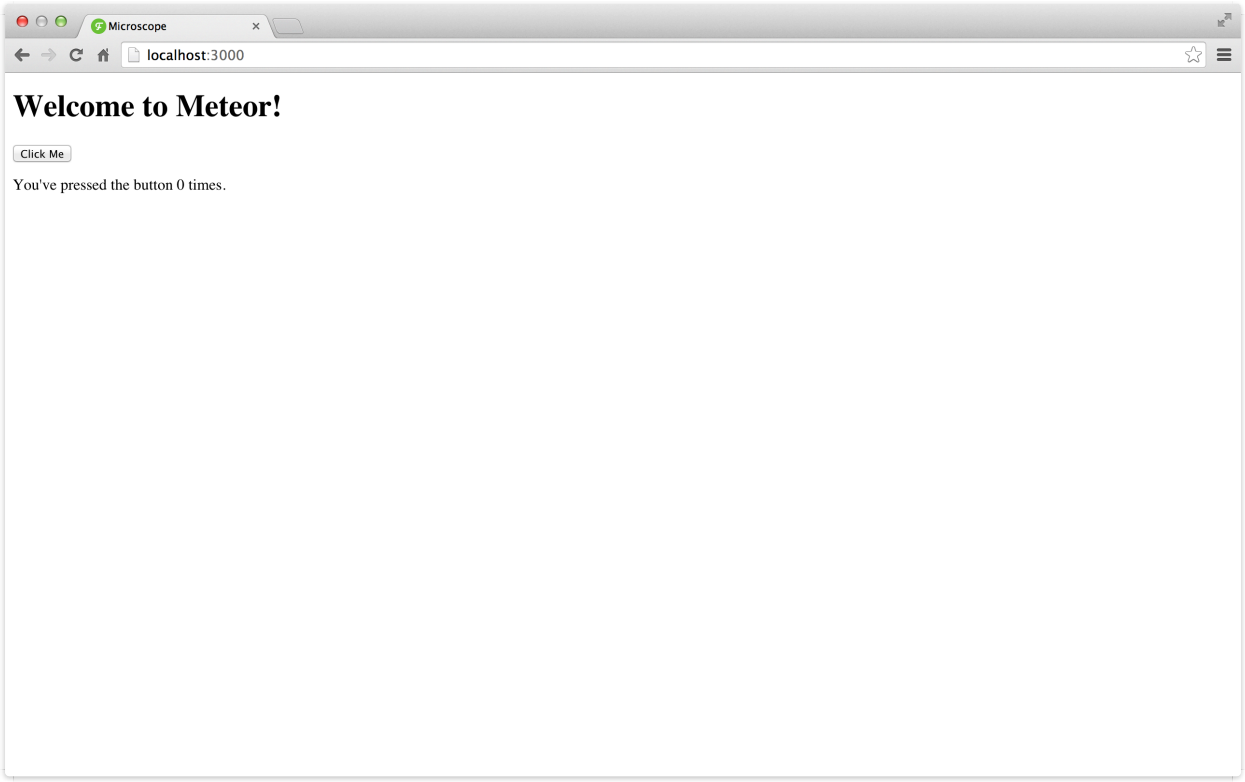
```
.meteor
microscope.css
microscope.html
microscope.js
```

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 throughout this book or 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 Meteor community member **Andrew Mao** (the "mizzao" in `mizzao:bootstrap-3` is the package author's username) to keep everything up to date for us.

While we're at it, we'll also add the **Underscore** package. Underscore is a JavaScript utility library, and it's very useful when it comes to manipulating JavaScript data structures.

As of this writing, the `underscore` package is still part of the "official" Meteor packages, which is why it doesn't have an author:

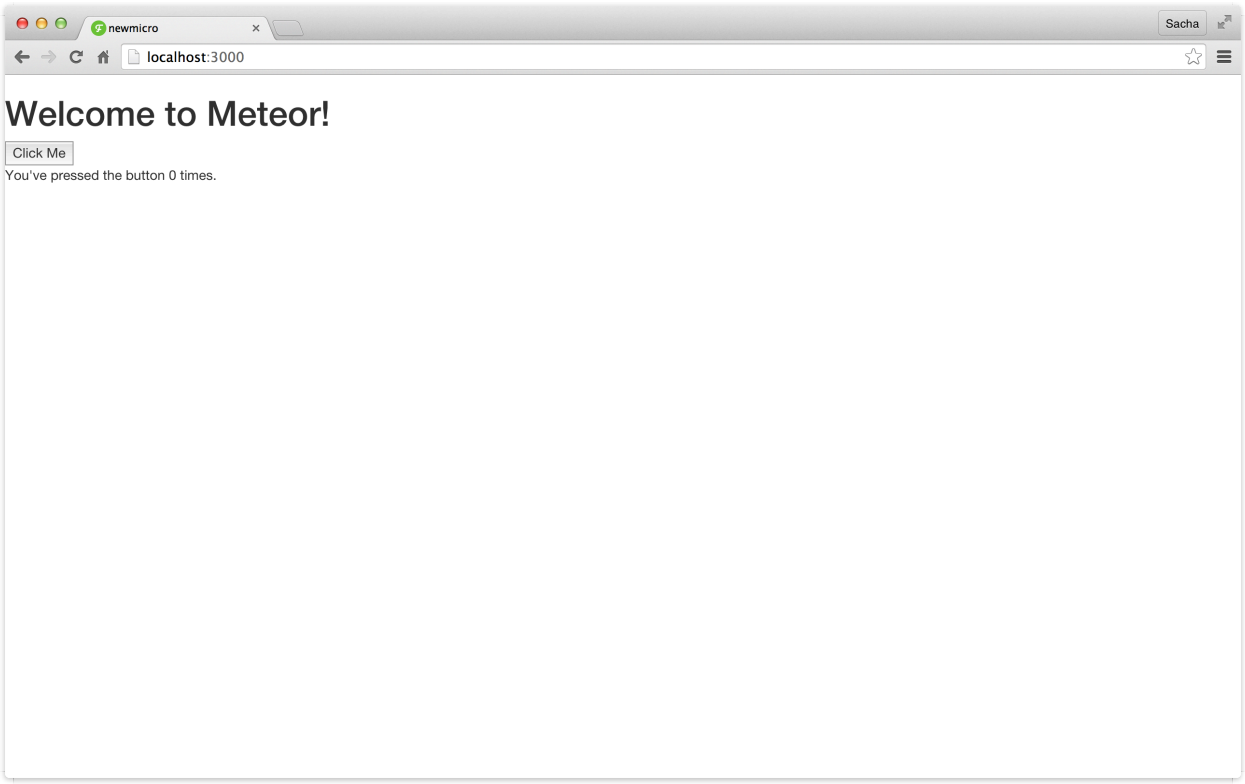
```
meteor add mizzao:bootstrap-3
meteor add underscore
```

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.

Commit 2-2
Added bootstrap and underscore packages.

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.

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 core itself is split into different **Meteor platform packages**. They are included with every Meteor app, and you will pretty much never need to worry about these.
- Regular Meteor packages are known as “**isopacks**”, or isomorphic packages (meaning they can work both on client and server). **First-party packages** such as `accounts-ui` or `appcache` are maintained by the Meteor core team and **come bundled with Meteor**.
- **Third-party packages** are just isopacks developed by other users that have been uploaded to Meteor’s package server. You can browse them on **Atmosphere** or with the `meteor search` command.
- **Local packages** are custom packages you can create yourself and put in the `/packages` directory.
- **NPM packages** (Node.js Packaged Modules) are Node.js packages. Although they don’t work out of the box with Meteor, they *can* be used by the previous types of packages.

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 `microscope` directory and delete `microscope.html`, `microscope.js`, and `microscope.css`.

Next, create four root directories inside `/microscope`: `/client`, `/server`, `/public`, and `/lib`.

Next, we’ll also create empty `main.html` and `main.js` files inside `/client`. Don’t worry if 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/stylesheets/` directory now, and put this `style.css` file inside it:

```
.grid-block, .main, .post, .comments li, .comment-form {
  background: #fff;
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  -ms-border-radius: 3px;
  -o-border-radius: 3px;
  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; }

.navbar {
  margin-bottom: 10px; }
  /* line 32, ../sass/style.scss */
  .navbar .navbar-inner {
    -webkit-border-radius: 0px 0px 3px 3px;
    -moz-border-radius: 0px 0px 3px 3px;
    -ms-border-radius: 0px 0px 3px 3px;
    -o-border-radius: 0px 0px 3px 3px;
    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;
```

```
    -webkit-transition-delay: ease-in;
    -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;
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  -ms-border-radius: 3px;
  -o-border-radius: 3px;
  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;
}
```

client/stylesheets/style.css

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 Meteor

Deploying on a Meteor subdomain (i.e. `http://myapp.meteor.com`) is the easiest option, and the first one we'll try. This can be useful to showcase your app to others in its early days, or to quickly set up a staging server.

Deploying on Meteor is pretty simple. Just open up your terminal, go to to your Meteor app's directory, and type:

```
meteor deploy myapp.meteor.com
```

Of course, you'll have to take care to replace "myapp" with a name of your choice, preferably one that isn't already in use.

If this is your first time deploying an app, you'll be prompted to create a Meteor account. And if all goes well, after a few seconds you'll be able to access your app at `http://myapp.meteor.com`.

You can refer to **the official documentation** for more information on things like accessing your hosted instance's database directly, or configuring a custom domain for your app.

Deploying On Modulus

Modulus is a great option for deploying Node.js apps. It's one of the few PaaS (platform-as-a-service) provider that officially support Meteor, and there are already quite a few people running production Meteor apps on it.

> ### Demeteorizer
>
> Modulus open-sourced a tool called **demeteorizer** which converts your Meteor app into a standard Node.js app.

Start by **creating an account**. To deploy our app on Modulus, we'll then need to install the Modulus command line tool:

```
npm install -g modulus
```

And then authenticate with:

```
modulus login
```

We'll now create a Modulus project (note that you can also do this via Modulus' web dashboard):

```
modulus project create
```

The next step will be creating a MongoDB database for our app. We can create a MongoDB database with **Modulus itself**, **MongoHQ** or with any other cloud MongoDB provider.

Once we've created our MongoDB database, we can get the `MONGO_URL` for our database from Modulus' web UI (go to Dashboard > Databases > Select your database > Administration), then use it to configure our app like so:

```
modulus env set MONGO_URL "mongodb://<user>:<pass>@mongo.onmodulus.net:27017/<database_name>"
```

It's now time to deploy our app. It's as simple as typing:

```
modulus deploy
```

We've now successfully deployed our app to Modulus. Refer to **the Modulus documentation** for more information about accessing logs, custom domain setup, and SSL.

Meteor Up

Although new cloud solutions are appearing every day, they often come with their own share of problems and limitations. So as of today, deploying on your own server remains the best way to put a Meteor application in production. The only thing is, deploying yourself is not that simple, especially if you're looking for production-quality deployment.

Meteor Up (or `mup` for short) is another attempt at fixing that issue, with a command-line utility that takes care of setup and deployment for you. So let's see how to deploy Microscope using Meteor Up.

Before anything else, we'll need a server to push to. We recommend either **Digital Ocean**, which

starts at $5 per month, or **AWS**, which provides Micro instances for free (you'll quickly run into scaling problems, but if you're just looking to play around with Meteor Up it should be enough).

Whichever service you choose, you should end up with three things: your server's IP address, a login (usually `root` or `ubuntu`), and a password. Keep those somewhere safe, we'll need them soon!

Initializing Meteor Up

To start out, we'll need to install Meteor Up via `npm` as follows:

```
npm install -g mup
```

We'll then create a special, separate directory that will hold our Meteor Up settings for a particular deployment. We're using a separate directory for two reasons: first, it's usually best to avoid including any private credentials in your Git repo, especially if you're working on a public codebase.

Second, by using multiple separate directories, we'll be able to manage multiple Meteor Up configurations in parallel. This will come in handy for deploying to production and staging instances, for example.

So let's create this new directory and use it to initialize a new Meteor Up project:

```
mkdir ~/microscope-deploy
cd ~/microscope-deploy
mup init
```

> Sharing with Dropbox
>
> A great way to make sure you and your team all use the same deployment settings is to simply create your Meteor Up configuration folder inside your Dropbox, or any similar service.

Meteor Up Configuration

When initializing a new project, Meteor Up will create two files for you: `mup.json` and `settings.json`.

`mup.json` will hold all our deployment-related settings, while `settings.json` will contain all app-related settings (OAuth tokens, analytics tokens, etc.).

The next step is to configure your `mup.json` file. Here is the default `mup.json` file generated by `mup init`, and all you have to do is fill in the blanks:

```
{
  //server authentication info
  "servers": [{
    "host": "hostname",
    "username": "root",
    "password": "password"
    //or pem file (ssh based authentication)
    //"pem": "~/.ssh/id_rsa"
  }],

  //install MongoDB in the server
  "setupMongo": true,

  //location of app (local directory)
  "app": "/path/to/the/app",

  //configure environmental
  "env": {
    "ROOT_URL": "http://supersite.com"
  }
}
```

mup.json

Let's walk through each of these settings.

Server Authentication

You'll notice that Meteor Up supports password based and private key (PEM) based authentication, so it can be used with almost any cloud provider.

Important note: if you choose to use password-based authentication, make sure you've installed `sshpass` first (**refer to this guide**).

MongoDB Configuration

The next step is to configure a MongoDB database for your app. We recommend using **MongoHQ** or any other cloud MongoDB provider, since they offer professional support and better management tools.

If you've decided to use MongoHQ, set `setupMongo` as `false` and add the `MONGO_URL` environmental variable in `mup.json`'s `env` block. If you decided to host MongoDB with Meteor Up, just set `setupMongo` as `true` and Meteor Up will take care of the rest.

Meteor App Path

Since our Meteor Up configuration lives in a different directory, we'll need to point Meteor Up back to our app using the `app` property. Just input your full local path, which you can get using the `pwd` command from the terminal when located inside your app's directory.

Environment Variables

You can specify all of your app's environment variables (such as `ROOT_URL`, `MAIL_URL`, `MONGO_URL`, etc.) inside the `env` block.

Setting Up and Deploying

Before we can deploy, we'll need to set up the server so it's ready to host Meteor apps. The magic of Meteor Up encapsulates this complex process in a single command!

```
mup setup
```

This will take a few minutes depending on the server's performance and the network connectivity. After the setup is successful, we can finally deploy our app with:

```
mup deploy
```

This will bundle the meteor app, and deploy to the server we just set up.

Displaying Logs

Logs are pretty important and Meteor Up provides a very easy way to handle them by emulating the `tail -f` command. Just type:

```
mup logs -f
```

This wraps up our overview of what Meteor Up can do. For more infomation, we suggest visiting **Meteor Up's GitHub repository**.

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" class="row-fluid">
      {{> postsList}}
    </div>
  </div>
</body>
```

client/main.html

This will be our main app template. As you can see it's all HTML except for a single `{{> postsList}}` 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.

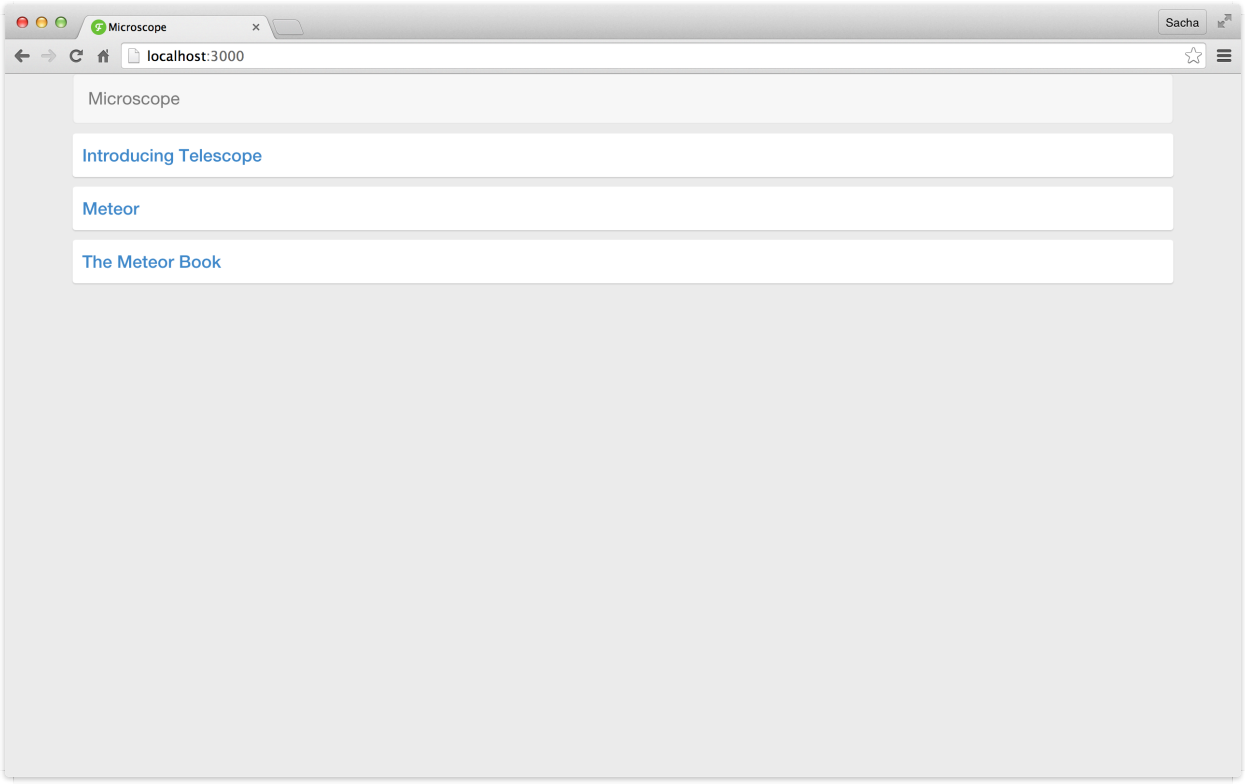
> ### Controllers?
>
> It might be tempting to think of the file containing all of a template's helpers as a controller of sorts. But that can be ambiguous, as controllers (at least in the MVC sense) usually have a slightly different role.
>
> So we decided to stay away from that terminology, and simply 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://themeteorbook.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">
    {{#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

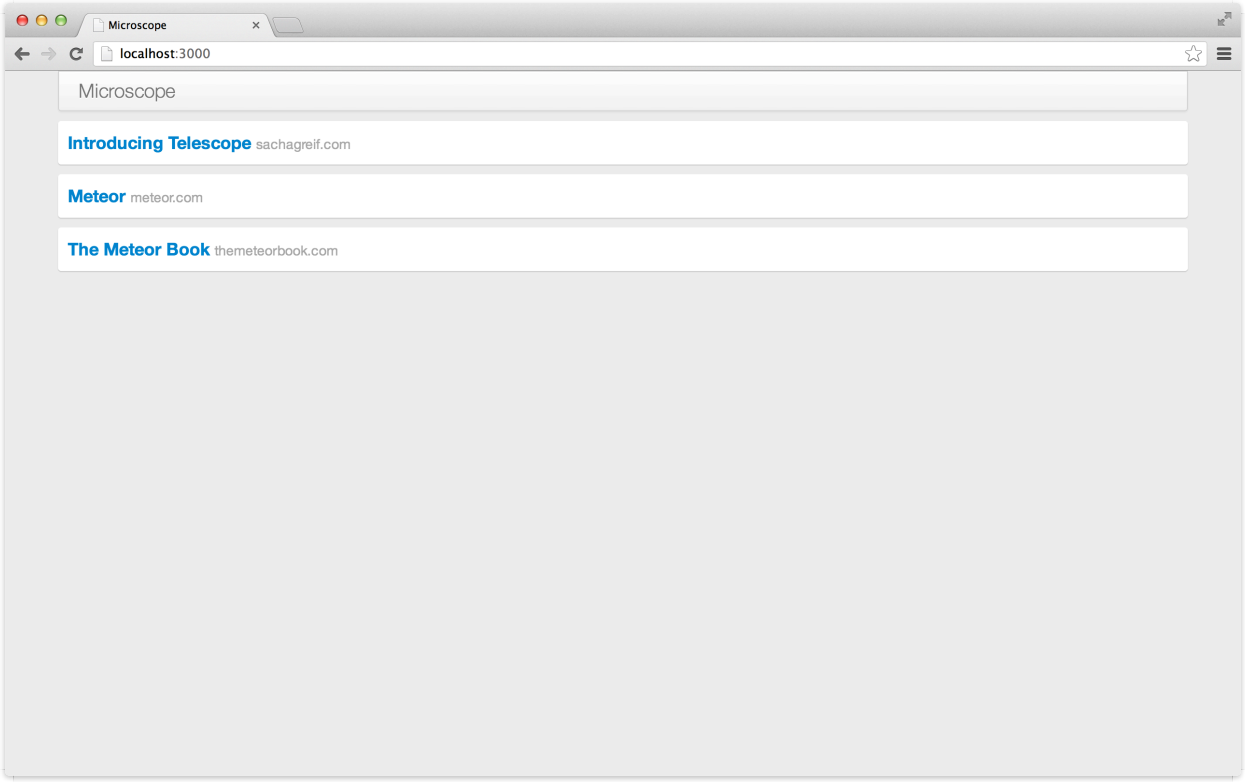
The `domain` Helper

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*.

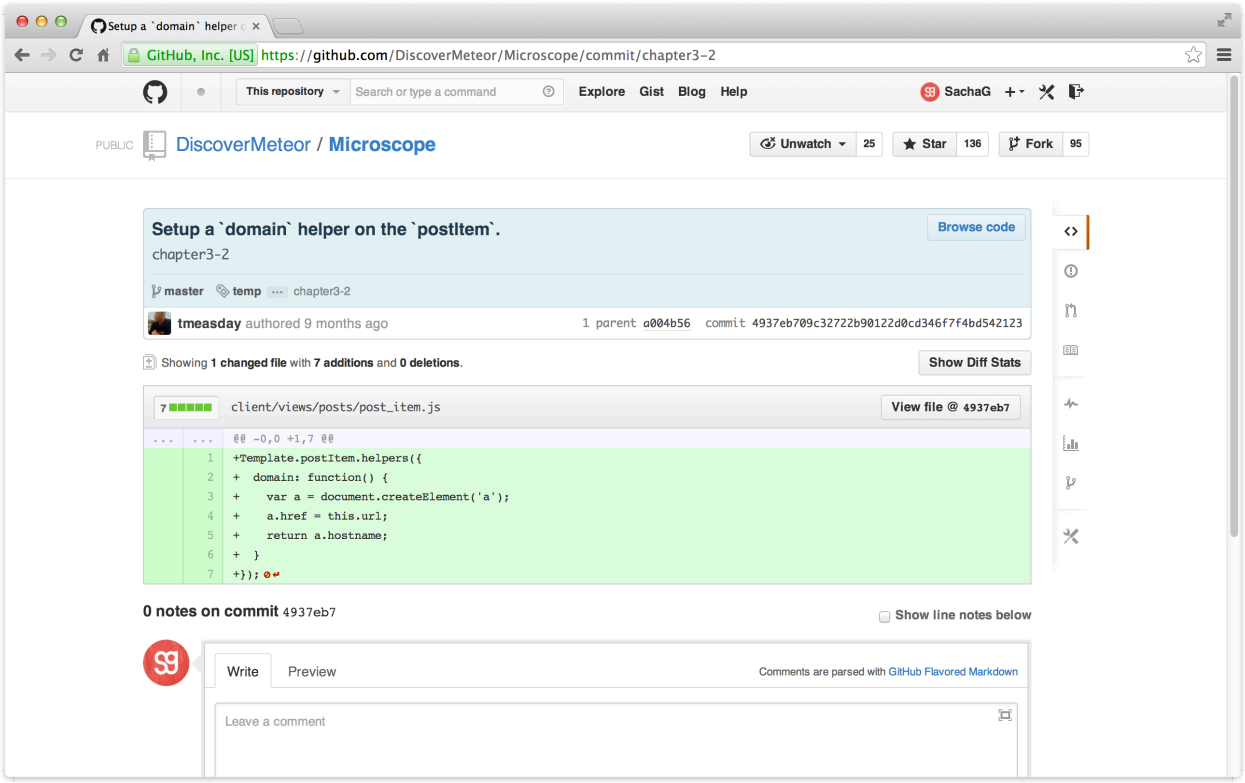
This sidebar assumes you're not that familiar with Git and GitHub. If you're already comfortable with both, feel free to skip on to the next chapter!

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.

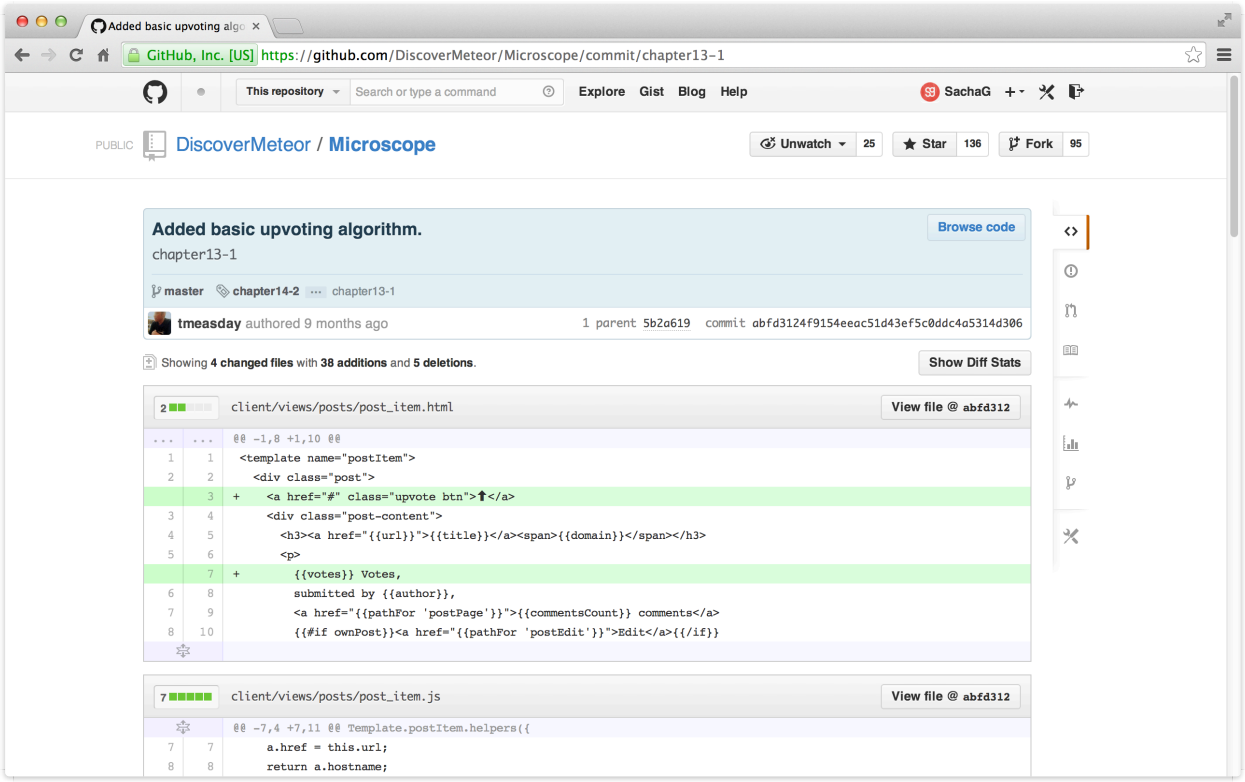
For example, this is what **the last commit of the previous chapter** looks like:



A Git commit as shown on GitHub.

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.

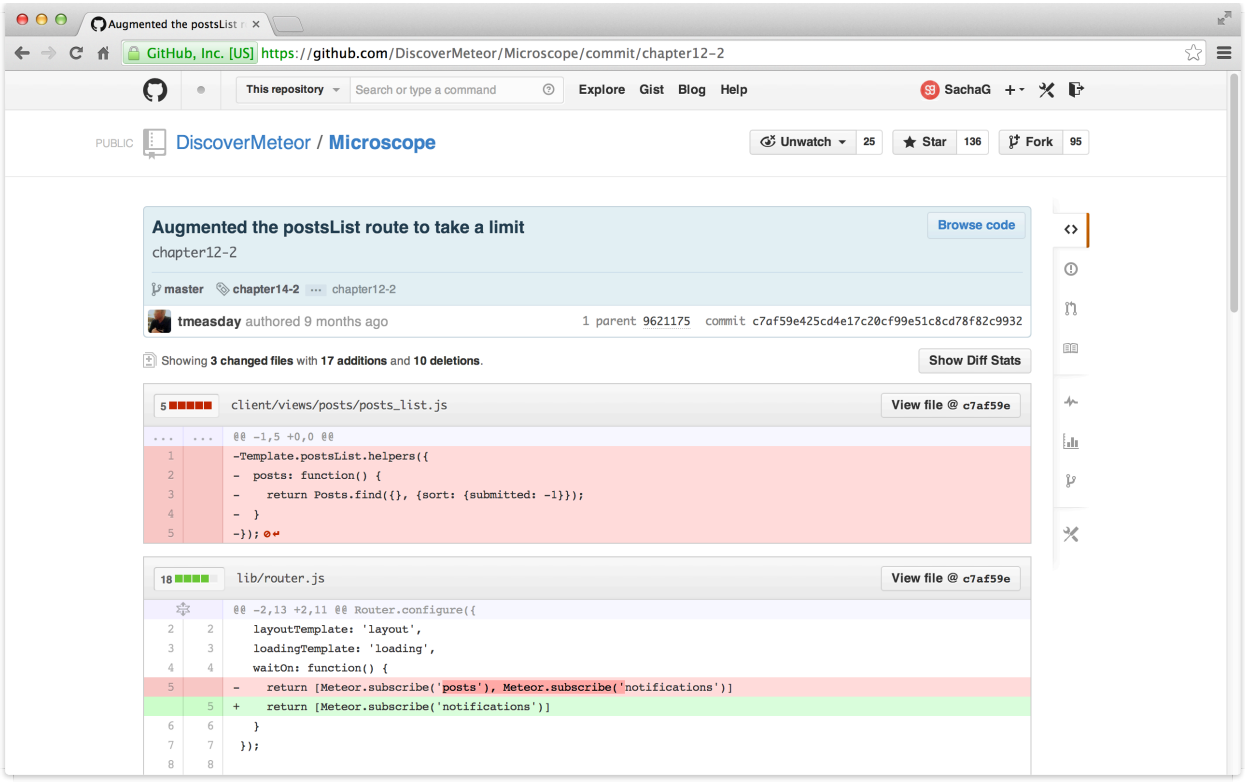
Let's compare with an example from **later on in the book**:



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**:



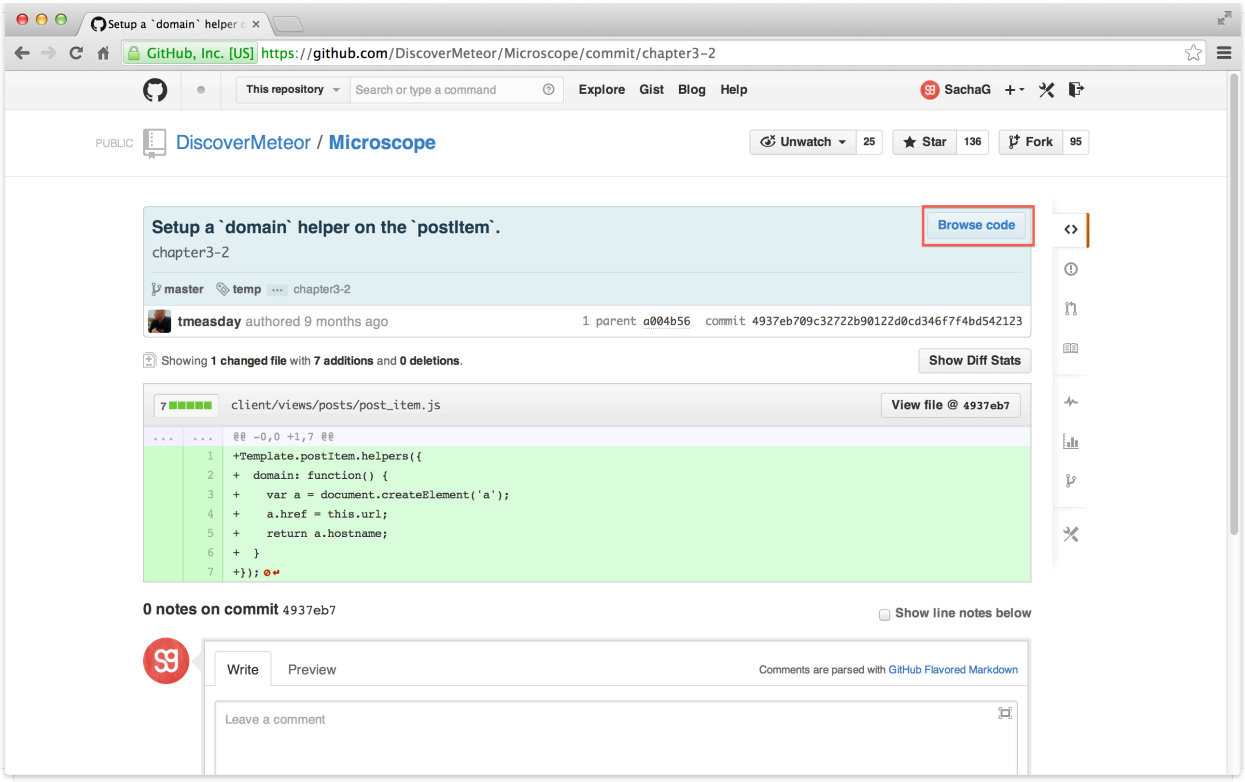
Deleting code.

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

Browsing A Commit's Code

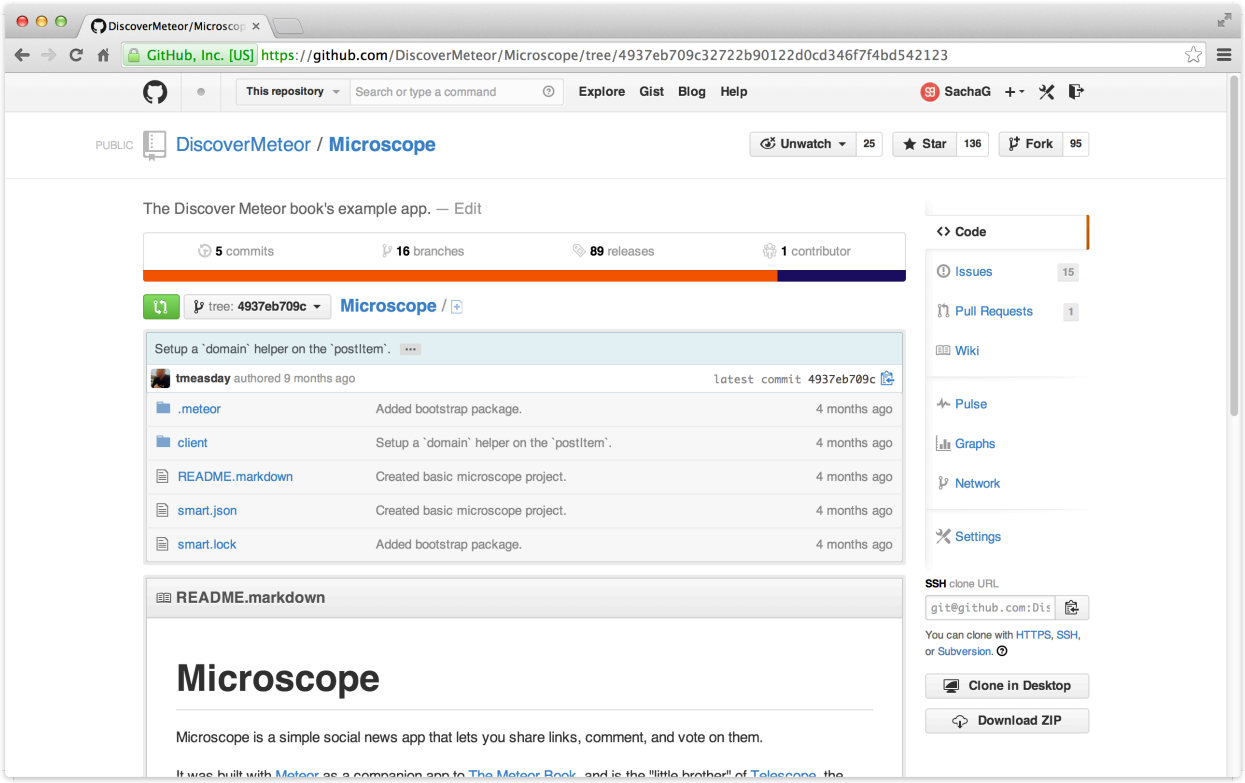
Git's commit view shows us the changes included in this commit, but sometimes you might want to look at files that *haven't* changed, just to make sure what their code is supposed to look like at this stage of the process.

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



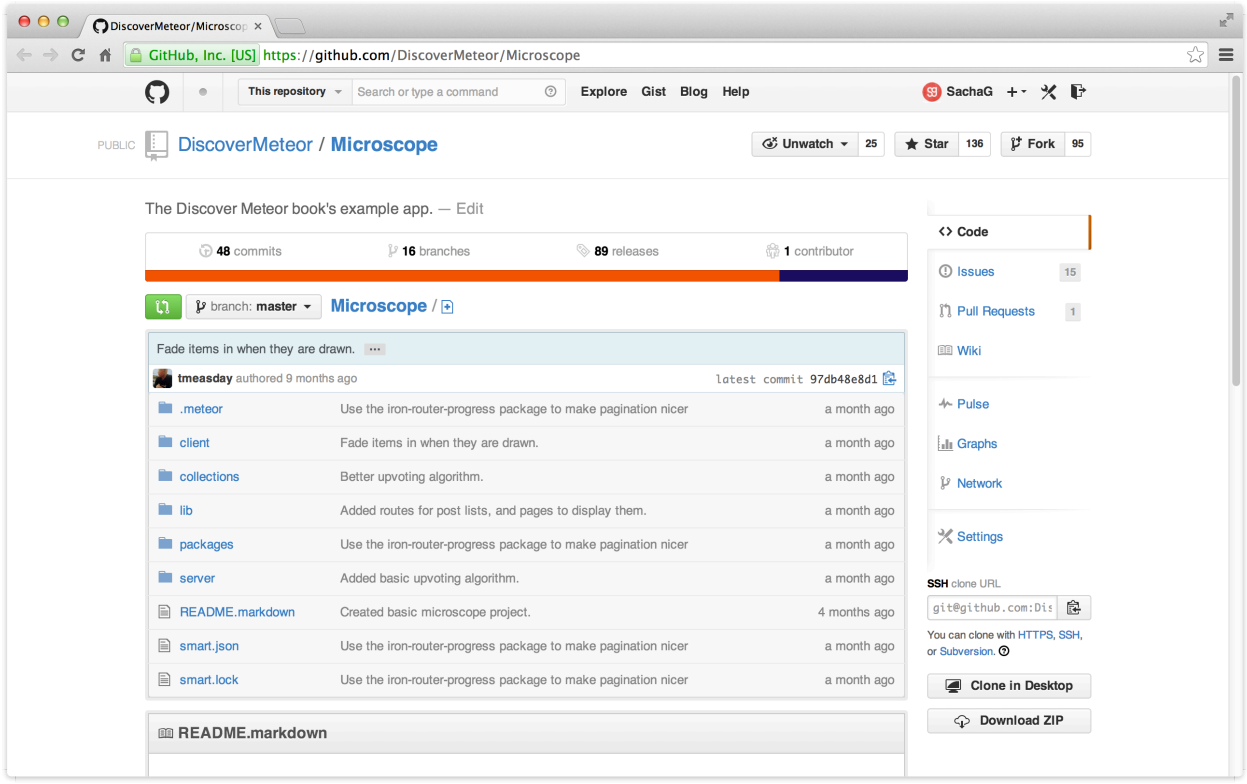
The Browse code button.

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 git@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.
```

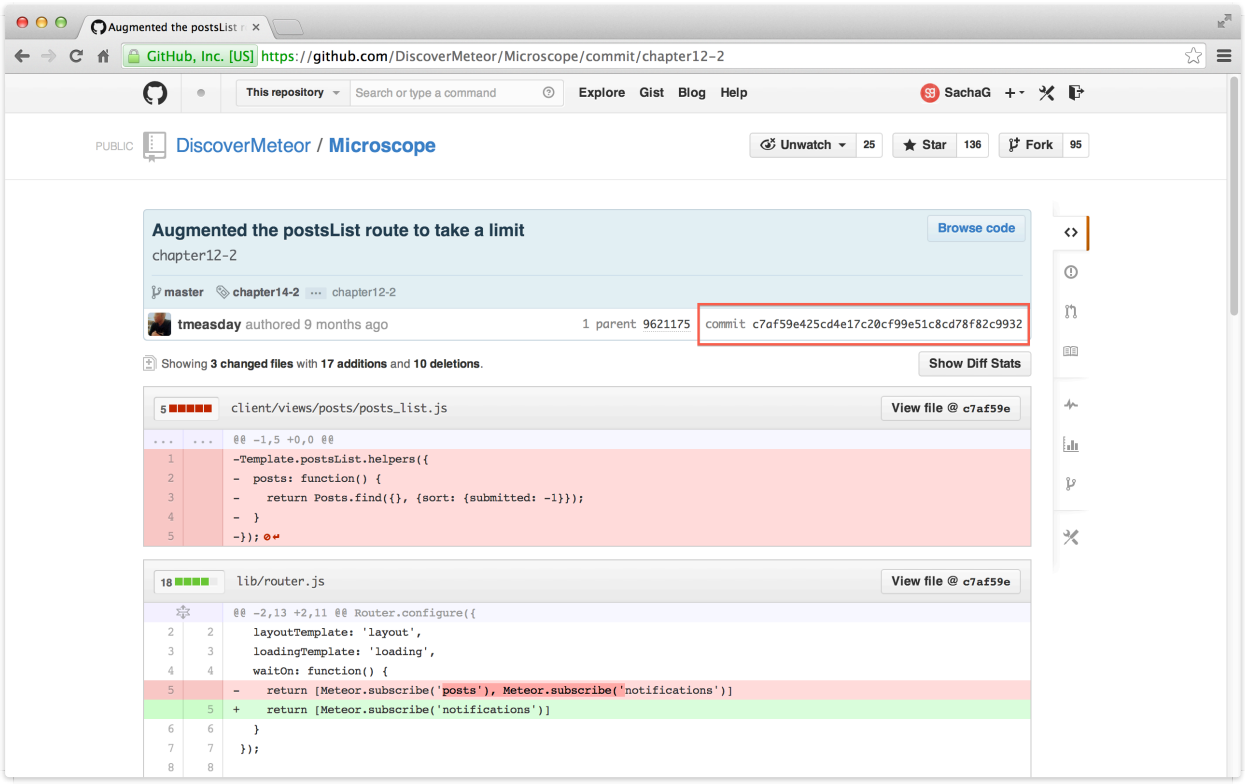
Git informs us that we are in "detached HEAD" state, which means that as far as Git is concerned, we can observe past commits but we can't modify them. You can think of it as a wizard inspecting the past through a crystal ball.

(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:

```
git checkout c7af59e425cd4e17c20cf99e51c8cd78f82c9932
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:

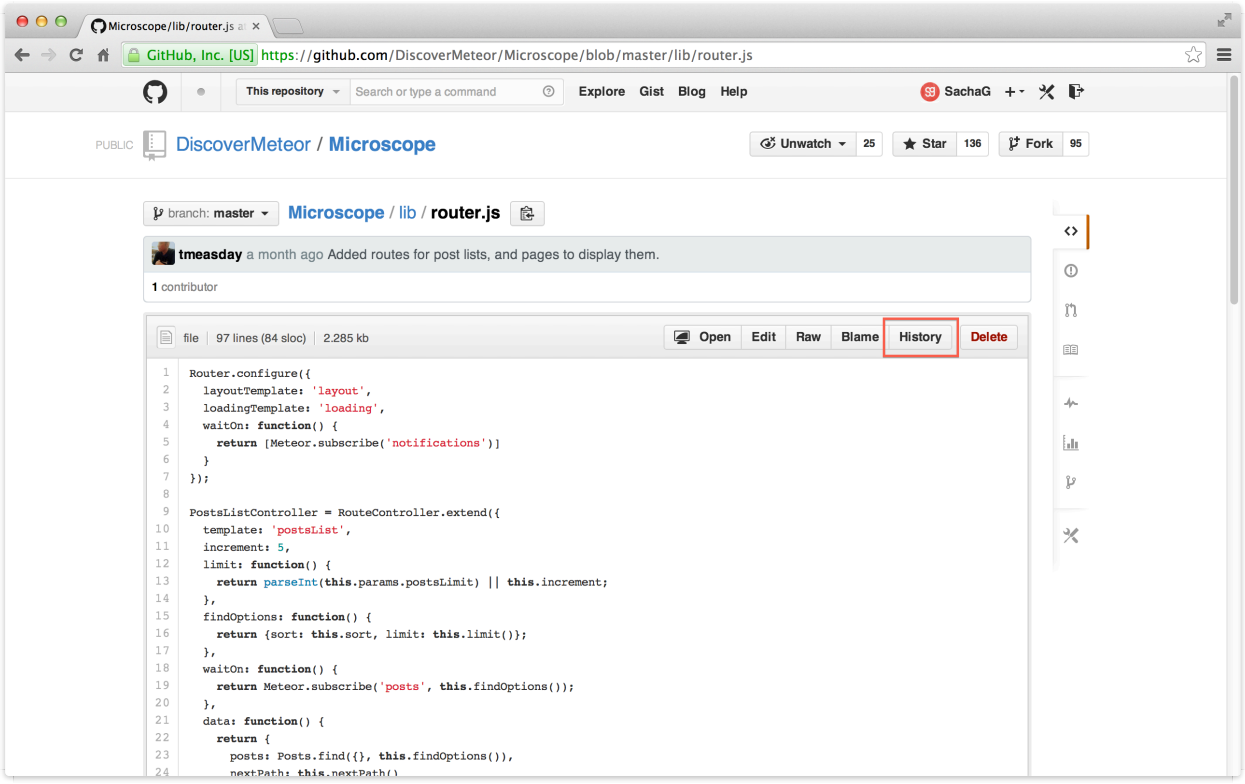
```
git checkout master
```

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

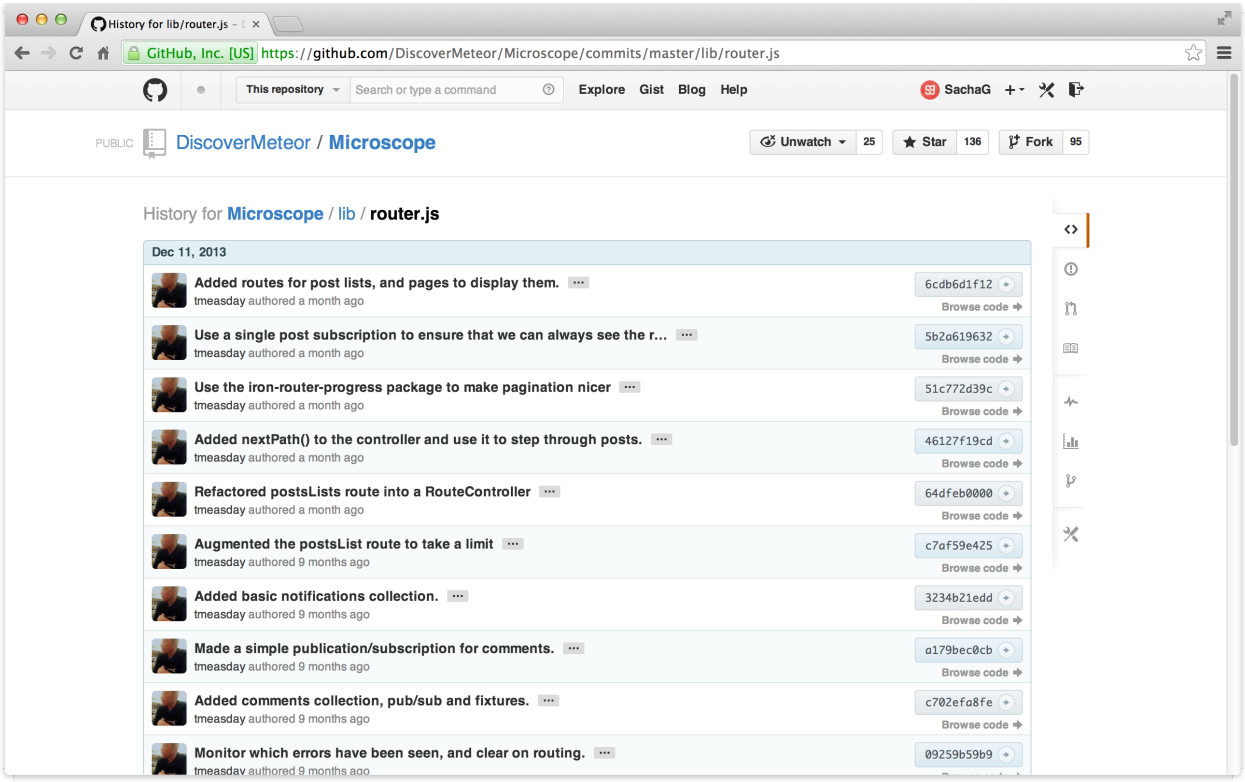
Here's another common scenario: you're looking at a file and notice some changes you hadn't seen before. The thing is, you can't remember *when* the file changed. You could just look at each commit one by one until you find the right one, but there's an easier way thanks to GitHub's **History** feature.

First, access one of your repository's files on GitHub, then locate the "History" button:



GitHub's History button.

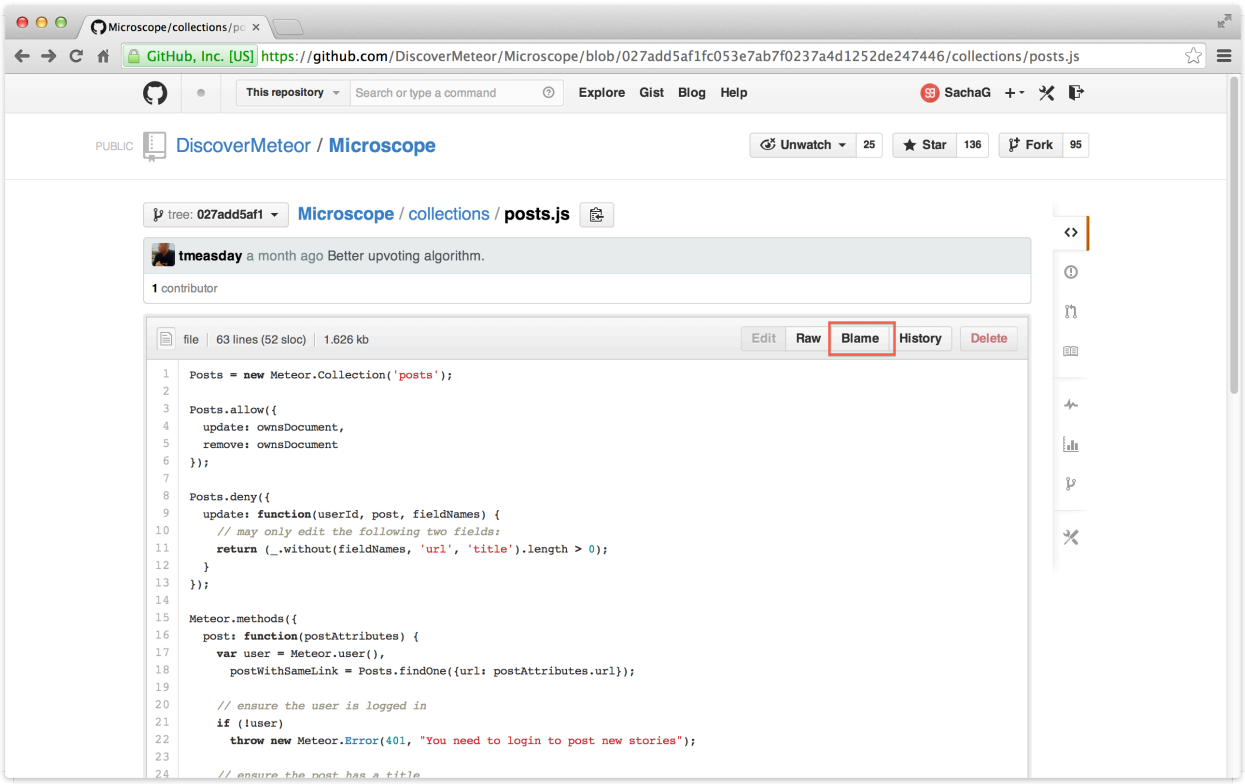
You now have a neat list of all the commits that affected this particular file:



Displaying a file's history.

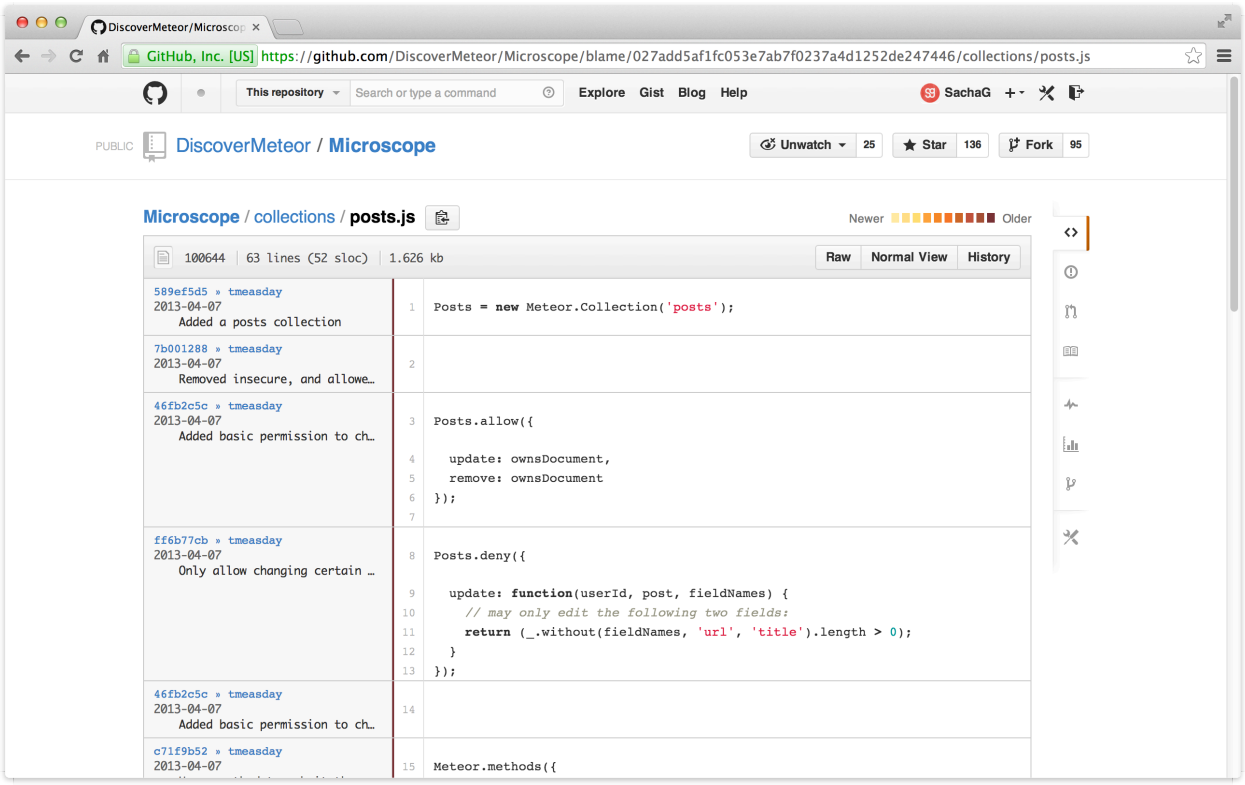
The Blame Game

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



GitHub's Blame button.

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):



GitHub's Blame view.

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.

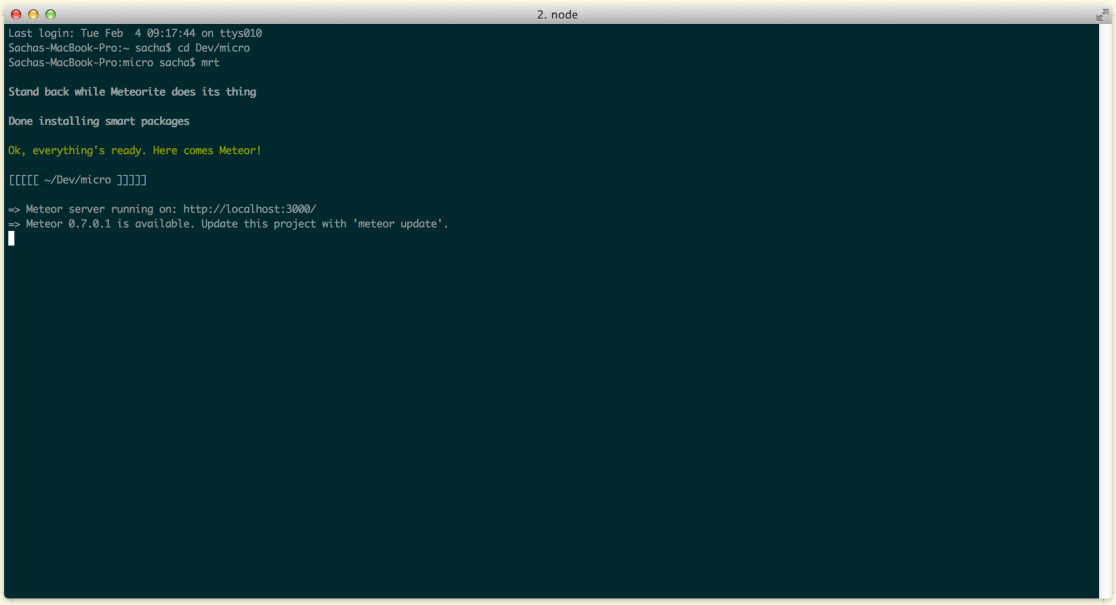
On the server, the collection has the job of talking to the MongoDB database, and reading and writing any changes. In this sense, it can be compared to a standard database library.

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** or the **Mongo shell**. Here's a quick primer on each of them.

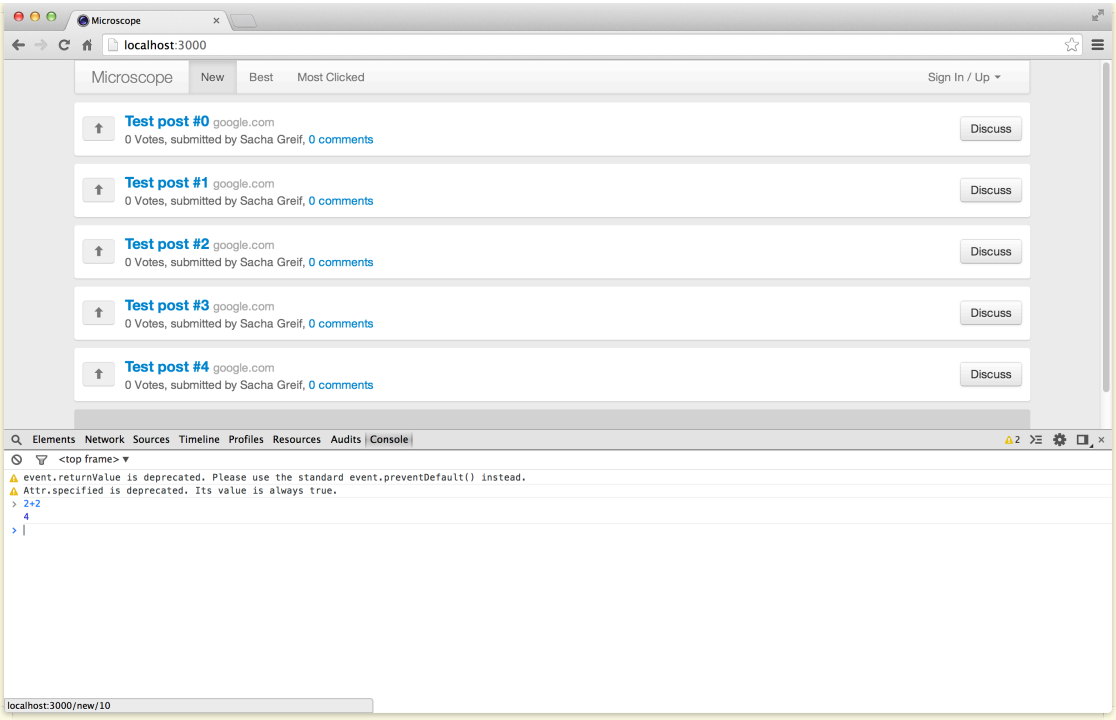
Terminal



The Terminal

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

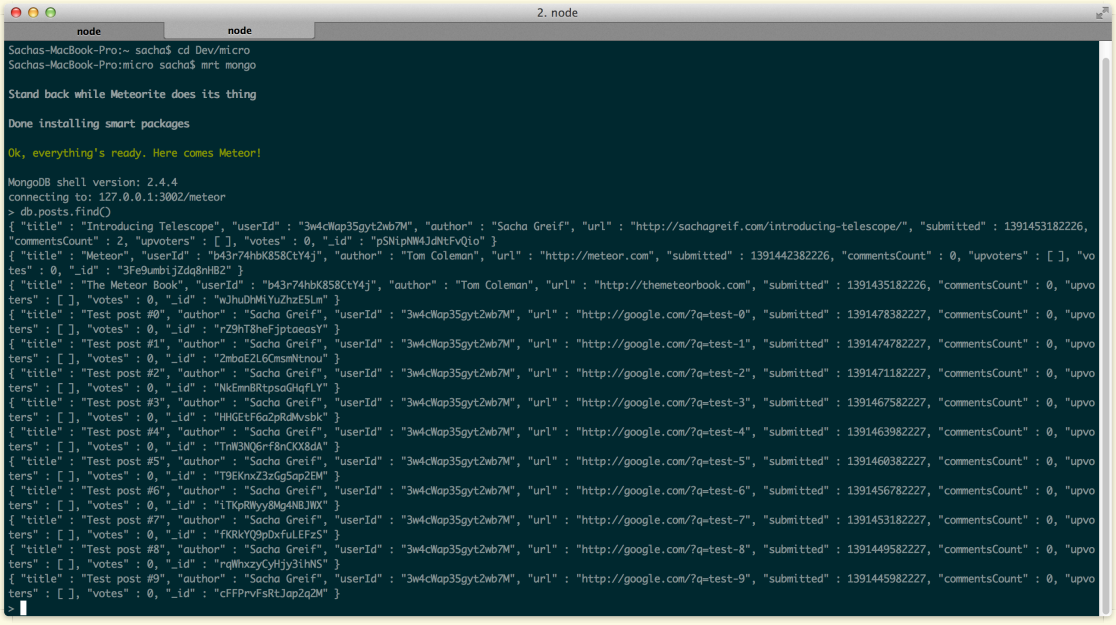
Browser Console



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

Mongo Shell



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://themeteorbook.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.

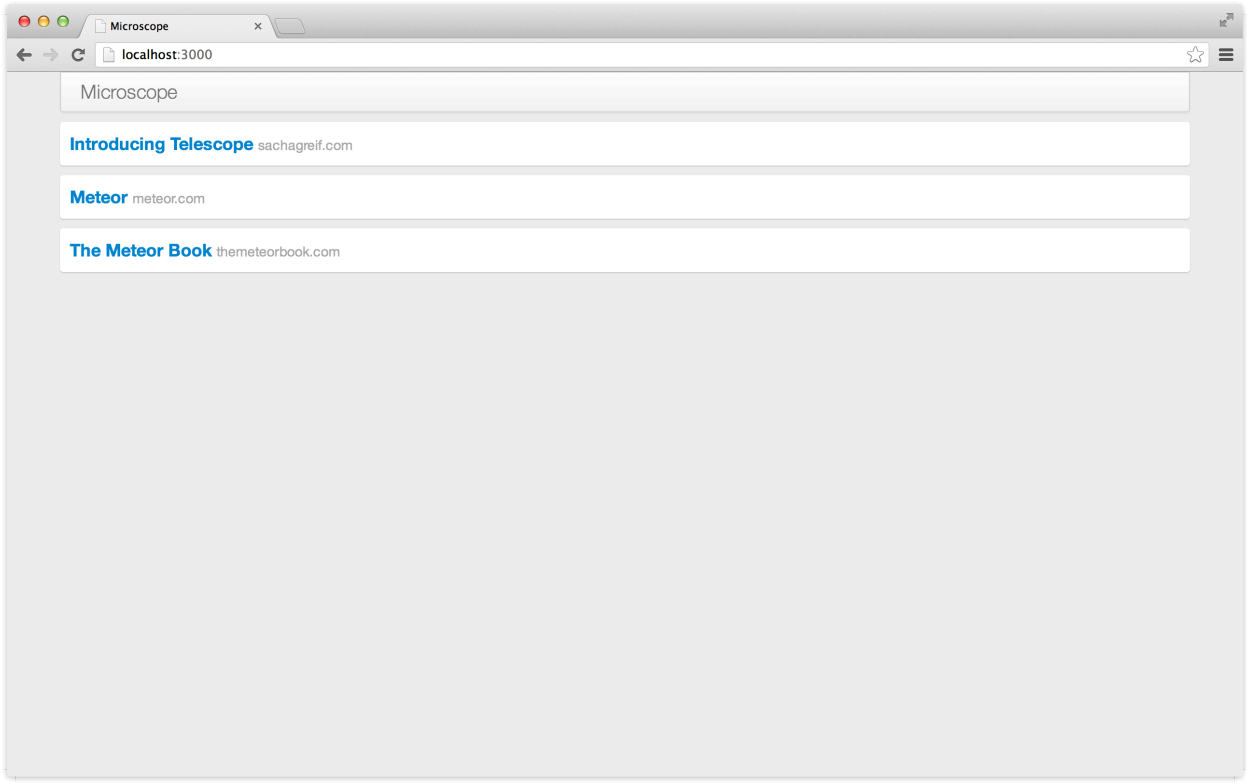
View on GitHub | Launch Instance

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

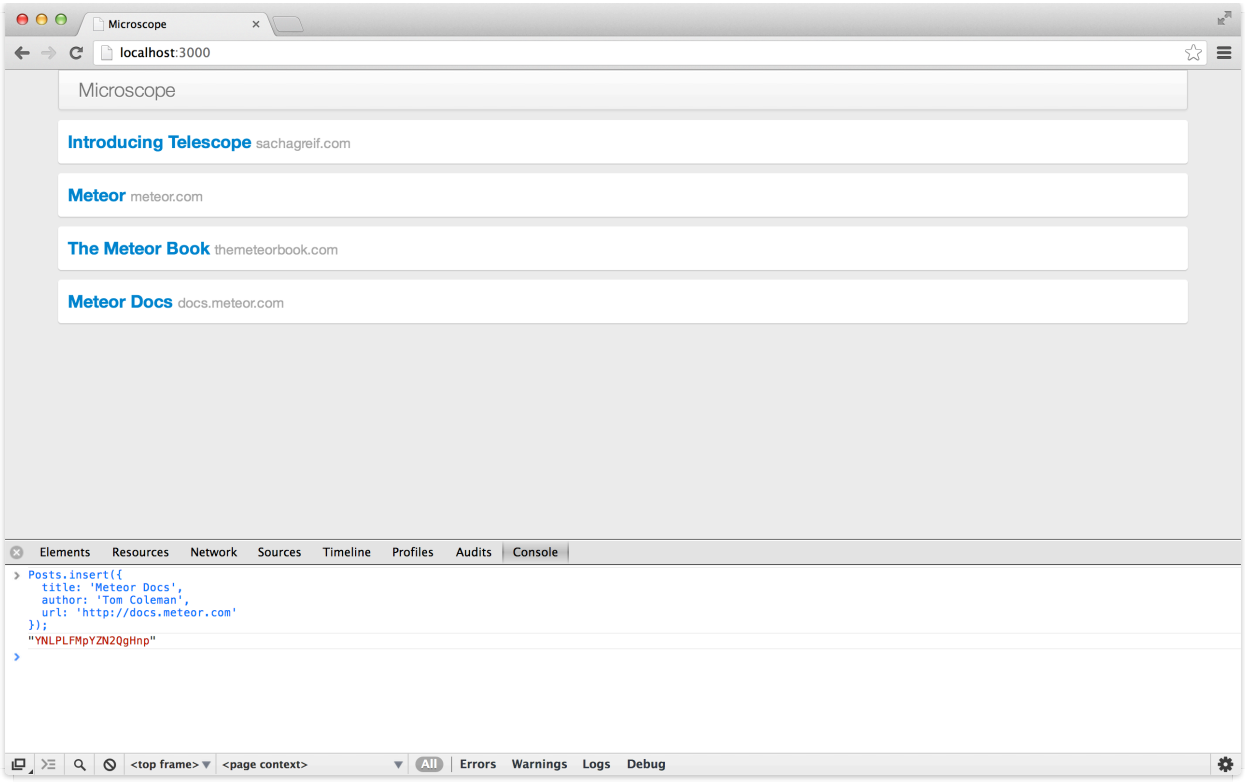
Our `{{#each}}` helper has iterated over all of our `Posts` and displayed them on the screen. The server-side collection pulled the posts from Mongo, passed them over the wire to our client-side collection, and our Spacebars helper passed them into the template.

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.

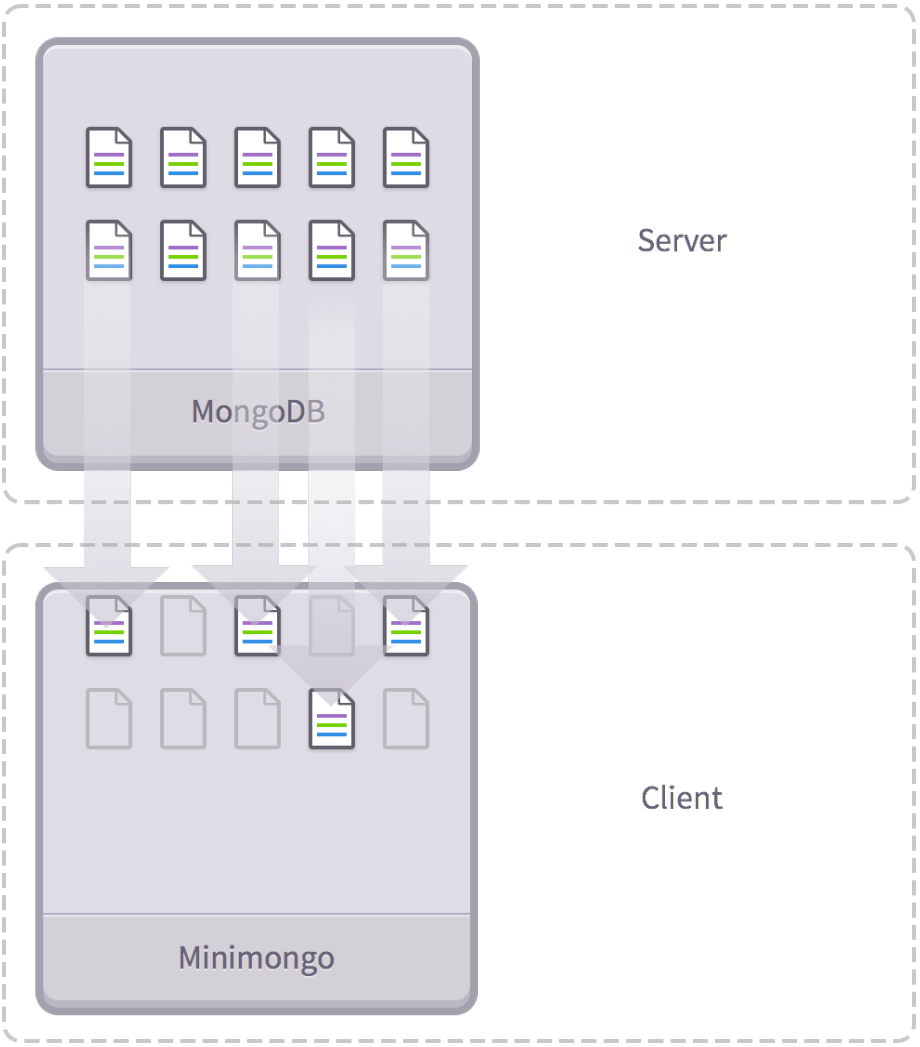
The app's first job is to figure out what data the user needs to see. This could be page 12 of search results, Mary's user profile information, Bob's 20 latest tweets, and so on. You can basically think of it as a bookstore clerk browsing through the aisles for the book you asked for.

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.

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 to Microscope. 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.

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:

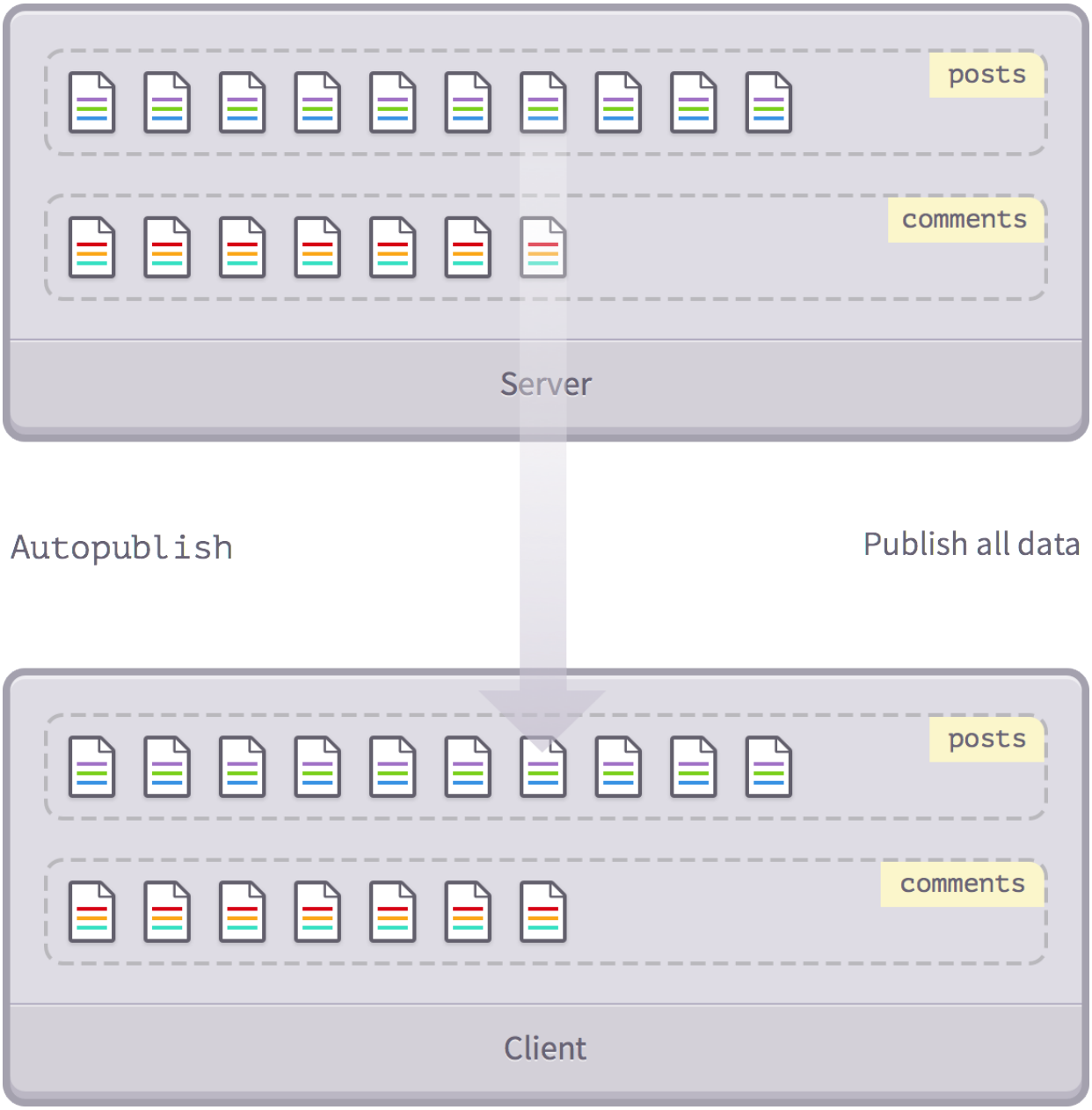
```
// on the client
Template.posts.helpers({
    posts: function(){
        return Posts.find({author: 'bob-smith', category: 'JavaScript'});
    }
});
```

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).

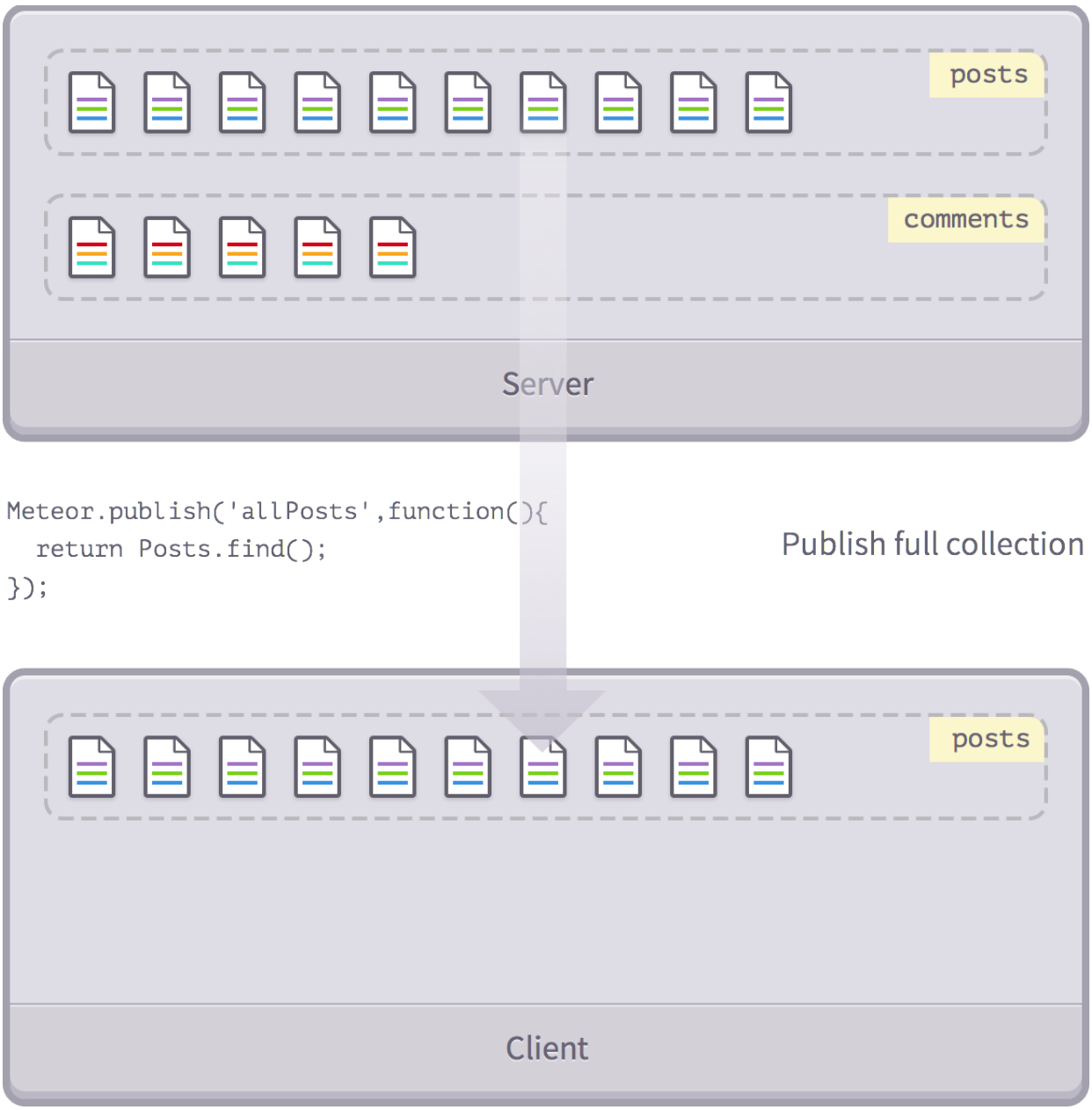
So if you are using `autopublish`, you don't need to think about publications. Data is ubiquitous, and things are simple. Of course, there are obvious problems with having a complete copy of your app's database cached on every user's machine.

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();
});
```



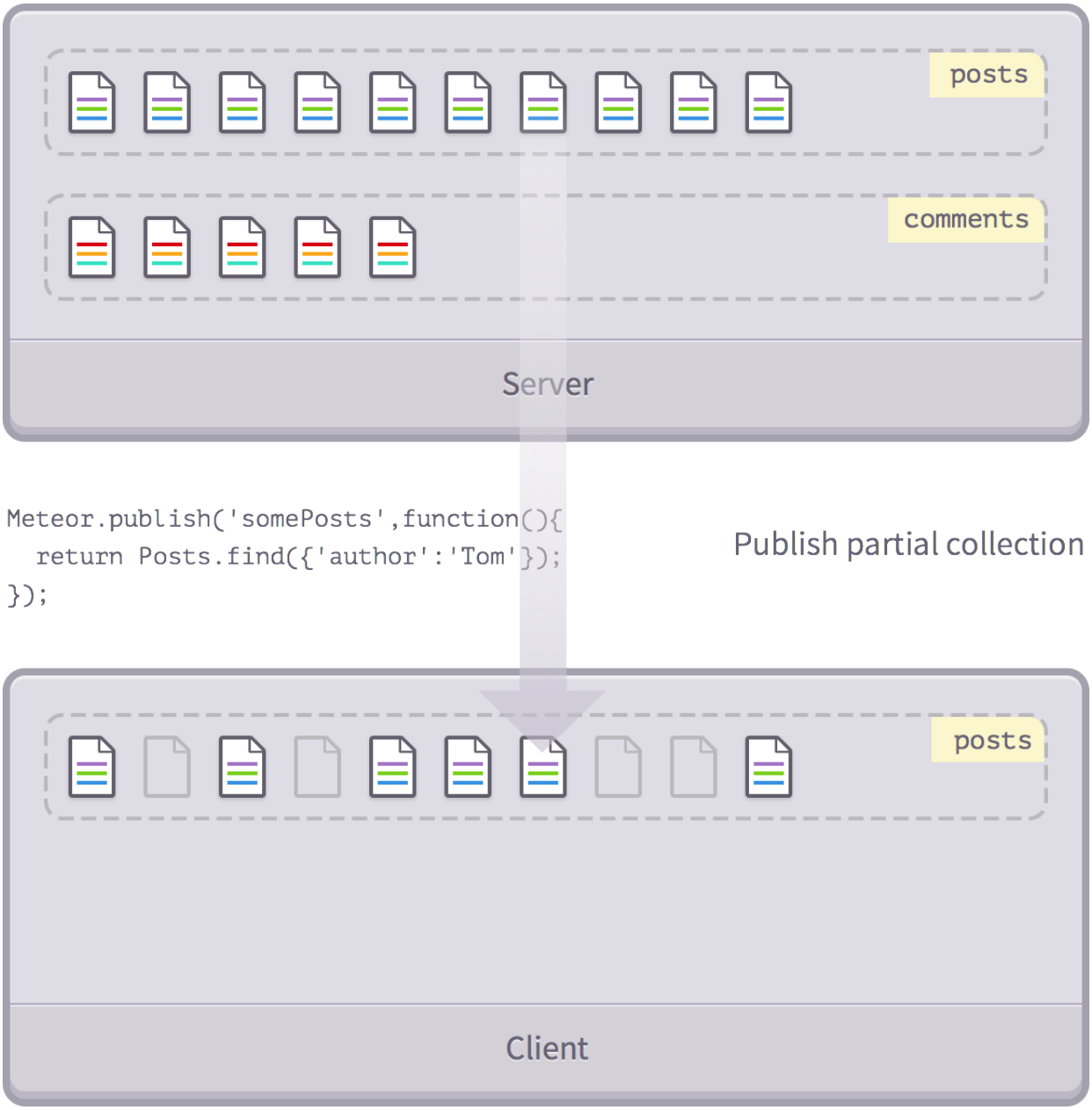
Publishing a full collection

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).

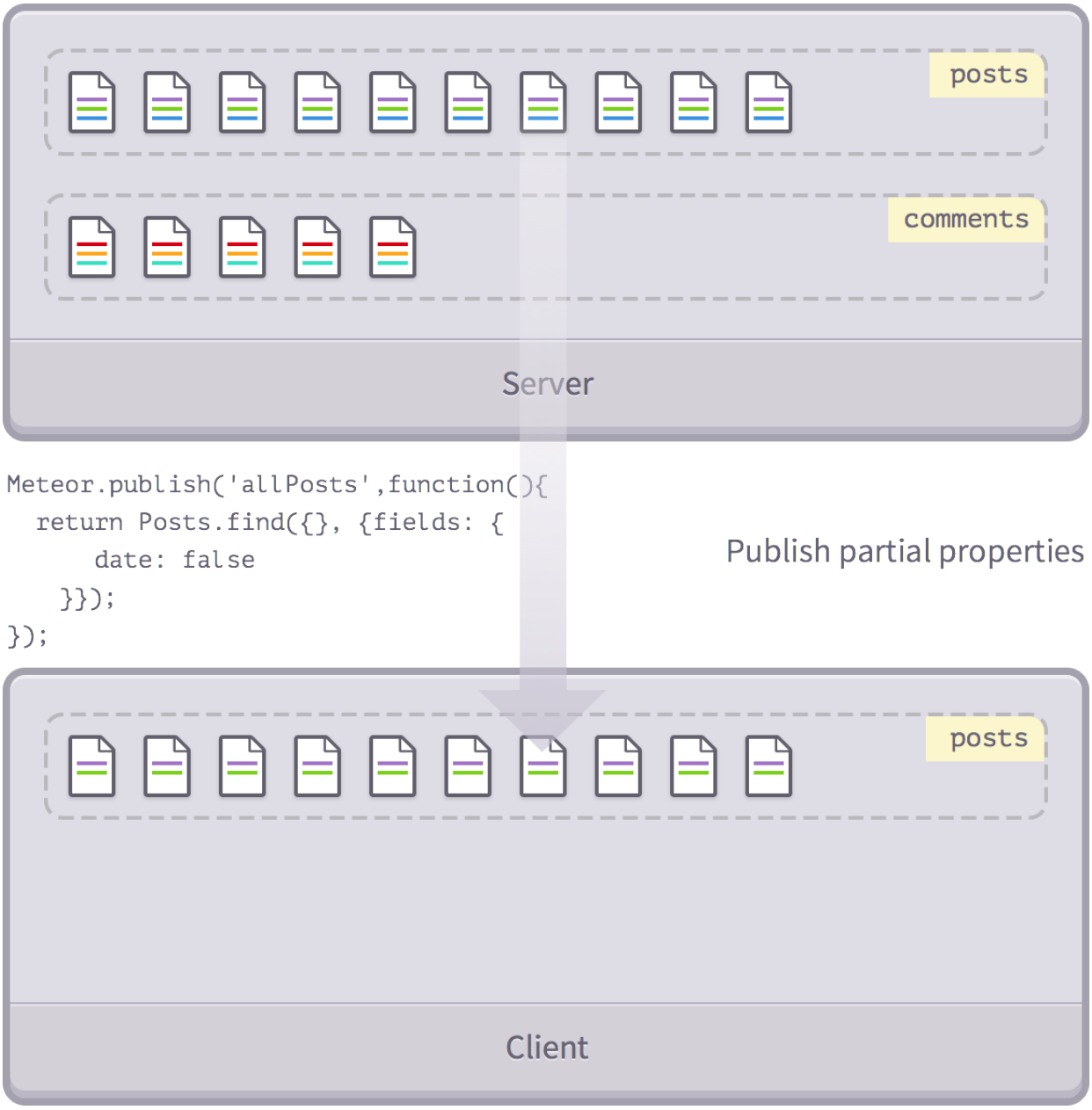
So in the example above, we are able to make sure that the user only has the posts that they are interested in (the ones written by Tom) available to them in their client side cache.

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" class="row-fluid">
      {{> 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 go blank and an error will show up in the browser console. 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 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 sacha:spin`, 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">
  {{> postItem}}
</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'
});
```

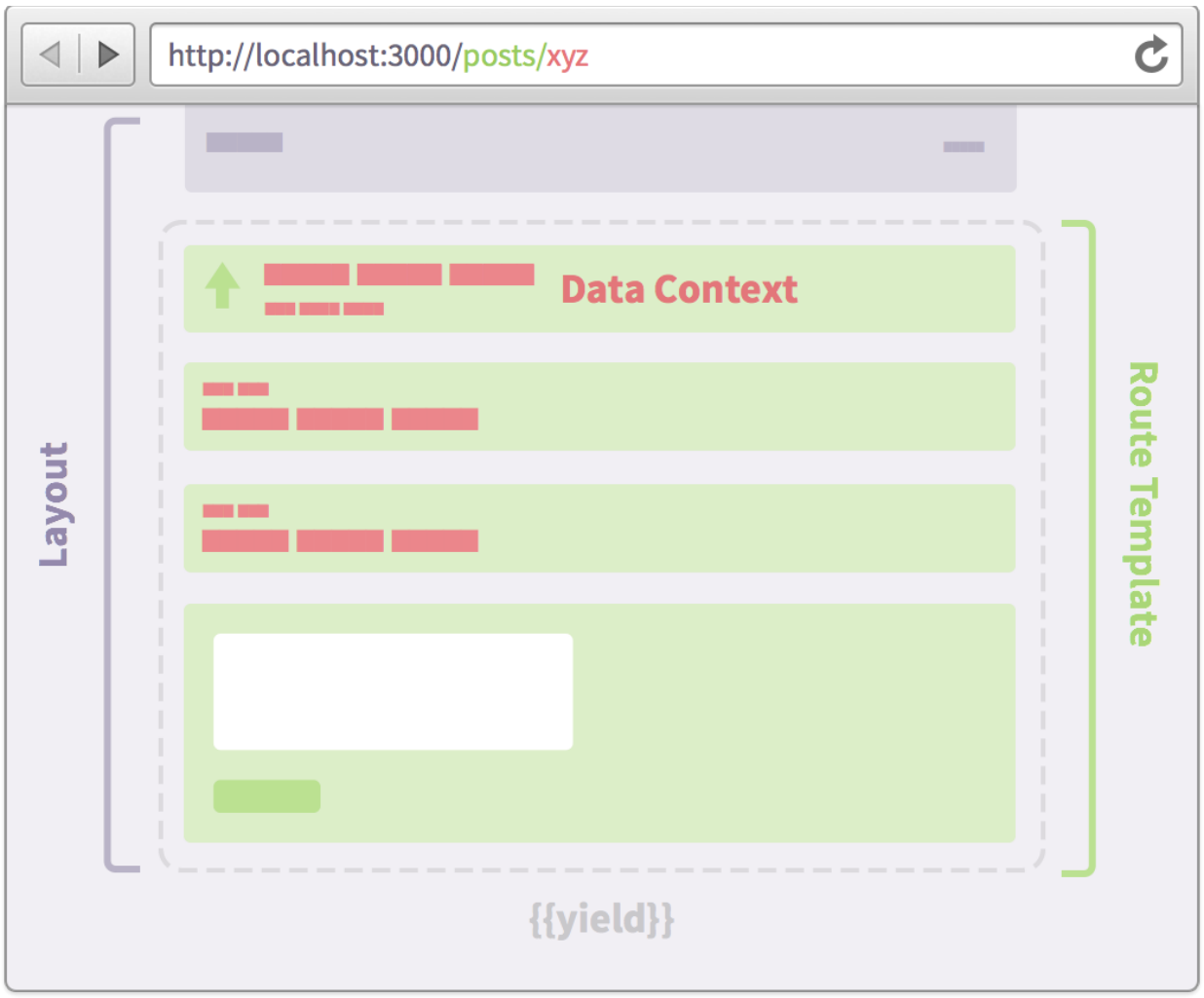
lib/router.js

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 `<a href="/posts/{{_id}}">`, 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`.

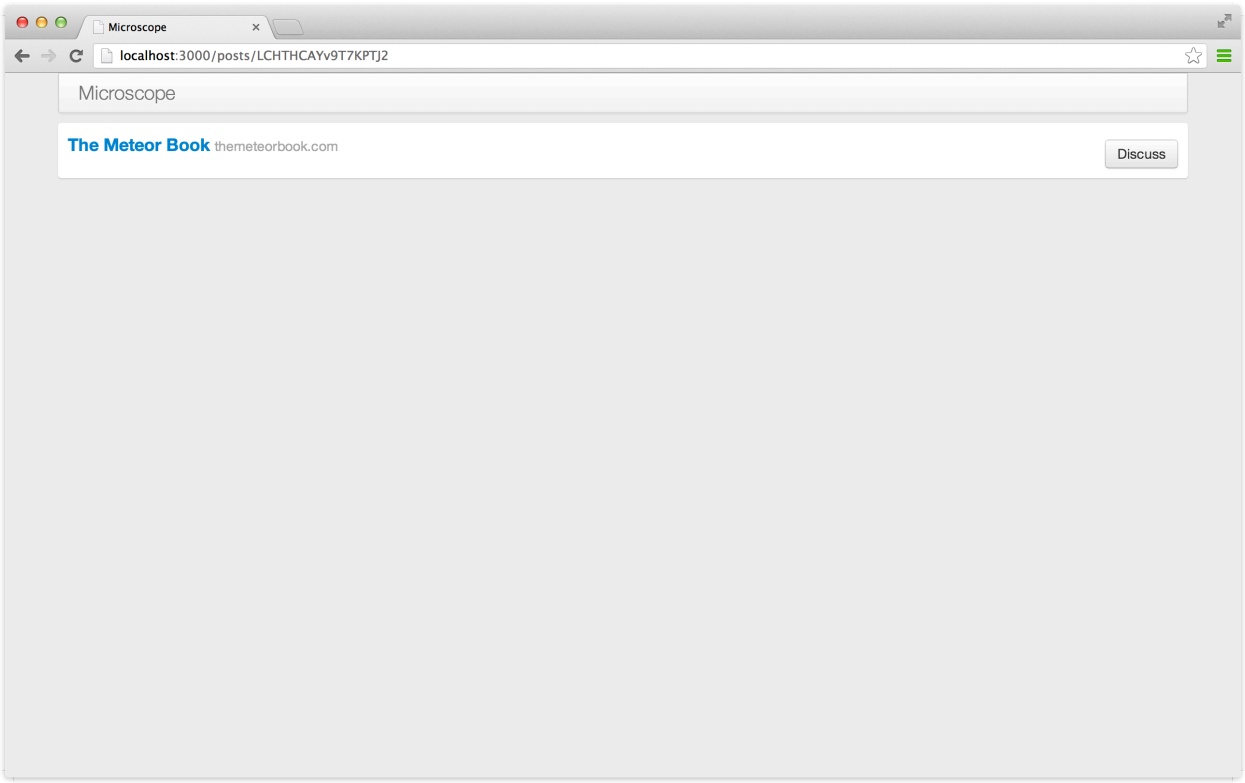
It turns out that Iron Router is smart enough to figure it out by itself. We're telling the router to use the `postPage` route, and the router knows that this route requires an `_id` of some kind (since that's how we defined our `path`).

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 realise 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 way: 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 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 tell 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.

The Session SIDEBAR

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('pageTit
le')); } );
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 Bootstrap classes to make sure everything looks nice:

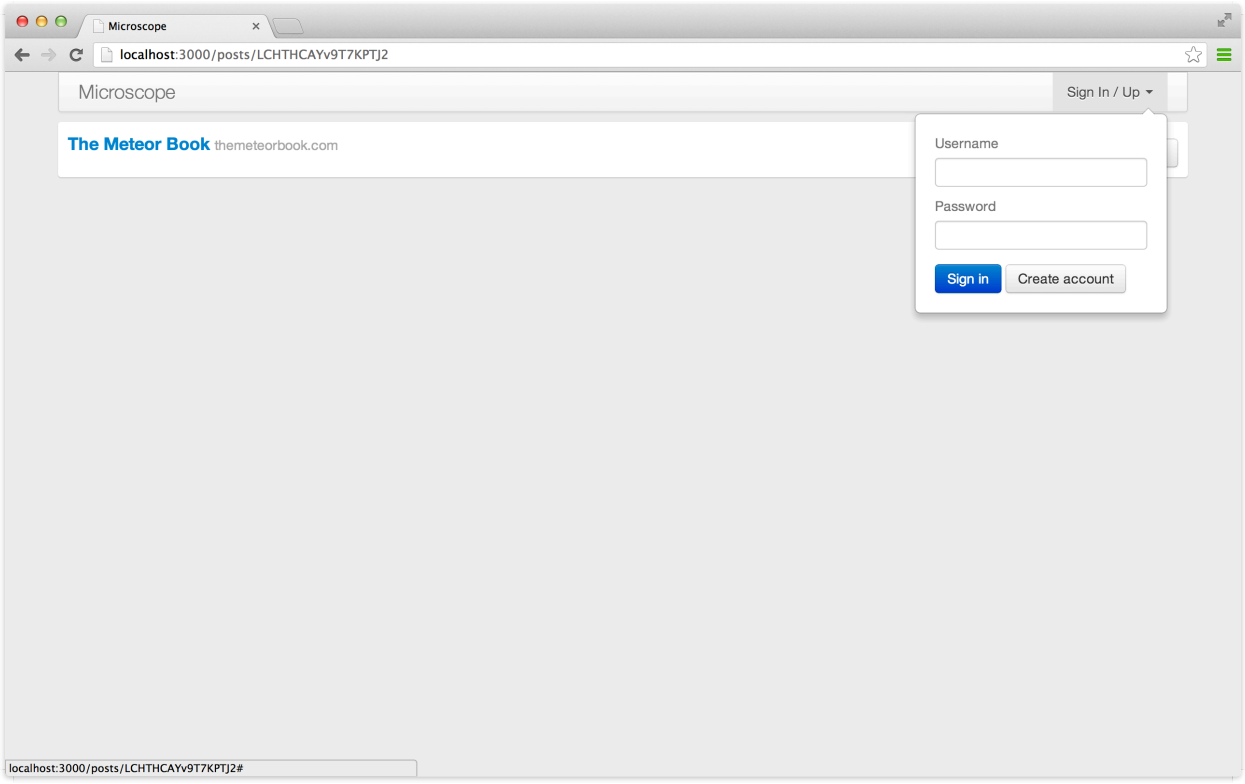
```
<template name="layout">
  <div class="container">
    {{> header}}
    <div id="main" class="row-fluid">
      {{> yield}}
    </div>
  </div>
</template>
```

client/templates/application/layout.html

```
<template name="header">
  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-toggle="coll
apse" 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>
    </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.findOne()
{
    "createdAt" : 1365649830922,
    "_id" : "kYdBd9hr3fWPGPcii",
    "services" : {
        "password" : {
            "srp" : {
                "identity" : "qyFCnw4MmRbmGyBdN",
                "salt" : "YcBjRa7ArXn5tdCdE",
                "verifier" : "df2c001edadf4e475e703fa8cd093abd4b63afccbca48fad1
d2a0986ff2bcfba920d3f122d358c4af0c287f8eaf9690a2c7e376d701ab2fe1acd53a5bc3e8439
05d5dcaf2f1c47c25bf5dd87764d1f58c8c01e4539872a9765d2b27c700dcdedadf5ac825214673
56d3f91dbeaf9848158987c6d359c5423e6b9cabf34fa0b45"
            }
        },
        "resume" : {
            "loginTokens" : [
                {
                    "token" : "BMHipQqjfLoPz7gru",
                    "when" : 1365649830922
                }
            ]
        }
    },
    "username" : "tmeasday"
}
```

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 `<li>`. 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="container-fluid">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-toggle="coll
apse" 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>
    </div>
  </nav>
</template>
```

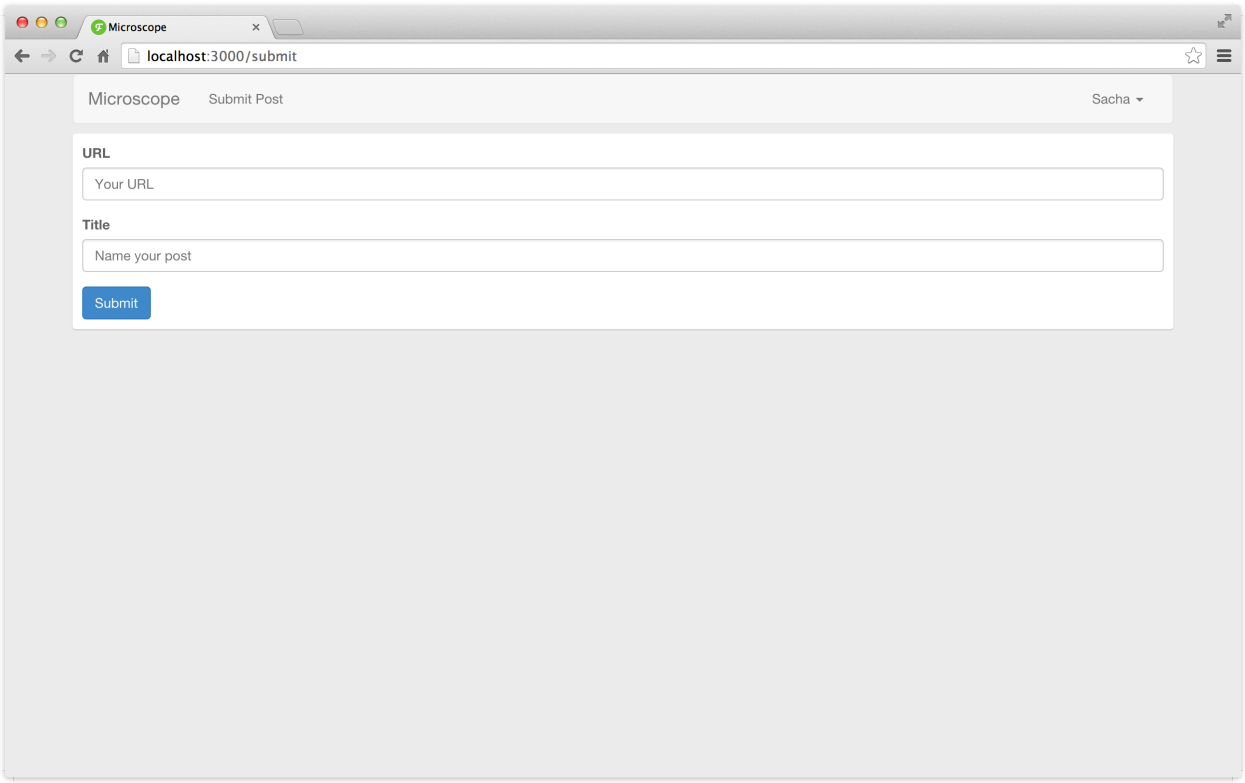
client/templates/includes/header.html

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">
    <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:



The post submit form

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

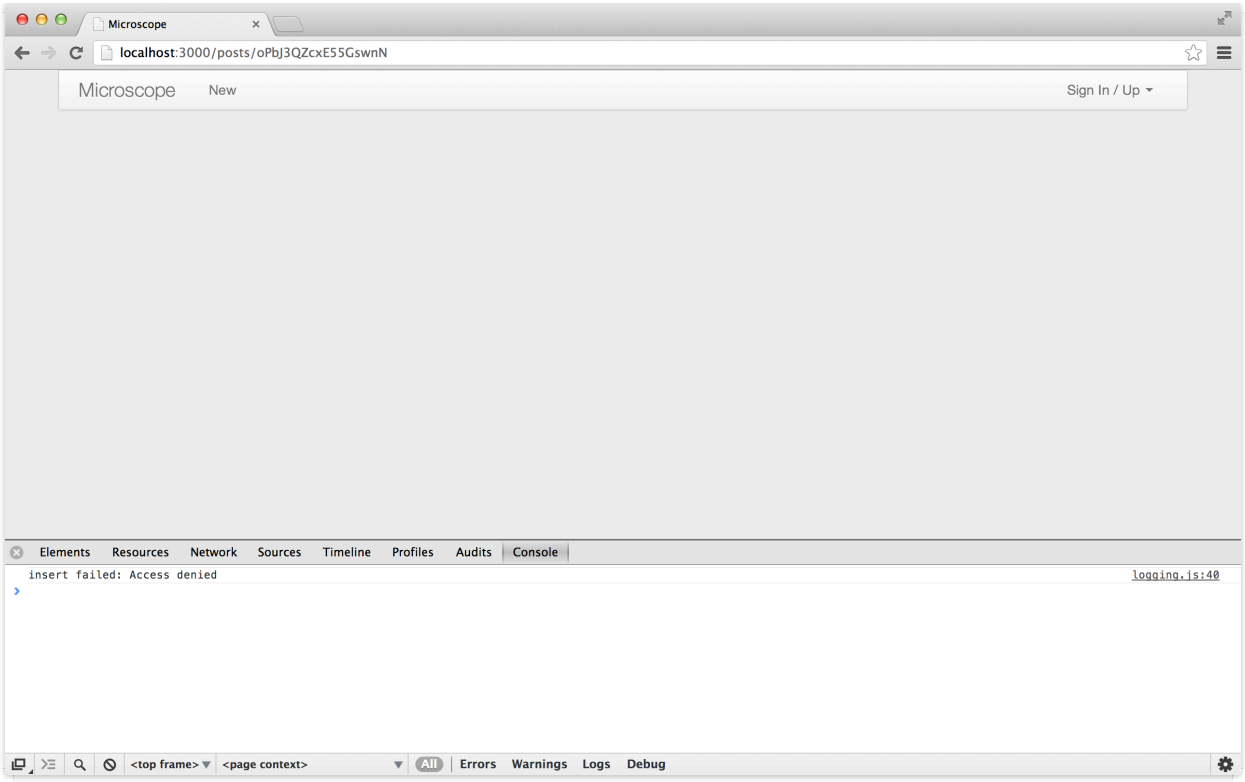
Commit 7-2
Removed insecure, and allowed certain writes to posts.

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 jumbotron">
    <h2>Access Denied</h2>
    <p>You can't get here! Please log in.</p>
  </div>
</template>
```

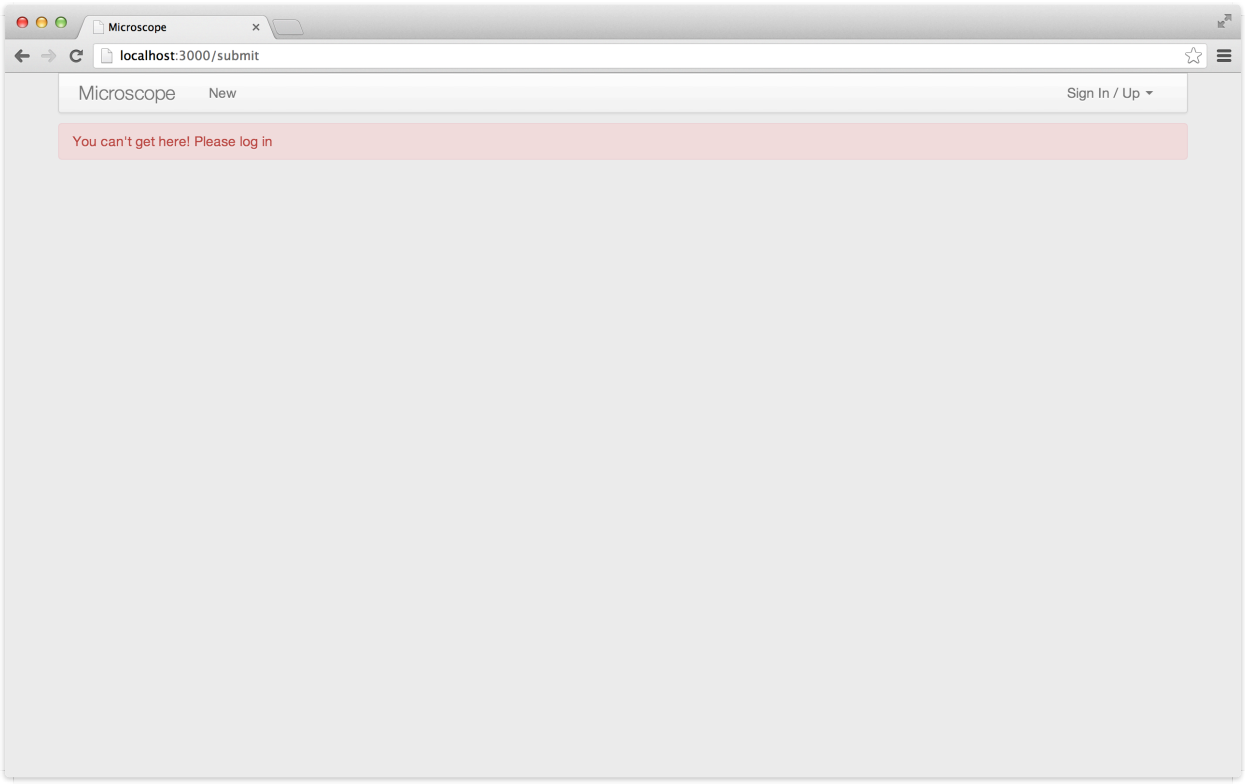
client/templates/includes/access_denied.html

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 this:



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
>{{/if}}
</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's working as it should, we'll redirect the user to the freshly created post's discussion page.

Security Check

We'll take advantage of this opportunity to add some security to our method by using the `audit-argument-checks` package.

This package lets you check any JavaScript object against a predefined pattern. In our case, we'll use it 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 `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, before inserting the whole thing in our database and returning 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
    };
  }
});
```

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
    };
  }
});
```

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

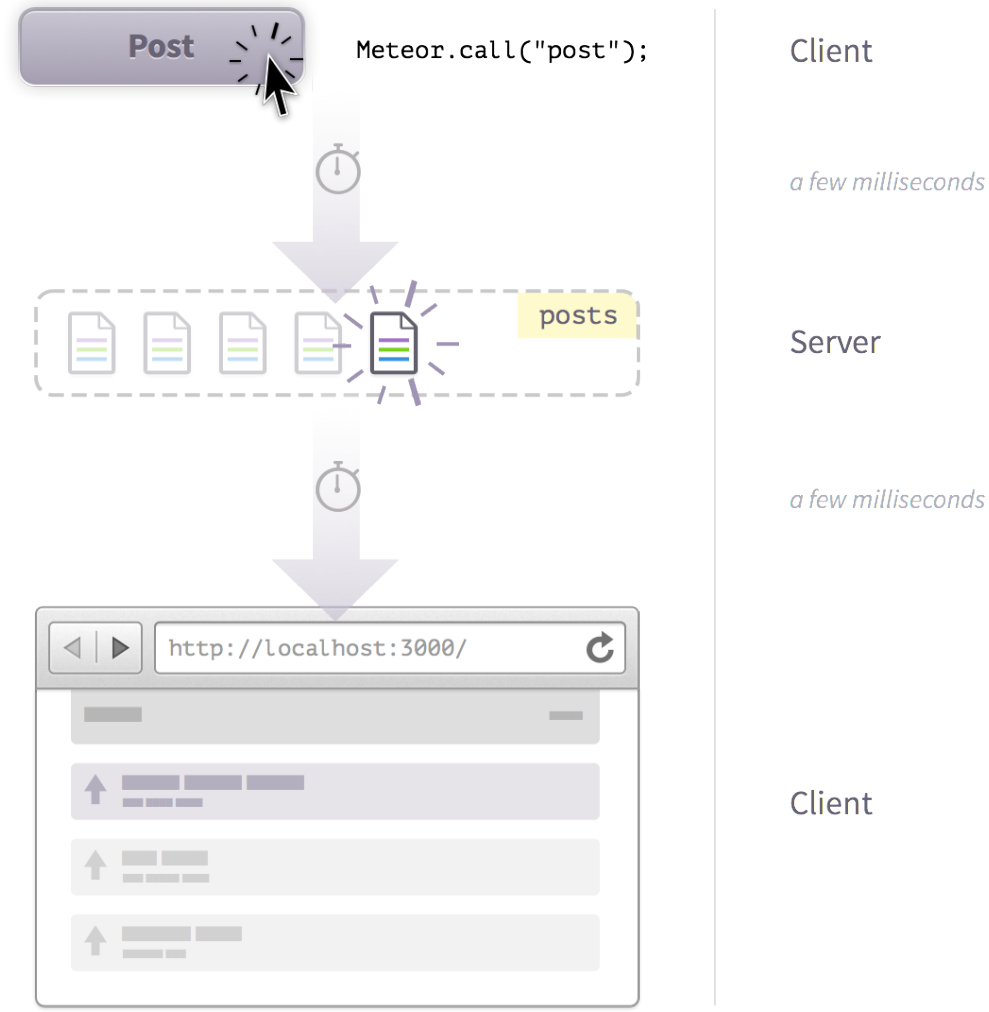
Sort posts by submitted timestamp.

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.

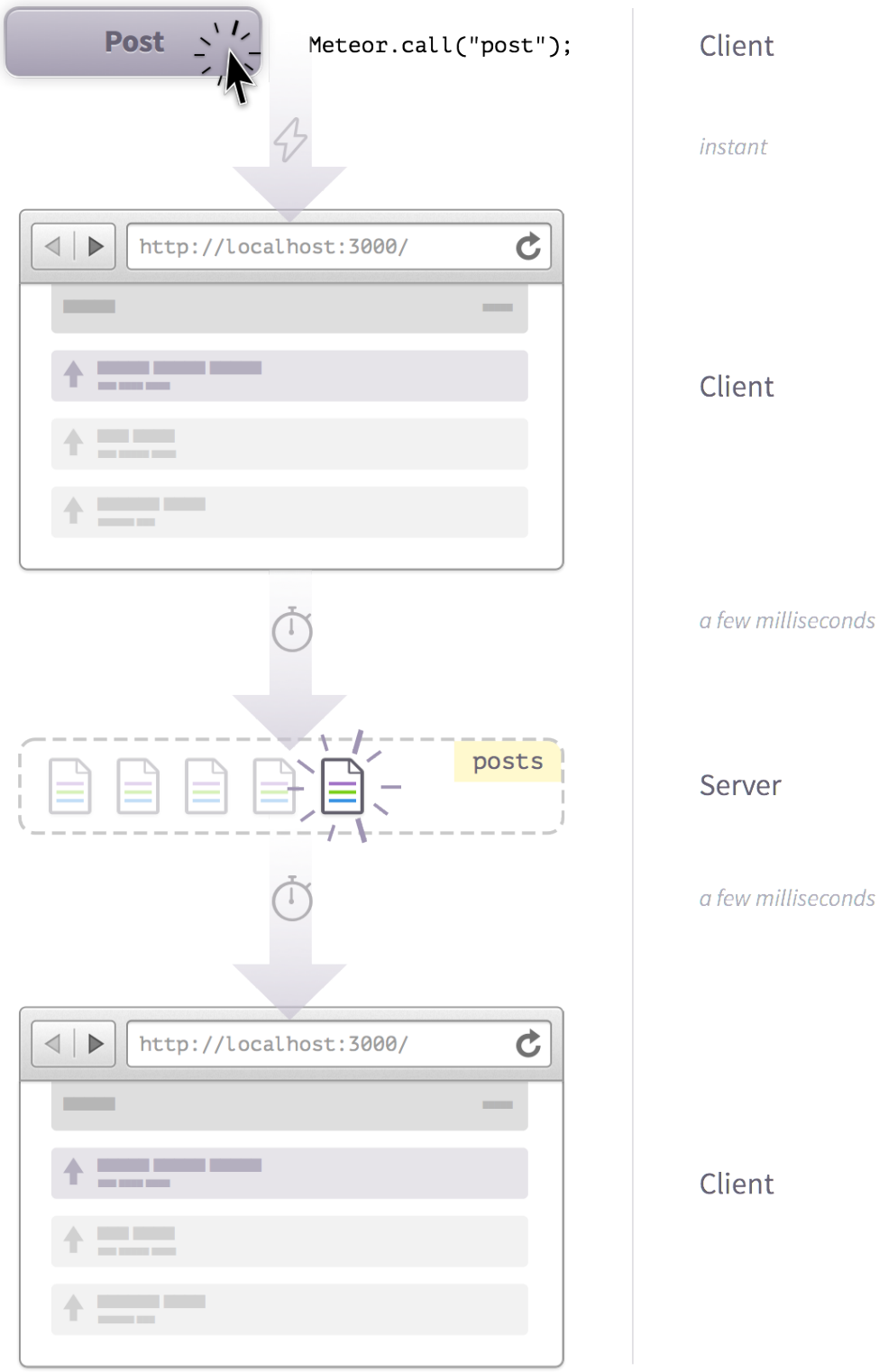
However, if Meteor executed Methods in the most basic way, we'd have a problem. Consider the following sequence of events (note: the timestamps are random values picked for illustrative purpose only):

- *+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



With 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');

  }
});
```

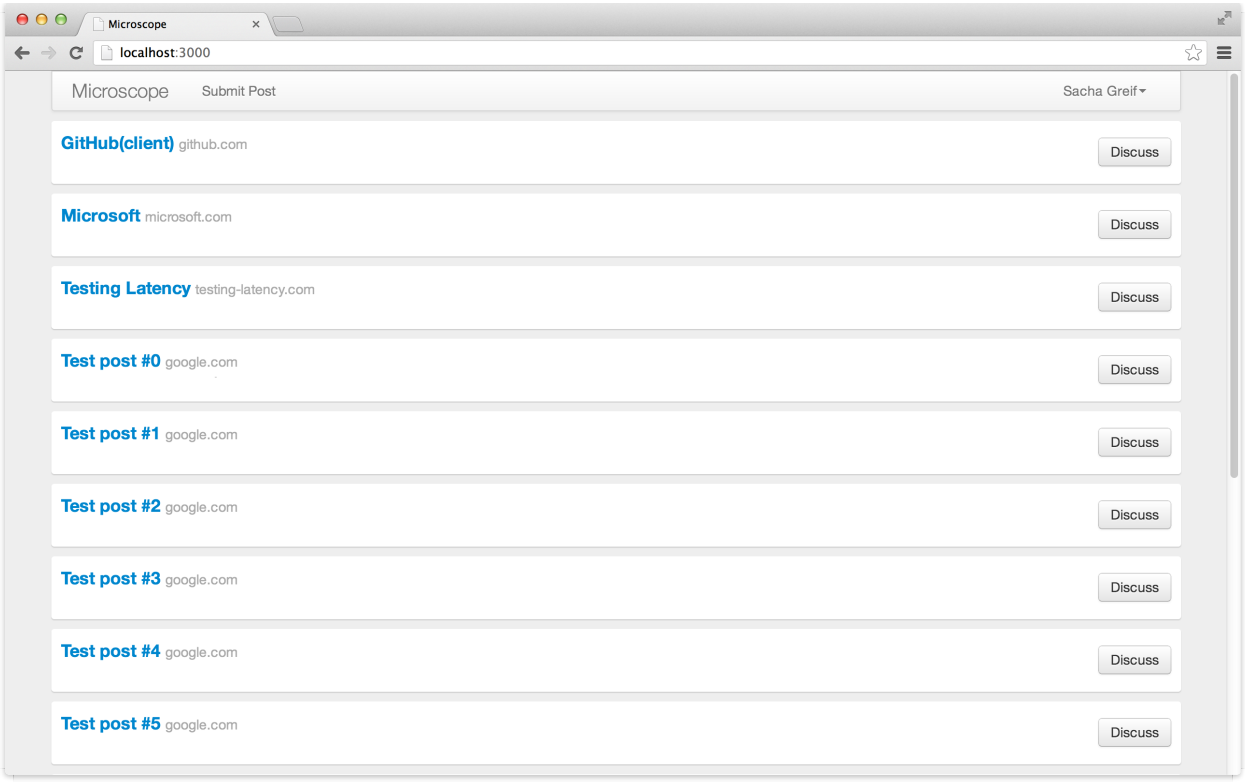
client/templates/posts/post_submit.js

Commit 7-5-1

Demonstrate the order that posts appear using a sleep.

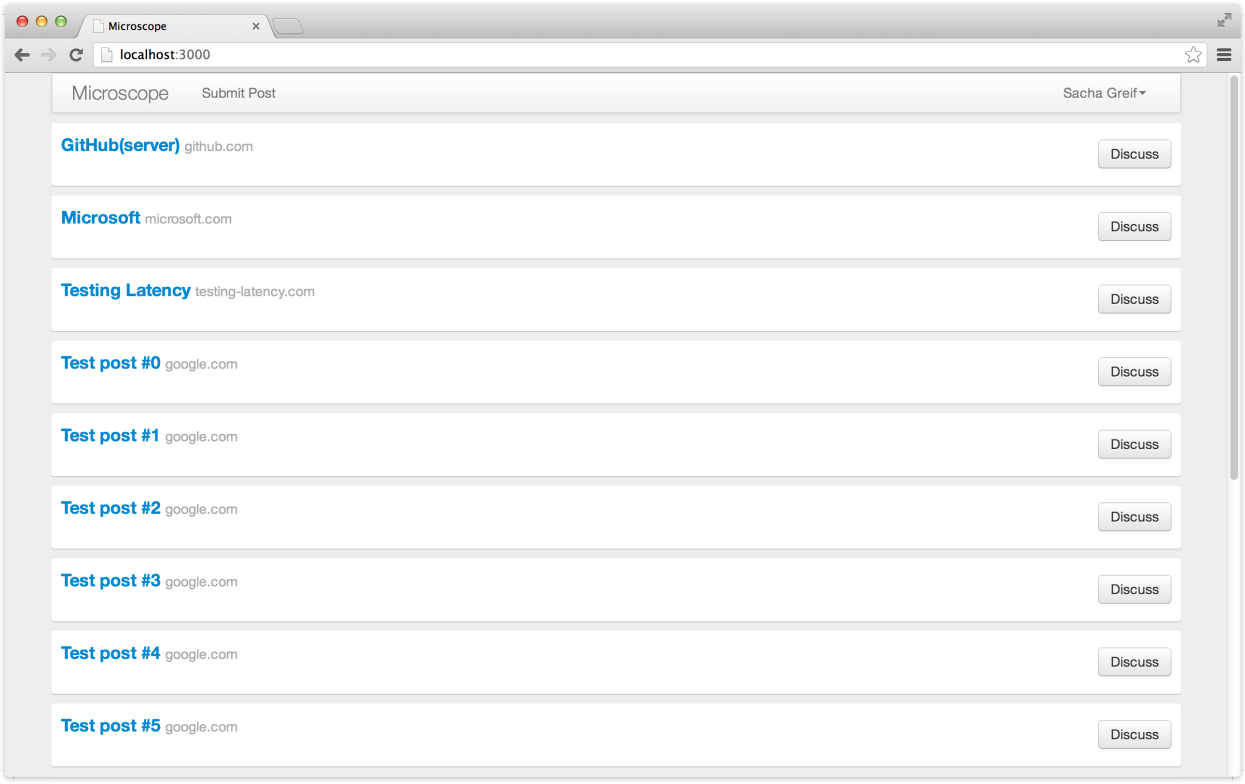
View on GitHub | Launch Instance

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'});
```

lib/router.js

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">
    <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) {
    e.preventDefault();

    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:

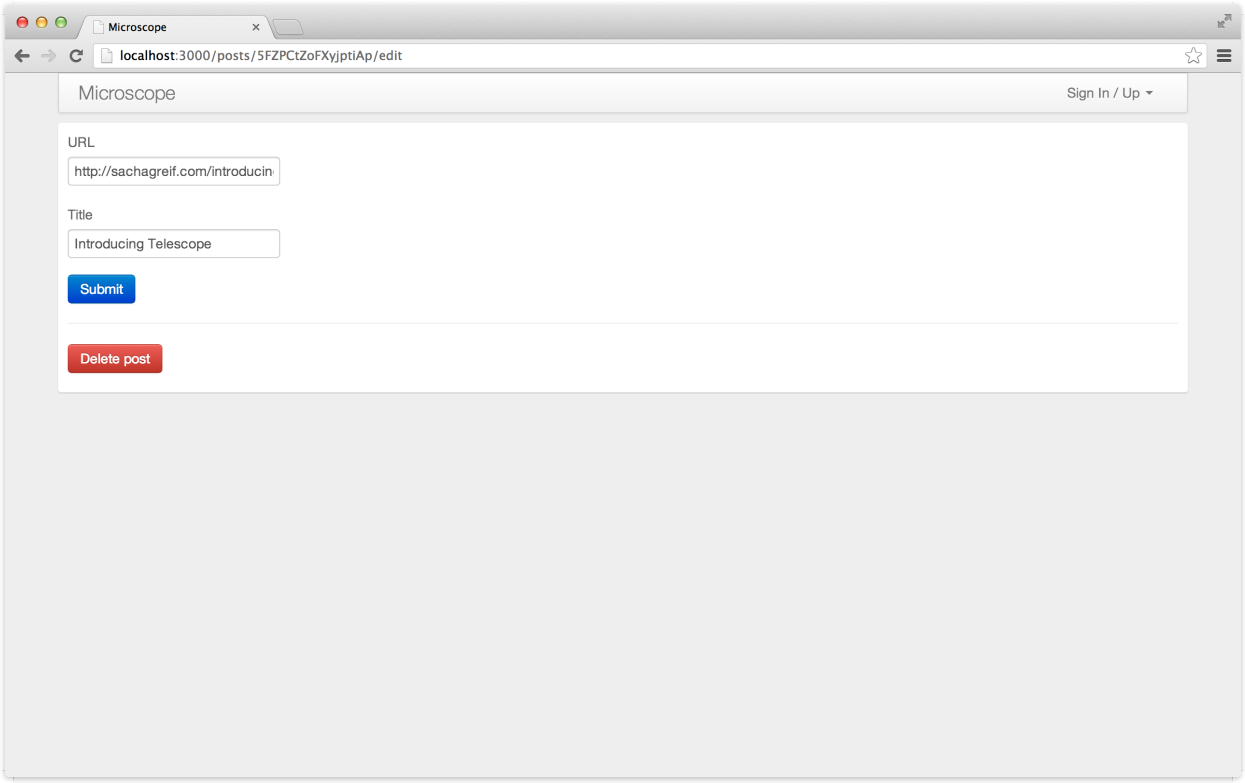
```
<template name="postItem">
  <div class="post">
    <div class="post-content">
      <h3><a href="{{url}}">{{title}}</a><span>{{domain}}</span></h3>
      <p>
        submitted by {{author}}
        {{#if ownPost}}<a href="{{pathFor 'postEdit'}}">Edit</a>{{/if}}
      </p>
    </div>
    <a href="{{pathFor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

//...
```

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: ownsDocument,
  remove: ownsDocument
});

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

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`.

When is it appropriate to do one and not the other?

When things are relatively straightforward and you can adequately express your rules via `allow` and `deny`, it's usually simpler to do things directly from the client.

However, as soon as 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), it's probably better to use a Method.

Method calls are also more appropriate 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.

Congratulations!

You've wrapped up the Starter Edition of *Discover Meteor*.

Get the full book to learn about:

- Managing user accounts.
- Making your app secure.
- Publishing and subscribing to data.
- Submitting, editing, and voting on posts.
- And much more.

You'll be up and running in no time!

> *"This book is a magnificent complement to the Meteor documentation, and a great option for anyone wanting to learn Meteor."* – ***Meteor co-founder Matt Debergalis***