Fabian Vogelsteller, Isaac Strack
Marcelo Reyna

Meteor: Full-Stack Web Application Development

Learning Path

Learn how to create mobile and full-stack web applications in JavaScript by getting a deeper insight into Meteor

Packt

Meteor: Full-Stack Web Application Development

Learn how to create mobile and full-stack web applications in JavaScript by getting a deeper insight into Meteor

A course in three modules

BIRMINGHAM - MUMBAI

Meteor: Full-Stack Web Application Development

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: November 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-78728-775-4

www.packtpub.com

Credits

Authors

Fabian Vogelsteller

Isaac Strack

Marcelo Reyna

Reviewers

Riccardo Mancinelli

Rohit Mukherjee

Isaac Strack

Jebin B V

Ryan Wilcox

Brad Cypert

David Ryan Speer

Content Development Editor

Onkar Wani

Graphics

Abhinash Sahu

Production Coordinator

Shraddha Falebhai

Preface

Meteor is the weapon of choice for start-ups in today's world as it's an incredibly powerful and fun platform that allows you to build real-time web applications quickly and easily and in pure JavaScript. It provides developers with tools and flexibility to build a fully-fledged production app.

Meteor aims to create single-page applications where real time is the default. It takes care of the data synchronization and updating of the DOM. If data changes, your screen will be updated. These two basic concepts make up a lot of the work we do as web developers, and with Meteor this happens without any extra line of code.

It is modular and declarative, supports data-on-the-wire, is well supported by a thriving development community, and implements full-stack reactivity to create a great impact on your day-to-day development efforts.

As Meteor is modular and well supported, it works easily with all of your favorite JavaScript frameworks. You can use the entire Meteor stack, or you can mix and match it with community packages to complement your existing infrastructure / skill set.

What this learning path covers

Module 1, Building Single-page Web Apps with Meteor, This module takes you from the installation of Meteor to building a fully working web blog. You will start with the basic concepts and folder structure of a Meteor project, learning how Meteor templates work. Learn how to retrieve and send data to the server and manipulate the database content. Routing will later make your example app look and behave like a real website. Also, you'll get to grips with Meteor's reactivity concept that can rerun functions when data changes while you're building your own reactive object, and package it later for drop-in use. After your app is ready, you will learn about ways of deploying your app on different types of servers. Finally, we will take a look at testing packages and the applications.

Module 2, Meteor Cookbook, This module starts with simple recipes designed for quick reference, and culminating in advanced recipes that walk you through building and deploying a complete application, it covers all the major areas of Meteor development, including lesser-known and undocumented features. You will have instant access to step-by-step recipes, accompanied by clear, concise explanations on optimizing your workflow.

Module 3, Meteor Design Patterns, Module starts off with a refresher on the basics of JavaScript programming such as templates, CoffeeScript, the Event Loop, and the Merge Box, amongst others. You then learn how to map real-world data and optimize the data's publishers to output data with the least amount of work done by the server with some subscribe and publish patterns. Following this, you will see how to optimize and secure the web application and maintain applications without breaking other features. Finally, you will learn how to deploy a secure production-ready application while learning to set up modulus, compose with Oplog tracking and SSL certificates, as well as error tracking with Kadira.

What you need for this learning path

Module 1:

To follow the examples in the modules, you will need a text editor to write the code. It's highly recommended to use Sublime Text as your IDE, as it has a wide range of plugins for almost every task a web developer could think of.

You will also need a modern browser to see your results. As many examples use the browser console to make changes to the database and to see the results of the code snippets, I recommend Google Chrome. Its Developer tools web inspector has everything a web developer needs to work and debug websites with ease.

Additionally, you can use Git and GitHub to store your success every step along the way and in order to go back to the previous versions of your code.

The code examples for each chapter will also be available on GitHub at

`https://github.com/frozeman/book-building-single-page-web-apps-with-meteor`, where each commit in this repository correlates with one chapter of the module, giving you an easy way to see what was added and removed in each step along the way

Module 2:

This module assumes that you have a working knowledge of JavaScript and HTML. Being familiar with Node, npm, GitHub and the command line/terminal will be very helpful (but not critical) to getting the most out of the recipes in this book.

You will find recipes to install Meteor on Mac OS X or Linux, with links to using Meteor on Windows and Google Chromebooks. In every instance, you will need access to the Internet to download Meteor and community packages, and you will need installation privileges for your developer machine regardless of the operating system.

For deployment to production environments or to mobile devices, the requirements will vary from recipe to recipe. To complete all of the recipes successfully, you will need your own hosted server and DNS domain as well as iOS, Android, or Windows mobile devices and SDKs.

Module 3:

- Meteor version 1.1.0.2 or above
- A Unix system such as a Mac or Linux computer

Who this learning path is for

If you are a web developer who is familiar with Meteor and has basic knowledge of web development, and you now want to explore new paradigms of single-page, real-time applications, this course is perfectly suited for you.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/Meteor-Full-Stack-Web-Application-Development`. We also have other code bundles from our rich catalog of books, videos, and courses available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.-

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1: Building Single-page Web Apps with Meteor

Chapter 1: Getting Started with Meteor 3

The full-stack framework of Meteor 4
Meteor's requirements 5
Installing Meteor 6
Creating our first app 7
Adding basic packages 9
Variable scopes 10
Meteor's folder conventions and loading order 11
Meteor's command-line tool 14
Summary 15

Chapter 2: Building HTML Templates 17

Writing templates in Meteor 18
Building the basic templates 20
Adding templates and partials 21
Displaying data with template helpers 22
Setting the data context for a template 24
"this" in template helpers and template callbacks 25
Adding events 27
Block helpers 28
Listing posts 30
Spacebars syntax 34
Summary 37

Chapter 3: Storing Data and Handling Collections 39

Meteor and databases 40
Setting up a collection 41
Adding post examples 41
Querying a collection 44

Updating a collection 45
Database everywhere 46
Differences between client and server collections 47
Summary 47

Chapter 4: Controlling the Data Flow 49
Syncing data – the current Web versus the new Web 50
Removing the autopublish package 51
Publishing data 51
Publishing only parts of data 52
Publishing specific fields 54
Lazy loading posts 56
Switching subscriptions 58
Some notes on data publishing 60
Summary 61

Chapter 5: Making Our App Versatile with Routing 63
Adding the iron:router package 64
Setting up the router 64
Switching to a layout template 66
Adding another route 67
Moving the posts subscription to the Home route 68
Setting up the post route 70
Changing the website's title 74
Summary 75

Chapter 6: Keeping States with Sessions 77
Meteor's session object 77
Using sessions in template helpers 79
Rerunning functions reactively 82
Using autorun in a template 84
The reactive session object 86
Summary 86

Chapter 7: Users and Permissions 89
Meteor's accounts packages 90
Adding the accounts packages 90
Adding admin functionality to our templates 91
Creating the template to edit posts 93
Creating the admin user 94
Creating routes for the admin 97
Summary 100

Chapter 8: Security with the Allow and Deny Rules 101
Adding a function to generate slugs 102
Creating a new post 102
Editing posts 104
Restricting database updates 105
Adding a deny rule 107
Adding posts using a method call 109
Calling the method 112
Summary 113

Chapter 9: Advanced Reactivity 115
Reactive programming 116
Building a simple reactive object 120
Creating an advanced timer object 122
Reactive computations 124
Summary 128

Chapter 10: Deploying Our App 129
Deploying on meteor.com 130
Deploying on other servers 134
Outlook 140
Summary 140

Chapter 11: Building Our Own Package 141
The structure of a package 141
Creating our own package 143
Releasing our package to the public 146
Summary 150

Chapter 12: Testing in Meteor 151
Types of tests 151
Testing packages 152
Testing our meteor app 156
Acceptance tests 165
Summary 166

Module 2: Meteor Cookbook

Chapter 1: Optimizing Your Workflow 1
Introduction 1
Installing Meteor 2
Finding documentation for Meteor 3

Getting help with questions 4
Setting up your project file structure 7
Setting up your development environment 11
Using the web console 14
Deploying a test app to Meteor 19
Deploying to Meteor using a CNAME redirect 23
Deploying to a custom hosted environment 25
Deploying with Meteor Up (MUP) 30
Using CoffeeScript 33
Using CSS compilers 35

Chapter 2: Customizing with Packages 39
Introduction 39
Adding Meteor packages 40
Removing Meteor packages 41
Discovering new packages with Atmosphere 42
Creating a multipage application with Iron Router 46
Building a custom package 51
Using npm modules 55
Publishing custom packages to Atmosphere 57

Chapter 3: Building Great User Interfaces 61
Introduction 61
Inserting templates with Spacebars 62
Inserting raw HTML using triple braces 64
Creating dynamic lists 66
Building a smooth interface with Bootstrap 71
Creating customized global helpers 74
Creating custom components 77
Using reactivity with HTML attributes 80
Using dynamic templates 84
Animating DOM elements 87

Chapter 4: Creating Models 93
Introduction 93
Implementing a simple collection 93
Using the Session object 96
Sorting with MongoDB queries 99
Filtering with MongoDB queries 101
Creating upsert MongoDB queries 103
Implementing a partial collection 106

Chapter 5: Implementing DDP 111
Introduction 111
Reading the DDP stream 111
Using client-only collections 114
Implementing multiserver DDP 125
Integrating DDP with other technologies 128

Chapter 6: Mastering Reactivity 133
Introduction 133
Creating and consuming a reactive value 134
Using Ajax query results in ReactiveVar 139
Making a custom library reactive 145
Updating Blaze templates without Mongo 149
Using inline data to modify UI elements reactively 152
Integrating a jQuery UI 157

Chapter 7: Using Client Methods 165
Introduction 165
Creating dynamic graphs with SVG and Ajax 165
Using the HTML FileReader to upload images 177
Creating a coloring book with the Canvas element 189

Chapter 8: Integrating Third-party Libraries 207
Introduction 207
Using npm packages directly 208
Building graphs with D3.js 213
Creating cutting-edge UIs with Polymer 221

Chapter 9: Securing Your Application 235
Introduction 235
Basic safety – turning off autopublish 236
Basic safety – removing insecure 241
Securing data transactions with allow and deny 244
Hiding data with façades 247
Protecting the client with browser-policy 249

Chapter 10: Working with Accounts 257
Introduction 257
Implementing OAuth accounts packages 257
Customizing the accounts login 264
Performing two-factor authentication 267

Chapter 11: Leveraging Advanced Features 281
Introduction 281
Building custom server methods 281

Creating custom EJSON objects 287
Handling asynchronous events 291
Using asynchronous functions 297

Chapter 12: Creating Useful Projects 301
Introduction 301
Creating RESTful web services 301
Creating a complete app with Iron Router 307
Deploying apps to mobile devices 324
Adding social sharing 330

Module 3: Meteor Design Patterns

Chapter 1: Getting Started with Meteor 1
CoffeeScript for Meteor 2
Jade for Meteor 7
Stylus for Meteor 11
Templates, helpers, and events 15
The event loop and the merge box 18
The beginning of our online shop 20
Summary 25

Chapter 2: Publish and Subscribe Patterns 27
Template-level subscriptions 28
Database relationships 30
Publishing with relations 35
Aggregation publishers 41
External API publishers 46
Summary 48

Chapter 3: Front-end Patterns 49
Responsive design 49
Super helpers 57
Variable types 59
Forms 72
Loading data 80
Animations and transitions 81
SEO 86
Summary 90

Chapter 4: Application Patterns 91
- Filtering and paging collections 91
- Security 99
- External APIs 118
- Summary 125

Chapter 5: Testing Patterns 127
- Behavior tests 127
- Unit tests 135
- Summary 137

Chapter 6: Deployment 139
- Setting up Modulus 139
- Setting up Compose 144
- Setting up Kadira 147
- Setting up an SSL certificate 150
- Summary 156

Bibliography 157

Module 1

Building Single-page Web Apps with Meteor

Build real-time apps at lightning speed using the most powerful full-stack JavaScript framework

1
Getting Started with Meteor

Welcome to this book on Meteor. Meteor is an exciting new JavaScript framework, and we will soon see how easy it is to achieve real and impressive results with less code.

In this chapter, we will learn what the requirements are and what additional tools we need to get started. We will see how simple it is to get our first Meteor application running and what a good basic folder structure for a Meteor app could be. We will also learn about Meteor's automatic build process and its specific way of loading files.

We will also see how to add packages using Meteors official packaging system. At the end of the chapter, we will take a short look at Meteor's command-line tool and some of its functions.

To bring it together, we will cover the following topics:

- The full-stack framework of Meteor
- Meteor's requirements
- Installing Meteor
- Adding basic packages
- Meteor's folder conventions and loading order
- Meteor's command-line tool

The full-stack framework of Meteor

Meteor is not just a JavaScript library such as jQuery or AngularJS. It's a full-stack solution that contain frontend libraries, a Node.js-based server, and a command-line tool. All this together lets us write large-scale web applications in JavaScript, on both the server and client, using a consistent API.

Even with Meteor being quite young, already a few companies such as `https://lookback.io`, `https://respond.ly`, and `https://madeye.io` use Meteor in their production environment.

If you want to see for yourself what's made with Meteor, take a look at `http://madewith.meteor.com`.

Meteor makes it easy for us to build web applications quickly and takes care of the boring processes such as file linking, minifying, and concatenating of files.

Here are a few highlights of what is possible with Meteor:

- We can build complex web applications amazingly fast using templates that automatically update themselves when data changes
- We can push new code to all clients on the fly while they are using our app
- Meteor core packages come with a complete account solution, allowing a seamless integration of Facebook, Twitter, and more
- Data will automatically be synced across clients, keeping every client in the same state in almost real time
- Latency compensation will make our interface appear super fast while the server response happens in the background.

With Meteor, we never have to link files with the `<script>` tags in HTML. Meteor's command-line tool automatically collects JavaScript or CSS files in our application's folder and links them in the `index.html` file, which is served to clients on initial page load. This makes structuring our code in separate files as easy as creating them.

Meteor's command-line tool also watches all files inside our application's folder for changes and rebuilds them on the fly when they change.

Additionally, it starts a Meteor server that serves the app's files to the clients. When a file changes, Meteor reloads the site of every client while preserving its state. This is called a **hot code reload**.

In production, the build process also concatenates and minifies our CSS and JavaScript files.

By simply adding the `less` and `coffee` core packages, we can even write all styles in LESS and code in CoffeeScript with no extra effort.

The command-line tool is also the tool for deploying and bundling our app so that we can run it on a remote server.

Sounds awesome? Let's take a look at what's needed to use Meteor.

Meteor's requirements

Meteor is not just a JavaScript framework and server. As we saw earlier, it is also a command-line tool that has a whole build process for us in place.

Currently, the operating systems that are officially supported are as follows:

- Mac OS X 10.6 and above
- Linux x86 and x86_64 systems
- Windows

> The Windows installer is still in development at the time of writing this book. Please follow the wiki page at `https://github.com/meteor/meteor/wiki/Preview-of-Meteor-on-Windows`.

This book and all examples use *Meteor 1.0*.

Using Chrome's developer tools

We will also need Google Chrome or Firefox with the Firebug add-on installed to follow examples that require a console. The examples, screenshots, and explanations in this book will use Google Chrome's developer tools.

Using Git and GitHub

I highly recommend using **GitHub** when working with web projects, such as the one we will work on in this book. Git and GitHub help us to back up our progress and let us always go back to previous states while seeing what we've changed.

Git is a version control system, which was created in 2005 by the inventor of Linux, Linus Torvalds.

With Git, we can *commit* any state of our code and later go back to that exact state. It also allows multiple developers to work on the same code base and merge their results together in an automated process. If conflicts appear in this process, the merging developer is able to resolve those *merge conflicts* by removing the unwanted lines of code.

I also recommend registering an account at `http://github.com`, as this is the easiest way to browse our code history. They have an easy to use interface as well as a great Windows and Mac app.

To follow the code examples in this book, you can download all code examples for each chapter from the book's web page at `https://www.packtpub.com/books/content/support/17713`.

Additionally, you will be able to clone the book's code from `http://github.com/frozeman/book-building-single-page-web-apps-with-meteor`. Every tag in this repository equals to one chapter of the book and the commit history will help you to see the changes, which were made in each chapter.

Installing Meteor

Installing Meteor is as easy as running the following command in the terminal:

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

That's it! This will install the Meteor command-line tool (`$ meteor`), the Meteor server, MongoDB database, and the Meteor core packages (libraries).

> All command-line examples are run and tested on Mac OS X and can differ on Linux or Windows systems.

Installing Git

To install Git, I recommend installing the GitHub app from `https://mac.github.com` or `https://windows.github.com`. We can then simply go inside the app to **Preferences** and click on the **Install Command Line Tools** button inside the **Advanced** tab.

If we want to install Git manually and set it up via the command line, we can download the Git installer from `http://git-scm.com` and follow this great guide at `https://help.github.com/articles/set-up-git`.

Now, we can check whether everything was installed successfully by opening the terminal and running the following command:

```
$ git
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

This should return us a list of Git options. If we get `command not found: git`, we need to check whether the `git` binary was correctly added to our `PATH` environment variable.

If everything is fine, we are ready to create our first Meteor app.

Creating our first app

To create our first app, we open the terminal, go to the folder where we want to create our new project, and enter the following commands:

```
$ cd my/developer/folder
$ meteor create my-meteor-blog
```

Meteor will now create a folder named `my-meteor-blog`. The HTML, CSS, and JavaScript files that Meteor created for us inside this folder are already a fully working Meteor app. To see it in action, run the following commands:

```
$ cd my-meteor-blog
$ meteor
```

Meteor will now start a local server for us on port `3000`. Now, we can open our web browser and navigate to `http://localhost:3000`. We will see the app running.

This app doesn't do much, except showing a simple reactive example. If you click on the **Click Me** button, it will increase the counter:

For later examples, we will need Google Chrome's developer tools. To open the console, we can press *Alt* + *command* + *I* on Mac OS X or click on the menu button on the upper-right corner of Chrome, select **More tools,** and then **Developer tools**.

The **Developer tools** allow us to inspect the DOM and CSS of our website, as well as having a console where we can interact with our website's JavaScript.

Creating a good folder structure

For this book, we will build our own app from scratch. This also means we have to set up a sustainable folder structure, which helps us to keep our code organized.

With Meteor, we are very flexible concerning our folder structure. This means we can put our files wherever we want, as long as they are inside the app's folder. Meteor treats specific folders differently, allowing us to expose files only on the client, the server, or both. We will take a look at those specific folders later.

But, first let's get our hands dirty by deleting all preadd files in our newly created application folder and creating the following folder structure:

```
- my-meteor-blog
  - server
  - client
    - styles
    - templates
```

Preadd style files

To fully focus on the Meteor code but still have a pretty-looking blog, I strongly recommend to download the code that accompanies this chapter from the book's web page at `http://packtpub.com/books/content/support/17713`. They will contain already two drop-in-place style files (`lesshat.import.less` and `styles.less`), which will let your example blog look pretty in the upcoming chapters.

You can also download these files directly from GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter1/my-meteor-blog/client/styles` and copy them to the `my-meteor-blog/client/styles` folder manually.

Next, we need to add some basic packages so that we can start building our app.

Adding basic packages

Packages in Meteor are libraries that can be added to our projects. The nice thing about Meteor packages is that they are self-contained units, which run out of the box. They mostly add either some templating functionality or provide extra objects in the global namespace of our project.

Packages can also add features to Meteor's build process such as the `stylus` package, which lets us write our app's style files with the `stylus` preprocessor syntax.

For our blog, we will need two packages at first:

`less`: This is a Meteor core package and will compile our style files on the fly to CSS

`jeeeyul:moment-with-langs`: This is a third-party library for date parsing and formatting

Adding a core package

To add the `less` package, we can simply open the terminal, go to our projects folder, and enter the following command:

```
$ meteor add less
```

Now, we are able to use any `*.less` files in our project, and Meteor will automatically compile them in its build process for us.

Adding a third-party package

To add a third-party package, we can simply search for packages on either `https://atmospherejs.com`, which is the frontend for Meteors packaging system, or use the command-line tool, `$ meteor search <package name>`.

For our blog, we will need the `jeeeyul:moment-with-langs` package that allows us later to simply manipulate and format dates.

Packages are namespaced with the authors name followed by a colon.

To add the `moment` package, we simply enter the following command:

```
$ meteor add jeeeyul:moment-with-langs
```

After the process is done, and we restarted our app using `$ meteor`, we will have the `moment` object available in our app global namespace and we can make use of it in the upcoming chapters.

Should we ever want to add only specific version of a package, we can use the following command:

```
$ meteor add jeeeyul:moment-with-langs@=2.8.2
```

If you want a version in the 1.0.0 (but not the 2.0.0) range use the following command:

```
$ meteor add jeeeyul:moment-with-langs@1.0.0
```

To update only packages we can simply run the following command:

```
$ meteor update --packages-only
```

Additionally, we can update only a specific package using the following command:

```
$ meteor update jeeeyul:moment-with-langs
```

That's it! Now we are fully ready to start creating our first templates. You can jump right into the next chapter, but make sure you come back to read on, as we will now talk about Meteor's build process in more detail.

Variable scopes

To understand Meteor's build process and its folder conventions, we need to take a quick look at variable scopes.

Meteor wraps every code files in an anonymous function before serving it. Therefore, declaring a variable with the `var` keyword will make it only available in that file's scope, which means these variables can't be accessed in any other file of your app. However, when we declare a variable without this keyword, we make it a globally available variable, which means it can be accessed from any file in our app. To understand this, we can take a look at the following example:

```
// The following files content
var myLocalVariable = 'test';
myGlobalVariable = 'test';
```

After Meteor's build process, the preceding lines of code will be as follows:

```
(function(){
  var myLocalVariable = 'test';
  myGlobalVariable = 'test';
})();
```

This way, the variable created with *var* is a local variable of the anonymous function, while the other one can be accessed globally, as it could be created somewhere else before.

Meteor's folder conventions and loading order

Though Meteor doesn't impose restrictions concerning our folder names or structure, there are naming conventions that help Meteor's build process to determine the order in which the files need to be loaded.

The following table describes the folder and their specific loading order:

Folder name	Load behavior
`client`	This is loaded only on the client.
`client/compatibility`	This will not be wrapped in an anonymous function. This is made for libraries that declare top-level variables with `var`. Additionally, files in this folder will be loaded before other files on the client.
`server`	Files in this folder will only be served on the server.

Folder name	Load behavior
`public`	This folder can contain assets used on the client, such as images, `favicon.ico`, or `robots.txt`. Folders and files inside the public folder are available on the client from root, `/`.
`private`	This folder can contain assets that will only be available on the server. These files are available through `Assets` API.
`lib`	Files and subfolders inside a `lib` folder will be loaded before other files, where `lib` folders in deeper folders will be loaded before the files in `lib` folders of their parent folders.
`tests`	Files inside this folder won't be touched or loaded by Meteor at all.
`packages`	When we want to use local packages, we can add them to this folder and Meteor will use those packages, even if one with the same name exists in Meteor's official package system. (However, we still have to add the packages using `$ meteor add ....`)

The following table describes filenames that have created a specific loading order:

Filename	Load behavior
`main.*`	Files with this name are loaded last, whereas files in deeper folders are loaded before the files of their parent folders
`*.*`	Files outside of the former mentioned folders in this table are loaded on both the client and server

So, we see that Meteor gathers all files except the ones inside `public`, `private`, and `tests`.

Additionally, files are always loaded in the alphabetical order, and files in subfolders are loaded before the ones in parent folders.

If we have files outside the `client` or `server` folder and want to determine where the code should be executed, we can use the following variables:

```
if(Meteor.isClient) {
  // Some code executed on the client
}

if(Meteor.isServer) {
```

```
  // Some code executed on the server.
}
```

We also see that code inside a `main.*` file is loaded last. To make sure a specific code only loads when all files are loaded and the DOM on the client is ready, we can use the Meteor's `startup()` function:

```
Meteor.startup(function(){
  /*
  This code runs on the client when the DOM is ready,
  and on the server when the server process is finished starting.
  */
});
```

Loading assets on the server

To load files from inside the `private` folder on the server, we can use the `Assets` API as follows:

```
Assets.getText(assetPath, [asyncCallback]);
// or
Assets.getBinary(assetPath, [asyncCallback])
```

Here, `assetPath` is a file path relative to the `private` folder, for example, `'subfolder/data.txt'`.

If we provide a callback function as the second parameter, the `Assets()` method will run asynchronously. So, we have two ways of retrieving the content of an assets file:

```
// Synchronously
var myData = Assets.getText('data.txt');

// Or asynchronously
Assets.getText('data.txt', function(error, result){
  // Do somthing with the result.
  // If the error parameter is not NULL, something went wrong
});
```

If the first example returns an error, our current server code will fail. In the second example, our code will still work, as the error is contained in the `error` parameter.

Now that we understand Meteor's basic folder structure, let's take a brief look at the Meteor's command-line tool.

Meteor's command-line tool

Now that we know already about Meteor's build process and folder structure, we will take a closer look at what we can do with the command-line tool that Meteor provides.

As we saw when using the `meteor` command, we need to be inside a Meteor project so that all actions will be performed on this project. For example, when we run `meteor add xxx`, we add a package to the project where we are currently in.

Updating Meteor

If Meteor releases a new version, we can simply update our project by running the following command:

```
$ meteor update
```

If we want to go back to a previous version, we can do this by running the following command:

```
$ meteor update --release 0.9.1
```

This would set our project back to release version 0.9.1.

Deploying Meteor

Deploying our Meteor app to a public server is as easy as running the following command:

```
$ meteor deploy my-app-name
```

This would ask us to register a Meteor developer account and deploy our app at `http://my-app-name.meteor.com`.

For a full introduction on how to deploy a Meteor app, refer to *Chapter 10, Deploying Our App*.

In the *Appendix*, you can find a full list of Meteor commands and their explanations.

Summary

In this chapter, we learned what Meteor requires to run, how to create a Meteor application, and how the build process works.

We understand that Meteor's folder structure is rather flexible, but that there are special folders such as the `client`, `server`, and `lib` folder, which are loaded in different places and order. We also saw how to add packages and how to use the Meteor command-line tool.

If you want to dig deeper into what we've learned so far, take a look at the following parts of the Meteor documentation:

- `https://www.meteor.com/projects`
- `https://www.meteor.com/tool`
- `https://docs.meteor.com/#/full/whatismeteor`
- `https://docs.meteor.com/#/full/structuringyourapp`
- `https://docs.meteor.com/#/full/usingpackages`
- `https://docs.meteor.com/#/full/assets`
- `https://docs.meteor.com/#/full/commandline`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter1`.

Now that we've set up our project's basic folder structure, we are ready to start with the fun part of Meteor—templates.

2
Building HTML Templates

After we successfully installed Meteor and set up our folder structure, we can now start building the basic templates for our blog.

In this chapter, we will learn how templates are built. We will see how to display data and how some parts can be altered using helper functions. We will take a look on adding events, using conditions, and understanding data contexts, all in templates.

The following is an overview of what will be covered in this chapter:

- The basic template structure
- Displaying data
- Writing template helper functions
- Using conditions in templates
- Data contexts and how those can be set
- Nesting templates and data context inheritance
- Adding events
- Building block helpers

If you jump right into this chapter without setting up the folder structure in the *Chapter 1*, *Getting Started with Meteor*, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter1`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Writing templates in Meteor

Normally when we build websites, we build the complete HTML on the server side. This was quite straightforward; every page is built on the server, then it is sent to the client, and at last JavaScript added some additional animation or dynamic behavior to it.

This is not so in single-page apps, where every page needs to be already in the client's browser so that it can be shown at will. Meteor solves this problem by providing templates that exists in JavaScript and can be placed in the DOM at some point. These templates can have nested templates, allowing for an easy way to reuse and structure an app's HTML layout.

Since Meteor is so flexible in terms of folder and file structure, any `*.html` page can contain a template and will be parsed during Meteor's build process. This allows us to put all templates in the `my-meteor-blog/client/templates` folder, which we created in the *Chapter 1*, *Getting Started with Meteor*. This folder structure is chosen as it helps us organizing templates when our app grows.

Meteor's template engine is called **Spacebars**, which is a derivative of the handlebars template engine. Spacebars is built on top of **Blaze**, which is Meteor's reactive DOM update engine.

> Blaze can generate reactive HTML directly using its API, though it's more convenient to use Meteor's Spacebars or a third-party template language built on top of Blaze such as Jade for Meteor.
>
> For more detail about Blaze, visit `https://docs.meteor.com/#/full/blaze` and `https://github.com/mquandalle/meteor-jade`.

What makes Spacebars so exciting is its simplicity and reactivity. Reactive templates mean that some parts of the template can automatically change when the underlying data changes. There is no need of manual DOM manipulation and inconsistent interfaces belong to the past. To get a better understanding of Meteor, we will start with the basic HTML files for our app:

1. Let's create an `index.html` file in our `my-meteor-blog/client` folder with the following lines of code:

```
<head>
  <title>My Meteor Blog</title>
</head>
<body>
  Hello World
</body>
```

Note that our `index.html` file doesn't contain the `<html>...</html>` tags, as Meteor gathers all `<head>` and `<body>` tags in any file and builds up its own `index.html` file, which will be delivered to the user. Actually, we can also name this file `myapp.html`.

2. Next, we run our Meteor app from the command line by typing the following command:

   ```
   $ cd my-meteor-blog
   $ meteor
   ```

   This will start a Meteor server with our app running.

3. That's it! We can open our browser, navigate to `http://localhost:3000`, and we should see **Hello World**.

What happens here is that Meteor will look through all the HTML files available in our app's folder, concatenating the content of all `<head>` and `<body>` tags, which it finds and serve them to the clients as its index file.

If we take a look at the source code of our app, we will see that the `<body>` tag is empty. This is because Meteor sees the content of the `<body>` tag as its own templates, which will be injected with its corresponding JavaScript template when the DOM is loaded.

To see the source code, don't use the Developer Tools' **elements panel**, as this will show us the source code after the JavaScript is executed. Right-click on the website instead and select **View page source** in Chrome.

We will also see that Meteor already linked all kinds of JavaScript files in our `<head>` tag. These are Meteor's core packages and our add third-party packages. In production, these files will be concatenated into one. To see this in action, go to the terminal, quit our running Meteor server using *Ctrl* + *C*, and run the following command:

```
$ meteor --production
```

If we now take a look at the source code, we will see that there is only one cryptic-looking JavaScript file linked.

For the next steps, it is better to go back to our developer mode by simply quitting Meteor and running the `meteor` command again, since this will reload the app faster when file changes occur.

Building the basic templates

Now, let's add the basic templates to our blog by creating a file called `layout.html` in the `my-meteor-blog/client/templates` folder. This template will serve as the wrapper template for our blog layout. To build the basic templates, perform the following steps:

1. Add the following lines of code to `layout.html`, which we just created:

```
<template name="layout">
  <header>
    <div class="container">
      <h1>My Meteor Single Page App</h1>
      <ul>
        <li>
          <a href="/">Home</a>
        </li>
        <li>
          <a href="/about">About</a>
        </li>
      </ul>
    </div>
  </header>

  <div class="container">
    <main>
    </main>
  </div>
</template>
```

2. Next, we will create the home page template, which will later list all our blogs posts. In the same templates folder as `layout.html`, we will create a file named `home.html` with the following lines of code:

```
<template name="home">
{{#markdown}}
## Welcome to my Blog
Here I'm talking about my latest discoveries from the world of
JavaScript.
{{/markdown}}
</template>
```

3. The next file will be a simple **About** page and we save it as `about.html` with the following code snippet:

```
<template name="about">
{{#markdown}}
## About me
```

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
minim veniam,
quis nostrud **exercitation ullamco** laboris nisi ut aliquip ex
ea commodo
consequat.

Link to my facebook: [facebook.com][1]

[1]: http://facebook.com
{{/markdown}}
</template>
```

As you can see, we used a `{{#markdown}}` block helper to wrap our texts. The curly braces are handlebars syntax, which Blaze uses to bring logic to the HTML. The `{{#markdown}}...{{/markdown}}` block will transform all markdown syntax inside into HTML when the template gets rendered.

> The markdown text cannot be indented as we do with the HTML tags because the markdown syntax interprets indentation as code.

4. To be able to use `{{#markdown}}` block helper, we need to first add the `markdown` core package to our app. To do this, we quit our running app in the terminal using *Ctrl* + *C* and type the following command:

   ```
   $ meteor add markdown
   ```

5. Now we can run the `meteor` command again to start our server.

However, when we now go to our browser, we will still see **Hello World**. So how can we make now our templates visible?

Adding templates and partials

To show the home template in the app, we need to open `index.html`, which we created earlier, and perform the following steps:

1. We replace `Hello World` with the following template inclusion helper:

   ```
   {{> layout}}
   ```

2. If we go back to our browser now, we see that the text is gone and the `layout` template, which we created earlier, has appeared with its header and menu.

3. To complete the page, we need to show the `home` template in the `layout` template. We do this by simply adding another template inclusion helper to the `main` section of the `layout` template in our `layout.html` file, as follows:

```
<main>
  {{> home}}
</main>
```

4. If we go back to the browser, we should see the following screenshot:

If we would now switch `{{> home}}` for `{{> about}}`, we would see our `about` template instead.

Displaying data with template helpers

Each template can have functions, which are called `template` helpers, and they can be used inside the template and child templates.

In addition to our custom helper functions, there are three callback functions that are called when the template is created, rendered, and destroyed. To display data with template helpers, perform the following steps:

1. To see the three callback functions in action, let's create a file called `home.js` and save it to our `my-meteor-blog/client/templates/` folder with the following code snippet:

```
Template.home.created = function(){
  console.log('Created the home template');
};
```

```
Template.home.rendered = function(){
  console.log('Rendered the home template');
};

Template.home.destroyed = function(){
  console.log('Destroyed the home template');
};
```

If we now open the console of our browser, we will see the first two callbacks are being fired. The last one will only fire if we dynamically remove the template.

2. To display data in the `home` template, we will create a helper function that will return a simple string as follows:

```
Template.home.helpers({
  exampleHelper: function(){
    return 'This text came from a helper with some <strong>HTML</
strong>.';
  }
});
```

3. Now if we go to our `home.html` file, add the `{{exampleHelper}}` helper after the `{{markdown}}` block helper, and save the file, we will see the string appearing in our browser, but we will notice that the HTML is escaped.
4. To make Meteor render the HTML correctly, we can simply replace the double curly braces with triple curly braces, as shown in the following line of code, and Blaze won't let the HTML escape:

```
{{{exampleHelper}}}
```

> Note that in most of our templates helper, we *shouldn't* use triple stache `{{{...}}}` as this opens the door for XSS and other attacks. Only use it if the HTML returned is safe to be rendered.

5. Additionally, we can return unescaped HTML using double curly braces, but we need to return the string passed through the `SpaceBars.SafeString` function, as shown in the following example:

```
Template.home.helpers({
  exampleHelper: function(){
    return new Spacebars.SafeString('This text came from a helper
with some <strong>HTML</strong>.');
  }
});
```

Setting the data context for a template

Now that we've seen how we can display data using a helper, let's see how we can set the whole data context of a template:

1. For the next examples, we will create a file called `examples.html` in our `my-meteor-blog/client/templates` folder and add the following code snippet:

```
<template name="contextExample">
  <p>{{someText}}</p>
</template>
```

2. Now that we have our `contextExample` template, we can add it to our `home` template by passing some data as follows:

```
{{> contextExample someText="I was set in the parent template's
helper, as an argument."}}
```

   This will show the text in the `contextExample` template because we were displaying it using `{{someText}}`.

Remember that filenames don't really matter as Meteor is collecting and concatenating them anyway; however, the template name matters since we use this to reference templates.

   Setting the context in HTML is not very dynamic, as it is hardcoded. To be able to dynamically change the context, it is better to set it using a `template` helper function.

3. To do this, we must first add the helper to our `home` templates helpers, which returns the data context, as follows:

```
Template.home.helpers({
  // other helpers ...
  dataContextHelper: function(){
    return {
      someText: 'This text was set using a helper of the parent
template.',
      someNested: {
        text: 'That comes from "someNested.text"'
      }
    };
  }
});
```

4. Now we can add this helper as the data context to our `contextExample` template inclusion helper, as follows:

   ```
   {{> contextExample dataContextHelper}}
   ```

5. Also, to show the nested data object we return, we can use Blaze dot syntax in the `contextExample` template by adding the following line of code to the template:

   ```
   <p>{{someNested.text}}</p>
   ```

This will now display both the `someText` and the `someNested.text`, which was returned by our helper functions.

Using the {{#with}} block helper

Another way of setting the data context is by using the `{{#with}}` block helper. The following code snippet has the same result as the former inclusion helper that utilizes the helper function:

```
{{#with dataContextHelper}}
  {{> contextExample}}
{{/with}}
```

We would even get the same results in the browser when we don't use a subtemplate and just add the content of the `contextExample` template inside the `{{#with}}` block helper, as follows:

```
{{#with dataContextHelper}}
  <p>{{someText}}</p>
  <p>{{someNested.text}}</p>
{{/with}}
```

"this" in template helpers and template callbacks

In Meteor, `this` in template helpers is used differently in template callbacks such as `created()`, `rendered()`, and `destroyed()`.

As already mentioned, templates have three callback functions that are fired in different states of the template:

- `created`: This fires when the template gets initiated but is not yet in the DOM

- `rendered`: This fires when the template and all its sub templates are attached to the DOM
- `destroyed`: This fires when the template is removed from the DOM and before the instance of the template gets destroyed

In these callback functions, `this` refers to the current template instance. The instance object can access the templates DOM and comes with the following methods:

- `this.$(selectorString)`: This method finds all elements that match `selectorString` and returns a jQuery object from those elements.
- `this.findAll(selectorString)`: This method finds all elements that match `selectorString`, but returns the plain DOM elements.
- `this.find(selectorString)`: This method finds the first element that matches `selectorString` and returns a plain DOM element.
- `this.firstNode`: This object contains the first element in the template.
- `this.lastNode`: This object contains the last element in the template.
- `this.data`: This object contains the templates data context
- `this.autorun(runFunc)`: A reactive `Tracker.autorun()` function that is stopped when the template instance is destroyed.
- `this.view`: This object contains the `Blaze.View` instance for this template. `Blaze.View` are the building blocks of reactive templates.

Inside helper functions, `this` refers only to the current data context.

To make these different behaviors visible, we will take a look at some examples:

- When we want to access the DOM of a template, we must do it in the rendered callback because only at this point, the template elements will be in the DOM. To see it in action, we edit our `home.js` file as follows:

  ```
  Template.home.rendered = function(){
    console.log('Rendered the home template');

  this.find('p').innerHTML = 'We just replaced that text!';
  };
  ```

  This will replace the first `p` tag that is created by the `{{#markdown}}` block helper, which we put there before, with the string we set. Now when we check the browser, we will see that the first `<p>` tag that contained our blog's introduction text has been replaced.

- For the next example, we need to create an additional template JavaScript file for our `contextExample` template. To do this, we create a new file called `examples.js` in our `templates` folder and save it using the following code snippet:

  ```
  Template.contextExample.rendered = function(){
    console.log('Rendered Context Example', this.data);
  };

  Template.contextExample.helpers({
    logContext: function(){
      console.log('Context Log Helper', this);
    }
  });
  ```

  This will add the rendered callback as well as a helper called `logContext` to our `contextExample` template helpers. To make this helper run, we also need to add this helper to our `contextExample` template as follows:

  ```
  <p>{{logContext}}</p>
  ```

When we now go back to the console of our browser, we see that the data context object has been returned for all the `rendered` callbacks and helpers from our rendered `contextTemplates` template. We can also see that helpers will run before the rendered callback.

In case you need access to the templates instance from inside a template helper, you can use `Template.instance()` to get it.

Now let's use make our template interactive using events.

Adding events

To make our template a bit more dynamic, we will add a simple event, which will reactively rerun the `logContext` helper we created earlier.

First, however, we need to add a button to our `contextExample` template:

```
<button>Get some random number</button>
```

To catch the click event, open `examples.js` and add the following `event` function:

```
Template.contextExample.events({
  'click button': function(e, template){
    Session.set('randomNumber', Math.random(0,99));
  }
});
```

This will set a session variable called `randomNumber` to a random number.

> We will talk in depth about sessions in the next chapter. For now, we only need to know that when a session variable changes, all functions that get that session variable using `Session.get('myVariable')` will run again.

To see this in action, we will add a `Session.get()` call to the `logContext` helper, and return the former set's random number as follows:

```
Template.contextExample.helpers({
  logContext: function(){
    console.log('Context Log Helper',this);

    return Session.get('randomNumber');
  }
});
```

If we go to the browser, we will see the **Get some random number** button. When we click on it, we see a random number appearing just above the button.

> When we use the `contextTemplates` template multiple times in our `home` template, we will see that each instance of that template helper will display the same random number. This is because the session object will rerun all its dependencies, all of which are instances of the `logHelper` helper.

Now that we have covered template helpers, let's create a custom block helper.

Block helpers

Block helpers are templates that wrap the content of the block. They can be used to show content in different ways depending on conditions, or they can be used to add extra functionality to the blocks content, for example, some JavaScript calculation on its DOM elements.

In the following example, we will build a simple block helper that will show content based on a Boolean condition.

To do this, we will to add the following code snippet at the end of our `example.html` file:

```
<template name="blockHelperExample">
  <div>
```

```
      <h1>My Block Helper</h1>
      {{#if this}}
        <p>Content goes here: {{> Template.contentBlock}}</p>
      {{else}}
        <p>Else content here: {{> Template.elseBlock}}</p>
      {{/if}}
    </div>
  </template>
```

The `{{> Template.contentBlock}}` is a predefined placeholder for the block's content. The same applies for `{{> Template.elseBlock}}`.

When `this` (in this example, we use the template's context as a simple Boolean) is `true`, it will show the given `Template.contentBlock`. Otherwise, it will show the `Template.elseBlock` content.

To see how we can use the recently created template as a block helper, take a look at the following example, which we can add to `home` template:

```
{{#blockHelperExample true}}
  <span>Some Content</span>
{{else}}
  <span>Some Warning</span>
{{/blockHelperExample}}
```

Now we should see the following screenshot:

When we now change `true`, which we pass to `{{#blockHelperExample}}`, to `false`, we should see the content after the `{{else}}` instead.

We can also use a helper function to replace the Boolean value, so that we can switch the block helper dynamically. Additionally, we can pass key-value arguments and access them by their key inside the block helper template, as shown in the following code example:

```
{{#blockHelperExample myValue=true}}
...
{{/blockHelperExample}}
```

We can also access the given argument by its name in the block template as follows:

```
<template name="blockHelperExample">
  <div>
    <h1>My Block Helper</h1>
    {{#if myValue}}
    ...
    {{/if}}
  </div>
</template>
```

Note that the data context for the block's content will be the one from the template in which the block appears, not the one of the block helper template itself.

Block helpers are a powerful tool because they allow us to write self-contained components that, when packed into a package, can be used as a drop-in-place functionality by others. This feature has the potential to allow for a vibrant marketplace, like the marketplace we see in jQuery plugins.

Listing posts

Now that we have walked through all ways of using helpers and data, I want to introduce a block helper named `{{#each}}`, which we will probably find the most useful.

If we go through all the examples completed up to now, we can see that it is better to delete all the examples of data context from our `home` template, the `examples.html` file, and its `examples.js` JavaScript file so that we can continue to build our blog cleanly.

The next step is to add a list of blog entries to our home page. For this, we need to create a template for a post preview. This can be done by performing the following steps:

1. We create a file called `postInList.html` in our `my-meteor-blog/client/templates` folder and save it with the following code snippet:

   ```
   <template name="postInList">
     <div class="postListItem">
       <h2><a href="#">{{title}}</a></h2>
       <p>{{description}}</p>
       <div class="footer">
         Posted by {{author}}
       </div>
     </div>
   </template>
   ```

   This template will be used for each post we display in the home page.

2. To make it appear, we need to add a `{{#each}}` helper to the `home` template, as follows:

   ```
   {{#each postsList}}
     {{> postInList}}
   {{/each}}
   ```

   When the `postsList` helper, which we pass to the `{{#each}}` block helper, returns an array, the content of `{{#each}}` will be repeated for each item in the array, setting the array item as the data context.

3. To see this in action, we add the `postsList` helper in our `home.js` file to the template helpers, as follows:

   ```
   Template.home.helpers({
     // other helpers ...
     postsList: function(){
       return [
         {
           title: 'My Second entry',
           description: 'Borem sodum color sit amet, consetetur
   sadipscing elitr.',
           author: 'Fabian Vogelsteller',
           timeCreated: moment().subtract(3, 'days').unix()
         },
         {
           title: 'My First entry',
   ```

```
        description: 'Lorem ipsum dolor sit amet, consetetur
sadipscing elitr.',
        author: 'Fabian Vogelsteller',
        timeCreated: moment().subtract(7, 'days').unix()
      }
    ];
  }
});
```

4. As we can see, we return an array where each item is an object containing our post's data context. For `timeCreated`, we use the `moment` function of our previously added third-party package. This will generate dummy timestamps of a few days in the past. If we now go to our browser, we will see the two posts listed, as shown in the following screenshot:

5. To display `timeCreated` from our post item in the correct format, we need to create a helper function to format the timestamp. However, because we want to use this helper in other templates later, we need to make it a global helper that can be accessed by any template. To do this, we create a file named `template-helpers.js` and save it to our `my-meteor-blog/client` folder, as it doesn't belonging to any specific template.
6. To register a global helper, we can use Meteor's `Template.registerHelper` function:

```
Template.registerHelper('formatTime', function(time, type){
  switch(type){
    case 'fromNow':
      return moment.unix(time).fromNow();
    case 'iso':
      return moment.unix(time).toISOString();
    default:
      return moment.unix(time).format('LLLL');
  }
});
```

7. Now, we only have to add the helper to our `postInList` template by replacing the content of the footer with the following code snippet:

```
<div class="footer">
  <time datetime="{{formatTime timeCreated "iso"}}">Posted
    {{formatTime timeCreated "fromNow"}} by {{author}}</time>
</div>
```

Now, if we save both the files and go back to our browser, we will see a relative date added to our blog post's footer. This works because we pass the time and a type string to the helper, as follows:

```
{{formatTime timeCreated "fromNow"}}
```

The helper then returns the formatted date using a `moment` function.

With this global helper, we can now format any Unix timestamp, in any template to relative times, ISO time strings, and a standard date format (using the LLLL format, which converts to Thursday, September 4, 1986, 8:30 P.M.).

Now that we have already used the `{{#with}}` and `{{#each}}` block helpers, let's take a look at the other default helpers and syntax that Blaze uses.

Spacebars syntax

To wrap it all up, lets summarize the Spacebars syntax:

Helper	Description
`{{myProperty}}`	The template helper can be a property from the template's data context or a template helper function. If a helper function and a property with the same name exist, the template helper will use the helper function instead.
`{{> myTemplate}}`	The inclusion helper is for a template and always expects a template object or null.
`{{> Template.dynamic template=templateName [data=dataContext]}}`	With the `{{> Template.dynamic ...}}` helper, you can render a template dynamically by providing a template helper returning a template name for the template parameter. When the helper would rerun and return a different template name, it will replace the template on this position with the new one.

`{{#myBlockHelper}}`
...
`{{/myBlockHelper}}`
	A block helper that contains both HTML and the Spacebars syntax.

By default, Spacebars comes with the following four default block helpers:

- `{{#if}}..{{/if}}`
- `{{#unless}}..{{/unless}}`
- `{{#with}}..{{/with}}`
- `{{#each}}..{{/each}}`

The `{{#if}}` block helper allows us to create simple conditions, as follows:

```
{{#if myHelperWhichReturnsABoolean}}
  <h1>Show me this</h1>
{{else}}
  <strong>If not<strong> show this.
{{/if}}
```

The `{{#unless}}` block helper works the same as `{{#if}}`, but with swapped logic.

The `{{#with}}` block, as seen earlier, will set a new data context to its content and containing templates, and the `{{#each}}` block helper will render multiple times, setting a different data context for each iteration.

Accessing parent data contexts

To complete our journey through the Spacebars syntax, let's take a closer look at the template helper syntax that we used to display data. As we've already seen, we can display data using the double curly braces syntax, as follows:

```
{{myData}}
```

Inside this helper, we can access the properties of an object using the dot syntax:

```
{{myObject.myString}}
```

We can also access a parent data context using a path-like syntax:

```
{{../myParentsTemplateProperty}}
```

Additionally, we can move more than just one context up:

```
{{../../someParentProperty}}
```

This feature allows us to be very flexible about the data context.

If we want to do the same from inside a template helper, we can use the Template API `Template.parentData(n)`, where `n` is the number of steps up to access the data context of parent templates. `Template.parentData(0)` is the same as `Template.currentData()`, or `this` if we are in a template helper.

Passing data to helpers

Passing data to helpers can be done in two different ways. We can pass arguments to a helper as follows:

```
{{myHelper "A String" aContextProperty}}
```

Then, we can access it in the helper as follows:

```
Template.myTemplate.helpers({
   myHelper: function(myString, myObject){
     // And we get:
     // myString = 'aString'
     // myObject = aContextProperty
   }
});
```

Besides this, we can pass data in the form of key-values:

```
{{myHelper myString="A String" myObject=aDataProperty}}
```

This time, however, we need to access them as follows:

```
Template.myTemplate.helpers({
   myHelper: function(Parameters){
     // And we can access them:
     // Parameters.hash.myString = 'aString'
     // Parameters.hash.myObject = aDataProperty
   }
});
```

Be aware that block and inclusion helpers act differently because they always expect objects or key-values as arguments:

```
{{> myTemplate someString="I will be available inside the template"}}

// Or

{{> myTemplate objectWithData}}
```

If we want to pass only a single variable or value to an inclusion or block helper, Meteor would objectify the argument, as we can see with the following code snippet:

```
{{#myBlock "someString"}}
...
{{/myBlock}}
```

We would then need to typecast the passed argument if we want to use it in a helper function as follows:

```
Template.myBlock.helpers({
   doSomethingWithTheString: function(){
     // Use String(this), to get the string
     return this;
   }
});
```

Beisdes, we can also simply display the string in our block helper template using `{{Template.contentBlock}}` as follows:

```
<template name="myBlock">
  <h1>{{this}}</h1>
  {{Template.contentBlock}}
</template>
```

We can also pass another template helper as an argument to an inclusion or block helper, as shown in the following example:

```
{{> myTemplate myHelperWhichReturnsAnObject "we pass a string and a
number" 300}}
```

Though passing data to template helpers and inclusion/block helpers are slightly different, arguments can be quite flexible when using helpers to generate them.

Summary

Reactive templates are one of the most impressive features of Meteor, and once we get used to them, we probably won't look back to manual DOM manipulation anymore.

After reading this chapter, we should know how to write and use templates in Meteor. We should also understand its basic syntax and how to add templates.

We saw how to access and set data in templates and how to use helpers. We learned about different types of helpers, such as inclusion helpers and block helpers. We also built our own custom block helpers and used Meteor's default helpers.

We learned that templates have three different callbacks, for when the template gets created, rendered, and destroyed.

We learned how to pass data to helpers, and how this differs in normal helpers and block helpers.

To dig deeper, take a look at the following documentations:

- `https://docs.meteor.com/#/full/templates_api`
- `https://www.meteor.com/blaze`
- `https://docs.meteor.com/#/full/blaze`
- `https://atmospherejs.com/meteor/spacebars`
- `http://momentjs.com`

You can find this chapter's code examples either at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter2`.

With all this new knowledge about templates, we are ready to add data to our database and see how we can display it in our home page.

3
Storing Data and Handling Collections

In the previous chapter, we learned how to build templates and display data in them. We built the basic layout of our app and listed some post examples on the front page.

In this chapter, we will add post examples persistently to our database on the server. We will learn how we can access this data later on the client and how Meteor syncs data between clients and the server.

In this chapter, we'll cover the following topics:

- Storing of data in Meteor
- Creating collections
- Adding data to a collection
- Querying data from a collection
- Updating data in a collection
- What "database everywhere" means
- The difference between the server's and the client's databases

If you've jumped right into the chapter and want to follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter2`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Meteor and databases

Meteor currently uses MongoDB by default to store data on the server, although there are drivers planned for use with relational databases too.

> If you are adventurous, you can try one of the community-built SQL drivers, such as the `numtel:mysql` package from `https://atmospherejs.com/numtel/mysql`.

MongoDB is a NoSQL database. This means it is based on a flat document structure instead of a relational table structure. Its document approach makes it ideal for JavaScript as documents are written in BJSON, which is very similar to the JSON format.

Meteor has a *database everywhere* approach, which means that we have the same API to query the database on the client as well as on the server. Yet, when we query the database on the client, we are only able to access the data that we *published* to a client.

MongoDB uses a data structure called **collection**, which is the equivalent of a table in a SQL database. Collections contain documents, where each document has its own unique ID. These documents are JSON-like structures and can contain properties with values, even with multiple dimensions, as follows:

```
{
  "_id": "W7sBzpBbov48rR7jW",
  "myName": "My Document Name",
  "someProperty": 123456,
  "aNestedProperty": {
    "anotherOne": "With another string"
  }
}
```

These collections are used to store data in the server's MongoDB as well as the client-side `minimongo` collection, which is an in-memory database mimicking the behavior of the real MongoDB.

> We'll discuss more about `minimongo` at the end of this chapter.

The MongoDB API allows us to use a simple JSON-based query language to get documents from a collection. We can pass additional options to only ask for *specific fields* or *sort* the returned documents. These are very powerful features, especially on the client side, to display data in various ways.

Setting up a collection

To see all this in action, let's get right on it by creating our first collection.

We create a file called `collections.js` inside our `my-meteor-blog` folder. We need to create it in the root folder so that it will be available on both the client and the server. Now let's add the following line of code to the `collections.js` file:

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

This will make the `Posts` variable globally available, as we haven't used the `var` keyword, which would restrict it to the scope of this file.

`Mongo.Collection` is the API used to query the database and it comes with the following basic methods:

- `insert`: This method is used to insert documents into the database
- `update`: This method is used to update documents or parts of them
- `upsert`: This method is used to insert or update documents or parts of them
- `remove`: This method is used to delete documents from the database
- `find`: This method is used to query the database for documents
- `findOne`: This method is used to return only the first matched document

Adding post examples

To query the database for posts, we need to add some post examples. This has to be done on the server, as we want to add them persistently. To add an example post, perform the following steps:

1. We create a file called `main.js` inside our `my-meteor-blog/server` folder. Inside this file, we will use the `Meteor.startup()` function to execute the code on the start of the server.
2. We then add the post example, but only when the collection is empty. So to prevent this, we add them every time we restart the server, as follows:

```
Meteor.startup(function(){

  console.log('Server started');

  // #Storing Data -> Adding post examples
  if(Posts.find().count() === 0) {

    console.log('Adding dummy posts');
```

```
var dummyPosts = [
  {
    title: 'My First entry',
    slug: 'my-first-entry',
    description: 'Lorem ipsum dolor sit amet.',
    text: 'Lorem ipsum dolor sit amet...',
    timeCreated: moment().subtract(7,'days').unix(),
    author: 'John Doe'
  },
  {
    title: 'My Second entry',
    slug: 'my-second-entry',
    description: 'Borem ipsum dolor sit.',
    text: 'Lorem ipsum dolor sit amet...',
    timeCreated: moment().subtract(5,'days').unix(),
    author: 'John Doe'
  },
  {
    title: 'My Third entry',
    slug: 'my-third-entry',
    description: 'Dorem ipsum dolor sit amet.',
    text: 'Lorem ipsum dolor sit amet...',
    timeCreated: moment().subtract(3,'days').unix(),
    author: 'John Doe'
  },
  {
    title: 'My Fourth entry',
    slug: 'my-fourth-entry',
    description: 'Sorem ipsum dolor sit amet.',
    text: 'Lorem ipsum dolor sit amet...',
    timeCreated: moment().subtract(2,'days').unix(),
    author: 'John Doe'
  },
  {
    title: 'My Fifth entry',
    slug: 'my-fifth-entry',
    description: 'Korem ipsum dolor sit amet.',
    text: 'Lorem ipsum dolor sit amet...',
    timeCreated: moment().subtract(1,'days').unix(),
    author: 'John Doe'
  }
];
```

```
        // we add the dummyPosts to our database
        _.each(dummyPosts, function(post){
          Posts.insert(post);
        });
      }
    });
```

Now, when check out the terminal, we should see something similar to the following screenshot:

```
2. meteor (mongod)
➜ my-meteor-blog  meteor
[[[[[ ~/Dropbox/Documents/Writing Single Page Apps using Meteor/Book - 1st Draft
/Chapter 3 rewritten/Code/my-meteor-blog ]]]]]

=> Started proxy.
=> Started MongoDB.
I20141105-14:30:52.747(1)? Server started
I20141105-14:30:53.054(1)? Adding dummy posts
=> Started your app.

=> App running at: http://localhost:3000/
```

We can also add dummy data using the Mongo console instead of writing it in our code.

To use the Mongo console, we start the Meteor server using `$ meteor`, and then in a second terminal we run `$ meteor mongo`, which brings us to a Mongo shell.

Here, we can simply add documents using MongoDB's syntax:

```
db.posts.insert({title: 'My First entry',
  slug: 'my-first-entry',
  description: 'Lorem ipsum dolor sit amet.',
  text: 'Lorem ipsum dolor sit amet...',
  timeCreated: 1405065868,
  author: 'John Doe'
})
```

Querying a collection

The server did restart when we saved our changes. At this point, Meteor added five post examples to our database.

If the server didn't restart, it means that we made a mistake in the syntax somewhere in our code. When we manually reload our browser or check out the terminal, we will see the error that Meteor gives us and we can fix it.

In case we messed up something in the database, we can always reset it using the `$ meteor reset` command in the terminal.

We can see these posts by simply opening up the console in our browser and typing the following command:

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

This will return an array with five items, each of them being one of our example posts.

To list these newly inserted posts in our front page, we need to replace the content of our `postsList` helper in the `home.js` file with the following lines of code:

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

As we can see, we returned the collections cursor directly in the helper. This return value then gets passed to the `{{#each}}` block helper in our `home` template, which will then iterate over each post while rendering the `postInList` template.

Note that `Posts.find()` returns a cursor, which is more efficient when used in an `{{#each}}` block helper, whereas `Posts.find().fetch()` will return an array with the document objects. Using `fetch()`, we can manipulate the documents before returning them.

We pass an options object as the second parameter to the `find()` function. The option we are passing will sort the result based on `timeCreated` and `-1`. The `-1` value means it will be sorted in descending order (`1` means ascending order).

Now, when we check out our browser, we will see that all of our five posts are listed, as shown in the following screenshot:

Updating a collection

Now that we know how to insert and fetch data, let's take a look at how to update data in our database.

As we've already seen before, we can use the console of our browser to play with the database. For our next examples, we will use only the console to see how Meteor reactively changes the templates when we change data.

To be able to edit a post in our database, we first need to know the `_id` field of its entry. To find this out, we need to type the following command:

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

This will return us all the documents in the `Posts` collection, as we are not passing any specific query object.

In the returned array, we need to take a look at the last item, with the **My Fifth entry** title, and copy the `_id` field to the clipboard using *Cmd* + *C* (or *Ctrl* + *C* if we're on Windows or Linux).

We can also simply use `Posts.findOne()`, which will give us the first document it finds.

Now that we have `_id`, we can simply update the title of our fifth post by typing the following command:

```
Posts.update('theCopied_Id', {$set: {title: 'Wow the title changed!'}});
```

As soon as we execute this command, we will notice that the title of the fifth post has changed to our new title, and if we now reload the page we will see that the title stays the same. This means the change was persistently made to the database.

To see Meteor's reactivity across clients, open up another browser window and navigate to `http://localhost:3000`. When we now change our title again by executing the following command, we will see that all the clients get updated in real time:

```
Posts.update('theCopied_Id', {$set: {title: 'Changed the title again'}});
```

Database everywhere

In Meteor, we can use the browser console to update data, which means that we can update the database from the client. This works because Meteor automatically syncs these changes to the server and updates the database accordingly.

This happens because we have the `autopublish` and `insecure` core packages added to our project by default. The `autopublish` package automatically publishes all documents to every client, whereas the `insecure` package allows every client to update database records by its `_id` field. Obviously, this works well for prototyping but is infeasible for production, as every client can manipulate our database.

If we remove the `insecure` package, we will need to add "allow and deny" rules to determine what a client is allowed to update and what they are not; otherwise, all updates will get denied. We will take a look at setting these rules in a later chapter, but for now this package serves us well, as we can immediately manipulate the database.

In the next chapter, we will see how to manually publish only certain documents to a client. We will start that by removing the `autopublish` package.

Differences between client and server collections

Meteor has a *database everywhere* approach. This means it provides the same API on the client as well as on the server. The data flow is controlled using a publication subscription model.

On the server sits the real MongoDB database, which stores data persistently. On the client, Meteor has a package called `minimongo`, which is a pure in-memory database mimicking most of MongoDB's query and update functions.

Every time a client connects to its Meteor server, Meteor downloads the documents that the client has subscribed to and stores them in its local `minimongo` database. From here, they can be displayed in a template or processed by functions.

When the client updates a document, Meteor syncs it back to the server, where it is passed through any allow/deny functions before being persistently stored in the database. This also works the other way; when a document in the server-side database changes, it will automatically sync to every client that is subscribed to it, keeping every connected client up to date.

Summary

In this chapter, we learned how to store data persistently in Meteor's MongoDB database. We also saw how we can query collections and update documents. We understood what the "database everywhere" approach means and how Meteor keeps every client up to date.

To dig deeper into MongoDB and to query and update collections, take a look at the following resources:

- `https://www.meteor.com/full-stack-db-drivers`
- `https://www.meteor.com/mini-databases`
- `https://docs.meteor.com/#/full/collections`
- `http://docs.mongodb.org/manual/core/crud-introduction/`
- `http://docs.mongodb.org/manual/reference/operator/query/`

You can find this chapter's code examples either at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter3`.

In the next chapter, we will see how to control the data flow using publications and subscriptions so that we send only the necessary documents to the clients.

4
Controlling the Data Flow

In the previous chapter, we learned how to store data in our database persistently. In this chapter, we will take a look at how we can tell Meteor what to send to the clients.

Until now, this all worked magically because we used the `autopublish` package, which synced all of the data with every client. Now, we will control this flow manually, sending only the necessary data to the client.

In this chapter, we'll cover the following topics:

- Synchronizing data with the server
- Publishing data to clients
- Publishing partial collections
- Publishing only the specific fields of documents
- Lazy loading more posts

If you want to jump right into the chapter and follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713`, or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter3`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Syncing data – the current Web versus the new Web

In the current Web, most pages are either static files hosted on a server or dynamically generated by a server on a request. This is true for most server-side-rendered websites, for example, those written with PHP, Rails, or Django. Both of these techniques required no effort besides being displayed by the clients; therefore, they are called *thin* clients.

In modern web applications, the idea of the browser has moved from thin clients to *fat* clients. This means most of the website's logic resides on the client and the client asks for the data it needs.

Currently, this is mostly done via calls to an API server. This API server then returns data, commonly in JSON form, giving the client an easy way to handle it and use it appropriately.

Most modern websites are a mixture of thin and fat clients. Normal pages are server-side-rendered, where only some functionality, such as a chat box or news feed, is updated using API calls.

Meteor, however, is built on the idea that it's better to use the calculation power of all clients instead of one single server. A pure fat client or a single-page app contains the entire logic of a website's frontend, which is send down on the initial page load.

The server then merely acts as a data source, sending only the data to the clients. This can happen by connecting to an API and utilizing AJAX calls, or as with Meteor, using a model called **publication/subscription**. In this model, the server offers a range of publications and each client decides which dataset it wants to subscribe to.

Compared with AJAX calls, the developer doesn't have to take care of any downloading or uploading logic. The Meteor client syncs all of the data automatically in the background as soon as it subscribes to a specific dataset. When data on the server changes, the server sends the updated documents to the clients and vice versa, as shown in the following diagram:

If this does sound insecure, be assured that we can set rules that filter changes on the server side. We will take a look at these possibilities in *Chapter 8, Security with the Allow and Deny Rules.*

Removing the autopublish package

To work with Meteor's publications/subscriptions, we need to remove the `autopublish` package, which was added by default to our project.

This package is useful for rapid prototyping, but infeasible in production since all of the data in our database would be synced to all the clients. This is not only insecure but also slows down the data loading process.

We just run the following command from inside our `my-meteor-blog` folder on the terminal:

```
$ meteor remove autopublish
```

Now we can run `meteor` again to start our server. When we check out the website, we will see that all our posts from the previous chapter are gone.

They are not really gone, however. The current server just didn't publish any yet, and the client just didn't subscribe to any; therefore, we can't see them.

Publishing data

In order to access the post on the client again, we need to tell the server to publish it to subscribing clients.

To do so, we will create a file called `publications.js` inside the `my-meteor-blog/server` folder and add the following lines of code:

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

The `Meteor.publish` function will create a publication called `all-posts` and return a cursor with all the posts from the `Post` collection in that publication.

Now, we only have to tell the client to subscribe to this publication and we will see our posts again.

We create a file called `subscriptions.js` inside the `my-meteor-blog/client` folder with the following content:

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

Now, when we check out our website, we can see that our blog posts have reappeared.

This happens because the client will subscribe to the `all-posts` publication when the `subsciptions.js` file is executed, which happens right before the page is fully loaded, as Meteor adds the `subsciptions.js` file automatically to the head of the document for us.

This means that the Meteor server sends the website first and the JavaScript builds the HTML on the client; then, all the subscriptions get synced, which populate the client's collections, and the template engine, **Blaze**, can display the posts.

Now that we have our posts back, let's see how we can tell Meteor to send only a subset of the documents from the collection.

Publishing only parts of data

To make our front page future-ready, we will need to limit the amount of posts shown on it, as we will probably have a lot of posts added with time.

For this, we will create a new publication called `limited-posts`, where we can pass a `limit` option to the posts' `find()` function and add it to our `publications.js` file, as follows:

```
Meteor.publish('limited-posts', function () {
  return Posts.find({}, {
    limit: 2,
    sort: {timeCreated: -1}
  });
});
```

We add a `sort` option, with which we sort the posts in descending order on the `timeCreated` field. This is necessary to ensure that we get the latest posts and then limit the output. If we only sort the data on the client, it might happen that we leave out newer posts, as the server publication would send only the first two documents it found, regardless of whether they are the latest ones or not.

Now we just have to go to `subscriptions.js` and change the subscription to the following line of code:

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

If we check out our browser now, we will see that only the last two posts appear on our front page, since we only subscribed to two, as shown in the following screenshot:

We must be aware that if we keep the code for the old subscription alongside the code for the new subscription, we will subscribe to both. This means Meteor merges both subscriptions and therefore keeps all the subscribed documents in our client-side collections.

We need to either comment out the old subscription or remove it before adding the new one.

Publishing specific fields

To improve publications, we can also determine which fields we want to publish from the document. For example, we can only ask for the `title` and `text` properties instead of all other properties.

This speeds up the synchronization of our subscriptions since we don't require the whole post but only the necessary data and short descriptions when listing posts on the front page.

Let's add another publication to our `publications.js` file:

```
Meteor.publish('specificfields-posts', function () {
  return Posts.find({}, {
    fields: {
      title: 1
    }
  });
});
```

As this is just an example, we pass an empty object as a query to find all the documents, and as the second parameter to `find()`, we pass an options object containing the `fields` object.

Every field that we give a value of `1` will be included in the returned document. If we rather want to work by excluding fields, we can use the field name and set the value to `0`. However, we can't use both including and excluding fields, so we need to choose what fits better, depending on the document size.

Now we can simply change the subscription in our `subscriptions.js` file to the following line of code:

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

Now, when we open the browser, it will present us with a list of our posts. Only the titles are present and the description, time, and author fields are empty:

Lazy loading posts

Now that we've gone through these simple examples, let's put them all together and add a nice lazy load feature to our posts' list on the front page.

Lazy loading is a technique that loads additional data only when the user desires it or when they scroll to the end. This can be used to increase page load, since the data to be loaded is limited. To do this, let's perform the following steps:

1. We need to add a lazy load button to the bottom of the list of posts on the front page. We go to our `home.html` file and add the following button at the end of our `home` template, right below the `{{#each postsList}}` block helper:

   ```
   <button class="lazyload">Load more</button>
   ```

2. Next, we will add the publication that will send a flexible number of posts to our `publications.js` file, as follows:

   ```
   Meteor.publish('lazyload-posts', function (limit) {
     return Posts.find({}, {
       limit: limit,
       fields: {
         text: 0
       },
       sort: {timeCreated: -1}
     });
   });
   ```

Basically, it's a combination of what we learned earlier.

- We used the `limit` option, but instead of setting a fixed number, we used the `limit` parameter, which we will later pass to this publication function.
- Previously, we used the `fields` option and excluded the `text` field.
- We can just include `fields` to get the same result. This will be safer, as it ensures that we won't get any extra fields in case the documents get extended:

  ```
  fields: {
    title: 1,
    slug: 1,
    timeCreated: 1,
    description: 1,
    author: 1
  }
  ```

- We sorted the output to make sure we are always returning the latest posts.

Now that we have set our publication, let's add a subscription so that we can receive its data.

Be aware that we need to remove any other subscription beforehand so that we are not subscribing to any other publication.

To do this, we need to make use of Meteor's `session` object. This object can be used on the client side to set variables reactively. This means every time we change this session's variable, it will run every function that uses it again. In the following example, we will use the session to increase our posts' lists' number when clicking on the lazy load button:

1. First, in the `subscription.js` file, we add the following lines of code:

   ```
   Session.setDefault('lazyloadLimit', 2);
   Tracker.autorun(function(){
   Meteor.subscribe('lazyload-posts', Session.get('lazyloadLimit'));
   });
   ```

2. Then we set the `lazyloadLimit` session variable to `2`, which will be the initial number of posts shown on the front page.
3. Next, we create a `Tracker.autorun()` function. This function will run at the start time and later at any time when we change the `lazyloadLimit` session variable to another value.
4. Inside this function, we subscribe to `lazyload-posts`, giving the `lazyloadLimit` value as a second parameter. This way, every time the session variable changes, we change our subscription with a new value.
5. Now we only need to increase the session value by clicking on the lazy load button and the subscription will change, sending us additional posts. To do this, we add the following lines of code to our `home.js` file at the end:

   ```
   Template.home.events({
     'click button.lazyload': function(e, template){
     var currentLimit = Session.get('lazyloadLimit');

     Session.set('lazyloadLimit', currentLimit + 2);
     }
   });
   ```

   This code will attach a `click` event to the lazy load button. Every time we click on this button, we get the `lazyloadLimit` session and it increases by two.

6. When we check out our browser, we should be able to click on the lazy load button at the bottom of our posts list and it should add two more posts. This should happen every time we click on the button until we reach our five example posts.

This doesn't make much sense when we have only five posts, but when there are more than 50 posts, limiting the initially shown posts to 10 will noticeably speed up page loading time.

We then need to change only the session's default value to 10 and increase it by 10, and we have a nice lazy loading effect.

Switching subscriptions

Now that we have the nice logic of lazy loading in place, let's take a look at what happens here under the hood.

The `.autorun()` function , which we created earlier, will run the first time the code gets executed, subscribing us to the `lazyload-posts` publication. Meteor then sends the first two documents of the `Posts` collection, as the `limit` we first sent is `2`.

The next time we change the `lazyloadLimit` session, it changes the subscription by changing the limit to the value we passed to the publication function.

Meteor then checks which documents exist in our client-side database in the background and requests to download the missing ones.

This will also work the other way when we decrease the session value. Meteor removes the documents that don't match the current subscription/subscriptions.

So, we can try this; we open the console of our browser and set the session limit to `5`:

```
Session.set('lazyloadLimit', 5);
```

This will immediately display all five example posts in our list. When we now set it back to a smaller value, we will see how they are removed:

```
Session.set('lazyloadLimit', 2);
```

To ensure that they are gone, we can query our local database to check, as follows:

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

This will return us an array of two items, showing that Meteor removed the posts that we are not subscribing to anymore, as shown in the following screenshot:

Some notes on data publishing

The publication and subscription model makes it fairly easy to receive and send data to the client, but as with every call to the server, sending and requesting data is expensive, as the server and the client both have to process the requests. Therefore, keep a few things in mind when building an app:

- Subscribe only to the documents that are necessary to make up the screen.
- Avoid sending fields with large content when we don't need them. This keeps our data stream leaner and faster.
- If we're using `limit` or `skip` in our publication, we need to make sure we're sorting it on the server so that we get the required data first and not some wrong tail of it.

You also should be aware that the `Meteor.publish()` function is not reactive. This means you can't use make one cursor depending on the result of another one, like you would mostly do on the client. For example, the following code snippet will not work, as it will never rerun when the comment count in the `Posts` collection changes:

```
Meteor.publish('comments', function (postId) {
    var post = Posts.find({_id: postId});

    return Comments.find({_id: {$in: post.comments}});
});
```

To solve this, you can either publish posts and comments separately and connect them in the client or use a third-party package, which allows for reactive publications such as the great `reywood:publish-composite` package at `https://atmospherejs.com/reywood/publish-composite`.

Note that the only case where the `Meteor.publish()` function reruns is when the *current user* changes so that `this.userId` which is accessible in this function will change.

Summary

In this chapter, we created a few publications and subscribed to them. We used the `fields` and `limit` options to modify the number of published documents and created a simple lazy load logic for the front page of our blog.

To dig deeper into what we learned, we can take a look at *Chapter 3*, *Storing Data and Handling Collections*. While the following Meteor documentation will give us details about the options we can use in the collections `find()` functions:

- `https://www.meteor.com/livequery`
- `https://www.meteor.com/ddp`
- `https://docs.meteor.com/#/full/publishandsubscribe`
- `https://docs.meteor.com/#/full/collections`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter4`.

In the next chapter, we will give our app what makes a real app—different pages and routes.

5
Making Our App Versatile with Routing

Since we've made it to this chapter, we should already have a good understanding of Meteor's template system and how data synchronization between a server and clients works. After digesting this knowledge, let's get back to the fun part and make our blog a real website with different pages.

You might ask, "What do pages do in a single-page app?" The term "single page" is a bit confusing, as it doesn't mean that our app consists of only one page. It's rather a term derived from the current way of doing things, as there is only one page sent down from the server. After that, all the routing and paging happens in the browser. There aren't any pages requested from the server itself anymore. A better term here would be "client-side web application," though **single page** is the current used name.

In this chapter, we will cover the following topics:

- Writing routes for our static and dynamic pages
- Changing subscriptions based on routes
- Changing the title of the website for each page

So let's not waste time and get started by adding the `iron:router` package.

If you've jumped right into the chapter and want to follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter4`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Adding the iron:router package

Routes are the URLs of a specific page in our app. In a server-side-rendered app, routes are defined either by the server's/framework's configuration or the folder structure on the server.

In a client-side app, routes are simply paths that the app will use to determine which pages to render.

The steps to perform inside the client are as follows:

1. The website is sent down to the client.
2. The JavaScript file (or files) is loaded and parsed.
3. The router code will check which current URL it is on and run the correct route function, which will then render the right templates.

> To use routes in our app, we will make use of the `iron:router` package, a router specifically written for Meteor, which makes it easy to set up routes and combine them with subscriptions.

4. To add the package, we cancel any running Meteor instance, go to our `my-meteor-blog` folder, and type the following command:

   ```
   $ meteor add iron:router
   ```

5. If we are done with this, we can start Meteor again by running the `$ meteor` command.

When we go back to the console of our browser, we will see an error saying: `Error: Oh no! No route found for path: "/"`. Don't worry; we will deal with this in the next section.

Setting up the router

In order to use the router, we need to set it up. To keep our code organized, we will create a file called `routes.js` directly in the root of our `my-meteor-blog` folder with the following code:

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

The router configuration allows you to define the following default templates:

`layoutTemplate`	The layout template will be used as the main wrapper. Here, subtemplates will be rendered in the `{{> yield}}` placeholder, which has to be placed somewhere in the template.
`notFoundTemplate`	This template will be rendered if the current URL has no defined route.
`loadingTemplate`	This template will be shown when subscriptions for the current route are loading.

For our blog, we will just define the `layoutTemplate` property for now.

Perform the following steps to set up the router:

1. To create our first route, we need to add the following lines of code to the `routes.js` file:

```
Router.map(function() {

    this.route('Home', {
        path: '/',
        template: 'home'
    });

});
```

You can also name the `Home` route as `home` (in lowercase). Then we can leave the manual template definition out, as `iron:router` will look automatically for a template called `home` for that route.

For simplicity, we define the template manually to keep all routes consistent throughout the book.

2. If we now save this file and get back to our browser, we will see the `layout` template rendered twice. This happens not because `iron:router` adds `layoutTemplate` by default to the body of our app, but because we added it manually as well as by using `{{> layout}}` in `index.html`, it gets rendered twice.

To prevent the double appearance of the `layout` template, we need to remove the `{{> layout}}` helper from the `<body>` tag inside our `index.html` file.

When we check out the browser, we will now see the `layout` template rendered only once.

Switching to a layout template

Even though we passed a template to our `Home` route using `template: home`, we are not rendering this template dynamically; we are just showing the layout template with its *hardcoded* subtemplates.

To change this, we need to replace the `{{> home}}` inclusion helper inside our layout template with `{{> yield}}`.

The `{{> yield}}` helper is a placeholder helper provided by `iron:router`, where route templates get rendered.

After doing this, when we check out the browser, we shouldn't see any change, as we are still rendering the `home` template, but this time dynamically. Then we proceed as follows:

1. In order to see whether this is true, we will add a not found template to our app, by adding the following template to our `layout.html` file after the layout template:

```
<template name="notFound">
  <div class="center">
    <h1>Nothing here</h1><br>
    <h2>You hit a page which doesn't exist!</h2>
  </div>
</template>
```

2. Now we need to add the `notFoundTemplate` property to the `Router.configure()` function of `route.js`:

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

When we now navigate to `http://localhost:3000/doesntexist` in our browser, we will see the `notFound` template being rendered instead of our `home` template:

If we click on the **Home** link in the main menu, we will get back to our front page, as this link was navigating to `"/"`. We have successfully added our first route. Now let's move on to create the second route.

Adding another route

Having a front page doesn't make a real website. Let's add a link to our **About** page, which has been in our drawer since *Chapter 2*, *Building HTML Templates*.

To do this, just duplicate the `Home` route and change the values to create an `About` route, as follows:

```
Router.map(function() {

    this.route('Home', {
        path: '/',
        template: 'home'
    });
```

```
        this.route('About', {
            path: '/about',
            template: 'about'
        });
    });
```

That's it!

Now, when we go back to our browser, we can click on the two links in the main menu to switch between our **Home** and **About** pages, and even typing in `http://localhost:3000/about` will bring us straight to the corresponding page, as shown in the following screenshot:

Moving the posts subscription to the Home route

In order to load the right data for each page, we need to have the subscription in the routes instead of keeping it in the separate `subscriptions.js` file.

The `iron:router` has a special function called `subscriptions()`, which is ideal for that purpose. Using this function, we can reactively update subscriptions belonging to a specific route.

To see it in action, add the `subscriptions()` function to our `Home` route:

```
this.route('Home', {
    path: '/',
    template: 'home',
    subscriptions: function(){
        return Meteor.subscribe("lazyload-posts", Session.
get('lazyloadLimit'));
    }
});
```

The **`Session.setDefault('lazyloadLimit', 2)`** line from the **`subscriptions.js`** file needs to be placed at the start of the `routes.js` file and before the `Router.configure()` function:

```
if(Meteor.isClient) {
    Session.setDefault('lazyloadLimit', 2);
}
```

This has to wrapped inside the `if(Meteor.isClient){}` condition, as the session object is only available on the client.

The `subscriptions()` function is *reactive* like the `Tracker.autorun()` function we used before. This means it will rerun and change the subscription when the `lazyloadLimit` session variable changes.

In order to see it working, we need to delete the `my-meteor-blog/client/subscriptions.js` file, so there are not two points where we subscribe to the same publication.

When we now check the browser and refresh the page, we will see the `home` template still shows all the example posts. Clicking on the lazy-load button increases the number of posts listed, though this time everything happens through our reactive `subscriptions()` function.

The `iron:router` comes with more hooks, which you can find as a short list in the *Appendix*.

To complete our routes, we only need to add the post routes, so we can click on a post and read it in full detail.

Setting up the post route

To be able to show a full post page, we need to create a post template, which can be loaded when the user clicks on a post.

We create a file called `post.html` inside our `my-meteor-blog/client/templates` folder with the following template code:

```
<template name="post">
  <h1>{{title}}</h1>
  <h2>{{description}}</h2>

  <small>
    Posted {{formatTime timeCreated "fromNow"}} by {{author}}
  </small>

  <div class="postContent">
    {{#markdown}}
{{text}}
    {{/markdown}}
  </div>
</template>
```

This simple template displays all the information of a blog post and even reuses our `{{formatTime}}` helper we created earlier in this book from our `template-helper.js` file. We used this to format at the time the post was created.

We can't see this template yet, as we first have to create a publication and route for this page.

Creating a single-post publication

In order to show the full post's data in this template, we need to create another publication that will send the complete post document to the client.

To do so, we open our `my-meteor-blog/server/publication.js` file and add the following publication:

```
Meteor.publish("single-post", function(slug) {
  return Posts.find({slug: slug});
});
```

The `slug` parameter, which has been used here, will be later provided from our subscription method so that we can use the `slug` parameter to reference the correct post.

A slug is a document title, which is formatted in a way that is usable in a URL. Slugs are better than just appending the document ID to the URL, as they are readable and understandable by visitors. They are also an important part of a good SEO.

So that we can use slugs, every slug has to be unique. We will take care of that when we create the posts.

Assuming that we pass the right slug such as `my-first-entry`, this publication will send down the post containing this slug.

Adding the post route

In order for this route to work, it needs to be dynamic because every linked URL has to be different for each post.

We will also render a loading template until the post is loaded. To start, we add the following template to our `my-meteor-blog/client/templates/layout.html`:

```
<template name="loading">
  <div class="center">
    <h1>Loading</h1>
  </div>
</template>
```

Additionally, we have to add this template as the default loading template to our `Router.configure()` call in the `routes.js`:

```
Router.configure({
    layoutTemplate: 'layout',
    notFoundTemplate: 'notFound',
    loadingTemplate: 'loading',
    ...
```

We then add the following lines of code to our `Router.map()` function to create a dynamic route:

```
this.route('Post', {
    path: '/posts/:slug',
    template: 'post',

    waitOn: function() {
        return Meteor.subscribe('single-post', this.params.slug);
    },
```

```
        data: function() {
            return Posts.findOne({slug: this.params.slug});
        }
    });
```

The `'/posts/:slug'` path is a dynamic route, where `:slug` can be anything and will be passed to the routes functions as `this.params.slug`. This way we can simply pass the given slug to the `single-post` subscription and retrieve the correct document for the post matching this slug.

The `waitOn()` function works like the `subscriptions()` function, though will automatically render `loadingTemplate`, we set in the `Router.configure()` until the subscriptions are ready.

The `data()` function in this route will set the data context of the `post` template. We basically look in our local database for a post containing the given slug from the URL.

> The `findOne()` method of the `Posts` collection works like `find()`, but returns only the first found result as a JavaScript object.

Let's sum up what happens here:

1. The route gets called (through a clicked link or by reloading of the page).
2. The `waitOn()` function will then subscribe to the right post identified by the given `slug` parameter, which is a part of the URL.
3. Because of the `waitOn()` function, the `loadingTemplate` will be rendered until the subscription is ready. Since this will happen very fast on our local machine, so we probably won't see the loading template at all.
4. As soon as the subscription is synced, the template gets rendered.
5. The `data()` function will then rerun, setting the data context of the template to the current post document.

Now that the publication and the route are ready, we can simply navigate to `http://localhost:3000/posts/my-first-entry` and we should see the `post` template appear.

Linking the posts

Although we've set up the route and subscription, we can't see it work, as we need the right links to the posts. As each of our previously added example posts already contains a `slug` property, we just have to add them to the links to our posts in the `postInList` template. Open the `my-meteor-blog/client/templates/postInList.html` file and change the link as follows:

```
<h2><a href="posts/{{slug}}">{{title}}</a></h2>
```

Finally, when we go to our browser and click on the title of a blog post, we get redirected to a page that shows the full post entry, like the entry shown in the following screenshot:

Changing the website's title

Now that we have the routes of our posts working, we are only missing the right titles being displayed for each page.

Sadly, `<head></head>` is not a reactive template in Meteor, where we could let Meteor do the work of changing titles and meta tags.

> It is planned to make the `head` tag a reactive template, but probably not before version 1.0.

To change the document title, we need to come up with a different way of changing it, based on the current route.

Luckily, `iron:router` has the `onAfterAction()` function, which can also be used in the `Router.configure()` function to run before every route. In this function, we have access to the data context of the current route, so we can simply set the title using native JavaScript:

```
Router.configure({
    layoutTemplate: 'layout',
    notFoundTemplate: 'notFound',

    onAfterAction: function() {
        var data = Posts.findOne({slug: this.params.slug});

        if(_.isObject(data) && !_.isArray(data))
            document.title = 'My Meteor Blog - '+ data.title;
        else
            document.title = 'My Meteor Blog - '+ this.route.
getName();
    }
});
```

Using **`Posts.findOne({slug: this.params.slug})`**, we get the current post of the route. We then check whether it's an object; if so, we add the post title to the `title` metatag. Otherwise, we just take the route name.

Doing this in `Router.configure()` will call the **`onAfterAction`** for every route.

If we now take a look at our browser's tab, we will see that the title of our website changes when we move throughout the website:

If we want to make our blog cooler, we can add the `mrt:iron-router-progress` package. This will add a progress bar at the top of our pages when changing routes. We just need to run the following command from our app's folder:

```
$ meteor add mrt:iron-router-progress
```

Summary

That's it! Our app is now a fully working website with different pages and URLs.

In this chapter, we learned how to set up static and dynamic routes. We moved our subscriptions to the routes so that they change automatically, based on the route's needs. We also used slugs to subscribe to the right posts and displayed them in the `post` template. Finally, we changed our website's title so that it matches the current route.

To learn more about `iron:router`, take a look at its documentation at `https://github.com/EventedMind/iron-router`.

You can find this chapter's code examples either at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter5`.

In the next chapter, we will take a deeper look at Meteor's session object.

6

Keeping States with Sessions

We already used Meteor's session object when we implemented our lazy load technique in an earlier chapter. In this chapter, we want to take a deeper look at it and learn how it can be used to create template-specific reactive functions.

In this chapter, we will cover the following topics:

- What sessions are
- How hot code pushes affect sessions
- Rerunning template helpers using sessions
- Rerunning functions
- Creating template-specific reactive functions

If you've jumped right into the chapter and want to follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter5`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Meteor's session object

The `Session` object provided by Meteor is a reactive data source and serves mainly to preserve global states throughout hot code reloads, though it won't preserve its data when the page is manually reloaded, making it different from PHP sessions.

A hot code reload happens when we upload new code and the server pushes those updates to all clients.

The `Session` object is a reactive data source. This means wherever this session variable is used in a reactive function, it will rerun that function when its value changes.

One use of the session variable can be to maintain global states of our app, for example, to check whether the user has their sidebar visible or not.

The session object is not useful for simple data communication between templates and other parts of the app, as maintaining this would quickly become a nightmare and naming collisions could occur.

A better way for simple reactivity

If we wanted to use something for intra-app communication, it's better to use Meteors `reactive-var` package, which comes with a `Session` like `ReactiveVar` object.

To use it, we can simply add it using `$ meteor add reactive-var`.

This object then needs to be instantiated and comes with a reactive `get()` and `set()` function like the `session` object:

```
Var myReactiveVar = new ReactiveVar('my initial value');

// now we can get it in any reactive function
myReactiveVar.get();

// and set it, to rerun depending functions
myReactiveVar.set('my new value');
```

For more custom reactivity, we can build our own custom reactive object using Meteor's `Tracker` package. To read more about this, refer to Chapter 9, *Advanced Reactivity*.

For reactive variables that are tied to a specific template instance, check out my `frozeman:template-var` package at `https://atmospherejs.com/frozeman/template-var`.

Using sessions in template helpers

As all template helper functions are reactive functions, a good place to use a session object is inside such a helper.

Reactive means that when we use a reactive object inside this function, that function will rerun when the reactive object changes, additionally rerendering this part of the template.

Template helpers are not the only reactive functions; we can also create our own using `Tracker.autorun(function(){…})`, as we saw in earlier chapters.

To demonstrate the usage of sessions in a template helper, perform the following steps:

1. Let's open our `my-meteor-blog/client/templates/home.js` file and add the following helper code anywhere in the file:

   ```
   Template.home.helpers({
     //...
     sessionExample: function(){
       return Session.get('mySessionExample');
     }
   });
   ```

   This creates the `sessionExample` helper, which returns the value of the `mySessionExample` session variable.

2. Next, we need to add this helper to our home template itself by opening the my-meteor-blog/client/templates/home.html file and adding the helper above our {{#each postsList}} block helper:

   ```
   <h2>This comes from our Session: <strong>{{sessionExample}}</
   strong></h2>
   ```

3. Now, let's open up our browser at `http://localhost:3000`. We will see the static text we add appearing in our blog's home page. Yet, to see Meteor's reactive session at work, we need to open up the browser's console and type the following line of code:

   ```
   Session.set('mySessionExample', 'I just set this.');
   ```

This is illustrated in the following screenshot:

Immediately after we pressed *Enter*, we saw the text added to our template. This is because when we call `Session.set('mySessionExample', ...)`, Meteor will rerun every reactive function wherein we called `Session.get('mySessionExample')` before. For template helpers, this will rerun only this specific template helper, rerendering only this part of the template.

We can try this by setting different values for the `mySessionExample` session variable so that we can see how the text will change at all times.

Session and hot code pushes

A hot code push is when we change files and the Meteor server pushes these changes to the clients. Meteor is smart enough to reload the page, without losing the values of HTML forms or sessions. Therefore, sessions can be used to keep user states consistent over hot code pushes.

In order to see this, we set the value of `mySessionExample` to anything we want and see the website updating to this value.

When we now go to our `home.html` file and make a minor change, for example, removing `<strong>` around the `{{sessionExample}}` helper and saving the file, we see that our sessions state is kept, even though the page reloads with the new changed template. This is demonstrated in the following screenshot:

If we manually reload the page using the browser's refresh button, the session will not be able to persist the change and the text will disappear. To overcome this limitation, there are many packages in Meteor's package repository that reactively store data in the browser's local storage to persist across page reloads. One of them is called `persistent-session` and can be found at `http://atmospherejs.com/package/persistent-session`.

Rerunning functions reactively

To rerun functions based on session changes, Meteor provides the `Tracker.autorun()` function, which we used before to change the lazy load subscription.

The `Tracker.autorun()` function will make every function we pass to it reactive. To see a simple example, we will create a function that will alert a text every time the function reruns.

> The `Tracker` package is what the session uses under the hood to make the reactivity work. In *Chapter 9*, *Advanced Reactivity*, we will take a deeper look at this package.

Perform the following steps to rerun functions reactively:

1. Let's create a new file called `main.js`, but this time in the root of the `my-meteor-blog` folder, with the following content:

```
if(Meteor.isClient) {

    Tracker.autorun(function(){
        var example = Session.get('mySessionExample');
        alert(example);
    });
}
```

> We will need the `main.js` file in later chapters. Therefore, we created it in the root folder, making it accessible on the client and the server.
>
> However, as Meteor's session object is only available on the client, we will use the `if(Meteor.isClient)` condition in order to execute the code only on the client.

   When we now check out our browser, we will see an alert that displays `undefined`. This is because the function passed to `Tracker.autorun()` will also run when the code is executed, at a time when we haven't set our session.

2. To set a session variable's default value, we can use `Session.setDefault('mySessionExample', 'My Text')`. This will set the session without running any reactive functions, when the value of the session is undefined. If the value of the session variable was already set, `setDefault` won't change the variables at all.

3. In our example, we probably don't want an alert window to appear when the page is loaded. To prevent this first run, we can use the `Tracker.Computation` object, which is passed as the first argument to our function and which provides us with a property called `firstRun`. This property will be set to `true` at the first run of the function. When we use this, we can prevent the display of the alert at the start:

```
Tracker.autorun(function(c){
    var example = Session.get('mySessionExample');

    if(!c.firstRun) {
        alert(example);
    }
});
```

4. Now let's go to the browser's console and set the session to any value to see the alert appear:

```
Session.set('mySessionExample','Hi there!');
```

The output of this code is demonstrated in the following screenshot:

When we run the same command again, we will not see the alert window show up, as Meteor is smart enough to prevent reruns when the session's value doesn't change. If we set it to another value, the alert will appear again.

Stopping reactive functions

The `Tracker.Computation` object, passed as the first argument, also gives us a way to stop the function from being reactive at all. To try this, we will change the function so that it stops its reactivity when we pass the `stop` string to the session:

```
Tracker.autorun(function(c){
    var example = Session.get('mySessionExample');

    if(!c.firstRun) {
        if(Session.equals('mySessionExample', 'stop')) {
            alert('We stopped our reactive Function');
            c.stop();
        } else {
            alert(example);
        }
    }
});
```

Now, when we go to our browser's console and run `Session.set('mySessionExample', 'stop')`, the reactive function will stop being reactive. To test this, we can try to run `Session.set('mySessionExample', 'Another text')` and we will see that the alert window won't appear.

If we make a code change and a hot code reload happens, the reactive function will become reactive again, as the code was executed again.

The preceding example also uses a function called `Session.equals()`. This function can compare two scalar values while preventing unnecessary recomputations, compared to using `Session.get('mySessionExample) === 'stop'`. Using `Session.equals()` would only rerun this function when the session variable changes *to* or *from* that value.

In our example, however, this function doesn't make a difference, as we called `Session.get()` before as well.

Using autorun in a template

Although it could be useful to use `Tracker.autorun()` globally in our app in some cases, it can become quickly hard to maintain those global reactive functions as our app grows.

Therefore, it is good practice to bind reactive functions to the templates for which they perform actions.

Luckily, Meteor offers a special version of `Tracker.autorun()` that is tied to a template instance and stops automatically when the template gets destroyed.

To make use of this, we can start the reactive function in the `created()` or rendered callback. To start, let's comment out our previous example from the `main.js` file so that we won't get two alert windows.

Open our `home.js` file and add the following lines of code:

```
Template.home.created = function(){

    this.autorun(function(){
        alert(Session.get('mySessionExample'));
    });
};
```

This will create the reactive function when the home template is created. When we go to the browser's console and set the `mySessionExample` session to a new value, we will see the alert window appear, as shown in the following screenshot:

Now, when we switch the templates by clicking on the **About** link in the menu and we set the `mySessionExample` session variable again to another value using the browsers console, we won't see the alert window appear as the reactive `this.autorun()` was stopped when the template was destroyed.

Note that all `Tracker.autorun()` functions return a `Tracker.Computation` object, which can be used to stop the reactivity of the autorun at any time using `Tracker.Computation.stop()`:

```
Var myReactiveFunction = Tracker.autorun(function()
{...});
// Do something which needs to stop the autorun
myReactiveFunction.stop();
```

The reactive session object

We've seen that the session object can rerun a function when its value is changed. This is the same behavior as that of the `find()` and `findOne()` functions of collections, which will rerun functions when the underlying data in the collection changes.

We can use sessions to keep user states across hot code pushes, such as states of drop-down menus or pop-ups. However, keep in mind that without a clear naming convention, these session variables can soon become hard to maintain.

For more specific reactive behavior, it is good to build a custom reactive object using Meteor's `Tracker` core package, which we will cover in *Chapter 9*, *Advanced Reactivity*.

Summary

In this chapter, we learned what we can do with Meteor's reactive session object. We used it to rerun template helpers and our own custom functions, and we made a reactive function template specific using the `created()` and `destroyed()` callbacks.

To dig deeper, take a look at Meteor's documentation about sessions and reactivity at the following resources:

- `https://docs.meteor.com/#/full/reactivity`
- `https://docs.meteor.com/#/full/session`
- `https://docs.meteor.com/#/full/reactivevar_pkg`
- `https://www.meteor.com/tracker`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter6`.

In the next chapter, we will create the admin user and backend for our blog, laying down the foundation to create and edit posts.

7
Users and Permissions

Having worked through the previous chapters, we should have a working blog by now. We can click on all links and posts, and even lazy load more posts.

In this chapter, we will add our backend login and create the admin user. We will also create the template to edit posts and make an edit button visible to the admin user so that they can edit and add new content.

In this chapter, we will learn the following concepts:

- Meteor's `accounts` package
- Creating users and a log in
- How to restrict certain routes to only logged-in users

You can delete all the session examples from the previous chapter, as we won't need them to progress with our app. Delete the session's code from `my-meteor-blog/main.js`, `my-meteor-blog/client/templates/home.js`, and `my-meteor-blog/client/templates/home.html`, or download a fresh copy of the previous chapter's code.

If you've jumped right into the chapter and want to follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter6`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Meteor's accounts packages

Meteor makes it very easy to add authentication to our web app using its `accounts` package. The `accounts` package is a complete login solution tied to Meteor's core. Created users can be identified by ID in many of Meteor's server-side functions, for example, in a publication:

```
Meteor.publish("examplePublication", function () {
  // the current loggedin user id can be accessed via
  this.userId;
}
```

Additionally, we can add support for login via Facebook, GitHub, Google, Twitter, Meetup, and Weibo by simply adding one or more of the `accounts-*` core packages.

Meteor also comes with a simple login interface, an extra template that can be added using the `{{> loginButtons}}` helper.

All registered user profiles will be stored in a collection called `Users`, which Meteor creates for us. All the processes in authentication and communication use the **Secure Remote Password** (**SRP**) protocol and most external services use OAuth.

For our blog, we will simply create one admin user, which when logged in will be able to create and edit posts.

If we want to use one of the third-party services to log in, we can work through this chapter first, and then add one of the previously mentioned packages.

After we add the additional packages, we can open up the **Sign in** form. We will see a button where we can configure the third-party services for use with our app.

Adding the accounts packages

To start using a login system, we need to add the `accounts-ui` and `accounts-password` packages to our app, as follows:

1. To do so, we open up the terminal, navigate to our `my-meteor-blog` folder, and type the following command:

   ```
   $ meteor add accounts-ui accounts-password
   ```

2. After we have successfully added the packages, we can run our app again using the `meteor` command.

3. As we want to prevent the creation of additional user accounts by our visitors, we need to disallow this functionality in our `accounts` package, `config`. First, we need to open up our `my-meteor-blog/main.js` file, which we created in the previous chapter, and remove all of the code, as we won't need the session examples anymore.
4. Then add the following lines of code to this file, but make sure you don't use `if(Meteor.isClient)`, as we want to execute the code on both the client and the server this time:

   ```
   Accounts.config({
       forbidClientAccountCreation: true
   });
   ```

   This will forbid any call of `Accounts.createUser()` on the client and the `accounts-ui` package will not show the **Register** button to our visitors.

This option doesn't seem to work for third-party services. So, when using third-party services, everybody can sign up and edit posts. To prevent this, we will need to create "deny" rules for user creation on the server side, which is beyond the scope of this chapter.

Adding admin functionality to our templates

The best way to allow editing of our post is to add an **Edit post** link to our post's page, which can only be seen if we are logged in. This way, we save rebuilding a similar infrastructure for an additional backend, and make it easy to use as there is no strong separation between frontend and backend.

First, we will add a **Create new post** link to our `home` template, then add the **Edit post** link to the post's `pages` template, and finally add the login buttons and form to the main menu.

Adding a link for new posts

Let's begin by adding a **Create new post** link. Open the `home` template at `my-meteor-blog/clients/templates/home.html` and add the following lines of code just above the `{{#each postsList}}` block helper:

```
{{#if currentUser}}
    <a href="/create-post" class="createNewPost">Create new post</a>
{{/if}}
```

The `{{currentUser}}` helper comes with the `accounts-base` package, which was installed when we installed our `accounts` packages. It will return the current logged-in user, or return null if no user is logged in. Using it inside an `{{#if}}` block helper allows us to show content only to logged-in users.

Adding the link to edit posts

To edit posts, we simply add an **Edit post** link to our `post` template. Open up `post.html` from the same folder and add `{{#if currentUser}}..{{/if}}` after `{{author}}`, as follows:

```
<small>
    Posted {{formatTime timeCreated "fromNow"}} by {{author}}

    {{#if currentUser}}
        | <a href="/edit-post/{{slug}}">Edit post</a>
    {{/if}}
</small>
```

Adding the login form

Now that we have both links to add and edit posts, let's add the login form. We can create our own form, but Meteor already comes with a simple login form, which we can style to fit our design.

Since we added the `accounts-ui` package previously, Meteor provides us with the `{{> loginButtons}}` template helper, which works as a drop-in-place template. To add this, we will open our `layout.html` template and add the following helper inside our menu's `<ul></ul>` tags, as follows:

```
<h1>My Meteor Single Page App</h1>
<ul>
    <li>
        <a href="/">Home</a>
    </li>
    <li>
        <a href="/about">About</a>
    </li>

</ul>

{{> loginButtons}}
```

Creating the template to edit posts

Now we are only missing the template to edit the posts. To add this, we will create a file called `editPost.html` inside our `my-meteor-blog/client/templates` folder, and fill it with the following lines of code:

```
<template name="editPost">
  <div class="editPost">
     <form>
        <label>
          Title
          <input type="text" name="title" placeholder="Awesome title"
value="{{title}}">
        </label>

        <label>
          Description
          <textarea name="description" placeholder="Short description
displayed in posts list" rows="3">{{description}}</textarea>
        </label>

        <label>
          Content
          <textarea name="text" rows="10" placeholder="Brilliant
content">{{text}}</textarea>
        </label>

        <button type="submit" class="save">Save Post</button>
    </form>
  </div>
</template>
```

As we can see, we have added the helpers for `{{title}}`, `{{description}}`, and `{{text}}`, which will come later from the post's data. This simple template, with its three text fields, will allow us to edit and create new posts later.

If we now check out our browser, we will notice that we can't see any of the changes we made so far, apart from the **Sign in** link in the corner of our website. To be able to log in, we first need to add our admin user.

Creating the admin user

Since we deactivated the creation of users from the client, as a security measure we will create the admin user on the server in the same way we created our example posts.

Open the `my-meteor-blog/server/main.js` file and add the following lines of code somewhere inside `Meteor.startup(function(){...})`:

```
if(Meteor.users.find().count() === 0) {

    console.log('Created Admin user');

    Accounts.createUser({
        username: 'johndoe',
        email: 'johndoe@example.com',
        password: '1234',
        profile: {
            name: 'John Doe'
        }
    });
}
```

If we now go to our browser, we should be able to log in using the user we just created, and we immediately see that all the edit links appear.

However, when we click any of the edit links, we will see the `notFound` template appearing because we didn't create any of our admin routes yet.

Adding permissions

Meteor's `account` package doesn't come by default with configurable permissions for users.

To add permission control, we can add a third-party package such as the `deepwell:authorization` package, which can be found on Atmosphere at `http://atmospherejs.com/deepwell/authorization` and which comes with a complex role model.

If we want to do it manually, we can add the simple `roles` properties to our user document when we create the user, and then check for these roles in our allow/deny roles when we create or update posts. We will learn about allow/deny rules in the next chapter.

If we create a user using the `Accounts.createUser()` function, we can't add a custom property, so we need to update the user document after we have created the user, as follows:

```
var userId = Accounts.createUser({
  username: 'johndoe',
  email: 'johndoe@example.com',
  password: '1234',
  profile: {
    name: 'John Doe'
  }
});
// add the roles to our user
Meteor.users.update(userId, {$set: {
  roles: {admin: true},
}})
```

By default, Meteor publishes the `username`, `emails`, and `profile` properties of the currently logged-in user. To add additional properties, such as our custom `roles` property, we need to add a publication, to access the `roles` property on the client as well, as follows:

1. Open the my-meteor/blog/server/publications.js file and add the following publication:

   ```
   Meteor.publish("userRoles", function () {
    if (this.userId) {
     return Meteor.users.find({_id: this.userId}, {fields: {roles:
   1}});
    } else {
     this.ready();
    }
   });
   ```

2. In the `my-meteor-blog/main.js` file, we add the subscription as follows:

   ```
   if(Meteor.isClient){
     Meteor.subscribe("userRoles");
   }
   ```

3. Now that we have the `roles` property available on the client, we can change `{{#if currentUser}}..{{/if}}` in the `home` and `post` templates to `{{#if currentUser.roles.admin}}..{{/if}}` so that only admins can see the buttons.

A note on security

The user can only update their own `profile` property using the following command:

```
Meteor.users.update(ownUserId, {$set: {profiles:{myProperty: 'xyz'}}})
```

If we want to update the `roles` property, we will fail. To see this in action, we can open up the browser's console and type the following command:

```
Meteor.users.update(Meteor.user()._id, {$set:{ roles: {admin: false}}});
```

This will give us an error stating: **update failed: Access denied**, as shown in the following screenshot:

If we want to allow users to edit other properties such as their `roles` property, we need to add a `Meteor.users.allow()` rule for that.

Creating routes for the admin

Now that we have an admin user, we can add the routes, which lead to the `editPost` template. Though in theory the `editPost` template is available to every client, it doesn't create any risk, as the allow and deny rules are the real security layer, which we will take a look at in the next chapter.

To add the route to create posts, let's open up our `my-meteor-blog/routes.js` file and add the following route to the `Router.map()` function:

```
this.route('Create Post', {
    path: '/create-post',
    template: 'editPost'
});
```

This will simply show the `editPost` template as soon as we click on the **Create new post** link on our home page, as shown in the following screenshot:

We see that the form is empty because we did not set any data context to the template, and therefore the `{{title}}`, `{{description}}`, and `{{text}}` placeholders in the template displayed nothing.

To make the edit post route work, we need to add subscriptions similar to those we did for the `Post` route itself. To keep things **DRY** (which means **Don't Repeat Yourself**), we can create a custom controller, which both routes will use, as follows:

1. Add the following lines of code after the `Router.configure(...);` call:

```
PostController = RouteController.extend({
    waitOn: function() {
        return Meteor.subscribe('single-post', this.params.slug);
    },

    data: function() {
        return Posts.findOne({slug: this.params.slug});
    }
});
```

2. Now we can simply edit the `Post` route, remove the `waitOn()` and `data()` functions, and add `PostController` instead:

```
this.route('Post', {
    path: '/posts/:slug',
    template: 'post',
    controller: 'PostController'
});
```

3. Now we can also add the `Edit Post` route by just changing the `path` and the `template` properties:

```
this.route('Edit Post', {
    path: '/edit-post/:slug',
    template: 'editPost',
    controller: 'PostController'
});
```

4. That's it! When we now go to our browser, we will be able to access any post and click on the **Edit** button, and we will be directed to `editPost` template.

If you are wondering why the form is filled in with the post data, take a look at `PostController`, which we just created. Here, we return the post document inside the `data()` function, setting the data context of the template to the post's data.

Now that we have these routes in place, we should be done. Shouldn't we?

Not yet, because everybody who knows the `/create-post` and `/edit-post/my-title` routes can simply see the `editPost` template, even if he or she is not an admin.

Preventing visitors from seeing the admin routes

To prevent visitors from seeing admin routes, we need to check whether the user is logged in before we show them the routes. The `iron:router` comes with a `Router.onBeforeAction()` hook, which can be run for all or some routes. We will use this to run a function to check whether the user is logged in; if not, we will pretend that the route doesn't exist and simply display the `notFound` template.

Add the following code snippet at the end of the `routes.js` file:

```
var requiresLogin = function(){
    if (!Meteor.user() ||
        !Meteor.user().roles ||
        !Meteor.user().roles.admin) {
        this.render('notFound');

    } else {
        this.next();
    }
};

Router.onBeforeAction(requiresLogin, {only: ['Create Post','Edit
Post']});
```

Here, first we create the `requiresLogin()` function, which will be executed before the `Create Post` and `Edit Post` routes because we pass them as the second arguments to the `Router.onBeforeAction()` function.

Inside the `requiresLogin()`, we check whether the user is logged in, which will return the user document when calling `Meteor.user()`, and if they have the role `admin`. If not, we simply render the `notFound` template and don't continue to the route. Otherwise, we run `this.next()`, which will continue to render the current route.

That's it! If we now log out and navigate to the `/create-post` route, we will see the `notfound` template.

If we log in, the template will switch and immediately show the `editPost` template.

This happens because the `requiresLogin()` function becomes reactive as soon as we pass it to `Router.onBeforeAction()`, and since `Meteor.user()` is a reactive object, any change to the user's status will rerun this function.

Now that we have created the admin user and the necessary templates, we can move on to actually creating and editing the posts.

Summary

In this chapter, we learned how to create and log in users, how we can show content and templates only to logged-in users, and how routes can be altered depending on the login status.

To learn more, take a look at the following links:

- `https://www.meteor.com/accounts`
- `https://docs.meteor.com/#/full/accounts_api`
- `https://docs.meteor.com/#/full/meteor_users`
- `http://en.wikipedia.org/wiki/Secure_Remote_Password_protocol`
- `https://github.com/EventedMind/iron-router/blob/devel/Guide.md#using-hooks`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter7`.

In the next chapter, we will learn how we can create and update posts and how to control updates to the database from the client side.

8
Security with the Allow and Deny Rules

In the previous chapter, we created our admin user and prepared the `editPost` template. In this chapter, we will make this template work so that we can create and edit posts using it.

To make it possible to insert and update documents in our database, we need to set constraints so that not everybody can change our database. In Meteor, this is done using the allow and deny rules. These functions will check documents before they are inserted into the database.

In this chapter, you will cover the following topics:

- Adding and updating posts
- Using the allow and deny rules to control the updating of the database
- Using methods on the server for more flexibility
- Using method stubs to enhance user experience

If you've jumped right into the chapter and want to follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter7`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Adding a function to generate slugs

In order to generate slugs from our post's titles, we will use the `underscore-string` library, which comes with a simple `slugify()` function. Luckily, a wrapper package for this library already exists on the Meteor package servers. To add it, we run the following command from the terminal in our `my-meteor-blog` folder:

```
$ meteor add wizonesolutions:underscore-string
```

This will extend `underscore`, which is used by default in Meteor, with extra string functions such as `_.slugify()`, to generate a slug from strings.

Creating a new post

Now that we can generate slugs for each created page, we can proceed to add the saving process to the `editPost` template.

To do so, we need to create a JavaScript file for our `editPost` template by saving a file called `editPost.js` to the `my-meteor-blog/client/templates` folder. Inside this file, we will add an event for the **Save** button of the template:

```
Template.editPost.events({
  'submit form': function(e, template){
    e.preventDefault();
    console.log('Post saved');
  }
});
```

Now, if we go to the `/create-post` route and click on the **Save Post** button, the **Post saved** log should appear in the browser's console.

Saving a post

In order to save the post, we will simply take the form's content and store it in the database. Later, we'll redirect to the newly created post page. To do so, we extend our click event with the following lines of code:

```
Template.editPost.events({
  'submit form': function(e, tmpl){
    e.preventDefault();
    var form = e.target,
        user = Meteor.user();
```

We get the current user so that we can add him later as the post's author. We then generate a slug from the post's title using our `slugify()` function:

```
var slug = _.slugify(form.title.value);
```

Following this, we insert the post document into the `Posts` collection using all other form fields. For the `timeCreated` property, we get the current Unix timestamp using the `moment` package, which we added in *Chapter 1*, *Getting Started with Meteor*.

The `owner` field will later help us to determine by which user this post was created:

```
Posts.insert({
title: form.title.value,
slug: slug,
description: form.description.value,
text: form.text.value,
timeCreated: moment().unix(),
author: user.profile.name,
owner: user._id

}, function(error) {
if(error) {
// display the error to the user
alert(error.reason);
} else {
// Redirect to the post
Router.go('Post', {slug: slug});
}
});
```

The second argument we pass to the `insert()` function is a callback function provided by Meteor that will receive an error argument if something goes wrong. If an error happens, we alert it, and if everything goes fine, we redirect to the newly inserted post using our generated slug.

Since our route controller will then subscribe to a post with this slug, it will be able to load our newly created post and display it in the post template.

Now, if we go to the browser, fill in the form, and click on the **Save** button, we should have created our first own post!

So saving works. What about editing?

When we click on the **Edit** button in the post, we will be shown the `editPost` template again. This time, however, the form fields are filled with the data from the post. So far so good, but if we press the **Save** button now, we will create another post instead of updating the current one.

Updating the current post

Since we set the data context of the `editPost` template, we can simply use the presence of the post's `_id` field as an indicator to update, instead of inserting the post data:

```
Template.editPost.events({
    'submit form': function(e, tmpl){
        e.preventDefault();
        var form = e.target,
            user = Meteor.user(),
            _this = this; // we need this to reference the slug in the
callback

        // Edit the post
        if(this._id) {

            Posts.update(this._id, {$set: {
                title:          form.title.value,
                description:    form.description.value,
                text:           form.text.value

            }}, function(error) {
                if(error) {
                    // display the error to the user
                    alert(error.reason);
                } else {
                    // Redirect to the post
                    Router.go('Post', {slug: _this.slug});
                }
            });


        // SAVE
        } else {

            // The insertion process ...

        }
    }
});
```

Knowing the `_id`, we can simply update the current document using the `$set` property. Using `$set` will only overwrite the `title`, `description`, and `text` fields. The other fields will be left as they are.

Note that we now also need to create the `_this` variable on top of the function in order to access the `slug` property of the current data context in the callback later. This way, we can later redirect to our edited post page.

Now, if we save the file and go back to our browser, we can edit the post and click on **Save**, and all changes will be saved as expected to our database.

Now, we can create and edit posts. In the next section, we will learn how to restrict updates to the database by adding the allow and deny rules.

Restricting database updates

Until now, we simply added the insert and update functionality to our `editPost` template. However, anybody can insert and update data if they just type an `insert` statement into their browser's console.

To prevent this, we need to properly check for insertion and update rights on the server side before updating the database.

Meteor's collections come with the allow and deny functions, which will be run before every insertion or update to determine whether the action is allowed or not.

The allow rules let us allow certain documents or fields to be updated, whereas the deny rules overwrite any allow rules and definitely deny any action on its collection.

To make this more visible, let's visualize an example where we define two allow rules; one will allow certain documents' `title` fields to be changed and another will allow only editing of the `description` fields, but an additional deny rule can prevent one specific document to be edited in any case.

Removing the insecure package

To start using the allow and deny rules, we need to remove the `insecure` package from our app so that no client can simply make changes to our database without passing our allow and deny rules.

Quit the running `meteor` instance using *Ctrl* + *C* in the terminal and run the following command:

```
$ meteor remove insecure
```

After we have successfully removed the package, we can run Meteor again using the `meteor` command.

When we now go to our browser and try to edit any post, we will see an alert window stating **Access denied**. Remember that we added this `alert()` call before, when an update or insert action failed?

Adding our first allow rules

In order to make our posts editable again, we need to add allow rules to enable database updates again.

To do so, we will add the following allow rules to our `my-meteor-blog/collections.js` file, but in this case we'll execute them only on the server side by checking against Meteor's `isServer` variable, as follows:

```
if(Meteor.isServer) {

    Posts.allow({
        insert: function (userId, doc) {
            // The user must be logged in, and the document must be
owned by the user
            return userId && doc.owner === userId && Meteor.user().
roles.admin;
        },
```

In the insertion *allow* rule , we will insert the document only if the post owner matches the current user and if the user is an admin, which we can determine by the `roles.admin` property we added in the previous chapter.

If the allow rule returns `false`, the insertion of the document will be denied. Otherwise, we will successfully add a new post. Updating works the same way, just that we only check whether the current user is an admin:

```
        update: function (userId, doc, fields, modifier) {
            // User must be an admin
            return Meteor.user().roles.admin;
        },
        // make sure we only get this field from the documents
        fetch: ['owner']
    });
}
```

The arguments passed to the `update` function are listed in the following table:

Field	Description
`userId`	The user ID of the current logged-in user, who performs that `update` action
`doc`	The document from the database, without the proposed changes
`fields`	An array with field parameters that will be updated
`modifier`	The modifier the user passed to the `update` function, such as `{$set: {'name.first': "Alice"}, $inc: {score: 1}}`

The `fetch` property, which we specify last in the allow rule's object, determines which fields of the current document should be passed to the update rule. In our case, we only need the `owner` property for our update rule. The `fetch` property exists for performance reasons, to prevent unnecessarily large documents from being passed to the rule's functions.

Additionally, we can specify the `remove()` rule and the `transform()` function. The `remove()` rule will get the same arguments as the `insert()` rule and allow or prevent removal of documents.

The `transform()` function can be used to transform the document before being passed to the allow or deny rules, for example, to normalize it. However, be aware that this won't change the document that gets inserted into the database.

If we now try to edit a post in our website, we should be able to edit all posts as well as create new ones.

Adding a deny rule

To improve security, we can fix the owner of the post and the time when it was created. We can prevent changes to the owner and the `timeCreated` and `slug` fields by adding an additional deny rule to our `Posts` collection, as follows:

```
if(Meteor.isServer) {

  // Allow rules

  Posts.deny({
    update: function (userId, docs, fields, modifier) {
      // Can't change owners, timeCreated and slug
```

```
        return _.contains(fields, 'owner') || _.contains(fields,
'timeCreated') || _.contains(fields, 'slug');
      }
    });
  }
```

This rule will simply check whether the `fields` argument contains one of the restricted fields. If it does, we deny the update to this post. So, even if our previous allow rules have passed, our deny rule ensures that the document doesn't change.

We can try the deny rule by going to our browser's console, and when we are at a post page, typing the following commands:

```
Posts.update(Posts.findOne()._id, {$set: {'slug':'test'}});
```

This should give you an error stating **update failed: Access denied**, as shown in the following screenshot:

Though we can add and update posts now, there is a better way of adding new posts than simply inserting them into our `Posts` collection from the client side.

Adding posts using a method call

Methods are functions that can be called from the client and will be executed on the server.

Method stubs and latency compensation

The advantage of methods is that they can execute code on the server, having the full database and a stub method on the client available.

For example, we can have a method do something on the server and simulate the expected outcome in a stub method on the client. This way, the user doesn't have to wait for the server's response. A stub can also invoke an interface change, such as adding a loading indicator.

One native example of a method call is Meteor's `Collection.insert()` function, which will execute a client-side function, inserting the document immediately into the local `minimongo` database as well as sending a request executing the real `insert` method on the server. If the insertion is successful, the client has the document already inserted. If an error occurs, the server will respond and remove the inserted document from the client again.

In Meteor, this concept is called **latency compensation**, as the interface reacts immediately to the user's response and therefore compensates the latency, while the server's round trip will happen in the background.

Inserting a post using a method call enables us to simply check whether the slug we want to use for the post already exists in another post. Additionally, we can use the server's time for the `timeCreated` property to be sure we are not using an incorrect user timestamp.

Changing the button

In our example, we will simply use the method stub functionality to change the text of the **Save** button to `Saving...` while we run the method on the server. To do so, perform the following steps:

1. To start, let's first change the **Save** button's static text with a template helper so that we can change it dynamically. Open up `my-meteor-blog/client/templates/editPost.html` and replace the **Save** button code with the following code:

   ```
   <button type="submit" class="save">{{saveButtonText}}</button>
   ```

2. Now open `my-meteor-blog/client/templates/editPost.js` and add the following template helper function at the beginning of the file:

   ```
   Session.setDefault('saveButton', 'Save Post');
   Template.editPost.helpers({
     saveButtonText: function(){
       return Session.get('saveButton');
     }
   });
   ```

   Here, we return the session variable named `saveButton`, which we set to the default value, `Save Post`, earlier.

Changing the session will allow us to change the text of the **Save** button later while saving the document.

Adding the method

Now that we have a dynamic **Save** button, let's add the actual method to our app. For this, we will create a new file called `methods.js` directly in our `my-meteor-blog` folder. This way, its code will be loaded on the server and the client, which is necessary to execute the method on the client as a stub.

Add the following lines of code to add a method:

```
Meteor.methods({
    insertPost: function(postDocument) {

        if(this.isSimulation) {
            Session.set('saveButton', 'Saving...');
        }
    }
});
```

This will add a method called `insertPost`. Inside this method, the stub functionality is already added by making use of the `isSimulation` property, which is made available in the `this` object of the function by Meteor.

The `this` object also has the following properties:

- `unblock()`: This is a function that when called will prevent the method from blocking other method calls
- `userId`: This contains the current user's ID
- `setUserId()`: This a function to connect the current client with a certain user
- `connection`: This is the connection on the server through which this method is called

If `isSimulation` is set to `true`, the method is not run on the server side but as a stub on the client. Inside this condition, we simply set the `saveButton` session variable to `Saving...` so that the button text will change:

```
Meteor.methods({
  insertPost: function(postDocument) {

    if(this.isSimulation) {

      Session.set('saveButton', 'Saving...');

    } else {
```

To complete the method, we will add the server-side code for post insertion:

```
      var user = Meteor.user();

      // ensure the user is logged in
      if (!user)
      throw new Meteor.Error(401, "You need to login to write a
post");
```

Here, we get the current user to add the author name and owner ID.

We throw an exception with `new Meteor.Error` if the user is not logged in. This will stop the execution of the method and return an error message we define.

We also search for a post with the given slug. If we find one, we prepend a random string to the slug to prevent duplicates. This makes sure that every slug is unique, and we can successfully route to our newly created post:

```
      if(Posts.findOne({slug: postDocument.slug}))
      postDocument.slug = postDocument.slug +'-'+ Math.random().
toString(36).substring(3);
```

Before we insert the newly created post, we add `timeCreated` using the `moment` library and the `author` and `owner` properties:

```
      // add properties on the serverside
      postDocument.timeCreated = moment().unix();
      postDocument.author      = user.profile.name;
      postDocument.owner       = user._id;

      Posts.insert(postDocument);
```

After we insert the document, we return the corrected slug, which will then be received in the callback of the method call as the second argument:

```
        // this will be received as the second argument of the method
callback
        return postDocument.slug;
    }
  }
});
```

Calling the method

Now that we have created our `insertPost` method, we can change the code in the submit event, where we inserted the post earlier in our `editPost.js` file, with a call to our method:

```
var slug = _.slugify(form.title.value);

Meteor.call('insertPost', {
  title:          form.title.value
  slug:           slug,
  description:    form.description.value
  text:           form.text.value,

}, function(error, slug) {
  Session.set('saveButton', 'Save Post');

  if(error) {
    return alert(error.reason);
  }

  // Here we use the (probably changed) slug from the server side
method
  Router.go('Post', {slug: slug});
});
```

As we can see in the callback of the method call, we route to the newly created post using the `slug` variable we received as the second argument in the callback. This ensures that if the `slug` variable is modified on the server side, we use the modified version to route to the post. Additionally, we reset the `saveButton` session variable to change the text to `Save Post` again.

That's it! Now, we can create a new post and save it using our newly created `insertPost` method. However, editing will still be done from the client side using `Posts.update()`, as we now have allow and deny rules, which make sure that only allowed data is modified.

Summary

In this chapter, we learned how to allow and deny database updates. We set up our own allow and deny rules and saw how methods can improve security by moving sensitive processes to the server side. We also improved our procedure of creating posts by checking whether the slug already exists and adding a simple progress indicator.

If you want to dig deeper into the allow and deny rules or methods, take a look at the following Meteor documentations:

- `http://docs.meteor.com/#/full/allow`
- `http://docs.meteor.com/#/full/deny`
- `https://docs.meteor.com/#/full/methods_header`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter8`.

In the next chapter, we will make our interface real time by constantly updating the post's timestamps.

9
Advanced Reactivity

Now our blog is basically complete, as we can create and edit entries. In this chapter, we will make use of Meteor's reactive templates to make our interface timestamps update itself. We will build a reactive object that will rerun the template helper, which displays the time when the blog entries were created. This way, they will always display the correct relative time.

In this chapter, we will cover the following topics:

- Reactive programming
- Rerunning functions manually
- Building a reactive object using the `Tracker` package
- Stopping reactive functions

If you've jumped right into the chapter and want to follow the examples, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter8`.

These code examples will also contain all the style files, so we don't have to worry about adding CSS code along the way.

Reactive programming

As we already saw throughout the book, Meteor uses something called **reactivity**.

One problem that a developer has to solve when building a software application is the consistency of the data represented in the interface. Most modern applications use something called **Model-View-Controller** (**MVC**), where the controller of a view makes sure that it always represents the current state of the model. The model is mostly a server API or a JSON object in the browser memory.

The most common ways of keeping consistent interfaces are as follows (courtesy: `http://manual.meteor.com`):

- **Poll and diff**: Periodically (for example, every second), fetch the current value of the thing, see whether it's changed, and if so, perform the update.
- **Events**: The thing that can change emits an event when it changes. Another part of the program (often called a controller) arranges to listen for this event, gets the current value, and performs the update when the event fires.
- **Bindings**: Values are represented by objects that implement some interface, such as `BindableValue`. Then, a "bind" method is used to tie two `BindableValues` together so that when one value changes, the other is updated automatically. Sometimes, as a part of setting up the binding, a transformation function can be specified. For example, `Foo` can be bound to `Bar` with the `toUpperCase` transformation function.

These patterns are good, but they still need a lot of code to maintain the consistency of the data represented.

Another pattern, although not yet as commonly used, is **reactive programming**. This pattern is a declarative way of binding data. It means when we use a reactive data source such as a `Session` variable or `Mongo.Collection`, we can be sure that reactive functions or template helpers that use these will rerun as soon as its value changes, always keeping the interface or calculations based on these values updated.

The Meteor manual gives us an example use case where reactive programming comes in handy:

> *Reactive programming is perfect for building user interfaces, because instead of attempting to model all interactions in a single piece of cohesive code, the programmer can express what should happen upon specific changes. The paradigm of responding to a change is simpler to understand than modeling which changes affect the state of the program explicitly.*
>
> *For example, suppose that we are writing an HTML5 app with a table of items, and the user can click on an item to select it or ctrl-click to select multiple items. We might have an <h1> tag and want the contents of the tag to be equal to the name of the currently selected item, capitalized, or else "Multiple selection" if multiple items are selected. And we might have a set of <tr> tags and want the CSS class on each <tr> tag to be "selected" if the items corresponding to that row is in the set of selected items, or the empty string otherwise.*

To make this example happen in the aforementioned patterns, we can quickly see how complex it gets compared to reactive programming (courtesy: `http://manual.meteor.com`):

- If we use poll and diff, the UI will be unacceptably laggy. After the user clicks, the screen won't actually update until the next polling cycle. Also, we have to store the old selection set and diff it against the new selection set, which is a bit of a hassle.
- If we use events, we have to write some fairly tangled controller code to manually map changes to the selection or to the name of the selected item, onto updates to the UI. For example, when the selection changes, we have to remember to update both the `<h1>` tag and (typically) two affected `<tr>` tags. What's more, when the selection changes, we have to automatically register an event handler on the newly selected item so that we can remember to update `<h1>`. It is difficult to structure clean code and maintain it, especially as the UI is extended and redesigned.
- If we use bindings, we will have to use a complex **domain-specific language** (**DSL**) to express the complex relationships between the variables. The DSL will have to include indirection (bind the contents of `<h1>` not to the name of any fixed item, but to the item indicated by the current selection), transformation (capitalize the name), and conditionals (if more than one item is selected, show a placeholder string).

With Meteor's reactive template engine, Blaze, we can simply use the `{{#each}}` block helper to iterate over a list of elements and add some conditions for each element based on user interaction or on an item's property to add a selected class.

If the user now changes the data or the data coming in from the server changes, the interface will update itself to represent the data accordingly, saving us a lot of time and avoiding unnecessary complex code.

The invalidating cycle

One key part of understanding the reactive dependencies is the invalidate cycle.

When we use a reactive data source inside a reactive function, such as `Tracker.autorun(function(){…})`, the reactive data source itself sees that it is inside a reactive function and adds the current function as a dependency to its dependency store.

Then, when the value of the data source changes, it invalidates (reruns) all its dependent functions and removes them from its dependency store.

In the rerun of the reactive function, it adds the reactive function back to its dependency store so that they will rerun on its next invalidation (value change) again.

This is the key to understand the reactive concept, as we will see in the following example.

Imagine that we have two `Session` variables set to `false`:

```
Session.set('first', false);
Session.set('second', false);
```

Moreover, We have the `Tracker.autorun()` function, which uses both these variables:

```
Tracker.autorun(function(){
    console.log('Reactive function re-run');
    if(Session.get('first')){
        Session.get('second');
    }
});
```

We can now call `Session.set('second', true)`, but the reactive function will not rerun, because it was never called in the first run, as the `first` session variable was set to `false`.

If we now call `Session.set(first, true)`, the function will rerun.

Additionally, if we now set `Session.set('second', false)`, it will rerun as well, as in the second rerun, `Session.get('second')` can add this reactive function as a dependency.

Because the reactive data sources source will always remove all dependencies from its store on every invalidation and add them back in the rerun of the reactive function, we can set `Session.set(first, false)` and try to switch it to `Session.set('second', true)`. The function will *not* rerun again, as `Session.get('second')` was never called in this run!

Once we understand this, we can make more fine-grained reactivity, keeping reactive updates to a minimum. The console output of the explanation looks similar to the following screenshot:

Building a simple reactive object

As we saw, a **reactive object** is an object that when used inside a reactive function, will rerun the function when its value changes. The Meteor's `Session` object is one example of a reactive object.

In this chapter, we will build a simple reactive object that will rerun our `{{formatTime}}` template helper at time intervals so that all the relative times are updated correctly.

Meteor's reactivity is made possible through the `Tracker` package. This package is the core of all reactivity and allows us to track dependencies and rerun these whenever we want.

Perform the following steps to build a simple reactive object:

1. To get started, let's add the following code to the `my-meteor-blog/main.js` file:

   ```
   if(Meteor.isClient) {
       ReactiveTimer = new Tracker.Dependency;
   }
   ```

   This will create a variable named `ReactiveTimer` on the client with a new instance of `Tracker.Dependency`.

2. Below the `ReactiveTimer` variable, but still inside the `if(Meteor.isClient)` condition, we will add the following code to rerun all dependencies of our `ReactiveTimer` object every 10 seconds:

   ```
   Meteor.setInterval(function(){
       // re-run dependencies every 10s
       ReactiveTimer.changed();
   }, 10000);
   ```

   The `Meteor.setInterval` will run the function every 10 seconds.

> Meteor comes with its own implementation of `setInterval` and `setTimeout`. Even though they work exactly as their native JavaScript equivalents, Meteor needs these to reference the right timeout/interval for a specific user on the server side.

Meteor comes with its own implementation of `setInterval` and `setTimeout`. Even though they work exactly as their native JavaScript equivalents, Meteor needs these to reference the right timeout/interval for a specific user on the server side.

Inside the interval, we call `ReactiveTimer.changed()`. This will invalidate every dependent function, causing it to rerun.

Rerunning functions

So far, we have no dependency created, so let's do that. Add the following code below `Meteor.setInterval`:

```
Tracker.autorun(function(){
    ReactiveTimer.depend();
    console.log('Function re-run');
});
```

If we now get back to our browser console, we should see **Function re-run** every 10 seconds, as our reactive object reruns the function.

We can even call `ReactiveTimer.changed()` in our browser console and the function will rerun as well.

These are good examples, but don't make our timestamps update automatically.

To do this, we need to open up `my-meteor-blog/client/template-helpers.js` and add the following line at the top of our `formatTime` helper function:

```
ReactiveTimer.depend();
```

This will make every `{{formatTime}}` helper in our app rerun every 10 seconds, updating the relative time while it passes. To see this, go to your browser and create a new blog entry. If you save the blog entry now and watch the time created text, you will see that it changes after a while:

Creating an advanced timer object

The previous example was a simple demonstration of a custom reactive object. To make it more useful, it is better to create a separate object that hides the `Tracker.Dependency` functions and adds additional functionality.

Meteor's reactivity and dependency tracking allows us to create dependencies even when the `depend()` function is called from inside another function. This dependency chain allows more complex reactive objects.

In the next example, we will take our `timer` object and add a `start` and `stop` function to it. Additionally, we will also make it possible to choose a time interval at which the timer will rerun:

1. First, let's remove the previous code examples from the `main.js` and `template-helpers.js` files, which we added before, and create a new file named `ReactiveTimer.js` inside `my-meteor-blog/client` with the following content:

   ```
   ReactiveTimer = (function () {

       // Constructor
       function ReactiveTimer() {
           this._dependency = new Tracker.Dependency;
           this._intervalId = null;
       };

       return ReactiveTimer;
   })();
   ```

   This creates a classic prototype class in JavaScript, which we can instantiate using `new ReactiveTimer()`. In its constructor function, we instantiate a `new Tracker.Dependency` and attach it to the function.

2. Now, we will create a `start()` function, which will start a self-chosen interval:

   ```
   ReactiveTimer = (function () {

       // Constructor
       function ReactiveTimer() {
           this._dependency = new Tracker.Dependency;
           this._intervalId = null;
       };
   ```

```
    ReactiveTimer.prototype.start = function(interval){
        var _this = this;
        this._intervalId = Meteor.setInterval(function(){
            // rerun every "interval"
            _this._dependency.changed();
        }, 1000 * interval);
    };

    return ReactiveTimer;
})();
```

This is the same code as we used before with the difference that we store the interval ID in `this._intervalId` so that we can stop it later in our `stop()` function. The interval passed to the `start()` function must be in seconds;

3. Next, we add the `stop()` function to the class, which will simply clear the interval:

```
ReactiveTimer.prototype.stop = function(){
    Meteor.clearInterval(this._intervalId);
};
```

4. Now we only need a function that creates the dependencies:

```
ReactiveTimer.prototype.tick = function(){
    this._dependency.depend();
};
```

Our reactive timer is ready!

5. Now, to instantiate the `timer` and start it with whatever interval we like, add the following code after the `ReactiveTimer` class at the end of the file:

```
timer = new ReactiveTimer();
timer.start(10);
```

6. At last, we need to go back to our `{{formatTime}}` helper in the `template-helper.js` file, and `add` the `time.tick()` function, and every relative time in the interface will update as time goes by.
7. To see the reactive timer in action, run the following code snippet in our browser's console:

```
Tracker.autorun(function(){
    timer.tick();
    console.log('Timer ticked!');
});
```

8. We should now see **Timer ticked!** logged every 10 seconds. If we now run `time.stop()`, the timer will stop running its dependent functions. If we call `time.start(2)` again, we will see Timer ticked! now appearing every two seconds, as we set the interval to `2`:

As we can see, our `timer` object is now rather flexible, and we can create any number of time intervals to be used throughout the app.

Reactive computations

Meteor's reactivity and the `Tracker` package is a very powerful feature, as it allows event-like behavior to be attached to every function and every template helper. This reactivity is what keeps our interface consistent.

Although we only touched the `Tracker` package until now, it has a few more properties that we should take a look at.

We already learned how to instantiate a reactive object. We can call `new Tracker.Dependency`, which can create and rerun dependencies using `depend()` and `changed()`.

Stopping reactive functions

When we are inside a reactive function, we also have access to the current computational object, which we can use to stop further reactive behavior.

To see this in action, we can use our already running `timer` and create the following reactive function using `Tracker.autorun()` in our browser's console:

```
var count = 0;
var someInnerFunction = function(count){
    console.log('Running for the '+ count +' time');

    if(count === 10)
        Tracker.currentComputation.stop();
};
Tracker.autorun(function(c){
    timer.tick();

    someInnerFunction(count);

    count++;
});

timer.stop();
timer.start(2);
```

Here, we create `someInnerFunction()` to show how we can access the current computation as well from nested functions. In this inner function, we get the computation using `Tracker.currentComputation`, which gives us the current `Tracker.Computation` object.

We use the `count` variable, we created before the `Tracker.autorun()` function, to count up. When we reach 10, we call `Tracker.currentComputation.stop()`, which will stop the dependency of the inner and the `Tracker.autorun()` functions, making them nonreactive.

To see the results quicker, we stop and start the `timer` object with an interval of two seconds at the end of the example.

If we copy and paste the previous code snippet into our browser's console and run it, we should see **Running for the xx time** appearing 10 times:

```
<top frame>
timer.stop();
timer.start(2);
Running for the 0 time
undefined
Running for the 1 time
Running for the 2 time
Running for the 3 time
Running for the 4 time
Running for the 5 time
Running for the 6 time
Running for the 7 time
Running for the 8 time
Running for the 9 time
Running for the 10 time
>
```

The current computational object is useful to give us control over reactive dependencies from inside the dependent functions.

Preventing run at start

The `Tracker.Computation` object also comes with the `firstRun` property, which we have used in an earlier chapter.

Reactive functions, for example, when created using `Tracker.autorun()` also run when they are parsed by JavaScript for the first time. If we want to prevent this, we can simply stop the function before any code is executed when checking whether `firstRun` is `true`:

```
Tracker.autorun(function(c){
    timer.tick();

    if(c.firstRun)
        return;

    // Do some other stuff
});
```

We don't need to get the current computation here using `Tracker.currentComputation`, as `Tracker.autorun()` gets it already as its first argument.

Also, when we stop a `Tracker.autorun()` function, as described in the following code, it will never create the dependency for the session variable, as `Session.get()` was never called in the first run:

```
Tracker.autorun(function(c){
  if(c.firstRun)
    return;

  Session.get('myValue');
}):
```

To make sure that we make the function depending on the `myValue` session variable, we need to put it before the `return` statement.

Advanced reactive objects

The `Tracker` package has a few more advanced properties and functions that allow you to control when dependencies are invalidated (`Tracker.flush()` and `Tracker.Computation.invalidate()`) and allow you to register additional callbacks on it (`Tracker.onInvalidate()`).

These properties allow you to build complex reactive objects, which are out of the scope of this book. If you want to get a deeper understanding of the `Tracker` package, I recommend that you take a look at the Meteor manual at `http://manual.meteor.com/#tracker`.

Summary

In this chapter, we learned how to build our own custom reactive object. We learned about `Tracker.Dependency.depend()` and `Tracker.Dependency.changed()` and saw how reactive dependencies have their own computational objects, which can be used to stop its reactive behavior and prevent running at start.

To dig deeper, take a look at the documentation for the `Tracker` package and see detailed property descriptions for the `Tracker.Computation` object at the following resources:

- `https://www.meteor.com/tracker`
- `https://docs.meteor.com/#/full/tracker`
- `https://docs.meteor.com/#/full/tracker_computation`
- `https://docs.meteor.com/#/full/tracker_dependency`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter9`.

Now that we have finalized our blog, we will take a look at how to deploy our app on servers in the next chapter.

10
Deploying Our App

Our app is now ready to be deployed. In this chapter, we will see how we can deploy our app on different servers to make it public and show the world what we built.

Meteor makes it easy to deploy applications on its own server infrastructure. It's free and quick to do, but probably not the right place for a production environment. Therefore, we will take a look at manual deployment as well as some great tools built to deploy on any Node.js server.

In this chapter, we will cover the following topics:

- Registering a Meteor developer account
- Deploying on Meteor's own server infrastructure
- Bundling and deploying Meteor manually
- Deploying using Demeteorizer
- Deploying using Meteor Up

If you want to have the full app we've built in this book to deploy, download the code from the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter10`.

This code won't have the part where dummy posts are created, so you can have a clean blog to start with on your own server.

Deploying on meteor.com

Meteor provides its own hosting environment, where everybody can deploy apps with a single command, for free. In order to deploy apps, Meteor creates a developer account for us so that we can manage and deploy our apps later. To start, let's perform the following steps to deploy our app on `meteor.com`:

1. Deploying on a subdomain of meteor.com is as simple as running the following command in the terminal from our app's folder:

   ```
   $ meteor deploy myCoolNewBlog
   ```

   We can freely choose the subdomain we want to deploy on. If `myCoolNewBlog.meteor.com` is already taken, Meteor will ask us to log in to the owner's account to overwrite the currently deployed app, or we will have to choose another name.

2. If the domain name is free, Meteor will ask us to provide an e-mail address so that it can create a developer account for us. After entering the e-mail address, we will receive an e-mail with a link to set up our Meteor Developer account, as shown in the following screenshot:

3. To create our account, we need to follow the link given by Meteor so that we can fully set up our account by adding a username and a password, as shown in the next screenshot:

4. After we have done that, we get access to our developer account's page, where we can add e-mail addresses, check our last login, and authorize other Meteor developers to log in to our apps (though we have to add the `accounts-meteor-developer` package first).
5. Now, to finally deploy our app, we need to log in with our Meteor Developer account in the terminal by using `$ meteor login`, entering our credentials, and running the `deploy` command again:

```
$ meteor deploy myCoolNewBlog
```

6. Using the `$ meteor authorized -add <username>` command, we can allow other Meteor developers to deploy to our app's subdomain, as shown in the following screenshot:

7. If we want to update our deployed app, we can simply run `$ meteor deploy` from inside our app's folder. Meteor will ask us for our credentials, and we can then deploy our app.

If we're on a friend's computer and want to use our Meteor account, we can do so using `$ meteor login`. Meteor will keep us logged in and everybody can redeploy any of our apps. We need to make sure we use `$ meteor logout` when we're finished.

Deploying on meteor.com using a domain name

We can also host our app on `meteor.com`, but can define our own domain name.

To do this, we simply deploy using our domain name, as follows:

```
$ meteor deploy mydomain.com
```

This will host the app on meteor.com, but with no direct URL such as `myapp.meteor.com`.

To point our domain to the app on the Meteor servers, we need to change the **A record** of our domain to the IP address of `origin.meteor.com` (which is `107.22.210.133` at the time of writing this book), or the `CNAME` record to `origin.meteor.com`. You can do this at the provider where you registered your domain under the DNS configuration.

Meteor will then get a request from our domain and redirect internally to the server where our app is located.

Backup and restore databases hosted on meteor.com

If you ever need to back up your database or move it to another server, you can get temporary Mongo database credentials for the deployed database using the following command:

```
$ meteor mongo myapp.meteor.com –url
```

This will get something like the following credentials:

```
mongodb://client-ID:xyz@production-db-b1.meteor.io:27017/yourapp_meteor_com
```

You can then use the credentials from the preceding output to back up your database using `mongodump`:

```
$ mongodump -h production-db-b1.meteor.io --port 27017 --username client-ID --password xyz --db yourapp_meteor_com
```

This will create a folder named `dump/yourapp_meteor_com` where you are and put the dump files of the database inside.

To restore it to another server, use `mongorestore`, with the last argument being the folder where you put the database dump:

```
$ mongorestore -h mymongoserver.com --port 27017 --username myuser
--password xyz --db my_new_database dump/yourapp_meteor_com
```

If you simply want to put the data into your local Meteor app's database, start the Meteor server using `$ meteor` and run the following command:

```
$ mongorestore --port 3001
```

Deploying on other servers

Meteor's free hosting is great, but when it comes to using an app in production, we want to be in control of the server we're using.

Meteor allows us to create a bundle of our application so that we can deploy it on any Node.js server. The only downside to this is that we need to install certain dependencies ourselves. Additionally, there are two packages out there that make deploying apps almost as simple as Meteor itself, though their configuration is still needed.

Bundling our app

In order to deploy our app on our own server, we need to have a Linux server with the latest version of Node.js and NPM installed. The server should have the same platform as our local machine on which we will create the bundle. If you want to deploy your app on another platform, take a look at the next section. Now let's build the app by performing the following steps:

1. If our server fits the aforementioned requirements, we can go to our app's folder on our local machine and run the following command:

   ```
   $ meteor build myAppBuildFolder
   ```

2. This will create `myAppBuildFolder` with a `*.tar.gz` file inside. We can then upload this file to our server and extract it under `~/Sites/myApp` for example. Then we go to the extracted folder and run the following commands:

   ```
   $ cd programs/server
   $ npm install
   ```

3. This will install all the NPM dependencies. After they're installed, we set the necessary environment variables:

```
$ export MONGO_URL='mongodb://user:password@host:port/
databasename'
$ export ROOT_URL='http://example.com'
$ export MAIL_URL='smtp://user:password@mailhost:port/'
$ export PORT=8080
```

   The `export` commands will set the `MONGO_URL`, `ROOT_URL`, and `MAIL_URL` environment variables.

4. As this manual deployment doesn't come with preinstalled MongoDB, we need to either install it on our machine or use a hosted service such as Compose (`http://mongohq.com`). If we rather want to install MongoDB on our server ourselves, we can follow the guide at `http://docs.mongodb.org/manual/installation`.

5. The `ROOT_URL` variable should be the URL of the domain pointing to our server. If our app sends e-mails, we can additionally set our own SMTP server or use a service such as Mailgun (`http://mailgun.com`) and change the SMTP host in the `MAIL_URL` variable.

   We can also specify the port on which we want our app to run using the `PORT` environment variable. If we don't set the `PORT` variable, it will use port `80` by default.

6. After we set these variables, we go to the root folder of our app and start the server using the following command:

```
$ node main.js
```

> If you want to make sure your application is restarted in case it crashes or when the server is rebooted, take a look at the `forever` NPM package, which is explained at `https://github.com/nodejitsu/forever`.

If everything goes fine, our app should be reachable at `<your server's ip>:8080`.

In case we run into trouble by manually deploying our app, we can use the next approach.

Deploying using Demeteorizer

The disadvantage of using `$ meteor build` is that most node modules are already compiled, and therefore can cause problems in the server's environment. Hence comes Demeteorizer, which is very similar to `$ meteor build` but will additionally unpack the bundle and create a `package.json` file with all the node dependencies and the correct node version for the project. Here is how we deploy using Demeteorizer:

1. Demeteorizer comes as an NPM package, which we can install using the following command:

   ```
   $ npm install -g demeteorizer
   ```

   [If the `npm` folder doesn't have the right permissions, use `sudo` before the command.]

2. Now we can go to our app's folder and type the following command:

   ```
   $ demeteorizer -o ~/my-meteor-blog-converted
   ```

3. This will output the ready-to-distribute app to the `my-meteor-blog-converted` folder. We just copy this folder to our server, set the same environment variables as described earlier, and run the following command:

   ```
   $ cd /my/server/my-meteor-blog-converted
   $ npm install
   $ node main.js
   ```

This should start our app on the port we specified.

Deploying using Meteor Up

The previous steps help us to deploy our app on our own server, but this method still requires us to build, upload, and set the environment variables.

Meteor Up (**mup**) aims to make deploying as easy as running `$ meteor deploy`. However, if we want to use Meteor Up, we need to have full admin rights on the server.

Additionally, this allows us to auto-restart the app in case it crashes, using the `forever` NPM package, as well as start the app when the server reboots, using the `upstart` NPM package. We can also revert to the previously deployed version, which gives us a good basis for deployment on the production environment.

The next steps are for more advanced developers, as they require setting up `sudo` rights on the server machine. Therefore, if you're inexperienced in deployment, consider using a service such as Modulus (`http://modulus.io`), which offers online Meteor deployment using its own command-line tool, available at `https://modulus.io/codex/meteor_apps`.

Meteor Up will set up the server and deploy our app as follows:

1. To install `mup` on our local machine, we type the following command:

   ```
   $ npm install -g mup
   ```

2. Now we need to create a folder for our deployment configuration, which could be in the same folder where our app is located:

   ```
   $ mkdir ~/my-meteor-blog-deployment
   $ cd ~/my-meteor-blog-deployment
   $ mup init
   ```

3. Meteor Up creates a configuration file for us, which will look like the following:

   ```
   {
     "servers": [
       {
         "host": "hostname",
         "username": "root",
         "password": "password"
         // or pem file (ssh based authentication)
         //"pem": "~/.ssh/id_rsa"
       }
     ],
     "setupMongo": true,
     "setupNode": true,
     "nodeVersion": "0.10.26",
     "setupPhantom": true,
     "appName": "meteor",
     "app": "/Users/arunoda/Meteor/my-app",
     "env": {
       "PORT": 80,
       "ROOT_URL": "http://myapp.com",
       "MONGO_URL": "mongodb://arunoda:fd8dsjsfh7@hanso.mongohq.
   com:10023/MyApp",
       "MAIL_URL": "smtp://postmaster%40myapp.mailgun.
   org:adj87sjhd7s@smtp.mailgun.org:587/"
     },
     "deployCheckWaitTime": 15
   }
   ```

4. Now we can edit this file to work for our server environment.
5. First, we will add the SSH server authentication. We can provide either our RSA key file or a username and a password. If we want to use the latter, we need to install `sshpass`, a tool used to provide SSH passwords without using the command line:

```
"servers": [
    {
      "host": "myServer.com",
      "username": "johndoe",
      "password": "xyz"
      // or pem file (ssh based authentication)
      //"pem": "~/.ssh/id_rsa"
    }
],
```

To install `sshpass` for our environment, we can follow the steps at `https://gist.github.com/arunoda/7790979`, or if you're on Mac OS X, take a look at `http://www.hashbangcode.com/blog/installing-sshpass-osx-mavericks`.

6. Next, we can set some options, such as choosing to install MongoDB on the server. If we use a service such as Compose, we will set it to `false`:

```
"setupMongo": false,
```

If we already have Node.js installed on our server, we will also set the next option to `false`:

```
"setupNode": false,
```

If we want to mention a specific Node.js version, we can set it as follows:

```
"nodeVersion": "0.10.25",
```

Meteor Up can also install PhantomJS for us, which is necessary if we use Meteor's spiderable package, which makes our app crawlable by search engines:

```
"setupPhantom": true,
```

In the next option, we will set the name of our app, which can be the same as our app's folder name:

```
"appName": "my-meteor-blog",
```

Finally, we point to our local app folder so that Meteor Up knows what to deploy:

```
"app": "~/my-meteor-blog",
```

7. Meteor Up also lets us preset all the necessary environment variables, such as the correct `MONGO_URL` variable:

```
"env": {
    "ROOT_URL": "http://myServer.com",
    "MONGO_URL": "mongodb://user:password@host:port/databasename",
    "PORT": 8080
},
```

8. The last option sets the time Meteor Up will wait for before checking whether the app started successfully:

```
"deployCheckWaitTime": 15
```

Setting up the server

In order to set up the server using Meteor Up, we need a no-password access to `sudo`. Perform the following steps to set up the server:

1. To enable no-password access, we need to add our current user to the server's `sudo` group:

```
$ sudo adduser <username> sudo
```

2. Then add `NOPASSWD` to the `sudoers` file:

```
$ sudo visudo
```

3. Now replace the `%sudo ALL=(ALL) ALL` line with the following line:

```
%sudo ALL=(ALL) NOPASSWD:ALL
```

Deploying with mup

If everything has worked fine, we can set up our server. The following steps explain how we can deploy with `mup`:

1. Run the following command from inside the local `my-meteor-blog-deployment` folder:

```
$ mup setup
```

   This will configure our server and install all requirements chosen in our configuration file.

   Once this process is done, we can deploy our app any time by running the following command from the same folder:

```
$ mup deploy
```

This way, we can also create production and staging environments by creating two separate Meteor Up configurations with two distinct app names, and deploy it to the same server.

Outlook

Currently, Meteor limits native deployment to its own servers, with limited control over the environment. Planned is an enterprise-grade server infrastructure called **Galaxy**, which will make deploying and scaling Meteor apps as simple as Meteor itself.

Nonetheless, with Meteor's simplicity and great community, we already have a rich set of tools available to deploy to any Node.js-based hosting and PaaS environment.

For example, if we wanted to deploy on Heroku, we can take a look at the build pack by Jordan Sissel at `https://github.com/jordansissel/heroku-buildpack-meteor`.

Summary

In this chapter, we learned how to deploy Meteor and how simple deploying on Meteor's own server infrastructure can be. We also used tools such as Demeteorizer and Meteor Up to deploy on our own server infrastructure.

To read more about the specific deployment methods, take a look at the following resources:

- `https://www.meteor.com/services/developer-accounts`
- `https://docs.meteor.com/#/full/deploying`
- `https://www.meteor.com/services/build`
- `https://github.com/onmodulus/demeteorizer`
- `https://github.com/arunoda/meteor-up`

You can find the full example code of this app, ready for deployment, at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter10`.

In the next chapter, we will create a package of our previously created `ReactiveTimer` object and publish it to Meteor's official package repository.

11
Building Our Own Package

In this chapter, we will learn how to build our own package. Writing packages allows us to create closed-functionality components that can be shared between many apps. In the second half of the chapter, we will publish our app on Atmosphere, Meteor's third-party package repository, at `https://atmospherejs.com`.

In this chapter, we will cover the following topics:

- Structuring a package
- Creating a package
- Publishing your own package

In this chapter, we will package the `ReactiveTimer` object that we built in *Chapter 9*, *Advanced Reactivity*. To follow the examples in this chapter, download the previous chapter's code examples from either the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter10`.

The structure of a package

A package is a bundle of JavaScript files that exposes only specific variables to a Meteor app. Other than in a Meteor app, package files will get loaded in the loading order we specify.

Every package needs a `package.js` file that contains the configuration of that package. In such a file, we can add a name, description, and version, set the loading order, and determine which variables should be exposed to the app. Additionally, we can specify unit tests for our packages to test them.

An example of a `package.js` file can look like this:

```
Package.describe({
  name: "mrt:moment",
  summary: "Moment.js, a JavaScript date library.",
  version: "0.0.1",
  git: "https://..."
});

Package.onUse(function (api, where) {
  api.export('moment');

  api.addFiles('lib/moment-with-langs.min.js', 'client');
});

Package.onTest(function(api){
  api.use(["mrt:moment", "tinytest"], ["client", "server"]);
  api.addFiles("test/tests.js", ["client", "server"]);
});
```

We can structure the files and folders in our package as we wish, but a good basis is the following arrangement:

- `tests`: This contains the package's unit tests and the `tests.js` file
- `lib`: This contains third-party libraries used by the package
- `README.md`: This contains simple instructions on how to use the package
- `package.js`: This contains the package's metadata
- `myPackage.js`: These are one or more files that contain the package code

To test a package, we can use Meteor's `tinytest` package, which is a simple unit testing package. If we have tests, we can run them using the following command:

```
$ meteor test-packages <my package name>
```

This will start a Meteor app at `http://localhost:3000`, which runs our package tests. To see how to write a package, take a look at the next chapter.

Creating our own package

To create our own package, we will use our `ReactiveTimer` object, which we built in *Chapter 9*, *Advanced Reactivity*:

1. We go to our terminal, in our app's folder and run the following command:

   ```
   $ meteor create --package reactive-timer
   ```

2. This will create a folder named `packages` with a `reactive-timer` folder inside it. Inside the `reactive-timer` folder, Meteor has already created a `package.js` file and some example package files.
3. Now we can delete all the files inside the `reactive-timer` folder, except the `package.js` file.
4. Then we move the `my-meteor-blog/client/ReactiveTimer.js` file, which we created in *Chapter 9*, *Advanced Reactivity*, to our newly created `reactive-timer` package folder.
5. Lastly, we open the copied `ReactiveTimer.js` file and remove the following lines:

   ```
   timer = new ReactiveTimer();
   timer.start(10);
   ```

   Later, we'll instantiate the `timer` object inside the app itself and not in the package file.

We should now have a simple folder with the default `package.js` file and our `ReactiveTimer.js` file. This is almost it! We just need to configure our package and we are ready to use it in our app.

Adding the package metadata

To add the package's metadata, we open the file called `package.js` and add the following lines of code:

```
Package.describe({
  name: "meteor-book:reactive-timer",
  summary: "A simple timer object, which can re-run reactive functions
based on an interval",
  version: "0.0.1",
  // optional
  git: "https://github.com/frozeman/meteor-reactive-timer"
});
```

This adds a name to the package as well as a description and a version.

Note that the package name is namespaced with the author's name. This exists so that packages with the same name can be made distinct through the names of their authors. In our case, we choose `meteor-book`, which is not a real username. To publish the package, we need to use our real Meteor developer username.

After the `Package.describe()` function come the actual package dependencies:

```
Package.onUse(function (api) {
  // requires Meteor core packages 1.0
  api.versionsFrom('METEOR@1.0');

  // we require the Meteor core tracker package
  api.use('tracker', 'client');

  // and export the ReactiveTimer variable
  api.export('ReactiveTimer');

  // which we find in this file
  api.addFiles('ReactiveTimer.js', 'client');
});
```

Here, we define the version of the Meteor core packages this package should use:

- With `api.use()`, we define an additional package (or packages) this package depends on. Note that these dependencies won't be accessible to the app itself, which uses this package.

> Additionally, there exists `api.imply()`, which not only makes another package available in the package's files, but also adds it to the Meteor app itself so that it can be accessed by the app's code.

- If we use a third-party package, we must specify the minimum package version as follows:

  ```
  api.use('author:somePackage@1.0.0', 'server');
  ```

> We can also pass in a third parameter, `{weak: true}`, to specify that the dependent package will only be used if it is already added to the app by the developer. This can be used to enhance a package when other packages are present.

- In the second parameter of the `api.use()` function, we can specify whether to load it on the client, server, or both, using an array:

  ```
  api.use('tracker', ['client', 'server']);
  ```

> We don't really need to import the `Tracker` package, as it's already a part of Meteor's core `meteor-platform` package (added by default to any Meteor app); we do this here for the sake of an example.

- We then use `api.export('ReactiveTimer')` to define which variable of the package should be exposed to the Meteor app using this package. Remember that we created the `ReactiveTimer` object inside the `ReactiveTimer.js` file using the following lines of code:

  ```
  ReactiveTimer = (function () {
    ...
  })();
  ```

> Note that we didn't use `var` to create the variable. This way, it is accessible in all the other files of the package and can also be exposed to the app itself.

- Lastly, we tell the package system which files belong to the package, using `api.addFiles()`. We can have multiple calls of `api.addFiles()` one after the other. This order will then specify the loading order of the files.

  Here, we can again tell Meteor where to load the file—on the client, the server, or both—using `['client', 'server']`.

  In this case, we only provide the `ReactiveTimer` object on the client, as Meteor's reactive functions exist only on the client side.

> If you want to see a full list of methods on the `api` object, take a look at Meteor's documentation at `http://docs.meteor.com/#packagejs`.

Adding the package

Copying a package folder to the `my-meteor-blog/packages` folder is not enough to tell Meteor to use the package. There are additional steps that we need to follow:

1. To add the package, we need to go to our app's folder from the terminal, quit any currently running `meteor` instance, and run the following command:

   ```
   $ meteor add meteor-book:reactive-timer
   ```

2. We then need to instantiate the `ReactiveTimer` object in our app. To do this, we add the following lines of code to our `my-meteor-blog/main.js` file:

   ```
   if(Meteor.isClient) {
       timer = new ReactiveTimer();
       timer.start(10);
   }
   ```

3. Now we can start the Meteor app again using `$ meteor` and open our browser at `http://localhost:3000`.

We shouldn't see any difference, as we just replaced the `ReactiveTimer` object that was already there in our app with the `ReactiveTimer` object from our `meteor-book:reactive-timer` package.

To see the timer run, we can open our browser's console and run the following code snippet:

```
Tracker.autorun(function(){
    timer.tick();
    console.log('timer run');
});
```

This should log `timer run` every 10 seconds, showing us that the package is actually working.

Releasing our package to the public

It's very easy to release a package to the world, but for people to use our package, we should add a readme file so they can know how to use our package.

Create a file called `README.md` in the package folder we created earlier and add the following code snippet:

```
# ReactiveTimer

This package can run reactive functions in a given interval.
```

```
## Installation

    $ meteor add meteor-book:reactive-timer

## Usage

To use the timer, instantiate a new interval:

    var myTimer = new ReactiveTimer();

Then you can start an interval of 10 seconds using:

    myTimer.start(10);

To use the timer just call the following in any reactive function:

    myTimer.tick();

To stop the timer use:

    myTimer.stop();
```

As we can see, this file uses the markdown syntax. This way, it will look good on GitHub and `http://atmospherejs.com`, which is the website where you can browse all the available Meteor packages.

With this readme file, we will make it easy for other people to use the package and appreciate our work.

Publishing our package online

After we have saved the readme file, we can push the package to GitHub or any other online Git repository, and add the repository's URL to the `Package.describe({git: …})` variable of `package.js`. Keeping the code on GitHub keeps it safe and allows others to fork and improve it. Let's perform the following steps to push our package online:

1. To publish our package, we can simply run the following command from inside the `pages` folder in the terminal:

   ```
   $ meteor publish --create
   ```

   This will build and bundle the package and upload it to Meteor's package servers.

2. If everything goes fine, we should be able to find our package by typing the following command:

   ```
   $ meteor search reactive-timer
   ```

   This is illustrated in the following screenshot:

3. We can then show all of the information about the found package using the following command:

   ```
   $ meteor show meteor-book:reactive-timer
   ```

   This is illustrated in the following screenshot:

4. To use the package version from the Meteor server, we can simply move the `packages/reactive-timer` folder somewhere else, remove the `package` folder, and run `$ meteor` to start the app.

   Now Meteor won't find any package with that name in the `packages` folder and will look online for that package. Since we published it, it will be downloaded and used in our app.

5. Should we want to use a specific version of our package in the app, we can run the following command from inside our app's folder in the terminal:

   ```
   $ meteor add meteor-book:reactive-timer@=0.0.1
   ```

Now our package is released and we can see it on Atmosphere at `http://atmospherejs.com/meteor-book/reactive-timer`, as shown in the following screenshot:

Note that this is just an example of a package and was never actually released. However, a published version of this package under my name can be found at `http://atmospherejs.com/frozeman/reactive-timer`.

Updating our package

If we want to release a new version of our package, we can simply increase the version number in the `package.js` file and publish a new version using the following command from inside the `packages` folder:

```
$ meteor publish
```

To make our app use the latest version of our package (as long as we didn't specify a fixed version), we can simply run the following command from inside our app's folder:

```
$ meteor update meteor-book:reactive-timer
```

If we want to update all packages, we can run the following command:

```
$ meteor update --packages-only
```

Summary

In this chapter, we created our own package from our `ReactiveTimer` object. We also learned how simple it is to publish a package on Meteor's official packaging system.

To dig deeper, read the documentations at the following resources:

- `https://docs.meteor.com/#/full/writingpackages`
- `https://docs.meteor.com/#packagejs`
- `https://www.meteor.com/services/package-server`
- `https://www.meteor.com/isobuild`

You can find this chapter's code examples at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter11`.

This code example contains only the package, so in order to add it to the app, use the code example of the previous chapter.

In the next chapter, we will take a look at testing our app and package.

12
Testing in Meteor

In this final chapter, we will discuss how we can test a Meteor app.

Testing is a comprehensive topic and it goes beyond the scope of this chapter. To keep it simple, we will briefly cover two tools available, as they are certainly different, and show a simple example for each.

In this chapter, we will cover the following topics:

- Testing the `reactive-timer` package
- Using Jasmine to conduct unit tests on our app
- Using Nightwatch to conduct acceptance tests on our app

If you want to jump right into the chapter and follow the examples, download the code of *Chapter 10*, *Deploying Our App*, which contains the finished example app, either from the book's web page at `https://www.packtpub.com/books/content/support/17713` or from the GitHub repository at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter10`.

Types of tests

Tests are pieces of code that test other pieces of code or functionality of an app.

We can divide tests into four general groups:

- **Unit test**: In this test, we test only a small unit of our code. This can, for example, be a function or a piece of code. Unit tests should not call other functions, write to the hard disk or database, or access the network. If such functionality is needed, one should write stubs, which are functions that return the expected value without calling the real function.

- **Integrations test**: In this test, we combine multiple tests and run them in different environments to make sure that they still work. The difference in this test compared to the unit test is that we are actually running connected functionalities, such as calling the database.
- **Functional test**: This can be a unit test or tests in the interface, but will only test the functionality of a feature/function without checking for side effects, such as whether or not variables were cleaned up properly.
- **Acceptance test**: This runs tests on the full system, which can, for example, be a web browser. The idea is to mimic the actual user as much as possible. These tests are very similar to user stories that define a feature. The downside is that they make it hard to track down bugs, as the test occurs on a higher level.

In the following examples, we will mostly write functional tests for simplicity.

Testing packages

In the previous chapter, we built a package out of the `ReactiveTimer` object. A good package should always contain unit tests so that people can run them and be sure that changes to that package don't break its functionality.

Meteor provides a simple unit test tool for packages, called `TinyTest`, which we will use to test our package:

1. To add tests, we need to copy the `meteor-book:reactive-timer` package, which we built in the previous chapter, to the `my-meteor-blog/packages` folder of our app. This way, we can make changes to the package, as Meteor will prefer the package in the `packages` folder over one in its package servers. If you removed the package, simply add it back using the following command:

   ```
   $ meteor add meteor-book:reactive-timer
   ```

   Additionally, we need to make sure we delete the `my-meteor-blog/client/ReactiveTimer.js` file, which we should have if we used the code example from *Chapter 10*, *Deploying Our App*, as a basis.

2. Then we open the `package.js` file from our `packages` folder and add the following lines of code to the end of the file:

   ```
   Package.onTest(function (api) {
     api.use('meteor-book:reactive-timer', 'client');
   ```

```
  api.use('tinytest', 'client');

  api.addFiles('tests/tests.js', 'client');
});
```

This will include our `meteor-book:reactive-timer` package and `tinytest` when running tests. It will then run the `tests.js` file, which will contain our unit tests.

3. Now, we can create the tests by adding a folder called `tests` to our package's folder and create a file called `tests.js` inside.

   Currently, the `tinytest` package is not documented by Meteor, but it is tiny, which means it is very simple.

   Basically, there are two functions, `Tinytest.add(test)` and `Tinytest.addAsync(test, expect)`. They both run a simple test function, which we can pass or fail using `test.equal(x, y)`, `test.isTrue(x)`, or `test.isUndefined(x)`.

   For our package tests, we will simply test whether `ReactiveTimer._intervalId` is not null after we started the timer, and we will know whether the timer runs or not.

Adding package tests

The test is built by first describing what will be tested.

To test for `_intervalId`, we add the following lines of code to our `tests.js` file:

```
Tinytest.add('The timer set the _intervalId property', function (test)
{
    var timer = new ReactiveTimer();
    timer.start(1);

    test.isTrue(timer._intervalId !== null);

    timer.stop();
});
```

Then we start a timer and test whether its `_intervalId` property is not null anymore. At the end, we stop the timer again to clean up the test.

The next test we will add to our `tests.js` file will be asynchronous, as we need to wait for the timer to run at least once:

```
Tinytest.addAsync('The timer run', function (test, expect) {
    var run = false,
        timer = new ReactiveTimer();
    timer.start(1);

    Tracker.autorun(function(c){
        timer.tick();

        if(!c.firstRun)
            run = true;
    });

    Meteor.setTimeout(function(){
        test.equal(run, true);
        timer.stop();

        expect();
    }, 1010);
});
```

Let's take a look at what is happening in this asynchronous test:

- First, we started the timer again with an interval of 1 second and created a variable called `run`. We then switched this variable to `true` only when our reactive `Tracker.autorun()` function ran. Note that we used `if(!c.firstRun)` to prevent the `run` variable from being set when the function runs the first it's executed, as we only want the "tick" after 1 second to count.
- We then used the `Meteor.setTimeout()` function to check whether `run` was changed to `true`. The `expect()` tells `Tinytest.addAsync()` that the test is over and outputs the result. Note that we also stopped the timer, as we always need to clean up after each test.

Running the package tests

To finally run the test, we can run the following command from our app's root folder:

```
$ meteor test-packages meteor-book:reactive-timer
```

This will start a Meteor app and run our package tests. To see them, we navigate to `http://localhost:3000`:

We can also run a test for more than one package at the same time by naming multiple packages separated by spaces:

```
$ meteor test-packages meteor-book:reactive-timer
iron:router
```

To see if the test works, we will deliberately make it fail by commenting out `Meteor.setInterval()` in the `my-meteor-book/packages/reactive-timer/ReactiveTimer.js` file, as shown in the following screenshot:

We should always try to make our test fail, as a test could also be written in a way that it never succeeds or fails (for example, when `expect()` was never called). This would stop the execution of other tests, as the current one could never finish.

A good rule of thumb is to test functionality as if we are looking at a black box. If we customize our tests too much depending on how a function is written, we will have a hard time fixing tests as we improve our functions.

Testing our meteor app

To test the app itself, we can use Velocity Meteor's official testing framework.

Velocity itself doesn't contain tools for testing, but rather gives testing packages such as Jasmine or Mocha a unified way to test Meteor apps and report their output in the console or the apps interface itself using the `velocity:html-reporter` package.

Let's quote their own words:

> *Velocity watches your tests/ directory and sends test files to the correct testing plugin. The testing plugin performs the tests and sends results for each test back to Velocity as they complete. Velocity then combines the results from all of the testing plugins and outputs them via one or more reporting plugins. When the app or tests change, Velocity will rerun your tests and reactively update the results.*

This is taken from `http://velocity.meteor.com`. Additionally, Velocity adds features such as Meteor stubs and automatic stubbing. It can create mirror apps for isolated testing and run setup code (fixtures).

We will now take a look at unit and integration tests using Jasmine and acceptance tests using Nightwatch.

Testing using Jasmine

To use Jasmine with Velocity, we need to install the `sanjo:jasmine` package along with the `velocity:html-reporter` package.

To do this, we'll run the following command from inside our apps folder:

```
$ meteor add velocity:html-reporter
```

Then we install Jasmine for Meteor using the following command:

```
$ meteor add sanjo:jasmine
```

In order that Velocity can find the tests, we need to create the following folder structure:

```
- my-meteor-blog
  - tests
```

```
    - jasmine
    - client
      - unit
      - integration
    - server
      - unit
```

Now, when we start the Meteor server using `$ meteor`, we will see that the Jasmine package has already created two files in the `/my-meteor-blog/tests/jasmine/server/unit` folder, which contains stubs for our packages.

Adding unit tests to the server

Now we can add unit tests to the client and the server. In this book, we will only add a unit test to the server and later add integration tests to the client to stay within the scope of this chapter. The steps to do so are as follows:

1. First, we create a file called `postSpecs.js` within the `/my-meteor-blog/tests/jasmine/server/unit` folder and add the following command:

   ```
   describe('Post', function () {
   ```

   This will create a test frame describing what the test inside will be about.

2. Inside the test frame, we call the `beforeEach()` and `afterEach()` functions, which will run before and after each test, respectively. Inside, we will create stubs for all Meteor functions using `MeteorStubs.install()` and clean them afterwards using `MeteorStubs.uninstall()`:

   ```
   beforeEach(function () {
       MeteorStubs.install();
   });

   afterEach(function () {
       MeteorStubs.uninstall();
   });
   ```

> A stub is a function or object that mimics its original function or object, but doesn't run actual code. Instead, a stub can be used to return a specific value that the function we test depends on.
>
> Stubbing makes sure that a unit test tests only a specific unit of code and not its dependencies. Otherwise, a break in a dependent function or object would cause a chain of other tests to fail, making it hard to find the actual problem.

3. Now we can write the actual test. In this example, we will test whether the `insertPost` method we created previously in the book inserts the post, and makes sure that no duplicate slug will be inserted:

```
it('should be correctly inserted', function() {

    spyOn(Posts, 'findOne').and.callFake(function() {
        // simulate return a found document;
        return {title: 'Some Tite'};
    });


    spyOn(Posts, 'insert');

    spyOn(Meteor, 'user').and.returnValue({_id: 4321, profile:
{name: 'John'}});

    spyOn(global, 'moment').and.callFake(function() {
        // simulate return the moment object;
        return {unix: function(){
            return 1234;
        }};
    });
```

   First, we create stubs for all the functions we are using inside the `insertPost` method to make sure that they return what we want.

   Especially, take a look at the `spyOn(Posts, "findOne")` call. As we can see, we call a fake function and return a fake document with just a title. Actually, we can return anything as the `insertPost` method only checks whether a document with the same slug was found or not.

4. Next, we actually call the method and give it some post data:

```
    Meteor.call('insertPost', {
        title: 'My Title',
        description: 'Lorem ipsum',
        text: 'Lorem ipsum',
        slug: 'my-title'
    }, function(error, result){
```

5. Inside the callback of the method, we add the actual tests:

```
        expect(error).toBe(null);

        // we check that the slug is returned
```

```
            expect(result).toContain('my-title');
            expect(result.length).toBeGreaterThan(8);

            // we check that the post is correctly inserted
            expect(Posts.insert).toHaveBeenCalledWith({
                title: 'My Title',
                description: 'Lorem ipsum',
                text: 'Lorem ipsum',
                slug: result,
                timeCreated: 1234,
                owner: 4321,
                author: 'John'
            });
        });
    });
```

First, we check whether the error object is null. Then we check whether the resultant slug of the method contains the `'my-title'` string. Because we returned a fake document in the `Posts.findOne()` function earlier, we expect our method to add some random number to the slug such as `'my-title-fotvadydf4rt3xr'`. Therefore, we check whether the length is bigger than the eight characters of the original `'my-title'` string.

At last, we check whether the `Post.insert()` function was called with the expected values.

To fully understand how you can test Jasmine, take a look at the documentation at `https://jasmine.github.io/2.0/introduction.html`.

You can also find a good cheat sheet of Jasmine functions at `http://www.cheatography.com/citguy/cheat-sheets/jasmine-js-testing`.

6. Finally, we close the `describe(...` function at the beginning:

```
});
```

If we now start our Meteor app again using `$ meteor`, after a while we'll see a green dot appearing in the top-right corner.

Clicking on this dot gives us access to Velocity's `html-reporter` and it should show us that our test has passed:

To make our test fail, let's go to our `my-meteor-blog/methods.js` file and comment out the following lines:

```
if(Posts.findOne({slug: postDocument.slug}))
    postDocument.slug = postDocument.slug +'-'+ Math.random().
toString(36).substring(3);
```

This will prevent the slug from getting changed, even if a document with the same slug already exists, and fail our test. If we go back and check in our browser, we should see the test as failed:

We can add more tests by just adding a new `it('should be xyz', function() {...});` function.

Adding integration tests to the client

Adding integration tests is as simple as adding unit tests. The difference is that all the test specification files go to the `my-meteor-blog/tests/jasmine/client/integration` folder.

Integration tests, unlike unit tests, run in the actual app environment.

Adding a test for the visitors

In our first example test, we will test to ensure that visitors can't see the **Create Post** button. In the second test, we will log in as an administrator and check whether we are able to see it.

1. Let's create a file named `postButtonSpecs.js` in our `my-meteor-blog/tests/jasmine/client/integration` folder.
2. Now we add the following code snippet to the file and save it:

```
describe('Vistors', function() {
    it('should not see the create posts link', function () {
        var div = document.createElement('DIV');
        Blaze.render(Template.home, div);

        expect($(div).find('a.createNewPost')[0]).not.
toBeDefined();
    });
});
```

Here we manually create a `div` HTML element and render the `home` template inside. After that, we check whether the `a.createNewPost` link is present.

If we go back to our app, we should see the integration test added and passed:

In case the test doesn't show up, just quit and restart the Meteor app in the terminal again.

Adding a test for the admin

In the second test, we will first log in as administrator and then check again whether the button is visible.

We add the following code snippet to the same `postButtonSpecs.js` file as the one we used before:

```
describe('The Admin', function() {
    afterEach(function (done) {
        Meteor.logout(done);
    })

    it('should be able to login and see the create post link',
function (done) {
        var div = document.createElement('DIV');
        Blaze.render(Template.home, div);

        Meteor.loginWithPassword('johndoe@example.com', '1234',
function (err) {

            Tracker.afterFlush(function(){

              expect($(div).find('a.createNewPost')[0]).toBeDefined();
                expect(err).toBeUndefined();

                done();
            });

        });
    });
});
```

Here we add the `home` template to a `div` again, but this time we log in as an admin user, using our admin credentials. After we have logged in, we call `Tracker.afterFlush()` to give Meteor time to re-render the template and then check whether the button is now present.

Because this test runs asynchronously, we need to call the `done()` function, which we passed as an argument to the `it()` function, telling Jasmine that the test is over.

Our credentials inside the test file are secure, as Meteor doesn't bundle files in the `tests` directory.

If we now go back to our browser, we should see the two integration tests as passed:

After creating a test, we should always make sure we try to fail the test to see whether it actually works. To do so, we can simply comment out the `a.createNewPost` link in `my-meteor-blog/client/templates/home.html`.

You can run Velocity tests using PhantomJS as follows:

```
$ meteor run --test
```

You first need to install PhantomJS globally with `$ npm install -g phantomjs`. Be aware that this feature is experimental at the time of writing this book and might not run all your tests.

Acceptance tests

Though we can test client and server code separately with these tests, we can't test the interaction between the two. For this, we need acceptance tests, which, if explained in detail, would go beyond the scope of this chapter.

At the time of this writing, there is no acceptance testing framework that is implemented using Velocity, though there are two you can use.

Nightwatch

The `clinical:nightwatch` package allows you to run an acceptance test in a simple way as follows:

```
"Hello World" : function (client) {
     client
        .url("http://127.0.0.1:3000")
        .waitForElementVisible("body", 1000)
        .assert.title("Hello World")
        .end();
}
```

Though the installation process is not as straightforward as installing a Meteor package, you need to install and run MongoDB and PhantomJS yourself before you can run the tests.

If you want to give it a try, check out the package on atmosphere-javascript website at `https://atmospherejs.com/clinical/nightwatch`.

Laika

If you want to test the communication between the server and the client, you can use Laika. Its installation process is similar to Nightwatch, as it requires separate MongoDB and PhantomJS installations.

Laika spins up a server instance and connects multiple clients. You then can set up subscriptions or insert and modify documents. You can also test their appearance in the clients.

To install Laika, go to `http://arunoda.github.io/laika/`.

At the time of this writing, Laika is not compatible with Velocity, which tries to run all the files in the test folder in Laika's environment, causing errors.

Summary

In this final chapter, we learned how to write simple unit tests using the `sanjo:jasmine` package for Meteor's official testing framework, Velocity. We also took a brief look at possible acceptance test frameworks.

If you want to dig deeper into testing, you can take a look at the following resources:

- `http://velocity.meteor.com`
- `http://jasmine.github.io`
- `http://www.cheatography.com/citguy/cheat-sheets/jasmine-js-testing`
- `http://doctorllama.wordpress.com/2014/09/22/bullet-proof-internationalised-meteor-applications-with-velocity-unit-testing-integration-testing-and-jasmine/`
- `http://arunoda.github.io/laika/`
- `https://github.com/xolvio/velocity`

You can find this chapter's code files at `https://www.packtpub.com/books/content/support/17713` or on GitHub at `https://github.com/frozeman/book-building-single-page-web-apps-with-meteor/tree/chapter12`.

Now that you have read the whole book, I assume you know a lot more about Meteor than before and are as excited about this framework as I am!

If you have any questions concerning Meteor, you can always ask them at `http://stackoverflow.com`, which has a great Meteor community.

I also recommend reading through all Meteor subprojects at `https://www.meteor.com/projects`, and study the documentation at `https://docs.meteor.com`.

I hope you had a great time reading this book and you're now ready to start making great apps using Meteor!

Module 2

Meteor Cookbook

Over 65 hands-on recipes that cover every aspect of building and deploying elegant, full-stack web applications with Meteor

1

Optimizing Your Workflow

In this chapter, we will cover the following topics:

- Installing Meteor
- Finding documentation for Meteor
- Getting help with questions
- Setting up your project file structure
- Setting up your development environment
- Using the web console
- Deploying a test app to Meteor
- Deploying to Meteor using a CNAME redirect
- Deploying to a custom hosted environment
- Deploying with Meteor Up (MUP)
- Using CoffeeScript
- Using CSS compilers

Introduction

Welcome to the wonderful world of Meteor! This chapter will walk you through some of the best practices for optimizing your Meteor workflow. From installing Meteor to deploying your finished application on your own custom server, the recipes found here will get you up and running and provide a great reference as you start working with Meteor on a daily basis.

Installing Meteor

The folks at Meteor have made installation a breeze. As long as the system you're working on (Linux or Mac OS X) is reasonably up to date, installing Meteor should be very simple. We have included it as a recipe so that when you're installing Meteor on a new machine, you have it handy. We will also include installation instructions directly from GitHub, just in case you want to use a development branch.

Getting ready

You must have `curl` installed in order to install Meteor. If you are on Mac OS X, `curl` is already installed. If you are on Linux, `curl` is usually installed. To check for `curl`, open a terminal window and execute the following command:

```
$ curl
```

If `curl` is installed, you will receive the following message:

```
curl: try 'curl --help' or 'curl --manual' for more information
```

If `curl` isn't installed, you will receive a message similar to the following:

```
-bash: /usr/bin/curl: No such file or directory
```

To install `curl`, use `apt-get` or `yum` (depending on your Linux version). The following is the command to install `curl` on Ubuntu. Simply replace `apt-get` with `yum` in the command to install it on Fedora, Debian, or CentOS:

```
$ sudo apt-get install curl
```

You should see an installation message, with perhaps a prompt to install. Complete the installation process; you will then be ready to install Meteor.

How to do it...

Open a terminal window and execute the following command:

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

How it works...

The `curl https://install.meteor.com/` command is used to retrieve an installation script directly from Meteor. Once the script is downloaded, the `| sh` argument tells your system to execute the script, which installs Meteor at `/usr/local/bin/meteor` (which should be in your path). Once the entire script is executed, you will be able to run the `meteor` command from any location.

There's more...

The preceding recipe covers the default/stable Meteor installation. If you'd like to install Meteor directly from the source code or if you'd like to install a development branch (for example, the current nightly), you can do so using `git`.

The Meteor repository is located at `https://github.com/meteor/meteor`. You can find various development branches at this location, in addition to the main Meteor repository.

Assuming you have `git` installed and would like to install the main Meteor build, open a terminal window and execute the following command:

```
$ git clone git://github.com/meteor/meteor.git
```

This will clone the latest Meteor build into a subfolder named `meteor`. Note that this does not install Meteor globally on your machine. This means that in order to run the `meteor` command, you will need to navigate to (or reference) the `meteor` folder where you just cloned the Meteor repository.

> Various other installation arguments and customizations can be used, depending on your situation. To see more comprehensive instructions, visit the Meteor GitHub main `README.md` page at `https://github.com/meteor/meteor`.

Finding documentation for Meteor

Meteor has finally reached version 1.0, and as such, it's various libraries and features are beginning to stabilize. However, there are still some parts of Meteor that are in flux. It is therefore important to have the latest documentation handy. This recipe will show you where you can find the official Meteor documentation when you need it.

How to do it...

Simply navigate to `http://docs.meteor.com/` in a browser and bookmark the page.

How it works...

That's all there is to it. The Meteor folks keep the documentation up to date, and you can be sure to find all production-ready features, along with a lot of great advice, at that location.

There's more...

If you would like to have an offline copy of the documentation and still make sure you have the latest and greatest version of documentation, you can use Meteor's GitHub repository to serve up a local copy of the latest documentation.

To get the initial copy of the documentation, navigate to where you would like to store the documentation (for example, your `Documents` directory) and clone the latest Meteor repository with the following command:

```
$ git clone git://github.com/meteor/meteor.git
```

If you've already done this, and would like to make sure you have the latest and greatest, navigate to the `meteor` subfolder (created when you did the initial Git clone) and run the following command:

```
$ git pull origin master
```

This will update the Meteor repository with the latest and greatest.

Once you've got the latest documentation, you can access it via a web browser by navigating to the `docs` directory:

```
$ cd docs/
```

Then, run the `meteor` command:

```
$ meteor
```

Once the packages are updated, the Meteor documentation project will start and you will see the following in the command line:

```
=> Meteor server running on: http://localhost:3000/
```

You can now use a browser to navigate to `http://localhost:3000/` and read the latest documentation.

Getting help with questions

The more you use Meteor, the more familiar the major packages will become. However, you may also want to start branching out and doing more advanced things with some of the lesser-known packages. You may also come up with an idea for a new application of Meteor. In any case, you will eventually run into a situation where you need more information than what is contained in the documentation. Here are some quick tips on where to go when you need your questions answered.

How to do it...

There are three major sources for getting your questions answered:

1. For specific technical questions or problems, head over to `http://stackoverflow.com/questions/tagged/meteor`. Search for questions others have already asked; if you can't find an answer, go ahead and submit a question of your own.
2. For general questions and comments, visit the Meteor forums at `https://forums.meteor.com/`.
3. To work in real time with multiple Meteor developers (including members of the core team), you can visit the IRC #meteor chat room on `irc.freenode.net`.

How it works...

Let's look at some tips on using each of the previously mentioned resources.

Stack Overflow

Stack Overflow is a very active community of developers, many of whom are eager to help others with their technical questions. For Meteor-specific help, once you've navigated to `http://stackoverflow.com/questions/tagged/meteor`, you can perform a search using the keywords that best describe your question, for example, if you are interested in getting Meteor to work inside a PhoneGap application, you may want to search all the tagged **[meteor]** questions for `cordova`, as follows:

This will search all the previously submitted questions tagged with the keyword **[meteor]**, which also contain the word `cordova` somewhere in the question. You will want to look through the results to see whether your question has already been asked. If one of the questions (and the answer to that question) satisfies your needs, great! If your question hasn't been answered already, you can ask your own question.

Click on the button labeled **Ask Question**:

Fill out the details of your question. Don't forget to include tags to more easily identify and categorize your question. Submit your question and the Stack Overflow community will most likely have an answer for you very quickly.

Meteor forums

Meteor forums are also very active, with lots of enthusiastic Meteor developers visiting the site daily. You can ask questions, view discussions, and leave comments on a lot of different Meteor-related topics. Most of the time, if you have a technical question or problem, it's best to use Stack Overflow. However, forums are also a great resource for more abstract or specific questions about Meteor.

The #meteor on IRC

The #meteor IRC channel is a great channel to join in and listen to. You'll be able to interact with all kinds of brilliant developers, many of whom are willing to answer your questions immediately. Do keep in mind that most questions or problems you're having (especially when you're new to Meteor) have already been addressed on Stack Overflow or the Meteor forums, and it is courteous to first check and make sure your question hasn't been answered there before asking in the IRC chat.

There's more...

Keep in mind that since Meteor is still new, there are many who are struggling with questions that you may be able to answer. Also, when you find the answer to your own question, it may be worth putting it up on Stack Overflow. This will help you save time for other people who may run into the same problem.

Answering other people's questions on Stack Overflow, the Meteor forums, and even on #meteor IRC is a great way to contribute to the community.

Don't forget to answer your own questions on Stack Overflow, if you happen to find the answers yourself!

Setting up your project file structure

When you spin up a new project in Meteor, the default file configuration is built to get you up and running immediately. That's great, but if you're looking to keep your code optimized and easily understood, you'll need to tweak the file structure a bit. The following recipe maps out the file structure that you can use for the majority of your projects.

Getting ready

You'll want to create a new base Meteor project, which we will then modify to suit our needs. In a terminal window, navigate to the folder where you would like to create the project and enter the following command to create a project called `FileTemplate`:

```
$ meteor create FileTemplate
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. Optionally, the code samples can also be downloaded from `https://github.com/strack/PacktMeteorRecipes`.

This will create the base file structure. Now, navigate to the folder in the File Explorer window of your choice. For this exercise, we'll use a Mac OS X Finder window so that we can visualize the folder structure:

Always start with a fresh Meteor project (don't try to create your folder structure manually). Even though the file structure looks simple, there are hidden files and folders created by the `meteor create` command that you can't live without!

How to do it...

To set up your project file structure, proceed with the following steps:

1. In your root folder, delete the default `.css`, `.js`, and `.html` files. **FileTemplate** should now appear empty, similar to the following screenshot:

> Instead of deleting the base `.html`, `.css`, and `.js` files, you could break them apart using the code/file entries discussed in the following steps.

2. Create the following subfolders in your root folder:
   - `client`
   - `server`
   - `both`
   - `private`
   - `public`

3. Navigate to the `client` subfolder. Create a new file named `main.html`. Edit this file and add the following code (taken from `FileTemplate.html`):

```
<head>
  <title>FileTemplate</title>
</head>

<body>
  <h1>Welcome to Meteor!</h1>

  {{> hello}}
</body>

<template name="hello">
  <button>Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>
</template>
```

4. Create a new subfolder in the `client` folder named `scripts`.

5. Inside `scripts`, create a new file called `main.js`. Edit this file, adding the following code (taken from the `Meteor.isClient` section of `FileTemplate.js`):

```
 // counter starts at 0
Session.setDefault('counter', 0);

Template.hello.helpers({
  counter: function () {
    return Session.get('counter');
  }
});

Template.hello.events({
  'click button': function () {
    // increment the counter when button is clicked
    Session.set('counter', Session.get('counter') + 1);
  }
});
```

6. Create a new subfolder in the `client` folder named `lib`. Create two subfolders named `scripts` and `styles`.
7. Inside `styles`, create a new file called `style.css`.
8. Navigate to the `server` subfolder. Create a new file named `server.js`. Add the following code to the `server.js` file and save the changes:

```
Meteor.startup(function () {
  // code to run on server at startup
});
```

Your completed file structure should now look like the following screenshot:

How it works...

We'll break down this file structure in pairs for the most part. This is because for almost every folder, there is another folder that serves the opposite purpose.

client/server

The `client` folder is interpreted by Meteor as code that belongs exclusively to the client. The `server` folder is the exact opposite and tells Meteor that any files contained inside should only apply to server-side processing.

Having separate `client` and `server` folders logically separates your code (making it easier to visualize), saves unnecessary processing, and prevents errors (client templates and rendering code will throw errors if processed by the server).

Inside of the `client` folder, we've placed a `scripts` subfolder and two files, both with the main prefix (`main.html` and `main.js`). The `scripts` subfolder is for our benefit, separating out HTML and JavaScript so that when we go back and review our code, it will be easy to read and segment.

main/lib

The `main` prefix is used to tell Meteor: *run this code last*. When Meteor goes about processing and compiling files, it waits to process anything named `main` until the very end. This helps in situations where libraries or scripts need to be initialized before you can take advantage of them.

Conversely, the `lib` folder tells Meteor: *run this code first*. Anything placed inside, or as a child of, the `lib` folder will be run first by Meteor at runtime. This is a great place to put global variables, common libraries not already included as packages (see *Chapter 2*, *Customizing with Packages*), and of course any style sheets. This is why we've included a `scripts` and `styles` subfolder and have created an initial generic `style.css` file.

Using the `main` prefix and the `lib` folder together helps us maintain proper sequences and ensures that our styles load as quickly as possible.

public/private

The `public` folder is a repository for assets that need to be accessed globally. It's a very good place to store non-sensitive information, such as images and text files. It's also easy to remember. If you need something to be shown publicly, put it in the `public` folder.

The `private` folder is the opposite. It's a place to store assets that can only be seen by the server. This is a good place to put moderately sensitive information (you'll want additional security, such as a database, for the information you want to keep more secure). Any files placed in the `private` folder can be referenced by the server with no special path or navigation arguments.

both

The `both` folder stands all by itself as the folder accessible by both the client and the server. When you are creating models and collections (see *Chapter 4*, *Creating Models*) you'll want to put the declarations in a file inside of the `both` folder. The main reason for doing this is so that you don't duplicate code, and you can ensure that both the client and the server are working from the same model declarations.

There's more...

Obviously, you don't need to create this file structure from scratch every time. You can use your favorite script engine (Grunt, Gulp, Shell Script, and so on) and create a script that can make the necessary changes with a single command. Alternatively, you can create the template once, copy the entire folder to a new location, and then use it as if it were a new project.

If you do decide to use this recipe rather than an automated script, make sure you update your base template to the latest Meteor build periodically and each time you make a copy. This can be done very easily in a terminal window. Navigate to the root folder (for example `~/Documents/Meteor/FileTemplate`) and run the following command:

```
$ meteor update
```

You will either receive a message letting you know that the folder is already up to date, or Meteor will update the project to the latest version.

See also

- *Chapter 2*, *Customizing with Packages*

Setting up your development environment

While you're developing your project, there are several default packages and settings that make your debugging and development work much easier. This recipe will give you suggestions to help create an ideal development environment.

Getting ready

You'll want to make sure you have Meteor installed (of course), and you will need a default project file structure set up. You can either use the *Setting up your project file structure* recipe from earlier in this chapter, or create a brand new Meteor project using the `meteor create [...]` command.

This recipe assumes that you have some Mongo collections and/or some kinds of user accounts set up. If you don't have a collection added, you may want to add one just for testing purposes. To do so, create a file named `[project root]/both/model.js` and enter the following line of code:

```
testCollection = new Mongo.Collection('testCollection');
```

With a project in place and at least one Mongo collection to monitor, we're ready to start developing and debugging!

How to do it...

First, we want to make sure that the `insecure` and `autopublish` packages are installed, which is usually the case with a default Meteor project.

1. In a terminal window, navigate to the root folder of your project and enter the following command:

   ```
   $ meteor list
   ```

   This will list all the packages used by your project. You should receive something similar to the following:

   ```
   autopublish      1.0.2
   insecure         1.0.2
   meteor-platform  1.2.1
   …
   ```

2. If the `autopublish` or `insecure` packages are missing, you can add them manually by entering the following commands:

   ```
   $ meteor add autopublish
   $ meteor add insecure
   ```

3. Next, install the `msavin:mongol` smart package. In the terminal window, enter the following command:

   ```
   $ meteor add msavin:mongol
   ```

   After a short installation process, you should receive a message indicating that the `msavin:mongol` package was installed correctly.

4. Start your `meteor` project by entering the following command in your terminal window:

   ```
   $ meteor
   ```

5. Open your app in a browser by navigating to `http://localhost:3000`. Once there, press *Ctrl* + *M* and look towards the bottom-left of the screen. You will see a small expandable dashboard, similar to the following screenshot:

Clicking on any of the collection names or the **Account** tab will let you view, update, delete, and insert records, as shown in the following screenshot:

This comes in very handy when you're building your app, as you have instant access to your collections and user account profiles.

How it works...

The following line is all that's needed to install a package:

```
$ meteor add msavin:mongol
```

This tells Meteor to go and find the `msavin:mongol` package, with all of its dependencies, and add the package to your project. This is a third-party package, and more details on the package can be found at `https://atmospherejs.com/msavin/mongol`.

Once installed, you can access it on your browser page by the *Ctrl + M* shortcut. Under the hood, the `mongol` package is monitoring for any collections, specifically filtering user-account-related collections into the **Account** tab. Other than that, all collections are treated the same, and this interface just simplifies your debugging/development process when you need to see what's going on inside your collections.

There's more...

Mongol is very new but it is getting better all the time. You can read about all it's current features and preview upcoming features by visiting `https://github.com/msavin/Mongol`.

See also

- The *Adding Meteor packages*, *Removing Meteor packages*, and *Using npm modules* recipes in *Chapter 2*, *Customizing with Packages*
- The *Basic safety – turning off autopublish* and *Basic safety – removing insecure* recipes in *Chapter 9*, *Securing Your Application*

Using the web console

Sometimes modifying your code while debugging is just too slow. This recipe will introduce you to the web console and give you some pointers on using it.

Getting ready

We will need a sandbox to play in. For the purpose of this recipe, we'll use one of the default examples built into Meteor.

In a terminal window, enter the following command:

```
$ meteor create --example leaderboard
```

Once created, navigate to the `leaderboard` folder and start Meteor:

```
$ cd leaderboard
$ meteor
```

We can now navigate to our leaderboard example page (`http://localhost:3000`) using the web browser of our choice.

We need to enable the web console in various browsers. In no particular order, here's how to get to the web console on the browsers you're likely to be working with:

Safari

Enable the **Develop** menu by going to **Safari** | **Preferences** | **Advanced** (*CMD+*, shortcut if you prefer) and making sure the **Show Development Menu in menu bar** option is enabled:

Now, on any web page, click on **Show Error Console** under **Develop** (or use the *CMD* + *Alt* + *C* shortcut):

This will bring up the web console, as shown in the following screenshot:

Firefox

On any web page, click on the **Web Console** menu option (*CMD* + *Alt* + *K* shortcut) under **Tools** | **Web Developer**:

The web console now appears as shown in the following screenshot:

Chrome

On any web page, click on the **Developer Tools** menu item (*CMD* + *Alt* + *I* shortcut) under **View** | **Developer**:

This will bring up the web console, as shown in the following screenshot:

How to do it...

To show what's possible with a web console, let's create a new scientist, and add some points to Tesla. Because, Imma let you finish, but Nikola Tesla was the greatest scientist of all time. Of. All. Time.

1. If you haven't done so already, navigate to `http://localhost:3000/`.
2. Now, open the web console if it's not already open.
3. In the console, run the following command:

   ```
   > Players.insert({name:"Stephen Hawking"})
   ```

4. The name **Stephen Hawking** will now be in the list. You can select his name and add some points:

5. Now, let's give Tesla the love he deserves. First, let's find the `_id` value for the Tesla record. Run the following command in the console:

```
> Players.findOne({name:"Nikola Tesla"})
```

6. Harvest the `_id` valuefrom the result and modify the score, similar to the following:

```
> Players.update({_id:"bPDpqp7hgEx4d6Eui"},
{$set:{score:334823}})
```

   Tesla will now be placed at the top of the leaderboard:

7. One last manipulation, let's deselect any of the scientists mentioned in the preceding screenshot. Enter the following command in the console:

```
> Session.set("selectedPlayer",null)
```

   You will notice that, now no scientist is selected.

You can run through the previous commands in the web console by pressing the up or down arrow keys.

How it works...

The web console acts as an extension of your code. You are able to receive log messages, call exposed methods and events, or change and instantiate global variables. You are able to do this dynamically in real time, which can have a lot of advantages as you're testing or debugging.

With a few well-composed lines of code, we were able to replicate the behavior found in the event handlers and methods that belong to the app. In other words, we were able to manually test things, rather than having to modify, save, and retest the code. This kind of impromptu testing can really save us time.

There's more...

The web console and the associated development tools can perform all kinds of client-side debugging and testing tasks. You can run prewritten scripts, pause code during execution, monitor variables and the call stack, and even create your own methods and variables dynamically.

The web console varies a little bit from browser to browser, but you can find instructions that apply to nearly every web console at `https://developers.google.com/chrome-developer-tools/docs/console`.

Deploying a test app to Meteor

As you've come to expect, Meteor makes just about everything easier to do. Testing your application on a server is no exception. This recipe will show you how to deploy your application to Meteor servers, where you can test (and show off!) your new application.

Getting ready

The only requirement for this recipe is that you have a working application.

Your app doesn't need to be fancy or complete, but it should at least render something on the screen so that you can verify it is running on Meteor servers.

For this recipe, we will use the default `leaderboard` example, which is created using the following command in a terminal window:

```
$ meteor create --example leaderboard
```

How to do it...

To deploy a test app to Meteor, proceed with the following steps:

1. First, you need to pick an application name.

   The application name will appear on Meteor servers as a subdomain of the URL where your app is being served, such as `http://myproject.meteor.com`.

   However, there are a lot of people testing applications on Meteor servers. Chances are pretty good that a generic name, such as "myproject" or "leaderboard", is already taken. So, we need to pick something unique.

   For this recipe, I will use this application name `packtrecipe`. You will obviously need to pick something else. I got here first!

   Your project and application name do *not* need to match. The application name is only an identifier so that Meteor servers know how to route to your application.

2. Now that we've selected an application name, we will open a terminal window and navigate to the root directory of our project:

   ```
   $ cd leaderboard
   ```

3. Once we're in the correct folder, we will issue the `deploy` command as follows:

   ```
   $ meteor deploy packtrecipe
   ```

4. Meteor will bundle and deploy your application. Upon completion, you should see something similar to the following in the terminal window:

   ```
   Deploying to packtrecipe.meteor.com.  Bundling...
   Uploading...
   Now serving at packtrecipe.meteor.com
   ```

5. Navigate to the URL Meteor gave you (in this case, `http://packtrecipe.meteor.com`) in a browser and you will see your newly deployed application, as follows:

How it works...

It's magic! The folks at Meteor are sorcerers!

Actually, the core development team has worked very hard to make deployment as easy as possible. Under the hood, your local Meteor installation is packaging the correct files, compiling package lists and dependencies, and then sending a request to the Meteor servers. Once the Meteor servers receive the request, all the necessary error checking is done, and the packages, database, and application are created and initialized. A virtual host address is added, and voila! Your app is up and running.

There's a lot more detail (and code) involved, but this should give you a general idea of what goes into deploying to the Meteor test servers. Aren't you glad you didn't have to write all of that code?

There's more...

You probably noticed that you had to create a Meteor developer account in order to deploy. This is as it should be, because if there were no login/security details, someone could come along and override your deployment with one of their own.

This requirement, however, is quite flexible, allowing you to add other users or even organizations so that multiple people can deploy or update your app.

To set up an organization, navigate to `https://www.meteor.com/account-settings/organizations` and click on the button labeled **NEW ORGANIZATION** at the bottom of the screen. You can then add individual users to the organization, similar to the following screenshot:

Once you've deployed your application, you can authorize the changes made by an individual or an organization through the `meteor authorized` command. Use the following syntax to add authorized accounts:

```
$ meteor authorized [your-url.meteor.com] --add [user/organization]
```

So, for example, we would use the following to add the `packtmeteor` organization to our deployed app:

```
$ meteor authorized packt.meteor.com --add packtmeteor
```

Likewise, if we wanted to remove authorization (for example, let's remove the `strack2` account), we would enter something like this:

```
$ meteor authorized packt.meteor.com --remove strack2
```

Meteor developer accounts have other useful functions as well. To explore what you can do with your account, please visit `https://www.meteor.com/account-settings`.

See also

- The *Building a custom package* recipe in *Chapter 2*, *Customizing with Packages*

Deploying to Meteor using a CNAME redirect

Deploying to Meteor test servers is all well and good, but what if you want to use your own custom domain name? The folks at Meteor have created a simple way for you to do just that. This recipe will show you how to take advantage of this simple, yet powerful feature.

Getting ready

You will need to create a **CNAME** redirect to `origin.meteor.com` on the hosting service where your domain is registered. How to do this varies pretty widely, so consult your hosting service's knowledge base on the exact steps. For this recipe, we'll use the cPanel interface of a hosting service for the `packtpub.com` domain.

Enter the subdomain you wish to use in your **CNAME** redirect (for example, `meteor.packtpub.com`) and set the redirect location to `origin.meteor.com`. Click on **Add CNAME Record** to submit the record:

How to do it...

Let's assume the subdomain we're going to use is `meteor.packtpub.com`. In order to deploy to the Meteor environment, perform the following steps:

1. Once you have your **CNAME** redirect properly set, open a terminal window, navigate to the root directory of your project, and enter the following:

   ```
   $ meteor deploy meteor.packtpub.com
   ```

2. Meteor will deploy your app and provide feedback as the bundling, uploading, and serving steps are completed:

   ```
   Deploying to meteor.packtpub.com.  Bundling...
   Uploading...
   Now serving at meteor.packtpub.com
   ```

3. To verify the application, navigate to your application's URL (for example, `http://meteor.packtpub.com`) in a browser. If everything was deployed correctly, you will see your application up and running.

How it works...

This Meteor deployment feature is nearly identical to the default deployment, with just a little bit of extra code built in to interpret the origin of **CNAME** redirect.

When a request comes in to `origin.meteor.com` via a **CNAME** redirect, Meteor takes the **CNAME** redirect's original destination and uses it as the unique identifier for the application deployment. Meteor also uses future requests to `origin.meteor.com` from this **CNAME** redirect to serve the application.

In this particular case, the original **CNAME** destination was `meteor.packtpub.com`. When the request is redirected to `origin.meteor.com`, Meteor recognizes the **CNAME** redirect and uses it to direct traffic to the deployed application.

See also

- The *Deploying a test app to Meteor* recipe in this chapter

Deploying to a custom hosted environment

When you are ready to actually deploy your app to a production environment, you will need to convert your Meteor application to a straight up Node application. There are multiple methods to do this, all of which are (currently) manual processes and not for the beginner. A lot can go wrong. This recipe will show you how to manually deploy your Meteor application to a production server as a Node application.

Getting ready

As mentioned, there are multiple methods and server configurations that can successfully host a Node application. There are, however, some commonalities, all of which you will need for this recipe.

First, you will need a hosted **MongoDB** database.

Your MongoDB database can be hosted remotely or on the same machine from where you will deploy your Node application. For this recipe, we have deployed a default MongoDB database to the same machine where we will host the app. Our `MONGO_URL` value will therefore be:

```
MONGO_URL=mongodb://localhost:27017
```

> Installation instructions (and tutorials) for MongoDB can be found at `http://docs.mongodb.org/manual/installation/`.

Free or cheap but dedicated MongoDB service hosting also exists. Run a quick Internet search for *MongoDB hosting* or visit `http://www.mongodb.com/partners/cloud` for a list of providers.

Next, your hosting environment will need the latest stable Node and npm modules.

Installation instructions for each of these programs are beyond the scope of this book. For installation instructions and advice on how to get started, visit the following:

- Node (`http://nodejs.org`)
- npm (`https://www.npmjs.com/`)

The latest or nightly builds of these programs will likely cause you problems in a production environment. Be sure to use the stable versions, unless you have thoroughly tested a different version.

Lastly, you will need the `forever` npm module installed in your hosting environment. From a terminal window on your hosted environment, run the following command:

```
$ npm install -g forever
```

Note that you may need to use the `sudo` command to install the package globally. This will depend on how npm was installed on your server. The preceding command will install `forever` on your machine, and you are now ready to prepare and deploy your application to production.

How to do it...

To deploy to a custom hosted environment, proceed with the following steps:

1. Open a terminal window in your development environment and navigate to the root folder of your Meteor project. In this root folder, execute the following command:

   ```
   $ meteor build [your-build-location] --architecture [x]
   ```

   Replace the placeholders in the preceding line of code with your build location (for example: `~/Documents/builds/mybuild`) and an architecture (the options are `os.osx.x86_64`, `os.linux.x86_64`, and `os.linux.x86_32`). The `--architecture` option is optional if you are building this on the same server where you will deploy your app.

2. Meteor will bundle, extract, and repackage a clean production copy of your Meteor project, preparing it to be used in your hosting environment. While it is being built, Meteor will update you with the status in the terminal window.
3. Once the build is finished, navigate to the build folder you specified, for example, if you specified `~/Documents/builds/mybuild` as your build location, you would need to enter the following command:

   ```
   $ cd ~/Documents/builds/mybuild
   ```

4. In the build folder, you will see a `tarball` file with a name similar to `[your-project-name].tar.gz`.

   So, for example, if the name of my project was `leaderboard`, the name of the `tarball` file would be `leaderboard.tar.gz`.

5. Make a note of the name and location because you'll need it when you copy and extract the build to your server.

6. For this recipe, let's assume you're using a Linux server to host your production application. Let's create a `meteorapps` folder using the following terminal command:

   ```
   $ mkdir /home/meteorapps
   ```

7. Next, copy the `tarball` file from your development environment to the `/home/meteorapps/` folder in your production-hosted environment.

   If you build your application on a different machine, you can transfer it via SCP, FTP, a common file server, a Git repository, and so on. It really doesn't matter how you copy it over, as long as a copy of the `tarball` file is now in your hosted environment.

8. Once it's copied over, run the following command on the `tarball` file:

   ```
   $ tar -xf  [your-tarball-name].tar.gz
   ```

   This will extract the file to a folder named `bundle`. If you navigate to the `bundle` folder, you will find a `README` file. The next steps in this recipe are taken from this `README` file, so feel free to check them out for a more concise set of instructions. If you check the contents of the folder, you should see something similar to the following:

   ```
   README   main.js   programs   server   star.json
   ```

9. We will now install the npm packages needed to run our application. Navigate to `bundle/programs/server` and execute the following command:

   ```
   $ npm install
   ```

   Note that we did not use the `-g` argument, as we are only installing the local npm packages specified in the bundled folder.

10. Next, we will need to set the `PORT` and `MONGO_URL` export arguments so that Node knows how to host our application. Enter the following commands in the terminal window:

    ```
    $ export PORT=8080
    $ export MONGO_URL=mongodb://localhost:27017
    ```

    These two export commands tell your Node server what port to listen on, and where to find the `mongodb` hosted instance (we're using a local instance in this example).

    You will also want to configure the `ROOT_URL` and `MAIL_URL` environment variables. The syntax to enter them is similar to the following export commands:

    ```
    $ export ROOT_URL='http://[your-hostname.com]'
    $ export MAIL_URL='smtp://user:password@mailhost:port/'
    ```

Now, we can run our application. Instead of using the default `node` command, remember that we installed the npm `forever` package. The `forever` package allows us to run our Node application and will automatically restart it if we encounter an error. Run the following command in the terminal window:

```
$ forever start main.js
```

This command instructs Node to start `main.js` as a node application, and to restart it if there are any issues.

You can stop the application later on by issuing the following command from the `bundle/` directory:

```
$ forever stop main.js
```

Now it's time to test whether your application is successfully running by opening a browser and pointing it to your host environment, on the port you specified, for example, if our production environment was hosting the `meteorapp.packtpub.com` subdomain, and we specify `port 8080`, as shown in the preceding example, we would navigate to `http://meteorapp.packtpub.com:8080` in a browser.

Your app should be up and serving pages.

How it works...

Node is built to run as quickly as possible. To do so, it will run a little differently on different hardware and software configurations. This means that the Node and npm packages you use in your development environment (for example, on MAC OS X) are slightly different than the corresponding Node and npm packages in your production environment (for example, Linux Ubuntu 12.4 LTS Precise). This is especially true for foundational packages, such as the npm fibers package.

Also, although Meteor is built on top of Node, it isn't a native Node application. There are some additional layers of abstraction and processing that make your life easier as a developer, but they don't make for the prettiest native Node production environment.

The `meteor build` command takes care of this for us and creates a build without the npm packages installed. Instead, it lists any of the npm packages as dependencies. Because the specific npm packages aren't included (Meteor listed them in a package manifest file instead), there are no compatibility issues. We simply tell Node to find and install the packages specific to the current environment, using the package manifest file as a sort of a laundry list. We did this when we issued the `npm install` command.

Once npm has read the manifest file, retrieved and installed all the needed packages, and informed us that the installation was completed correctly, we can run our new native Node application.

We then set some export variables (`PORT`, `MONGO_URL`, `ROOT_URL`, and `MAIL_URL`) and ran our application using the `forever` npm package, rather than the normal node command. Using `forever` helps us with not having to go back to the server every time it crashes. Ideally, we would never have an application crash, but we live in the real world, and `forever` is a huge time saver when an app restart is needed.

There's more...

If our application is truly production-ready, we also want it to start automatically whenever the server reboots. There are various ways to do this, depending on the operating system your production server is running on, and we won't cover all of them.

We will give an example script and instructions on how to do this on Ubuntu and let you modify the script as needed for other environments.

Whenever an Ubuntu server restarts, it runs any `*.conf scripts` script found in the `/etc/init/` folder. We'll assume that our application is located at `/home/meteorapps/prodapp/bundle/`, that we are going to listen on `port 8080`, and that we are using the local MongoDB service (feel free to adjust these settings as appropriate).

Using a terminal window with `sudo` privileges, add the following `meteorapp.conf` script to `/etc/init/` to your production server:

```
#  prodapp configuration file
#  /etc/init/meteorapp.conf

start on (local-filesystems)
stop on shutdown
script
    cd /home/meteorapps/prodapp/bundle/
    export MONGO_URL=mongodb://localhost:27107 PORT=8080
    export ROOT_URL='http://example.com'
    export MAIL_URL='smtp://user:password@mailhost:port/'
    exec forever start main.js
end script
```

Let's break down what this script does.

The first two lines (`start on` and `stop on`) tell the OS when to run this script. In this case, we run it as soon as the local file systems are ready, and we stop it as soon as a shutdown request comes in.

We then have our script that will be run. We navigate to our application folder using `cd /home/meteorapps/prodapp/bundle`.

Then, we declare the location of our MongoDB service and the port we would like the Node to listen on.

Finally, we execute the `forever` command, asking it to start `main.js` as a Node application.

The Meteor Group is currently working on a project called **Galaxy**, which will make deployment to custom servers as easy as deployment to their test servers. Once Galaxy is available, it will make this recipe obsolete, but that's a good thing! As mentioned, this recipe covers only one way of deploying to production. You may find another, easier method just by browsing `https://forums.meteor.com` or visiting other sources for Meteor information.

See also

- The *Getting help with questions* recipe in this chapter
- The *Deploying with Meteor Up (MUP)* recipe in this chapter
- The *Deploying apps to mobile devices* recipe in *Chapter 12*, *Creating Useful Projects*

Deploying with Meteor Up (MUP)

We would be remiss if we didn't add one final method of deploying Meteor apps to a production server. This method is called **Meteor Up** (**MUP**). Using a fairly simple configuration file, MUP vastly simplifies the server deployment process. And until we get to see what Galaxy (Meteor's deployment platform) looks like, MUP is the easiest way to deploy to a custom environment. This recipe will show you how to deploy your app to a custom server using Meteor Up.

Getting ready

To complete this recipe, you will need a working Meteor application. It doesn't need to be fancy, but it does need to be functional so you can see whether it was installed properly on your server at a glance.

How to do it...

MUP is installed via npm. It needs to be installed globally, so we will use the `-g` argument. To deploy your app to a custom server using MUP, perform the following steps:

1. To install MUP, open a terminal window and execute the following command:

   ```
   $ npm install -g mup
   ```

   This will take a bit to install, but once complete, you'll be ready to configure and use MUP to deploy your application.

2. To run `mup init`, navigate to the root folder of your project in a terminal window and enter the following command:

   ```
   $ mup init
   ```

   You will receive a message similar to the following:

   ```
   Empty Project Initialized!
   ```

   This process will create two files in your project's root folder. We don't need to worry about the first file (`settings.json`). We will configure our app using the second file, named `mup.json`.

3. Now, we'll configure `mup.json`. Open the `[project root]/mup.json` file in an editor. The top part of the file will have our remote server information. It should look similar to the following lines of code:

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

   The `host` property will be the name of the server you will be accessing via SSH/SCP. If our server name was `my-production-server.com`, the host property would look similar to the following:

   ```
   "host": "my-production-server.com",
   ```

   The `username` and `password` properties are the user/pass combination you would use if you were to `ssh` into the remote server.

   The next four properties help us to configure our server. If we want MUP to install Mongo, Node, or PhantomJS (all necessary to deploy via MUP), we can specify the code as follows:

   ```
     "setupMongo": true,
     "setupNode": true,
     "nodeVersion": "0.10.33",
     "setupPhantom": true,
   ```

As of the printing of this book, the latest stable Node version is `0.10.33`. You may want to check this and modify the `nodeVersion` property as appropriate.

If Mongo/Node/PhantomJS is already installed, you can change the preceding properties to `false` (which will speed up your deployment).

Next, we specify specify what we want our app to be called. This is done by modifying the `appName` property:

```
"appName": "[your-app-name-here]",
```

We need to specify the folder on our local machine where our to-be-deployed app is located. This is determined in the `app` property:

```
"app": "/path/to/the/app",
```

The last property we need to set is `env`. This instructs Node which `ROOT_URL` and `PORT` to run our Meteor application from. If, for example, we were redirecting incoming HTTP traffic to a localhost, on `port 1337` (which is done using a reverse-proxy, such as `nginx`, or a virtual host, such as `apache`), our `env` configuration would look like the following code:

```
"env": {
  "ROOT_URL": "http://localhost",
  "PORT": 1337
},
```

4. Now let's configure our remote server with `mup setup`. In the terminal window, navigate to your project's root folder and enter the following command:

   ```
   $ mup setup
   ```

   This will install MongoDB, Node, and PhantomJS on the remote server. It will also configure our remote server environment and install some helper npm packages, such as `upstart`.

5. Let's deploy our app with `mup deploy`. Once the `mup setup` command is complete, we'll be ready to deploy our app. Execute the following command in the terminal window:

   ```
   $ mup deploy
   ```

MUP will bundle your app locally, upload the build, configure the requisite npm packages on the remote server, and then serve up your app. As it runs through this process, MUP will give you status updates in the terminal.

Once complete, your app should be up and running. Test it out by either visiting your external URL (for example, `http://my-custom-server.com`) or by logging in to a remote server via `SSH`, and testing the build with a `curl` command on the localhost (for example, `curl http://localhost:3000`).

You may have a bit of troubleshooting to do to make sure your virtual host or reverse-proxy is configured properly, but after some small adjustments, you'll find that, at present, MUP is definitely the best way to deploy to a custom server.

How it works...

MUP takes all the steps we would normally have to implement manually (as in the *Deploying to a custom hosted environment* recipe in this chapter) and implements them automatically. There are three main parts to pulling this off successfully.

First, MUP creates a default configuration file when we execute the `mup init` command. We edit the newly created file with all the settings to install the right software on the remote server, configure environment variables, and upload our production build.

Second, we use `mup install` to install all the needed software on the remote server, as specified in the `mup.json` configuration file.

Lastly, our application is bundled, uploaded, extracted, initialized with environment variables, and set to run on the remote server. This is all accomplished with the `mup deploy` command.

MUP performs these tasks based on the configuration file we edited.

> There are many additional settings we can configure and some great features of MUP that we can explore by visiting the MUP repo, which is found at `https://github.com/arunoda/meteor-up`.

See also

- The *Deploying to a custom hosted environment* recipe in this chapter

Using CoffeeScript

Many people prefer CoffeeScript to the standard JavaScript syntax. If you are one of those people with a preference, Meteor has you covered. This recipe will show you how to quickly enable CoffeeScript in your development environment.

Getting ready

Nothing is needed to prepare for this recipe, other than having Meteor installed, and a project created so that you can begin using CoffeeScript.

How to do it...

To use CoffeeScript, proceed with the following steps:

1. Open a terminal window and navigate to the root folder of your project.
2. Enter the following command:

   ```
   $ meteor add coffeescript
   ```

   You should see the following response in the terminal window:

   ```
   coffeescript  added, version 1.0.
   ```

   CoffeeScript is now installed and ready to go!

You can test the installation by creating a `.coffee` file and adding a script (for example, add `test.coffee` to your `client` folder). When you start your Meteor application with the `meteor` command, the script should execute as designed.

You can uninstall CoffeeScript easily. Simply execute the following terminal command:

```
$ meteor remove coffeescript
```

How it works...

Behind the scenes, the CoffeeScript package you just installed looks for any and all `*.coffee` and `*.litcoffee` files, compiles them into native JavaScript, and then bundles them for use in your running Meteor application.

As with all other files in a running Meteor application, changes to your CoffeeScript are immediately processed.

See also

- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*

Using CSS compilers

As any CSS junkie already knows, using standard CSS to create style sheets can be tedious, redundant work. Many designers and developers prefer to use a dynamic style sheet language or preprocessors, such as Less, Stylus, and SCSS/SASS.

Meteor not only enables the use of preprocessors, but also treats them just like any other file so that changes are reflected immediately.

This recipe will show you how to enable some of the more popular CSS compilers in your Meteor application.

Getting ready

Nothing is needed to prepare for this recipe, other than having Meteor installed, and a project created so that you can begin using CSS compilers.

How to do it...

We're going to cover three different preprocessors, as they all work in a similar way.

Using Stylus

1. Open a terminal window and navigate to the root folder of your project.
2. Enter the following command:

   ```
   $ meteor add stylus
   ```

   You should see a response similar to the following in the terminal window:

   ```
   stylus  added, version 1.0.7
   ```

   Stylus is now installed and ready to be used. You can test this by creating a `.styl` file and adding a script (for example, add `test.styl` to your `client/styles` folder).

When you start your Meteor application with the `meteor` command, Stylus files will be processed and proper CSS will be rendered.

You can use the `nib` code in Meteor as well. Just add `@import 'nib'` to your `.styl` files, and Meteor takes care of the rest.

Using Less

1. Open a terminal window and navigate to the root folder of your project.
2. Enter the following command:

   ```
   $ meteor add less
   ```

   You should see the following response in the terminal window:

   ```
   less  added, version 1.0.14
   ```

   Now the Less package is installed, and you can use the Less stylesheet syntax to create your CSS.

As with Stylus, you can test this by creating a `.less` file and adding some style declarations (for example, add `test.less` to your `client/styles` folder).

When you start your Meteor application with the `meteor` command, the Less files will be compiled by Meteor into standard CSS and rendered as usual.

If you're fond of using `@import` statements in your Less stylesheets, make sure you use the `.lessimport` extension. Otherwise, Meteor will automatically import and compile any and all `.less` files it can find.

Using SCSS / SASS

1. Open a terminal window and navigate to the root folder of your project.
2. Enter the following command:

   ```
   $ meteor add fourseven:scss
   ```

   You should see a response similar to the following in the terminal window:

   ```
   fourseven:scss  added, version 2.1.1
   ```

SCSS and SASS files can now be used to style your CSS. Just as before, you can test this by creating a `.scss` or `.sass` file and adding some style declarations (for example, add `test.sass` to your `client/styles` folder).

When you start your Meteor application with the `meteor` command, the SCSS or SASS files will be compiled by Meteor into standard CSS and rendered.

How it works...

When you installed any of the preprocessors with the `meteor add` command, it installed the corresponding npm packages tailored to work inside of Meteor.

As with other files, Meteor will monitor changes to any `*.styl`, `.less`, `.scss`, and `.sass` files, compile the changes into CSS, and render the changes immediately.

See also

- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*

2

Customizing with Packages

In this chapter, we will cover the following topics:

- Adding Meteor packages
- Removing Meteor packages
- Discovering new packages with Atmosphere
- Creating a multipage application with Iron Router
- Building a custom package
- Using npm modules
- Publishing custom packages to Atmosphere

Introduction

The package system in Meteor makes your development life much easier. It also speaks to one of Meteor's core principles—modular development. If you want to use the entire default Meteor stack, great! If you don't like a particular part and want to swap it out with a third-party package, great! It's completely up to you. Meteor allows you to quickly add and remove functionality, use the latest code shared by others, and create your own reusable code segments. This chapter will provide you with the recipes needed to take full advantage of the Meteor packages system.

Adding Meteor packages

The core **Meteor Development Group** (**MDG**) have developed over 140 packages for you to use. These packages provide features and functionality ranging from simple display tweaks, to fully integrated account management. Not only are these packages useful, but they're extremely easy to add to your project. In addition to the core MDG packages, there are hundreds of third-party packages available, all of which are free and could be just as easily added. This recipe will show you how to add Meteor packages to your project.

Getting ready

You will need Meteor installed and have a project created. Any project will do. You should also have a terminal window open and navigate to the root folder of your project, for example, if the name of your project is `packagesTest`, located in the `~/Documents/MeteorProjects` folder, you would enter the following command in a terminal window:

```
$ cd ~/Documents/MeteorProjects/packagesTest
```

How to do it...

Let's install the `fastclick` package as an example. This package removes the 300 ms delay for mobile/touchscreens.

In your terminal window, in the root folder, enter the following command:

```
$ meteor add fastclick
```

This will install the bootstrap package into your project, with a message similar to the following:

How it works...

The `meteor add [package name]` command tells Meteor to find the named package in the Meteor packages registry and copy the appropriate files over to your project. In addition, the package name is added to the declaration files of your project so that when Meteor starts up your project, the appropriate files for the named project are added and executed.

There's more...

As mentioned, you can install third-party packages using the same `meteor add` command. The difference between a core MDG package and a third-party package is that third-party packages have the creator's account ID as a prefix, for example, to add the excellent HammerJS package to your app, you need to enter the following command:

```
$ meteor add chriswessels:hammer
```

Other recipes in this chapter will walk you through discovering and implementing third-party packages, so stay tuned!

See also

- The *Discovering new packages with Atmosphere* recipe in this chapter

Removing Meteor packages

Removing Meteor packages is just as easy as adding them. This recipe will show you how to quickly remove a Meteor package.

Getting ready

You will need Meteor installed and have a project created. You should also have a terminal window open and navigate to the root folder of your project. For example, if the name of your project is `packagesTest`, located in the `~/Documents/MeteorProjects` folder, enter the following command in a terminal window:

```
$ cd ~/Documents/MeteorProjects/packagesTest
```

How to do it...

Let's remove the `insecure` Meteor package. In your terminal window, enter the following command:

```
$ meteor remove insecure
```

This will remove the `insecure` package.

How it works...

The `meteor remove [package name]` command will direct Meteor to look for the named package in your project declaration files and remove the declaration and source files for the package from your project.

There's more...

It's sometimes very helpful to check and see which packages you have added to your project. To see this list, enter the following command in your terminal window:

```
$ meteor list
```

This will give you a quick list of all the packages contained in your project.

Discovering new packages with Atmosphere

Meteor is an emerging platform, growing in popularity every day. The Meteor community is coming up with new packages and integrations with existing JavaScript libraries on almost a daily basis. Because the core Meteor team doesn't have time to test and apply every new package made by the community, a package registry, with a streamlined installation process, has been created. This package registry is called Atmosphere. Atmosphere has a clean, simple UI, which allows you to search for, rate, and discover new packages by popularity. This recipe will show you how to use Atmosphere to find and implement both Meteor and third-party packages.

Getting ready

There's nothing really to do here, but you'll definitely want to have Meteor installed and a project created to start using the packages listed on Atmosphere right away!

How to do it...

To discover new packages with Atmosphere, proceed with the following steps:

1. In a browser, navigate to `https://atmospherejs.com`. You should see a very simple page with a centered search input box. Let's look for the official Twitter `bootstrap` package.

2. Type in the word `bootstrap` and watch the search results appear as you type. You should see something similar to the following screenshot:

You'll notice that each card in the search results has some statistics at the bottom. The down arrow statistic is the number of downloads, which (with the age of the package) is the primary statistic in the search results ranking. You will see a star, which represents the number of developers that have favorited/starred the package. Finally, you will see a gray line in the background, indicating popularity over time.

> The second result (`twbs:bootstrap`) is the official Twitter Bootstrap package.

3. Click on the `twbs:bootstrap` card and an overview of the package will appear. You will see additional information about the package. This includes usage instructions, a link to the GitHub repository (if available), an expanded history of the package, related packages, dependants, and dependencies. At the very top of the page will be the instructions to add the package to your project:

4. You can copy the text and execute it in a terminal window, in your root project folder, as shown in the following code:

   ```
   $ meteor add twbs:bootstrap
   ```

   After a brief download process, the official Bootstrap package will be installed in your project.

   As mentioned in the previous example, Atmosphere has several additional discovery options. There are links to the package contributors' profiles (so you can see other packages they may have contributed to), as shown in the following screenshot:

   There are links to related packages, as shown in the following screenshot:

   There are links to dependants and dependencies as well, as shown in the following screenshot:

Any of these links will help you to discover other packages that will complement the original package and could be useful in your application.

Yet even more options exist for discovering packages in Atmosphere. If you navigate back to the home page at `https://atmospherejs.com/` and scroll down, you will see sections for **Trending**, **Most Used**, **Recent**, and **Top Searches**. All of these are worth perusing to discover some of the latest and greatest packages available for use:

Feel free to explore and discover. You'll find some hidden gems in Atmosphere, and more than anything else, you'll come to realize that a Meteor package exists for almost everything. If it doesn't, create one yourself, with the *Building a custom package* recipe found later in this chapter.

How it works...

The Atmosphere site simply puts a user-friendly wrapper on top of the Meteor package repository. It does this very well, and the contributors/creators of the site are to be commended. Atmosphere is improving over time, and you can contribute to its success if you are so inclined. You can check out what's coming down the pipe by visiting the Atmosphere FAQ, found at `https://atmospherejs.com/i/faq`.

There's more...

If you're in a hurry or really really love the command line, you can perform searches in a terminal window, in the root folder of your project. Here's an example of a CLI-based search:

```
$ meteor search bootstrap
```

The results will be listed, with a brief explanation for each result, in alphabetical order:

Be sure to note that the results from this can be pretty hefty (there's no prioritization of results), so the effectiveness in using a CLI-based search will vary.

See also

- The *Building a custom package* and *Publishing custom packages to Atmosphere* recipes in this chapter

Creating a multipage application with Iron Router

Iron Router is an extremely useful Atmosphere package. It allows you to quickly and easily add multiple server pages to your Meteor project. This recipe will show you how to configure your project to use Iron Router.

Getting ready

You need to have Meteor installed. You will also need a blank Meteor project (see the *Setting up your project file structure* recipe in *Chapter 1, Optimizing Your Workflow*).

Finally, you will need to add the Iron Router package to your Meteor project. You can reference the *Adding Meteor packages* recipe found previously in this chapter, or enter the following command in your terminal window, in the root folder of your project:

```
$ meteor add iron:router
```

How to do it...

We are going to create a very simple example of multiple pages and we will use a fresh blank project to do so.

1. First, set up the route paths for the multiple pages. Create a subfolder called `router` in the `both` subfolder of your project. This can be done in a terminal window with the following command after navigating to the root folder of your project:

   ```
   $ mkdir both/router
   ```

2. Now, create a new file named `router.js` in the `router` folder. Open the `router.js` file in a text editor, add the following code, and save your changes:

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

   Router.map(function (){
       this.route('thing1', {
       path: '/'
       });
     this.route('thing2', {
       path: '/second'
       });
   });
   ```

3. Next, open your `main.html` file, located in the `client` subfolder, in an editor. Add the following template declarations to the bottom of the file and save your changes:

   ```
   <template name="routeexample">
   {{yield}}
   </template>

   <template name="thing1">
   <div> I am the FIRST page</div>
   </template>

   <template name="thing2">
   <div> I am the SECOND page</div>
   </template>
   ```

4. Now, open the `main.js` file found at `[project folder]/client/scripts/main.js` in an editor. Locate the `Template.hello.events` method and change the `'click button'` function body to the following:

```
Template.hello.events({
    'click button' : function () {
      Router.current().route.getName()=='thing1'?
Router.go('thing2'): Router.go('thing1');
    }
  });
```

5. Save your changes to all the files and run your application using the `meteor` command.

   When you navigate to your project's home page in a browser (`http://localhost:3000/`), you should see something similar to the following screenshot:

6. Click on the button marked **Click Me**, or navigate to `http://localhost:3000/second/` and the screen should change to the following:

7. You can continue to toggle between the two pages by clicking on the onscreen button.

How it works...

Iron Router uses the path in your URL to invoke events in both the client and server. In most cases (unless you explicitly set the Iron Router path to render on the server, using the `where` parameter) the events and subsequent rendering happen in the client only.

In essence, Iron Router uses the path as an arguments list, invoking functions and rendering appropriate templates based on the information contained in the path.

Let's break down the code we just entered to get Iron Router up and running, starting with `router.js`:

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

This code snippet tells Iron Router to use the `<template>` tag named `routeexample` as the main/default template for the page.

We created the `routeexample` template in `main.html`, like so:

```
<template name="routeexample">
{{yield}}
</template>
```

The `yield` statement found in this template is known as the main yield. It is a placeholder for other templates that will be rendered once Iron Router completes its mapping process.

Continuing, in `router.js`, we have the `Router.map` function:

```
Router.map(function (){
  this.route('thing1', {
    path: '/'
  });
  this.route('thing2', {
    path: '/second'
  });
});
```

This creates two mapped routes, using the `this.route` function. In each function call, the first parameter specifies the name of the route, and because we didn't specify a specific template, it also implies the name of the template to use. In this case, we used the `thing1` and `thing2` templates, which we created in `main.html`:

```
<template name="thing1">
<div> I am the FIRST page</div>
```

```
</template>

<template name="thing2">
<div> I am the SECOND page</div>
</template>
```

These are very simple templates, and this is what makes them easy to understand for our example. Each template renders a simple `div` element with a message inside of it, indicating what page the user is on.

Going back to `router.js`, each `this.route` function call has a path argument inside of it. This path argument tells Iron Router what to look for in the URL to determine mapping. In the case of our first page (rendering the `thing1` template), the `path` element was set to "/", or the root path. In the case of our second page (rendering the `thing2` template), the `path` was set to "/second".

At this point, Iron Router was properly mapped, meaning this will have rendered the appropriate template based on the mapped routes. We wanted to make switching back and forth between the two routes easy, so we modified the `main.js` script as follows:

```
Template.hello.events({
    'click button' : function () {
      Router.current().route.getName()=='thing1'?
Router.go('thing2'): Router.go('thing1');
    }
  });
```

Now, when our button is clicked, we are checking to see whether the name of the currently mapped route (found using `Router.current().route.getName()`) is `thing1`. If it is, we tell Iron Router to change the currently rendered template (and the URL path) to our second template, named `thing2`, with a `path` value set to "/second". If the currently mapped route isn't `thing1`, then we toggle back to the original mapped route.

The `Router.go` statement changes the URL path, thereby setting events in motion to render the appropriate template and insert it into the main yield.

There's more...

Iron Router is a very sophisticated and powerful package, with many helpful features including pre-rendering events, setting data context, and access to server-side functions. If you are looking to build a multipage Meteor application, you will want to peruse the full documentation, found at `https://github.com/EventedMind/iron-router`.

See also

- The *Creating a complete app with Iron Router* recipe in *Chapter 12*, *Creating Useful Projects*

Building a custom package

As you become more familiar with Meteor, you will want to start creating your own custom package, which consolidates the code you may find useful in multiple projects. This recipe will walk you through the basics of creating your own personal Meteor package.

Getting ready

The only thing you need for this recipe is Meteor and a text editor.

How to do it...

We will create a package that will allow us to easily write to the console in not only the client web console, but also in the server terminal console.

1. To create a new baseline package, open a terminal window and navigate to where you would like your package to reside. Once there, execute the following command:

   ```
   $ meteor create --package [myMeteorID]:testpack
   ```

   Make sure you replace `[myMeteorID]` with your own Meteor Developer Account ID or with the ID of the Meteor organization you belong to, for example (and for the rest of this chapter), we will use the `packtmeteor` organization. So our command will look like the following:

   ```
   $ meteor create --package packtmeteor:testpack
   ```

2. We will now add the code for our package. Open the file named `testpack.js` and add the following code:

   ```
   TestPack = {
     log: function(msg){
       console.log(msg);
       Meteor.call('serverlog',msg);
     }
   };
   ```

Save your changes and create a new file in the same folder named `servertestpack.js`. Open this file for editing and add the following code:

```
Meteor.methods({
  serverlog : function(msg){
    console.log(msg);
  }
});
```

3. We now need to make a few modifications to `package.js`, to make sure our package will work properly. Open `package.js` and modify the `Package.onUse()` method to look like the following:

```
Package.onUse(function(api) {
  api.versionsFrom('1.0.3.2');
  api.addFiles('testpack.js', 'client');
  api.addFiles('servertestpack.js', 'server');
});
```

Next, add the following code inside the `Package.onUse()` method, just after the last `api.AddFiles()` call:

```
Package.onUse(function(api) {

  api.addFiles('servertestpack.js', 'server');

  if (api.export){
    api.export('TestPack');
  }
});
```

Finally, modify the `Package.onTest()` method by removing the prefix from the test's filename, like so:

```
Package.onTest(function(api) {
  api.use('tinytest');
  api.use('packtmeteor:testpack');
  api.addFiles('testpack-tests.js');
});
```

4. Save all your changes, and your package will be ready to be used in an application. Select a project you have underway, or create a default Meteor project using the `meteor create` command, and add a subfolder named `[project root]/packages/`. Copy the `testpack/` folder and paste it into the `[project root]/packages/` folder. Then, in a terminal window, execute the following command:

   ```
   $ meteor add packtmeteor:testpack
   ```

   Your custom package will be registered for use in your project, and you can now start your application:

   ```
   $ meteor
   ```

   Open a browser, navigate to `http://localhost:3000` and open the web console. In the web console, enter the following command:

   ```
   > TestPack.log('my package is worky!')
   ```

   You should see the following screen in the browser console:

   You should see the following in the server terminal console:

Congratulations! You've just created your own personal package.

How it works...

Packages in Meteor are really just collections of organized files, with a configuration file that tells Meteor how to use those files.

In this case, we created a default package template using the `meteor create --package` command. Note that we followed the `[owner]:[package name]` naming convention, which Meteor requires if we want to publish this package on Atmosphere for others to use (refer to the *Publishing custom packages to Atmosphere* recipe in this chapter for details).

Once created, we added our functionality, which was very simple. We just called `console.log()` on the client and on the server.

Then, we modified the configuration file, which is always named `package.js`. The following table shows the three sections that this file contains:

`Package.describe()`	This contains the name, summary, and version number of our package.
`Package.onUse()`	This lists dependencies and exposes the methods available in our package.
`Package.onTest()`	This lists dependencies and test methods to be run during testing.

For this recipe, we didn't modify the `Package.describe()` method. Meteor autocreated this section for us and gave us a *version number*, which is required.

Version numbers for packages follow the **Semantic Versioning Specification (SemVer)** specification (3 digits + an optional wrap, such as `1.0.2-rc_beta`). More information on the SemVer specification can be found at `http://semver.org/`

The bulk of our changes occurred in `Package.onUse()`, so we'll go through each item in this method.

The `api.versionsFrom()` method specifies which version of Meteor the core packages used in your package should come from. This comes in handy because Meteor is updated pretty regularly, and some of the dependencies needed to run your package may change in later versions. This method allows you to *freeze* the release versions of the dependency packages you use in your app to prevent compatibility issues.

The `api.addFiles()` method allows us to include files found in our package directory and to specify whether they are client or server-specific. In our case, we used `api.addFiles()` with a `client` parameter to tell Meteor that the `testpack.js` file was client-specific. We then used a `server` parameter to tell Meteor that the `servertestpack.js` file was server-specific.

We then used the `api.export()` method to declare/expose the `TestPack` object. This is what enabled us to use the `TestPack.log()` call in the browser's web console.

Finally, we have the `Package.onTest()` method. It contains two `api.use()` method calls, which declare the package dependencies needed to run the `testpack-test.js` file during testing.

The `api.use()` method isn't exclusive to testing. You can use it in the `Package.onUse()` method as well to include the MDG core or third-party packages.

After completing our modifications, we copied our entire package over to the `packages/` subfolder in our project. This is necessary for any non-deployed package. When you use the `meteor add` command to add a package, Meteor checks for the package in two places: in the online Meteor package repository (Atmosphere) and in the `[project root]/packages/` subfolder. In this case, because our package isn't deployed to Atmosphere, we had a copy of our package in the `packages/` subfolder.

There's more...

Several other options exist for configuring packages (some of which are included in other recipes in this chapter). A complete list of these can be found in the Meteor documentation, found at `http://docs.meteor.com/#/full/packagejs`.

See also

- The *Using npm modules* and *Publishing custom packages to Atmosphere* recipes in this chapter
- The *Building custom server methods* recipe in *Chapter 11*, *Leveraging Advanced Features*

Using npm modules

Node Package Manager (**NPM**) has thousands of modules available. Knowing how to add an npm module for use in your Meteor project gives you access to all of those modules. This recipe will show you how to do this.

Getting ready

You will need to make sure that you have Meteor and Node/npm installed.

You will also need a project that makes use of a custom-built package. We will be using the project from the *Building a custom package* recipe, found in this chapter.

How to do it...

We first need to add a reference to the npm package that we would like to add, which in this case is going to be the `colors` module.

1. Inside your custom package (the one in the `packages/` subfolder of your app), in the `package.js` file, add an `Npm.depends` statement to the end of the file, as shown in the following example:

   ```
   Npm.depends({
   ```

```
    "colors": "0.6.2"
});
```

2. Now we need to add an `Npm.require` reference to the `colors` module and change our `console.log()` command to use a rainbow of fruit flavors. Change your `server-test.js` file so it looks like the following example:

```
var colors = Npm.require('colors');

Meteor.methods({
  serverlog : function(msg,warn){
    console.log(msg.rainbow);
  }
});
```

3. Save your changes and we are ready to test our updated package. Navigate to `http://localhost:3000` in a browser, open the browser web console, and execute the following command a few times:

```
> TestPack.log('rainbows and unicorns')
```

4. If you switch to your console window, you should see a multicolored response, as shown in the following screenshot:

```
packtmeteor:testpack: updating npm dependencies -- colors...
=> Meteor server restarted
=> Meteor server restarted
I20150313-00:12:48.739(-6)? rainbows and unicorns
I20150313-00:12:49.306(-6)? rainbows and unicorns
I20150313-00:12:49.801(-6)? rainbows and unicorns
```

How it works...

The `Npm.depends` declaration in `package.js` tells Meteor that whenever it starts, it needs to go and install the npm modules listed. In this case, we are telling Meteor to go and fetch the `colors` module, version `0.6.2`, from the npm registry.

The `Npm.require` function call in `server-test.js` makes the `colors` module functionality available on the server. Now, any time we make a `console.log()` call, the string passed into the call can have a `.color` reference added to it. In our case, we chose `.rainbow`, which makes every letter from the string printed out to the console appear in a different color.

To see all the various options available in the npm `colors` module, visit `https://npmjs.org/package/colors`.

There's more...

You can use any npm module you would like in a custom Meteor package. Some of them rely on asynchronous methods and events, so you will have to wrap those modules with specific asynchronous Meteor methods. You can read more about these methods in *Chapter 11*, *Leveraging Advanced Features*.

See also

- The *Using npm packages directly* recipe in *Chapter 8*, *Integrating Third-party Libraries*
- The *Handling asynchronous methods and using asynchronous functions* recipe in *Chapter 11*, *Leveraging Advanced Features*

Publishing custom packages to Atmosphere

When you've created a useful package and want to share it with the world, Atmosphere is the way to go. This recipe will show you how to upload your custom package to Atmosphere and provides some best practices for maintaining your package moving forward.

Getting ready

To create a quick Atmosphere package, please complete the *Using npm modules* recipe found in this chapter.

You will also need to create a blank repository on GitHub. Assuming you have a GitHub login, navigate to the home page (`https://github.com`) and create a new repository:

Add a title and description to your repository and click on the button marked **Create repository**.

To populate your new Git repository, you will need the Git URL, which can be copied from the home page of your new Git project. It should look something like `https://github.com/strack/packt-testpack.git`.

Remember this URL, as you will be using it to populate the content of your new repository.

How to do it...

To register and upload our package, we need to check our configuration file, upload our code to GitHub, and then deploy our package.

1. To configure `package.js`, open the `package.js` file found in the root folder of your package. In our case, the file is located at `[project root]/packages/testpack/package.js`. We will first modify the summary, as shown in the following example:

   ```
   // Brief, one-line summary of the package.
   summary: 'Demonstration of an Atmosphere Package',
   ```

   Next, we will add the GitHub URL to the `git` property:

   ```
   // URL to the Git repository...
   git: 'https://github.com/strack/packt-testpack.git',
   ```

   You will notice that the documentation property references the `README.md` file, which means that our GitHub documentation will serve a dual purpose. Open `README.md` and add the following line:

   ```
   Demonstration of an Atmosphere Package, deployed to Atmosphere
   ```

2. To push the code to GitHub, save all of your files and navigate to the root folder of your package (not the root folder of your application) in a terminal window. In our case, the root folder is `[project root]/packages/testpack/`. Enter the following lines to populate the GitHub repository:

   ```
   $ git init
   $ git add -A
   $ git commit -m "Created color server console"
   $ git remote add origin https://github.com/strack/packt-testpack.
   git
   $ git pull origin master –m "initial pull"
   $ git push origin master
   ```

3. To publish to Atmosphere, call `meteor publish` with the following command once it is completed:

   ```
   $ meteor publish --create
   ```

   Voila! You have added your package to Atmosphere.

We deployed all packages directly from the `packages` subfolder of an application. A best practice is to develop and deploy your package on its own, independent of any application. Refer to the *Building a custom package* recipe.

You can see your package live on Atmosphere by navigating to `https://atmospherejs.com` and searching for it by name, as shown in the following screenshot:

As with any other Meteor package found in Atmosphere, you can add it to any of your projects using the `meteor add` command in the terminal.

How it works...

Atmosphere reads the `package.js` file for the data that needs to be published, checks to make sure all information is accurate, and then publishes a link to your project from a GitHub repository. In this specific case, we added a summary to our file, specified the GitHub URL in the `git` property, and (after creating the GitHub repository) called `meteor publish` to deploy packages.

There's more...

More instructions on how to publish to Atmosphere can be found at `https://atmospherejs.com/i/publishing`.

See also

- The *Adding Meteor packages* and *Discovering new packages with Atmosphere* recipes in this chapter

3

Building Great User Interfaces

In this chapter, we will cover the following topics:

- Inserting templates with Spacebars
- Inserting raw HTML using triple braces
- Creating dynamic lists
- Building a smooth interface with Bootstrap
- Creating customized global helpers
- Creating custom components
- Using reactivity with HTML attributes
- Using dynamic templates
- Animating DOM elements

Introduction

Like everything else in Meteor, UI development is designed to be simple, elegant, and powerful. With just a few imperative commands, you can create sophisticated and dynamic user interfaces, significantly speeding up your prototyping/wireframing efforts.

The recipes in this chapter will walk you through the most common UI- and template-building techniques, which will form the foundation for your Meteor apps moving forward.

Inserting templates with Spacebars

Spacebars is Meteor's templating language. All **Document Object Model** (**DOM**) elements in Meteor are created programmatically using Meteor's default templating engine, **Blaze**. Elements can therefore be created using pure JavaScript. Using JavaScript to create HTML elements, however, isn't exactly quick and easy.

Inspired by the popular HTML templating language **Handlebars**, Spacebars makes it incredibly easy to structure and declare your HTML elements using templates. And, it gives you access to all the reactive and dynamic features of Meteor. This recipe will show you how to create templates using the Spacebars syntax.

Getting ready

You will need Meteor installed and have a project created. We recommend that you have a simple project, similar to the one found in the *Setting up your project file structure* recipe in *Chapter 1*, *Optimizing Your Workflow*, but any project will do. You will need to locate your `<body>` tag, which is usually found inside your `main.html` file.

How to do it...

To make this recipe just a little bit more useful, we are going to create two templates and nest one inside of the other, using the following steps:

1. First, create a file named `templates.html` and add it to your `client` folder:

2. Open `templates.html` in an editor, add the following code, and save your changes:

```
<template name="firstTemplate">
    <div id="title">This is my first ever template</div>
    <svg height="200" width="100%" id="shapes">
        {{> one}}
    </svg>
</template>


<template name="one">
    <circle cx="100" cy="50" r="40" stroke="#ffc348"
stroke-width="3" fill="#31be4f" />
</template>
```

3. Next, open your `main.html` file (or the `.html` file that contains your `<body>` tag) and edit your `body` tag by adding the following code:

```
<body>
  {{> firstTemplate}}
</body>
```

4. Save the changes, start Meteor if it isn't already running, and navigate to your Meteor-hosted page in a browser (usually `http://localhost:3000`):

How it works...

We created two templates, with the first one being a very simple `svg` element:

```
<template name="one">
    <circle cx="100" cy="50" r="40" stroke="#ffc348"
stroke-width="3" fill="#31be4f" />
</template>
```

We gave the template a `name` attribute value as `"one"` and added the well-formed HTML `<circle>` element with position and appearance attributes. We also created a parent template, as follows:

```
<template name="firstTemplate">
    <div id="title">This is my first ever template</div>
    <svg height="200" width="100%">
        {{> one}}
    </svg>
</template>
```

This template also has a name (`"firstTemplate"`) and contains a `<div>` element, containing the title of our template. We declared an `<svg>` element, and we inserted our template named `one` between the `<svg>` and `</svg>` tags using the `{{>}}` command.

Finally, in `main.html`, we inserted the `firstTemplate` template between the main `<body>` and `</body>` tags, again using the `{{>}}` command, which is also known as **template inclusion**.

As you can see, the content of our templates consists primarily of HTML tags. This is the fundamental building block of our UI, and it's how we've been creating templates from the very beginning. Anything contained inside double braces (`{{...}}`) is a command meant for Blaze, Meteor's templating engine. In this case, we told Blaze to:

1. Insert the `firstTemplate` template into our `<body>` element.
2. Insert the `one` template into our `<svg>` element, inside `firstTemplate`.

There's more...

You can read all about templates in the official Meteor documentation, found at `http://docs.meteor.com/#/full/templates`.

Inserting raw HTML using triple braces

Occasionally, you will want to insert raw HTML into your Meteor page. This is typically done when you have HTML generated by another process or library. This recipe will show you how to render your raw HTML inside a Meteor template.

Getting ready

You will need Meteor installed and have a project created with at least one template in it. We recommend that you use the files created in the *Inserting templates with Spacebars* recipe found in this chapter.

If you don't have a `templates.js` file in your `client` folder already, you should create one to keep your template `helpers` and `events` logic separate from other JavaScript you may be running.

How to do it...

Identify the name of your root template and open the file containing your template definition. In our case, the root template is named `firstTemplate`, which can be found in our `templates.html` file, which is in our `[project root]/client` folder. Proceed with the following steps to insert raw HTML using triple braces:

1. Insert the following line, just before the `</template>` tag, and save your changes:

```
<template name="firstTemplate">
    ...
    {{{rawInsert}}}
</template>
```

Now, open your `templates.js` file and add the following helper function:

```
Template.firstTemplate.helpers({
  rawInsert: function () {
    return "<div><strong>Raw HTML!</strong></div>";
  }
});
```

2. Save your changes, start Meteor if it isn't already running, and navigate to your Meteor-hosted page in a browser (usually `http://localhost:3000`). You should see the rendered HTML at the bottom of your screen, as shown in the following screenshot:

How it works...

We created and then called our `Template.firstTemplate.rawInsert` helper function by adding the `{{{rawInsert}}}` tag at the bottom of our template and declaring the `rawInsert` helper in our `templates.js` file.

The triple-braced tag informs Blaze that the object returned from the helper will be raw HTML and needs to be rendered as such. Accordingly, Blaze expects a string and will attempt to render it.

To see how Blaze would normally treat raw HTML, change the triple-braced tag to a double-braced tag (`{{rawInsert}}`) save your file, and view the result in your browser, as shown in the following screenshot:

See also

- The *Creating dynamic graphs with SVG and Ajax* recipe in *Chapter 7*, *Using Client Methods*

Creating dynamic lists

The Meteor Spacebars syntax supports the rapid development of lists, tables, and other displays of data through the use of template block tags. Used within templates, block tags can quickly add dynamic lists and even customize the look and feel of each element, based on its unique attributes. This recipe will show you how to use the `#each` block tag to display multiple elements of a collection inside a Blaze template.

Getting ready

We will use the code from the *Inserting templates with Spacebars* recipe, found in this chapter, as our baseline project.

If you haven't already done so, create and save a new file in your `[project root]/client` folder called `templates.js`.

Create and save a new file in your `[project root]/both` folder called `collections.js`.

We will want to make one modification to the HTML in the `templates.html` file. We will change the height of our `<svg>` element from `200` to `800`:

```
<svg height="800" width="100%" id="shapes">
```

Finally, we want to add the `random` package to our project to help us generate some random colors and positions. In a terminal window, navigate to your project root and enter the following command:

```
$ meteor add random
```

How to do it...

To create a dynamic list, proceed with the following steps:

1. First, let's declare a `shapes` collection. Open `collections.js`, add the following code, and save your changes:

   ```
   Shapes = new Mongo.Collection('shapes');
   ```

2. Now, let's add shapes to the page dynamically. Open your `templates.html` file, make the following changes to the `one` template, and save your changes:

   ```
   <template name="one">
       {{#each svgShapes}}
       <circle cx="{{x}}" cy="{{y}}" r="40" stroke="{{border}}"
   stroke-width="3" fill="{{color}}" />
       {{/each}}
   </template>
   ```

3. We will now create the template helper, which returns our dynamic collection. Open `templates.js`, add the following helper function, and then close and save your changes:

   ```
   Template.one.helpers({
     svgShapes: function () {
       return Shapes.find();
     }
   });
   ```

4. We can now dynamically add shapes to the `shape` collection, and they will be rendered on our page. Open the JavaScript console in your browser and enter the following command:

```
> Shapes.insert({x:200,y:50,border:'#123456',color:'#bada55'})
```

You should see a green circle with a dark border appear on the screen, similar to the following screenshot:

Let's add more circles to prove that our template is rendering all the shapes in the `Shapes` collection.

5. In `templates.js`, add the following code:

```
Template.firstTemplate.events({
  'click svg': function (e,t){
    var xpos,ypos;
    if(e.offsetX==undefined)
    {
      xpos = e.pageX-$('#shapes').offset().left;
      ypos = e.pageY-$('#shapes').offset().top;
    }
```

```
    else
    {
      xpos = e.offsetX;
      ypos = e.offsetY;
    }
        choices = ['#bada55','#B43831', '783BA3',
'#00AB1B', '#143275', '#FFA700'],
        color = Random.choice(choices),
        border = Random.choice(choices);
    Shapes.insert({x:xpos,y:ypos,border:border,color:color});
    },
    'dblclick': function(e,t){
        Meteor.call('resetShapes');
    }
});
```

6. We now want to display the total number of shapes, so we need to add a `shapeCount` function to the `Template.firstTemplate.helpers` method call:

```
Template.firstTemplate.helpers({
  shapeCount: function () {
    return Shapes.find().count();
  },
  rawInsert: function () {...
We will also need to Modify templates.html to display a count of
the shapes collection:
<template name="firstTemplate">
    <div id="title">shapes collection count: {{shapeCount}}</div>
```

7. Let's add a `resetShapes` method for good measure. Open/create the `server.js` file, found in your `[project root]/server` folder, and add the following code:

```
Meteor.methods({
    resetShapes: function(){
        Shapes.remove({});
    }
});
```

8. Save all your changes, start Meteor if it isn't already running, and navigate to your Meteor-hosted page in a browser (usually `http://localhost:3000`).

Now, as you click in your browser, new circles will appear and the upper-left part of the screen will display your `shapes` collection count:

Feel free to add as many shapes as you would like or reset back to zero by double-clicking anywhere on the screen.

How it works...

We created the `Template.one.svgShapes` helper in `templates.js`, which returns a cursor to the `shapes` collection. This sets the data context for our template and is accessible using the `{{#each svgShapes}}` block tag, which we added to our `one` template in `templates.html`.

Because templates and template helpers are reactive computations, they react when the collection is updated. This reaction causes Blaze to re-render the page, using the dynamic template found within the `#each` block tag.

There's more...

Other block helpers exist and they are just as easy to use. Refer to the Spacebars documentation on GitHub for a more detailed explanation, which is available at `https://github.com/meteor/meteor/blob/devel/packages/spacebars/README.md`.

See also

- The *Implementing a simple collection* recipe in *Chapter 4*, *Creating Models*
- The *Creating dynamic graphs with SVG and Ajax* recipe in *Chapter 7*, *Using Client Methods*

Building a smooth interface with Bootstrap

As a web developer, one of the biggest drains on your time comes from styling your pages. Bootstrap offers an elegant, simple design with enough ready-to-use component styles to get you up and running quickly, without having to get bogged down writing CSS. This recipe will walk you through a few of the Bootstrap component styles and give you a sampling of how easy they are to use.

Getting ready

We will be using the *Creating dynamic lists* recipe found in this chapter as a baseline for our code.

We will also need to add the official Bootstrap package to our Meteor project using Meteor's package repository. Open a terminal window, navigate to the root folder of your project, and enter the following command:

```
$ meteor add twbs:bootstrap
```

There are other packages that are complementary to the official Bootstrap package. You can find a complete list of these at `https://atmospherejs.com/?q=bootstrap`.

How to do it...

We will make our title and collection count presentable using the `btn` and `navbar` styles built in to Bootstrap. Proceed with the following steps to build a smooth interface with Bootstrap:

1. First, let's modify our title, adding a sub `<div>` tag and styling it like a Bootstrap button. Open `templates.html`, make the following code changes, and save your file:

```
<div id="title">Shapes Collection Count:
    <div id="resetShapes" class="btn btn-warning">
       {{shapeCount}}
       </div>
   </div>
```

2. After this change, your title should look much more presentable, as shown in the following screenshot:

3. We will now add the `navbar` style to the entire title, giving it a distinct, intentional look. Open `templates.html` again and replace the title `<div>` tag with the following code:

```
    <nav class="navbar navbar-default">
        <div class="container-fluid">
            <div class="navbar-header">
                <div id="title" class="navbar-brand">Shapes
Collection Count</div>
            </div>
            <div class="nav navbar-left">
                <div id="resetShapes" class="btn btn-warning
navbar-btn">
                    {{shapeCount}}
                </div>
            </div>
        </div>
    </nav>
```

4. Save these changes and your page should be updated. The **Shapes Collection Count** section should now appear in a distinct `navbar` style, similar to the following screenshot:

5. Lastly, let's turn our count button into an actual button. Open `templates.js` and change the `dblclick` event to a specific `click` event. This event is placed at the bottom of the `Template.firstTemplate.events` function:

```
Template.firstTemplate.events({
    …
    'click #resetShapes': function(e,t){
        Meteor.call('resetShapes');
    }
})
```

We have now disabled double-click as the method of resetting our `shapes` collection and transferred it to the **Shapes Collection Count** total. By clicking on the orange button, the collection will reset to zero:

How it works...

Changing the look and feel of a page using Bootstrap is simply a matter of adding the correct CSS `class` declarations and structuring our HTML appropriately.

To style the `shapes` collection count, we added the `btn` and `btn-warning` value to the `class` attribute. This allows the Bootstrap CSS to appropriately style the `<div>` tag, making it appear as an orange button.

To style our title, we had to first create the scaffolding for the `navbar` tag. We did this by essentially wrapping our `<div>` tag with a `<nav>` tag. We then appropriately structured the containing tags by adding `nav`, `navbar`, `navbar-header`, and `navbar-left` to the `class` attributes. The other `class` declarations (`navbar-default`, `navbar-brand`, and `navbar-btn`) are strictly style declarations, changing the color/shape but not affecting the overall structure or layout.

There's more...

Bootstrap is very robust and comes with excellent documentation. You can learn more about Bootstrap at `http://getbootstrap.com/`.

See also

- The *Adding Meteor packages* and *Discovering new packages with Atmosphere* recipes in *Chapter 2*, *Customizing with Packages*

Creating customized global helpers

As you begin to really dig into templates, you'll soon discover how truly great template helpers are. You can access data inline and drastically reduce the amount of code you need to write.

You will, however, eventually run into a situation where you find yourself repeating the same logic in multiple, in-template helpers. Wouldn't it be great if there was a way to create global helpers accessible from any template? Well, Meteor has a solution for this too! This recipe will show you how to create global template helpers using the `Template.registerHelper` function.

Getting ready

We will use the codebase from the *Building a smooth interface with Bootstrap* recipe, found in this chapter. Please follow that recipe first or download the corresponding codebase.

How to do it...

We're going to make a global random color generator and add random colors to all the objects on the screen by performing the following steps:

1. Open `templates.html`, found in your `[project root]/client` folder, and make the following changes to the color attributes of the `one` template:

```
<template name="one">
    {{#each svgShapes}}
    <circle cx="{{x}}" cy="{{y}}" r="40"
stroke="{{randColor}}" stroke-width="3"
fill="{{randColor}}" />{{/each}}
</template>
```

2. We will also use the `randColor` helper in our `<div>` title to give it a bit of (random) color. Inside the `firstTemplate` template, look for and modify the `<div>` title as follows:

```
<div class="navbar-header">
                <div id="title" class="navbar-brand"
style="color:{{randColor}}">Shapes Collection Count</div>
```

3. Save your changes and start your Meteor app if it isn't already started. As you view your Meteor page in your browser, you will notice that the circles have all turned black. This is because although we are referencing the global helper, we haven't actually created it yet.
4. So, let's take care of that. Open `templates.js`, found in your `[project root]/client` folder, and add the following function at the bottom:

```
Template.registerHelper('randColor',function(){
    choices = ['#bada55','#B43831', '783BA3', '#00AB1B',
'#143275', '#FFA700'];
    return Random.choice(choices);
});
```

5. Once you save the changes, your page will refresh and the circles will now have a random color, as will the title bar of your app. To prove that it is indeed using the global `randColor` helper, open another browser window, navigate to your page (usually `http://localhost:3000`), and view the results side by side. As you click on the page to add circles or refresh the page, the colors will change at random and will be different from one browser window to the next, as shown in the following screenshot:

How it works...

The `Template.registerHelper` function declares a helper that can be seen in context with any template. Think of it as a way to declare global functions, accessible within the Spacebars syntax.

The first argument of `Template.registerHelper` is a string value, which contains the name of our helper (the name to be used to reference it inline, inside our templates). When deciding on a name for a global helper, keep in mind that it is indeed global. It should therefore be unique, to prevent conflicts, or the value could be overridden by a template-specific helper. Our example is very simple, so we can get away with `randColor`; however, as you build more sophisticated templates, it's a good idea to give your helpers more distinct names. The value returned is obviously a randomly selected color, using the same technique that we used in the original `firstTemplate` click event handler.

We then used the global helper in two different templates. First, we replaced `{{color}}` and `{{border}}` in the circle's `svg` declaration with `{{randColor}}`, causing our circles to appear with a random color every time. Finally, we added a `style="color:{{randColor}}"` attribute to our `<div>` title, which randomly assigns a text color upon page refresh.

There's more...

There are obviously some very advanced techniques you can accomplish with global helpers, using other template-related methods such as `Template.instance()` and `Template.currentData`. All of these can't be covered in this recipe, but we encourage you to explore the Meteor documentation to familiarize yourself with what global helpers can do.

For a good overview of the available template utilities, consult the Meteor documentation at `http://docs.meteor.com/#/full/templates_api`.

See also

- The *Inserting templates with Spacebars* and *Using reactivity with HTML attributes* recipes in this chapter

Creating custom components

Adding and using templates in Meteor is a great way to speed up your workflow. Using Meteor's built-in content block helpers can further streamline your workflow by allowing nested and deferred template calls. In essence, you can combine and nest templates to create your own "components". This recipe will show you how to build a basic Spacebars custom block helper using `Template.contentBlock`.

Getting ready

We will use the codebase from the *Building a smooth interface with Bootstrap* recipe found in this chapter. Please follow that recipe first, or download the corresponding codebase.

How to do it...

Proceed with the following steps to create custom components:

1. We need to make the template-specific `shapeCount` helper global so we can use it inside our component. In `templates.js`, delete the `shapeCount` function found in the `Template.firstTemplate.helpers()` method call, and add the following function to the bottom of the page:

```
Template.registerHelper('shapeCount',function (){
  return Shapes.find().count();
});
```

2. Now, we will create our component. We will first create a `navbutton` template, which will use the newly created `shapeCount` helper. Create a new template file, named `navbar.html`, in your `[project_root]/client` folder. Open this file in a text editor and add the following template code:

```
<template name="navbutton">
    <div class="nav navbar-left">
        <div id="resetShapes" class="btn btn-warning
        navbar-btn">
            {{shapeCount}}
        </div>
    </div>
</template>
```

3. To create a custom `navbar` block tag, add the `navbar` template in the same file, complete with a `Template.contentBlock` reference, as follows:

```
<template name="navbar">
    <nav class="navbar navbar-default">
        <div class="container-fluid">
            <div class="navbar-header">
                <div id="title" class="navbar-brand">Shapes
                Collection Count</div>
            </div>
            {{> Template.contentBlock}}
        </div>
    </nav>
</template>
```

4. Our component is now complete! Let's use it in our code. Open `templates.html` and completely remove the `<nav>` element block, including the code inside the `firstTemplate` template. The only element left after this change will be the `<svg>` element. Just above the `<svg>` element, we are going to call our `navbar` template using the `{{#[template name]}}` block tag, as follows:

```
<template name="firstTemplate">
    {{#navbar}}
      {{> navbutton}}
    {{/navbar}}
    <svg>
```

5. Save all your changes, start your Meteor instance if it isn't already running, and view the results in your browser. You will notice that nothing has changed. We have the same functionality as before; only now, we have created a custom Spacebars component (formally, a custom block helper), and we could add more components to the `navbar` tag as needed or enter explicit HTML, for example, replacing `{{> navbutton}}` with `<div>{{shapeCount}}</div>` will display the **Shapes collection count** without the use of the `navbutton` template, similar to the following screenshot (not very pretty, but you get the point):

If you inspect the element in your browser, you will notice that manually entered `<div>...</div>` has been inserted in the placeholder block, similar to the following example:

```
<nav class="navbar navbar-default">
  ...
    <div>15</div>
  ...
</nav>
```

How it works...

The key to building block helpers resides in the `Template.contentBlock` function. We created a new `navbar` template and used `{{> Template.contentBlock}}` inside that template to specify a placeholder. This placeholder (called a **content block**) can be specified later, rather than having to be specified explicitly inside the `navbar` template. This simple addition of a content block turns our `navbar` template into a *component*.

Once we created our block helper, along with the template meant to go inside the block helper, we added a `Spacebars` block tag with `{{#navbar}}` and the ending `{{/navbar}}` calls. Inside this block, we called our new `navbutton` template using `{{> navbutton}}`, which is inserted in the `{{> Template.contentBlock}}` placeholder. At runtime, all these template functions get executed in order, and Blaze renders the actual HTML DOM elements.

There's more...

Custom block helpers can also utilize the `#if`, `#else`, and `#unless` logic through the use of `UI.elseBlock`. Using these commands, you can create reusable, dynamic block helpers and templates, further streamlining your development process.

> For a thorough walkthrough of all the options available, see the GitHub documentation on Spacebars, found at `https://github.com/meteor/meteor/blob/devel/packages/spacebars/README.md`.

See also

- The *Inserting templates with Spacebars* and *Creating customized global helpers* recipes in this chapter

Using reactivity with HTML attributes

One of the hidden gems in Meteor is the extremely granular control you have over how HTML is rendered. Blaze was designed in such a way that you can modify, add, or remove HTML element attributes dynamically without affecting anything else in the DOM! This recipe will walk you through the use of dynamic, reactive data inside element attributes.

Getting ready

We will use the codebase from the *Creating custom components* recipe, the previous recipe in this chapter. Please follow that recipe first or download the corresponding codebase.

How to do it...

Proceed with the following steps to use reactivity with HTML attributes:

1. We first need to create a new `navcolor` template. Open `navbar.html` in your `[project root]/client` folder and add the following code:

```
<template name="navcolor">
    <div class="nav navbar-form navbar-right">
        <input type="checkbox" id='bgCheck'
        checked="{{useBG}}"/>
        <input type="text" id='bgColor'
        placeholder="#bgcolor" value="{{bgColor}}" />
    </div>
</template>
```

2. We now need to add our new template to our `navbar` content block so that we can input a new background color and turn it on and off. Open `templates.html` (also in the `client` folder) and modify the `firstTemplate` template, adding a template call just below the `{{> navbutton}}` call, similar to the following:

```
{{#navbar this}}
      {{> navbutton}}
      {{> navcolor}}
    {{/navbar}}
```

3. On the very next line, add a `style` attribute to the `<svg>` element so that we can change the background color, like so:

```
<svg height="800" width="100%" id="shapes"
style="background-color:{{bgColor}}">
```

4. In the preceding code, we have set up references to two helper functions: `bgColor` and `useBG`. We will now create these global template helpers. Open `templates.js` and append the following two functions:

```
Template.registerHelper('bgColor', function () {
    if (Session.equals('useBG', true)) {
        return Session.get('bgColor');
    } else return null;
});

Template.registerHelper('useBG', function(){
    return Session.get('useBG');
});
```

5. If we save these changes and open the web console in your browser, we can now programmatically change and toggle the background color. In the web console, execute the following two lines:

```
> Session.set('bgColor','lightgrey')
> Session.set('useBG',true)
```

These three things should have changed on your screen:

- The background should now be light grey in color
- The checkbox to the top-right of your `navbar` tag should be checked
- The word **lightgrey** should now appear in the top-right text box

Your screen should now look similar to the following screenshot:

6. All that's left is to hook up events to the checkbox's and textbox's `<input>` elements. Create a new file named `navbar.js` in your `[project_root]/client` folder and add the following `events` handler declarations:

```
Template.navcolor.events({
  'click #bgCheck': function (e, c) {
    Session.set('useBG', e.currentTarget.checked);
  },
  'change #bgColor, keyup #bgColor': function (e, c) {
    if (!Session.equals('bgColor',
    e.currentTarget.value)) {
      Session.set('bgColor', e.currentTarget.value);
    }
  }
});
```

7. Save your changes and your elements will now affect the background. Checking/unchecking the checkbox will toggle the background color, and changing the value in the textbox will dynamically change the displayed background.
8. We want to observe the effect this has on the DOM. Open the `developer` panel in your browser and locate the `<svg>` element. As you toggle the background on and off using the checkbox, observe the attribute list inside the top `<svg>` tag. You'll notice that the `style` attribute appears and disappears, depending on the state of the checkbox, similar to the following screenshot:

As you can see, the attribute doesn't just become empty. It is removed altogether. You will also notice that when this happens, neither the screen nor the elements view are refreshing. That's because the DOM is not being rebuilt. The attribute is being added and removed, according to the state of `{{bgColor}}`. Similarly, when you modify the value in the textbox, nothing else is affected (no screen refreshes, or lost states).

How it works...

With the introduction of the Blaze templating engine, Meteor stopped manipulating strings and started interacting with DOM trees / DOM elements directly. This change simplifies the development process in comparison to other templating libraries because DOM elements don't have to be re-rendered every time there is a change to the HTML.

In the case of the `style` attribute, the only argument inside the attribute is a reference to the `bgColor` helper function. If the value from a helper function or data context is `null/undefined`, and if that value is the only value in an attribute, Blaze automatically removes the attribute, rather than rendering it with no value in it.

So, in this case, when the `{{bgColor}}` helper function returned `null` (when `Session.useBG` is `false` or `undefined`), Blaze was removing the `style` attribute from the `<svg>` element.

See also

- The *Inserting templates with Spacebars* recipe in this chapter
- The *Using the Session object* recipe in *Chapter 4*, *Creating Models*

Using dynamic templates

Meteor has yet another templating trick up its sleeve! In addition to custom code blocks, reactive data contexts, and all the other various template niceties, Meteor also allows you to dynamically switch between rendered templates. This recipe will show you how to dynamically render templates, using the `Template.dynamic` helper.

Getting ready

We will use the codebase from the *Creating custom components* recipe found in this chapter. Please follow that recipe first or download the corresponding codebase.

How to do it...

We will add a dynamic template that will switch our shapes from circles to squares, when rendered. Proceed with the following steps to use dynamic templates:

1. First, we need to create our alternate template. Inside `templates.html`, located in your `[project root]/client` folder, add the following template, just below the declaration for the existing `one` template:

```
<template name="two">
    {{#each svgShapes}}
        <rect x="{{x}}" y="{{y}}" width="80" height="80"
stroke="{{border}}" stroke-width="3" fill="{{color}}" />
    {{/each}}
</template>
```

2. We will need a data context for this template. Fortunately, we can reuse the `Shapes` collection. Open `templates.js`, make a copy of the `svgShapes` function, and add it to a new `Template.two.helpers()` method call, as follows:

```
Template.two.helpers({
  svgShapes: function () {
    return Shapes.find();
  }
});
```

   Now, we will add logic to toggle between the `one` and `two` templates, based on the `Session.curTemplate` variable. At the bottom of `templates.js`, add the following `helpers()` method call:

```
Template.firstTemplate.helpers({
  curTemplate: function () {
    var curTempl = Session.get('curTemplate');
    if (!curTempl) {
      curTempl = 'one';
      Session.set('curTemplate', curTempl);
    }
    return curTempl;
  }
});
```

3. All that's left to do is to add the dynamic template declaration and create an easy way to toggle between the templates. Let's take care of the declaration first. Open `templates.html` and modify the double brackets inside the `<svg>` element as follows:

```
<svg height="800" width="100%" id="shapes">
    {{> Template.dynamic template = curTemplate}}
</svg>
```

4. Now, we will add some toggle buttons and hook up the events to change the value of the `Session.curTemplate` variable. Open `navbar.html` and add the following template at the bottom:

```
<template name="navshapes">
    <div class="nav navbar-right">
        <button id="btnCircles" class="btn btn-success navbar-
btn">one</button>
        <button id="btnSquares" class="btn btn-danger navbar-
btn">two</button>
    </div>
</template>
```

5. We need to include the `navshapes` template in our block helper. Open `templates.html` and add the following template inclusion just before the `{{/navbar}}` block tag:

```
{{#navbar}}
  ...
  {{> navshapes}}
{{/navbar}}
```

6. One last addition, to hook up the events, and we're ready to roll. Create a `navbar.js` file in your `[project root]/client` folder and add the following code:

```
Template.navshapes.events({
  'click button.btn': function(e,c){
    Session.set('curTemplate',e.currentTarget.textContent);
  }
});
```

7. Save all your changes, start your Meteor project if it isn't already running, and navigate to your project in a browser (usually `http://localhost:3000`). You will see two buttons at the top-right corner of the screen. Click on them, back and forth, to see the shapes on the screen toggle between circles and squares, similar to the following screenshot:

How it works...

`Template.dynamic` takes a `template` argument and renders the template with the appropriate name. Because we make the template argument dynamic and reactive, using the `curTemplate` helper, the rendered template will change as the value of `Session.curTemplate` changes. This ability to dynamically render templates is the basis for the popular Meteor package `iron:router`. It can be used in multiple situations where you would like to switch out templates, based on reactive data sources.

There's more...

Iron Router takes dynamic templates to an entirely new level. The documentation for Iron Router can be found at `https://github.com/iron-meteor/iron-router/blob/devel/Guide.md`.

See also

- The *Creating a multipage application with Iron Router* recipe in *Chapter 2*, *Customizing with Packages*
- The *Creating custom components* recipe in this chapter
- The *Using the Session object recipe* in *Chapter 4*, *Creating Models*
- The *Creating a complete app with Iron Router* recipe in *Chapter 12*, *Creating Useful Projects*

Animating DOM elements

Meteor has a very elegant way of rendering DOM elements in our web apps, but until now, this hasn't included animations. Fortunately for us, Meteor now supports animations, including animations when an element is first added to the DOM. This recipe will walk you through how to use standard CSS animation techniques inside Meteor.

Getting ready

We will use the codebase from the *Using dynamic templates* recipe found in this chapter. Please follow that recipe first or download the corresponding codebase.

How to do it...

We will animate the shapes from our previous recipe, demonstrating animation both before and after the elements are first created and rendered.

Because we used SVG elements instead of DOM elements, we need to modify jQuery to be able to use `.addClass` and `.removeClass`. Fortunately, there is a custom package available that will add SVG support and add Meteor UI event support at the same time.

1. In a terminal window, navigate to the root folder of your project and execute the following line:

   ```
   $ meteor add appmill:animation-hooks
   ```

This will install the `animation-hooks` package, which includes support for SVG in jQuery.

2. We will now add a very simple hover animation effect. Open or create a `style.css` file in your `[project root]/client/lib/styles` folder and add the following code:

```
circle,rect {
    opacity:1;
    transition:all 500ms 0ms ease-in;
}

circle:hover, rect:hover{
    opacity:0;
}
```

3. Save your changes, start your Meteor project if it isn't already started, and navigate to your project URL in a browser (usually `http://localhost:3000`). If you don't have shapes on your screen, click to add some and then hover over the shapes. As you hover, you will notice that the shapes will fade out slowly, and as you move the mouse away, the shapes will fade in slowly until they return to full opacity. Now we will use jQuery to move all the shapes on the screen at once. Inside `navbar.html`, add the following template at the bottom:

```
<template name="navoffset">
    <div class="nav navbar-right">
        <button id="btnNegative" class="btn btn-danger navbar-btn"
data-offset="-">red</button>
        <button id="btnPositive" class="btn btn-info navbar-btn"
data-offset="">blue</button>
    </div>
</template>
```

4. Inside `templates.html`, add a call to the new `navoffset` template, just before the `navshapes` declaration:

```
    {{#navbar this}}
     {{> navbutton}}
     {{> navoffset}}
     {{> navshapes}}
    {{/navbar}}
```

We now add the code for the `button` events by modifying `navbar.js`, adding the following function at the bottom:

```
Template.navoffset.events({
  'click button.btn': function(e,c){
    var shapes = ($('rect').length)? $('rect'):$('circle');
    if (shapes.length==0) return;
```

```
    var offset = e.currentTarget.dataset.offset;
    _.each(shapes,function(d,i){
      var randVal = Math.ceil(Math.random()*200)+'px';
      var randOffset = offset+randVal;
      var translate = ('translate('+randOffset+','+randOffs
et+')');
      $(d).css('transform',translate);
    });
  }
});
```

5. We also want to add a little bit of polish to our `nav` buttons, so add the following CSS to the `style.css` file, at the bottom:

```
div.nav {
    margin-left: 20px;
}
```

6. Save all these changes and your web page should now contain two new buttons, labeled **red** and **blue**. As you click on these buttons alternately, all the shapes on the screen will move their location randomly, back and forth diagonally across your screen.

7. We will now use the built-in animation hooks, called `_uihooks`, to animate shapes as they are added dynamically to the DOM. Open `templates.html` and make the following modifications to both the `one` and `two` templates:

```
{{#each svgShapes}}
  {{#Animate}}
    <circle class="shape animate" ...
  {{/Animate}}
{{/each}
...
{{#each svgShapes}}
  {{#Animate}}
    <rect class="shape animate" ...
  {{/Animate}}
{{/each}}
```

8. We have one final step and that is to add the initial *before* state to the shapes, via our `.animate` CSS declaration. Open `style.css` and add the following declaration at the bottom:

```
rect.animate, circle.animate {
    transform: translate(-300px,-100px);
    opacity:0;
}
```

9. Save your changes and test out the new animations functionality by either adding new shapes to the screen (clicking anywhere), or by changing the shapes from circles to squares, and back again (clicking on the **one** and **two** buttons). As you click, you will see the shape(s) fade in and move into place from the left.

How it works...

Because Blaze now renders objects according to the DOM, rather than rendering according to text-based HTML tags, it is aware of the styles and conditions of the created elements, for example, because we added the `opacity:0` style to our `:hover` shape elements in `style.css`, elements underneath the mouse will fade out and stay faded out, even when we click on / add new shapes. Using jQuery, we can programmatically add styles and transformations, as we did in `navbar.js` with the `.css()` functions. Blaze will respect these changes as well because programmatic changes to the CSS/DOM are remembered, even when new elements are added.

The `_uihooks` declaration contains three event handlers:

`insertElement`	This is fired before a DOM node is rendered so that we can manipulate its position, map where the node will go when animated, and then animate the node as it is rendered.
`removeElement`	This is fired before a DOM node is removed. We check to see whether there is an outgoing animation and animate the node if so. Otherwise, we just remove it.
`moveElement`	This is fired when a DOM node changes the indexed position in the DOM. We can add any animations before, and then during the index change, to create an animation effect.

For our sample, we only need to worry about inserting and removing elements. Each element has the CSS `animate` class assigned to it, by virtue of the template, which offsets the intended position by (-300,-100). The way we animate is by having the element appear on the screen in the offset position (with the `animate` class in effect), and then by removing the `animate` class, which triggers a transition/animation because the element moves to its intended position. Thanks to `_uihooks` and the `insertElement` event handler, we can programmatically remove `class` declarations to elements, right after they are initially rendered, causing them to animate. This makes the shapes seem to fade in from the top-left. Similarly, we can then add them back right before the elements are removed in order to perform an outgoing animation.

We declare which elements will be affected by wrapping the elements in the `{{#Animate}}`... `{{/Animate}}` block helpers, which causes the `insertElement` and `removeElement` handlers to fire on the wrapped elements.

There's more...

We expect some changes as Meteor continues to improve animation support, although the major functionality is already there. Meanwhile, you can walk through how `_uihooks` works by adding breakpoints to the client source, in the `packages/appmill:animations-hooks/animation-hooks.js` file, similar to the following screenshot:

```
(index)   strack:animation-hooks.js ×   deps.js   view.js
 9
10 Template['Animate'].rendered = function(){
11     var animationElements = this.findAll('.animate');
12
13     // HACK: initial animation rendered, as insertElement, doesn't seem to fire
14     _.each(animationElements, function(item){
15         var $item = $(item);
```

You can also get a (slightly) more thorough explanation at `https://github.com/strack/meteor-animation-hooks`.

See also

- The *Using reactivity with HTML attributes* recipe in this chapter

4

Creating Models

In this chapter, we will cover the following topics:

- Implementing a simple collection
- Using the Session object
- Sorting with MongoDB queries
- Filtering with MongoDB queries
- Creating upsert MongoDB queries
- Implementing a partial collection

Introduction

The Meteor data model is designed to be very easy to develop. Gone are the days of worrying about lengthy SQL statements, database drivers, and rigidly structured database tables. In its place is a straightforward, JSON-based document model that lets you focus on the functionality of your application. This chapter contains the most common recipes to interact with MongoDB and the reactive model context inside Meteor.

Implementing a simple collection

Collections are the medium of communication between the client and the server, with changes being pushed down to the client and requests being pushed up to the server. This recipe will show you how and where to declare a collection for use on both the client and the server.

Getting ready

First, you will need Meteor installed and have a project created. To create a standard project file structure, please see the *Setting up your project file structure* recipe in *Chapter 1*, *Optimizing Your Workflow*.

For this particular exercise, you will also need to have the `autopublish` and `insecure` packages installed (which are installed by default). If you need to add them, please see the *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*.

How to do it...

To implement a simple collection, proceed with the following steps:

1. Create a new file in your `both` folder, called `simple.js`.
2. Open the `simple.js` file in an editor, and declare a MongoDB comments collection by entering the following:

   ```
   Comments = new Mongo.Collection('comments');
   ```

3. Now, open the `main.js` file in your client's `scripts` folder (`[project root]/client/scripts/main.js`), add an action to the `click button` event handler, which will insert a record into the `Comments` collection:

   ```
   Template.hello.events({
     'click button': function () {
       // increment the counter when button is clicked
       Session.set('counter', Session.get('counter') + 1);
       Comments.insert({text:'This is comment #'
       + (Comments.find().count()+1)});
     }
   });
   ```

4. We need to add a comments helper as well. Locate the `Template.hello.helpers` method and add the following helper to the beginning:

   ```
   Template.hello.helpers({
     comments: function () {
       return Comments.find();
     },
     ...
      });
   ```

5. Save these changes.

6. Open your `main.html` page (`[project root]/client/main.html`), add a `{{#each..}}` template fragment to the `hello` template as specified in the following example, and save your changes:

```
<template name="hello">
  {{#each comments}}
  <div>{{text}}</div>
  {{/each}}
  ...
</template>
```

7. Now, open a terminal window, navigate to the `[root]` folder of your project, and start up Meteor:

```
$ cd [project root]
$ meteor
```

8. In a web browser, navigate to `http://localhost:3000/` and click on the button on the screen several times. The result should add comments with each click, similar to this:

How it works...

The declaration in `simple.js` is read by both the client and the server:

```
Comments = new Mongo.Collection('comments');
```

This instantiates the model and manifests as a collection called `Comments`.

The changes to `main.js` consist of an extra action in the `click` event:

```
Comments.insert({text:'This is comment #' +
(Comments.find().count()+1)});
```

This adds a comment object to the `Comments` collection on the client, which is quickly propagated to a minimongo on the client and then to MongoDB on the server. The UI is updated instantly because of the second change in `main.js`:

```
comments: function () {
    return Comments.find();
  }
```

The `comments` helper is a **reactive computation**, which means that it reruns itself every time there is a change in one of the **reactive contexts** (observed properties) it contains. In this example, `Comments.find()` is a reactive context, and therefore, whenever there is a change in the `Comments` collection, this `comments` helper will rerun.

Provided that the MongoDB collection (server side) ratifies the change, the UI will stay updated. If there was a conflict or some problem with the transaction, the server will send a corrective message, and Minimongo will be updated with the correct state. But, in this case, since we have no conflicts or latency, the change stuck, and comments are added after each click.

This template is reactive, which means when a change is found in the `Comments` collection, this function will be updated, and Blaze will re-render the `{{#each...}}` template block added to `main.html`:

```
{{#each comments}}
  <div>{{text}}</div>
  {{/each}}
```

See also

- The *Creating dynamic lists* and *Using reactivity with HTML attributes* recipes in *Chapter 3*, *Building Great User Interfaces*

Using the Session object

The **Session object** is a global client-side object, and as such, is part of the client model. Though it's not part of any collection, the Session object can be used in a reactive context, which means you can use it to make reactive methods rerun whenever it is changed. This recipe will show you how to use the Session object to update the elements on your page.

Getting ready

You will need to have Meteor installed and a project created. To create a standard project file structure, please see the *Setting up your project file structure* recipe in *Chapter 1*, *Optimizing Your Workflow*. A quick default project will work just as well for this recipe.

How to do it...

To use the Session object, proceed with the following steps:

1. Open the `main.js` file in your client's `scripts` folder (`[project root]/client/scripts/main.js`) and add a `greeting` helper to the beginning of `Template.hello.helpers`, as follows:

```
Template.hello.helpers({
  greeting: function() {
    return Session.get('greeting')||'Welcome to Chapter 4';
  },
  ...
```

2. Open `main.html` and add the greeting to your `hello` template, as follows:

```
<template name="hello">
  <h3>{{greeting}}</h3>
```

3. If Meteor is not already running, initiate it by navigating to your project root folder in a terminal window and run the `meteor` command:

```
$ cd [project root]
$ meteor
```

4. Now, open a browser, navigate to `http://localhost:3000/`, and open your web console.
5. In the web console, enter and execute the following command:

```
> Session.set('greeting','I just changed the Session Object')
```

The greeting on your screen should have changed from **Welcome to Chapter 4** to **I just changed the Session Object**, as shown in the following screenshot:

How it works...

Adding a call to `Session.get` inside the `greeting` template helper tells Meteor to look for a value placed in the greeting key of the `Session.keys` collection. As the collections are reactive contexts in Meteor, changes made in the `Session` collection (even when executed through the web console) are instantly reflected in the UI.

The `Session` object is literally a *Session* object. It lasts only until the next manual page refresh. If you want something more permanent, you will need to use offline storage or a MongoDB collection. Make sure to include this fact in your design/development plans.

There's more...

As of `0.9.1`, Meteor also provides single reactive variables. They can be used in place of the `Session` object, have a cleaner syntax, can be any type of object, and support checking for changes before a reactive invalidation is triggered.

To use reactive variables, add the `reactive-var` package to your Meteor project:

```
$ meteor add reactive-var
```

You can then declare and use a variable instead of the `Session` object:

```
greeting = ReactiveVar("Welcome to Chapter 4");

greeting: function () {
    return greeting.get();
}
```

This variable (declared properly) can be manipulated in the same way as the Session object, with cleaner syntax. The following command entered in the web debug console will change the display accordingly:

```
> greeting.set('I just changed the reactive variable')
```

The preceding command will result in something similar to the following screenshot:

See also

- The *Using reactivity with HTML attributes* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Creating and consuming a reactive* value and *Updating Blaze template without Mongo* recipes in *Chapter 6*, *Mastering Reactivity*

Sorting with MongoDB queries

There are many times when you will need to sort a collection. Most recent comments, alphabetized lists, and bank transactions by amount are all good examples. This recipe will show you how to sort MongoDB collections, using the options in your `find()` request.

Getting ready

We will be using the *Implementing a simple collection* recipe found in this chapter as a baseline. please follow that recipe in preparation for this activity.

How to do it...

To perform sorting with MongoDB queries, proceed with the following steps:

1. In your `main.html` file (located at `[project root]/client/main.html`), make the following modification to the `{{#each...}}` template fragment:

```
{{#each comments}}
  <div>{{text}}{{number}}</div>
  {{/each}}
  {{greeting}}
```

2. Next, modify the `Comments.insert` action inside the `click` event handler, found in `[project root]/client/script/main.js`:

```
Template.hello.events({
    'click button' : function () {
      // template data, if any, is available in 'this'
      if (typeof console !== 'undefined')
        console.log("You pressed the button");
        Comments.insert({text:'This is comment #',
        number:Random.hexString(3)});
    }
  });
```

3. Finally, inside `main.js`, modify the `find()` arguments to specify a `sort` comments by its `number` in descending order (`-1`):

```
Template.hello.helpers({
  ...
  comments: function () {
  return Comments.find({},{sort:{number:-1}});
  },
  ...
});
```

4. Save all the changes and run Meteor if necessary. Navigate to `http://localhost:3000/`, click on the button on the screen several times, and watch as the comments are sorted by a comment number:

How it works...

The crux of this recipe is found in the `find()` method arguments. We modified how the number was being stored by adding a random number so that if we do nothing else, the comments will appear out of order. But by adding `{sort:{number:-1}}` to the `comments` helper, we are giving results to Blaze that are sorted in descending order by the randomly generated `number` property. To sort in ascending order, use `1` as your parameter. To sort in descending order, use `-1`.

The rest of the changes are to support the sorting change. First, `main.js` was modified so that the `number` property could be assigned at random and added to the comment object when it is inserted into the `Comments` collection:

```
Comments.insert({text:'This is comment #',
number:Random.hexString(3)});
```

The last change is made to `main.html` to display the new `number` property in the UI:

```
<div>{{text}}{{number}}</div>
```

There's more...

You can perform complex sorts using multiple properties if the need arises. To do so, simply add the additional property key to the `sort` option, with either an ascending (`1`) or descending(`-1`) value, similar to the following:

```
return Collection.find({},{sort:{prop1:-1, prop2:1}});
```

See also

- The *Inserting templates with Spacebars* recipe in *Chapter 3*, *Building Great User Interfaces*

Filtering with MongoDB queries

Whether you're performing searches, organizing records, or narrowing down results, sooner or later, you'll want to filter the results of your collections. This recipe shows you how to limit the number of records in a collection, using MongoDB's `find` method options.

Getting ready

We will use the project created in the *Sorting with MongoDB queries* recipe, found in this chapter. Please complete that recipe, and use the files as a baseline for this recipe.

How to do it...

To filter MongoDB queries, proceed with the following steps:

1. Make the following change to the `comments` helper function in `[project root]/client/scripts/main.js`:

```
Template.hello.helpers({
  ...
  comments: function () {
    return Comments.find{number:/[2468]/},{sort:{number:-1}});
  },
  ...
});
```

2. Save `main.js` and start Meteor if necessary. Navigate to `http://localhost:3000/`; click the button on the screen several times and watch as only the comments that contain an even number are displayed. Your results after multiple clicks should look similar to the following screenshot:

3. Change the regular expression in `main.js` to only show comments that contain an odd number by making the following change to the query:

```
return Comments.find({number:/[13579]/},
{sort:{number:-1}});
```

4. Save your changes and observe the change to your UI, which should look similar to the following screenshot:

How it works...

By adding a selector to the `find()` statement in the query, the collection is paired down to match whatever is specified in the selector. In this case, the selector was a regular expression, which returned results only if the `number` property contained at least one even number, and subsequently, if the `number` property contained at least one odd number.

Notice that this didn't change the `{sort:{number:-1}}` predicate in any way and that it works exactly the same, regardless of how we changed the query.

There is a healthy range of selectors that you can use to limit/filter your collection. For a complete list, check out the MongoDB query operators list found at `http://docs.mongodb.org/manual/reference/operator/query/`.

There's more...

Moving the filter to the server side provides security and performance advantages as the non-matching results are never sent down the wire to the client in the first place. In fact, `find()` is most commonly found on the server, using the Meteor publish and subscribe functionality. For an example of `find()` being used on the server, please see the *Implementing a partial collection* recipe found in this chapter.

See also

- The *Implementing a partial collection* and *Sorting with MongoDB queries* recipes in this chapter

Creating upsert MongoDB queries

As you are developing applications, on occasion, you will run into a situation where you need to either update an existing record, or insert a record if it doesn't exist already. This is done traditionally using conditional operators such as `if`.

Through MongoDB, Meteor removes the burden of checking by allowing you to upsert (update + insert) records using a simple syntax. This recipe will show you how.

Getting ready

We will again use the project created in the *Sorting with MongoDB queries* recipe, found in this chapter. Please create a fresh copy of this recipe, and use the files as a baseline for this recipe.

We will also be using the official `momentjs` package. To add the `momentjs` package, navigate to your project's root folder in a terminal window. Once there, execute the following command:

```
$ meteor add momentjs:moment
```

This will add the `momentjs` package, and you're now ready to complete this recipe.

How to do it...

We first need to increase the likelihood of duplicate records and identify the time at which each comment was inserted or updated. Proceed with the following steps to create upsert MongoDB queries:

1. Open the `[project root]/client/scripts/main.js` file and modify `Template.hello.events` as follows:

```
  'click button': function () {
  ...
  Session.set('counter', Session.get('counter') + 1);
  var newC = {
    text:'This is comment #',
    number:Random.hexString(1),
    time: moment().format('ll, hh:mm:ss')
  };
  Meteor.call('commentUpsert',newC);
}
```

2. We will now create a server method for the `upsert` function. By default, Meteor only allows modifications on the client by `_id`, and since an `_id` value may not exist, we need to create a server method for our `upsert` function. Open `[project root]/server/server.js` and add the following method to the bottom of the file:

```
Meteor.methods({
  commentUpsert: function(newC){
    Comments.upsert({number:newC.number},{$set:newC});
  }
});
```

3. Finally, we will modify `[project root]/client/main.html` to display the timestamp on each comment in the `comments` collection:

```
{{#each comments}}
  <div>{{text}}{{number}} at: {{time}}</div>
  {{/each}}
```

4. Save your changes, start the Meteor if it is not already running, and navigate to `http://localhost:3000/`.
5. Continuously, click on the button on the screen, and observe how, instead of new records being added on each and every click, if a record already exists, the record's timestamp is updated.

How it works...

In `server.js`, we created a method by declaring the `commentUpsert` method inside `Meteor.methods`.

This method receives a new comment object (`newC`) as an argument and calls `upsert` on the `Comments` collection. The first argument (the selector) tells MongoDB to look for any entries that match the `number` property found in the `newC` object. The second argument (the modifier) tells MongoDB which fields to insert/update on the `upsert` object.

If a match is found, the fields are updated. If no match is found, a new record is inserted, with the `newC` object providing the values.

In `main.html`, we simply added the new `time` property to the display:

```
<div>{{text}}{{number}} at: {{time}}</div>
```

In `main.js`, we first remove the `Comments.insert()` statement. We then create the `newC` object, populating it with a random number, some text, and a timestamp using `moment().format()` to convert the timestamp into a readable format. Finally, we make a call to the `commentUpsert` server-side method, with a `Meteor.call` statement.

The result confirms that our `upsert` function is working properly, as each new comment (with a new `number` property) is added to our list, and each existing comment (with an already existing `number` property) has its timestamp updated.

There's more...

Meteor simplifies things a bit by adding the actual `upsert` function, rather than setting `{upsert:true}` inside a traditional MongoDB `update` query. Both `update` or `upsert` can be used in Meteor, and it's really up to you to choose.

> Extensive documentation exists on updating and upserting records in a MongoDB collection. You can find more details at `http://docs.mongodb.org/manual/reference/method/db.collection.update/`.

See also

- The *Building custom server methods* recipe in *Chapter 11*, *Leveraging Advanced Features*

Implementing a partial collection

Partial collections are collections sent down the wire from the server that contain only part of the information available on each record. This is useful for hiding properties or fields, and also for paring down the size of records containing a lot of information. In other words, partial collections can help with security and performance. This recipe will show you how to implement a partial collection on the server.

Getting ready

Let's use the *Creating upsert MongoDB queries* recipe found in this chapter as a baseline for this recipe. Create a fresh copy of this recipe and then proceed to the other preparations.

Let's spruce things up just a bit as well while we're at it. In a terminal window, navigate to the root project folder and execute the following command:

```
$ meteor add twbs:bootstrap
```

This will add the `bootstrap` CSS framework. We now want to take advantage of `bootstrap`, so open up your `main.html` file (found in `[project root]/client/`) and make the following changes to the `hello` template:

```
<template name="hello">
  ...
  <div class="btn-group-vertical">
    {{#each comments}}
    <div class="btn btn-default">{{text}}{{number}}
      <span class="label label-warning">
        {{time}}
      </span>
    </div>
    {{/each}}
  </div>
  <p><button>Click Me</button></p>
  ...
</template>
```

These changes will use some of the default `bootstrap` component styles to make our display look similar to the following screenshot:

We now need to remove the `autopublish` default Meteor package. In the terminal window, execute the following command:

```
$ meteor remove autopublish
```

This will temporarily *break* your application; in that no records from the `Comments` collection are being sent down the wire to the client, so no records will be displayed in the UI. Don't worry, we'll fix this!

You are now ready to move on to the recipe.

How to do it...

First, since we removed `autopublish`, we need to prepare the client to subscribe to the `comments` data stream. To implement a partial collection, proceed with the following steps:

1. Open `[project root]/client/scripts/main.js` and add the following code to the very top of the document:

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

2. While we have `main.js` open, let's also change what we sort, by making the following change to the `comments` helper function and save your changes:

   ```
   comments: function () {
     return Comments.find({},{sort:{time:-1}});
   },
   ```

3. Next, we need to have the server publish the `Comments` data collection. We'll simultaneously remove the text field from the response stream, implementing a **partial collection**. Open `[project root]/server/server.js`, add the following code, and save your changes:

   ```
   Meteor.publish ('comments',function(){
     return Comments.find({} , {fields:{text:false}});
   });
   ```

We can now immediately see that the text field is no longer being displayed in the UI, as shown in the following screenshot:

4. Let's clean up our UI a little bit now and make it obvious that the records are being reordered. Open `main.html` again, make the following changes, and save the file:

```
<div class="btn btn-default">{{time}}
  <span class="label label-warning">
    {{number}}
  </span>
</div>
```

Now, when you click on the button to update the `time` property, the timestamp will be updated, and the ordering of the records will change with each click (click several times to really see the effect in action):

How it works...

The crux of this recipe is found inside `server.js` where we added the `fields` field specifier to the MongoDB query (in the `Meteor.publish(...)` statement):

```
Meteor.publish ('comments',function(){
  return Comments.find({} , {fields:{text:false}});
})
```

This field specifier tells the MongoDB query to exclude the text field/property from the results. Once this change has been made, `main.html` still tries to call `{{text}}` in the template, but since it's null, it doesn't appear. We then removed the call to display `{{text}}` altogether, because it wasn't needed.

There's more...

The `fields` field specifier can be made **exclusive**, as we have seen in the preceding section, by only listing the fields we don't want to see. To do this, simply use the `:false` (or `:0`) parameter for each field you don't want to see, and the rest of the fields will be included by default. For example:

```
record.find({} , {fields:{f1:false , f2:false}});
```

The field specifier can also be made **inclusive**, which means that only the fields specified will be included by using the `:true` (or `:1`) parameter. This would look similar to the following code:

```
record.find({} , {fields:{f3:true , f4:true}});
```

To read more about the `fields` field specifier, check out the official Meteor documentation at `http://docs.meteor.com/#/full/fieldspecifiers`.

See also

- The *Basic safety – turning off autopublish* recipe in *Chapter 9*, *Securing Your Application*

5
Implementing DDP

In this chapter, we will cover the following topics:

- Reading the DDP stream
- Using client-only collections
- Implementing a multiserver DDP
- Integrating DDP with other technologies

Introduction

Fundamental to Meteor is the concept of real-time applications, with the client and server interacting with each other using the principle of **data on the wire**. The folks at Meteor have developed a protocol for this real-time client-server communication, called the **Distributed Data Protocol** (**DDP**). As with everything else in Meteor, DDP is a discreet (albeit core) library, which you can use separately. The recipes in this chapter will show you how DDP works and give you some ideas about what you can do with it outside of its default use in Meteor.

Reading the DDP stream

To truly understand what is going on with DDP (and for debugging purposes) it's useful to be able to read the DDP stream. Seeing DDP in action can not only help with the debugging process, but can also help you to better understand how flexible and useful this protocol is. This recipe will show you how to use Arunoda Susiripala's most excellent DDP Analyzer.

Getting ready

You will need Meteor, Node, and npm installed. For installation instructions, visit the following URLs:

- Node—`http://nodejs.org`
- npm—`https://www.npmjs.com/`

You will also need a project. You can use pretty much any project you like. Because it offers a decent, controllable amount of client-server communication (in other words, DDP streams), we will use the `todos` example application. Note that you can also use your own Meteor project if you like. We are only using this as an example because it's convenient and because we can control the flow of data.

To install the `todos` example, open a terminal window, navigate to a directory of your choice, and enter the following command:

```
$ meteor create --example todos
```

Go ahead and start your Meteor application by entering the following commands:

```
$ cd todos
$ meteor
```

You are now ready to install and run the DDP Analyzer on the `todos` app.

How to do it...

1. First, we need to install the DDP Analyzer. In a terminal window, enter and execute the following command:

   ```
   $ npm install -g ddp-analyzer
   ```

   Note that you may need to use `sudo` when using the `-g` operator to make the Analyzer available from anywhere and to make sure it has proper permissions. After a brief installation process, the Analyzer will be installed and ready to be used.

2. Start the Analyzer proxy by entering the following command:

   ```
   $ ddp-analyzer-proxy
   ```

   You will get a message similar to the following:

   ```
   DDP Proxy Started on port: 3030
   ==============================
   Export following env. variables and start your meteor app
     export DDP_DEFAULT_CONNECTION_URL=http://localhost:3030
     meteor
   ```

3. We now need to restart our Meteor instance with the environmental variable mentioned in the preceding step. In your terminal window, stop the `meteor` if it is still running and enter the following commands:

```
$ export DDP_DEFAULT_CONNECTION_URL=http://localhost:3030
$ meteor
```

4. Once the `meteor` starts the backup, navigate to `http://localhost:3000` in a browser and the DDP Analyzer will be ready for action.
5. Perform several tasks, such as adding new `todo` items, completing/editing existing items, and so on.
6. Take a look at the DDP stream in the terminal window where you started the DDP Analyzer. It will look similar to the following screenshot:

```
1  IN   2   {"msg":"added","collection":"lists","id":"5ZvEgXMZM8TA5hw6W","fields
":{"name":"List A","incompleteCount":0}}
1  IN   0   {"msg":"updated","methods":["5"]}
1  OUT  86  {"msg":"sub","id":"xSw7uXy4SwdkCDXS9","name":"todos","params":["5Zv
EgXMZM8TA5hw6W"]}
1  OUT  1   {"msg":"unsub","id":"P4hi9sw6NQoborLaC"}
1  IN   9   {"msg":"ready","subs":["xSw7uXy4SwdkCDXS9"]}
1  IN   0   {"msg":"removed","collection":"todos","id":"kSz9oNBos5izsM6AL"}
1  IN   0   {"msg":"removed","collection":"todos","id":"EYYr8fZvJWLBoXuMk"}
1  IN   0   {"msg":"removed","collection":"todos","id":"FTrsdR3aCzc2QpqQm"}
1  IN   0   {"msg":"removed","collection":"todos","id":"5rhu6kczXSsRXidfj"}
1  IN   1   {"msg":"removed","collection":"todos","id":"bNbaSxF7rsJXhDPDC"}
1  IN   0   {"msg":"removed","collection":"todos","id":"TkNAs64k4hKwbizH9"}
1  IN   0   {"msg":"removed","collection":"todos","id":"XTMjtXX9pqeNsWQKr"}
1  IN   0   {"msg":"nosub","id":"P4hi9sw6NQoborLaC"}
1  IN   501  {"msg":"ping"}
1  OUT  7   {"msg":"pong"}
1  OUT  11698  {"msg":"method","method":"/lists/update","params":[{"_id":"5ZvEg
XMZM8TA5hw6W"},{"$set":{"name":"Stuff for Chapter 5"}},{}],"id":"6"}
1  IN   4   {"msg":"result","id":"6","result":1}
1  IN   2   {"msg":"changed","collection":"lists","id":"5ZvEgXMZM8TA5hw6W","fiel
ds":{"name":"Stuff for Chapter 5"}}
1  IN   0   {"msg":"updated","methods":["6"]}
1  OUT  6140  {"msg":"method","method":"/todos/insert","params":[{"listId":"5Zv
```

You will be able to see all the messages intended for the clients. The numbers in yellow tell you which client it was sent to/from. Messages preceded by **IN** indicate that the message was sent from the server to the client. If the message is preceded by **OUT**, this indicates the message and outbound request from the client to the server. The messages are in plain text following the color-coded texts.

With the DDP Analyzer, you can really get under the hood, seeing exactly how DDP works. You'll quickly realize how flexible and simple the protocol is, and you will gain a better understanding of what you can do with it inside your Meteor (or other) applications.

How it works...

DDP, oversimplified, is a series of messages between two end points. The messages show updates, additions, and deletions, and are transmitted over websockets. This means the messages can be sent in real time, rather than waiting for the traditional client request polling. This cuts down on traffic (chatter) and ensures that the messages are delivered as quickly as possible.

The DDP Analyzer uses the DDP protocol and listens in on the streaming conversation between your browser and the Meteor application. When you changed the port that DDP executes through using the `DDP_DEFAULT_CONNECTION_URL=` command, you rerouted your app through port `3030`. The DDP Analyzer listened on this port, proxied the connection to port `3000`, and logged all the traffic approaching it in the terminal window.

There's more...

Once you have finished, be sure to either open a new terminal window or change your environment variable back to port `3000` using the following command:

```
$ export DDP_DEFAULT_CONNECTION_URL=http://localhost:3000
```

If you don't, your application will stop working because DDP is still running through port `3030`.

See also

- The *Using the web console* recipe in *Chapter 1*, *Optimizing Your Workflow*

Using client-only collections

To truly understand how DDP works, it's helpful to try *rolling your own* as far as communicating with data over the wire is concerned. The best way to do this (and a quite helpful recipe to boot!) is to let the client continue to take advantage of the built-in Mongo collections and programmatically send updates from the server based on some simple logic. This recipe will walk you through creating a client-only collection and manipulating this collection with calls on the server side.

Getting ready

We are going to need a sample project to use as a baseline. For this baseline, we will quickly create a bookmark application. No bells and whistles, just a simple list of bookmark titles and URLs.

To do this, we first need our default template scaffolding. Please create a new project called `bookmarks` using the *Setting up your project file structure* recipe in *Chapter 1*, *Optimizing Your Workflow*, as your starting file structure.

We need to add and remove some Meteor packages. Start by removing the `autopublish` package. Open a terminal window, navigate to the root folder of your project, and execute the following command:

```
$ meteor remove autopublish
```

Next, let's add the `bootstrap` package. Execute the following command in your terminal window:

```
$ meteor add twbs:bootstrap
```

We will now add the HTML and JavaScript needed to create our `bookmarks` program.

Replace the contents of your `[project root]/client/main.html` file with the following:

```
<head>
    <title>Chapter 05</title>
</head>
<body>
    {{> urls}}
</body>
<template name="urls">
    <div class="url-container">
      {{#each bookmarks}}
      <div class="bookmark panel {{selected}}">
        <div class="panel-heading">
          <button type="button" class="close"><span
          class="glyphicon glyphicon-remove"></span>
          </button>
          {{#if editing}}
            <h3 class="panel-title"><input type="text"
            value="{{title}}"></h3>
          {{else}}
          <h3 class="panel-title"> {{title}}</h3>
            {{/if}}
        </div>
        <div class="panel-body">
          {{#if src}}
            <a href="{{src}}">{{src}}</a>
          {{else}}
            <div class="input-group">
              <input type="text" class="form-control"
              placeholder="enter URL here" />
```

```
            </div>
          {{/if}}
        </div>
      </div>
      {{/each}}
      <input type="button" class="btn btn-primary" id="btnNewBM"
      value="add" />
    </div>
</template>
```

Next, in your `[project root]/client/scripts` folder, add the following to `main.js`:

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

Then, create a `templateHelpers.js` file in the same `scripts` folder and add the following `helpers` function:

```
Template.urls.helpers({
    bookmarks: function () {
      return URLs.find();
    },
    title: function () {
      return this.title || this.src;
    },
    editing: function () {
      return Session.equals('selMark', this._id) &&
      Session.equals('editMark', this._id);
    },
    selected: function () {
      return Session.equals('selMark', this._id) ? "panel-warning"
: "panel-success";
    },
    src: function () {
        return this.src || false;
    }
});
```

In the same `templateHelpers.js` file, add the following `events` function:

```
Template.urls.events({
    'click .bookmark': function (e) {
        Session.set('selMark', this._id);
    },
    'dblclick .bookmark': function (e) {
        Session.set('editMark', this._id);
    },
    'blur .panel-title > input': function (e) {
```

```
        if (e.currentTarget.value !== this.title) {
            Meteor.call('updateTitle', this._id, e.currentTarget.
value, function (err, succ) {
                console.log(succ);
            });
        }
        Session.set('editMark', null);
    },
    'keypress .panel-title > input': function (e) {
        if (e.keyCode == 13 && e.currentTarget.value !== this.title) {
            Meteor.call('updateTitle', this._id, e.currentTarget.
value, function (err, succ) {
                console.log(succ);
                Session.set('editMark', null);
            });
        }
    },
    'click #btnNewBM': function (e) {
        URLs.insert({
            title: 'new bookmark'
        });
    },
    'blur .input-group > input': function (e) {
        if (e.currentTarget.value !== this.src) {
            Meteor.call('updateSRC', this._id, e.currentTarget.value,
function (err, succ) {
                console.log(succ);
            });
        }
    },
    'keypress .input-group > input': function (e) {
        if (e.keyCode == 13 && e.currentTarget.value !== this.src) {
            Meteor.call('updateSRC', this._id, e.currentTarget.value,
function (err, succ) {
                console.log(succ);
            });
        }
    },
    'click .close': function (e) {
        Meteor.call('removeBM', this._id, function (err, succ) {

        });
    }
});
```

Now, we need a bit of styling. Open the `style.css` file found in your `[project root]/libs/styles` folder and replace the existing styles with the following:

```
/* CSS declarations go here */
body {
    font-size:1.5rem;
}

.url-container{
    background-color: rgb(255, 255, 255);
    border-color: rgb(221, 221, 221);
    border-width: 1px;
    border-radius: 4px 4px 0 0;
    -webkit-box-shadow: none;
    box-shadow: none;
    position: relative;
    padding: 45px 15px 15px;
    margin: 20px;
    margin-left:auto;
    margin-right:auto;
    max-width: 90%;
    border-style:solid;
}

.input-group {
    width:90%;
}
```

We need to declare our URLs collection, so create a `collections.js` file in your `[project root]/both` folder and add the following `Mongo.Collection` declaration and `String.prototype` function:

```
URLs = new Mongo.Collection("urls");

if (typeof String.prototype.startsWith != 'function') {
    String.prototype.startsWith = function (str){
        return this.slice(0, str.length) == str;
    };
}
```

Finally, we add some server logic. Open `[project root]/server/server.js` and replace any existing code with the following:

```
Meteor.publish('urls',function(){
   return URLs.find();
});

Meteor.methods({
    updateTitle: function(id,title){
        var bmark = URLs.findOne(id);
        if (!bmark) return;
        if (title&& title!==bmark.title){
            URLs.update(id,{$set:{title:title}});

        return "updated";
        }
        return "same title";
    },
    updateSRC: function(id,src){
        var bmark = URLs.findOne(id);
        if (!bmark) return;
        if (src&& src!==bmark.src){

            //Adding the http if it doesn't already have it...
            src = src.startsWith('http')? src: 'http://'+src;
            URLs.update(id,{$set:{src:src}});

            return "updated";
        }
        return "same src";
    },
    removeBM: function(id){
        URLs.remove(id);
        return "removed";
    }
});
```

Save all your changes, start up your `meteor` instance, and navigate to your project in a browser (usually `http://localhost:3000`). If everything went correctly, you will now have a working `bookmarks` app, where you can add URLs and (by double-clicking) add/change the titles for the bookmarks. The following screenshot is a sample of how the browser will look when it is complete:

How to do it...

We are going to add a counter at the bottom of our `bookmarks` page. This counter will tell us how many secure and non-secure links we have (determined by whether the link starts with `https`).

1. First, we will add the client-only collection called `bmcounts` and subscribe to it. If we declare the collection anywhere in the `clients` folder, the server will not be able see it. Hence, it will be a client-only collection. Open the `main.js` file in your client `scripts` folder (`[project root]/client/scripts/main.js`) and add the following two lines around the existing `Meteor.subscribe('urls')` command, as follows:

```
BMCounts = new Mongo.Collection("bmcounts");
Meteor.subscribe('urls');
Meteor.subscribe('bmcounts', new Mongo.ObjectID());
```

2. While we are in the `scripts` folder, let's open the `templateHelpers.js` file and add the reactive `Template.helpers` function which we'll need to display the counts. Add the following function to the bottom of the file:

```
Template.linkcount.helpers({
    BMCount: function(){
        return BMCounts.findOne();
    }
});
```

3. Now let's add the HTML template. Open `main.html`, found in your `client` folder, and add the following code around the `<body>` tag:

```
<body>
    {{> urls}}
    {{> linkcount}}
</body>
<template name="linkcount">
    {{#with BMCount}}
    <div id="counts">
        <button class="btn btn-lg btn-info">
            <span class="glyphicon glyphicon-link"></span>
            {{unsecureCount}} /
            <span class="glyphicon glyphicon-lock"></span>
            {{secureCount}}
        </button>
    </div>
    {{/with}}
</template>
```

4. We'll pretty up our counter a bit with some CSS as well. Open your `style.css` file, found in your `lib/styles` director, and add the following CSS declaration:

```
#counts{
    position:relative;
    float:right;
    margin-right:50px;
}
```

5. We have some real work to do now on the server side. We need to count and sort the bookmarks as they are added, removed, or changed. Open your `server.js` file, found in your `server` folder, and add the following `Meteor.publish` function:

```
Meteor.publish('bmcounts', function (id) {
  var self = this;
  var count = 0;
  var secCount = 0;
  var initializing = true;
```

```
  var handle = URLs.find().observeChanges({
    //TODO: Added
    //TODO: Removed
    //TODO: Changed
  });
  initializing = false;
  self.added("bmcounts", id, {
    unsecureCount: count,
    secureCount: secCount
  });
  self.ready();
  self.onStop(function () {
    handle.stop();
  });
});
```

6. Inside the preceding code block, we have three `TODO` comments. We first need to create the `added` listener. Directly under the `//TODO: Added` comment, add the following code:

```
//TODO: Added
added: function (idx, doc) {
  if (doc.src && doc.src.toLowerCase().startsWith('https')) {
    secCount++;
    if (!initializing)
      self.changed("bmcounts", id, {
        secureCount: secCount
      });
  } else {
    count++;
    if (!initializing)
      self.changed("bmcounts", id, {
        unsecureCount: count
      });
  }
},
```

7. Next is the `removed` listener. Directly under the `//TODO: Removed` comment, add the following code:

```
//TODO: Removed
removed: function (idx, doc) {
    //really inefficient...
    var bms = URLs.find().fetch();
    secCount = _.filter(bms,function(bm){
      return bm.src && bm.src.toLowerCase().startsWith('https');
```

```
    }).length;
    count = bms.length - secCount;
    self.changed("bmcounts", id, {
      unsecureCount: count,
      secureCount: secCount
    });

  },
```

8. Finally, the `changed` listener. Add the following code directly under the `//TODO: Changed` comment:

```
//TODO: Changed
changed: function (idx, doc) {
  if (doc.src && doc.src.toLowerCase().startsWith('https')) {
    secCount++;
    count--;
    self.changed("bmcounts", id, {
      unsecureCount: count,
      secureCount: secCount
    });
  }
}
```

9. Save all your changes, start your `meteor` instance if it isn't already started, and navigate to your project in a browser (usually `http://localhost:3000`). You should see a small information button at the bottom-right corner of your screen with a count of secure and non-secure links. Add some new links, delete others, and watch the count change instantly in your browser, similar to the following screenshot:

How it works...

All of the client-side code we just added is exactly the same as it would be for a regular Mongo collection. As far as the client is concerned, everything is normal.

On the server, however, we'd need to manually update the DDP stream with the `.added` and `.changed` messages, based on the changes made to the `URLs` collection.

To accomplish this, we start by using the function passed as an argument in the `Meteor.publish()` function arguments:

```
Meteor.publish('bmcounts', function (id) {
...
});
```

This function is rerun every time a client subscribes to the DDP stream. Inside the function, we create a `handle` using the `Mongo.Cursor.observeChanges()` function:

```
var handle = URLs.find().observeChanges({
  ...
});
```

Inside the `observeChanges` function, if a document is added (except when initializing), we check whether the `src` property begins with `https` to see whether we can increment the `secCount` (secure count) variable; otherwise, we'll just increment the normal `count` variable.

If a document is removed, we have to do a bit of trickery, but we will recalculate the `secCount` and `count` variables.

If a document's `src` variable is changed, we will perform the same check as we did in `added` and move the increment from `count` to `secCount`.

These three situations will happen as the `URLs` collection changes in real time. Therefore, after each change, we will call the `self.changed()` function, which sends a message to the client via DDP, mimicking the automatic change messages that would be sent if we were modifying the `bmcounts` collection normally.

You will notice that the `bmcounts` collection is never used on the server. Instead, we send messages to the client through the `"bmcounts"` subscription channel, as in the following example:

```
self.changed("bmcounts", id, {
  ...
});
```

This is an important distinction. The client does not subscribe to the actual MondoDB collection. Instead, it subscribes to a `DDP` messaging channel. In this case, we're letting the client do the heavy lifting for us by sending the `"bmcounts"` subscription messages into the `bmcount` client-only collection. In essence, the server is sending data changes over the wire, and the client, being none the wiser, is treating the messages as actual data changes.

The rest of the method is cleanup for when a client unsubscribes, using `handle.stop()` to complete the initialization. It includes making a `self.added()` call, which does the same thing as a `self.changed()` call (mimics a data change and sends it over the wire to the subscribed clients). The final step in initialization is to call `self.ready()`, which lets the subscribed client know that all the initial data changes have been sent.

See also

- The *Implementing a partial collection* recipe in *Chapter 4*, *Creating Models*
- The *Basic safety – turning off autopublish* recipe in *Chapter 9*, *Securing Your Application*

Implementing multiserver DDP

The fun doesn't stop with a single client and server DDP connection. Oh no! Indeed, Meteor has invited everybody to the DDP party, and in this recipe, we're going to see how two Meteor servers can speak to each other using DDP.

Getting ready

We will be using the code base from the *Using client-only collections* recipe, found in this chapter. Please create an instance of this project and start your `meteor` server.

For this project, we will change the color of the bookmark titles, based on whether the links are secure or not; so we need to make one change to our first project to make the color dependent on the `secType` property (which we will create in this recipe). Open the `templateHelpers.js` file, found in the `[project root]/client/scripts` folder of the first project. Locate the `Template.urls.helpers` section and make the following change to the `selected` function:

```
Template.urls.helpers({
    …
    selected: function () {
        return Session.equals('selMark', this._id) ? "panel-warning" :
this.secType;
    },
    …
```

As we will be implementing another Meteor server instance, we will also need to create a new Meteor project. In a terminal window, navigate to a location outside your root project folder, and enter the following commands in the terminal window:

```
$ meteor create typecheck
$ cd typecheck
$ meteor remove autopublish
$ meteor --port 3030
```

This will properly initialize our second server instance and start it up on port `3030`.

We're now ready to get our second Meteor project talking to our first project!

How to do it...

We're going to change the color of the bookmark titles using a new bookmark property called `secType`. Our new (second) project will check bookmarks as they come across the DDP wire and add a `secType` property if needed. Let's get started.

1. First, we will want to make use of the `String.startsWith()` function we're familiar with from the *Using client-only collections* recipe. Add the following prototype modifier to the bottom of the `typecheck.js` file, found in your second project's root folder:

   ```
   if (typeof String.prototype.startsWith != 'function') {
       String.prototype.startsWith = function (str){
           return this.slice(0, str.length) == str;
       };
   }
   ```

2. Now, we want to create a server-only file so that we can concentrate on the task at hand. Create a `server` folder in your project root and then create and open a file named `DDPChecker.js` for editing.
3. In the `DDPChecker.js` file, add the following code to connect and subscribe to the `urls` collection from our first project:

   ```
   conn = DDP.connect("http://localhost:3000");
   URLs = new Mongo.Collection("urls",conn);
   conn.subscribe("urls");
   ```

4. Now, we create a function that checks a record to see whether the `secType` variable has been set or not. If it hasn't and if there's an `src` property to check, it will update the record with the appropriate `secType` variable. Add the following function to the bottom of `DDPChecker.js`:

   ```
   function checkSecType(idx,doc){
       if (!doc.src || doc.secType)
          return;
   ```

```
    if (doc.src.toLowerCase().startsWith('https'))
        doc.secType = 'panel-success';
    else
        doc.secType = 'panel-danger';
    URLs.update(idx,doc);
}
```

5. The last piece of the puzzle is to add a call to the `checkSecType` function we just created, whenever a record is added or changed. We do this by calling the `observeChanges` function and the `URLs.find()` cursor. Add the following code to the bottom of `DDPChecker.js`:

```
URLs.find().observeChanges({
    added:checkSecType,
    changed:checkSecType
});
```

6. Open a browser and navigate to your project (usually `http://localhost:3000`). The colors of each bookmark should correspond to the security types of the bookmark links, similar to the following screenshot:

How it works...

Inside our second project, we are declaring the `URLs` collection and subscribing to changes just like we would in a normal `client` app. The difference here is that we're establishing a connection to an existing Meteor server, using the `DDP.Connect()` function:

```
conn = DDP.connect("http://localhost:3000");
URLs = new Mongo.Collection("urls",conn);
```

We passed the handle to this connection as an argument, called `conn`, when we declared our `URLs` collection. This tells Meteor that we're interested in the collection (more accurately, the DDP messages) coming from our first server.

To avoid redundant code, we created the `checkSecType` function, which listens to the changes made to the `URLs` collection and sets the `secType` property to correspond with the security type of the `src` property. We then use the `checkSecType` function as the event handler for both `added` and `changed` events coming through the DDP wire via `observeChanges`.

There's more...

Using DDP as a client isn't unique to Meteor. You can use DDP with nearly any technology. You'll see an example in the *Integrating DDP with other technologies* recipe, found later in this chapter.

See also

- The *Implementing a partial collection* recipe in *Chapter 4*, *Creating Models*
- The *Integrating DDP with other technologies* recipe in this chapter

Integrating DDP with other technologies

The party never stops with DDP! DDP libraries are available for most major programming languages and platforms, and you can very quickly get your client, server, or even hardware devices talking to your Meteor server without much effort. This recipe will show you how to connect to a Meteor server using DDP from a `Node.js` program.

Getting ready

We'll want to use the code base from the *Using client-only collections* recipe, found in this chapter. Please create an instance of that project and start your `meteor` server.

You will need Node and npm installed to complete this recipe. They are installed with Meteor by default. However, if you need instructions on how to install Node.js and npm, they can be found on the following websites:

- Node—`http://nodejs.org`
- NPM—`https://www.npmjs.com/`

We will also need to make sure that the `ddp` npm module is installed. Open a terminal window and execute the following command:

```
$ npm -g install ddp
```

When installing `npm` modules globally, you may need to use the `sudo` command, similar to the following command:

```
$ sudo npm -g install ...
```

How to do it...

We will create a Node service that will check the `src` properties for records in our `URLs` collection to make sure they're secure (HTTPS). If they aren't secure, the Node service will call our Meteor `updateSRC` method and change the `src` properties to be secure. To integrate DDP with other technologies, perform the following steps:

1. We first create the project folder. In a terminal window, create a `nodecheck` folder and `cd` to this directory, and add the `ddp npm` module by executing the following commands:

   ```
   $ mkdir nodecheck
   $ cd nodecheck
   $ npm install ddp
   ```

2. We now want to create our service. Create a file in the `nodecheck` directory named `main.js`. Open it for editing and declare the DDP connection by adding the following code:

   ```
   var DDPClient = require("ddp");

   var ddpclient = new DDPClient({
       host: "localhost",
       port: 3000,
       path: "websocket",
       ssl: false,
       autoReconnect: true,
       autoReconnectTimer: 500,
   ```

```
    maintainCollections: true,
    ddpVersion: '1'
});
```

3. We now want to connect to our Meteor server using the `.connect()` function and display the result of a successful subscription in the console window. Append the following code to `main.js`:

```
ddpclient.connect(function (error) {
    if (error) {
        console.log(error);
    } else console.log('successful connection');

    ddpclient.subscribe('urls', [], function () {
        var urls = Object.keys(ddpclient.collections.urls);
        console.log(urls);
    })
});
```

4. Finally, we will listen to the messages coming across the DDP wire using the `.on()` function. We will check all the `added` and `changed` messages and replace the URL with a secure URL where needed, using the `.call()` function to invoke the remote Meteor server `updateSRC` method. Append the following code to `main.js`:

```
ddpclient.on('message', function (msg) {
    var message = ddpclient.EJSON.parse(msg);
    switch (message.msg) {
    case "added":
    case "changed":
        var url = message.fields.src;
        if (url) {
            if (!url.startsWith('https')) {
                message.fields.src = url.replace('http:',
'https:');
                ddpclient.call('updateSRC',
                    [message.id, message.fields.src],
                    function (err, success) {
                        if (!err) console.log(success);
                    })
            }
        }
        break;
    default:
        break;
    }
});
if (typeof String.prototype.startsWith != 'function') {
```

```
    String.prototype.startsWith = function (str) {
        return this.slice(0, str.length) == str;
    };
}
```

Okay, we're now ready to run our Node service. Save all your changes, and make sure you have an instance of your original Meteor project open in a browser (usually `http://localhost:3000`) so that you can see the changes. They'll happen quickly, so don't blink!

5. In a terminal window, navigate to your `nodecheck` folder and enter the following command:

   ```
   $ node main.js
   ```

   In your web browser, every link that was previously using `http` will now be using `https`. If you add another link and intentionally try to enter the link as `http`, the running node service will immediately change it to `https`. If you look in the terminal window where you ran the node `main.js` command, you'll see the word **updated** for every converted link and your web browser will look something like the following screenshot:

How it works...

The Meteor server is listening for client subscription requests using DDP. Since we are using a DDP `client` library built for Node (using the `ddp npm` module), we are able to declare our connection using the new `DDPClient()` function, connect and subscribe to the `urls` channel, and monitor all messages coming through DDP using the `.on()` function:

```
var ddpclient = new DDPClient({
  host: "localhost",
  port: 3000,
  ...
});
ddpclient.connect(function (error) {
  ...
  ddpclient.subscribe('urls', ... );
  ...
});
ddpclient.on('message', function (msg) {
  ...
});
```

Finally, if we want to make a modification, we can do so by invoking the Meteor server's remote `methods` using the `.call()` function:

```
ddpclient.call('updateSRC', ... );
```

You'll notice that we had to do nothing on the Meteor server side to get this to work. That's the beauty of DDP; the protocol is the same for any compatible DDP library. As long as both systems are using the DDP protocol, the languages, hardware, and operating systems of the systems are immaterial and don't affect communications between the servers.

As you can tell, the DDP protocol is quite powerful because it's simple and clean. There are many other aspects to DDP, for which you will want to consult the Meteor and the corresponding GitHub documentation.

There's more...

You can see a fairly up-to-date list of available DDP clients by visiting Meteoropedia at `http://www.meteorpedia.com/read/DDP_Clients`.

See also

- The *Using client-only collections* and *Implementing multiserver DDP* recipes in this chapter

6

Mastering Reactivity

In this chapter, you will learn the following topics:

- Creating and consuming a reactive value
- Using Ajax query results in ReactiveVar
- Making a custom library reactive
- Updating Blaze templates without Mongo
- Using inline data to modify UI elements reactively
- Integrating the jQuery UI

Introduction

Reactive programming is an emerging development methodology, where changes to data automatically trigger changes to the rest of the system. This allows you, the developer, to write code declaratively and let the reactive elements manage any changes. Meteor is, perhaps, the best and most fully developed implementation of reactive programming available today. By understanding the core concepts of reactive programming, you can use the `Tracker` (formerly `Deps`) library to create simple, declarative code while avoiding the usual pitfalls associated with reactive and asynchronous JavaScript programming. The recipes in this chapter will give you simple, clear examples of how the major components of Meteor's reactive model work.

Creating and consuming a reactive value

`Tracker`, simply put, is Meteor's variable tracking system. It is used to manage reactive values, data structures, and computations (functions that consume reactive values). This recipe will show you how to create reactive values, and perform computations on those values, using `Tracker.autorun()`. In other words, it will teach you how reactive programming works inside Meteor. This recipe will come in handy as a foundation for more complex functionalities.

Getting ready

For the sake of simplicity, we will be using a default Meteor project, with the `reactive-var` package added to it. Open a terminal window, navigate to where you would like to create your root project, and execute the following commands:

```
$ meteor create reactiverecipes
$ cd reactiverecipes
$ meteor add reactive-var
$ meteor
```

You are now ready to start using reactive variables.

How to do it...

We are going to modify the text of a button, based on a reactive variable; so we will need to create the button and hook up the reactive context.

1. Open your project in your favorite text editor and edit the `reactiverecipes.html` file, adding an ID to the `<button>` element, as shown in the following example:

   ```
   <button id='btnReact'>Click Me</button>
   ```

2. Now, open `reactiverecipes.js` and add the following lines of code just below the `if (Meteor.isClient)` condition:

   ```
   if (Meteor.isClient) {
      btnText = new ReactiveVar('Click Me');
      Tracker.autorun(function () {
          $('#btnReact').text(btnText.get());
      });
   ```

3. Finally, add the following line inside the `Template.hello.events` declaration, just below the `Session.set()` function in the `'click button'` function:

   ```
   Session.set("counter"...);
   btnText.set('Again!');
   ```

4. Save your changes and navigate to `http://localhost:3000` in a browser. Once there, click on the button labeled **Click Me** and watch the text change to **Again!**:

You can manually change the value of the button text by opening a console window in your browser and using the `btnText.set()` command, as shown in the following example:

```
> btnText.set('Pretty please...')
```

The button text will change to whatever value you set it to, instantly.

How it works...

The preceding example seems oversimplified, but we have two things to say about that.

First, it's simple because Meteor was built to make your code very simple and easy to follow. This makes your development and debug efforts significantly less time consuming.

Notice how the process of declaring a reactive variable is made up of one line of code. When we added the `btnText = new ReactiveVar('Click Me')` statement, we were simply declaring a variable (and initializing its value to `'Click Me'`), but we know by the declaration that it is a reactive variable.

Next, we encapsulated an extremely straightforward jQuery statement inside a `Tracker.autorun()` block. This block is called a **reactive computation**. Reactive computations run once initially and then rerun (are recomputed) whenever a change is made to any reactive variables contained inside. So, in this example, we told `Tracker` to monitor the value of `btnText`(a reactive variable) and automatically run (hence the term `autorun`) the code block again when it would change.

Notice that we don't have to worry about any time conditions, such as "is this the first run?" or "okay, when there's a change...." We just simply declare the jQuery statement and let `Tracker` figure out the timing for us.

This is what the term **transparent reactive programming** means. There is one set of code for the initialization and another set of code for changes. Besides the variable declarations, your entire code base can be written as normal, plain old JavaScript.

Second, what it's doing under the hood is anything but simple! To create this frontend simplicity for you, the programmer, Meteor implements reactive providers and reactive computations.

We'll sacrifice a bit of fidelity to make the concept simpler to understand. When `Tracker.autorun` is called, it does four things:

- It creates a computation
- It sets `Tracker.currentComputation` to that computation
- It calls the function that was passed to it
- It sets `Tracker.currentComputation` to `null`

A computation is essentially an event handler function and contains a reference to the function that was passed to `Tracker.autorun`. The event that the computation is waiting for is a call to the `computation.invalidate` method. When the `invalidate` method is called, the computation reruns the function it contains.

Now we come to the function. The function passed to `Tracker.autorun` is considered a reactive function if it contains **reactive providers**. A reactive provider is an object that has functions to get and set some value, and keeps track of dependencies. When the `get` function is called, it does three things:

1. It checks to see whether `Tracker.currentComputation` has a value.
2. If it does, the computation is added to an internal list.
3. It returns the value of the variable that the getter requested.

Both steps 1 and 2 are performed by making a call to `depend()`, which is a method found on a dependency object. The `reactive-var` library automates this part, so you don't have to call the `depend()` method directly. All you have to do is use the reactive variable!

Likewise, when the `set` function is called, it does two things:

- It changes the value of the internal variable
- For every computation in the internal list, the `invalidate()` method is called

The `invalidate()` methods are called in a loop by making a call to `changed()`, which is a `helper` method found on a dependency object. Again, `reactive-var` takes care of this for you. You're welcome!

When each computation is invalidated, it reruns the function that it contains. And the entire loop starts over with the newly rerun function calling the getters, which return their values (like the `text` value of our `btnReact` button) and add the computations to the provider's internal list, waiting once again for the setter to be called.

Even though they are extremely oversimplified (any core MDG members reading this are probably spitting coffee right now...), here is what the `ReactiveVar` and `autorun` objects do:

```
function ReactiveVar(value) {
    // create the dependency object (a helper class)
    var _dependency = new Tracker.Dependency;
    //set the default internal value
    var _internalVal = value;

    var getValue = function () {
        // call depend(), which adds the computation
        _dependency.depend();
        // return the internal value
        return _internalVal;
    };

    var setValue = function (newValue) {
        // update the internal value
        _internalVal = newValue;
        // loop through computations and call invalidate()
        _dependency.changed();
    };

    return this;
}


function autorun(func) {
    // creates computation and assigns it to
    // Tracker.currentComputation
    var computation = new Tracker.Computation(func, Tracker.
currentComputation);

    // Calls the onInvalidate method the first time,
    // so that func function will run
    Tracker.onInvalidate({...});

    return computation;
}
```

There are two things that we left out, for clarity, but are important in order to have a complete understanding. First, the `depend()` helper method also sets up an `onInvalidate()` listener, which removes the computation from the reactive provider's internal list. Second, computations are checked to see whether they already exist in an internal list before they are added.

Why are the computations removed when the computation is invalidated, you may ask? The answer, in short, is that it makes the entire computation `add-execute-remove` loop very elegant. It keeps all the computations up to date and the dependent functions only run once each, no matter how many times a getter is called inside the same function. If they weren't removed, the functions would be run multiple times, which is *no bueno*.

So let's review what we did in this recipe:

- The `autorun` method creates a computation and passes your function to this computation
- The computation has an `onInvalidate` method that, among other things, runs your function
- After `autorun` has created a computation, it runs your function once, using the `onInvalidate` method
- Your function has reactive variables in it, which have to-do lists
- As your function runs, getters are called, which add the computations to the to-do lists
- Setters are also called, which execute the to-do lists and clear them
- Because your functions in the to-do lists have reactive variables, the process is repeated (the functions are rerun)
- Lather, rinse, and repeat

Again, this explanation is drastically simplified, and therefore, pretty inaccurate; however, conceptually, it will hopefully give you a good understanding of what's happening under the hood.

There's more...

Meteor also provides the `ReactiveDict` object, which runs exactly like `ReactiveVar`, but can store collections of reactive variables in key-value pairs. The syntax is exactly the same as for `ReactiveVar`, but you will need to add a key to the `set` and `get` methods, as shown in the following code:

```
btnText.get(); // ReactiveVar
txtFields.get('btnText'); // ReactiveDict
...
btnText.set('Click Me'); // ReactiveVar
txtFields.set('btnText','Click Me') // ReactiveDict
```

To use `ReactiveDict`, simply add the `reactive-dict` package using the following terminal command:

```
$ meteor add reactive-dict
```

Lastly, you don't have to use `ReactiveVar`, or `ReactiveDict`, and can instead *roll your own* reactive providers. Please see the *Making a custom library reactive* recipe found in this chapter as an example.

See also

- The *Creating dynamic lists* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Making a custom library reactive* recipe in this chapter

Using Ajax query results in ReactiveVar

Whenever we use Ajax, requesting (and even receiving) data is pretty easy. The complications come in when we have to update the UI with new or updated data. With Meteor's reactive programming capabilities, this is no longer an issue. In this recipe, you will see how to update your UI with Ajax results, using Meteor's `ReactiveVar` library.

Getting ready

To get up and run quickly, we will use a default Meteor project with a few packages added. Open a terminal window, navigate to where you would like to create your root project, and execute the following commands:

```
$ meteor create ajaxreactive
$ cd ajaxreactive
$ meteor add reactive-var
$ meteor add http
$ meteor add twbs:bootstrap
$ meteor
```

We are now ready to build a reactive Ajax query!

How to do it...

We will be pulling the weather data from `openweathermap.org`, using their free (but for testing only) API. We will take the results from our `openweathermap.org` queries and put them into a `ReactiveVar` library so that they can then be consumed reactively by our Blaze templates.

Let's start by modifying the UI:

1. Open `ajaxreactive.html` and add a call to our soon-to-be-created `weather` template, just under the call to the existing `hello` template, as shown in the following code:

   ```
   {{> hello}}
   {{> weather}}
   ```

2. We will also want to repurpose the counter in our `hello` template to tell us what the starting longitude will be. Change the description in the `<p>` element inside the `hello` template, as follows:

   ```
   <template name="hello">
       <button>Click Me</button>
       <p>You are starting at {{counter}} longitude.</p>
   </template>
   ```

3. Next, add our `weather` template, which is just a simple table with a bit of prettiness added, thanks to bootstrap! At the bottom of `ajaxreactive.html`, add the following template:

   ```
   <template name="weather">
       {{#if reports}}
       <table class="table">
           <thead>
               <th>name</th>
               <th>weather</th>
               <th>temp</th>
               <th>humidity</th>
           </thead>
           <tbody>
       {{#each reports}}
               <tr class={{severity}}>
                   <td>{{name}}</td>
                   <td>{{description}}</td>
                   <td>{{temp}}</td>
                   <td>{{humidity}}</td>
               </tr>
       {{/each}}
           </tbody>
       </table>
       {{/if}}
   </template>
   ```

4. Save your changes and navigate to `http://localhost:3000`. While nothing (except the description of the counter) will change, we will very soon want to view our weather data.
5. Let's open `ajaxreactive.js` and declare our `ReactiveVar` libraries. Just below `Session.setDefault()`, add the following code:

```
Session.setDefault("counter", 0);
weatherlist = new ReactiveVar;
```

6. We will now modify the `'click button'` function to increment the counter and make our Ajax call. Locate the function inside the `Template.hello.events` section and modify the code as follows:

```
// increment the counter when button is clicked
if (Session.get("counter") <= 60)
    Session.set("counter", Session.get("counter") + 4);
else
    Session.set("counter", 0)

getWeather();
```

> Don't forget to update the counter increment to `4`; otherwise, your `weather` data won't change much as you click.

7. Next, we need to add `Template.weather.helpers` so that our UI will populate properly. Just after the `Template.hello.events` section, add the following:

```
Template.weather.helpers({
    reports: function () {
        if (!weatherlist) return false;
        return weatherlist.get();
    },
    severity: function () {
        if (this.weather && this.weather[0].main == "Clear")
            return "success";
        else
            return "warning";
    },
    description: function () {
        if (this.weather && this.weather[0])
            return this.weather[0].description;
        else
            return "";
    },
```

```
    temp: function () {
        if (this.main)
            return this.main.temp;
        else
            return "";
    },
    humidity: function () {
        if (this.main)
            return this.main.humidity;
        else
            return "";
    }
});
```

8. Finally, we need to add our Ajax call and the asynchronous callback function once the result comes in. Just after the `Template.weather.helpers` section, add the following two functions:

```
function getWeather() {
    var long1 = +Session.get("counter"),
        long2 = long1+5;
    HTTP.get("http://api.openweathermap.org/data/2.5/
  box/city?bbox=12,"+long1+",15,"+long2+",10&cluster=yes",
  harvestWeather);
}
function harvestWeather(error, data) {
    if (data && data.content) {
        var weather = EJSON.parse(data.content);
        weatherlist.set(weather.list);
    }
}
```

9. Save all your changes and click on the button on the project page in your browser. You should see something similar to the following screenshot:

name	weather	temp	humidity
Ganye	overcast clouds	24.97	81
Meiganga	few clouds	18.43	43
Garoua Boulai	Sky is Clear	19.81	37
Tchollire	Sky is Clear	16.43	74
Batouri	overcast clouds	23.06	78
Nanga Eboko	broken clouds	22.48	71
Nkoteng	broken clouds	22.63	75
Bertoua	broken clouds	22.18	61
Belabo	broken clouds	22.31	65
Ngaoundere	few clouds	16.26	41
Tibati	Sky is Clear	21.26	31

As you click, the weather results for a given area, moving towards north, will be displayed.

How it works...

In this instance, we didn't use a Mongo collection, which is automatically reactive. Instead, we made calls to `openweather.org api` and updated a `ReactiveVar` library (in this case, the `weatherList` variable) using the `set` method. Because the template helpers rely on a call to the `get` method of that same `ReactiveVar`, they are automatically rerun when the `ReactiveVar` is updated. Let's break it down.

We first created the `weather` template in our HTML:

```
<template name="weather">
  {{#if reports}}
  ...
  {{/if}}
</template>
```

The template iterates over the `reports` helper object using an `{{#each...}}` block, populating an HTML table with the results.

Next, inside our client-side JavaScript, we declared our reactive variable, `weatherlist`, using a new `ReactiveVar` library:

```
weatherlist = new ReactiveVar;
```

We then used `weatherlist.get()` in our `reports` helper object, which is in `Template.weather.helpers`:

```
reports: function () {
  ...
  return weatherlist.get();
}
```

By using it here, we set up a dependency so that anytime `weatherlist.set()` is called, the data for the template is refreshed and the template is updated/rerun.

Finally, we hooked up our button to an Ajax call using `HTTP.get()`, and we passed the `harvestWeather` function as a callback (`HTTP.get(url,arg,callback)`):

```
function getWeather() {
  var long1 = ...
  HTTP.get("...", harvestWeather);
}
```

Once the callback is triggered, it massages the data from the Ajax call and repopulates our reactive variable using `weatherlist.set()`:

```
function harvestWeather(error, data) {
  ...
    weatherlist.set(weather.list);
}
```

As mentioned in the preceding section, when this `set` function is called, it invalidates the template functions and reactively updates our UI.

You can see very clearly in the callback function (`harvestWeather`), and in the weather template helper function (`reports`), that the calls are regular, plain JavaScript. We're literally just calling a `get` or `set` function. Because the object we're calling those functions on is a `ReactiveVar`, all the reactive dependencies and UI updates are handled for us.

You can begin to quickly see how powerful Meteor's reactive programming model is. Instead of messing around with events and handlers or worrying about callback hell, we used a simple, clean `set` command and let Meteor handle all the details for us.

See also

- The *Creating dynamic lists* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Creating and consuming a reactive value* recipe in this chapter

Making a custom library reactive

Because we so often deal with variables and data, it can go unnoticed that Meteor's reactivity doesn't only work with reactive values. Any function from any JavaScript library can be turned into a reactive provider. This recipe will show you how to create your own reactive providers using the `Tracker.depend()` and `Tracker.changed()` commands.

Getting ready

To keep the example simple, we will use a default Meteor project, with a `bootstrap` package, and a random color generator. Open a terminal window, navigate to where you would like to create your root project, and execute the following commands:

```
$ meteor create customreactive
$ cd customreactive
$ meteor add twbs:bootstrap
$ meteor add rzymek:randomcolor
$ meteor
```

How to do it...

Let's pretend you have a (pretty awesome) library called `colorsaurus`. Your `colorsaurus` object likes to roar. A lot. Mostly because "rawr" means "I love you" in dinosaur, but also because the colorsaurus wants to share as many random colors as possible with all his friends. Whenever you ask this motley beast for a color, he instantly gives you a random color. This is obviously the most useful library ever written, so let's get to work building it!

1. Open `customreactive.js` and add the following `colorsaurus` object just below the `Template` declarations, inside the `Meteor.isClient` conditional:

```
colorsaurus = {
    color: function(){
        return randomColor();
    },
    rainbowRoar: function (){
        console.log('rawr');
    }
};
```

2. While we're in `customreactive.js`, let's add the `numcolor` helper function. Locate the `Template.hello.helpers` method and add the following to the top of the method:

```
Template.hello.helpers({
  numcolor: function(){
    return colorsaurus.color();
  },
  counter: ...
});
```

3. Save your changes, navigate to `http://localhost:3000`, and in the console window, type the following commands:

```
> colorsaurus.color()
> colorsaurus.rainbowRoar()
```

4. You should get a random color and a nice, short *I love you* from our friend, the `colorsaurus` function:

```
> colorsaurus.color()
<· "#4dd666"
> colorsaurus.rainbowRoar()
  rawr
```

5. It's not reactive yet, but we need to prepare our UI first, for all the awesomeness that a reactive `colorsaurus` function can unleash. Open `customreactive.html` and make the following modifications to the `hello` template:

```
<button id='btnColor'>Click Me</button>
<p style="color:{{numcolor}}">You've pressed the button
{{counter}} times.</p>
```

6. We now need to change our `click` event to make `colorsaurus` roar. In `customreactive.js`, modify the `Template.hello.events` section as follows:

```
Template.hello.events({
    'click button': function () {
      // increment the counter when button is clicked
      Session.set("counter", Session.get("counter") + 1);
      colorsaurus.rainbowRoar();
    }
  });
```

7. All that's left to do is make `colorsaurus` reactive and set up an `autorun` function. Open `customreactive.js` again and add the following reactive statements, including adding the `Tracker.Depenency` object:

```
var colorDep = new Tracker.Dependency;

colorsaurus = {
    color: function(){
        colorDep.depend();
        return randomColor();
    },
    rainbowRoar: function (){
        console.log('rawr');
        colorDep.changed();
    }
};
```

8. Now, add a `Tracker.autorun` function, immediately after the `colorsaurus` code block, as shown in the following code:

```
Tracker.autorun(function(){
    $('#btnColor').css('background-color' , colorsaurus.color());
    $('body').css('background-color', colorsaurus.color());
 });
```

9. Save your changes, go to your browser, and click on the button as many times as your little 'colorsaurus-lovin' heart wishes. You'll get some really great color combinations, as shown in the following screenshot:

How it works...

Notice how the button, text, and the page background change, and they all change to random, separate colors. That's because we used a reactive library function that returns a random color each time it is called instead of returning a set variable.

By having our `color()` function return the result of `randomColor()`, we are ensuring that every time `colorsaurus.color()` is called, we get a different result.

We added `colorDep.depend()` to the returning function, which logs the computations that are created by either `Tracker.autorun` or by a reactive template (see the *Creating and consuming a reactive value* recipe found in this chapter for a full explanation).

Finally, we called `colorDep.changed()`, which runs the logged computations.

Every part of the code is separate from—and therefore not dependent on—the other code parts or libraries. Through the `Tracker.Dependency` object, Meteor keeps track of everything for us so we can add or remove reactive dependencies at will. Try, for example, running the following line in your browser console:

```
> Tracker.autorun(function(){console.log(colorsaurus.color());});
```

Now, every time you click the button on the page, you get yet another random color from the `colorsaurus`, printed to your console:

```
rawr
#5a2ea5
rawr
#bf05a9
rawr
#ffcceb
rawr
#ed9aa3
rawr
#f7ff16
```

This is reactive programming at its very best. Rawr!

See also

- The *Using reactivity with HTML attributes* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Creating and consuming a reactive value* recipe in this chapter

Updating Blaze templates without Mongo

Not everything in our UI has to be dependent on Mongo collections. We can, in fact, use pretty much any reactive object inside our templates, and changes will appear instantly. This recipe will quickly show you how to use custom collections to populate and update UI templates.

Getting ready

Let's start with a base Meteor project and add the `bootstrap` and `reactive-var` packages. In a terminal window, navigate to where you would like your project to reside and enter the following commands:

```
$ meteor create mongoless
$ cd mongoless
$ meteor add twbs:bootstrap
$ meteor add reactive-var
$ meteor
```

Finally, open a browser and navigate to `http://localhost:3000` so that you can see updates in real time.

How to do it...

We will create a simple array of button press snapshots, with a new element being added to the array every time the button on the page is clicked. Subsequently, these buttons will be added to the UI using a `{{#each}}` template block.

1. First, open `mongoless.html` and add the following block to the `hello` template, just after the `<p>` element and just before the closing `</template>` tag, as shown in the following example:

```
{{#each buttonPresses}}
    <div class="btn btn-info pressed">#{{btnRank}}</div>
{{/each}}
</template>
```

2. We now need to add a reactive variable and append some helpers to the `Template.hello.helpers` object. Open `mongoless.js` and add the following highlighted code:

```
if (Meteor.isClient) {
  presses = new ReactiveVar;
  // counter starts at 0
  Session.setDefault("counter", 0);

  Template.hello.helpers({
    counter: function () {
      return Session.get("counter");
    },
    buttonPresses: function(){
        return presses.get();
    },
    btnRank: function(){
        return this.rank;
    }
  });
```

3. All that's left to do is to update the `presses` variable each time the button is pressed. Inside `Template.hello.events`, in the `'click button'` event handler, add the following lines of code immediately after the `Session.set()` call:

```
Session.set("counter", Session.get("counter") + 1);
var _presses = presses.get() || [];
_presses.push({rank:Session.get("counter")});
presses.set(_presses);
```

4. Save all of your changes, go to your browser, and start clicking on the button labeled **Click Me**. You should see a new button created each time you click on the button, similar to the following:

How it works...

When we added the `buttonPresses` helper function to the `Template.hello.helpers` object, we simply replaced what we would usually use a Mongo collection for with a simple array stored inside a `ReactiveVar` element:

```
presses = new ReactiveVar;
```

Collections are reactive providers, which means they track and rerun computations as appropriate. The `presses` object is also a reactive provider, and therefore, does the exact same thing. It reruns any stored computations / reactive functions whenever the value is updated. In this case, it reruns computations when the array is modified, and `presses.set()` is called as a result.

See also

- The *Creating dynamic lists* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Creating and consuming a reactive value* recipe in this chapter

Using inline data to modify UI elements reactively

Typically, when elements in an HTML page are rendered, then that rendering isn't directly linked to any of the data used to create them, for example, if we have an array of objects, we may generate some HTML by iterating over the array and adding `<div>` elements for each object in the array. Unless we do something to manually link them to the array of objects, these newly created elements aren't associated with the data that created them in any way. This leads to all kinds of development shenanigans, as developers try to shoehorn in associative data, which is used in events and other downstream functions. Long story short, using only existing web technologies, it's difficult to keep all of the data exactly in sync with your HTML DOM elements. Meteor has been designed to help solve this problem gracefully, keeping track of the context of each DOM element and therefore allowing instant access to the data used to create the element in the first place. This recipe will walk you through how to retrieve and use the data associated with individual DOM elements.

Getting ready

We will use the code base from the *Updating Blaze templates without Mongo* recipe, found in this chapter. Please complete that recipe and then add the `randomcolor` package by running the following terminal command in the root folder of your project:

```
> meteor add rzymek:randomcolor
```

Have your browser open to `http://localhost:3000` as well so we can see the changes in real time.

How to do it...

We are going to add some functionality to the existing button creation. First, we'll add a random color to each new button; second, we'll remove this color when the buttons are clicked; and third, we'll restore the color based on the associative inline data.

So let's get started. We need to update the `hello` template, setting the initial background colors for new buttons. We also need a way to remove those colors at random. We'll do this by adding a new control button.

1. Open your `.html` file (probably `mongoless.html`) and modify the `hello` template to look like the following:

```
<template name="hello">
  <button id="addBtn">Click Me</button>
  <button id="chgColor">Or Me!</button>
  <p>You've pressed the button {{counter}} times.</p>
  {{#each buttonPresses}}
```

```
    <div class="btn btn-info pressed"
         style="background-color:{{btnColor}}">
    #{{btnRank}}</div>
  {{/each}}
</template>
```

2. We now need to add the `btnColor` helper to `Template.hello.helpers` and modify the data object being stored to make room for the new color. We also need to refine the `click` events on the buttons to differentiate between adding a new button and removing a button's color. Open your `.js` file (probably called `mongoless.js`) and make the following changes to the variable declarations and the helpers:

```
if (Meteor.isClient) {
  presses = new ReactiveVar;
  counter = new ReactiveVar(0);


  Template.hello.helpers({
    counter: function () {
      return counter.get();
    },
    buttonPresses: function () {
      return presses.get();
    },
    btnRank: function () {
      return this.rank;
    },
    btnColor: function () {
      return this.color;
    }
  });
  ...
```

3. Now, we need to modify the existing `click` event, add a new event to remove color, and add one final handler to set up all the new buttons to regain their color when clicked. Make the following changes to the `Template.hello.events` section:

```
Template.hello.events({
    'click #addBtn': function () {
      // increment the counter when button is clicked
      counter.set(counter.get() + 1);
      var _presses = presses.get() || [];
      var newBtn = {
        color: randomColor(),
        rank: counter.get()
      };
```

```
      _presses.push(newBtn);
      presses.set(_presses);
    },
    'click #chgColor': function () {
      var rndBtn = ~~(Math.random() * counter.get());
      $('.pressed')[rndBtn].style.backgroundColor = '';

    },
    'click .pressed': function (e, n) {
      e.currentTarget.style.backgroundColor = this.color;
    }
  });
```

4. Save all your changes and click on the **Click Me** Button on your screen 5-10 times. You'll notice that all the new buttons being added have a random color assigned to them, as shown in the following screenshot:

5. Now, click on the **Or Me!** button multiple times. At random, the buttons will lose their random color and change to the default `btn-info` blue color provided by `bootstrap`, as shown in the following screenshot:

Any of the buttons that have lost their color can regain the color by clicking directly on the button.

How it works...

The crux of this recipe is found inside the `'click .pressed'` event handler. There, we assign the `backgroundColor` attribute of the clicked button to `this.color`:

```
'click .pressed': function (e, n) {
  e.currentTarget.style.backgroundColor = this.color;
}
```

In this case, there is a reference to the associated data object for that DOM element. Meteor keeps track of how each element is created. So when an event is fired inside the template, Meteor provides the data object as the *context* (the `this`) in the event handler. In this way, Meteor can provide instant access to the inline data for each rendered element.

Notice that even after we manually changed the `backgroundColor` using the `'click #chgColor'` event, Meteor still has a reference to the data object used to render the element. This becomes important because we now no longer need to store data as an attribute of the DOM element—no more `data-color` or `data-whatever` attributes that clutter up the UI and potentially expose data. The data objects, though hidden from the UI, are available instantly and inline. So no fancy calculations or DOM manipulation needs to be done in order to access the necessary contextual data.

There's more...

The preceding example uses the default Meteor `Template` event handlers, so of course, the data is available. But, even if you bypass Meteor's `Template` event handlers and use, say, jQuery events instead, the associated data will be available by calling the `Blaze.getData()` function of the element.

Setting up the event handler is a little bit tricky in this case. We have to first move the population of each button to a new template because the jQuery `click` event handler must be run inside of a `rendered()` function callback. To accomplish this, make the following changes to your `.html` file (probably `mongoless.html`):

```
<template name="hello">
  <button id="addBtn">Click Me</button>
  <button id="chgColor">Or Me!</button>
  <p>You've pressed the button {{counter}} times.</p>
  {{#each buttonPresses}}
    {{> helloBtn}}
  {{/each}}
</template>
```

```
<template name="helloBtn">
  <div class="btn btn-info pressed"
    style="background-color:{{btnColor}}">
    #{{btnRank}}</div>
</template>
```

You will also need to remove the `'click .pressed'` event handler from `Template.hello.events`. Once you've removed the event handler, add the `jQuery.click()` event handler inside a `Template.btnRank.rendered` code block, to be run immediately upon the rendering of a new button, as shown in the following code:

```
Template.helloBtn.rendered = function () {
    this.$('.pressed').click(function (e) {
      e.currentTarget.style.backgroundColor = Blaze.getData(this).
color;
    });
  };
```

Lastly, because we moved the `div` rendering to the new `helloBtn` template, we need to move the `btnRank` and `btnColor` helpers from `Template.hello.helpers` to a newly created `Template.helloBtn.helpers` block, as shown in the following code:

```
Template.helloBtn.helpers({
  btnRank: function () {
    return this.rank;
  },
  btnColor: function () {
    return this.color;
  }
});
```

That's a lot of work for the same result, but it helps to illustrate the flexibility of the Blaze/Meteor associative data capabilities. We only had to slightly modify the event helper to point to `Blaze.getData(this).color` rather than `this.color`. Although the same keyword exists in each event handler, the `this` inside the jQuery event handler refers to the DOM element, rather than the associated data object inside the original Meteor event handler. `Blaze.getData(element)` takes a DOM element as an argument and retrieves the associated inline data for that element.

In either case, getting to the associated data is very straightforward and allows you to do anything to your UI programmatically, without having to worry about destroying/altering the data associated with each rendered element.

See also

- The *Inserting templates with Spacebars* and *Creating dynamic lists* recipes in *Chapter 3*, *Building Great User Interfaces*

Integrating a jQuery UI

The jQuery library is crazily popular, and for good reason. When used properly, it can speed up the development process and give us reliable ways of doing things that would otherwise take a lot of coding effort.

A complement to jQuery is jQuery UI, which is a set of widgets, themes, and animation effects. With jQuery UI, you can quickly create drag and drop components, sortable lists, and lots of other useful UI niceties.

This recipe will walk you through creating a jQuery UI-sortable widget inside a Meteor template.

Getting ready

For this recipe, we will definitely want client and server folders to keep the code clean and readable. To accomplish this, we will rely on our default template scaffolding. Please create a new project called `swatches` using the *Setting up your project file structure* recipe in *Chapter 1*, *Optimizing Your Workflow*, as your starting file structure.

Once you've completed the scaffolding, we will need to add the `randomcolor` package to our project. In a terminal window, navigate to the root folder of your project and run the following command:

```
> meteor add rzymek:randomcolor
```

We will also want a nice theme, so let's use the adapted version of Google's Material Design theme, for funzies. Enter the following command in the same terminal window:

```
> meteor add html5cat:bootstrap-material-design
```

Next, we want to get a customized version of jQuery UI directly from `jqueryui.com`. Navigate to `http://jqueryui.com/` in a browser and click on **Custom Download**, placed toward the right:

In the download builder, make the following selections:

- Select the latest stable version
- Uncheck the **Toggle All** checkbox under **Components**
- Check the **Toggle All** checkbox under **UI Core**
- Check the **Sortable** checkbox under **Interactions**
- At the bottom, select **No Theme** from the **Theme** dropdown

Your selections will look similar to the following screenshot:

All we need is the sortable interaction and the UI core. Everything else just adds to our file size, so we'll leave it out. Click on **Download**, unzip the downloaded file once complete, locate the `jquery-ui.min.js` file, and copy it to your `[project root]/client/lib/scripts` folder.

We could simply pull in jQuery UI using a community package, but it's a bit bulkier and it doesn't help us see how non-packaged third-party libraries can be used inside Meteor. So, we'll go with this manual installation.

Open a browser to `http://localhost:3000` so we can see the changes in real time. We are now ready to add the jQuery UI-sortable widget to our project.

How to do it...

We will create color swatches, which will display the hexadecimal code they represent, and they will be sortable, which means we can move them around via drag and drop.

1. To accomplish this, we first need to create a `Swatches` collection, accessible on both the client and the server. In your `[project root]/both/` folder, create/edit a file named `model.js` and add the following collection declaration:

   ```
   Swatches = new Mongo.Collection('swatches');
   ```

2. Next, let's create our UI using a template named `swatches`. Open/create the `[project root]/client/main.html` file, remove all the contents, and add the following code:

   ```
   <head>
     <title>Swatches</title>
   </head>

   <body>
     {{> colors}}
   </body>

   <template name="colors">
     <h1>Yay Colors!</h1>
     <div id="cList">
         {{#each swatches}}
             <div class="swatch"
    style="background-color:{{color}}">
   {{color}}</div>{{color}}</div>
         {{/each}}
     </div>
     <input id="btnNew" type="button" class="btn btn-primary"
   value="New Color" />
   </template>
   ```

3. We will want to style it just a little bit to make the swatches consistently sized. Open/create a file named `[project root]/client/lib/styles/style.css` and add the following CSS declarations:

```
.swatch {
    min-height: 100px;
    min-width: 100px;
    display:inline-block;
    color:white;
    text-align: center;
}
#cList {
    width: 520px;
}
```

4. Finally, we will create the logic needed to add swatches and be able to drag them around the screen. Open/create the `[project root]/client/scripts/main.js` file, delete anything inside the file, and add the following `Template.helpers` and `Rankings` declarations:

```
Template.colors.helpers({
    swatches: function(){
        return Swatches.find({},{ sort: { rank:1}});
    }
})

Rankings = {
    beforeFirst: function(first) { return first - 1;},
    middle: function(before,after){ return (before+after)/2;},
    afterLast: function(last){ return last + 1; }
};
```

5. Now, the fun part! We will create the `jQuery.sortable` object with its `stop()` function and hook up our `button.click` event handler using regular jQuery. In order for the `sortable` and `click` event handlers to be added properly, we need to declare them inside a `Template.rendered()` function. In the same `main.js` file, just below the `Rankings` declaration, enter the following code:

```
Template.colors.rendered = function(){
    this.$('#cList').sortable({
        stop: function (e,ui){
            var el = ui.item.get(0);
            var before = ui.item.prev().get(0);
            var after = ui.item.next().get(0);
            var newRank = null;
            if (!before){
```

```
                newRank = Rankings.beforeFirst(Blaze.
getData(after).rank);
            } else if (!after) {
                newRank = Rankings.afterLast(Blaze.
getData(before).rank);
            } else {
                newRank = Rankings.middle(Blaze.getData(before).
rank,Blaze.getData(after).rank);
            }
            Swatches.update(Blaze.getData(el)._id,
                            {$set: {rank:newRank}});
        }
    });

    this.$('#btnNew').click(function(e){
        var newColor = randomColor({luminosity:'random',
hue:'random'});
       Swatches.insert({color:newColor, rank: Swatches.find().
count()+1});
    });
};
```

Save all of your changes and hop over to your browser. The page should have a nice, stylish blue button labeled **NEW COLOR**. Every time you click on this button, a new swatch will be added with a random color. If you drag and drop any of the swatches from one position to another, the swatches will be reordered appropriately. This reordering is not temporary. If you refresh the page or open another browser window, the reordering you did via drag and drop will remain.

So, for example, let's say you move a swatch from the last element to the first. The change will stick and any other clients/browsers that open to the same page will instantly reflect the change.

When the following purple swatch is dragged and dropped, the changes will be as shown, as displayed on the right-hand side of the following screenshot:

It is instantly updated in every UI, as shown in the following screenshot:

How it works...

Thanks to our `click` event handler, whenever a swatch is added using `Swatches.insert()`, a rank is assigned to that swatch:

```
this.$('#btnNew').click(function(e){
  ...
  Swatches.insert({..., rank: Swatches.find().count()+1});
});
```

If we look at the `swatches` helper inside `Template.colors.helpers`, we can see that the Mongo `Collection.find()` query is sorted by `rank`:

```
swatches: function(){
  return Swatches.find({},{ sort: { rank:1 }});
}
```

This preserves the order of the swatches in the UI and allows us to manipulate their order as a result of drag and drop.

Inside of our `sortable.stop()` function, we were identifying where in the list the swatch had been dragged and dropped to. Once we determine where the swatch is located, we calculate a new rank for that swatch using the `Rankings` helper functions. We then immediately update the `swatches` collection with the new rank, which propagates and makes the position of the UI change permanent.

There's more...

The key takeaway here is that while jQuery (or any other third-party library) may be used to make direct DOM manipulations, these manipulations won't persist beyond a single user's session, or even beyond the next DDP change from the server.

To make the manipulations permanent and to fully utilize the Blaze rendering engine's fantastic reactive programming model, we need to modify the datasource (in this case, the `swatches` collection). The modification is handled immediately, and with no effort on our part, through reactivity.

So, to review, third-party libraries can go right ahead and manipulate DOM elements in a Meteor application, with the third-party code being executed inside a `Template.rendered()` function block.

To make the changes "stick", we simply update the corresponding Mongo collection as well. Using this technique, we can integrate nearly every JavaScript library out there (if someone hasn't already done it for us on `https://atmospherejs.com/`).

See also

- The *Setting up your project file structure* recipe in *Chapter 1*, *Optimizing Your Workflow*
- The *Using inline data to modify UI elements reactively* recipe in this chapter

7
Using Client Methods

In this chapter, you will learn the following topics:

- Creating dynamic graphs with SVG and Ajax
- Using the HTML FileReader to upload images
- Creating a coloring book with the Canvas element

Introduction

Web programming, on the client side, has come a long way since the inception of HTML. Up until recently, it was necessary to write/use third party **polyfill** libraries to do simple things, such as build graphs, read files, and draw on the screen. But that cute lil' bucktoothed kid we all knew and disliked has grown up! With powerful, easy to use scripting and graphics APIs and objects now supported in nearly every browser, web programming is finally ready to be used on its own. The recipes in this chapter will go through some of the most popular Web API objects, and show you how to implement them inside Meteor.

Creating dynamic graphs with SVG and Ajax

While there are a lot of graphing libraries out there (all of which you can use in Meteor, by the way!) it's good to know how to implement fundamental shapes inside an HTML page, using **SVG templates**. Sometimes implementing a full-fledged graphing library is overkill, and having a working knowledge of SVG in your back pocket can be very useful. However, this is a Meteor recipe; we not only want to see an example of SVG, but we want to see it in action! This recipe will teach you how to stream a live data source into MongoDB collections, and then represent that dynamic collection graphically, using SVG tags.

Getting ready

We will be using a default project installation, with `client`, `server`, and `both` folders, to keep the code clean and readable. In a terminal window, navigate to where you would like your project to reside, and execute the following commands:

```
$ meteor create svggraph
$ cd svggraph
$ rm svggraph.*
$ mkdir client
$ mkdir server
$ mkdir both
```

We also want to make use of a few packages, and remove the `autopublish` package, so let's do that now. Run the following commands in the terminal window:

```
$ meteor remove autopublish
$ meteor add http
$ meteor add meteorhacks:aggregate
$ meteor add rzymek:randomcolor
$ meteor
```

We're now ready to start creating our SVG streaming data graph.

How to do it...

Our project is going to read the comments stream from `http://www.meetup.com/`, which is freely available, and very easy to use. We will record the comments from the stream, and display a sum total of comments by state. We will need to create the interface, which will display state totals as a vertical bar chart, and will also show us the last comment made, along with a picture of the Meetup Group it came from.

Note

We are using the data stream collecting code for demonstration purposes only! There is no redundancy checking or hardening built into it. We strongly recommend against using this code as-is in anything even resembling a production application.

Proceed with the following steps to create dynamic graphs with SVG and Ajax:

1. We will first create the SVG element with the `id` of `stateBars`. We will then create a `<rect>` element for each individual state, and adjust the height using the `stateStat` template. Inside the `[project root]/client/` folder, create a `client.html` file, add the following HTML code, and save the file:

```
<body>
  {{> cPic}}
</body>

<template name="cPic">
  <svg id="stateBars" width="800" height="600">
    {{#each stateStats}}
    {{> stateStat}}
    {{/each}}
  </svg>
  <div id="cComment">{{curComment}}</div>
  <img id="cPic" src="{{curPic}}"/>
</template>


<template name="stateStat">
  <rect width="{{width}}" height="{{stackHeight}}"
y="{{stackPosition}}"
        style="fill:{{color}};fill-opacity:0.8;" />
  <text x="5" y="{{textYPos}}" fill="black">{{state}}: {{total}}
</text>
</template>
```

2. We want to provide some basic styling on our page, so let's get that out of the way. In the `[project root]/client` folder, create a `style.css` file, and add the following style declarations:

```
body {
  font-family: 'Helvetica-Neue', sans-serif;
  font-size: 12px;
}
#stateBars {
  border:dashed 3px #ccc;
}
#cComment {
  margin-top: 10px;
}
```

```
#cPic {
  width: 200px;
  margin-top: 10px;
}
```

3. Next, let's set up the `Comments` and `CountryTotals` collections. To keep things cleaner, logic-wise, on the client we will use some server-side logic to aggregate the incoming comments. We first need to declare our collections for use on both the client and the server. In the `[project root]/both/` folder, create a new file called `model.js`, and add the following two declarations to the file:

```
Comments = new Mongo.Collection('comments');
CountryTotals = new Mongo.Collection('countryTotals');
```

4. Next, let's set up monitoring for the `commentsStream` data, and add a code to count the totals. As we removed the `autopublish` package, we need to declare our `Comments` publication, and at the same time, we want to update the totals for a state whenever any new comments come in. We can accomplish this by using the `Meteor.publish()` function and using the `cursor.observeChanges()` method. In your `[project root]/server/` folder, create a new file called `svggraph-server.js`, and add the following method:

```
Meteor.publish("commentsStream", function(country){
  var cursor = Comments.find({country:country});
  var initializing = true;
  cursor.observeChanges({
    added:function(id,doc){
      if (initializing) return;
      var cTots = Meteor.call('totalsByCountry', doc.country);
      var sTots = Meteor.call('totalsByState',doc.country);
      var existingTots = CountryTotals.findOne({country:doc.
country});

      if (!sTots || !cTots) return;
      sTots = _.map(sTots,function(s,i,d){
        s._id.total = s.total;
        if (existingTots){
          var existingState = _.findWhere(existingTots.
states,{state:s._id.state});
          if (existingState) s._id.color = existingState.color ||
randomColor({luminosity: 'light',
hue: 'blue'});
          else s._id.color = randomColor({luminosity:
'light', hue: 'blue'});
        }
        return s._id;
      });
```

```
      var cObj = {country:doc.country, total:cTots[0].total,
states: sTots};
      CountryTotals.upsert({country:cObj.country},cObj);
    }
  });
  initializing = false;

  return cursor;
});
```

5. We also need to declare the `Meteor.publish()` method for our state totals by country, so let's do that as well. Append the following to the end of the `svggraph-server.js` file:

```
Meteor.publish("graphData", function(country){
  return CountryTotals.find({country:country});
});
```

6. We need to keep track of the country that we are monitoring, and what the last comment and picture from the Meetup Group were. The most straightforward way to do this is through `Session` variables, so let's create a file called `svggraph-client.js` in the `[project root]/client/` folder, and add the following three variables at the top:

```
Session.setDefault("country", "us");
Session.setDefault("msgComment","No comments yet");
Session.setDefault("msgPic",
"https://d14jjfgstdxsoz.cloudfront.net/meteor-logo.png");
```

7. Now that we have specified the country that we will be monitoring, we can add our `Meteor.subscribe` statements, which both take the country as a parameter. In the `svggraph-client.js` file, add the following statements inside a `Tracker.autorun` block:

```
Tracker.autorun(function(){
  Meteor.subscribe("graphData", Session.get('country'));
  Meteor.subscribe("commentsStream",
  Session.get('country'));
});
```

8. We will hook up the data later, but we still need to add the `Template.helpers` method for displaying the latest pictures and comments, based on the `Session` variables declared in the preceding steps. In the `svggraph-client.js` file, add the following code just below the `Tracker.autorun` block:

```
Template.cPic.helpers({
  curPic: function () {
    return Session.get('msgPic');
  },
```

```
  curComment: function() {
    return Session.get('msgComment');
  }
});
```

9. In anticipation of incoming data from the client, we need a few `helper` methods, to message data and perform *aggregate queries* on our Mongo DB collections. We also want a quick way to reset our collections, because the sheer amount of data we're collecting can get messy for a sample application. As such, we will create the following server-side methods:
   - `addMsg`: This is used for inserting a message into our `Comments` collection.
   - `totalsByState`: This is used to aggregate the total number of `Comments`, by state.
   - `totalsByCountry`: This is used to aggregate totals, by country.
   - `resetDB`: This is used to reset the `Comments` and `CountryTotals` collections.

   Open `svggraph-server.js` and add the following code:

```
Meteor.methods({
  addMsg : function (msg) {
    var upMsg = {};
    try {
    upMsg.country = msg.group.country;
    upMsg.state = msg.group.state;
    upMsg.category = msg.group.category.name;
    upMsg.thumb = (msg.group.group_photo ?
    msg.group.group_photo.thumb_link: "");
    upMsg.createdAt = Date.now();
    }
    catch(e){
      console.log(e.message);
      return null;
    }
    Comments.insert(upMsg);
  },
  totalsByState: function (country){
    return Comments.aggregate([
      {$match:{country:country}},
      {$group:{_id:{state:"$state"},total:{$sum:1}}},
      {$sort:{"total":-1}}
    ]);
  },
  totalsByCountry: function(country){
```

```
    return Comments.aggregate([
      {$match:{country:country}},
      {$group:{_id:{},total:{$sum:1}}}
    ]);
  },
  resetDB: function(){
    Comments.remove({});
    CountryTotals.remove({});
    console.log('Collections have been reset');
  }
});
```

10. With our server-side helper logic all in place, it's time to read and parse the data stream. In the `svggraph-client.js` file, add the following `connect`, `disconnect`, `onopen`, `onclose`, and `onmessage` websocket functions:

```
function MeetupsStream() {
  var ms = {};
  var ws;
  var sURL = "ws://stream.meetup.com/2/event_comments";
  ms.connect = function (url) {
    sURL = url || sURL;
    ws = new WebSocket(sURL);
    ws.onopen = ms.onopen;
    ws.onmessage = ms.onmessage;
    ws.onclose = ms.onclose;
    return ms;
  };
  ms.disconnect = function () {
    ws && ws.close();
  };
  ms.onopen = function () {
    console.log("Meetup stream started...");
  };
  ms.onmessage = function (e) {
    var rec_msg = EJSON.parse(e.data);
    if (rec_msg.group.group_photo)
      Session.set('msgPic',
      rec_msg.group.group_photo.photo_link);
    Session.set('msgComment', rec_msg.comment);
    Meteor.call('addMsg', rec_msg);
//
  };

  ms.onclose = function () {
```

```
    console.log("Meetup stream closed.");
  };

  return ms;
}
```

11. With the data stream all ready to go, all that's left is to add our display logic, and flip the switch. In the `svggraph-client.js` file, locate the `Template.cPic.helpers` code block, and add the following helper function just below the `curComment` event (don't forget the comma!):

```
curComment: function() {
    return Session.get('msgComment');
  },
  stateStats: function () {
    var ct = CountryTotals.findOne(
      {country:Session.get('country')});
    if (!ct) return [];
    var stateTotals = ct.states;
    var ctotal = ct.total;
    var SVGWidth = 800;
    var SVGHeight = 600;
    return _.map(stateTotals, function(s,i,l){
      var retObj = {};
      retObj.state = s.state;
      retObj.index = i;
      retObj.total = s.total;
      retObj.width = ~~(SVGWidth * (s.total/ctotal));
      retObj.stackHeight = ~~(SVGHeight/l.length);
      retObj.stackPosition = i*retObj.stackHeight;
      retObj.color = s.color;
      return retObj;
    });
  }
});
```

12. We need a few style helpers to properly position our text data, and to let us know where the most recent comment was added (to make the things all fancy!). Append the following `Template.helpers` code block to the bottom of the `svggraph-client.js` file:

```
Template.stateStat.helpers({
  textYPos: function () {
    return this.stackPosition + ~~(this.stackHeight/2);

  },
  textXPos : function(){
    return this.width - ~~(this.width*.2);
  },
  color : function(){
    if (Session.equals('lastState',this.state))
    return '#2ecc71';
    return this.color;
  }
});
```

13. Using the `Template.rendered` method block, we will turn our data stream on and update our `lastState` session variable whenever a new comment is posted, using a simple `autorun` method. Enter the following code at the bottom of your `svggraph-client.js` file:

```
Template.cPic.rendered = function(){
  MStream = new MeetupsStream();
  MStream.connect();
  this.autorun(function(){
    var last = Comments.findOne(
      {country:Session.get('country')},
      {sort:{createdAt:-1}});
    if (last) Session.set('lastState',last.state);
  });
}
```

14. Save all changes, and open a browser (if you haven't already) to `http://localhost:3000`. Wait just a bit, and you should see the new comments coming in, with state totals being updated, as shown in the following screenshot:

15. Let it run for a bit, so that more states appear. If you let it run long enough, you'll eventually get all 50 states (and maybe even Puerto Rico!). If you would like to reset your collection, open your browser console and enter the following command:

```
> Meteor.call('resetDB')
```

How it works...

As mentioned in the recipe introduction, there are two critical pieces to graphing dynamic data. The first has to do with rendering SVG objects, and can be found mostly in the HTML template code, in the `[project root]/client/client.html` file. We first used the `{{> cPic}}` template inclusion to reference our `cPic` template. The template itself declares our `<svg>` element, then runs a standard `{{#each }}` control structure on the `stateStats` collection:

```
<template name="cPic">
  <svg ...>
    {{#each stateStats}}
    {{> stateStat}}
    {{/each}}
  </svg>
  ...
</template>
```

We set the width and height of the `<svg>` element directly (note that they are *not* CSS style attributes, but are actual element attributes instead), and add a unique `id` attribute for CSS styling as well, as shown in the following example:

```
<svg id="stateBars" width="800" height="600">
```

In the `stateStat` template, we create a `<rect>` element and set pretty much every attribute dynamically:

- `width`: This is set based on the number of comments.
- `height`: This is set based on the total number of states.
- `y`: This is the vertical position and is set based on position in the data collection.
- `fill`: This is the color value which is set to green if it was the last state updated.

We likewise set the position of the text SVG element to make sure it lines up with the corresponding `<rect>` element:

```
<template name="stateStat">
  <rect width="{{width}}" height="{{stackHeight}}"
y="{{stackPosition}}"
        style="fill:{{color}};fill-opacity:0.8;" />
  <text x="5" y="{{textYPos}}" fill="black">{{state}}: {{total}}
</text>
</template>
```

It's important to understand that SVG elements are not positioned relative to other elements inside the same `<svg>` block. This is why we had to set the position of each element. Also, some attributes (`width` and `height` for example) can be set directly. SVG elements are truly a hybrid between a bonafide DOM element and a graphic element.

> For a great introduction to SVG, you can read the MDN SVG tutorial, found at `https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial`.

The second critical piece surrounds the use of the `WebSocket` HTML5 Web API object, found in our `MeetupsStream` function of the `svggraph-client.js` file. In this function, we prepare a pretty standard Ajax (actually `WebSocket`) call by setting handlers for `.ondisconnect`, `.onopen`, and `.onmessage`:

```
function MeetupsStream() {
  ms.onopen = ...
  ms.onmessage = ...
  ms.onclose = ...
}
```

The one we are most interested in is the `.onmessage` handler, where we parse the data (which comes in as a string) into an `EJSON` object:

```
var rec_msg = EJSON.parse(e.data);
```

We then (conditionally) set the `msgPic` variable, which immediately changes the image displayed in our `cPic` template. We likewise update `msgComment` to change the displayed comment:

```
if (rec_msg.group.group_photo)
      Session.set('msgPic', ...);
    Session.set('msgComment', rec_msg.comment);
```

Finally, we make a `Meteor.call` method to the server-side `'addMsg'` method, which puts things in motion to update our state totals:

```
Meteor.call('addMsg', rec_msg);
```

> The `WebSocket` object also has a good MDN tutorial, found at `https://developer.mozilla.org/en-US/docs/WebSockets`.

A few more odds and ends can be found in the code we used, such as the `meteorhacks:aggregate` package, which lets us do summations and groupings on the MongoDB collections (official support coming soon!) via the `.aggregate` method, but this recipe was meant to focus on the SVG and Ajax (`WebSocket`) aspects.

See also

- The *Creating dynamic lists* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Handling asynchronous events* recipe in *Chapter 11*, *Leveraging Advanced Features*

Using the HTML FileReader to upload images

Adding files via a web application is a pretty standard functionality nowadays. That doesn't mean that it's easy to do, programmatically. New browsers support Web APIs to make our job easier, and a lot of quality libraries/packages exist to help us navigate the file reading/uploading forests, but, being the coding lumberjacks that we are, we like to know how to roll our own! In this recipe, you will learn how to read and upload image files to a Meteor server.

Getting ready

We will be using a default project installation, with `client`, `server`, and `both` folders, and with the addition of a special folder for storing images. In a terminal window, navigate to where you would like your project to reside, and execute the following commands:

```
$ meteor create imageupload
$ cd imageupload
$ rm imageupload.*
$ mkdir client
$ mkdir server
$ mkdir both
$ mkdir .images
```

Note the dot in the `.images` folder. This is really important because we don't want the Meteor application to automatically refresh every time we add an image to the server! By creating the images folder as `.images`, we are hiding it from the eye-of-Sauron-like monitoring system built into Meteor, because folders starting with a period are "invisible" to Linux or Unix.

Let's also take care of the additional Atmosphere packages we'll need. In the same terminal window, execute the following commands:

```
$ meteor add twbs:bootstrap
$ meteor add voodoohop:masonrify
```

We're now ready to get started on building our image upload application.

How to do it...

We want to display the images we upload, so we'll be using a layout package (`voodoohop:masonrify`) for display purposes. We will also initiate uploads via drag and drop, to cut down on UI components. Lastly, we'll be relying on an npm module to make the file upload much easier. Let's break this down into a few steps, starting with the user interface.

1. In the `[project root]/client` folder, create a file called `imageupload.html` and add the following templates and `template` inclusions:

```
<body>
  <h1>Images!</h1>
  {{> display}}
  {{> dropzone}}
</body>

<template name="display">
  {{#masonryContainer
    columnWidth=50
    transitionDuration="0.2s"
    id="MasonryContainer"
  }}
  {{#each imgs}}
  {{> img}}
  {{/each}}
  {{/masonryContainer}}
</template>

<template name="dropzone">
  <div id="dropzone" class="{{dropcloth}}">Drag images here...
</div>
</template>

<template name="img">
  {{#masonryElement "MasonryContainer"}}
  <img src="{{src}}"
    class="display-image"
    style="width:{{calcWidth}}"/>
  {{/masonryElement}}
</template>
```

2. We want to add just a little bit of styling, including an "active" state for our drop zone, so that we know when we are safe to drop files onto the page. In your `[project root]/client/` folder, create a new `style.css` file and enter the following CSS style directives:

```
body {
  background-color: #f5f0e5;
  font-size: 2rem;

}

div#dropzone {
  position: fixed;
  bottom:5px;
  left:2%;
  width:96%;
  height:100px;
  margin: auto auto;
  line-height: 100px;
  text-align: center;
  border: 3px dashed #7f898d;
  color: #7f8c8d;
  background-color: rgba(210,200,200,0.5);
}

div#dropzone.active {
  border-color: #27ae60;
  color: #27ae60;
  background-color: rgba(39, 174, 96,0.3);
}

img.display-image {
  max-width: 400px;
}
```

3. We now want to create an `Images` collection to store references to our uploaded image files. To do this, we will be relying on **EJSON**. EJSON is Meteor's extended version of JSON, which allows us to quickly transfer binary files from the client to the server. In your `[project root]/both/` folder, create a file called `imgFile.js` and add the MongoDB collection by adding the following line:

```
Images = new Mongo.Collection('images');
```

4. We will now create the `imgFile` object, and declare an EJSON type of `imgFile` to be used on both the client and the server. After the preceding `Images` declaration, enter the following code:

```
imgFile = function (d) {
  d = d || {};
  this.name = d.name;
  this.type = d.type;
  this.source = d.source;
  this.size = d.size;
};
```

5. To properly initialize `imgFile` as an EJSON type, we need to implement the `fromJSONValue()`, `prototype()`, and `toJSONValue()` methods. We will then declare `imgFile` as an EJSON type using the `EJSON.addType()` method. Add the following code just below the `imgFile` function declaration:

```
imgFile.fromJSONValue = function (d) {
  return new imgFile({
    name: d.name,
    type: d.type,
    source: EJSON.fromJSONValue(d.source),
    size: d.size
  });
};

imgFile.prototype = {
  constructor: imgFile,

  typeName: function () {
    return 'imgFile'
  },
  equals: function (comp) {
    return (this.name == comp.name &&
      this.size == comp.size);
  },
  clone: function () {
    return new imgFile({
      name: this.name,
      type: this.type,
      source: this.source,
      size: this.size
    });
  },
  toJSONValue: function () {
    return {
```

```
      name: this.name,
      type: this.type,
      source: EJSON.toJSONValue(this.source),
      size: this.size
    };
  }
};

EJSON.addType('imgFile', imgFile.fromJSONValue);
```

The EJSON code used in this recipe is heavily inspired by Chris Mather's Evented Mind file upload tutorials. We recommend checking out his site and learning even more about file uploading at `https://www.eventedmind.com`.

6. Even though it's usually cleaner to put client-specific and server-specific code in separate files, because the code is related to the `imgFile` code we just entered, we are going to put it all in the same file. Just below the `EJSON.addType()` function call in the preceding step, add the following `Meteor.isClient` and `Meteor.isServer` code:

```
if (Meteor.isClient){
  _.extend(imgFile.prototype, {
    read: function (f, callback) {

      var fReader = new FileReader;
      var self = this;
      callback = callback || function () {};

      fReader.onload = function() {
        self.source = new Uint8Array(fReader.result);
        callback(null,self);
      };

      fReader.onerror = function() {
        callback(fReader.error);
      };

      fReader.readAsArrayBuffer(f);
    }
  });

  _.extend (imgFile, {
    read: function (f, callback){
```

```
      return new imgFile(f).read(f,callback);
    }
  });
};

if (Meteor.isServer){
  var fs = Npm.require('fs');
  var path = Npm.require('path');
  _.extend(imgFile.prototype, {
    save: function(dirPath, options){
      var fPath = path.join(process.env.PWD,dirPath,this.name);
      var imgBuffer = new Buffer(this.source);
      fs.writeFileSync(fPath, imgBuffer, options);
    }
  });
};
```

7. Next, we will add some `Images` collection `insert` helpers. We will provide the ability to add either references (URIs) to images, or to upload files into our `.images` folder on the server. To do this, we need some `Meteor.methods`. In the `[project root]/server/` folder, create an `imageupload-server.js` file, and enter the following code:

```
Meteor.methods({
  addURL : function(uri){
    Images.insert({src:uri});
  },
  uploadIMG : function(iFile){
    iFile.save('.images',{});
    Images.insert({src:'images/'
    +iFile.name});
  }
});
```

8. We now need to establish the code to process/serve images from the `.images` folder. We need to circumvent Meteor's normal asset serving capabilities for anything found in the (hidden) `.images` folder. To do this, we will use the `fs` npm module, and redirect any content requests accessing the `Images/` folder address to the actual `.images` folder found on the server. Just after the `Meteor.methods` block entered in the preceding step, add the following `WebApp.connectHandlers.use()` function code:

```
var fs = Npm.require('fs');
WebApp.connectHandlers.use(function(req, res, next) {
  var re = /^\/images\/(.*)$/.exec(req.url);
  if (re !== null) {
```

```
    var filePath = process.env.PWD
    + '/.images/'+ re[1];
    var data = fs.readFileSync(filePath, data);
    res.writeHead(200, {
      'Content-Type': 'image'
    });
    res.write(data);
    res.end();
  } else {
    next();
  }
});
```

9. Our images `display` template is entirely dependent on the `Images` collection, so we need to add the appropriate reactive `Template.helpers` function on the client side. In your `[project root]/client/` folder, create an `imageupload-client.js` file, and add the following code:

```
Template.display.helpers({
  imgs: function () {
    return Images.find();
  }
});
```

10. If we add pictures we don't like and want to remove them quickly, the easiest way to do that is by double clicking on a picture. So, let's add the code for doing that just below the `Template.helpers` method in the same file:

```
Template.display.events({
  'dblclick .display-image': function (e) {
    Images.remove({
      _id: this._id
    });
  }
});
```

11. Now for the fun stuff. We're going to add drag and drop visual feedback cues, so that whenever we drag anything over our drop zone, the drop zone will provide visual feedback to the user. Likewise, once we move away from the zone, or actually drop items, the drop zone should return to normal. We will accomplish this through a `Session` variable, which modifies the CSS class in the `div.dropzone` element, whenever it is changed. At the bottom of the `imageupload-client.js` file, add the following `Template.helpers` and `Template.events` code blocks:

```
Template.dropzone.helpers({
  dropcloth: function () {
    return Session.get('dropcloth');
  }
```

```
});

Template.dropzone.events({
  'dragover #dropzone': function (e) {
    e.preventDefault();
    Session.set('dropcloth', 'active');
  },
  'dragleave #dropzone': function (e) {
    e.preventDefault();
    Session.set('dropcloth');

  }
});
```

12. The last task is to evaluate what has been dropped in to our page drop zone. If what's been dropped is simply a URI, we will add it to the `Images` collection as is. If it's a file, we will store it, create a URI to it, and then append it to the `Images` collection. In the `imageupload-client.js` file, just before the final closing curly bracket inside the `Template.dropzone.events` code block, add the following event handler logic:

```
  'dragleave #dropzone': function (e) {
    ...
  },
  'drop #dropzone': function (e) {
    e.preventDefault();
    Session.set('dropcloth');

    var files = e.originalEvent.dataTransfer.files;
    var images =
$(e.originalEvent.dataTransfer.getData('text/html')).find('img');
    var fragment = _.findWhere(e.originalEvent.dataTransfer.items,
{
      type: 'text/html'
    });
    if (files.length) {
      _.each(files, function (e, i, l) {
        imgFile.read(e, function (error, imgfile) {
          Meteor.call('uploadIMG', imgfile, function (e) {
            if (e) {
              console.log(e.message);
            }
          });
        })
      });
    } else if (images.length) {
```

```
        _.each(images, function (e, i, l) {
          Meteor.call('addURL', $(e).attr('src'));
        });
      } else if (fragment) {
        fragment.getAsString(function (e) {
          var frags = $(e);
          var img = _.find(frags, function (e) {
            return e.hasAttribute('src');
          });
          if (img) Meteor.call('addURL', img.src);
        });

      }

    }
  });
```

13. Save all your changes and open a browser to `http://localhost:3000`. Find some pictures from any web site, and drag and drop them in to the drop zone. As you drag and drop the images, the images will appear immediately on your web page, as shown in the following screenshot:

As you drag and drop the dinosaur images in to the drop zone, they will be uploaded as shown in the following screenshot:

Similarly, dragging and dropping actual files will just as quickly upload and then display images, as shown in the following screenshot:

14. As the files are dropped, they are uploaded and saved in the `.images/` folder:

How it works...

There are a lot of moving parts to the code we just created, but we can refine it down to four areas.

First, we created a new `imgFile` object, complete with the internal functions added via the `Object.prototype = {...}` declaration. The functions added here (`typeName`, `equals`, `clone`, `toJSONValue` and `fromJSONValue`) are primarily used to allow the `imgFile` object to be *serialized* and *deserialized* properly on the client and the server. Normally, this isn't needed, as we can just `insert` into Mongo Collections directly, but in this case it is needed because we want to use the `FileReader` and Node `fs` packages on the client and server respectively to directly load and save image files, rather than write them to a collection.

Second, the underscore `_.extend()` method is used on the client side to create the `read()` function, and on the server side to create the `save()` function. `read` takes the file(s) that were dropped, reads the file into an `ArrayBuffer`, and then calls the included `callback`, which uploads the file to the server. The `save` function on the server side reads the `ArrayBuffer`, and writes the subsequent image file to a specified location on the server (in our case, the `.images` folder).

Third, we created an `ondropped` event handler, using the `'drop #dropzone'` event. This handler determines whether an actual file was dragged and dropped, or if it was simply an HTML `<img>` element, which contains a URI link in the `src` property. In the case of a file (determined by `files.length`), we call the `imgFile.read` command, and pass a callback with an immediate `Meteor.call('uploadIMG'...)` method. In the case of an `<img>` tag, we parse the URI from the `src` attribute, and use `Meteor.call('addURL')` to update the `Images` collection.

Fourth, we have our `helper` functions for updating the UI. These include `Template.helpers` functions, `Template.events` functions, and the `WebApp.connectedHandlers.use()` function, used to properly serve uploaded images without having to update the UI each time a file is uploaded. Remember, Meteor will update the UI automatically on any file change. This unfortunately includes static files, such as images. To work around this, we store our images in a file invisible to Meteor (using `.images`). To redirect the traffic to that hidden folder, we implement the `.use()` method to listen for any traffic meant to hit the `'/images/'` folder, and redirect it accordingly.

As with any complex recipe, there are other parts to the code, but this should cover the major aspects of file uploading (the four areas mentioned in the preceding section).

There's more...

The next logical step is to not simply copy the URIs from remote image files, but rather to download, save, and serve local copies of those remote images. This can also be done using the `FileReader` and Node `fs` libraries, and can be done either through the existing client code mentioned in the preceding section, or directly on the server, as a type of *cron* job.

> For more information on `FileReader`, please see the MDN FileReader article, located at `https://developer.mozilla.org/en-US/docs/Web/API/FileReader`.

See also

- The *Using npm packages directly* recipe in *Chapter 8*, *Intergrating Third-party Libraries*
- The *Creating custom EJSON objects* recipe in *Chapter 11*, *Leveraging Advanced Features*

Creating a coloring book with the Canvas element

There are now multiple ways to display graphics on a web page. DOM elements, SVG, WebGL, and, perhaps the most user-friendly, the `Canvas` element. Using JavaScript, the `Canvas` element provides a flexible graphics area where you can paint, erase, color, cut and paste to your heart's content. A good representation of what you can do with the `Canvas` element is found in this recipe, where you will learn how to build a coloring book app from scratch.

Getting ready

Get your crayons! Okay, maybe those won't be needed (don't use them on your monitor!) but what we're about to create is the next best thing, so let's get the app structure and packages out of the way, so we can start coloring!

In addition to the usual suspects, we will need some specialty folders to make the grouping of our code cleaner and more manageable. Open a terminal window, navigate to where you would like to create your root project, and execute the following commands:

```
$ meteor create coloringbook
$ cd coloringbook
$ rm coloringbook.*
$ mkdir -p client/scripts
$ mkdir {both,server,public}
$ meteor
```

How to do it...

It may not seem like it on the surface, but there's actually a lot to developing a coloring book application. You have to worry about user preferences, undo/redo, picking colors, erasing, and so on. We will tackle these steps one at a time, adding packages and functionality as we go along.

1. First, we will add user management packages. Meteor has some fantastic out-of-the-box functionality for user account management, and we will take advantage of that functionality. In a terminal window, navigate to your `[project root]` folder and add the following two Meteor libraries by entering the following commands:

   ```
   $ meteor add accounts-password
   $ meteor add accounts-ui
   ```

2. We will want to display these on the screen, so let's add the necessary UI elements and style them appropriately. Create a file called `cb-ui.html` in your `[project root]/client/` folder, and add the following code:

```
<body>
  {{> loginButtons}}
</body>
```

Your page will look very spartan at the moment, but just like that, we already have accounts and account creation! In a browser, navigate to `http://localhost:3000` and you should see the accounts dialog dropdown at the top left of your screen, similar to the following screenshot:

3. Let's finish up the visual aspects of our UI by adding a toolbar, where we can see color swatches and adjust the size of our brush; and let's also add our `canvas` element, with a background `<div>` to display the picture we want to color. In the same `cb-ui.html` file, add the following template inclusions in the `body` element block:

```
<body>
  {{> loginButtons}}
  {{> toolbar}}
  <div id="bgpicture"></div>
  {{> picture}}
</body>
```

4. The `picture` template is actually the easiest, so we'll add the code for that first. Add the following template code just below the `</body>` tag:

```
<template name="picture">
  <canvas id="picture" class="no-copy">Images go here...</canvas>
</template>
```

5. Now we add the `toolbar` and the `swatch` templates, which use a bit of SVG and `Masonry.js` to display our brush and color swatches. Just below the template code added in the preceding step, add the following code:

```
<template name="toolbar">
```

```
  <div class="brush-size" id="brush-size">
    <svg id="brush-preview"
     height="70" width="70"
     style="display:{{eraseMode}}">
      <circle cx="35" cy="35"
      r="{{preview.size}}"
      fill="{{preview.color}}" />
    </svg>
  </div>
  {{#masonryContainer columnWidth=50
    transitionDuration="0.2s"
    id="MasonryContainer" }}

     {{#each swatches}}
      {{> swatch}}
     {{/each}}

  {{/masonryContainer}}
</template>
<template name="swatch">
  {{#masonryElement "MasonryContainer"}}
    <div class="swatch"
   style="background-color:{{color}};">
{{color}}</div>
  {{/masonryElement}}
</template>
```

6. To get everything all pretty, we need to add some Atmosphere packages, and just a bit of CSS styling. In a terminal window, navigate to your `[project root]` folder and run the following commands:

   ```
   $ meteor add twbs:bootstrap
   $ meteor add voodoohop:masonrify
   ```

7. Next, create a file named `cb-style.css` in the `[project root]/client/` folder, and add the following style declarations:

   ```
   #picture {
     color: #ccc;
     border: 3px dashed #ccc;
     width:800px;
     height:600px;
     border-radius: 4px;
     line-height: 3em;
     text-align: center;
     left: 100px;
   ```

```
    position: absolute;

  }

  #bgpicture {
    pointer-events:none;
    touch-events:none;
    position:absolute;
    background: url('Rawr.GIF');
    top:0px;
    left:100px;
    width: 800px;
    height:600px;
    z-index: 999;

  }
  .no-copy {
    -webkit-user-select: none;
  }
  #login-buttons {
    position : absolute;
    min-width: 220px;
    right: 20px;
    top: 10px;
  }
  .login-link-text {
    position:absolute;
    right: 0px;
  }
  .swatch {
    height:50px;
    width:50px;
    border-radius: 4px;
    border: solid #ccc 2px;
    line-height: 50px;
    font-size:0.8em;
    text-align: center;
    margin-bottom: 5px;
  }
  .masonry_container {
    position:absolute;
    top:100px;
    left:23px;
    width: 50px;
```

```
}
.brush-size {
  position:absolute;
  left:10px;
  top:10px;
  border: dashed 3px #ccc;
  border-radius: 40px;
  width:76px;
  height:76px;
}
```

8. Note that the `bgpicture` element has a background image in it, called `'RAWR.GIF'`—You can either obtain this picture from the source files for this recipe, or you can add your own image to be colored. The image *must* have a transparent background, and will look better if it is `800x600` pixels, but in any case, the image you want displayed should go into the `[project root]/public/` folder, and you should change the `background` property in the preceding step, so that it will display properly on the screen.
9. We will need to be able to change brush sizes, change colors, and paint/erase on the canvas, and we want to do that no matter what device we are on, so we will use the official `Hammer.js` package. We also want to include global shortcut keys for undo (*control* + *Z*) and redo (*shift* + *control* + *Z*), so we'll add a handy library that does that! In a terminal window, in your `[project root]` folder, enter the following two commands:

```
$ meteor add hammer:hammer@=2.0.4_1
$ meteor add gwendall:body-events
```

10. It's cleaner and easier to use `Hammer.js` events in the `Template.events` block, so we want to add the jQuery plugin for the `Hammer.js` file. As Meteor piggybacks on jQuery event handlers, if the plugin is added to jQuery, it's added to Meteor! As of this writing, the most reliable way to add the plugin is to manually copy the `plugin` script file into your `[project root]/client/scripts/` folder. You can either obtain the script by going to `http://hammerjs.github.io/jquery-plugin/` and following the instructions there, or by creating a file in your `scripts` folder called `jquery.hammer.js`, and adding the following code:

```
(function(factory) {
  if (typeof define === 'function' && define.amd) {
    define(['jquery', 'hammerjs'], factory);
  } else if (typeof exports === 'object') {
    factory(require('jquery'), require('hammerjs'));
  } else {
    factory(jQuery, Hammer);
  }
}(function($, Hammer) {
```

```
  function hammerify(el, options) {
    var $el = $(el);
    if(!$el.data("hammer")) {
      $el.data("hammer", new Hammer($el[0], options));
    }
  }

  $.fn.hammer = function(options) {
    return this.each(function() {
      hammerify(this, options);
    });
  };

  // extend the emit method to also trigger jQuery events
  Hammer.Manager.prototype.emit = (function(originalEmit) {
    return function(type, data) {
      originalEmit.call(this, type, data);
      $(this.element).trigger({
        type: type,
        gesture: data
      });
    };
  })(Hammer.Manager.prototype.emit);
}));
```

11. We will now add the `Activities`, `Prefs`, and `Swatches` collections. As mentioned previously, we will need to keep track of our undo/redo, through recording activities. We will also need to keep track of color swatches, user preferences for brush size, and so on. To do this, we will declare three MongoDB collections. In your `[project root]/both/` folder, create a file called `cb-model.js` and add the following code:

    ```
    Swatches = new Mongo.Collection('swatches');
    Prefs = new Mongo.Collection('prefs');
    Activities = new Mongo.Collection('activities');
    ```

    We'll be using these later on, on both the client and on the server, but for now we're all done. You can close the `cb-model.js` file and move on to the next step.

12. Let's add the server-side logic for the preferences and undo/redo history. As unbelievable as this may seem, the server-side code for the coloring book is very light, compared to the client code. So, we're going to get it out of the way. We need to do several things on the server:

    - Initialize swatches and other preferences
    - Allow new swatches to be added, if desired

- Track and recall our painting activity (undo/redo)
- Clear everything out, so we can start over

In your `[project root]/server/` folder, create a file called `cb-server.js` and add the following `Meteor.methods` declaration:

```
Meteor.methods({

  initSwatches : function(userId){
    // no user = nothing to do. return.
    if (!userId) return;
    // if we already have swatches, return.
    if (Swatches.findOne({user:userId})) return;
    // add initial swatches
    Swatches.insert({color: '#ecf0f1', user:userId});
    Swatches.insert({color: '#ccc', user:userId});
    Swatches.insert({color: '#f1c40f', user:userId});
    Swatches.insert({color: '#e67e22', user:userId});
    Swatches.insert({color: '#e74c3c', user:userId});
    Swatches.insert({color: '#2ecc71', user:userId});
    Swatches.insert({color: '#2980b9', user:userId});
    Swatches.insert({color: '#000', user:userId});
  },

  addSwatch : function (color){
    // no user = nothing to do. return.
    if (!this.userId) return;
    // if it doesn't already exist, add the swatch
    if (!Swatches.findOne({color:color})){
      Swatches.insert({color:color, user:this.userId});
    }
  },

  clearActivity : function(){
    // no user, return.
    if (!this.userId) return;
    // clear the undo history from Activities collection
    Activities.remove({user:this.userId});
  },

  breakHistory : function(snapShot){
    // if we don't have a valid snapshot,
    // or user isn't logged in, return.
    if (!snapShot||!this.userId) return;
    // remove all snapshots after this one in the undo chain
```

```
    Activities.remove({$and: [{createdAt:{$gt:snapShot.
createdAt}},{user:this.userId}]})
  }
});
```

13. Now add the `Accounts.onLogin()` event handler in the same file:

```
Accounts.onLogin(function(login){
  // first, confirm that we have a valid userId
  userId = login.user._id;
  if (!userId) return;
  // if so, and if we don't have preferences, let's initialize
  if (!Prefs.findOne({user:userId})){
    Prefs.insert({user:userId, size:11, color:'#e74c3c'});
  }
  // likewise, let's initialize swatches
  Meteor.call('initSwatches', userId);
});
```

14. It's time to build the main logic that allows us to draw on the screen, using the `canvas` element. Create a file called `cb-client.js` in the `[project root]/client/` folder and add the following initial default values that control what we are drawing, and what our paintbrush color should be:

```
Session.setDefault('drawing', false);
Session.setDefault('color', '#e74c3c');
```

15. Because it's used in multiple places, we want to create a `drawLine` function that will draw/erase parts of the screen as needed. Add the following code just below the `Session.setDefault` declarations:

```
// **drawLine** -- helper function to draw / erase lines
drawLine = function (from, to, color,size) {
  if (size)
    ctx.lineWidth = size;
  if (color)
    ctx.strokeStyle = color;
  if (Session.get('erase')){
    ctx.globalCompositeOperation = 'destination-out';
  } else {
    ctx.globalCompositeOperation = 'source-over';
  }
  ctx.beginPath();
  ctx.moveTo(from.x, from.y);
  ctx.lineTo(to.x, to.y);
  ctx.closePath();
  ctx.stroke();
}
```

16. Now, add some helper functions that help us clean up the screen, or calculate the paint stroke positioning, relative to the page:

```
// **getPosition** -- helper function to calculate cursor position
getPosition = function (event) {
  return {
    x: parseInt(event.gesture.center.x - event.currentTarget.
offsetLeft),
    y: parseInt(event.gesture.center.y - event.currentTarget.
offsetTop)
  };
}

// **wipe** -- function to clear the painting area
wipe = function (emitAlso) {
  ctx.clearRect(0, 0, canvas.attr('width'),
canvas.attr('height'));
}
```

Whenever the user lifts their mouse/stylus/finger off of the screen, we want to record what they just drew, so let's add a snapshot-taking function just below our helpers:

```
// **addSnapshot** -- helper function to save strokes and update
// undo history
addSnapshot = function(){
  var userId = Meteor.userId();

  if (!userId) return;

  //Convert Canvas into a Picture
  ctx.globalCompositeOperation = 'source-over';
  var canvasPic = canvas[0].toDataURL();
  var timestamp = Date.now();

  //  check current history. if we are in undo-land, need to clean
  // up snapshots
  var curHist = Session.get('history');
  if (curHist){
    var curSnap = Session.get('currentSnapshot');
    Meteor.call('breakHistory',curSnap);
    Session.set('history',0);
  }

  // Save it to our Activities History
  Activities.insert({
    user:userId,
    canvas:canvasPic,
```

```
    createdAt:timestamp
  });
};
```

17. When we first log in, or when we go up and down the undo/redo history chain, we need to repaint the screen. Add the following helper function just below the `addSnapshot` helper:

```
// **paintActivity** -- helper function to redraw screen on undo/
// redo/draw
paintActivity = function(userId,idx){
  var latestActs = Activities.find({user:userId},
{sort:{createdAt:-1}}).fetch();
  if (!latestActs.length) {
    return;
  }
  if(!latestActs[idx]) idx = latestActs.length-1;
  wipe();
  var imageObj = new Image();
  imageObj.onload = function () {
    ctx.drawImage(this, 0, 0);
  };
  Session.set('currentSnapshot',latestActs[idx]);
  imageObj.src = latestActs[idx].canvas;
};
```

18. Okay, we're ready for the final touches, which include enabling our toolbar, and hooking up all the touch/mouse events to adjust brush sizes, change colors, paint and erase. In your `[project root]/client/` folder, create a file named `tmpl-toolbar.js`, and add the following interaction event handler code:

```
Template.toolbar.rendered = function(){
  // we first need to turn on hammer.js touch events...
  var brushSize = this.$('#brush-size').hammer();
  // ...we then change the pan threshold from 10 to 2
  var mgr = brushSize.data('hammer');
  mgr.get('pan').set({ threshold: 2 });
};
```

19. In that same file, add the `Template.helpers` function for displaying brush size, color preference, and so on:

```
Template.toolbar.helpers({
  swatches: function () {
    // Return the swatches for this user
    return Swatches.find({ user: Meteor.userId() });
  },
  preview : function(){
```

```
    // gets preferences for toolbar, with one modification...
    var prefs = Prefs.findOne({user:Meteor.userId()});
    // ...because brush is a circle, we need radius, not diameter
    if (prefs) prefs.size= ~~(prefs.size/2);
    return prefs;
  },
  eraseMode : function(){
    // if we're in erase mode, the brush circle is hidden
    return (Session.get('erase')? 'none':null);
  }
});
```

20. Let's also add the interactivity logic, via the `Template.events` helper block:

```
Template.toolbar.events({
  'panstart #brush-size' : function(ev){
    // record our offset position, and turn on resizing
    Session.set('brushFrom',ev.gesture.center.x);
    Session.set('brushResize',true);
  },
  'pan  #brush-size': function(ev){
    // if we're not resizing, no need to continue
    if (!Session.equals('brushResize',true)) return;
    // likewise, if there are no prefs, just return
    var prefs = Prefs.findOne({user:Meteor.userId()});
    if (!prefs) return;
    // calculate the delta from last we checked...
    var adjustment = Session.get('brushFrom');
    adjustment = ev.gesture.center.x - adjustment;
    // ...and create a new brush size
    var newbrushSize = prefs.size + adjustment;
    // reset offset position, in case resizing continues
    Session.set('brushFrom', ev.gesture.center.x);
    // new brush size needs to be the 3rd bowl of porridge...
    if (newbrushSize<=70&&newbrushSize>=3){
      // adjust the preferences record and update the collection
      prefs.size = newbrushSize;
      Prefs.update({_id:prefs._id}, prefs);
    }
  },
  'panstop #brush-size': function(ev){
    // job's done. clean up.
    Session.set('brushFrom');
    Session.set('brushResize',false);
  },
```

```
  'doubletap #brush-size': function(ev){
    // turn on 'erase' mode
    Session.set('erase',(!Session.get('erase')));
  }
});
```

21. We have a little bit of logic/event handling for the `swatch` template as well, so let's create a file called `tmpl-swatch.js` in the `[project root]/client/` folder, and add the following event listener and event interaction logic:

```
// suuuuper simple turning on of touch events
// using hammer.js
Template.swatch.rendered = function () {
  this.$('.swatch').hammer();
};

Template.swatch.events({
  'tap .swatch': function (ev) {
    // if no preference, return;
    var prefs = Prefs.findOne({user:Meteor.userId()});
    if (!prefs) return;
    // change the color to whatever swatch we tapped on
    prefs.color = this.color;
    // update Prefs collection
    Prefs.update({_id:prefs._id},prefs);
  }
});
```

22. Two more files and we're done! We have quite a bit of interaction to handle in the `picture` template (where our `canvas` element is), so let's create a file called `tmpl-picture.js` in the `[project root]/client/` folder, and add our initialization logic, via the `Template.rendered` method block. Add the following code to the `tmpl-picture.js` file:

```
Template.picture.rendered = function () {
  // set the canvas we will be drawing on
  canvas = this.$('#picture');
  // set the context for the canvas
  ctx = canvas[0].getContext('2d');
  // need to properly size the canvas
  canvas.attr({
    width: 800, height: 600
    // ...AND set up tap listeners via hammer.js
  }).hammer();
  // we want to change the default threshold from 10 to 2
  canvas.data('hammer').get('pan').set({threshold:2});
```

```
  // we now set the line and line cap style
  ctx.lineJoin = ctx.lineCap = 'round';
  // Stops iOS from doing that bouncy, janky thing
  document.ontouchmove = function (event) {
    event.preventDefault();
  };
```

23. On preference changes, or on login/logout of a user, we have to perform some reactive logic to clean up or initialize our canvas. Add two `autorun` method blocks just below `Template.picture.rendered`, as follows:

```
// Reactive function that reruns whenever
  // preference are updated
  this.autorun(function () {
    // if no prefs exist, return
    var prefs = Prefs.findOne({user:Meteor.userId()});
    if (!prefs) return;
    // set stroke color and width
    ctx.strokeStyle = prefs.color;
    ctx.lineWidth = prefs.size;
  });

  // Reactive function that reruns whenever
  // User logs in, or our undo history position changes
  this.autorun(function(){
    // if we're not logged in (no userId), return
    var userId = Meteor.userId();
    if (!userId){
      wipe();
      return;
    }
    // otherwise, paint the proper screen,
    // using the undo chain history position
    paintActivity(userId,Session.get('history')||0);
  });
};
```

24. The last item of business for the `picture` template is to hook up the drawing events themselves. When we start drawing, we need event handlers for `panstart`, `panmove`, `panend`, and we also need one to clean everything up when we `doubletap` on the page. Add the following `Template.events` method block to the bottom of `tmpl-picture.js`:

```
Template.picture.events({
  'panmove #picture': function (ev) {
    // we must be in drawing mode...
```

```
      if (Session.equals('drawing', true)
          && Meteor.userId()) {
        // find our cursor position
        to = getPosition(ev);
        // physically draw the stroke
        drawLine(from,to);
        // update our from position
        from = to;
      }
    },
    'panstart #picture': function (ev) {
      // get our from position, when we start drawing
      from = getPosition(ev);
      // tell everyone that we are in drawing mode
      Session.set('drawing', true);
    },
    'panend #picture': function (ev) {
      // drawing mode is over!
      Session.set('drawing', false);
      // we now record the screen, add to undo chain
      addSnapshot();
    },
    'doubletap #picture': function (ev) {
      // clear the screen
      wipe();
      // wipe out our undo history
      Meteor.call('clearActivity');
    }
  });
```

25. All that's left now is the `keydown` event logic, so that we have proper undo/redo. Quickly make a file called `tmpl-body.js` in your `[project root]/client/` folder and add the following code:

```
Template.body.events({
  'keydown' : function(ev){

    // if there's no undo history, no reason to continue,
// so return.
    var histLength = Activities.find({user:Meteor.userId()}).
fetch().length;
    if (!histLength) return;

    // If it's not a CTRL+Z or CMD+Z, we don't care, so
// return.
```

```
    if ((!ev.metaKey && !ev.ctrlKey)||(ev.keyCode!==90))
return;

    // find the current position in the undo chain, if any.
    var curHist = Session.get('history')||0;

    // if it was SHIFT+CMD+Z, it means redo, so decrement
// the history
    if (ev.shiftKey)
      curHist--;
    // otherwise, increment the history
    else
      curHist++;

    // if we're past the boundaries of TIME and SPACE we
    // certainly don't care about JavaScript anymore, so
let's return.
    if(curHist<0 || curHist> histLength-1 ) return;

    // after all that, set the new undo chain position
    Session.set('history',curHist);
  }
});
```

Whew! We made it through! Save all of your changes and navigate to your project in a browser, via `http://localhost:3000`. After you log in/create an account, you should see a nice color palette, a picture of your choice, and a brush size/preview. The following features are available:

- Double-click on the brush preview to toggle `'eraser'` mode
- Click and drag left and right on the brush preview to resize the brush
- Double-click on the page to erase and clear your undo history
- Press *Ctrl* + *Z* or *CMD* + *Z* to undo your strokes
- Press *Shift* + *Ctrl* + *Z* or *Shift* + *CMD* + *Z* to redo your strokes
- Log out and in with another user, to allow multi-user use

Do note that your undo/redo history chain is preserved, even if you log out. Once you log back in, you can go through your paint creations, stroke by stroke.

If you are a particularly awesome artist like my daughter, and you have properly coded everything, your page will look similar to the following masterpiece:

How it works...

Oh boy! That was a lot of code, just to demonstrate how the `canvas` element works, wasn't it? Instead of going through all of it step by step, let's go through the important pieces, related to the `Canvas` element.

Inside of `tmpl-picture.js`, we initialize our `canvas`. In the `Template.picture.rendered()` helper block, we first find our `canvas` element using `this.$('#picture')`:

```
canvas = this.$('#picture');
```

We then get the context (the handle) using the very appropriately-named `getContext()` function. We only want the 2D representation and not the 3D, so we pass `'2d'` as the argument to `getContext`:

```
ctx = canvas[0].getContext('2d');
```

Moving on to `cb-client.js`, we have several uses for the context, passed around in the global `ctx` variable. In the `drawLine` function, we set the size (`lineWidth`), color (`strokeStyle`) and we set the type of stroke (`globalCompositeOperation`) according to whether we are erasing (`'destination-out'`) or laying down some paint (`'source-over'`):

```
drawLine = function (from, to, color,size) {
  ...
    ctx.lineWidth = size;
  ...
    ctx.strokeStyle = color;
  if (Session.get('erase')){
    ctx.globalCompositeOperation = 'destination-out';
  } else {
    ctx.globalCompositeOperation = 'source-over';
  }
  ...
```

Once all that is set/determined, we tell the canvas that we are starting to draw (`beginPath`); we move our stroke (`moveTo`, `lineTo`) and then we clean up (`closePath`, `stroke`):

```
ctx.beginPath();
ctx.moveTo(from.x, from.y);
ctx.lineTo(to.x, to.y);
ctx.closePath();
ctx.stroke();
```

To save database space/transactions, we don't record every stroke. Instead, we wait for a stroke to be finished (the `'panend #picture'` event) and we add a snapshot of the entire canvas. We did this inside the `addSnapshot` helper function, with a call to the `canvas.toDataURL()` method:

```
var canvasPic = canvas[0].toDataURL();
```

Once we have the `canvas` graphics element represented as data, we simply save it to the `Activities` collection, via `Activities.insert()`:

```
Activities.insert({
  user:userId,
  canvas:canvasPic,
  createdAt:timestamp
});
```

Retrieving and displaying a saved screenshot from the `Activities` Collection is just as easy. The `ctx` handle takes care of the drawing for us. All we have to do to get that monkey to dance is give it the data. So, inside `paintActivity`, we create a new `Image()` object and we add a `.src` to that `imageObj`. We then call `ctx.drawImage()` when the `.onLoad` event callback is triggered:

```
var imageObj = new Image();
imageObj.onload = function () {
  ctx.drawImage(this, 0, 0);
};
```

Lastly, and most simply, if we want to `wipe` the screen, we simply call `ctx.clearRect()` with the dimensions we want cleared (in this case, the `width` and `height` of the canvas):

```
ctx.clearRect(0, 0, canvas.attr('width'), canvas.attr('height'));
```

There's a lot of Meteor magic going on, in coordination, so that we are using as little functional programming as possible, but that coordination is covered in other recipes.

There's more...

Actually, there's a whole lot more! We could have added custom color swatches (use `Meteor.call('addSwatch','#yourcolorhere')` to do that from the console, by the way), or enabled drag and drop, and the storing of other pictures. You can use the *Using the HTML FileReader to upload images* recipe found in this chapter to do that, if you would like.

The core functionality, however, will remain the same: reference a `<canvas>` object, grab the context, and draw, baby draw!

For an informative and thorough tutorial on HTML canvas, please visit: `http://www.html5canvastutorials.com/`.

See also

- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*
- The *Inserting templates with Spacebars* and *Creating customized global helpers* recipes in *Chapter 3*, *Building Great User Interfaces*
- The *Creating and consuming a reactive value* recipe in *Chapter 6*, *Mastering Reactivity*
- The *Using the HTML FileReader to upload images* recipe in this chapter

8

Integrating Third-party Libraries

In this chapter, you will learn the following topics:

- Using npm packages directly
- Building graphs with D3.js
- Creating cutting-edge UIs with Polymer

Introduction

With so many third-party packages available via Atmosphere it's very easy to build, basically, anything! If you have a specific development need—some type of logic or library—chances are someone else has already packaged it up for you (and if they haven't, what a great opportunity for you!).

That being said, not all packages are created equal. Some will have fantastic documentation, with samples and tutorials on how to implement a library inside Meteor. Others, not so much. In either case, you will at some point need to roll up your sleeves and do a bit of work yourself. In this chapter, we will go over a few of the more popular third-party libraries, and show you how to implement them in Meteor.

Using npm packages directly

We have seen from the *Using npm modules* recipe in *Chapter 2*, *Customizing with Packages*, how to wrap npm packages inside your own personal package, using the `Npm.depends()` directive. That's all well and good, but what if we want to just use npm directly, without creating a custom package? Arunoda Susiripala — one of the most prolific (and brilliant) Meteor community developers, has written a package that helps us do just that. In this recipe, you will learn how to use the `meteorhacks:npm` package to implement a npm module directly in your code. Specifically, we will use the `Highlight.js` module on the server to properly format JavaScript code, as you might see in markdown or in an online code editor.

Getting ready

We will be using a default project installation, with the `client`, `server`, and `both` folders, to keep the code clean and readable. We will also use the `private` folder for some static content, so let's add that while we're at it. In a terminal window, navigate to where you would like your project to reside, and execute the following commands:

```
$ meteor create npmdirect
$ cd npmdirect
$ rm npmdirect.*
$ mkdir {client,server,private,both}
```

We also want to make use of the `meteorhacks:npm` package, so let's do that now. Run the following commands in the terminal:

```
$ meteor add meteorhacks:npm
$ meteor
```

Lastly, we will want to open a browser to `http://localhost:3000`, so we can watch the fun! We're now ready to start creating our direct npm integrations.

How to do it...

We will be adding the `Highlight.js` Node module and adding some sample code to leverage the module, using an inline template as our work area. Proceed with the following steps to use npm packages directly:

1. We will first add the `Highlight.js` Node module. To add a direct-use npm module, we will make a simple declaration in the `packages.json` file that was automatically added to our project when we added `meteorhacks:npm`. Open `[project root]/packages.json` and add the following declaration:

   ```
   {
     "highlight.js" : "8.4.0"
   }
   ```

   If you look in the terminal window where Meteor is running, you will see something similar to the following text:

   ```
   npm-container: updating npm dependencies -- highlight.js...
   => Meteor server restarted
   ```

   Our `Highlight.js` module is now ready to use!

2. We will now create the server logic needed to use `Highlight.js`. Create a file named `server.js` in the `[project root]/server/` directory. Open that file for editing and add the following code:

   ```
   Meteor.methods({
     highlight : function(){
       return setC();
     }
   });

   var setC = function(){
     var hLight = Meteor.npmRequire('highlight.js');
     var code = Assets.getText('code.txt');
     code = hLight.highlight('javascript',code,true);
     return code.value;
   };
   ```

3. Next, let's create a sample code file in the `private` folder. We are going to use a base text file for the code we will highlight to make the syntax much easier (rather than try to do it in a JavaScript `var`, which can be messy with escape sequences). Meteor lets us use the `[project root]/private/` folder to store static files only visible on the server. Create a file in the `private` folder, called `code.txt`, and add some (valid) JavaScript to the file. We will reuse the `var setC = ...` function from the preceding step, but you can put anything you want inside there, provided it's valid JavaScript. Once you've added some code, save your changes.
4. We now need to create a simple code panel template. Create a file in `[project root]/client/` named `main.html`. Open that file for editing, and add the following code:

```
<body>
{{> code}}
</body>
<template name="code">
  <pre class="hljs">{{{highlighted}}}</pre>
</template>
```

5. With the template in place, we can add `Template.code.helpers`, and use `Meteor.call()`. Create a file in `[project root]/client/` called `tmpl-code.js` and add the following code:

```
Template.code.helpers({
  highlighted : function(){
    return Session.get('code');
  }
});

Meteor.startup(function(){
  Meteor.call('highlight', function(e,d){
    if (e) return;
      Session.set('code',d);
  });
});
```

After saving these changes, your code from the `code.js` file will be visible, albeit with little/no CSS formatting. If you check your browser, you should see something similar to the following screenshot:

6. Well, that's not all that impressive, is it? Let's really highlight what is happening with some custom CSS. In a browser, navigate to the official list of `Highlights.js` themes, found at `https://github.com/isagalaev/highlight.js/tree/master/src/styles` and pick any theme you like. Simply click on the name of the theme, click on the button labeled **Raw**, and copy the corresponding CSS code. Then, create a file named `style.css` in your `[project root]/client/` folder, paste the code you copied, and save the file. Your formatted text should now have some noticeable formatting and coloring, similar to the following screenshot (this is the `Solarized Light` theme, copied from `solarized_light.css` at the link that we mentioned previously):

How it works...

Inside of our `server.js` file, we created a simple `Meteor` method, available on (and called by) the client via `Meteor.call()`. We created a separate function that is called inside of the method even though we didn't have to (we could have just put the logic directly in the method call), because we wanted to keep our code clean and illustrative. Inside the `setC()` function, we make direct use of a npm module by using `Meteor.npmRequire()`:

```
var hLight = Meteor.npmRequire('highlight.js');
```

Quickly referencing our text file via `Assets.getText()`, we pipe our sample code into the `Highlights.js` file in the `.highlight()` method, which takes the language (`'javascript'`) as an argument, to let the module know what language highlighting we would like to see in the resultant formatted text:

```
var code = Assets.getText('code.txt');
code = hLight.highlight('javascript',code,true);
return code.value;
```

We return the formatted text, which is subsequently updated in the client UI, through the `Session.get('code')` variable. The template renders the raw HTML string, thanks to the use of `{{{triple moustaches}}}` rather than double, and the CSS file we added takes care of the eye candy for us.

In a nutshell, it really is that simple — by adding the `meteorhacks:npm` package, we can simply declare which npm modules we would like to use in our `packages.json` file, and then use them directly in Meteor, via the `Meteor.npmRequire()` function.

There's more...

As you may be aware, npm runs strictly asynchronously and is non-blocking. This means that a lot of the times when you want to use an npm module in Meteor, and want to do so synchronously, you need to wrap the call in an `Async` wrapper. We will explore how to do this in more depth in *Chapter 11*, *Leveraging Advanced Features,* but the `meteorhacks:npm` package allows us to do this quickly/easily as well, via the `Async.wrap()` and `Async.runSync()` methods.

> To learn more about `Async.wrap` and other methods available in `meteorhacks:npm`, please see the introduction available at `https://atmospherejs.com/meteorhacks/npm`.

Lastly, even though it's undocumented (and therefore not officially supported/subject to change), there are some npm modules used by the core Meteor server libraries that are exposed and usable on the server side. To use any of these, simply use `Npm.require()` in your server code, and you don't need to implement anything else. You can use them directly. The current list, as of `v1.0.2`, is as follows:

- `child_process`
- `crypto`
- `fibers`
- `fibers/future`
- `fs`
- `http`
- `os`
- `path`
- `semver`
- `source-map-support`
- `underscore`
- `url`

See also

- The *Using npm modules* recipe in *Chapter 2*, *Customizing with Packages*
- The *Inserting raw HTML using triple braces* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Handling asynchronous events* and *Using asynchronous functions* recipes in *Chapter 11*, *Leveraging Advanced Features*

Building graphs with D3.js

Using third-party libraries to render content inside Meteor is simple and easy to do, and there are a lot of great libraries out there. One of our personal favorites is the `D3.js` library (`http://d3js.org/`). It's well-documented, well-supported, and is a representative example of how to implement other (mostly) self-contained graphics rendering libraries. In this recipe we will create a demo graph application, complete with animations, using `D3.js`.

Getting ready

We will be using a default project installation, with the `client`, `server`, and `both` folders, and we will of course need the official `D3.js` package. In a terminal window, navigate to where you would like your project to reside, and execute the following commands:

```
$ meteor create d3sample
$ cd d3sample
$ rm d3sample.*
$ mkdir {client,server,both}
$ meteor add d3js:d3
$ meteor
```

With that out of the way, let's build us some sample graphs in D3!

How to do it...

Normally, the `D3.js` data sources are updated manually, or as the result of a feed, with an `update` statement of some kind being called. Because Meteor enables reactive computations (see the *Creating and consuming a reactive value* recipe in *Chapter 6*, *Mastering Reactivity*), we are going to put the `D3.js` rendering logic right into a `Tracker.autorun()` computation instead. Let's set up our reactive data source, throw that rendering logic into an `autorun`, and add some animations for good measure!

1. We first need to create and initialize the `Letters` collection. Create a file named `model.js` in your `[project root]/both/` folder, and add the following code:

   ```
   Letters = new Mongo.Collection('letters');
   ```

   Let's add an initialization of that collection as well. In your `[project root]/server/` folder, create a file named `server.js` and add the following `Meteor.startup` function code:

   ```
   Meteor.startup(function(){
     if (!Letters.find().fetch().length){
       Letters.insert({letter:'A',frequency:.08167});
       Letters.insert({letter:'B',frequency:.01492});
       Letters.insert({letter:'C',frequency:.02782});
       Letters.insert({letter:'D',frequency:.04253});
       Letters.insert({letter:'E',frequency:.12702});
       Letters.insert({letter:'F',frequency:.02288});
       Letters.insert({letter:'G',frequency:.02015});
       Letters.insert({letter:'H',frequency:.06094});
       Letters.insert({letter:'I',frequency:.06966});
   ```

```
    Letters.insert({letter:'J',frequency:.00153});
    Letters.insert({letter:'K',frequency:.00772});
    Letters.insert({letter:'L',frequency:.04025});
    Letters.insert({letter:'M',frequency:.02406});
    Letters.insert({letter:'N',frequency:.06749});
    Letters.insert({letter:'O',frequency:.07507});
    Letters.insert({letter:'P',frequency:.01929});
    Letters.insert({letter:'Q',frequency:.00095});
    Letters.insert({letter:'R',frequency:.05987});
    Letters.insert({letter:'S',frequency:.06327});
    Letters.insert({letter:'T',frequency:.09056});
    Letters.insert({letter:'U',frequency:.02758});
    Letters.insert({letter:'V',frequency:.00978});
    Letters.insert({letter:'W',frequency:.02360});
    Letters.insert({letter:'X',frequency:.00150});
    Letters.insert({letter:'Y',frequency:.01974});
    Letters.insert({letter:'Z',frequency:.00074});
  }
});
```

2. While we have `server.js` open, let's add a quick helper method to update the frequency, which we will use a bit later. Add the following code to `server.js`:

```
Meteor.methods({
  updateFrequency : function(letter,frequency){
    Letters.update({letter:letter},
      {$set:{frequency:frequency}});
  }
});
```

3. Now, we will add some simple scaffolding and styling. Our HTML is extremely simple, because `D3.js` is doing the heavy lifting. That said, we do want to put our `svg` element into a Meteor Template. Navigate to your `[project root]/client/` folder, create a file named `client.html`, and add the following code:

```
<body>
  {{> diagram}}
</body>

<template name="diagram">
  <svg class="chart"></svg>
</template>
```

Let's add some simple styling to adjust fonts and colors, and so on. Create a file named `[project root]/client/style.css` and add the following CSS declarations:

```
.bar {
  fill: steelblue;
}
.bar:hover {
  fill: brown;
}
.axis {
  font: 10px sans-serif;
}
.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
.x.axis path {
  display: none;
}
```

We are now ready to add the `D3.js` rendering logic, inside `rendered/autorun`. The following code is taken almost line-for-line from Mike Bostock's awesome introductory tutorial on `D3.js`. The easiest thing to do would be to take the existing code base from that tutorial, copy and paste it into a local file, and then modify it. To do this, navigate in a browser to `http://bl.ocks.org/mbostock/3885304` and copy everything inside the `<script>` tag to the ending `</script>` tag. Create a file named `client.js` in your `[project root]/client/` folder, and paste the code into the new file (about 60 lines of code).

We will now modify the code from that page in the following four ways:

- Wrap all of the code inside a `Template.rendered` callback
- Wrap and extend the `d3.selectAll` rendering code inside an `autorun` computation
- Move and modify the axes rendering code outside the `autorun`
- Change data to point to our `Letters.find()` Mongo query

The completed `Template.diagram.rendered` callback is listed in the following code. It mainly consists of the `D3.js` sample code, with the preceding four modifications highlighted. If it's easier, you can simply copy and paste from below, rather than making the modifications yourself. However, it's worthwhile to at least go over the changes, to see how the code was modified. In your `client.js` file, make the following modifications to the existing code:

```
Template.diagram.rendered = function(){
  //we wrap everything in the Template.rendered() callback,
  // so that we don't interfere with Blaze
  var margin = {top: 20, right: 20, bottom: 30, left: 40},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;

  //We are going to set the domains for x an y immediately
  //(assuming the alphabet isn't going to change)
  x = d3.scale.ordinal()
  .domain('ABCDEFGHIJKLMNOPQRSTUVWXYZ'.split(''))
  .rangeRoundBands([0, width], .1);

  //ALSO: note that we *removed* the 'var' declarations,
  //so that x and y are global / accessible
  y = d3.scale.linear()
  .domain([0,0.15])
  .range([height, 0]);

  var xAxis = d3.svg.axis()
  .scale(x)
  .orient("bottom");

  var yAxis = d3.svg.axis()
  .scale(y)
  .orient("left")
  .ticks(10, "%");

  // We are moving the axes creation (and SVG init)
  // to be *outside* our autorun()
  var svg = d3.select("body").append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom)
  .append("g")
  .attr("transform", "translate(" + margin.left + "," +
  margin.top + ")");

  svg.append("g")
```

```
.attr("class", "x axis")
.attr("transform", "translate(0," + height + ")")
.call(xAxis);

svg.append("g")
.attr("class", "y axis")
.call(yAxis)
.append("text")
.attr("transform", "rotate(-90)")
.attr("y", 6)
.attr("dy", ".71em")
.style("text-anchor", "end")
.text("Frequency");


//We move D3.js rendering inside Tracker.autorun()
this.autorun(function(){
  // Instead of reading data from a static file,
  // we access the Letters collection
  var data = Letters.find().fetch();
  if (!data.length) return;

  // To use D3.js's built-in update tracking,
  // we need access to our d3.selectAll() object..
  var bars = svg
  .selectAll(".bar")
  .data(data, function(d){return d._id;});

  // On new (when initializing), we append and animate
  bars.enter()
  .append("rect")
  .attr("class", "bar")
  .attr("x", function(d) { return x(d.letter); })
  .attr("width", x.rangeBand())
  .attr("height",0)
  .attr("y", height)
  .transition()
  .attr("y", function(d) { return y(d.frequency); })
  .attr("height", function(d) { return height -
    y(d.frequency); });

  // On change, we just animate to the new position
  bars
  .transition()
```

```
      .duration(200)
      .ease("sin-out")
      .attr("y", function(d) { return y(d.frequency); })
      .attr("height", function(d) { return height -
        y(d.frequency); });

    });
  };
```

4. We can now create and call the `randomize` function. Our page should be rendering correctly now (check at `http://localhost:3000` and see for yourself), but it's not moving or doing anything else. Let's change that by changing to random frequency values for our letters. Towards the bottom of `client.js`, just after the `autorun` block, and just before the ending bracket for the rendered `callback` block, add the following timer call:

```
});//<--- end of autorun()

// everything's set! let's randomize 5 times / second...
  Meteor.setInterval(randomize,200);

};//<-- end of rendered()
```

Finally, add the `randomize()` function at the very bottom of the file, after everything else:

```
// Our randomize function
randomize = function(){
  // get a random position between 0-25
  var ranLetter = ~~(Math.random()*26),
  // and a random frequency between 0-15%
    ranFreq = (Math.random()*0.15);

  // get the actual character
  ranLetter = x.domain()[ranLetter];

// update the frequency using a server call,
// because it's easier than tracking down the _id
  Meteor.call('updateFrequency',ranLetter,ranFreq);
};
```

Save all of your changes, and your bar chart should be hopping along happily, similar to the following screenshot:

How it works...

There are two key rules to remember when integrating `D3.js` (and, by extension, any other rendering engine):

1. Put all your DOM/SVG manipulation logic inside `Template.rendered()`
2. Make the data source reactive, and wrap the rendering logic inside `Tracker.autorun()`

We followed rule #1 when we added the entire `D3.js` logic from the tutorial inside of `Template.diagram.rendered()`. This tells Blaze to keep its (awesome, yet grubby) hands off everything we'll be doing. We're essentially telling Blaze to go have a smoke break, while we implement some good ol' fashioned DOM/SVG manipulation (and animations!).

Then, instead of relying on a non-reactive data source, where we would have to periodically check for updates and/or call an `update()` function whenever results change, we used Meteor's declarative, reactive programming style by wrapping the `D3.js` logic inside an `autorun` function. That `autorun` function will re-run every time there's a change to the `Letters` collection, thanks to `var data = Letters.find().fetch()`; which is a reactive data source.

In other words, we followed rule #2 by creating a `D3.js`-friendly reactive computation, instead of using a functional (non-reactive) data model, like a hobo. Mike Bostock would be proud of us, as the avoidance of functional programming is one of the core tenets of `D3.js`.

It seems like we wrote a lot of code because of the `D3.js` sample code, and there are some details we're leaving out, such as moving the `y` and `x` axes rendering logic outside the `autorun` function (because we don't want to re-render it every time), but it really is that simple. Notice how incredibly simple (and loosely-coupled) our update statement is, inside the `randomize()` function: one. single. line.

When it comes to third-party renderers, wrap using `rendered()` and be reactive using `autorun()`.

> Wrap using `rendered()`, be reactive using `autorun()`.

There's more...

We strongly encourage you to look deeper into `D3.js`, which (as mentioned) runs on the same declarative programming principle as Meteor. There's a reason it integrates so cleanly, and for SVG graphs and animations, it really is top notch.

> Tutorials for `D3.js` can be found at `https://github.com/mbostock/d3/wiki/Tutorials`.

See also

- The *Creating dynamic lists* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Creating dynamic graphs with SVG and Ajax* recipe in *Chapter 7*, *Using Client Methods*

Creating cutting-edge UIs with Polymer

Polymer (`https://www.polymer-project.org/`) is ambitious. It is ambitious in the very best possible way, in that, if it succeeds, it will make our development lives so much better.

One of the largest time-wasting activities we as web developers go through is tweaking CSS, repositioning DOM elements, and trying to manually code animations. It's not that those things are bad (some of us quite enjoy well-designed components); It's that nearly every time, our efforts are thrown away when a new project or design theme is introduced.

Through the use of reusable, reliable components that work the same on every modern browser (or at least that's the goal — ambitious, right?), Polymer intends to abstract the design from the development, allowing developers to focus on development, and designers to focus on design.

This recipe will show you how to use Polymer components inside a Meteor application.

Getting ready

We are going to break down the actual recipe into two parts: configuring Meteor and Polymer and adding Polymer elements to our app. Because of this, we are going to handle the configuration and setup in the *Getting ready* step to keep the code and explanation of the code easier to understand.

Creating your app and folders

We will need several folders and subfolders we don't normally use, so let's get our entire file structure out of the way. Open a terminal window, navigate to where you would like to create your root project, and execute the following commands:

```
$ meteor create polymersample
$ cd polymersample
$ rm polymersample.*
$ mkdir -p client/templates/init
$ mkdir {both,server}
$ mkdir -p public/components
```

Creating your files

To save time creating files one by one later on, let's create them all right now. In the same terminal window, in your `[project root]` folder, execute the following commands:

```
$ touch .bowerrc
$ touch client/imports.html
$ touch client/main.html
$ touch client/styles.css
$ touch client/templates/tmpl-authors.html
$ touch client/templates/tmpl-authors.js
$ touch client/templates/tmpl-search.html
$ touch client/templates/tmpl-search.js
$ touch client/templates/init/head.html
$ touch both/model.js
$ touch server/server.js
```

Configuring Bower and installing Polymer

Bower is the safest and best way to install the Polymer libraries, as it makes sure we have the latest and greatest. You will want to make sure that you have Bower installed. If you don't, in your terminal window run the following command:

```
$ npm install -g bower
```

> You may need to run npm as `sudo` because you're installing globally. If so, run the following command:
>
> **`sudo npm install -g bower.`**

With Bower installed, we want to tell Bower where we would like the Polymer components installed. In Meteor, that is in the `public/components` folder. In a text editor, open the `[project root]/.bowerrc` file, add the following line and save your changes:

```
{"directory":"public/components/"}
```

We are now ready to initialize our project using Bower. In a terminal window, in your `[project root]/` directory, enter the following command:

```
$ bower init
```

Bower is about to ask you a lot of questions! The good news is the answer to most of them is the *return* key, or a *space* and then the *return* key. Even better news is that most of the answers don't really matter. But, you do need to go through this, so that Bower will generate the `bower.json` file for you, which will keep track of which Polymer libraries we install (more on that in a minute), so answer the questions similar to the following example:

```
? name: Polymer Sample
? version: 0.0.1
? description:
? main file:
? what types of modules does this package expose?:
? keywords:
? authors:
? license: none
? homepage:
? set currently installed components as dependencies?: No
? add commonly ignored files to ignore list?: Yes
? would you like to mark this package as private which prevents it from
being accidentally published to the registry?: Yes
```

You will then get a preview of your `bower.json` file, as shown in the following screenshot:

Provided that `private:true`, and `ignore: [...]` are in place, type *Y* and hit *Enter*. Your Bower configuration is now complete!

We are now ready to install the Polymer libraries. In the same terminal window, enter the following commands, one after another, allowing one to complete before moving on to the next:

```
$ bower install polymer --save
$ bower install polymer/core-elements --save
$ bower install polymer/paper-elements --save
$ bower install polymer-github-card --save
```

On the last install, Bower will ask you what version of `polymer` and `core-components` will be supporting `polymer-github-card`. Select the version that the other components are dependent upon (as of this writing, this is `[library]#^0.5.0`).

Adding helper Meteor packages

There are three third-party packages we need to add. One of them, (`voodoohop:masonrify`), will assist with our layout, and the other two, (`meteorhacks:inject-initial` and `differential:vulcanize`), will allow Polymer elements to be added to our Meteor project with the least amount of pain (we suspect that this will get better in the future, but for now this is the workaround; and honestly it's not really so bad). In a terminal window, in your `[project root]/` folder, enter the following commands:

```
$ meteor add voodoohop:masonrify
$ meteor add meteorhacks:inject-initial
$ meteor add differential:vulcanize
```

Configuring Meteor

Setup is almost done! Meteor generally gets along with other frameworks, but because Polymer has some pre-rendering DOM manipulations (which are really cool, but mess with Meteor's mojo...) we have to be able to inject an attribute into the `<body>` tag, and we also need to ensure that the base `polymer` and `webcomponents` files are referenced properly.

In an editor, open the `[project root]/server/server.js` file, and add the following HTML injection code:

```
Meteor.startup(function(){
  Inject.rawModHtml('addUnresolved',function(html){
    return html = html.replace('<body>', '<body unresolved>');
  })
});
```

Now we will take care of the baseline file references. Open `[project root]/client/templates/init/head.html` and add the following code:

```
<head>
  <script src="/components/webcomponentsjs/webcomponents.js"></script>
  <title>github authors</title>
</head>
```

One last step (this is for `vulcanize` to be able to consolidate all the component references), open `[project root]/client/imports.html` and add the following lines:

```
<!-- Components -->
<link rel="import" href="/components/polymer/polymer.html">

<!-- Styles -->
<link rel="import" href="/components/font-roboto/roboto.html">
```

Setup is complete! Let's start Meteor (run the `meteor` command in the terminal), open a browser to `http://localhost:3000`, and start using Polymer components!

How to do it...

Two caveats to what we're about to do: first, currently Polymer plays best with the Chrome browser. The samples below will work in other browsers, but there are some idiosyncrasies, such as `click` or `keypress` events, that may cause a little bit of frustration. Second, due to timing issues, as you're making changes to your code, there may be some instances where you will need to manually refresh your browser (which is a very rare thing when using Meteor!).

Keep in mind that Polymer is nowhere near being a `1.0` product — you are on the cutting edge with this recipe — so you'll have to bear with us, as we go through the fundamentals of using Polymer inside Meteor. Optimizing the timing and events and other behaviors across all other browsers would be a book in itself, so let's focus on the foundational part, and as Meteor and Polymer progress, these issues will work themselves out.

1. We will first add our CSS styling. Polymer does the heavy lifting here, so let's get what little CSS we have to contribute out of the way. Open `[project root]/client/styles.css` and add the following CSS declarations:

```
html,body {
  height: 100%;
  margin: 0;
  font-family: 'RobotoDraft', sans-serif;
}
.container {
  width: 80%;
  margin: 50px auto;
}
div.sText {
  display:inline-block;
  width: 20rem;
}
div.sBtn {
  display: inline-block;
  width: 5rem;
  vertical-align: text-top;
}
paper-button.colored {
  color: white);
}
paper-button[raised].colored {
  background: rgb(66, 133, 244);
  color: white;
}
```

2. Let's take a look at the `polymer-github-card` component, using a static representation, just to see what we're playing around with. Open `[project root]/client/imports.html`, and add the following line just below the `"…polymer.html"` entry in the `Components` section:

```
<link rel="import"
href="/components/polymer-github-card/dist/polymer-github-card.
html">
```

Now open `main.html`, found in the same folder, and add the following code:

```
<body>
  <div class="container">
    <polymer-github-card user="meteor"></polymer-github-card>
  </div>
</body>
```

In your browser, after `vulcanize` recompiles your `<header>`, you should see something similar to the following screenshot:

3. Now that we have confirmed that Polymer is working correctly, let's put the `polymer-github-card` component into a Meteor template, and create the ability to add multiple cards, based on a Mongo collection. Open `[project root]/both/model.js` and add the following line:

   ```
   Authors = new Mongo.Collection('authors');
   ```

4. Now let's create our template. You'll remember that we're using `voodoohop:masonrify` to control the layout, so we will need to wrap our template declarations inside the `{{#masonry...}}` blocks, as appropriate. Open `[project root]/client/templates/tmpl-authors.html` and add the following code:

   ```
   <template name="authors">
     {{#masonryContainer
       columnWidth=265
       gutter=5
       transitionDuration="0.2s"
       id="MasonryContainer"
     }}
   ```

```
    {{#each authors}}
      {{> authorCard}}
    {{/each}}
  {{/masonryContainer}}
</template>

<template name="authorCard">
  {{#masonryElement "masonryContainer"}}
    <div>
      <polymer-github-card
        user="{{userid}}">
      </polymer-github-card>
    </div>
  {{/masonryElement}}
</template>
```

Again, the majority of the previous code relates to layout. The actual call to dynamically populate `polymer-github-card` components is found in the `authorCard` template, and references `{{userid}}` in the `user` attribute. `userid` comes free, as it will be a property for each record in the `Authors` collection.

5. Let's now create the `Template.helper` function for passing the `Authors` collection, via the `Collection.find()` function. Open `[project root]/client/templates/tmpl-authors.js` and add the following code:

```
Template.authors.helpers({
  authors: function(){
    return Authors.find().fetch();
  }
});
```

6. While we're at it, lets make deleting records very simple. In `tmpl-authors.js`, add the following event handler at the bottom:

```
Template.authors.events({
  'dblclick polymer-github-card': function(e){
    Authors.remove({_id:this._id});
  }
});
```

7. We now need to modify our main template to reference our new dynamic template. Open `main.html` and modify the contents to look like the following:

```
<body>
  <div class="container">
    {{> authors}}
  </div>
</body>
```

8. Save all your changes, look in your browser (give `vulcanize` a second to do its thing), and you should see a completely blank page. Wow! That's only because our `Authors` collection is empty. Let's fix that. In your browser console, execute the following commands:

```
> Authors.insert({userid:'meteor'})
> Authors.insert({userid:'glasser'})
```

As you enter each command, a new card will pop into place, with the Meteor logo, and David Glasser's smiling happy childhood picture, as shown in the following screenshot:

The command line is so 1960's! Let's update our page to use some paper-elements components, and give us an easy way to add more GitHub authors.

9. Open `[project root]/imports.html` and add links for the components we will be using (which are found in our `public/components/` folder, by the way...) just below the `"...polymer.html"` declaration. When finished, your file should look like the following:

```
<!-- Components -->
<link rel="import" href="/components/polymer/polymer.html">
```

```
<link rel="import"
href="/components/paper-ripple/paper-ripple.html">
<link rel="import"
href="/components/paper-shadow/paper-shadow.html">
<link rel="import"
href="/components/paper-input/paper-input.html">
<link rel="import"
href="/components/paper-button/paper-button.html">
<link rel="import"
href="/components/polymer-github-card/dist/polymer-github-
card.html">


<!-- Styles -->
<link rel="import"
href="/components/font-roboto/roboto.html">
```

Let's create our `search` template, which will contain a `paper-input` component, and a `paper-button` component. Open `[project root]/client/templates/tmpl-search.html` and add the following code:

```
<template name="search">
  <div class="sText">
    <paper-input-decorator
      id="searchText"
      floatingLabel
      label="find an author">
      <input is="core-input" id="sInput">
    </paper-input-decorator>
  </div>
  <div class="sBtn">
    <paper-button
      raised
      class="colored"
      role="button"
      tabindex="0">
      search
      </paper-button>
  </div>
</template>
```

There's nothing special about the preceding code, Meteor-wise. It's all just straight up Polymer formatting and configuration.

To learn more about Polymer components, please visit:
`https://www.polymer-project.org/docs/polymer/polymer.html`.

10. We will want to add a new card whenever we perform a search, that is, whenever we hit *Enter* inside our `paper-input` component, or click on our `paper-button` component. Open `tmpl-search.js`, found in the same `client/templates/` folder, and add the following code:

```
Template.search.events({
  'keypress #sInput' : function(e){
    if (e.keyCode!=13) return;
    addAuthor();
  },
  'click paper-button': function(e){
    addAuthor();
  }
});

function addAuthor(){
  var sInput = $('#sInput'),
      sVal = sInput.val();
  sInput.blur();
  if (!sVal || (Authors.findOne({userid:sVal}))) return;
  sInput.val('');
  Authors.insert({userid:sVal});
}
```

11. The very last step is to add a template inclusion for our `search` template to `main.html`. Open `[project root]/client/main.html` and add the following highlighted code:

```
<body>
  <div class="container">
    {{> search}}
  </div>
  <div class="container">
    {{> authors}}
  </div>
</body>
```

All done! Check out your app in the browser, allowing for refresh/recompiling, and you should be able to add as many GitHub author cards as you would like. Add some authors (suggestions: `meteorhacks` , `arunoda` , `d3` , `mbostock` , `voodoohop` , `pazguille` , `polymer` , `addyosmani`) and new cards will appear after each entry, as shown in the following screenshot:

How it works...

Very rarely do we need to refer to the *Getting ready* section to fully understand what's going on, but the context around where and how to place Polymer files is important, so let's start there. We directed Bower to put any and all Polymer files into a subfolder belonging to `[project root]/public/`. Meteor treats files in the `public` folder differently than other folders—it treats them as static, and will not process the `.js` or `.html` files found inside.

This is to our advantage, because there are currently some pretty nasty conflicts between the Polymer and Meteor renderers. Specifically, both use the `{{double-stache}}` notation and the `<template>` tag, and interpret them differently. To resolve these conflicts (Nobel peace prize incoming!...) we "fence off" each framework, allowing Polymer to link to anything in the `public` folder, and having Meteor ignore everything in the same place. It's almost like the **Meteor Development Group** (**MDG**) planned it that way! Of course they did, anticipating and encouraging the use of other opinionated frameworks. MDG truly does want Meteor to play well with others, and in this case, it allows us to use Polymer in our applications.

A couple of other adjustments had to be made, such as inserting link elements into our `<head>` block, and injecting the unresolved attribute into the `<body>` element, but these are small, one-time issues, and once they're complete, we are free to use Polymer components to our heart's content.

As for the use of the Polymer components themselves, we can either use them directly (as shown in step #2) with no involvement from Meteor, or we can incorporate them into our `{{#each...}}` template or other template inclusions. We then have access to the normal Meteor `Template.helpers` and `Template.events`, which gives us easy, declarative-style access to reactive data.

Specifically, if we look at the `tmpl-authors.html` file, we will see that we are using the `{{#each authors}}` and `{{userid}}` template helpers, to iterate and render `polymer-github-card` components on the fly:

```
<template name="authors">
  ...
  {{#each authors}}
    {{> authorCard}}
  {{/each}}
  ...
</template>

<template name="authorCard">
  ...
    <polymer-github-card
          user="{{userid}}">
    </polymer-github-card>
  ...
</template>
```

In `tmpl-authors.js` and `tmpl-search.js`, we make use of `Template.events` to interpret `keypressed`, `click`, and `dblclick`, and are able to use the in-context (inline) data to modify records, such as when we delete a record, using the `Authors.remove({_id:this._id})` statement:

```
Template.authors.events({
  'dblclick polymer-github-card': function(e){
    Authors.remove({_id:this._id});
  }
});
```

There's more...

There are a very large number of interaction and compatibility/performance nuances between Polymer and Meteor, which are beyond the scope of this recipe (and this book). We recommend perusing Stack Overflow (`http://stackoverflow.com/questions/tagged/meteor`), the Meteor forums (`https://forums.meteor.com/`), or just lifting the hood and experimenting, to gain more experience in properly integrating Polymer and Meteor.

See also

- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*
- The *Creating custom components* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Implementing a simple collection* recipe in *Chapter 4*, *Creating Models*

9

Securing Your Application

In this chapter, you will learn the following topics:

- Basic safety – turning off `autopublish`
- Basic safety – removing `insecure`
- Securing data transactions with `allow` and `deny`
- Hiding data with façades
- Protecting the client with `browser-policy`

Introduction

Meteor makes development and prototyping as fast and easy as possible. To accomplish this, there are some default packages installed that have no business in a production application. As you prepare your app for production, you will want to remove the packages that make prototyping easier, and replace them with some security best practices, to make your application more secure. In this chapter, we will go through the baseline security mechanisms needed to prepare an application for production.

Basic safety – turning off autopublish

Quickly and easily accessing your data saves you an enormous amount of time when you're prototyping! The `autopublish` package, which is installed by default in every newly-created Meteor app, enables you to quickly manage and access your data collections, so that you can churn out great code. When the time comes, however, broadcasting every field in every data collection is inefficient and unsecure. This recipe will show you the basics of removing the `autopublish` package, and implementing your own `publish/subscribe` code to keep your app working as intended.

Getting ready

We will create a very basic application, displaying simple text cards on the screen, and then show the effects of `autopublish` and `subscribe/publish` on those cards. To do this, we need to create our folder structure, add some basic templates, and add a bit of styling.

Project setup

In a terminal window, create your root project by entering the following commands:

```
$ meteor create secure-autopublish
$ cd secure-autopublish
$ rm secure-autopublish.*
$ mkdir {client,server,both}
$ meteor
```

Creating a basic template

In a text editor, create a file named `collections.js` in your `[project root]/both/` folder, and add the following line:

```
Cards_open = new Mongo.Collection('open');
```

Next, create a file named `[project root]/client/main.html` and add the following `<template>` and `<body>` declarations:

```
<body>
  <div class="container">
    {{> open}}
  </div>
</body>


<template name="open">
  <h3 id="new-open">open:</h3>
```

```
  {{#each opens}}
  <div class="card {{shared}}">
    <div class="label id">id</div>
    <div class="id">{{_id}}</div>
    <div class="label text">text</div>
    <div class="text">{{text}}</div>
    <div class="label owner">owner</div>
    <div class="owner">{{owner}}</div>
  </div>
  {{/each}}
</template>
```

We need to add just a little bit of logic for display and creation, and we're ready to move on to styles. Create a new file named `[project root]/client/templatehelpers.js`, and add the following `Template.helpers` and `Template.events` functions:

```
Template.open.helpers({
  opens: function(){
    return Cards_open.find({},{sort:{text:1}}).fetch();
  },
  shared: function(){
    return (this.shared? 'shared':null);
  }
});

Template.open.events({
  'dblclick #new-open' : function(e){
    e.preventDefault();
    var txt = 'open card# ' + Cards_open.find({}).count();
    Cards_open.insert({text:txt});
  },
  'click .text' : function(e){
    e.preventDefault();
    var shrd = (!this.shared);
    Cards_open.update({_id:this._id},{$set:{shared:shrd}});
  },
  'dblclick .id' : function(e){
    e.preventDefault();
    Cards_open.remove({_id:this._id});
  }
});
```

Adding CSS styling

We need just a touch of CSS to make things more visually appealing. Create a file named `[project root]/client/styles.css` and add the following CSS:

```
body {
  font-family: 'helvetica neue';
}
.card {
  display: inline-block;
  min-width:10rem;
  height: 10rem;
  border: 2px dashed #ccc;
  border-radius: 0.21rem;
  margin: 0.25rem 0.25rem;
  padding: 0.5rem;
  vertical-align: top;
}

.container {
  width:90%;
  margin: auto;
}

.shared {
  background-color: rgba(25, 121, 36, 0.36);
}

.label {
  font-weight: bold;
  margin: 0.2rem 0;
  padding: 0.1rem;
  padding-left: 0.3rem;
}

.label:hover {
  background-color: rgba(7, 180, 21, 0.76);
  border-radius: 0.2rem;
}
```

Your app should now be up and running. Navigate to `http://localhost:3000` in a browser, and double click on the **open:** label to create some new cards. Click on the **text** tag in a card to modify the sharing property (the card will turn green) and double click on the **id** tag to delete a card. Your screen, after playing with it a bit, will look simlar to the following screenshot:

If everything is copacetic, we're ready to remove the `autopublish` package.

How to do it...

Proceed with the following steps to turn off `autopublish`:

1. In a new terminal window (keep Meteor running!), navigate to your `[project root]` and enter the following command:

   ```
   $ meteor remove autopublish
   ```

The screen on your web page will now show no results, and if you do a count on the `Cards_open` collection, even if you have many records created, the count will come back as **0**:

2. We will now add `publish` and `subscribe`. The `Cards_open` collection still exists. Because we removed the `autopublish` package, however, the communication between the client and the server has been severed. To restore it, we need to add a `publish` method on the server, and a `subscribe` method on the client. Create a file named `[project root]/server/collections-server.js` and add the following `publish` function call:

```
Meteor.publish('open',function(){
  return Cards_open.find({});
});
```

3. Now, create a file named `[project root]/client/collections-client.js` and add the following `subscribe()` function call:

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

All done! You have successfully removed the `autopublish` package, and re-created the `publish/subscribe` calls necessary to allow the client to still see the `Cards_open` collection. Your browser should now display results properly when you create, modify, and delete using the clicks and double clicks mentioned previously.

How it works...

In a nutshell, `autopublish` checks to see what collections exist, and automatically writes your `publish` and `subscribe` function calls for you. It does this for every collection it can find, and is therefore neither performant nor secure.

By removing `autopublish`, we stopped the `publish` and `subscribe` functions from automatically being called. Because of this, we had to re-create those calls, creating a simple `publish()` call (on channel `'open'`) on the server, and a `subscribe()` call (on the same `'open'` channel) on the client.

Our `find()` statement in the `publish` function retrieves everything, which is inherently not secure or performant, but we will be fixing that in other recipes. The focus of this recipe was on how to remove the `autopublish` package, without affecting the functionality of our application.

See also

- The *Removing Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*

Basic safety – removing insecure

Right after removing `autopublish`, we will want to control how data is added, removed, and updated, and put in some security measures, as appropriate. To enable this level of control, we need to remove the appropriately-named `insecure` package. To restore functionality after removing the `insecure` package, we will need to utilize a basic `collection.allow` declaration. This recipe shows you how to do exactly that.

Getting ready

We will use the *Basic safety – turning off autopublish* recipe found in this chapter as our baseline. Once you have completed that recipe, make a copy of the `secure-autopublish` folder (note: you will need all subfolders, including the hidden `.meteor` folder), rename it to `secure-rm-insecure`, start your app using the `meteor` command in the terminal, and you will be ready to proceed.

How to do it...

Just like the previous recipe for `autopublish`, we simply need to remove the `insecure` package, and then restore functionality.

1. In a new terminal window (keep Meteor running!), navigate to the root of your project and enter the following command:

   ```
   $ meteor remove insecure
   ```

   Your application now disallows any client changes to the `Cards_open` collection. Try to add a new card, share a card, or delete a card, and you will be unable to do so. Whether through the UI using clicks and double clicks, or even programmatically through the web console, you will be unable to make any changes, as shown in the following screenshot:

2. Okay, now we need to get our super powers back! Open the `[project root]/server/collections-server.js` file, and add the following code to the bottom of the file:

```
Cards_open.allow({
  insert : function(userId,doc){
    return true;
  },
  update : function(userId,doc,fieldNames,modifier){
    return true;
  },
  remove : function(userId,doc){
    return true;
  }
});
```

After saving these changes, our `insert`, `update`, and `remove` capabilities have been restored. You can now add, modify, and delete as many cards as you would like, either through the UI or programmatically through the web console.

How it works...

The `insecure` package does almost exactly the same thing for data collection security that `autopublish` does for publish security – it finds every collection it can and automatically creates a `collection.allow` function for all functions (`insert`, `update`, and `remove`). By removing the `insecure` package, we prevented our `Cards_open` collection from allowing any client-side changes.

To remedy this, and to prepare for more granular security (see later recipes in this chapter for details), we called `Cards_open.allow()` and enabled all collection modifications by returning `true` for every checking function.

So, although the net security of our application hasn't changed, we are now prepared to modify our publishing and security settings to make our application production-ready.

See also

- The *Removing Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*

Securing data transactions with allow and deny

Properly configured, Meteor collections are quite secure. The granular control we have over what is allowed and what is not allowed enables us to secure our applications appropriately. In this recipe, you will learn how to use `allow` and `deny` to secure your collections and control access.

Getting ready

Using the *Basic safety – removing insecure* recipe found in this chapter, we already have an application with both the `autopublish` and `insecure` packages removed. Once we add and configure the appropriate user `accounts` packages, we will be ready to proceed.

Using a copy of the *Basic safety – removing insecure* recipe as a baseline, open a terminal window, navigate to your project root, and execute the following commands:

```
$ meteor add accounts-ui
$ meteor add accounts-password
```

If your app isn't already running, make sure to start it using the `meteor` command.

We now need to add the `loginButtons` template, and modify our `insert` statement, to add an `owner` property to each record.

Open your `[project root]/client/main.html` file and add the `loginButtons` template inclusion just below the `<body>` tag, as shown in the following example:

```
<body>
  {{> loginButtons}}
  <div class="container">
    ...
```

Next, open your `[project root]/client/templatehelpers.js` file and modify the `Template.open.events` insert logic to add `owner`, and only fire if there is a logged-in user. Your code changes should look as follows:

```
Template.open.events({
  'dblclick #new-open' : function(e){
    e.preventDefault();
    if (!Meteor.userId()) return;
    var txt = 'open card# ' + Cards_open.find({}).count();
    Cards_open.insert({text:txt , owner: Meteor.userId()});
  },
```

Finally, in your browser, create a new user, and make sure you are logged in as that user (the name of the user doesn't matter – we suggest using a fake one such as `user1@test.com`).

Now, whenever you create new cards, the **owner** section will be filled in with the logged-in user's unique ID, as shown in the following screenshot:

How to do it...

The current state of the application is not secure. Anybody can `insert`, `remove` and `update` any of the cards, even if they belong to another user! We are going to fix this situation by using the `collection.allow()` and `collection.deny()` declarations.

1. First, we will require a logged-in user for `insert`. Open your `[project root]/server/collections-server.js` file, locate the `Cards_open.allow()` function call, and make the following modification to the `insert` function declaration:

   ```
   Cards_open.allow({
     insert : function(userId,doc){
       return(userId!=null);
     },
   ```

   You will now no longer be able to create new cards while being logged out (you can test this if you would like).

2. Next, we will allow only the owner of a record to `update` or `remove` cards. In the same `collections-server.js` file, modify the `update` and `remove` function declarations as follows:

   ```
   update : function(userId,doc,fieldNames,modifier){
       return (doc.owner==userId);
   ```

```
},
remove : function(userId,doc){
  return (doc.owner==userId);
}
```

3. Save your changes, and test your new rules by logging in as a new user (create one if needed), trying to change results, trying to add new cards without being logged in, and so on. With these rules in place, only a logged-in user will be able to create new cards, and only the owner of a card can modify the card or delete it.

How it works...

All client-side attempts to alter collections in any way flow through two callbacks: `allow` and `deny`. In order for a collection change to be accepted by the server, the incoming change must receive one `true` response from one of the `allow` functions (for example, we return `true` on the `insert` function if the `userId!=null`), and must receive zero `true` responses from any of the `deny` functions.

In this particular case, we are doing a simple check on `insert`, under the `allow` callback, to make sure the user is logged in, which translates to `userId!=null`. For `update` and `remove`, the check is to see if the logged-in user is the owner/creator of the card, through `return` (`doc.owner==userId`).

There is no limit to the number of `allow` or `deny` callbacks you can declare, although it's usually best to consolidate them when possible, using a "pessimistic" security model (only allow what is needed, rather than allowing everything and only denying what is needed).

There's more...

The preceding `allow` rules work just fine, if only the UI is being used. However, it's possible for someone to open the console window and make direct collection manipulation calls, which could cause some problems.

First, our check for `insert` is only `userId!=null`. Any additional fields, or even a malicious `insert`, can be added via the command line, for example, let's say that I am in possession of the `userId` for another user (not hard to get, it's found in the `owner` field of each card). I could easily `insert` a card with something nasty in the text, or `update` the text and owner of an existing cart, so as to make it appear that another user was the one that created the note.

For example, If I am logged in as `user2@test.com`, and I know that the `userId` value for `user1@test.com` is `'8v2GGh98RrYfso92c'`, I can run the following command in the browser console, and potentially get `user1` in trouble:

```
> Cards_open.insert({text:'NSFW !@##%!!!',owner:'8v2GGh98RrYfso92c'})
```

We can handle this in several ways. We can either make our `allow` callback functions more complex, with multiple `if...else` statements, or we can use the `deny` callback to prohibit certain behavior. In `[project root]/server/collections-server.js`, create a new `deny` callback with the following code:

```
Cards_open.deny({
  insert : function(userId,doc){
   return (doc.owner!=userId);
  },
  update : function(userId,doc,fieldNames,modifier){
    return (fieldNames.length!=1 || !(~fieldNames.indexOf('shared')));
  }
});
```

For `insert`, if `doc.owner!=userId`, the `deny` callback will return `true`. For `update`, if an attempt is made to modify any field except shared, the `deny` callback will return `true` as well. Using these two callback functions, we have further tightened security, and taken away any console line shenanigans.

See also

For a breakdown of what's possible with `allow` and `deny`, consult the official Meteor documentation available at `http://docs.meteor.com/#/full/allow`.

Hiding data with façades

Some of our security (and performance) problems can be resolved through limiting access to certain fields and records in our data collections, for example, if the `owner` field of a record isn't sent to the client, a potential hacker will never be able to get the `userId` value of another user. Likewise, if only records belonging to a certain `userId`, or ones marked for sharing, are passed to the client, private records can stay private and visible only to the user that created them. This recipe will show you how to create a façade to limit fields and records being sent to the client.

Getting ready

Please complete the *Securing data transactions with allow and deny* recipe found in this chapter, including the additional `deny` callback functions found in the *There's more...* section. Once completed, and your Meteor app is running, you are ready to use this recipe.

How to do it...

We are going to modify the `publish` function on the server so that it only returns records that are owned by or shared with the logged-in user, and we will stop broadcasting the `owner` field.

1. Open the `[project root]/server/collections-server.js` file, locate the `Cards_open.publish` section, and make the following changes to the `Cards_open.find()` method:

```
Meteor.publish('open',function(){
  return Cards_open.find({$or:
           [ {shared:true},
             {owner:this.userId}
           ]
         },
         {fields:{owner:0}});
});
```

2. Now that the `owner` field is no longer visible on the client, we can remove the following two lines from our open template in the `[project root]/client/main.html` file:

```
<div class="label owner">owner</div>
<div class="owner">{{owner}}</div>
```

3. With those changes saved, any given logged-in user will only be able to see cards that have been created by that same user, or cards that have been shared. If you log in to two different browsers, with two different users, you will be able to see how sharing makes records visible to the other user, and vice versa. The following screenshot shows an example of two users sharing some records, and not sharing others:

How it works...

By modifying the selector and the `fields` option, we were able to limit the recordset being published to the client. The client, try as it might, will never be able to see any of the records excluded by the selector, nor any of the excluded `fields`, because the server simply isn't sending them when it is publishing.

Specifically, we used a `{$or: [...]}` selector to limit which records are published, by including only records created by the current user (`owner:this.userId`), or records that have been shared (`shared:true`). We used the `{fields:{owner:0}}` option, to return all `fields` except `owner`. This *blacklist* approach is less secure than a *whitelist* approach, but to keep this recipe simpler, we decided to tell the query what fields to exclude (optimistic) rather than what fields to include (pessimistic).

To whitelist, rather than blacklist, enumerate the fields you would like displayed, and pass a value of `1` with them (for example: `{text:1 , _id:1 , shared:1}`). All fields not specified will automatically not return with the query.

See also

- The *Filtering with MongoDB queries* recipe in *Chapter 4*, *Creating Models*

Protecting the client with browser-policy

Securing your database is pretty straightforward in Meteor, but what about client-side security? Meteor has you covered there as well, using standard `Content-Security-Policy` and `X-Frame-Options` safeguards. This recipe will walk you through adding the `browser-policy` package, and configuring basic client-side security.

Getting ready

We will create a brand new project as usual, but we will be keeping the default files, creating some *unsafe* scripting functionality along the way.

Scaffolding setup

In a terminal window, navigate to where your project root will be, and execute the following commands:

```
$ meteor create secure-client
$ cd secure-client
$ mkdir {client,server,both}
$ mv secure-client.* client/
$ meteor
```

Add CDN-hosted bootstrap

Visit the official Bootstrap `Getting Started` page, located at `http://getbootstrap.com/getting-started/` and scroll to the section marked as **Bootstrap CDN**. Copy the contents from that section, and insert them into the `<head>` block of your `[project root]/client/secure-client.html` file. When finished, your changes should look similar to the following code:

```
<head>
  <title>secure-client</title>
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.2/css/bootstrap.min.css">

  <!-- Optional theme -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.2/css/bootstrap-theme.min.css">

  <!-- Latest compiled and minified JavaScript -->
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/
bootstrap.min.js"></script>
</head>
```

Add inline and eval() scripts

While we have `secure-client.html` open, let's modify the template, adding some indicators, an inline script in an `href` attribute, and a new section for displaying `eval()` results. Modify your `hello` template so that it looks like the following code:

```
<template name="hello">
  <button>Click Me</button>
  <p>You've pressed the button
    <div class="badge">{{counter}}</div>
  times.</p>
  <p>Which is 1/2 of our eval value: {{dblCounter}}</p>
  <a href="javascript:alert('hax0rz!'); Meteor.call('dropTable');">
      <div class="btn btn-info">Bootstrap!</div>
  </a>
</template>
```

We have a bit of logic to add, so that the template will be displayed properly. First, we will create a simple collection called `Test`. Create a file named `[project root]/both/model.js` and add the following line:

```
Test = new Mongo.Collection('test');
```

Now, configure the server method `dropTable` to simulate someone erasing the database. Create a file named `[project root]/server/methods.js` and add the following code:

```
Meteor.methods({
  dropTable: function(){
   Test.remove({});
  }
});
```

Next, we need to modify the `hello` template helpers and events to be "vulnerable" to our clever hacks! Open `[project root]/client/secure-client.js`, modify the `Template.hello.helpers` section to the following:

```
Template.hello.helpers({
  counter: function () {
    try {
      var x = Test.find().count();
      Session.set('counter', eval("x*2"));
    } catch (err) {
      console.log('ERROR: ', err);
    }
    return x;
  },
  dblCounter: function () {
    return Session.get('counter');
  }
});
```

Finally, modify `Template.hello.events` to add a record to the `Test` collection, rather than updating the `counter` variable. Your code should look similar to the following:

```
Template.hello.events({
  'click button': function () {
    // increment the counter when button is clicked
    Test.insert({action:'click'});
  }
});
```

Once all of these changes are saved, our application is thoroughly "hacked" with an `eval()` being used to double the normal click counter, an inline script that will remove all records from our `Test` collection, and with scripts and styles being used from an alternate site (the Bootstrap CDN).

Navigate to `http://localhost:3000/` and play around with the buttons for a bit. After a few clicks, your screen will look similar to the following screenshot:

To activate the inline "hack", click the button labelled **Bootstrap!** – a notice will come up that you've been hacked, and after you click on **OK,** the `Test` collection will be cleaned out. The notice will look something like the following screenshot:

You're now ready to shut down all these expert hacking techniques!

How to do it...

To secure our application, we will add the `browser-policy` package, and then configure it appropriately for our environment.

1. In a new terminal window, navigate to the root folder of your project (keep your app running!) and execute the following command:

   ```
   $ meteor add browser-policy
   ```

Your application will now have lost all Bootstrap formatting, and the `eval()` function which was doubling your counter and the eval() function, which was doubling your counter, will no longer work. When you click the **Click Me** button, your counter will increment but the double counter will not. Your screen should look similar to the following screenshot, with a lot of errors in the web console explaining that the unsafe "hacks" from before are no longer allowed:

2. We will now fine-tune our security settings with `BrowserPolicy.content`. The inline scripting vulnerability still exists (click on **Bootstrap!** if you would like to test it), and our formatting, which we don't consider a hack, is no longer working. So, we are currently not strict enough in one area (inline scripts) and too strict in another (refusing all content from the Bootstrap CDN, a trusted source). Let's rectify that. Create a new file named `policy.js` in the `[project root]/server/` folder. Add the following two lines, and save your changes:

```
BrowserPolicy.content.allowStyleOrigin(
'https://maxcdn.bootstrapcdn.com/');
BrowserPolicy.content.allowScriptOrigin(
'https://maxcdn.bootstrapcdn.com/');
```

3. Our Bootstrap formatting has been restored! Now, let's disallow inline scripts, as well as prevent connections to any servers. Add the following two lines to the `policy.js` file and save your changes:

```
BrowserPolicy.content.disallowInlineScripts();
BrowserPolicy.content.disallowConnect();
```

   The inline script that was erasing our `Test` collection will now no longer run. However, by disallowing all connections, we have inadvertently destroyed the `DDP` connection to our server. We need to rectify that by whitelisting our `//:localhost:3000` address for HTTP and for websockets (for `DDP`).

4. Add the following three lines to the end of the `policy.js` file and save your changes:

```
var rootUrl = __meteor_runtime_config__.ROOT_URL;
BrowserPolicy.content.allowConnectOrigin(rootUrl);
BrowserPolicy.content.allowConnectOrigin
(rootUrl.replace('http', 'ws'));
```

> As we are dealing with your browser's security policy, a manual refresh is required each time you make a change to `policy.js`.

   Refresh your browser one final time, and now everything that should be allowed is working, and everything that shouldn't (inline scripts, `eval()`, etc.) is prohibited. Your screen, after a few clicks, should look like the following screenshot:

How it works...

By default, installing the `browser-policy` package will disable the `eval()` scripts, and only allow resources (images, etc) from your site. This is why just adding `browser-policy` disabled the double counter `eval()` script, and stripped away all the Bootstrap files.

To allow Bootstrap resources, we whitelisted the styles and scripts of the Bootstrap CDN, using the two `BrowserPolicy.content` functions – `allowStyleOrigin()` and `.allowScriptOrigin()`.

Next, we shut off inline scripts with the `disallowInlineScripts()` function. We also prevented any and all AJAX / remote server calls, using the `disallowConnect()` function.

Being this strict also broke our server's DDP connection, which we restored using the `allowConnectOrigin()` function, for both HTTP and `ws`.

See also

There are many other options available to us (this is just a primer), so if you would like to really fine-tune your security settings, we recommend the most excellent article from Arunoda Susiripala, located at `https://meteorhacks.com/xss-and-meteor.html`, and the `browser-policy` documentation, found on Atmosphere: `https://atmospherejs.com/meteor/browser-policy`.

10
Working with Accounts

In this chapter, you will learn the following topics:

- Implementing OAuth accounts packages
- Customizing the accounts login
- Performing two-factor authentication

Introduction

Critical to nearly every application we build is some type of authentication and user identification. Usually, we will spend days and weeks developing an accounts system, when we could spend that time programming our app. Meteor solves this problem, and solves it well. From integrations with major OAuth providers (Twitter, Google, Facebook, etc.) to a simple, secure password-based system, adding accounts and authentication inside of your Meteor app is quick and painless. The recipes in this chapter will cover the most important aspects of Meteor's `accounts` packages, enabling you to easily take care of user accounts and move on to other things.

Implementing OAuth accounts packages

Today, there are so many popular authentication services available, with such large user bases, that it's kind of silly not to take advantage of those services. If you use the accounts system of a major service, like Twitter or GitHub, you instantly tap into an enormous user base which can increase the use of your app. This recipe will show you how to implement an OAuth accounts system in a Meteor app, using the Twitter accounts service as an example.

Getting ready

We are going to focus almost exclusively on the accounts and authentication piece of our application, and as such, we only need a very simple, baseline application.

In a terminal window, create your root project by entering the following commands:

```
$ meteor create twitter-login
$ cd twitter-login
$ mkdir {client,server,both}
$ mv twitter-login.* client/
$ meteor add twbs:bootstrap
$ meteor
```

That's all it takes. Everything else will be done inside our recipe, so let's get going!

How to do it...

We will add the appropriate `accounts` packages, and configure our Twitter login service. Proceed with the following steps:

1. In a new terminal window (keep your application running), navigate to the root folder of your project, and enter the following two commands:

   ```
   $ meteor add accounts-twitter
   $ meteor add accounts-ui
   ```

   This will add several dependent packages, so you don't have to. How thoughtful and gentlemanly of Meteor to do that for us!

   You can just as easily use another authentication service by replacing `accounts-twitter` used in this step with any of the following: `accounts-facebook`, `accounts-github`, `accounts-google`, `accounts-meetup`, `account-weibo`, or `accounts-meteor-developer`.

2. Open `[project root]/client/twitter-login.html` and add the following template inclusion just after the starting `<body>` tag:

   ```
   <body>
     {{> loginButtons}}
   ```

3. Save this change, and navigate to your app in a browser (usually `http://localhost:3000/`). At the top-left of your screen you will see a red button that says **Configure Twitter Login**, as shown in the following screenshot:

4. Click on **Configure Twitter Login** button, and a set of instructions for configuring your Twitter app will appear. Follow those instructions *exactly*, entering your `consumerkey` (API key) and consumer `secret` key(API secret) where appropriate.

> For this recipe, the value you enter in the website field is going to be `http://127.0.0.1:3000` rather than `http://localhost:3000`. Twitter doesn't allow the use of `localhost`.

5. With your service configured, you are ready to log in. Click on the blue (formerly red) button labeled **Sign In** with Twitter, and a new window should pop up, asking you to authorize your app, as shown in the following screenshot:

6. Once you authorize the app, you should be logged in, indicated by a status change in the login button, as shown in the following screenshot:

How it works...

This recipe is pretty packaged up, so there's not a lot of code involved. That said, we can dissect what's going on, so that we can understand where to tweak this recipe when needed.

Let's do a deep dive into how the authentication piece is working. `accounts-twitter` relies on the `accounts-base`, `accounts-oauth` and `twitter` packages. If we crack open each of those packages, we can see just how much legwork has been done for us:

Put as simply as possible, `accounts-ui` and `accounts-twitter` give you a responsive UI, and make calls to `accounts-base`, which handles all of the account administration. `accounts-oauth` helps `accounts-base` by handling the OAuth-specific events and calls. `accounts-oauth` is configured by the `twitter` package, which provides specific URLs and parameters needed to use Twitter's OAuth service.

Here's a more detailed explanation:

The `accounts-base` package is a generic accounts package that accepts different types of login methods, provides helper methods for account administration, and helps maintain the `users` collection. The `users` collection is where we store logged-in state, preferences, and profile information.

One of the exposed methods in `accounts-base` is `Accounts.registerLoginHandler()`, which can be used by more specific login packages (such as `accounts-oauth` or `accounts-password`) to register handlers for login information:

When `accounts-base` gets a request for a login, that request has a certain `type`, and has certain `service` parameters. `accounts-base` runs the information through all the login handlers that were registered, and lets each of those handlers respond with either `undefined` ("this is not my login method"), `error` ("the credentials were wrong"), or `serviceData` object ("login was accepted"), including a token for reconnecting easily.

The `accounts-oauth` package builds off of the `accounts-base` by registering a login handler of type: `'oauth'`, and exposing some helpers of its own. The `accounts-oauth` helpers allow us to configure a specific OAuth service. Each OAuth service requires customized URLs and parameters. We chose the Twitter OAuth service, and therefore used the `twitter` and `accounts-twitter` packages to configure those URLs and parameters.

The `accounts-oauth` packages is also responsible for handling the message/callback from the OAuth service, via the popup authentication form. That is, when the popup authentication from Twitter was complete, it redirected to `http://localhost:3000/_oauth/twitter` and had an OAuth token to pass to our app. The `accounts-oauth` package evaluates that particular URL (because we configured it with the `twitter` package), snatches up the token, and then attempts a login, using the `Accounts.callLoginMethod()` with some JSON that looks similar to the following:

```
{
methodArguments: [{oauth: {
credentialToken: "m3OHQUrRWU34anuq40Bx3q7JBoEmVgwKGICU1jY4H7_"
credentialSecret: "2qPoqew8m-AXiC2OVfrkWem0_M_APcdMpnz-cGsl6-k"
}},
...
]}
```

The login handler that `accounts-oauth` registered receives this JSON, and confirms with the (Twitter) OAuth service that the token is valid. If it is valid, a `user` profile is created/updated in the `users` collection, and `serviceData` (containing a login token, among other things) is passed to the client via a callback function. Because of the callback, and because the client is subscribing to the `users` collection, the client sees that there is a logged-in user, and acts accordingly.

The `twitter` and `accounts-twitter` packages contain convenience methods that work on top of `accounts-base` and `accounts-oauth`. For example, the `twitter` package has a server file (`twitter_server.js`) that declares the specific Twitter URLs, registers the `twitter` service through the `OAuth.registerService()` method, and even creates a Twitter-specific credential call, as shown in the following code:

```
Twitter.retrieveCredential =
 function(credentialToken, credentialSecret) {
    return OAuth.retrieveCredential(
      credentialToken, credentialSecret);
 };
```

The `accounts-twitter` packages creates the `Meteor.loginWithTwitter()` method, and declares which profile fields are visible on the client, using `Accounts.addAutoPublishFields()`.

There's more...

Sometimes, seeing how the user and configuration information is stored can be helpful in understanding what's going on under the hood. We can do this using the `meteor mongo` command in a terminal window. Open a new terminal window (keep your application running), navigate to your project root, and enter the following command:

```
$ meteor mongo
```

You will now have command line access to the collections that store user information and login service configurations.

To view the Twitter configuration settings, enter the following command:

```
> db.meteor_accounts_loginServiceConfiguration.find()
```

You will see the configuration for your Twitter login service, as shown in the following code:

```
{ "service" : "twitter",
  "consumerKey" : "th2is2i2safa333kecon442sume24r433key",
  "secret" : "th9isi9sa9fa87kesecr666e3t",
  "loginStyle" : "popup", "_id" : "DBfakeYnnFbmidbC"
}
```

If you would like to reconfigure your Twitter login service, you can remove the entry using the following command:

```
> db.meteor_accounts_loginServiceConfiguration.remove({})
```

Once you do this, you can follow the instructions on the screen and re-enter your Twitter credentials as you did in the preceding recipe.

To view the different states of a logged in user, run and re-run the following command:

```
> db.users.findOne()
```

Go ahead and experiment with this, running it when the user is logged out, when the user is logged in, and when the user doesn't exist yet. Pay special attention to the `services` section and you'll be able to see how logins are handled by both the `twitter` and `resume` login services.

See also

- The *Customizing the accounts login* recipe in this chapter
- The *Building custom server methods* and *Handling asynchronous events* recipes in *Chapter 11*, *Leveraging Advanced Features*

Customizing the accounts login

Packaged accounts logins are great and all, but they don't always go with the design of the rest of our page, or they provide too much functionality when all we need is a little functionality. This recipe will show you how to customize Meteor's accounts packages using the Twitter OAuth service as an example.

Getting ready

We will essentially be using the *Implementing OAuth accounts packages* recipe found in this chapter as our baseline, but we aren't going to add the `accounts-ui` package, and therefore will not be configuring the Twitter service through the UI, so we need to roll our own.

In a terminal window, create your root project by entering the following commands:

```
$ meteor create twitter-custom
$ cd twitter-custom
$ mkdir {client,server,both}
$ mv twitter-custom.* client/
$ meteor add twbs:bootstrap
$ meteor
```

Open a new terminal window (keep your app running) and add the `accounts-twitter` and the `service-configuration` packages:

```
$ meteor add service-configuration
$ meteor add accounts-twitter
```

We now need to configure our login service manually, using the API key and API secret from our existing Twitter service (the one we created with the *Implementing OAuth accounts packages* recipe). Create a file named `[project root]/server/auth-init.js` and add the following code, replacing the appropriate sections with your key and secret:

```
ServiceConfiguration.configurations.upsert({
  service:"twitter" },
  {
    $set: {
      "consumerKey" : "[your API Key from apps.twitter.com]",
      "secret" : "[your API secret from apps.twitter.com]"
    }
  }
);
```

When you copy your keys from the Twitter Apps page, there's usually an extra space character on the end. Make sure you remove that character (e.g. `"key123 "` needs to be `"key123"`) or your authentication will fail!

We are now ready to build our own customized login.

How to do it...

To build our own login, we'll need a couple of buttons, and some type of indicator that we're logged in. Easy peasy.

1. Open `[project root]/client/twitter-custom.html` and add the following template:

```
<template name="customLogin">
  {{#if currentUser}}
    <div id="logout" class="btn btn-info">Log out</div>
    <img src="{{profPic}}" alt="">
  {{else}}
    <div id="login" class="btn btn-default">Log in</div>
  {{/if}}
</template>
```

2. We now want to call our template. Make the following changes inside the `<body>` tag, and save the file:

```
<body>
  <div class="container">
    {{>customLogin}}

    <h1>Welcome to Meteor!</h1>

    {{> hello}}
  </div>
</body>
```

3. Create a new file named `[project root]/client/templatehelpers.js`, and add the following `customLogin` template helpers:

```
Template.customLogin.helpers({
  profPic: function(){
    var loggedin = Meteor.user();
    return loggedin &&
loggedin.services...url;
  }
});
```

4. Now, let's hook up our login and logout buttons. In the same `templatehelpers.js` file, add the following events declarations:

```
Template.customLogin.events({
  'click #login' : function(e){
    Meteor.loginWithTwitter();
  },
  'click #logout': function(e){
    Meteor.logout();
  }
});
```

5. Save your changes, and navigate to your app in a browser (usually `http://localhost:3000`). You should see a login button, as shown in following screenshot:

6. If you click the **Log in** button and authorize the Twitter app when prompted by the popup window, you will be authenticated and your Twitter avatar will appear, next to a **Log out** button, as shown in the following screenshot:

How it works...

Starting in the `customLogin` template, we make use of the `{{#if currentUser}}` helper, which checks to see whether `Meteor.user()` is `null` or not. In other words, if a user is logged in, `currentUser` returns `true`.

If `currentUser` is `true`, we add a **Log out** button, and an `<img>` tag, with the `src` attribute set to a property found on the user profile. Specifically, `profPic` returns the `services.twitter.profile_image_url` property if a user is logged in.

For the login and logout events, we simply call the `Meteor.loginWithTwitter()` (provided by the `accounts-twitter` package) and the `Meteor.logout()` methods. Meteor takes care of the rest for us.

There's more...

The accounts interface itself is quite customizable, with many packages available at `https://atmospherejs.com/?q=accounts-ui`.

We suggest installing the `accounts-ui-unstyled` package and experimenting with CSS/styling. You can get a great overview of what options and DOM elements are available by checking out the raw repository available at `https://github.com/meteor/meteor/tree/devel/packages/accounts-ui-unstyled`.

Pay particular attention to the `login_buttons.html` and `login_buttons.js` files, as they'll give you some pointers on what's possible.

See also

- The *Implementing OAuth accounts packages* recipe in this chapter

Performing two-factor authentication

We can make any application more secure (and safer from bots or hack attempts) by providing two-factor authentication. Two-factor authentication requires an individual to verify their identity using two separate methods. One such method, SMS text verification, has become quite popular, due to its convenience and difficulty in mimicking. This recipe will show you how to create two-factor authentication in a Meteor app, using the Twitter OAuth and Twilio SMS services.

Getting ready

Using the *Customizing the accounts login* recipe found in this chapter, we already have an application that authenticates against Twitter. We will expand that recipe, and add the Twilio SMS service to send a 6-digit verification code for our second authentication challenge. So that we can focus on the authentication part of the recipe, we will set up the Twilio service here rather than in the main recipe.

Creating our baseline application

Please follow the *Customizing the accounts login* recipe found in this chapter, and recreate that project, changing the name from `twitter-custom` to `two-factor`.

Signing up for the Twilio SMS service

Navigate to `https://www.twilio.com/try-twilio` in a browser, or visit the home page at `https://www.twilio.com/`, and click on the **SIGN UP** link on the top right.

Enter the necessary information to create an account, and click **Get Started**.

Verify that you're human by entering your phone number and clicking on **Text Me**. Shortly thereafter, you will receive a text message (if not, you can retry). Enter the code from that message into the verify section, and click on the **Verify** button.

You will now have a phone number generated for you. Accept the default number, or choose one for yourself (the default one is free), and click on the **Go To Your Account** button.

> Make a note of your assigned phone number, as you will need it later in this recipe.

Congratulations, you're all set up with a trial account from Twilio!

Creating an SMS service on Twilio

While logged in to Twilio, click on your name in the top right corner of the page. Select the **Account** option to go to the **Account Settings** page. There you will be presented with two sets of API keys. You can use either of those for testing, but would obviously want to use the Live credentials for a production application. Your screen should look like the following screenshot:

Decide which credentials you'll be using, and make note of both the **AccountSID** and the **AuthToken**.

Installing the twilio-node npm package

Stop your application (*Ctrl* + *C*) in the terminal, and enter the following command:

```
$ meteor add meteorhacks:npm
```

Run your application again by entering the following command:

```
$ meteor
```

You will receive a message letting you know that `meteorhacks:npm` has been initialized, as shown in the following example:

```
-> npm support has been initialized.
-> please start your app again.
```

Before starting our app again, we will need to declare that we are going to use the `twilio-node` npm package. Open the newly-created `[project root]/packages.json` file, and add the following declaration:

```
{
  "twilio":"1.10.0"
}
```

Now, start your application back up again in the terminal, using the `meteor` command, as shown in the preceding examples.

With the `twilio-node` npm package installed, all that's left to do is create a Twilio messaging method and test it.

Creating and testing the sendTwilio() method

Create a new file named `[project root]/server/twilio.js` and add the following code, replacing the `AccountSID`, `AuthToken`, and `Twilio Phone Number` as appropriate:

```
sendTwilio = function (phone, message) {
  return Meteor.wrapAsync(function (phone, message, callback) {
    var Twilio = Meteor.npmRequire('twilio')
      ('[YOUR AccountSID GOES HERE]',
        '[YOUR AuthToken GOES HERE]');
    var phoneNum = '+1' + phone;
    var twilioPhone = '[TWILIO NUMBER PATTERN: +1NUMBER]';

    Twilio.sendMessage({
      to: phoneNum,
      from: twilioPhone,
      body: message
    }, function (err, msg) {
      if (err) {
        callback && callback(err);
      } else {
        callback && callback(null, msg);
      }
    });
  })(phone, message);
};
```

A good chunk of time could be spent explaining what all the preceding code does, but sufficeth to say, the `Twilio.sendMessage()` method gets called using `Meteor.wrapAsync()` with a callback, because Twilio is an npm module and therefore requires the wrapper. You should now be able to send messages to your phone, which you can test by opening a new terminal window (keep your app running), navigating to your root folder, and using the `meteor shell` command to test.

If your mobile number were `555.867.5309`, you would enter the following:

```
$ meteor shell
> sendTwilio('5558675309','test',function(e,d){console.log(e,d);})
```

If everything is set up properly, you will get a text message on your phone, similar to the following screenshot:

If something went wrong, the console will spit out an error message, and you can trace down the source of your error.

Hopefully everything went right, and we are now ready to complete our two-factor authentication recipe!

How to do it...

We'll start by getting everything running smoothly on the server side, incorporating a new login state called `verified`. Once the server side is complete, we'll then build out the different user states in our UI.

We will need to generate a 6-digit code to be sent to the user, and we can leverage an existing Meteor package to do so.

1. In a new terminal window (keep your app running), in the project root, enter the following command:

   ```
   $ meteor add random
   ```

2. Now, create a file named `[project root]/both/helpers.js` and add the following code:

```
Random.digits = function(len){
  var numArr = [0,1,2,3,4,5,6,7,8,9];
  var ret = '';
  while(ret.length<len){
    ret+=Random.choice(numArr);
  }
  return ret;
};
```

3. We've just extended the random package to spit out simple codes of any length we choose. You can test this if you would like by entering `Random.digits(6)` in the web console.

4. Open `[project root]/server/auth-init.js` and append the following declaration:

```
var verifiedField = ['services.twofactor.verified',
                     'services.twofactor.phone'];
Accounts.addAutopublishFields({
  forLoggedInUser: verifiedField,
  forOtherUsers: []
});
```

5. Now, to make sure that the `services.twofactor.verified` property exists on each account, we will initialize it both when an account is created, and when a user logs in successfully. Append the following code to `auth-init.js`:

```
Accounts.onCreateUser(function(options,user) {
  check(options, Object);
  check(user, Object);
  user.services.twofactor = {};
  user.services.twofactor.code = Random.digits(6);
  user.services.twofactor.verified = false;
  user.profile = options.profile;
  return user;
});

Accounts.onLogin(function(options){
  if (options.type!=='resume'){
    Meteor.users.update(
      options.user._id,
      {$set:
       {"services.twofactor.verified":false,
        "services.twofactor.code":Random.digits(6)
       }
```

```
      }
    );
  }
});
```

6. Everything is in place for us to now create the SMS authentication challenge. Create a new file named `[project root]/server/auth-methods.js` and add the following `Meteor.methods` declarations:

```
Meteor.methods({
  sendChallenge : function (phone){
    if (!this.userId) return;
    var newCode = Random.digits(6);
    if (phone!=null){
      Meteor.users.update(
        this.userId,
        {$set:
         {"services.twofactor.phone":phone,
          "services.twofactor.code":newCode}});
    } else {
      Meteor.users.update(
        this.userId,
        {$set:
         {"services.twofactor.code":newCode}});

    }
    var curUser = Meteor.users.findOne(this.userId);
    return sendTwilio(curUser.services.twofactor.phone, curUser.
services.twofactor.code);
  },
  verifyCode : function(code){
    if (!this.userId) return;
    var curUser = Meteor.users.findOne(this.userId);
    if (!curUser) return;
    if (curUser.services.twofactor.code == code){
      Meteor.users.update(
        this.userId,
        {$set:
         {"services.twofactor.verified":true}});
    }
  }
});
```

7. With everything on the server set, we now need to update our templates and events on the client side. Let's first create some kind of visual sign that we've successfully authenticated. Open `[project root]/client/two-factor.html` and modify the `<button>` element as follows:

```
<button class="btn {{btnState}}">Click Me</button>
```

8. Now open `two-factor.js` in the same folder and make the following addition to the `Template.hello.helpers` declaration:

```
counter: function () {
      return Session.get('counter');
    },
    btnState: function(){
      var curUser =Meteor.user();
      if (curUser && curUser.services.twofactor.verified)
        return 'btn-success';
      return 'btn-danger';
    }
```

9. Finally, make the following changes to the `'click button'` event handler:

```
'click button': function () {
      // increment the counter when button is clicked
      var curUser =Meteor.user();
      if (curUser && curUser.services.twofactor.verified) {
        Session.set('counter', Session.get('counter') + 1);
      } else {
        alert ('not authorized!');
      }
    }
```

10. Everything else is done. We now only need to provide a way to make our server calls. Open `[project root]/client/two-factor.html` and make the following changes to the `customLogin` template:

```
<template name="customLogin">
  <div class="btn-toolbar">
    <div class="btn-group" role="group">
      {{#if currentUser}}
      <div type="button" id="logout"
      class="btn btn-info btn-lg">Log out</div>
      <div id="profile" class="btn btn-default btn-lg">
       <img src="{{profPic}}" alt="">
      </div>
      {{else}}
      <div type="button" id="login"
      class="btn btn-default btn-lg">Log in</div>
      {{/if}}
```

```
    </div>
    {{#if currentUser}}
    {{>secondLogin}}
    {{/if}}
  </div>
</template>
```

11. We now need to create the `secondLogin` template, with conditionals based on whether the user is verified or not. Append the following template to the bottom of `two-factor.html`:

```
<template name="secondLogin">
  {{#if verified}}
  <div class="btn btn-success btn-lg">
    <span class="glyphicon glyphicon-ok"></span>
  </div>
  {{else}}
  <div class="btn-group" role="group">
    <div class="btn btn-primary btn-lg" id="btnChallenge">
      <span class="glyphicon glyphicon-phone"></span>
    </div>
    <input type="text" id="phoneNum"
    class="btn btn-default btn-lg"
    placeholder="{{defaultPhone}}">
  </div>
  <div class="btn-group" role="group">
    <input type="text" id="verCode"
    class="btn btn-default btn-lg"
    placeholder="code...">
    <div class="btn btn-primary btn-lg" id="btnVerify">
      <span class="glyphicon glyphicon-check"></span>
    </div>
  </div>
  {{/if}}
</template>
```

12. We have the tiniest bit of CSS to add to make our profile pic behave. Open the file named `[project root]/client/two-factor.css` and add the following CSS declarations:

```
#profile img {
  max-height: 44px;
  margin: 0 0;
}

#profile {
  padding: 0 0;
}
```

13. Open `[project root]/client/templatehelpers.js` and add the following helpers:

```
Template.secondLogin.helpers({
  verified: function(){
    var curUser = Meteor.user();
    return (curUser&&curUser.services.twofactor.verified);
  },
  defaultPhone: function(){
    var curUser = Meteor.user();
    return curUser && curUser.services.twofactor.phone;
  }
});
```

14. Last of all, we need to add the event handlers for the buttons to send SMS texts and to verify the code found in the SMS messages. Append the following code to `templatehelpers.js`:

```
Template.secondLogin.events({
  'click #btnChallenge' : function (e){
    var phoneNum = $('#phoneNum').val();
    if (!phoneNum.length)
      phoneNum = $('#phoneNum').attr('placeholder');
    if (!phoneNum.length==10) return;
    Meteor.call('sendChallenge',phoneNum);
  },
  'click #btnVerify' : function(e){
    var verCode = $('#verCode').val();
    if (!verCode.length==6) return;
    Meteor.call('verifyCode',verCode);
  }
});
```

15. Save all your changes, and go ahead and test your new UI. Upon authenticating via Twitter, you will receive two text prompts and two buttons, as shown in the following screenshots:

The button on the left will send a randomized code to the phone number you specify (or the saved phone number, if one exists). The button on the right will submit a verification code. If you entered the correct code (found in the text message sent to your phone), you will be verified, and your screen will look similar to the following screenshot:

Congratulations, you've just implemented two-factor authentication in your Meteor app!

How it works...

Several things had to be accomplished in order for two-factor authentication to work. First, we had to extend the users collection with a new service named `twofactor`. Some of the `services.twofactor` properties needed to be exposed for use on the client, and we had to set `services.twofactor.verified` to `false` whenever a new user was created, or when a user logs in using Twitter OAuth. Inside of our `auth-init.js` file, we accomplished both of these tasks, first by calling the `Accounts.addAutopublishFields()` method, and then by listening and updating via the `Accounts.onCreateUser()` and the `Accounts.onLogin()` event handlers:

```
Accounts.addAutopublishFields({
  forLoggedInUser: verifiedField,
  forOtherUsers: []
});
Accounts.onCreateUser(function(options,user) {
  ...
  user.services.twofactor.code = Random.digits(6);
  user.services.twofactor.verified = false;
  ...
});
  Accounts.onLogin(function(options){
    ...
        {$set:
         {"services.twofactor.verified":false,
          "services.twofactor.code":Random.digits(6)
         }
    ...
  });
```

We added two server methods to helps us with verifying the user. The first, `sendChallenge()`, generates a new 6 digit random code, updates the `services.twofactor.code` property, and then sends the code to the specified phone number via the Twilio service. The second, `verifyCode()`, receives manual input from the user, checks the manually entered code against the `services.twofactor.code` property, and updates `services.twofactor.verified` to `true` if they match:

```
Meteor.methods({
  sendChallenge : function (phone){
    if (!this.userId) return;
    var newCode = Random.digits(6);
    ...
    return sendTwilio(...);
  },
  verifyCode : function(code){
    ...
    if (curUser.services.twofactor.code == code){
      Meteor.users.update(
        this.userId,
        {$set:
         {"services.twofactor.verified":true}});
    }
  }
});
```

With the addition of the `verified` property, and the server methods used to change the `verified` property from `false` to `true`, we can now use `verified` in our UI. We created a helper method, `Template.secondLogin.verified`, that checks to see if the `services.twofactor.verified` property is set to `true`. We then use this helper in our `secondLogin` template to show that the user is logged in and verified:

```
<template name="secondLogin">
  {{#if verified}}
  <div class="btn btn-success btn-lg">
    <span class="glyphicon glyphicon-ok"></span>
  </div>
  {{else}}
    ...
  {{/if}}
</template>
```

The rest of the event handlers and helpers on the UI are used for convenience, or to make calls to the server methods previously outlined.

The simplified version is this: we extended the `users` collection, adding a `verified` property. We used the SMS text and verification to change the value of the `verified` property. We disallow any activity in the client UI unless `verified==true`. This check, as a complement to checking if there is a logged in user, allows us to require two-factor authentication in our UI.

There's more...

This model of extending the `users` collection with new `services` properties can be used for pretty much anything, and is not just limited to SMS text challenges. By adding and exposing new `services` properties, you can control what features are available, depending on the user status. Imagine being able to limit some features of the UI, based on the subscription plan a user has purchased. Or imagine remembering layout and view preferences based on the saved preferences of the user. All of this, and more, is possible by extending the `users` collection.

See also

- The *Building a smooth interface with Bootstrap* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Using npm packages directly* recipe in *Chapter 8*, *Integrating Third-party Libraries*
- The *Customizing the accounts login* recipe in this chapter
- The *Using asynchronous functions* recipe in *Chapter 11*, *Leveraging Advanced Features*

11

Leveraging Advanced Features

In this chapter, we will cover the following topics:

- Building custom server methods
- Creating custom EJSON objects
- Handling asynchronous events
- Using asynchronous functions

Introduction

There are some unique functions Meteor offers that make our lives in a full-stack development environment even easier. This chapter will go into these lesser-advertised areas of functionality, providing you with even more under the hood knowledge to make your applications elegant and powerful.

Building custom server methods

The majority of communications we perform between the client and server in Meteor is done via DDP. There are times, however, where direct server calls can come in very handy. For example, while you're testing, or as an admin, you may create some `'hidden'` helper methods to make debugging easier. In other instances, you may want to make very secure or very simple methods available to reduce the vulnerability or complexity of your code. In any case, server methods are a staple of development, and this recipe will walk you through creating and using server-side methods, using the `Meteor.methods()` function.

Getting ready

To create server methods, we first need a basic application. We'll quickly create a color swatch app. Nothing fancy, but once we're done, we will be able to create server methods with good visual feedback.

Project setup

You will need Meteor installed, and have a project created. In a terminal window, create your root project by entering the following commands:

```
$ meteor create server-calls
$ cd server-calls
$ rm server-calls.*
$ mkdir {client,server,both}
$ meteor add twbs:bootstrap
$ meteor
```

Creating a simple app

Create a file named `[project root]/both/helpers.js` and add the following code:

```
Swatches = new Mongo.Collection('swatches');

randomColor = function(){
  var retCol = '#';
  while (retCol.length<4) {
    retCol += Random.choice('06F');
  }
  return retCol;
}
```

Next, create a file named `[project root]/client/client.js` and add the following code:

```
Template.colors.helpers({
  swatches: function(){
    return Swatches.find().fetch();
  }
})

Template.body.events({
  'click #newColor' : function(e){
    Swatches.insert({color:randomColor()});
  }
})
```

Create some simple styling by creating a file named `[project root]/client/styles.css` with the following style declarations:

```
.swatch{
  display:inline-block;
  height:8rem;
  width:8rem;
  border-radius: 0.5rem;
  margin-top: 1rem;
}

#newColor{
  display:block;
  margin-top: 0.5rem;
}
```

Finally, create your templates by creating `[project root]/client/main.html` and adding the following templates:

```
<body>
  <div class="container">
    <div id="newColor" class="btn btn-info btn-lg">
      <span class="glyphicon glyphicon-plus"></span>
    </div>
    {{> colors}}
  </div>
</body>


<template name="colors">
  {{#each swatches}}
  <div class="swatch" style="background-color:{{color}}"></div>
  {{/each}}
</template>
```

Save all your changes, navigate to `http://localhost:3000` in a browser, and click the button with the plus sign on it repeatedly to add random color swatches. You should see something similar to the following screenshot:

You are now ready to create some helper server methods.

How to do it...

We will create two server methods. One to clear the `Swatches` collection, and one that adds only unique colors.

1. First, let's build the `clearSwatches()` function. Create a file named `[project root]/server/methods.js` and add the following `Meteor.methods` declaration:

```
Meteor.methods({
  clearSwatches: function(){
    Swatches.remove({});
  }
});
```

   In your browser console, enter the following command:

```
> Meteor.call('clearSwatches')
```

   All your color swatches should disappear. Don't fret, friend, you can always create more swatches by clicking the button on the screen.

2. Next, let's create the `addUniqueSwatches()` function. Open `methods.js` from the preceding step and add the following declaration just after the `clearSwatches` declaration:

```
clearSwatches: function(){
    Swatches.remove({});
  },
  addUniqueSwatch: function(newColor){
    if (Swatches.findOne({color:newColor})) return null;
    Swatches.insert({color:newColor});
  }
```

3. Now, open `[project root]/client/client.js`, and make the following change to the `Template.body.events` declaration:

```
Template.body.events({
  'click #newColor' : function(e){
    Meteor.call('addUniqueSwatch' , randomColor());
  }
})
```

Now, as you click on the button to add colors, you will find that, eventually, no new swatches will be added. This is because the total number of unique colors is 27. If you start with no swatches, your screen will eventually look similar to the following screenshot:

You won't be able to add more than the 27 colors, no matter how many times you click the button.

How it works...

We created two calls on the server by creating code inside the `server` folder. These calls are exposed to the client because we declared them using the `Meteor.methods()` function.

The `clearSwatches()` method we created in the `methods.js` file is *hidden*, in that, unless you know it's there, there's no easy way to know that it exists. The `addUniqueSwatch()` method is used inside of the `onclick` event for our button, so someone could discover it by looking at our client code.

In both cases, the methods are not called through an Ajax/traditional server call, but are instead automatically exposed by Meteor, and , very easy to invoke via the `Meteor.call()` method, which accepts the name of the method as the first parameter, and any additional parameters, as method arguments.

There's more...

If you pass a callback function as the last argument in `Meteor.call()`, the callback will be automatically invoked when the server method has completed execution. You can use this to perform actions after the call is complete. The following is an example.

Open `[project root]/server/methods.js` and make the following changes to `addUniqueSwatch()` function:

```
addUniqueSwatch: function(newColor){
    if (Swatches.findOne({color:newColor})) return null;
    Swatches.insert({color:newColor});
    return Swatches.find().count();
  }
```

Now, in your browser console, quickly reset the `Swatches` collection:

```
> Meteor.call('clearSwatches')
```

Finally, make the following manual call to `addUniqueSwatch()` multiple times:

```
> Meteor.call('addUniqueSwatch',randomColor(),
function(err,data){console.log(data);})
```

Each time you run that command, the console will print out the total number of swatches. Meteor is auto-running the callback you passed, populating the data parameter with the results from the `addUniqueSwitch()` method.

See also

- The *Using the web console* recipe in *Chapter 1*, *Optimizing Your Workflow*
- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*

Creating custom EJSON objects

It's pretty easy to pass simple objects between the client and the server using DDP. But did you know that you can pass custom, named objects complete with methods? The folks at Meteor have extended JSON to allow customized, complete objects to be passed over DDP, without you having to worry about serialization/deserialization. This recipe will teach you how to create and use custom EJSON objects, and pass those custom objects between the client and the server.

Getting ready

We will be using the previous recipe found in this chapter, *Building custom server methods*, as a baseline. Please complete that recipe, and then make the following modifications.

Declaring the Swatch object

Create a new file named `[project root]/both/swatch.js` and add the following code to the file:

```
Swatch = function (color){
  this.color = color;
}

Swatch.prototype = {
  constructor: Swatch,

  switch: function(){
    this.color = randomColor();
  },
  toString: function(){
    return "My color is: " + this.color;
  }
}
```

Modifying Swatches.insert()

In preparation for using EJSON objects, which are not the normal, plain objects MongoDB expects, we need to layer our objects by one level when inserting. In the web console, execute the following line to clear out the `Swatches` collection:

```
> Meteor.call('clearSwatches')
```

Now, open `[project root]/server/methods.js` and modify the `Swatches.insert()` method as follows:

```
addUniqueSwatch: function(newColor){
  ...
  Swatches.insert({swatch:new Swatch(newColor)});
}
```

Changing Swatch colors

The layering on insert will break our UI a bit, but that's okay, we can recover because we're awesome. Inside the `[project root]/client/client.js` file, in the `Template.colors.helpers` section, and just below the `swatches` helper, add the following helper:

```
},
color: function(){
    return this.swatch.color;
}
```

Lastly, let's add the following `events` declaration, so that we can change the color of our swatches:

```
Template.colors.events({
  'click .swatch' : function(e){
    this.swatch.color = randomColor();
    Swatches.update(this._id,this);
  }
})
```

Save your changes, and test in a browser by clicking on any of the added swatches. The corresponding swatch should change to a random color each time you click. You are now ready to proceed with creating custom EJSON objects.

How to do it...

As mentioned, Meteor takes care of the serialization and deserialization for us. We just need to declare our object as an EJSON object. Proceed with the following steps to create a custom EJSON object:

1. Let's add the `typeName` and `toJSONValue` functions. Open the `[project root]/both/swatch.js` file, and add the following two functions to the `Swatch.prototype` declaration, just below the `toString()` function:

```
},
typeName: function(){
  return 'Swatch';
},
toJSONValue: function(){
  return {
    color:this.color
  };
}
```

2. Next we need to declare our object with `EJSON.addType`. In the same `swatch.js` file, at the very bottom, add the following function call:

```
EJSON.addType("Swatch", function fromJSONValue(value){
  return new Swatch(value.color);
});
```

3. We're now ready to use our EJSON object methods. Open the `[project root]/client/client.js` file, and make the following changes to `Template.colors.events`:

```
'click .swatch': function (e) {
    this.swatch.switch();
    Swatches.update(this._id, this);
    console.log(this.swatch.toString());
  }
```

In the browser, add some new swatches and click on them with your browser console window open. The swatches will change colors, and they will tell you what their new colors are in the console. Your screen should look similar to the following screenshot:

How it works...

When known `EJSON` objects are inserted into, or retrieved from MongoDB, Meteor automatically serializes and deserializes the objects using the `EJSON` library.

To do this properly, the objects themselves have to have at least two methods. First, the object must have the `typeName` function, which aids in mapping the object to the declared `EJSON` object. Second, the object must have the `toJSONValue()` function, so that Meteor can properly serialize the object to a string/standard JSON value. Without the `toJSONValue()` function, MongoDB would choke on the object, and refuse to insert it. Transporting the object over DDP would be equally unfeasible.

The other action needed is to declare the `EJSON` object, which we did in `swatch.js` with the `EJSON.addType()` function. This function takes a `fromJSONValue()` function as an argument, which (exactly as it sounds) takes the JSON value passed over the wire and instantiates an actual object of the appropriate type. In our case, this was a `Swatch` object, and the only property needing to be created was the `color` property, which was passed in through the constructor.

Once the previous methods are declared, and the `addType` function called, Meteor takes care of the rest for us, and stores our objects rather cleverly in MongoDB. As an example, here is what the raw JSON looks like for one of our serialized `Swatch` objects:

```
{
 "_id" : "tktEzxMGTGNZ8oB4R",
   "swatch" : {
     "EJSON$type" : "Swatch",
       "EJSON$value" : {
         "EJSONcolor" : "#66F"
       }
   }
}
```

There's more...

The `EJSON` object has other helper functions that you can declare in an object to aid in the development and use of `EJSON` objects.

The `.clone()` method can be implemented on an `EJSON` object to perform a deep copy with logic. If you don't declare the `.clone()` function, Meteor uses `toJSONValue()` instead.

The `.equals()` method takes another object as a parameter and performs a custom comparison of your choosing. If you don't declare the `.equals()` function, Meteor simply takes both objects, performs a `toJSONValue()` transformation on each, and compares the result.

> Learn more about the EJSON library by viewing the Meteor documentation, found here: `https://www.meteor.com/ejson`.

See also

- The *Building custom server methods* recipe in this chapter

Handling asynchronous events

Meteor is a reactive framework. As you've probably noticed, it's built on top of `Node.js`, but when writing or using packages, it somehow magically avoids all of the callback and event loop drama you'd normally face with `Node.js`. Meteor lets you code in a declarative, synchronous-like style. This recipe will show you one of the ways Meteor does this, and how you can handle asynchronous events coming in from a third-party package. In this case, we will read an incoming Twitter stream from the npm `twit` package.

Getting ready

We need to quickly set up a test application on Twitter and load the npm `twit` module, so we'll do that here, so as not to distract from the recipe itself. Note that you can use any asynchronous event stream you would like for this recipe, including the standard `setInterval()` method, which could be used to mimic an asynchronous event stream.

Creating a baseline Meteor app

Open a terminal window, navigate to where you would like your project to reside, and enter the following commands:

```
$ meteor create thirdpartyevents
$ cd thirdpartyevents
$ rm thirdpartyevents.*
$ mkdir {client,server,both}
$ meteor add meteorhacks:npm
$ meteor
```

Instead of starting Meteor like usual, this will instead create a file called `[project root]/packages.json`. Open that file in an editor and add the following declaration:

```
{
  "twit" : "1.1.20"
}
```

Save your changes, and in the terminal, run the `meteor` command again:

```
$ meteor
```

Obtaining your Twitter Access Tokens

Use a browser to log in to `http://twitter.com`, and then navigate to `https://apps.twitter.com`. Create a new app, following the instructions, and click on the **Keys and Access Tokens** tab to obtain your Consumer Key + Secret and a valid Access Token + Secret found on that page. *Keep this page open*, as we will need to reference it a bit later.

For Meteor-specific instructions on setting up a Twitter App, please see the *Implementing OAuth accounts packages* recipe, found in *Chapter 10, Working with Accounts.*

Initializing twit

Create a file named `[project root]/server/config-twit.js` and add the following code, replacing the `consumer_key`, `consumer_secret`, `access_token`, and `access_token_secret` with the information you obtained previously. When completed, your file should look similar to the following code:

```
Twitter = Meteor.npmRequire('twit');

Twit = new Twitter({
  consumer_key: 'egrdttfakeconsumerkeyFMx42339eMR8',
  consumer_secret: 'fR2r02CthisnJCDtVisMij2WjNiafakeo6QPqsecretnxztb',
  access_token: 'q8thisnEkn3xMiscUhafake9I5EOAtoken3DvDZM',
  access_token_secret: '7mel7Kr8fakeaccesstokensecretdzpiDuaqtRaij914'
});

simplifyTweet = function(tweet){
  var retObj = {};
  if (!tweet) return retObj;
  retObj.created_at = tweet.created_at;
  retObj.text = tweet.text;
  retObj.user = '@' + tweet.user.screen_name;

  return retObj;
}
```

Creating the Tweets collection, and building a stream reader

Create a file named `[project root]/both/model.js` and add the following line:

```
Tweets = new Mongo.Collection('tweets');
```

Now, create a file named `[project root]/server/twitter-stream.js` and add the following code:

```
stream = {};
Meteor.methods({
  TwitterStream: function (query) {
    if (query == 'off') {
      if (stream.stop != null) stream.stop();
      Tweets.remove({});
      return;
    }
    stream = Twit.stream('statuses/filter', {
      track: query
    });
    stream.on('tweet', function (tweet) {
```

```
        var simpleT = simplifyTweet(tweet);
        console.log(simpleT);
      });
    }
  })
```

Tracking and testing changes

Create a file named `[project root]/client/consoleTracking.js` and add the following code:

```
Tracker.autorun(function(){
    console.table(Tweets.find().fetch());
  });
```

Now, navigate in a browser to `http://localhost:3000`, open the console window, and enter the following command:

```
> Meteor.call('TwitterStream','JavaScript')
```

In your terminal window where you started Meteor with the `meteor` command, you should periodically see some JSON-formatted tweets. If tweets aren't showing up, it may be that the query you used isn't popular enough (this is a live Twitter feed!), so you can select something else, like *lmao* or *lebron*, if you want to see a steady stream.

Lastly, you will want to test the `Tweets` collection tracking. In the browser console, enter the following command, and run it multiple times:

```
> Tweets.insert({a:Tweets.find().count()})
```

If everything is set up properly, you will see a growing table of entries in your console.

To turn off the Twitter feed and to clear out the `Tweets` collection, enter the following in the browser console:

```
> Meteor.call('TwitterStream','off')
```

With everything all cleaned up, let's proceed to the recipe.

How to do it...

This recipe concentrates on only one thing: handling asynchronous events synchronously. As such, there are one two very simple steps as follows:

1. Open `[project root]/server/twitter-stream.js`, and add the following `insert` statement to the `stream.on` handler, as follows:

   ```
   stream.on('tweet', function (tweet) {
         var simpleT = simplifyTweet(tweet);
   ```

```
        console.log(simpleT);
        Tweets.insert(simpleT);
    });
```

Because the event handler is an asynchronous event, if you try turning the stream on as-is, you will get a very nasty error stating the following:

```
Meteor code must always run within a Fiber.
```

2. Modify the `stream.on` handler as follows:

```
stream.on('tweet', Meteor.bindEnvironment(
        function (tweet) {
        var simpleT = simplifyTweet(tweet);
        console.log(simpleT);
        Tweets.insert(simpleT);
    }));
```

You can now turn on the stream in the browser console, as follows:

```
> Meteor.call('TwitterStream','JavaScript')
```

As entries come in from the stream, they will populate in the Tweets collection, and you will see the results in your browser console, similar to the following:

(index)	_id	created_at	text	user
0	"GgdRdaGJeCXrwXGTL"	"Sun Mar 01 03:35:48 +0000 2015"	"UI resource, Other - India, 0 - 10 Yea…	"@tech_career"
1	"tu7FGqCSxkX4nxKYp"	"Sun Mar 01 03:35:48 +0000 2015"	"RT @leopixel19: Best Ways to Learn Pro…	"@jonDevBot"
2	"tnDAQhSXxuBfokbiN"	"Sun Mar 01 03:36:07 +0000 2015"	"PHP Developer, Other - India, 0 - 10 Y…	"@tech_career"
3	"pBFhr6R93pRRu3rWa"	"Sun Mar 01 03:36:09 +0000 2015"	"RT @WebInDev: Nothing to say.. just th…	"@AK_Nisha"
4	"aXpdwk75dp9hvBe2f"	"Sun Mar 01 03:36:12 +0000 2015"	"Angular JavaScript Developer Contract …	"@p2pWebMobileIt"
5	"JHaKXdz5NSwG3hprS"	"Sun Mar 01 03:36:21 +0000 2015"	"児ポ・シーンは未だにjavascriptすら使用しない…	"@taiyounomatecha"

How it works...

To accomplish the declarative, reactive environment we've come to know and love, the Meteor server intentionally runs on a single thread per request. Therefore, when you create some type of asynchronous action, such as listening to a `Node.js` event stream, the callback (handler) for that asynchronous action operates on a different thread. When the callback is fired, if you try to call Meteor functions (like `Tweets.insert()` for example), Meteor will throw an error, because the callback and the main Meteor server are on different threads. In addition, if you used any global variables with values, you can't guarantee that those values have remained the same while the asynchronous call was waiting. Some other operation could have changed the environment!

Meteor has an elegant way of dealing with this situation, using `Meteor.bindEnvironment()`.

By wrapping the callback function with `Meteor.bindEnvironment()`, a *Fiber* is created. The Fiber keeps track of the variables and their values (the environment), and knows that sooner or later the callback is going to be called.

In the meantime, the Fiber removes the operation from the event loop, so that the operation isn't blocking any other operations. Once the callback is ready, the Fiber puts the operation back into the event loop, restores the environment, and the operation completes.

In this particular case, when we use `Meteor.call('TwitterStream',…)` we are sending a request to the Meteor server on a single thread. The `Twit.stream` service is started on that thread but the events (the incoming stream) are asynchronous. By wrapping the handler for `stream.on()` with `Meteor.bindEnvironment()`, we are instructing Meteor to "take a snapshot" of the current environment. Meteor then takes the current operation out of the event loop so that other things can get through. Then, we wait.

When the stream has new data, the event fires, which triggers the callback. Meteor (or more accurately, the Fiber) sees that the callback is ready, restores the environment from the snapshot created using `Meteor.bindEnvironment()`, and puts the operation back on the event loop. This process happens again and again, as many times as needed, whenever a new Tweet comes in and triggers the callback function.

At the risk of oversimplifying, `Meteor.bindEnvironment()` wraps some code into the Fiber and then waits, without blocking any other code operations. Once the wrapped code is ready/activated, the Fiber makes sure that the code is executed on the correct thread with the correct data.

There's more...

We can be very granular about what code is wrapped, and we can separate out wrapped code, to make the overall code easier to read.

Add the following function to the top of the `twitter-stream.js` file:

```
wrappedInsert = Meteor.bindEnvironment(function(simpleT){
  Tweets.insert(simpleT);
});
```

Now, revert `stream.on` by removing the `Meteor.bindEnvironment()` wrapping, and call `wrappedInsert()` instead of `Tweets.insert()`, as follows:

```
stream.on('tweet', function (tweet) {
      var simpleT = simplifyTweet(tweet);
      console.log(simpleT);
      wrappedInsert(simpleT);
    });
```

This will operate exactly the same as wrapping the entire callback function.

One last alternative, we could take the entire callback out, wrap it with a user-friendly name, and use that name in the callback section, as shown in the following example:

```
wrappedCallback = Meteor.bindEnvironment(
  function (tweet) {
    var simpleT = simplifyTweet(tweet);
    console.log(simpleT);
    Tweets.insert(simpleT);
  }
);
...
stream.on('tweet', wrappedCallback);
```

The most concise explanation of Fibers, Futures, and `Meteor.bindEnvironment()` can be found here: `http://bit.ly/meteor-fibers-explained`. Make sure to thoroughly review all the links from that page!

See also

- The *Implementing OAuth accounts packages* recipe in *Chapter 10, Working with Accounts*

Using asynchronous functions

`Node.js`, for all its awesomeness, has a flaw: callbacks. Being asynchronous and non-blocking has a ton of advantages, and we can't imagine life without it. But, wouldn't it be great if there were a way to avoid the "callback hell" by writing our code in a synchronous style but still reaping the benefits of asynchronous code? As you might have guessed, Meteor has a way of doing just that. This recipe will show you how to write and handle asynchronous functions in a synchronous style using `Meteor.wrapAsync()`.

Getting ready

Because brevity breeds clarity, we will keep this recipe as simple as possible.

Open a terminal window, navigate to where you would like your project to reside, and enter the following commands:

```
$ meteor create wrap-sample
$ cd wrap-sample
$ mkdir server
$ meteor
```

How to do it...

We are going to simulate a delayed call to an asynchronous method, using the standard JavaScript `setTimeout()` function.

1. First, let's prep for the server call. Open `[project root]/wrap-sample.js` and modify the `Template.hello.events` function, as follows:

```
'click button': function () {
      var x = 0;
      while (x < 5) {
        x++;
        var q = "" + x + ". do work";
        Meteor.call('someCall', q, function (e, d) {
          console.log(d);
        });
      }
    }
```

2. Create a file named `[project root]/server/method.js` and add the following asynchronous function:

```
asyncCall = function(query,cb){
  var ranLen = ~~(Math.random()*3000);
  setTimeout(function(){
    cb && cb(null,query + " complete!");
  },ranLen);
};
```

3. Now, add a simple `Meteor.methods` declaration, using `Meteor.wrapAsync()` to write synchronous-style code, as follows:

```
Meteor.methods({
  someCall: function (query) {
    console.log('performing: '+query);
    this.unblock();
    var syncCall = Meteor.wrapAsync(asyncCall);
    var result = syncCall(query);
    return result;
  }
});
```

4. Open a browser to `http://localhost:3000`, open the browser console, and click the button on the screen. As you look at the server terminal, you will immediately see five log entries, printed in consecutive order, similar to the following:

```
performing: 1. do work
performing: 2. do work
```

```
performing: 3. do work
performing: 4. do work
performing: 5. do work
```

In the Browser window, you will see five log entries. These entries, however, will not be immediate, and will likely be out of order, similar to the following example:

```
1. do work complete!
4. do work complete!
2. do work complete!
5. do work complete!
3. do work complete!
```

5. Click the button again, and notice the random order and time of completion for each of the five server calls.

How it works...

`Meteor.wrapAsync()` is a bit of syntactic sugar for running asynchronous calls as if they were synchronous. As discussed in the *Handling asynchronous events* recipe from this chapter, Fibers and Futures are the way that Meteor handles asynchronous logic. In this case, we are dealing with what would be considered a *standard asynchronous function* in `Node.js`.

Meteor defines a standard asynchronous function as a function as follows:

- Takes a callback as the last argument
- Is non-blocking
- Executes the callback upon completion

The signature of the callback passed in is always the same. There are two arguments, the first of which is an `error` object. If there is an `error`, this object will have a value, otherwise, the `error` argument will be `null`.

The second argument is a `data` object. If the call does not return an `error` object as its first parameter, this data argument will have data in it that can be used by the callback function.

In our case, we declared a standard asynchronous function named `asyncCall()`. The signature of `asyncCall()` looks as shown in the following example:

```
function(query,cb)...
```

Nothing in `asyncCall()` will block the event loop, even though we're making a call to `setInterval()`. The `setInterval()` call is asynchronous, and is therefore non-blocking (very much like an ajax call, or a file i/o operation, etc). Once `setInterval()` is complete, it invokes our callback, `cb(null , query + '...')`.

In the `Meteor.methods` declaration, we created a method named `someCall`. Inside of `someCall`, we first make sure it won't clog up the event loop by calling `this.unblock()`.

We then take our standard asynchronous function (as defined previously) and we wrap it with the `Meteor.wrapAsync(asyncCall)` command.

Once wrapped, we can use the wrapped call as if it were synchronous. It's not. There will be a delay in the result, but our operation will wait until the asynchronous call is complete, which allows us to better understand what is happening. If we look at the lines of code after `Meteor.wrapAsync()`, we can easily wrap our heads around what is happening:

```
var result = syncCall(query);
return result;
```

The `result` variable is being assigned to the result of the call. We then return `result` like we would in a synchronous function. Easy peasy.

There's more...

The rule for using `Meteor.wrapAsync()` can be stated this way: any time you have a standard asynchronous function, where you would normally have to nest your logic in the callback, you can instead wrap the function and use the result as if it were a synchronous call.

Under the hood, Meteor is wrapping a Fiber/Future and automatically implementing the `.current`, `.yield()`, `.run()`, and `.wait()` pieces for you. But don't worry about that. If you just stick with the idea that `.wrapAsync()` is turning asynchronous functions with callbacks into standard functions without callbacks, you'll be well on your way to writing elegant, declarative code.

See also

- The *Handling asynchronous events* recipe in this chapter

12
Creating Useful Projects

In this chapter, we will cover the following topics:

- Creating RESTful web services
- Creating a complete app with Iron Router
- Deploying apps to mobile devices
- Adding social sharing

Introduction

Somewhere along your journey towards learning Meteor, you'll reach a point where you will have mastered the fundamentals. At that point, you will want to start putting all the fundamentals together into the applications or packages that are actually useful. It's a big development world out there, with many opportunities. The Meteor landscape is still in its infancy, and if you've followed all (or most) of the recipes in this book, you're probably ready to start defining that landscape. This last chapter will walk you through four of the most useful recipes that apply to most projects. It will give you an even more solid foundation for going out on your own to define, discover, and build!

Creating RESTful web services

Alas, alas, the entire development world hasn't caught on to how amazing the DDP protocol is (yet!). Data-over-the-wire is an amazing concept, and we believe that it (or something like it) is the future of how applications will communicate. In the meantime, we have some very useful and well-established protocols that, if implemented in our applications, increase the reach and accessibility of our apps for other platforms.

The granddaddy of all of these protocols is REST. REST is mature, well-defined, and has been implemented in nearly every programming language imaginable. It wouldn't surprise us to discover that, if/when we contact an alien race, the first signal will be a `POST` request. Given its current status (and our most likely option for peaceful interstellar negotiations), we will want to add REST capabilities to some of our applications. This recipe will walk you through implementing a server-side REST implementation using Iron Router.

Getting ready

Since we're only dealing with the REST part of an app in this recipe, we don't need anything fancy. A simple, standard Meteor app will do, with just a couple of server-side files.

Creating the baseline application

Open a terminal window, navigate to where you would like your project root to be, and enter the following:

```
$ meteor create RESTSample
$ cd RESTSample
$ rm RESTSample.*
$ mkdir -p server/api
$ mkdir server/model
$ meteor add iron:router
$ meteor
```

Installing and configuring Postman

We need a way to manually call our REST service, and the Postman plugin/app for Chrome is as good as any. If you have your own way of manually calling REST services, feel free to use it. If not, here's how to install Postman:

1. In a browser, navigate to `http://getpostman.com` and click on the **Get it now** link towards the bottom. A preview of the Postman app will pop up, and towards the top-right corner will be a button to install Postman. Click on that button, follow the directions, and open Postman.
2. In the request window in Postman, under the **Normal** tab, enter the following URL:

   ```
   http://localhost:3000/api/
   ```

3. Next, click on the **Headers** button, and enter the following under the **Header** | **Value** sections:

   ```
   Content-Type | application/json
   ```

4. When complete, your screen should look like the following screenshot:

We're now ready to build our REST service and test it using Postman.

How to do it...

Proceed with the following steps to create RESTful web services:

1. First, we will declare the `Quotes` collection. Create a file named `[project root]/server/model/quotes.js` and add the following code:

```
Quotes = new Mongo.Collection('quotes');
```

2. Next, we will add the `writeHeaders` function. Create a file named `[project root]/server/api/REST.js` and add the following function to the bottom:

```
function writeHeaders(self) {
  self.response.statusCode = 200;
  self.response.setHeader('Content-type',
'application/json');
  self.response.setHeader('Access-Control-Allow-Origin',
'*');
  self.response.setHeader('Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept');
}
```

3. We will now write the code to handle `GET` requests. At the very top of the `REST.js` file, create the following `Router.route()` method call:

```
Router.route('/api', {
    where: 'server'
  })
  .get(function () {
    //write headers
    writeHeaders(this);
```

```
    //send our response...
    this.response.end('GET not supported\n');
  });
```

Save all of your changes, make sure your app is running, and click on the **Send** button in Postman. In the results section, click on the **Body** tab and the **Raw** button. You should see the following message:

Congratulations, you've just created your first REST service!

4. Now, let's move on to `post` calls. We are going to assume that a `POST` query is asking to see any quotes in our `Quotes` collection that are owned by a particular user. In `REST.js`, remove the last semicolon from the `.get()` function call (we are chaining commands, so it's important that you do this) and add the following code where you removed the semicolon:

```
.post(function () {
    writeHeaders(this);
    var user = this.request.body.owner;
    if (!user) {
      this.response.end('No user specified...\n');
      return;
    }
    var quotes = Quotes.find({
      owner: user
    }).fetch();
    this.response.end(JSON.stringify(quotes));
  })
```

5. It doesn't make much sense to test a `POST` query until we have some data in our collection, does it? Let's take care of that right now by adding support for `PUT`. Immediately after the code you just entered (as a continuation of the function chain), add the following code:

```
.put(function () {
    writeHeaders(this)
```

```
    var upQuote = this.request.body.update;
    if (!upQuote) {
      this.response.end('nothing to update');
    }
    var update = Quotes.upsert({
      _id: upQuote.id
    }, {
      $set: upQuote.changes

    });
    this.response.end('Quote accepted!...\n');

  });
```

Note that we have a semicolon at the end, which means we're done and can start using our fully functional service!

6. In Postman, change the service type from `GET` to `PUT`. In the input field, click on the button labeled **raw**, and enter the following JSON code:

```
{"update":{
"changes":{"author":"Jerry Pournelle and Larry Niven",
"text":"Second guessing God is an old, old game.",
"owner":"me"
}
}}
```

Your screen should look like the following screenshot:

7. Click on the **Send** button, and in just a few milliseconds, the response area will display the following message:

   ```
   Quote accepted!...
   ```

8. Repeat the preceding steps as many times as you would like, varying the quote/author and owner as desired.
9. We can now test the `POST` query. Change the service type from `PUT` to `POST`, and enter the following JSON in the input field:

   ```
   {"owner":"me"}
   ```

10. Click on the **Send** button, and on the output window, click on the **Pretty** button. The output will be some well-formatted JSON with at least one entry, similar to that in the following screenshot:


```
[
  – {
        "_id": "b8PiXZxX4cHeCsd7Z",
        "author": "Jerry Pournelle and Larry Niven",
        "owner": "me",
        "text": "Second guessing God is an old, old game."
    }
]
```


You have just written a simple yet fully functional REST service in Meteor!

How it works...

The heavy lifting for this recipe is done by Iron Router. When we add the `iron:router` package, that package listens for all incoming requests on the server side. We configured Iron Router to listen on the `http://[yourapp.url]/api/` route for the `GET`, `POST`, and `PUT` requests. To get Iron Router to listen on that particular route, we used the `Router.route('/api'...)` method call. Importantly, we declared that, that route is to be handled on the server side using the `{where: 'server'}` parameter.

Chaining each type of request to the `.route()` call, we used the `.get()`, `.post()`, and `.put()` method calls. In each of these, we can read the JSON data passed in with the request by referencing `this.request.body.[key]`.

Finally, we are able to send a response by using Iron Router's `response` object, which we used to form the headers and to send messages back to the calling client.

There's more...

As usual, we've tried to keep the functionality to a minimum, to reduce the signal-to-noise of what this recipe is about. There's enough here, however, to give you a good taste of what is possible with REST services using Iron Router. If you would like more information and specifics on what the RESTful functions in Iron Router can do, please visit the Iron Router guide at: `https://github.com/iron-meteor/iron-router/blob/devel/Guide.md#server-routing`.

See also

- The *Creating a multipage application with Iron Router* recipe in *Chapter 2, Customizing with Packages*
- The *Creating a complete app with Iron Router* recipe in this chapter

Creating a complete app with Iron Router

From nearly the beginning of Meteor, the development emphasis has been on "smart" client-based applications. We left the server-dominated world behind a long time ago, and as a result, the single-page model is extremely well-supported inside the Meteor development stack.

Essential to this type of application is the `iron:router` package (`https://atmospherejs.com/iron/router`). Yes, you technically could build a multi-functional, mobile-first app without Iron Router, but it would be much more time consuming and complex by comparison. Think of Iron Router as the equivalent of using a microwave versus cooking outdoors on a campfire. With no utensils. In a rainstorm. With rabies-carrying raccoons all up in your business. Okay, okay, maybe it's not that bad, but you get the point—you should use Iron Router.

This recipe will walk you through the building of a fully functional, quote tracking service with a single-page app frontend built using Iron Router.

Getting ready

We would usually add a lot of preparation steps to a recipe, but in this one, we need to build everything in one go as part of the recipe. As such, we are only going to add the bare minimum here.

Open a terminal window, navigate to where you would like your application root to be, and enter the following commands:

```
$ meteor create QuotesApp
$ cd QuotesApp
```

```
$ rm QuotesApp.*
$ mkdir {client,both}
$ meteor add twbs:bootstrap
$ meteor
```

We've got a lot of work to do, so let's get started!

How to do it...

The major sections of this recipe are as follows:

- Creating the top `navbar`
- Adding user authentication
- Adding the `Quotes` collection
- Adding Iron Router routes
- Creating page transition animations
- Displaying the `Quotes` collection
- Adding and editing `Quotes`
- Deploying the App

There will be a little crossover from section to section, but we should be able to keep things fairly clean, so if you would like to switch out some functionality or add some features, feel free to do so as we move along.

1. First up, we need to create the top `navbar`.

   We will be using the standard Bootstrap `navbar` elements because we don't want to spend a large chunk of time with CSS styling. Once you're more comfortable with CSS, you can extend or replace the Bootstrap styling and make it your own.

   First, since this is a mobile-oriented app, let's set our headers, and use a Google font. Create a new file named `[project root]/client/header.html` and add the following declarations:

   ```
   <head>
     <meta name="viewport"
     content="width=device-width,
     initial-scale=1.0,
     maximum-scale=1.0,
     user-scalable=0">
     <link
   href='http://fonts.googleapis.com/css?family=Raleway'
     rel='stylesheet'
     type='text/css'>
   </head>
   ```

Create a new file named `[project root]/client/layout.html` and add the following template, which will act as the foundation for our navigation toolbar:

```
<template name="layout">
  <div class="navbar navbar-default navbar-fixed-top"
role="navigation">
    <div class="navbar-header">
      <a class="navbar-brand" id="btn-home" href="/">
        <span class="glyphicon glyphicon-book"></span></a>
      <button type="button"
              class="navbar-toggle collapsed"
              data-toggle="collapse"
              data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav navbar-right">
        <!-- Login buttons will go here -->
      </ul>
    </div>
  </div>
  <!-- App panels will go here-->
</template>
```

We will want to temporarily add a `body` element afterwards, just so we can check the display (we will remove it when we implement `iron:router`):

```
<body>
  {{> layout}}
</body>
```

Lastly, we have some minor CSS to add, to make sure everything upcoming runs smoothly. Create a file named `[project root]/client/styles.css` and add the following CSS declarations:

```
body {font-family: 'Raleway', sans-serif;}
.page {vertical-align: top;position: absolute;width: 100%;height:
100%;background-color: #eee;top: 50px;}
.page-detail {background-color: #fff;color: #555;}
.glyphicon-book {font-size: 3rem;line-height: 2rem;}
.toggle-edit{margin-top: 1.4rem;font-size: 2rem;
background-color: #777;border-radius: 50%;
border: none;padding:
1.5rem;width: 5rem;height: 5rem;}
```

```
.input-group {margin-top: 1.4rem;}
.quote {font-size:2rem;}
.title {font-size:3rem;}
.author {margin-bottom:2rem;margin-top:1rem;}
.tag {margin:0.25rem;}
.tags {margin-top:1rem;}
.add-quote {height:9rem;}
.add-quote .btn{color:#5bc0de;font-size:4rem;}
```

2. Next, we need to add user authentication.

   Because we've had experience with it in the past, we will enable authentication using Twitter. We will also use some stylized login buttons that fit in with our Bootstrap toolbar. Open a terminal window, navigate to your project root, and enter the following commands:

   ```
   $ meteor add accounts-twitter
   $ meteor add ian:accounts-ui-bootstrap-3
   ```

   We now need to add the `loginButtons` template to our toolbar. Open `[project root]/client/layout.html` and locate the comment that reads as follows:

   ```
   <!-- Login buttons will go here -->
   ```

   Replace this comment with the following code:

   ```
   {{> loginButtons}}
   ```

   Save your changes and navigate to your app in a browser. The top-right corner will contain a **Configure Twitter** button. Click on this button and configure Twitter according to the instructions, just as we did in the *Implementing OAuth accounts packages* recipe in *Chapter 10*, *Working with Accounts*.

   After configuration, you may want test the login with your Twitter account. If everything is satisfactory, we can move on to the meat of our application.

3. Let's add the `Quotes` collection.

   We're definitely ready to add our `Quotes` collection, but we want to have some control over what records will be sent to the client, based on whether the user is logged in or not. To accomplish this, we need to remove `autopublish`. In a terminal window, in our project root directory, enter the following command:

   ```
   $ meteor remove autopublish
   ```

   Next, create a file named `[project root]/both/model.js` and add the following line to that file:

   ```
   Quotes = new Mongo.Collection('quotes');
   ```

Ordinarily, we would add a `subscribe` command in a client-only file and a `publish` command in a server-only file, but the logic is so simple that we'll just add them directly to the `model.js` file in this case. Add the following code to `model.js` and save your changes:

```
if (Meteor.isClient) {
  Meteor.subscribe('quotes');
} else {
  Meteor.publish('quotes', function () {
    return Quotes.find({
      owner: this.userId
    });
  })
}
```

4. Up next, we need to add our Iron Router routes.

   When you add Iron Router, the changes needed in the running Meteor instance are pretty major. Technically, you could add Iron Router without stopping your app, but we recommend stopping your Meteor app while installing and configuring Iron Router.

   Once you've stopped your app, enter the following command in the terminal window:

   ```
   $ meteor add iron:router
   ```

   Let's now declare our default layout template. Create a file named `[project root]/both/router.js` and add the following line:

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

   Now that our layout template is being called by Iron Router, we no longer need the body element we temporarily put in the `layout.html` file. Open `[project root]/client/layout.html` and remove the entire `<body>...</body>` segment.

   We also have a placeholder in `layout.html` for our application pages, which we need to fill in with the customary Iron Router yield directive. In `layout.html`, locate the following line:

   ```
   <!-- App panels will go here-->
   ```

   Replace that line with the following code:

   ```
   {{> yield}}
   ```

   We have two routes to create, so let's do that now. Open `[project root]/both/router.js` and add the following `Router.map` declaration:

   ```
   Router.map(function(){
     this.route('main', {
       path: '/',
       template:'main'
     });
   ```

```
  this.route('detail', {
    path: '/quote/:_id',
    template:'detail',
    data: function () {
      return Quotes.findOne({_id: this.params._id});
    }
  });
});
```

We've created our routes, and now we need to create placeholder templates for those routes. Create a file named `[project root]/client/main.html` and add the following template:

```
<template name="main">
  <div class="page">
    <h1>this is the main page</h1>
  </div>
</template>
```

Create a file named `[project root]/client/detail.html` and add the following template:

```
<template name="detail">
  <div class="page page-detail">
    <h1>this is the detail page</h1>
  </div>
</template>
```

You can now start your app back up (enter the `meteor` command in your terminal) and test the routes. As you navigate to `http://localhost:3000` and to `http://localhost:3000/quote/1234`, you should see pages similar to those in the following screenshot:

5. We will now create page transition animations.

   There's an excellent Atmosphere package out there called `momentum`. With an Iron Router-specific package created by Percolate Studios, we can extend `momentum` to do Iron Router route transitions with very little fuss. In a terminal window, enter the following command:

   ```
   $ meteor add percolate:momentum-iron-router
   ```

   Create a file named `[project root]/client/transitions.js` and add the following transition declaration as a `Template` helper:

   ```
   Template.layout.helpers({
     transition: function () {
       return function (from, to, element) {
         if (to.template=="main") return 'left-to-right';
         return 'right-to-left';
       }
     }
   });
   ```

   We also need to surround our `{{> yield}}` inclusion with the `momentum` template helper. Open `layout.html` and add the following wrapper around the inclusion:

   ```
   {{#momentum plugin='iron-router' options=transition}}
         {{> yield}}
   {{/momentum}}
   ```

   Save your changes and navigate again to `http://localhost:3000/page/1234`. Once the page loads, click on the book icon in the top-left corner of the page. The main page should slide into place. The reverse animation will occur if you click the back button in your browser.

6. We are now ready to display the `Quotes` collection.

   We will display some light summary information for each quote in our main template and then expand that into a lot more detail for the detail template. Let's start with the main template.

   Open `[project root]/client/main.html`, remove the `<h1>` placeholder tag, and add the following code, which iterates over the `Quotes` collection and creates group items for each one:

   ```
   <template name="main">
    <div class="page">
      <div class="list-group">
        {{#each quotes}}
        <a class="list-group-item quote-title">
          <span class="glyphicon glyphicon-menu-right
          pull-right"></span>
          <h4 class="list-group-item-heading">{{title}}</h4>
   ```

```
        {{#each tagsFormatted}}
        <div class="btn btn-xs btn-primary">{{tag}}</div>
        {{/each}}
      </a>
      {{/each}}
    </div>
  </div>
</template>
```

We need to add a couple of helpers to support the previously created template, so create a file named `[project root]/client/main.js` and add the following code:

```
Template.main.helpers({
  quotes: function(){
    return Quotes.find().fetch();
  },
  tagsFormatted: function(){
    var tags = this.tags;
    if (!tags) return [];
    return _.map(tags.split(','),function(d){
      var retTag = '#' + d;
      return {tag:retTag};
    });
  }
});
```

We will also need a `click` event handler, which will get us to the `detail` page template whenever we click on a quote group item. Add the following code to the bottom of the `main.js` file:

```
Template.main.events({
  'click .quote-title' : function(e){
    Router.go('/quote/'+this._id);
  }
});
```

Save your changes. Now let's programmatically add a couple of entries to the `Quotes` collection. Navigate in a browser to `http://localhost:3000/`. Make sure that you are logged in with your Twitter account, open the browser console, and execute the following command two or so times:

```
Quotes.insert({owner:Meteor.userId(),
title:'quote title',
quote:'quote body',
author:'quote author',
tags:'test,tags,field'})
```

After doing so, your screen should look similar to the following screenshot:

If you click on one of the items, the generic detail page should slide into view. Let's now take care of the `detail` page template, so that it will display the details of our quotes. Open the `[project root]/client/detail.html` file and replace the template contents with the following:

```
<template name="detail">
  <div class="page page-detail">
    <div class="container">
      {{#if editing}}
      {{else}}
      <h3 class="title">{{title}}</h3>
      <div class="container quote">{{quote}}</div>
      <div class="row container author">
        <div class="pull-right">-{{author}}</div>
      </div>
      <div class="row container tags">
        {{#each tagsFormatted}}
        <div class="btn btn-xs btn-primary
tag">{{tag}}</div>
        {{/each}}
      </div>
      {{/if}}
    </div>
  </div>
</template>
```

Save your changes and click on one of the placeholder quotes. You should see a screen come into view that looks similar to the following screenshot:

The tags aren't showing up because we haven't added the necessary helpers yet. Create a file named `[project root]/client/detail.js` and add the following helpers declaration:

```
Template.detail.helpers({
  tagsFormatted: function () {
    var tags = this.tags;
    if (!tags) return [];
    return _.map(tags.split(','), function (d) {
      var retTag = '#' + d;
      return {
        tag: retTag
      };
    });
  }
});
```

Look back in your browser; the tags should now appear on the detail page template.

7. Now that we've got our display taken care of, let's add the ability to edit and insert new `quote` objects.

   First, let's adjust the `detail.html` file and add an editing toggle button. Open `detail.html` and add the following `div` element just before the last `</div>` tag:

```
</div>
<div class="container">
  <div class="btn btn-danger toggle-edit pull-right">
    <span class="glyphicon {{editbuttonstate}}"></span>
```

```
      </div>
    </div>
  </div>
</template>
```

Next, let's add the input fields for editing/inserting. In the same `detail.html` file, in the `{{#if editing}}` section, add the title input field. Do this using the Bootstrap `input-group` styling, like so:

```
{{#if editing}}
      <div class="input-group input-group-lg">
        <span class="input-group-addon"
         id="addon-title">Title</span>
        <input type="text"
               id="input-title"
               class="form-control"
               placeholder="title..."
               value="{{title}}">
      </div>
{{else}}
```

Next is the `input-group` element for the `quote` body. Add the following just after the `title` group in the preceding code snippet:

```
      </div>
      <div class="input-group">
        <span class="input-group-addon"
        id="addon-quote">Quote</span>
        <textarea name="input-quote"
                  id="input-quote" rows="4"
                  class="form-control">{{quote}}
        </textarea>
      </div>
{{else}}
```

We need an `input-group` element for the author. Add that just below the `quote` group, as follows:

```
      </div>
      <div class="input-group">
        <input type="text"
          id="input-author"
          class="form-control"
          placeholder="author..."
          value="{{author}}">
        <span class="input-group-addon"
        id="addon-author">Author</span>
      </div>
{{else}}
```

Last but not least for the template are the tags. Just below the `author` group, add the following code:

```
      </div>
      <div class="form tags">
        <textarea name="input-tags"
          id="input-tags"
          class="form-control"
          rows="5">{{tags}}</textarea>
      </div>
{{else}}
```

We have a couple of extra helpers to add. Open `detail.js` and add the following two helpers immediately after the `tagsFormatted` helper block, in the `Template.detail.helpers()` method (don't forget the extra comma at the top!):

```
  },
  editbuttonstate: function () {
    return (Session.get('editing')) ?
      'glyphicon-ok' : 'glyphicon-pencil';
  },
  editing: function () {
    return (Session.equals('editing', true));
  }
});
```

We now need to add the event handler for toggling between editing and display. We also need to add a submit button. So, let's kill two birds with one stone and give our toggle button the ability to turn into a submit button depending on the state. At the bottom of `detail.js`, add the following Template `events` declaration:

```
Template.detail.events({
  'click .toggle-edit': function (e) {
    e.preventDefault();
    var editflag = Session.equals('editing', true);
    if (editflag && Meteor.userId()) {
      var upQuote = {};
      upQuote.title = $('#input-title').val();
      upQuote.quote = $('#input-quote').val();
      upQuote.author = $('#input-author').val();
      upQuote.tags = $('#input-tags').val();
      upQuote.owner = Meteor.userId();
      Quotes.upsert({
        _id: this._id
      }, {
        $set: upQuote
      });
```

```
    }
    if (!editflag && !Meteor.userId()) return;
    Session.set('editing', (!editflag));
  }
});
```

Save your changes, navigate to display a single quote, and click on the pencil icon at the bottom right. You can change the generic quote into a specific quote, as shown in the following screenshot:

Click on the checkmark and your changes should be saved, as shown in the following screenshot:

We've taken care of editing; now we need to allow the user to add new quotes. The logic we're going to use will piggyback on our editing. When the "new qoute" button is clicked, we will instantly create a new `quote` object in the `Quotes` collection and then navigate to the editing screen. Open `[project root]/client/main.html` and add the following button group just after the last `{{/each}}` tag, but still inside the `list-group` `</div>` tag:

```
      {{/each}}
      <a href="#" class="list-group-item add-quote">
        <div class="btn btn-lg center-block">
          <span class="glyphicon glyphicon-plus"></span>
        </div>
      </a>
    </div>
  </div>
</template>
```

When this button is clicked, we want to insert a new object into the `Quotes` collection and then navigate to the editing screen. Open `main.js` and add the following event handler inside the `Template.main.events()` method call, just after the `.quote-title` event handler (again, don't forget the comma!):

```
  },
  'click .add-quote' : function(e){
    if (!Meteor.userId()) return;
    Quotes.insert({owner:Meteor.userId()},
      function(e,d){
        if (!e){
          Session.set('editing',true);
          Router.go('/quote/'+d);
        }
    });
  }
});
```

Save all of your changes, navigate to `http://localhost:3000` in a browser, and you should see a big blue plus sign at the bottom of the quote group. Click on the button and you'll be taken to a fresh, clean editing screen similar to the following sceenshot. Here, you can enter a new quote:

8. With all functionality in place, we are ready to deploy our app.

   We can deploy our app wherever we would like to (please refer to one of the many deployment recipes in *Chapter 1*, *Optimizing Your Workflow* for details), but for this recipe we'll pick the easiest/quickest way to deploy, which is directly to the Meteor servers.

   Stop your application from running in the terminal using *Ctrl* + *C* and execute the following command, replacing `[your-test-url]` with whatever you would like:

   ```
   $ meteor deploy [your-test-url].meteor.com
   ```

   In our case, we used the following command:

   ```
   $ meteor deploy packt-quotes.meteor.com
   ```

   Wait for your app to build and then upload, and you should get a familiar message, similar to the following:

   ```
   Deploying to packt-quotes.meteor.com.
   Now serving at http://packt-quotes.meteor.com
   ```

   Navigate to your newly-deployed app, configure Twitter if needed (you will need a new Twitter app ID, one that points to your new deployment URL), log in, and add some quotes. When you're finished, your app should look similar to the following screenshot:

Congratulations! You've just created and deployed a complete application in eight steps. You should be proud of yourself. Go get a soda or a celebratory slice of pizza. You deserve it!

How it works...

Since each of the preceding steps involves recipes from other chapters, we won't focus too much on the code specifics. Instead, we'll have a general overview of what each step does and add detail where needed.

We first added our top menu/navbar. Using the `navbar` and `navbar-fixed-top` CSS classes built in to Bootstrap, we were able to position our `navbar` at the top of our screen with very little effort. We added a web font (thank you, Google!) and set the parameters for use on mobile devices in our header file.

For user authentication, we relied on the default Meteor packages for accounts (specifically the `accounts-twitter` package). For formatting, we used the excellently designed `Bootstrap` package created by Ian Martorell. He did the heavy lifting for us, so all we had to do was add the package and place our `{{> loginButtons}}` inclusion in the correct place.

We declared the `Quotes` collection in a file visible to the client and the server, as usual, and added a `publish` function that will only send those results to the client which were created by the logged in user.

We added the Iron Router package and took advantage of the `Router.map()` and the `layoutTemplate` helpers of that package to implement a baseline layout template, a main template, and a details template.

We next leveraged a bit of wizardry from Percolate Studios, called `momentum`. They've created a plugin for `momentum` that allows you to wrap the `{{>yield}}` directive and point it to a template helper, where we perform some quick logic to say, "If we're going to the main page, go left to right. If not, go right to left." It's a well-designed package and is very easy to implement!

Our next steps (displaying, editing, and creating quotes) could be considered the meat of the recipe, but because all the scaffolding was already in place, and because this type of functionality is so easy to implement in Meteor, we simply used an `{{#each...}}` directive, some reactive data bindings, and a few `click` event handlers to display and edit the records from the `Quotes` collection.

In fact, the most complicated bit in the entire project was taking the `tags` property and parsing it into an array using `_.map()`. Think about that for a second. The most technically "challenging" part of our entire app was parsing a string. Meteor is pretty awesome, isn't it?

Our last step was deployment, which is also pretty straightforward, since we have already done it a few times in *Chapter 1*, *Optimizing Your Workflow*.

Taken as a whole, the building of an entire app from scratch seems daunting. Breaking it down into discreet steps made it not only straightforward, but actually pretty fun as well. We hope that you enjoyed it, and we hope that you will take this recipe, modify it and expand on it, and create some quick, useful apps of your own!

There's more...

If you examined the finished product of this recipe closely, you probably noticed that there is some missing functionality and some refinement that still needs to happen. For example, there's currently no way to delete a quote, we can't sort by tags/subject/author, and we actually deployed without removing the insecure package.

In other words, there are a lot of other recipes from previous chapters we could apply to improve the app. We will leave you to that, as you build and expand your own applications, and we can't wait to see what you come up with!

See also

- The *Deploying a test app to Meteor* recipe in *Chapter 1*, *Optimizing Your Workflow*
- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*
- The *Building a smooth interface with Bootstrap* recipe in *Chapter 3*, *Building Great User Interfaces*
- The *Implementing OAuth accounts packages* recipe in *Chapter 10*, *Working with Accounts*

Deploying apps to mobile devices

What good is building single-page apps with all kinds of cool functionality if you're only going to make it available as a web page? This recipe will take you through all the steps necessary to test and deploy your app to an iOS device.

Getting ready

You need an app to complete this recipe. We will be using the app created in the *Creating a complete app with Iron Router* recipe found earlier in this chapter. If you don't have your own app you would like to deploy, please complete that recipe first and then come back to this one to learn how to deploy your app.

How to do it...

There are two parts to deploying your application. The first is testing it on a device while using the server-side code that has already been deployed to a production URL. The second is to prepare an app build for deployment to the App Store.

1. First, let's set up our mobile app building environment. To deploy to an iOS device, you will need an Apple developer license, a machine running Mac OS X, and Xcode installed on that machine.

   To obtain an Apple Developer license, visit `https://developer.apple.com/programs/` and follow the links/instructions to join the iOS Developer program.

   Once you have joined the program, download and install Xcode from `https://developer.apple.com/xcode/downloads/`. You will need to properly configure Xcode, including adding certificates and mobile provisions as needed. There is a learning curve involved, and we strongly recommend that you take some time to learn the development, certification, and the App Store submission processes before proceeding.

   Open a terminal window, navigate to your app directory, and enter the following command:

   ```
   $ meteor install-sdk ios
   ```

   This can take a while, but make sure that the SDK is installed properly and fix any errors or outstanding installation items as directed.

   Once completed, enter the following command:

   ```
   $ meteor add-platform ios
   ```

   You're now properly configured, and can test/deploy your app on an iOS device.

2. Let's now test our app on an iOS device.

   Connect your iOS device to your machine via a USB cable. Enter the following command in the command line:

   ```
   $ meteor run ios-device --mobile-server [your-app-url.com]
   ```

   Here, `[you-app-url.com]` is replaced by the URL for your deployed server instance. In our case, we used the following command (all on one line, of course):

   ```
   $ meteor run ios-device --mobile-server
   packt-quotes.meteor.com
   ```

The application will be built and bundled, and then `Xcode` will open. Once the app has been compiled in `Xcode`, you will see the following message at the top-middle of the `Xcode` app:

QuotesApp | Build QuotesApp: **Succeeded** | Yesterday at 12:37 AM

Select your iOS device from the dropdown menu and click the play button, as displayed in this screenshot:

Make sure that your device is unlocked, and after some download and code push activity, an icon with the Meteor logo will appear on your device. The app will quickly open; then, after a short delay, your app will run just as if you pulled it up in a browser, but it is running as a full-fledged application on your device! Here's a screenshot of the `Quotes` app running on an iPhone:

3. The next step is to build a standalone mobile app file.

   Once you've tested everything and it's working as it should be, you will want to create an actual app file that you can submit to the Apple App Store. When you installed a test app on your mobile device, Meteor auto-created several assets for your app. You can view these assets by navigating to the `[project root]/.meteor/local/cordova-build/resources` folder, as shown in the following screenshot:

   You can use these resources or swap them out as needed. Once you've got your correct icons and splash screens in place, you will need to create a file named `[project root]/mobile-config.js` and add the following parameters, at a bare minimum:

```
App.info({
name: '[yourappname]',
description: 'Description of my app',
```

```
version: '0.0.1'
});

App.icons({
'iphone': 'resources/icons/icon-60.png',
'iphone_2x': 'resources/icons/icon-60@2x.png'
});

App.launchScreens({
'iphone': 'resources/splash/Default~iphone.png',
'iphone_2x': 'resources/splash/Default@2x~iphone.png',
'iphone5': 'resources/splash/Default-568h@2x~iphone.png'
});
```

You can, of course, change anything you would like in this file, and there are several other parameters you can add to further tweak the installation. See the `[project root]/.meteor/local/cordova-build/config.xml` file for a good solid list of options.

Once you're satisfied with your configuration, navigate to the root app directory and enter the following command in the terminal (all on one line, of course):

```
$ meteor build [your-output-directory]
--server=[your-app-url]
```

For example, our command for the `Quotes` application would look like this (all on one line, of course):

```
$ meteor build ~/builds/
--server=packt-quotes.meteor.com
```

Once completed, you can open the project directory in `Xcode` (in our case, the project directory would be `~/builds/ios/project`) and configure to your heart's content. Once everything is correct and running how you would like it, you can sign the app and submit it to the Apple App Store.

How it works...

Meteor leverages the `cordova` library, `Xcode`, and your Mac's operating system to bundle, compile, and package your application. The emphasis is to get it into `Xcode`, where you can then manipulate the app as you would any native app, testing or deploying or submitting to the App Store.

The specifics on what to do inside `Xcode` and while submitting to the Apple App Store are beyond the scope of this book. If you would like more Meteor-specific app building information, please visit `https://github.com/meteor/meteor/wiki/How-to-submit-your-iOS-app-to-App-Store`.

There's more...

Everything is brain surgery the first time. Building and deploying a mobile app is no exception. Here are some tips to help you troubleshoot, should things go wrong:

- **Do a "dial tone" test**: If your app is being built but is taking forever to "boot up" on the device, or if you just get a blank screen, first try making sure all of your network and Xcode configurations are correct. You can do this by building a brand new Meteor project using `meteor create my-test-app` and deploying that barebones app to your mobile device. If it comes up properly, you've successfully gotten a "dial tone" and you can start finding out which package or piece of code is the source of your slow-loading woes.
- **The Iron Router configuration must be on point**: If you are using Iron Router and you don't have your Iron Router routes configured properly, the app will usually refuse to load. If you are running into problems, implement the bare minimum routes, with no fancy code, and see if you can get the routes to come up on the device. Again, start simple.
- **Use the simulator**: You don't have to always push to the actual device. Using the `meteor run ios` or `meteor run android` commands will boot up a simulator in almost no time, where you can test and debug much more quickly than you could on an actual device.
- **Use Xcode's logs**: If there's a problem, check the logs, found in the bottom-right panel of Xcode. You can also set breakpoints using Xcode while the app is running on your device, which can help you isolate potential problems.

Above all else, stick with it. We can assure you that deploying to mobile devices, once you're properly configured, is extremely straightforward, with very high rates of success. Don't give up! Have faith that very soon it will just "click" and you won't have any problems/anxiety about building a mobile application. It will, in fact, become fun and extremely rewarding as you see your app in the App Store or on people's devices. It's worth the learning curve!

See also

- The *Creating a complete app with Iron Router* recipe in this chapter

Adding social sharing

With the ever-increasing demand to integrate social media into your apps, you'll want to gain a basic understanding of how to send/post/share directly from within your app. This recipe will show you how to add Twitter posting (tweets) to your application.

Getting ready

We will be using the Quotes application, from the *Creating a complete app with Iron Router* recipe found earlier in this in this chapter. Please complete that recipe before proceeding.

We will also be using the `npm twit` module to complete this recipe. For a step-by-step recipe with a full explanation of each step, please see the first part of the *Handling asynchronous events* recipe in *Chapter 11*, *Leveraging Advanced Features*.

Quickly, here are the steps you'll need to install and configure the `twit` module:

1. Execute the following terminal commands:

   ```
   $ meteor add meteorhacks:npm
   $ meteor
   ```

2. Open `packages.json` and add the following code:

   ```
   {
     "twit" : "1.1.20"
   }
   ```

3. Run the following terminal command:

   ```
   $ meteor
   ```

4. Create `[project root]/server/config-twit.js` and add the following configuration:

   ```
   Twitter = Meteor.npmRequire('twit');

   Twit = new Twitter({
     consumer_key: '…',
     consumer_secret:'…',
     access_token: '…',
     access_token_secret: '…'
   });
   ```

With our Twitter keys, tokens, and secrets all in place, and with `twit` initialized, we are ready to start tweeting!

How to do it...

We need to create a server method, update our UI, and give an indication that a quote has been shared. Thanks to twit, the actual call to the Twitter API is fairly easy. Let's get started.

1. First, let's create the `twuote` server method.

   Create a file named `[project root]/server/tweets.js` and add the following `Meteor.methods` declaration:

   ```
   Meteor.methods({
     twuote : function(id){
     }
   });
   ```

   We will first do our checking to make sure that a user is logged in and that we were passed the `id` argument of a valid `quote` variable. In the `twuote` function call, add the following lines:

   ```
   if (!id || !this.userId) return;
   var quote = Quotes.findOne({_id:id});
   if (!quote || !quote.quote || !quote.author) return;
   ```

   We now need to manipulate the `quote` and `author` strings, making sure it's only `140` maximum in length. Just below the previously inserted lines, add the following string manipulation logic:

   ```
   var tweet = '"';
   if (quote.quote.length>138){
     tweet += quote.quote.slice(0,135);
     tweet += '..."';
   } else {
     tweet += quote.quote + '" --' + quote.author;
   }
   if (tweet.length>140){
     tweet = tweet.slice(0,137) + '...';
   }
   ```

   After combing the quote, we are ready to send the tweet! Just below the string manipulation logic, add the following `Twit.post()` function and wrapped callback function:

   ```
   Twit.post('statuses/update', { status: tweet },
     Meteor.bindEnvironment(function(err, data, response) {
     Quotes.update({_id:id},{$set:{tweeted:true}});
   })
   );
   return tweet;
   ```

   Our method is complete. Let's modify our UI.

2. We'll start modifying our UI by adding the share button.

   Open the `[project root]/client/detail.html` file and add the following button just below the `toggle-edit` button, inside the parent container element:

```
<div class="container">
  <div class="btn btn-danger toggle-edit pull-right">
    ...
  </div>
  <div class="btn pull-left {{tweetColor}}"
   id="btn-tweet">{{tweetText}}
  </div>
</div>
```

   We want to give our button just a slight downward nudge, so add the following CSS directive at the bottom of the `[project root]/client/styles.css` file:

```
#btn-tweet{margin-top: 2.4rem;}
```

   We need to add a couple of additional helpers for our button, as well. Open `[project root]/client/detail.js`, and add the following two helper functions to the bottom of the `Template.detail.helpers` code block (don't forget the comma on the line before, as usual!):

```
editing: function () {
    ...
  },
  tweetText: function (){
    return (this.tweeted)?'shared':'share...';
  },
  tweetColor: function (){
    return (this.tweeted)?'btn-success':'btn-info';
  }
});
```

   All that's left is hooking up the click event. Open the `[project root]/client/detail.js` file, locate the `Template.detail.events` declaration, and add the following click `event` handler just inside the closing bracket (don't forget the comma on the line just before the handler!):

```
  },
  'click #btn-tweet' : function (e) {
    if (this.tweeted) return;
    Meteor.call('twuote',this._id);
  }
});
```

Save all of your changes, navigate to your app in a browser (usually `http://localhost:3000`), click on a quote to pull up the details page, and click on the button labeled **share...** shown in the following screenshot:

Give your server a moment or two to send the tweet. The **share...** button will turn green and the text will change to **shared**. Your tweet will be live on Twitter, as shown in the following screenshot:

How it works...

The important code is all found in the `[project root]/server/tweets.js` file, in the `Meteor.methods({})` declaration.

We are calling `Twit.post()`, which takes a callback for its last argument. That callback is asynchronous because `twit` is a Node module. Because it is asynchronous, we need a way of preserving the current state of the environment while the call is out and we need a way of putting the process back into the `event` loop when the callback is executed (see the *Handling asynchronous events* recipe in *Chapter 11*, *Leveraging Advanced Features*, for a full explanation).

We accomplish both of these items by wrapping the callback with `Meteor.bindEnvironment()`. When the callback does fire, we update the `Quotes` collection, adding a `{tweeted : true}` property to the corresponding quote.

This property, in turn, causes our UI to update with the green button and disallows additional tweets (checked in the `click event` handler, in `[project root]/client/detail.js`).

So, just like that, we've added social sharing. From here, you can take and expand this recipe to other social platforms to be even more seamlessly integrated in to your applications.

There's more...

It is very, very important that we caution against using the preceding code in production! We stripped out all the security and error handling. There is no validation, and the example is hard-wired for a single server-side user. If you use this code as it is in a production app, something not so pleasant will happen, and it won't be our fault (no blamezies allowed!).

Instead, we recommend that you flesh these elements out, particularly around allowing the use of the logged-in user's access token rather than that of a hard-wired user.

With just a little bit of spit and polish, you can easily use this code to complement your Meteor apps and extend the functionality to Facebook, Instagram, LinkedIn, Pinterest, and so on.

For a full write-up on what you can do with the `twit` module, please visit `https://github.com/ttezel/twit`.

See also

- The *Adding Meteor packages* recipe in *Chapter 2*, *Customizing with Packages*
- The *Handling asynchronous events* recipe in *Chapter 11*, *Leveraging Advanced Features*
- The *Creating a complete app with Iron Router* recipe in this chapter

Module 3

Meteor Design Patterns

Accelerate your code writing skills with over 20 programming patterns that will make your code easier to maintain and scale

1
Getting Started with Meteor

Meteor is a framework that is essentially fast for the purpose of development even if you program at a slow pace. The aim of this book is to increase your development speed and improve quality. There are two key ingredients required to improve development: compilers and patterns. Compilers add functionality to your coding language, while patterns increase the speed at which you solve common programming issues.

This book will mostly cover patterns, but we'll use this chapter to quick start compilers and understand how they relate to Meteor—a vast but simple topic. The compilers that we will look at are as follows:

- CoffeeScript
- Jade
- Stylus

We will review some basic knowledge you should have about Meteor. This will cover the following:

- Templates, helpers, and events
- The event loop and the merge box
- The must-have packages
- Folder structure

CoffeeScript for Meteor

CoffeeScript is a compiler for JavaScript that adds "syntactic sugar" inspired by Ruby, Python, and Haskell; it effectively makes the writing of JavaScript easier and more readable. CoffeeScript simplifies the syntax of functions, objects, arrays, logical statements, binding, managing scope, and much more. All CoffeeScript files are saved with a `.coffee` extension. We will cover functions, objects, logical statements, and bindings as these are some of the most commonly used features.

Objects and arrays

CoffeeScript gets rid of curly braces (`{}`), semicolons (`;`), and commas (`,`). This alone saves you from repeating unnecessary strokes on the keyboard. Instead, CoffeeScript emphasizes on the proper use of **tabbing**. Tabbing will not only make your code more readable, but it will be a key factor in making the code work as well. In fact, you are probably already tabbing the right way! Let's look at some examples:

```
#COFFEESCRIPT
toolbox =
  hammer:true
  flashlight:false
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Here, we are creating an object named `toolbox` that contains two keys: `hammer` and `flashlight`. The equivalent in JavaScript will be this:

```
//JAVASCRIPT - OUTPUT
var toolbox = {
  hammer:true,
  flashlight:false
};
```

Much easier! As you can see, we have to **tab** to indicate that both the `hammer` and `flashlight` properties are a part of `toolbox`. The `var` word is not allowed in CoffeeScript because CoffeeScript automatically applies it for you. Let's have a look at how we would make an array:

```
#COFFEESCRIPT
drill_bits = [
  "1/16 in"
  "5/64 in"
  "3/32 in"
  "7/64 in"
]

//JAVASCRIPT - OUTPUT
var drill_bits;
drill_bits = ["1/16 in","5/64 in","3/32 in","7/64 in"];
```

Here, we can see that we don't need to have any commas, but we do need to have brackets to determine that this is an array.

Logical statements and operators

CoffeeScript removes a lot of parenthesis (`()`) in logical statements and functions as well. This makes the logic of the code much easier to understand at first glance. Let's look at an example:

```
#COFFEESCRIPT
rating = "excellent" if five_star_rating

//JAVASCRIPT - OUTPUT
var rating;

if(five_star_rating){
  rating = "excellent";
}
```

In this example, we can clearly see that CoffeeScript is easier to read and write. CoffeeScript effectively replaces the entire **implied parenthesis** in any logical statement.

Operators such as `&&`, `||`, and `!==` are replaced by words to make the code more readable. Here is a list of the operators that you will be using the most:

CoffeeScript	JavaScript
`is`	`===`
`isnt`	`!==`
`not`	`!`
`and`	`&&`
`or`	`||`
`true, yes, on`	`true`
`false, no, off`	`false`
`@, this`	`this`

Let's look at a slightly more complex logical statement and see how it is compiled:

```
#COFFEESCRIPT
# Suppose that "this" is an object that represents a person and
their physical properties

if @eye_color is "green"
  retina_scan = "passed"
else
  retina_scan = "failed"

//JAVASCRIPT - OUTPUT
if(this.eye_color === "green"){
  retina_scan = "passed";
} else {
  retina_scan = "failed";
}
```

Notice how the context of `this` is passed to the `@` symbol without the need for a period, making `@eye_color` equal to `this.eye_color`.

Functions

A JavaScript function is a block of code designed to perform a particular task. JavaScript has a couple of ways of creating functions that are simplified in CoffeeScript. They look like this:

```
//JAVASCRIPT
//Save an anonymous function onto a variable
var hello_world = function(){
```

```
  console.log("Hello World!");
}

//Declare a function
function hello_world(){
  console.log("Hello World!");
}
```

CoffeeScript uses `->` instead of the `function()` keyword. The following example outputs a `hello_world` function:

```
#COFFEESCRIPT
#Create a function
hello_world = ->
  console.log "Hello World!"

//JAVASCRIPT - OUTPUT
var hello_world;
hello_world = function(){
  return console.log("Hello World!");
}
```

Once again, we will use a tab to specify the contents of the function so that there is no need for curly braces (`{}`). This means that you have to make sure that you have the entire logic of the function tabbed under its namespace.

What about our parameters? We can use `(p1,p2) ->` where `p1` and `p2` are parameters. Let's make our `hello_world` function output our name:

```
#COFFEESCRIPT
hello_world = (name) ->
  console.log "Hello #{name}"

//JAVSCRIPT - OUTPUT
var hello_world;
hello_world = function(name) {
  return console.log("Hello " + name);
}
```

In this example, we can see how parameters are placed inside parentheses. We are also doing **string interpolation**. CoffeeScript allows the programmer to easily add logic to a string by escaping the string with `#{}`. Also notice that, unlike JavaScript, you do not have to return anything at the end of the function, CoffeeScript automatically returns the output of the last command.

Binding

In Meteor, we will often find ourselves using the properties of `this` within nested functions and callbacks. **Function binding** is very useful for these types of cases and helps to avoid saving data in additional variables. Function binding sets the value of the `this` object inside the function to the value of `this` outside the function. Let's look at an example:

```
#COFFEESCRIPT
# Let's make the context of this equal to our toolbox object
# this =
#   hammer:true
#   flashlight:false

# Run a method with a callback
Meteor.call "use_hammer", ->
  console.log this
```

In this case, the `this` object will return a top-level object such as the browser window. This is not useful at all. Let's bind `this` now:

```
#COFFEESCRIPT
# Let's make the context of this equal to our toolbox object
# this =
#   hammer:true
#   flashlight:false

# Run a method with a callback
Meteor.call "use_hammer", =>
  console.log this
```

The key difference is the use of `=>` instead of the expected `->` to define the function. Using `=>` will make the callback's `this` object equal to the context of the executing function. The resulting compiled script is as follows:

```
//JAVASCRIPT
Meteor.call("use_hammer", (function(_this) {
  return function() {
    return Console.log(_this);
  };
})(this));
```

CoffeeScript will improve your coding quality and speed. Still, *CoffeeScript is not flawless*. When you start combining functions with nested arrays, things can get complex and difficult to read, especially when the functions are constructed with multiple parameters. Let's look at a common query that does not look as readable as you would expect it to be:

```
#COFFEESCRIPT
People.update
  sibling:
    $in:["bob","bill"]
,
  limit:1
  ->
    console.log "success!"
```

This collection query is passing three parameters: the `filter` object, the `options` object, and the callback function. To differentiate between the first two objects, we had to place a comma at the same level as the function and then, indent the second parameter. This is unwieldy, but we can use variables in order to make the query more readable:

```
#COFFEESCRIPT
filter =
  sibling:
    $in:["bob","bill"]
options =
  limit:1
People.update filter, options, ->
  console.log "success!"
```

Go to `coffeescript.org` and play around with the language by clicking on the "try coffeescript" link.

Jade for Meteor

Jade works much like CoffeeScript but it is used for HTML instead. I recommend that you install the `mquandalle:jade` package. All the Jade files are saved with a `.jade` extension. This section will cover the most used aspects of Jade in Meteor such as HTML tags, components, and helpers.

HTML tags

Much like CoffeeScript, Jade is a language that depends heavily on tabbing. When you want to add children to an HTML tag, you simply use tab. Tag IDs and classes can be added using the CSS selector notation (`'input#name.first'`). This means classes are expressed with a dot (`.`) and IDs are expressed with a pound (`#`). Let's look at an example:

```
//- JADE
div#container
  ul.list
    li(data-bind="clickable") Click me!

<!-- HTML - OUTPUT -->
<div id="container">
  <ul class="list">
    <li data-bind="clickable">Click me!</li>
  </ul>
</div>
```

As you can see, **special** attributes such as `data-bind` are added with parenthesis. Symbols such as `<`, `>`, and **closures** are not required anymore. In this example, we have a `div` tag with an `id` attribute of `"container"`, a `ul` tag with a `class` attribute of list, and a `li` tag with a special attribute of `data-bind`.

You will find yourself using special attributes often for the `input` tags to add `value`, `placeholder`, and other attributes.

Templates and components

Meteor templates are Jade components. In Meteor, we define a template with the template tag and apply the special `name` attribute to create a reusable HTML block. In Jade, when we create a template, we create a component as well. This looks as follows:

```
//- JADE
template(name="landing")
  h3 Hello World!

<!-- HTML - OUTPUT -->
<template name="landing">
  <h3>Hello World!</h3>
</template>
```

Now we can use this template as a Jade component anywhere in our views. To call a Jade component, you simply prepend a plus sign to the name of the template. Let's look at an example where we want to place a `landing` page inside a `main_layout` page:

```
//- JADE
template(name="landing")
  h3 Hello World!

template(name="main_layout")
  +landing

<!-- HTML - OUTPUT -->
<template name="landing">
  <h3>Hello World!</h3>
</template>

<template name="main_layout">
  {{> landing}}
</template>
```

That's it! Notice that we have prepended the plus (+) sign to the name of the template to call it. This is equivalent to using `{{> landing}}` in SpaceBars (Meteor's version of Handlebars). Components can have parameters as well, which can be later used in the templates' instance. Let's make our example output someone's name:

```
//- JADE
template(name="landing")
  h3 Hello {{name}}

template(name="main_layout")
  +landing(name="Mr Someone")

# COFFEESCRIPT
Template.landing.helpers
  "name": ->
    Template.instance().data.name

<!-- HTML - OUTPUT -->
<template name="landing">
  <h3>Hello {{name}}</h3>
</template>

<template name="main_layout">
  {{> landing name="Mr Someone"}}
</template>
```

Adding attributes to your templates can make your templates flexible as shown in the preceding example. Still, it is unlikely that you will have to use this as templates "soak up" data from their parent context.

Helpers

Helpers in Meteor are functions that return data before rendering to the view. We use helpers for iteration, logical statements, and variables. The two basic helpers are `each` and `if`, but adding the `raix:handlebar-helpers` package will add a dictionary of other useful helpers to keep your code from repeating. Let's have a look at how we can use our helpers:

```
//- JADE
template(name="list_of_things")
  each things
    if selected
      p.selected {{name}}
    else
      p {{name}}

<!-- HTML - OUTPUT -->
<template name="list_of_things">
  {{#each things}}
    {{#if selected}}
      <p class="selected">{{name}}</p>
    {{else}}
      <p>{{name}}</p>
    {{/if}}
  {{/each}}
</template>
```

In this example, the `each` helper is iterating through the return value of another helper named `things` and if the `selected` helper resolves to `true`, then we will render `p.selected` with the `name` variable.

It's important to understand that everything that is not an HTML tag is a helper, and that if you want to use a helper within a tag, you need to use `{{}}` or `#{}` to express this.

Go to `jade-lang.com` and `handlebars.js` to know more specific information. With this information, you should be able to do just about anything.

Stylus for Meteor

Stylus works much like CoffeeScript and Jade but it is for CSS. I recommend that you install `mquandalle:stylus`. This package is preinstalled with useful tools such as `Jeet` and `Rupture`. All Stylus files are saved with a `.styl` extension. There are only three things that we need to learn about Stylus: CSS tags, variables, and functions.

CSS tags

Stylus is a language that does away with the need for semicolons (`;`) and curly braces (`{}`) in exchange for making good use of tabbing. Let's look at an example:

```
// STYLUS
// Let's make a vertical positioning class
.vertical-align-middle
  //PART 1
  position:absolute
  width:100%
  height:100%
  display:table
  overflow-x:hidden

  .special
    background:black
```

We can see in `PART 1` how properties are defined for a class by tabbing those properties in `.special` is used to select an HTML tag with the `special` class that is a child of the `vertical-align-middle` class. Let's look at how `PART 1` compiles:

```
/* CSS - OUTPUT PART 1 */
.vertical-align-middle {
  position:absolute;
  width:100%;
  height:100%;
  display:table;
  overflow-x:hidden;
}
.vertical-align-middle .special {
  background:black;
}
```

Now let's add a more complex selector:

```
// STYLUS
// Let's make a vertical positioning class
.vertical-align-middle
  //PART 1
  ...

  //PART 2
  > *
    display:table-cell
    vertical-align:middle
```

`PART 2` has a combination of special CSS2 selectors: specific parent (`>`) and all elements (`*`). In this particular order, the CSS2 selectors are picking the "any first sibling" element only and applying the rules. Let's look at how `PART 2` compiles:

```
/* CSS - OUTPUT PART 2 */
.vertical-align-middle > * {
  display:table-cell;
  vertical-align:middle;
}
```

Let's add a new class to the current class that aligns the object to the top:

```
// STYLUS
// Let's make a vertical positioning class
.vertical-align-middle
  //PART 1
  ...
  //PART 2
  ...
  //PART 3
  &.whole-page
    top:0
```

`PART 3` uses an ampersand (`&`) to describe an element that is not a child but instead is concatenated with the extra class. Let's look at how `PART 3` compiles:

```
/* CSS - OUTPUT PART 3 */
.vertical-align-middle.whole-page {
  top:0;
}
```

Variables

Unlike CSS, Stylus supports variables. This keeps a lot of things manageable when we want to make major changes to the look of our site. Suppose we have two colors that we want to use throughout our site, but we know that these colors are going to change. Let's define them as variables so that we can easily modify them later:

```
// STYLUS
primary-color = #ffffff
$secondary-color = #333333

.text-default
  color:primary-color
  background:$secondary-color

.text-inverted
  color:$secondary-color
  background:primary-color
```

Easy right? In this example, both `primary-color` and `$secondary-color` are variables. Stylus optionally supports the use of the money sign (`$`) to indicate a variable. This can make it easier to spot variables.

Functions/mixins

Unlike CSS, Stylus supports functions too. LESS, Stylus, and **Sassy CSS** (**SCSS**) refer to functions as **mixins**. Functions will make your CSS concoctions much easier to share across a project. We will cover the two types of mixins in Stylus: mixins and transparent mixins.

Mixins are functions that take a defined set of parameters. Let's take a look at how we can write a mixin:

```
// STYLUS
animation(duration,delay,timing)
  -webkit-animation-duration:duration
  animation-duration:duration
  -webkit-animation-delay:delay
  animation-delay:delay
  -webkit-animation-timing-function:timing
  animation-timing-function:timing

button
  animation(500ms,0,ease-out)
```

```
/* CSS - OUTPUT */
button {
  -webkit-animation-duration:500ms;
  animation-duration:500ms;
  -webkit-animation-delay:0;
  animation-delay:0;
  -webkit-animation-timing-function:ease-out;
  animation-timing-function:ease-out;
}
```

In this example, we first define the `animation` mixin, and then we apply the mixin to the `button` HTML tag. However, there is a much easier and effective way of doing this via a transparent mixin.

A transparent mixin, basically, takes all the parameters and saves them in an `arguments` variable without you having to define anything. Let's have a look:

```
// STYLUS
animation()
  -webkit-animation:arguments
  animation:arguments

button
  animation(pulse 3s ease infinite)

/* CSS - OUTPUT */
button {
  -webkit-animation:pulse 3s ease infinite;
  animation:pulse 3s ease infinite;
}
```

Notice how we did not have to define every single parameter in the mixin, and the `arguments` variable simply passed all the arguments that it could find. This is especially useful for keeping the code flexible.

Stylus essentially upgrades CSS in such a way that it makes the code much easier to manage and therefore, ends up saving us a lot of development time.

Go to `stylus-lang.com` and `learnboost.github.io/stylus` to learn more about Stylus.

Templates, helpers, and events

Now that we are on the same page for the languages that we are going to use throughout the book, let's do a quick review of some of the elements that we will use during our development process.

Templates, helpers, and events are used to build the frontend of your application. Using them effectively is the key to how we design our backend as well (which we will address in *Chapter 2*, *Publish and Subscribe Patterns*).

Templates

Meteor templates are the special blocks of HTML code that generate Meteor template objects (`Template.<yourtemplate>`). It is through Meteor template objects that we wire the HTML code to logic. People who have worked with an MVC framework will refer to these templates as views. This is a key concept to understand.

Open up your terminal and create a new project:

```
meteor create basic_meteor
```

Now let's add our languages:

```
meteor add coffeescript
meteor add mquandalle:jade
meteor add mquandalle:stylus
```

Remove the three visible files from `/basic_meteor` (do not remove any of the files starting with a dot), and create `/client/layout.jade`. This is something that exists in one way or another in every Meteor project. Let's program:

```
//- layout.jade
head
  title Meteor Basics
  meta(name="viewport" content="user-scalable=no, initial-
  scale=1.0, maximum-scale=1.0")
  meta(name="apple-mobile-web-app-capable" content="yes")
body
  +basic_template
```

I highly recommend adding these metatags to make your site mobile-friendly right from the beginning. With this snippet of code, we are effectively starting up the very first thing that Meteor is going to render before running any code. Once this is rendered, Jade takes care of rendering `basic_template`. Let's program this in a new file, `/client/basic_template.jade`:

```
//- basic_template.jade
template(name="basic_template")
  h1 Hello World!
```

Behind the scenes, Meteor is compiling our Jade templates and putting them all in one big file. You will never have to worry about loading `basic_template.jade` before `layout.jade` when it comes to templating.

Throughout the book, we will use `meteorhacks:flow-router` and `meteorhacks:flow-layout` to easily navigate to different templates.

Creating helpers

We have already discussed what helpers are in Jade, but how do we create helpers in Meteor? Let's go back to our `basic_meteor` project and create `/client/basic_template.coffee`. It is important to understand that Meteor helpers are used to control the variables in our template. People who have worked with an MVC framework can view this file as a controller. Let's write our first helper:

```
#basic_template.coffee
Template.basic_template.helpers
  "name": ->
    "Mr Someone"
```

Notice that the helper is defined within the `helpers` function of the Meteor template object: `Template.<your_template>.helpers(<your_helpers>)`. Helpers are mostly functions that will return anything you want them to including Meteor collection cursors. Let's bring all this together now:

```
//- basic_template.jade
template(name="basic_template")
  h1 Hello {{name}}
```

This will output `Hello Mr Someone` inside the `h1` HTML tag. Let's add a slightly more complex helper:

```
#basic_template.coffee
Template.basic_template.helpers
  "person": ->
    name:"Someone"
```

```
    prefix: "Mr"
    children: [
      {
        name:"Billy"
      }
      {
        name:"Nancy"
      }
    ]

//- basic_template.jade
template(name="basic_template")
  with person
    h1 Hello {{prefix}} {{name}}

    ul
      each children
        li {{name}}
```

In this example, we are using `with` to set up the **data context** of the HTML tags that belong to it; this data context is equivalent to `person`. Data context refers to the value of `this` inside a helper. So if you set up an object as the data context, `this` will be equivalent to that object. Also, we iterate through `children` with an `each` statement so that we can list out their names.

Events

Meteor taps into common JavaScript HTML events such as click, change, and focus. An event is anything that happens to an HTML element that you are listening to. Suppose we want to be able to change the name of a person to one of the children by clicking on them. We do this through the templates' event map. Let's take a look at an example of how we can do this without using reactivity or collections:

```
#basic_template.coffee
Template.basic_template.events
  "click li": ->
    $("h1").text "Hello #{@name}"
```

Easy! So to catch template events, we need to use the `Template.<your_template>.events(<your_event_map>)` function. In this particular example, we are using jQuery to replace text.

The `event map` is an object where the properties specify a set of events to be handled. These events may be specified in any of the following ways:

```
# Runs any time you click anywhere
"click": ->

# Runs any time you click a li element
"click li": ->

#Runs any time you click a li element OR mouseover a li element
"click li, mouseover li": ->
```

The key `string` of the event is composed of two parts: the first is always a type of event (click, hover, change, and so on) while the second is always a CSS selector.

The event loop and the merge box

Before diving into Meteor, it is critical to understand what the event loop and the merge box are and how they can adversely affect your code. Both are relatively complex in the way that they were programmed, so we will focus on understanding the general concept.

The event loop

The event loop is like a queue; it runs a series of functions one by one. Because functions are processed sequentially, each function effectively blocks others from being processed until the function is done.

In other words, the event loop functions much like a single-line conveyor belt where things are being inspected. For every inspection made, the line is stopped and nothing moves.

Meteor uses Fibers – a NodeJS library – to get around this issue. Many of the functions that you will run will be on a separate fiber. What does this mean? This means that the functions will run on a separate conveyor belt for processing. Still, not all functions are built this way, you need to make sure your server-side functions do not block the server.

So which functions could potentially cause the server to get blocked? `Meteor.methods()`, `Meteor.publish()`, and `any` function that does not run inside a fiber on the server. Let's see how we can unblock each one and when we should do this.

Functions defined under the `Meteor.methods()` that you know are going to take a long time to process, should always run on a Fiber or defer time consuming code to a Fiber. We can quickly solve this by calling the `@unblock()` function from within the method. Let's look at an example:

```
# METEOR METHODS
Meteor.methods
  #BLOCKING
  time_consuming: ->
    Meteor.setTimeout ->
        console.log "done"
      ,60000

  #NON-BLOCKING
  time_consuming_unblock: ->
    @unblock()
    Meteor.setTimeout ->
        console.log "done"
      ,60000
```

In this example, when you run `Meteor.call("time_consuming")`, the server will be blocked. When the server is blocked, other visitors won't be able to reach your site! Instead if you run `Meteor.call("time_consuming_unblock")`, the server will continue to function properly but consume more resources to do so.

`Meteor.publish()` can be easily unblocked after installing the `meteorhacks:unblock` package as well. This one will be particularly useful when we start to make very complex publishers that might consume a lot of resources. Let's look at an example:

```
# METEOR PUBLISH
#BLOCKING
Meteor.publish "external_API_query", ->
  HTTP.get "http://connect.square.com/payments"

#NON-BLOCKING
Meteor.publish "external_API_query_unblocked", ->
  @unblock()
  HTTP.get "http://connect.square.com/payments"
```

In this example, we are waiting for an HTTP call to respond. This will certainly block the server if we subscribe to `external_API_query`, so we use `external_API_query_unblocked` instead.

All other functions that run on the server and you know are going to block the server, should run on a fiber. Meteor has a special function to help us make this easy. It is called `Meteor.wrapAsync()`. Let's see how this works:

```
# METEOR UNBLOCKED FUNCTION
unblock_me = Meteor.wrapAsync ->
  Meteor.setTimeout ->
      console.log "done"
    ,60000
```

It is very important to keep the event loop in mind, especially when we're connecting our web application to external services that are going to cause massive delays to our server.

The merge box

The merge box is the algorithm that identifies all the changes that are happening to the database. It basically handles publishers, subscribers, and reactivity. The merge box also handles the initial load of data using DDP messages.

It is important to understand that we can communicate directly with the merge box via all the commands that are available to us under the `Meteor.publish()` function. The more optimal we can make our `Meteor.publish` functions, the faster the site will load.

The beginning of our online shop

Throughout the book we will be developing an e-commerce website to help us understand the core concepts of advanced Meteor web development. Let's begin by creating a new project:

```
meteor create online_shop
```

The must-have packages

`Atmospherejs.com` has always been the "go to" website to find packages. Here you will find thousands of packages produced by the community for free. There are a handful of packages that we absolutely need to install to make our website function properly.

First, we install the languages:

```
meteor add coffeescript
meteor add mquandalle:jade
meteor add mquandalle:stylus
meteor add less
```

Next, the router and functions that will help us with SEO and routing:

```
meteor add kadira:flow-router
meteor add kadira:blaze-layout
meteor add meteorhacks:fast-render
meteor add nimble:restivus
meteor add yasinuslu:blaze-meta
meteor add dfischer:prerenderio
meteor add wizonesolutions:canonical
```

WARNING: Do not run Meteor yet! Canonical could mess up your project unless you have it set up correctly.

We will need a couple of packages as well to help us manage publishers:

```
meteor add lepozepo:publish-with-relations
meteor add tmeasday:publish-counts
meteor add meteorhacks:aggregate
meteor add http
meteor add meteorhacks:unblock
```

These next packages will extend Meteor's functions:

```
meteor add xorax:multiple-callbacks
meteor add aldeed:collection2
meteor add aldeed:autoform
meteor add fastclick
meteor add reactive-var
meteor add alanning:roles
meteor add accounts-password
meteor add u2622:persistent-session
meteor add ongoworks:security
```

We will need these packages to properly manage time:

```
meteor add momentjs:moment
meteor add mrt:moment-timezone
```

For the last set, we'll be using a couple of additional packages that will make the design process much faster:

```
meteor add kyleking:customizable-bootstrap-stylus
meteor add raix:handlebar-helpers
meteor add fortawesome:fontawesome
meteor add percolate:momentum
```

We need to remove some packages for security too:

```
meteor remove autopublish
meteor remove insecure
```

All of these packages will be explained in more detail throughout the book, but all of these are must-haves. The first package that we need to explain is the `wizonesolutions:canonical` package. This package makes sure that all incoming traffic is routed to your `ROOT_URL`, so it is particularly useful when you want all the traffic to go to your SSL site. The first thing that we need to do before running Meteor is set up canonical to run only in the production environment.

Create `/server/canonical.coffee`, and add this code:

```
#/server/canonical.coffee
if process.env.NODE_ENV is "development" or
process.env.ROOT_URL.indexOf("meteor.com") > -1
  Meteor.startup ->
    process.env.PACKAGE_CANONICAL_DISABLE = true
```

This code snippet effectively sets your `PACKAGE_CANONICAL_DISABLE` environment variable to make sure that canonical is inactive while you are developing.

What are environment variables? These variables are defined within the scope of the deployment, and they make sure that the project knows information before the build finishes on the server. Information such as what database to use, which domains to use, and other setup information can usually be found in these variables. We will cover this information in the final chapter.

File structure

A proper file structure is tremendously important in Meteor. We have found that the best way to work is with functional top-level modules. This means that every folder is a micro-service, and therefore can act on its own. This allows for a lot of modularity in the project, and it's very easy for others to understand what it is that you are trying to accomplish. In this section, we will cover this file structure and Meteor's special folders.

Let's look at a sample web application folder structure:

```
/online_shop
/online_shop/cart
/online_shop/cart/cart_route.coffee  #RUNS ON CLIENT AND SERVER
/online_shop/cart/client
/online_shop/cart/client/cart_view.jade  #RUNS ON CLIENT ONLY
/online_shop/cart/client/cart_controller.coffee
/online_shop/cart/client/cart.styl
/online_shop/cart/server
/online_shop/cart/server/cart_publisher.coffee  #RUNS ON SERVER
ONLY
/online_shop/cart/server/cart_methods.coffee
```

In this folder structure, `cart` is the micro-service, and it is composed of a route, view, controller, and publisher. The files placed under a `/client` directory will be published to the client and will only run on the client. The files placed under a `/server` directory will only run and be accessible on the server. If a file is placed in none of these directories, then the file will run on both the client and server. The expected structure goes like this:

```
/project_folder
/project_folder/_globals
  ./client/<global support function files>
  ./server/<global support function files>
  ./lib/collections/<collection>/<file>
  ./lib/collections/<collection>/server/<permissions file>

/project_folder/router
  ./client/layouts/<layout file>
  ./lib/<configuration file>
  ./lib/<middleware file>

/project_folder/<module>
  ./<route file>
  ./client/<template file>
```

```
./client/<logic file>
./client/<styles file>
./server/<publishers file>
./server/<methods file>
```

It's important to note that the `/lib` directory will always run before any other code does. Let's place our canonical file under the `/_globals/canonical/server` directory.

Let's create our first module: the router. Create the `/router/client/layout.jade` directory, and we will only have one layout throughout the project. Now let's code our layout:

```
//- LAYOUT.JADE
head
  title Online Shop
  meta(charset="utf-8")

  //- Allow saving to homescreen
  meta(name="apple-mobile-web-app-capable" content="yes")

  //- Do not try to detect phone numbers
  meta(name="format-detection" content="telephone=no")

  //- Make it mobile friendly
  meta(name="viewport" content="user-scalable=no, initial-
  scale=1.0, maximum-scale=1.0")

body

template(name="layout")
  +Template.dynamic(template=nav)
  +Template.dynamic(template=content)
```

Here, we have introduced the `Template.dynamic` component. This component may be used to render other templates dynamically by changing the value of a variable to the name of the template we want to render. We decided to use two variables—`nav` and `content`— that are controlled by the router. So, basically, the `content` variable will be changing to different strings that are equal to the names of our templates.

We will create our `landing` module in the next chapter to learn not only how to use the router but also how to properly subscribe to data.

Summary

We have addressed a lot of things in this chapter. We can now program faster because we have tools such as CoffeeScript, Jade, and Stylus to help us. Also, we have learned how to use templates, helpers, and events to work with our Meteor frontend. Understanding the event loop and the merge box has made us a bit more precautious when it comes to running complex, time-consuming operations. Finally, we began to build a project, and we adopted a folder structure that is going to make development quicker.

In the next chapter, we will cover two of the most important parts that make a Meteor application viable: Meteor publishers and Meteor subscribers. With these patterns, you will be able to produce sites that load quickly and that do not put too much strain on the server.

2
Publish and Subscribe Patterns

This is by far the most important chapter in this book. The way we control our publishers and subscribers is going to define how quickly our application responds in production. Publishers and subscribers are the link between our database and the client. The server uses publishers to publish information to the client, while the client requests information from the publishers by subscribing to them. This is all managed via the `Meteor.publish` and `Meteor.subscribe` functions. We should be able to produce the data that the client wants to see with two objectives in mind:

- Reduce stress from the server
- Send only the information that the client needs

This chapter will teach you the different patterns that you can use to attain these objectives for every template that you build. Here is an overview of the topics we will cover to understand these patterns:

- Template-level subscriptions
- Database relationships
- Publishing with relations
- Aggregation publishers
- External API publishers

Template-level subscriptions

This pattern attaches the `Meteor.subscribe` functions to templates. The key advantage of subscribing from the template is the ability to isolate the template and know that it still works when it is rendered.

Many Meteor developers attach their subscription methods to their routes. This means that the template will only render with the correct data at that particular route.

With this pattern, we will be able to reuse templates anywhere without worrying about data.

Setting up products for the online shop

Let's start by setting up a `Products` collection in MongoDB for our `online_shop` project. In *Chapter 1*, *Getting Started with Meteor* we learned that we need to place this definition under the `/globals/lib/collections` directory:

```
# /globals/lib/collections/products/products.coffee
@Products = new Mongo.Collection "products"

# fields:
#  name
#  description
#  sku
```

It's important to note that we are adding `@` at the beginning of the `Products` variable. This compiles into `this.Products`. In CoffeeScript for Meteor, we do this to define a globally scoped variable. This means that the `Products` variable now exists in every CoffeeScript file that we will create.

We need to add a permission file as well so that we can modify the collection from the console. The `allow`/`deny` functions are rules that add a layer of security to the collections. If an `allow` rule returns `true` for a given action, it will allow the change to pass. For now, we are going to set all the rules to allow everything. We will address permissions when we look at roles in *Chapter 4*, *Application Patterns*.

```
# /globals
# ./lib/collections/products/server/products_permissions.coffee
Meteor.startup ->
  Products.allow
    insert: -> true
    update: -> true
    remove: -> true
```

We use the `Meteor.startup` function to make sure that we have set up `Products` as a collection before we set the `allow/deny` rules. Now that we have a collection, let's make the landing page to show us a list of products. Let's build the **view** first in the `/products/client` directory:

```
//- /products/client/products.jade
template(name="products")
  h3.text-center products
```

This view is just a placeholder. We always need to create the template object before we can actually begin working with the publishers and subscribers.

Building the publisher

Let's construct a simple publisher for our view using the `Meteor.publish` function. This publisher will send only 10 documents to the subscribed clients. We'll discuss pagination in *Chapter 4*, *Application Patterns*.

```
# /products/server/products_pub.coffee
Meteor.publish "products_pub", ->
  Products.find {},
    limit:10
```

Subscribing to the publisher

This is where the magic begins. We are going to use the `Template.<template>.onCreated` function to subscribe to our publisher. The `onCreated` function runs whenever an instance of the template is created in the DOM. So, if we place a `Meteor.subscribe` function within this function, this will automatically resubscribe whenever the template is used. Let's give it a shot:

```
# /products/client/products.coffee
Template.products.onCreated ->
  @autorun =>
    @subscribe "products_pub"
```

The `this.autorun` function is, basically, a special version of the `Tracker.autorun` function. What does this do? This is meant to rerun whenever its dependencies change. The difference with this special function is that it will automatically stop subscriptions. You'll notice that we are binding `this.autorun` to the context by using `=>`, so we can use the special `this.subscribe` function within the `autorun` block. As expected, the `this.subscribe` functions exactly as `Meteor.subscribe` does. However, it plays a key part in automatically stopping subscriptions as well.

Wait! There's more. This snippet of code can be overwritten by other `onCreated` hooks. This means that if we were to define a second `onCreated` function, the first function will not run. We change this behavior using the `xorax:multiple-callbacks` package. This package, basically, concatenates `onCreated`, `onRendered`, and `onDestroyed` functions so that they cannot be overwritten. Let's take a look:

```
# /products/client/products.coffee
Template.created "products", ->
  @autorun =>
    @subscribe "products_pub"
```

We only need to change the first line of the code. This is done so that we can attach multiple functions to the `onCreated` template hook without overwriting any other functions.

Database relationships

Our collections will always be related in one way or another to other collections in our database. This topic is going to examine the three different types of relationships you should be thinking about when you are designing your database.

The shape that our database will take ultimately defines what our publishers are going to look like. If your data is all mashed up in just one or two different collections, you will very quickly find yourself struggling to filter data. If your data is spread too far between collections, the code will become difficult to maintain in the long run. So what is the solution to this problem?

The solution to database relationships is to understand how the data is going to be used in the client, how often it is going to be modified, and how large the set can be.

Let's build out the rest of our collections to get a notion of what good relationships look like.

One to one

A one to one database relationship describes how one MongoDB document in a collection will be linked to only one other MongoDB document in another collection. You can think of this as a JavaScript object inside another object.

In Meteor, you should create a one to one relationship for a subset of fields that you will use sparingly, that will have unique interfaces, or that you will use without necessarily requiring access to the product. With this relationship, it is going to be very easy to build an image uploader and a collage of images showing only `master_image`.

Start by adding a collection that will handle our products' images: `ProductImages`. We are going to assume that our frontend is going to have multiple types of images, each one being presented in a different part of the interface that we are building:

```
# /lib/collections/product_images/product_images_collection.coffee
@ProductImages = new Mongo.Collection "product_images"

# fields:
#  product
#  master_image
#  side_image
#  front_image
#  top_image
#  cart_image

# /lib/collections/
# ./product_images/server/product_images_permissions.coffee
Meteor.startup ->
  ProductImages.allow
    insert: -> true
    update: -> true
    remove: -> true
```

The `ProductImages` collection is going to contain a field named `product`. This field establishes our relationship by saving the unique `_id` field from the `Products` collection. This means that every time we want to publish a product with `ProductImages`, we need to query the database as well by searching for `Product_id` within `ProductImages`. So a helper would look like this:

```
# COFFEESCRIPT
Template.landing.helpers
  "images": ->
    ProductImages.findOne product:@_id
```

One to many

A one to many database relationship describes how one MongoDB document in one collection will be linked to many MongoDB documents in another collection. You can think of this as an array of JavaScript objects that exists under a particular field but the data is so complex that you need to separate it.

Now let's create an `Orders` collection. This collection is going to function as a cart. While your first instinct may be to create a `Carts` collection, you will quickly find that you are duplicating the order information (one for the cart and one for the order, once it is placed). We can easily identify whether an order is new or not by adding a `status` field:

```
# /lib/collections/
# ./orders/orders_collection.coffee
@Orders = new Mongo.Collection "orders"

# fields
  # status ("new","pending","complete")
  # total_products
  # subtotal
  # tax
  #   rate
  #   amount
  # discounts
  #   discount
  #   amount
  # total
  # date_created
```

Let's not forget to add permissions for the collection as well. We need to do this every time we create a new collection to secure the collection from hackers and ensure that the client can modify the collection accordingly. We will cover more on this in the *Security* section of *Chapter 4*, *Application Patterns*.

```
# /lib/collections/
# ./orders/server/orders_permissions.coffee
Meteor.startup ->
  Orders.allow
    insert: -> true
    update: -> true
    remove: -> true
```

Something is wrong with this, the products are not defined in the collection! We intentionally created an order summary collection that is going to have a one to many relationship with order details. We do not know how extensive the order actually might be. If we include a field that holds an array of products, this list will not only be difficult to manage, but also could potentially become large enough that it will crash the database.

Let's put the details of our order in a separate collection called `OrderDetails`:

```
# /lib/collections/
# ./order_details/order_details_collection.coffee
@OrderDetails = new Mongo.Collection "order_details"

# fields
  # order
  # product
  # price
  # quantity
  # subtotal
  # tax
  #   rate
  #   amount
  # total
  # discounts
  #   discount
  #   amount

# /lib/collections/
# ./order_details/server/order_details_permissions.coffee
Meteor.startup ->
  OrderDetails.allow
    insert: -> true
    update: -> true
    remove: -> true
```

Perfect! Now we can say one `Order` has many `OrderDetails`. In this scenario, each document in the `OrderDetails` collection represents a single product and the details about that product within the order. We have added the `order` field to identify to which specific order the details (or in this case, the products) belong to.

This is excellent design. By separating the details from the order, we are capable of controlling exactly which data the server sends down to the client. Remember, the goal here is to send the least amount of data possible to the client so that the site loads quickly and stress on the server is reduced. Managing inserts and updates as well becomes easier. As we do not need to deal with indexing a details array, we can simply use IDs to find and manipulate data. Now when the shop admin subscribes to the `Orders` collection, we can do this in such a way that the server only sends the data the admin needs to see the orders. Clicking on an order would then lead to subscribing to only that orders' details.

Many to many

A many to many database relationship describes how many MongoDB documents in one collection will be linked to many MongoDB documents in another collection. You can think of this as a table where you repeat information and only change the information of one field for every row so that you can filter the information by that row.

These types of relationships require a **mapping table**. In the case of MongoDB, a mapping table is a separate collection. A mapping table is the part that repeats information in every row without actually duplicating entries; it simply makes pairs between each collection ID.

In the example that we are going to program, we want to create a many to many relationship between products and tags because a product can have many tags and a tag can have many products. The mapping table in this case is going to save each ID pair. So if a product has two tags, there will be two entries in the mapping table with the same product ID but each with a different tag ID. If a tag is a part of two products, then the mapping table will have two entries with the same tag ID but each one with a different product ID.

It's important to note that these types of relationships are often overlooked because of their complexity. If for some reason you find yourself trying to match two collections with arrays from each collection, you are definitely attempting a "many to many" relationship.

To explain this relationship, we are going to create a `Tags` collection for our products:

```
# /lib/collections/
# ./tags/tags_collection.coffee
@Tags = new Mongo.Collection "tags"

# fields
#  name

# /lib/collections/
# ./tags/server/tags_permissions.coffee
Meteor.startup ->
  Tags.allow
    insert: -> true
    update: -> true
    remove: -> true
```

You will notice that in this type of relationship the `Tags` collection has nothing to associate to a product, but if it did, it would have to be an array of products to which the tag belongs to. To establish the relationship, we will create the mapping table, that is, the `ProductsTags` collection:

```
# /lib/collections/
# ./products_tags/products_tags_collection.coffee
@ProductsTags = new Mongo.Collection "products_tags"

# fields
#  product
#  tag

# /lib/collections/
# ./tags/server/products_tags_permissions.coffee
Meteor.startup ->
  ProductsTags.allow
    insert: -> true
    update: -> true
    remove: -> true
```

This collection allows us to have any combination of relations between `Products` and `Tags`. Suppose we want to see all the products related to one tag. In this case, we will first query the mapping table for that tag and then query the tags using the result.

Beware! Many to many relationships can be difficult to spot. If you are stuck at any point during your database design process, do not forget to consider this possibility.

Publishing with relations

We understand how our collections are related, but how can we make it easy to publish data with these relationships?

In Meteor, it can be problematic to publish relationships because of reactivity and the way publishers work. You would expect that by simply making two queries to two related collections and returning an array will publish a perfectly reactive collection. This is not the case. A `Meteor.publish` function does not rerun when dependencies change. This means that if a relationship is broken, the related document will remain published, or worse if a new relationship is made by another client, the related data will not publish.

To take care of database relationships and reactivity in Meteor, we use the `lepozepo:publish-with-relations` package. This package automatically takes care of subscribing to new data in the most efficient way possible when relationships are broken. If you are familiar with MySQL, this package makes JOINs a breeze.

Publishing products with images (one to one)

We will be working with the `/products` directory, this module will be our landing page. First, let's set up a route for our view:

```
# /products/products_route.coffee
FlowRouter.route "/",
  name:"products"
  action: ->
    FlowLayout.render "layout",
      content:"products"
```

We are using the `FlowRouter.route` function to define the path of our landing page and the `FlowLayout.render` function to define which layout template to use. You will notice that the `FlowLayout.render` function is taking in two parameters: the first one defines which layout template to render, and the second one defines where to render within that layout template.

Now we can work on the publisher. The goal is to publish 10 products paired with only their `master_image`:

```
# /products/server/products_pub.coffee
Meteor.publish "products", ->
  @relations
    collection:Products
    options:
      limit:10
    mappings:[
      {
        key:"product"
        collection:ProductImages
      }
    ]

  @ready()
```

Notice the `mappings` key of the object that we are passing through the `@relations` function. All the objects inside the array must have at least a `collection` key. Optionally, they can take `key` and `foreign_key`. In this case, we use `key` to express that the `_id` field in the `Products` collection is equal to the string of the `product` field in the `ProductImages` collection. This is the most efficient way to publish data. The package will automatically make sure that all changes to the collections are reflected in real time.

The `Meteor.publish` function has one peculiarity: when data changes, the cursor will react accordingly, but the function holding the cursor will not. The impact of this is obvious when it comes to creating relationships. Let's look at what our code would look like without using a package to handle our relationships:

```
# DO NOT CODE THIS INTO YOUR PROJECT
products_cursor = Products.find {},limit:10

# Make products an array with all the _ids
products = products_cursor.map (product) ->
  product._id

# Find their images
images_cursor = ProductImages.find
  product:
    $in:products

# Return the cursors in an array
[products_cursor,images_cursor]
```

In this case, suppose that for some reason the order of the `Products` collection changes. This will make `products_cursor` change reactively as you would expect on the client, but because there were no changes to the `ProductImages` collection and `Meteor.publish` does not rerun when there is a change in one of its dependencies, the images of the newly published products will not be published reactively!

Publishing orders with details (one to many)

Now we will work on our `Orders` and `OrderDetails` collections. Let's set up our template, route, and subscriber for the function that creates orders. We can name this module as `cart` and store it under the `orders` directory:

```
# /orders/cart/client/cart.jade
template(name="cart")
  h3.text-center cart
```

```
# /orders/cart/client/cart.coffee
Template.created "cart", ->
  @autorun =>
    order = Session.get "cart.order"
    @subscribe "cart",
      order:order
```

Notice that the `order` variable is going to get our order `_id` by using the `Session` variables. We do this because the `Session` variables are reactive; if for some reason the value changes, this makes sure that the subscriber reruns. The subscriber reruns because the variable is defined within the `@autorun` function.

Also, we are passing an object as an argument to the `@subscribe` function so that the publisher knows what order we are talking about:

```
# /orders/cart/cart_route.coffee
FlowRouter.route "/cart",
  name:"cart"
  action: ->
    FlowLayout.render "layout",
      content:"cart"
```

We need to publish three different collections to have all the information that our cart needs: `Orders`, `OrderDetails`, and `Products`. This publisher can follow the same logic that we found in the first publisher:

```
# /orders/cart/server/cart_pub.coffee
Meteor.publish "cart", (ops={}) ->
  if ops.order and not _.isEmpty ops.order
    @relations
      collection:Orders
      filter:
        _id:ops.order
        status:"new"
      mappings:[
        {
          key:"order"
          collection:OrderDetails
          options:
            limit:25
          mappings:[
            {
              foreign_key:"product"
              collection:Products
            }
          ]
```

```
        }
      ]

  @ready()
```

This publisher is predefining defaults for the function by passing `ops={}` within the parameters (another useful shortcut that CoffeeScript provides). After making sure that `ops.order` exists and is not a blank string, we establish our relationships. We need to make sure that the status of `Order` has the `"new"` value for security, so we hardcode it in the `filter` key.

Notice that in this case, we are making use of `foreign_key`. This indicates that the `OrderDetails` collection has a `product` field that contains a string equal to the `_id` field of the `Products` collection.

For now, we are going to limit `OrderDetails` to `25`.

Publishing a tag with products (many to many)

We definitely want to be able to filter our products via tags. We can follow the same pattern from the previous topic. Let's modify our `products` module.

First, our subscriber needs to reactively change to an array of tags _ids. We can use a `Session` variable for now to set an array of tag _ids so that we can easily modify this directly from the console:

```
# /products/client/products.coffee
Template.created "products", ->
  @autorun =>
    # tags is an array of tag _ids
    tags = Session.get "products.tags"
    filter = {}

    if tags and not _.isEmpty tags
      _.extend filter,
        tags:tags

    @subscribe "products", filter
```

We use underscore's `_.extend` function to make sure that the `filter` variable has a `tags` key, if `tags` exist. Now our publisher is going to get a bit more complex:

```
# /products/server/products_pub.coffee
Meteor.publish "products", (ops = {}) ->
  limit = 10

  if ops.tags and not _.isEmpty ops.tags
    @relations
      collection:Tags
      filter:
        _id:
          $in:ops.tags
      mappings:[
        {
          collection:ProductsTags
          key:"tag"
          mappings:[
            {
              collection:Products
              foreign_key:"product"
              options:
                limit:limit
              mappings:[
                {
                  collection:ProductImages
                  key:"product"
                }
              ]
            }
          ]
        }
      ]

  else
    @relations
      collection:Products
      options:
        limit:limit
      mappings:[
        {
          key:"product"
```

```
            collection:ProductImages
          }
        ]

    @ready()
```

While the code may seem long, the logic is easy to understand. First, we filtered our `Tags` collection by using the `tags` array that we had defined in our subscriber. Then we chained the relations using a combination of `key` and `foreign_key`. The function that the mapping table (`ProductsTags`) serves is clear in this scenario.

Key, foreign key, options, and filter

The core concepts to understand from the `lepozepo:publish-with-relations` package are simply the options provided within the `@relations` function.

Both `key` and `foreign_key` default to the `_id` field. The `key` always makes reference to a field within the collection while the `foreign_key` always makes reference to a field of the parent collection.

The `options` and `filter` options are equivalent to the second and first arguments (respectively) of a Meteor - MongoDB query: `Products.find(filter,options)`.

There are many packages that work similar to `lepozepo:publish-with-relations`, and we are not exploring them in this book, but they are certainly worth keeping an eye on: `reywood:publish-composite`, `lepozepo:reactive-publish`, and `cottz:publish-relations`. I have found the last package to be the best of the bunch because it is simple to use and forces the developer to create smarter database relationships.

Aggregation publishers

Sometimes our database has a considerable amount of information that we want to synthesize. Some developers choose to publish all the information to the client and have the client synthesize it. This, as we have learned so far, can have a negative impact on performance. Other developers might use `Meteor.method` to return the synthesized data. This is definitely better for the client, but it will take a toll on our server if the computation is large.

The best way to handle a problem like this is to use MongoDB's aggregation framework to take the hard work of the calculation to our database, and then we can pair the results with the `Meteor.publish` special functions: `@added`, `@changed`, and `@removed`.

The aggregation framework

MongoDB's aggregation framework uses the concept of a pipeline to process data. A pipeline is, basically, a series of steps that Mongo is going to follow to produce the data you need.

We have installed support for the aggregation framework by adding `meteorhacks:aggregate`. This gives us access to all of the framework's server-side commands. The most common commands that you will end up using are `$match`, `$group`, and `$project`.

Let's build a publisher for our `dashboard` module and start by building the template:

```
# /dashboard/client/dashboard.jade
template(name="dashboard")
  h3.text-center dashboard

# /dashboard/client/dashboard.coffee
Template.created "dashboard", ->
  @autorun =>
    @subscribe "dashboard"

# /dashboard/dashboard_route.coffee
FlowRouter.route "/dashboard",
  name:"dashboard"
  action: ->
    FlowLayout.render "layout",
      content:"dashboard"
```

Now we need to build a collection that exists only on the client. Our server will communicate with this collection manually. We specifically place this collection on the client side because we do not want the server to record data onto the database. We can append the collection to the (`/dashboard/client/dashboard.coffee`) client controller file:

```
# /dashboard/client/dashboard.coffee
@_dashboard = new Mongo.Collection "_dashboard"

Template.created "dashboard", ->
  @autorun =>
    @subscribe "dashboard"
```

As this collection is client-side, we have differentiated it with an underscore at the beginning of the name. This, of course, is not required, but it helps prevent duplicate collection names.

Our goal is to publish the sum of the total and subtotal for the `"pending"` orders. The steps for our pipeline are simple:

1. Filter the collection to show orders with status equal to `"pending"`.
2. Sum the totals of every filtered order:

```
# /dashboard/server/dashboard_pub.coffee
Meteor.publish "dashboard", ->
  totals = Orders.aggregate [
      {
        $match:
          status:"pending"
      }
      {
        $group:
          _id:null
          total:
            $sum:"$total"
          subtotal:
            $sum:"$subtotal"
          discount:
            $sum:"$discount.amount"
      }
    ]

  console.log totals
```

To use the aggregation framework, you use the `.aggregate` function that is attached to the collection object; `Orders`, in this case. This function only takes an array as a parameter—this is because the array is the ordered set of steps that the framework is going to follow. Each step is represented by an object within the array and always begins with an operator.

Here we have decided to use `$match` to filter the orders to find the pending orders, and then we have used `$group` to accumulate the values of `total` and `subtotal`. Notice that the `$group` expression has a mandatory `_id` key. This key defines how we want to group the collection. By setting the `_id` key to `null`, we are stating that we want all the documents in the collection grouped into one single object.

`$sum` is an accumulator operator. When you use an operator like this, you can access a document property by using the money sign (`$`) followed by the name of the field within a string. Also, you can access objects within objects using the dot notation (`"$discount.amount"`).

The result of `totals` is an array containing a single object with the keys: `total`, `subtotal`, and `discount`.

Publishing the results

Publishing the results is much easier than it seems. We only need to use the `@added` function that is bound to `Meteor.publish`. The `@added` function, basically, informs the subscriber that data has been added to the published set:

```
# /dashboard/server/dashboard_pub.coffee
Meteor.publish "dashboard", ->
  totals = Orders.aggregate [
      {
        $match:
          status:"pending"
      }
      {
        $group:
          _id:null
          total:
            $sum:"$total"
          subtotal:
            $sum:"$subtotal"
          discount:
            $sum:"$discount.amount"
      }
    ]

  if totals and totals.length > 0 and totals[0]
    @added "_dashboard","totals",totals[0]
```

The last two lines make sure that the `totals` array exists, and if it does not, then we publish the first object within the array to our `_dashboard` collection. The `@added` function has three required parameters. The first parameter is the name of the collection, the second is the `_id` of the document, and the third is the document.

Notice that these kinds of publishers are not reactive, which means that we do not need to add a `@changed` or `@removed` function. We can take this a step further though. Instead of creating a collection for every module that needs an aggregation publisher, we can create one master collection that manages all our aggregation publishers.

First, we remove our `_dashboard` collection and create a new `Aggregate` collection:

```
# /dashboard/client/dashboard.coffee
Template.created "dashboard", ->
  @autorun =>
    @subscribe "dashboard"
```

```
# /globals/lib/collections/aggregate/client/aggregate.coffee
@Aggregate = new Mongo.Collection "aggregate"
```

Now, we need to modify our publisher:

```
# /dashboard/server/dashboard_pub.coffee
Meteor.publish "dashboard", ->
  totals = Orders.aggregate [
      {
        $match:
          status:"pending"
      }
      {
        $group:
          _id:null
          total:
            $sum:"$total"
          subtotal:
            $sum:"$subtotal"
          discount:
            $sum:"$discount.amount"
      }
    ]

  if totals and totals.length > 0 and totals[0]
    @added "aggregate","dashboard.totals",totals[0]
```

With this, we can now access our values by calling the following function on our client-side console:

```
Aggregate.findOne "dashboard.totals"
```

It's important to understand that as soon as we leave the route, the subscriber will be stopped and the `aggregate` collection will be cleared. This default behavior gives this collection a lot of flexibility.

Why is using the aggregation framework with a publisher more effective than with `Meteor.method`? The Meteor methods are designed to trigger critical functions within our server and respond with a simple message. On the other hand, publishers are designed to control data sets. You will very quickly find that publishers are easier to control and optimize than methods.

External API publishers

These types of publishers should be avoided. They are great to get raw data from external services such as **Stripe** or **Square**, but they tend to be little bit slower because this involves actively communicating with another server. When we are integrating with other servers, we should always build a separate synchronization server. We will talk about APIs in another chapter.

Still, this publishing pattern can be useful in edge cases, so it is important to know that this option exists.

The HTTP package

HTTP is a protocol for collaborative systems; it is the protocol that allows users to connect to web pages. The HTTP protocol can be used to access data from other servers from our own server.

We are going to use Meteor's integrated HTTP `Request` module to communicate with Stripe's servers. We chose to integrate with Stripe because it is a payment processor that is easy to integrate and more reliable than most other payment processors in the market. We have added this package when we ran `meteor add http`. This module has all the functions that you would expect: `.get`, `.post`, `.put`, and `.del`. For this topic, we are only going to cover the `.get` function.

Our goal is to get data from Stripe. Start by creating a free account in Stripe. After you have created your account, go to your dashboard and set it to `"test"`.

Now create a payment by clicking on **+ Create Payment**. Use the following data for the payment:

- **Amount**: `10`
- **Card Number**: `4242 4242 4242 4242`

In our test environment, we do not need to add CVC, and Stripe automatically sets the expiration date to a year from today.

Click on **Account Settings** and copy your test secret key. We are going to use this to authorize our request.

We can start by creating a new publisher under `/dashboard_pub.coffee`; we are only going to get the last three charges—we will modify this later:

```
# /dashboard/server/dashboard_pub.coffee
Meteor.publish "dashboard", ->
  ...
```

```
Meteor.publish "latest_sales", ->
  @unblock()

  HTTP.get "https://api.stripe.com/v1/charges?limit=3",
    headers:
      "Authorization":"Bearer <TEST SECRET KEY>"
    (error,result) =>
      if not error
        _.each result.data.data, (payment) =>
          @added "aggregate",
          "dashboard.sales.#{Meteor.uuid()}",payment
      else
        console.log error
```

The `@unblock` function that we are using here has been made available via the `meteorhacks:unblock` package. It works the same way as the `@unblock` function does in `Meteor.methods`; this makes sure that the publisher does not block the server while it is waiting for information from Stripe to arrive. Unblocking the aggregation publishers is crucial! If we do not unblock the publisher, then we risk the client becoming unresponsive when the user navigates away from the page.

You might be asking yourself: can't we just make the HTTP request on the client and not worry about blocking the server? No, we cannot. If you were to run this function on the client, you would have to expose your secret key, and this is a major security flaw.

The `HTTP.get` function has three important fields: the URL, the `options` object, and the callback function. The URL is the address that Stripe provides in their API documentation—in this case, we have used the `/charges` URL and passed a parameter `limit` to get the last three sales. The options object is used to pass all the information that the request needs. In this case, we will use the `headers` key to set our `Authorization` header. There are several keys that the options object can take; you can find these at `docs.meteor.com`. Our callback function receives the result from the request. As with most functions in Meteor, it returns two arguments; the first one is an error object that is undefined if there is no error, while the second one is the actual result.

In this case, the data that we are looking for is contained within `result.data.data` as an array. We can then easily publish the data with our `@added` function. Notice that we are binding both the callback and the `_.each` function, so we can have access to `@added`.

Let's subscribe to our new publisher to see our data:

```
# /dashboard/client/dashboard.coffee
Template.created "dashboard", ->
  @autorun =>
    @subscribe "dashboard"
    @subscribe "latest_sales"
```

Try running `Aggregate.find().fetch()` in your console to see the results.

Summary

In this chapter, we learned how to isolate subscribers from templates using template hooks. Also, we learned how to optimize the structure of our database depending on how we are going to use it. When it comes to publishers, we learned how to return every possible data structure without breaking reactivity. We covered aggregation publishers and learned how to synthesize data before we publish it, without hurting the performance of the server.

In the next chapter, we will cover a handful of frontend techniques that will help us to keep our code flexible and our frontend looking great.

3
Front-end Patterns

This chapter will cover a useful set of patterns that will speed up your frontend workflow. Meteor makes it easy to build rich frontend experiences, but we have to be careful with the number of computations our interface requires to function. Too many computations will slow down mobile devices. In this chapter, we are going to learn how to keep our computations short, DOM simple, and animations effective. You will learn the following topics:

- Responsive design
- Super helpers
- Variable types
- Quick forms
- Loading
- Animations
- Search engine optimization

Responsive design

An application is always judged by how it looks. Now that we know how to make it perform, we need to learn how to make it look good. Nowadays, it is common practice to build the frontend while keeping the following two things in mind:

- **Minimalism**: This style of design makes your site concise by exposing only the necessary elements to have the site functioning. Less is more.
- **Responsive**: This style of programming makes your site usable everywhere, no matter how big the screen is. The HTML markup will always adapt depending on the width of the screen.

Here, we are only going to cover how to program a responsive frontend because minimalism is a vast and highly debatable topic.

General settings

The first layer of customization begins with the `kyleking:customizable-bootstrap-stylus` package that we have already installed. Follow these steps to set this up:

Create the following directory:

`/_globals/client/bootstrap/custom.bootstrap.json`.

(Leave the JSON file empty.)

Run the Meteor project using this:

```
meteor
```

You will notice three new files have been added. These files will compile at runtime to produce your version of Twitter Bootstrap.

Open `/_globals/client/bootstrap/custom.bootstrap.json`, and make sure that all variables are set to `true`:

```
{"_route":"/_globals/client/bootstrap/custom.bootstrap.json"},
{
  "modules" : {
    "variables": true,
    "mixins": true,

    "normalize": true,
    "print": true,
    "glyphicons": true,

    "scaffolding": true,
    "utilities": true,
    "type": true,
    "code": true,
    "grid": true,
    "tables": true,
    "forms": true,
    "buttons": true,

    "component-animations": true,
    "dropdowns": true,
    "button-groups": true,
```

```
      "input-groups": true,
      "navs": true,
      "navbar": true,
      "breadcrumbs": true,
      "pagination": true,
      "pager": true,
      "labels": true,
      "badges": true,
      "jumbotron": true,
      "thumbnails": true,
      "alerts": true,
      "progress-bars": true,
      "media": true,
      "list-group": true,
      "panels": true,
      "responsive-embed": true,
      "wells": true,
      "close": true,

      "modals": true,
      "tooltip": true,
      "popovers": true,
      "carousel": true,

      "responsive-utilities": true
    }
  }
```

By setting all the variables to `true`, we are reeling in all the functionality that `bootstrap` has. If there is anything you do not want to use from the framework, you can deactivate it by setting it to `false` in this file.

You will need to manually restart Meteor for these changes to take effect. So press *Ctrl* + *C* on your terminal to stop Meteor and run the `meteor` command again.

Now we can edit `bootstrap`. Start by opening the `/_globals/client/bootstrap/custom.bootstrap.import.styl` directory. Here, you are going to find a library of the Stylus variables that we can play with.

Let's remove rounded corners and modify button border colors:

```
// /_globals/client/bootstrap/custom.bootstrap.import.styl

...

// Line #114
$border-radius-base ?=       0px
$border-radius-large ?=      0px
$border-radius-small ?=      0px
```

```
...

// Line #156
$btn-default-color ?=            #333
$btn-default-bg ?=               #fff
$btn-default-border ?=           $btn-default-color

$btn-primary-color ?=            #fff
$btn-primary-bg ?=               $brand-primary
$btn-primary-border ?=           $btn-primary-bg

$btn-success-color ?=            #fff
$btn-success-bg ?=               $brand-success
$btn-success-border ?=           $btn-success-bg

$btn-info-color ?=               #fff
$btn-info-bg ?=                  $brand-info
$btn-info-border ?=              $btn-info-bg

$btn-warning-color ?=            #fff
$btn-warning-bg ?=               $brand-warning
$btn-warning-border ?=           $btn-warning-bg

$btn-danger-color ?=             #fff
$btn-danger-bg ?=                $brand-danger
$btn-danger-border ?=            $btn-danger-bg
```

What if we need to use bootstrap mixins and variables for custom style sheets? We can use the `@import` command whenever we need to gain access to everything:

```
@import "_globals/bootstrap/client/custom.bootstrap.import.styl"
```

Let's customize our products page by adding a large promoter. Rewrite the products template with a new `#promoter` header:

```
//- /products/client/products.jade

template(name="products")
  div#products.template
    header#promoter
      h3 Your Brand
```

Before we write our product styles, let's create a global style sheet for any application-wide styling at `/_globals/client/main.styl`. Here, we are going to make sure that our initial views are set correctly so that we may properly make use of relative units:

```
// /_globals/client/main.styl

html, body, #__flow-root, #__flow-root > .template
```

```
html, body, body > .template
    height:100%
```

With this set of rules, we are basically making sure that everything has the height of the viewport including our first `.template` DOM element. This technique allows us to have relative percentage heights work as expected.

We also need to set the root DOM element for BlazeLayout. Let's do this in a client/server configuration file:

```
# /_globals/router/config.coffee

if Meteor.isClient
  BlazeLayout.setRoot 'body'
```

Now, create the directory for the `products` styles: `/products/client/products.styl`. We will start by importing our variables into the file and setting some simple rules to make the promoter look good:

```
// products/client/products.styl
@import "_globals/bootstrap/custom.bootstrap.import.styl"

#products
  #promoter
    background: $brand-primary
    height: 80%
```

Great, this sets up our `promoter` section for any promotional images that we may want to add, and the site looks decent (but not good enough yet) on mobile.

Bootstrap

Bootstrap is a library of CSS rules that makes development of simple layouts quick, thanks to the library of classes that it includes.

Let's extend our products template to upgrade our `#promoter` header and add a `#features` section. We will use bootstrap to manage the grid in this case:

```
//- /products/client/products.jade

template(name="products")
  div#products.template
    header#promoter
      div.container
        div.row
```

```
            div.col-xs-12
              h3 Your Brand

      section#features
        div.container
          div.row
            div.col-xs-12.col-sm-4
              h3.text-center
                span.fa-stack.fa-lg
                  i.fa.fa-square.fa-stack-2x.text-primary
                  i.fa.fa-users.fa-inverse.fa-stack-1x
              h5.text-center 24 Hour Support

            div.col-xs-12.col-sm-4
              h3.text-center
                span.fa-stack.fa-lg
                  i.fa.fa-square.fa-stack-2x.text-primary
                  i.fa.fa-truck.fa-inverse.fa-stack-1x
              h5.text-center Next Day Delivery

            div.col-xs-12.col-sm-4
              h3.text-center
                span.fa-stack.fa-lg
                  i.fa.fa-square.fa-stack-2x.text-primary
                  i.fa.fa-bomb.fa-inverse.fa-stack-1x
              h5.text-center Mind Blown Guarantee
```

We are doing quite a few things here but nothing complex. First, we wrapped the contents of our `#promoter` header in bootstraps' grid. Bootstrap grids work only as advertised when they are under a `row` class. The `container` class controls our page-wide breakpoints.

Bootstrap columns are set using the `col-` class prefix followed by the target screen size, then an optional offset indicator, and finally, the number of columns that the element will span or offset:

```
div.col-<screen size>-<optional offset>-<number of columns>
```

Class	Breakpoint	Offset
`col-xs-*`	<768px	`col-xs-offset-*`
`col-sm-*`	≥768px	`col-sm-offset-*`
`col-md-*`	≥992px	`col-md-offset-*`
`col-lg-*`	≥1200px	`col-lg-offset-*`

Bootstrap has a total of 12 columns (this can be modified in our `variables` file); each column supports offsetting and nesting too. Remember, the smallest screen size will override all other screen sizes if the other screen sizes are not defined. So, if you want three columns of every size, you will only need to define it for the smallest size screen. In this example, we are separating our `#features` section into three columns of every size above 768 px and treating this as a single column in anything below 768 px.

If at any point you need to nest bootstrap columns, then you will need to place these new columns under a `row` class. The following example centers a six-column long `div` inside another six-column long `div`:

```
div.row
  div.col-xs-6
  div.col-xs-6
    div.row
      div.col-xs-6.col-xs-offset-3
```

We are using Font Awesome to create some icons for our `#features` section as well. Here, we are using one of the Font Awesome's lesser known **icon stacking** feature. To use this feature, you only need to wrap a set of Font Awesome icons with a `span.fa-stack` element.

Font Awesome has a list of about 500 icons. If you want to use a different set of icons, feel free to check out the list of icons online at `http://fortawesome.github.io/Font-Awesome/icons`.

Now let's add a little bit of Jeet and Rupture to our frontend.

Jeet grid systems with Rupture

The version of Stylus that we have installed (`mquandalle:stylus`) comes packaged with a lot of useful mixins, the most notable ones being **Jeet** and **Rupture**. Jeet is a grid system that is superior to Twitter Bootstrap because it is more flexible, and Rupture is a set of functions that simplifies media queries. Knowing how to leverage both Jeet and Twitter Bootstrap will increase the quality of your frontend designs.

Let's patch up our `#promoter` header to make our page more user friendly for users that are in landscape mode on smaller devices:

```
// products/client/products.styl
@import "_globals/client/bootstrap/custom.bootstrap.import.styl"
@import "jeet" // Add jeet
```

```
#products
  #promoter
    background: $brand-primary
    height: 80%

    +below($screen-sm-min) // If the screen width is smaller than
    -sm
      +portrait() // And the device is in portrait mode
        height: 30%

      +landscape() // And the device is in landscape mode
        col(1/3)
        height: 100%

  #features
    +below($screen-sm-min) // If the screen width is smaller than
    -sm
      +landscape() // And the device is in landscape mode
        col(2/3)
        height: 100%

        .container
          width:100% // Override bootstrap container
```

Rupture adds several mixins to easily control the `@media` queries: `below`, `above`, `between`, `landscape`, `portrait` and many more. You will find yourself using `below`, `landscape`, and `portrait` more often than the rest. To use these mixins in Stylus, we simply prefix a plus sign (`+`) before the command.

In this example, we are using `+below($screen-sm-min)` to define the CSS rules for all the devices whose widths are below bootstrap's `-sm` breakpoint. Then we will use `+portrait()` and `+landscape()` to define CSS rules for portrait and landscape modes.

Next, we will use Jeet to control master columns using the `col()` mixin. This mixin takes in a fraction that defines the width of the element as a fraction. Consider the numerator as the number of columns that you want the element to take up and the denominator as the total number of columns:

```
col(<column width>/<number of columns>)
```

In this example, when we switch to landscape mode on a small device, our `#promoter` header is one column wide out of three, while our `#features` section is two columns wide out of three.

Super helpers

In Meteor, you will quickly find yourself repeating helpers among templates for simple things such as formatting. We can prevent repetition by creating a global dictionary of functions—this is what we call a super helper. To do this, we are going to tap into Meteor's rendering engine—Blaze.

It is important to understand that Blaze is deeply integrated with Spacebars, and Spacebars is Meteor's updated version of HandlebarsJS. HandlebarsJS is a JavaScript templating engine that enables the use of helpers and components on the frontend using `{{}}`. This legacy means that Spacebars has a lot of the HandlebarsJS functionality. So, much of the documentation found in HandlebarsJS applies to Meteor helpers as well.

Defining a Blaze helper

Meteor exposes the `Template.registerHelper` function to create global helpers. Let's create something to help us format money:

```
# /_globals/client/formatters.coffee

Template.registerHelper "format_money", (value) ->
  if _.isNumber value
    "$#{(value / 100).toFixed(2)}"
```

As you can see, the syntax for a global helper is exactly the same as it would be for a template helper. Now, we can use this helper in any template. Let's create a product template under our `/products` directory for our `products` template and try it out here:

```
//- /products/client/product.jade

template(name="product")
  div.col-xs-12.col-sm-4.col-md-3
    article#product
      h5 {{name}}

      div.offer
        span.price {{format_money price}}
```

We will need to patch our products template as well:

```
//- /products/client/products.jade

template(name="products")
```

```
  div#products.template
    header#promoter
      ...
    section#features
      ...

    br
    section#featured_products
      div.container
        div.row
          each products
            +product
```

Let's add some temporary entries to our products collection. We will use these entries throughout the rest of the chapter:

```
@Products = new Mongo.Collection "products"

if Meteor.isServer
  if Products.find().count() is 0
    Products.insert
      name:"Nuka Cola"
      price: 1099

    Products.insert
      name:"1up Soda"
      price: 999

    Products.insert
      name:"JuggerNog"
      price: 899

# /products/client/products.coffee

Template.created "products", ->
  ...

Template.products.helpers
  products: ->
    Products.find()
```

Note that in this data schema we are defining `price` in cents. This is a common practice when dealing with money in JavaScript to avoid arithmetic floating point issues. What does this mean? Go to your web browser's console, and run the following operation: `0.1 + 0.2`. Oddly enough, JavaScript does not respond with `0.3`; it responds with `0.3000000000004` instead. The reason for this is JavaScript's internal 64-bit floating point representation of decimal numbers. This means that JavaScript does not fully understand what a decimal number is. This is why, we do not use decimal numbers when we deal with money, we just use the smallest available denomination.

Making a global dictionary

Wait, we can still make this formatter more widely available in our application. Let's turn this into a dictionary and share it with our controllers. In other words, we want to be able to use a helper such as `{{format.money amount}}` and use this in our CoffeeScript in the form of `format.money amount` as well:

```
# /_globals/client/formatters.coffee

@format =
  money: (value) ->
    if _.isNumber value
      "$#{(value / 100).toFixed(2)}"

Template.registerHelper "format", ->
  format
```

First, we create an object that holds our definitions, and we make it global using `@` (which is equal to `this`). Then we place this object of definitions inside a global helper. Simple! Now we know how to build a super helper that we can use anywhere. To test this, type `format.money(1099)` in your browser's console; this should return **"$10.99"**.

Variable types

There are four types of variables that we need to understand to optimize our frontend workflow: **session**, **persistent**, **file scope**, and **ReactiveVar**. With these variables, we can build dynamic sites easily based on how the user is interacting with the view.

The session variables are variables that exist only during a session. A session begins whenever a user hits the site. These variables can be shared across routes and are reactive.

The persistent variables are variables that are stored in local storage so that they can be read even after the session closes.

The file scoped variables are variables that only exist within the scope of the file. These are generally not reactive.

The ReactiveVar variables can be scoped to the file or globally and are reactive. These variables do not persist the way session variables do.

Session variables

Some Meteor developers believe that session variables are overused. This is definitely true, if you are not managing them. For the most part, session variables should be used to handle data that is shared across multiple routes in your application, have no place in your database, need reactivity, and persist through a hot code reload.

If you stop and think about it, not many things require all of these attributes. At first, you may rely on them heavily to control your frontend, but as your frontend becomes more minimal, you will realize there is no need for them.

Still, if you are going to use them, you should follow two rules to maintain them, as shown here:

- Template-based taxonomy
- Clear them if you are not going to use them

The taxonomy of session variables should always follow the `<template.variable>` pattern to keep them from polluting each other. Suppose we want to control a global alert using session variables, the session variable would look like this:

```
Session.set "products.alert",true
```

Once our user leaves the view, we should clear the session variable. The `clear` function is added by the `u2622:persistent-session` package that we installed in *Chapter 1*, *Getting Started with Meteor*:

```
# Clear "products.alert" variable after "products" template is
destroyed
Template.destroyed "products", ->
  Session.clear "products.alert"

# Clear all variables after the "products" template is destroyed
Template.destroyed "products", ->
  Session.clear()
```

Do not use session variables along with Meteor methods to obtain server data. While the method works, it makes your data vulnerable and easy to modify on the client.

Also, try to stay away from using session variables that are composed of complex objects with multiple keys or arrays. You will quickly find yourself using lesser lookup tools such as `_.find` and `_.findWhere` to use the data.

Persistent variables

Persistent variables are session variables that persist through a session. In Meteor, a common session variable will reset if the user refreshes the page. Persistent variables leverage AmplifyJS, which in turn uses HTML5 local storage and fallbacks to store the variable inside the user's browser. These types of variables exist because of the `u2622:persistent-session` package.

We can use this variable to keep track of our users' orders, in case they leave the site and are not logged in. So, if the user opens up an order and fills it with items, they can come back to it later without making a user account on our site.

Let's create an `add-to-cart` button that will leverage the persistent variables:

```
# /products/client/product.coffee

Template.product.events
  "click button.add-to-cart": (event) ->
    # Get the session variable
    order_id = Session.get "global.order"
    order = Orders.findOne order_id

    # Insert Order if it doesn't exist
    unless order
      order_id = Orders.insert
        status:"new"
        total_products:0
        subtotal:0
        total:0
    else
      order_id = order._id

    # Set the session variable for future reference
    Session.setPersistent "global.order",order_id
```

```
# Find the order
order = Orders.findOne order_id

# Check for details on this product
detail = OrderDetails.findOne
  product:@_id
  order:order._id

if detail
  # Increase by one if the details exist
  OrderDetails.update detail._id,
    $inc:
      quantity:1

  Orders.update order._id,
    $inc:
      total_products:1
      subtotal:@price
      total:@price
else
  # Insert if details do not exist
  OrderDetails.insert
    quantity:1
    product:@_id
    order:order._id

  Orders.update order._id,
    $inc:
      total_products:1
      subtotal:@price
      total:@price
```

We will need to build a publisher for this to work, but we have to make a choice. As we have both `products` and `product` template, each one could subscribe to a publisher and pull in the same data. Should we subscribe from the `product` or the `products` template? Let's do both! Extending the products publisher will ensure that we get all the data that the template needs while the attached publisher will help us manage our "product details" page and promos when we need it to:

```
# /products/server/products_pub.coffee
Meteor.publish "products", (ops={}) ->
  ...

  if ops.order and not _.isEmpty ops.order
    @relations
```

```
        collection:Orders
        filter:
          _id:ops.order
          status:"new"
        mappings:[
          {
            collection:OrderDetails
            key:"order"
          }
        ]
```

Then we wire up our subscriber with the order key on the client side:

```
# /products/client/products.coffee

Template.created "products", ->
  @autorun =>
    ...

    order = Session.get "global.order"
    if order and not _.isEmpty order
      _.extend filter,
        order:order

    @subscribe "products", filter
```

With this, now we can add products to an order, refresh the site, and still come back to our order. Let's finish building the publisher for the `product` template:

```
# /products/server/product_pub.coffee

Meteor.publish "product", (ops={}) ->
  if ops.product and not _.isEmpty ops.product
    @relations
      collection:Products
      options:
        _id:ops.product
      mappings:[
        {
          key:"product"
          collection:ProductImages
        }
        {
          collection:ProductsTags
          key:"product"
```

```
            mappings: [
              {
                collection:Tags
                foreign_key:"tag"
              }
            ]
          }
        ]

      if ops.order and not _.isEmpty ops.order
        @relations
          collection:Orders
          filter:
            _id:ops.order
            status:"new"
          mappings:[
            {
              collection:OrderDetails
              key:"order"
              filter:
                product:ops.product
            }
          ]

    @ready()
```

Notice that our product is bringing in only the data that is pertinent to that product and for that order. Now we can program our subscriber:

```
# /products/client/product.coffee

Template.created "product", ->
  @autorun =>
    filter = {}

    # Get the product ID from the context
    product = @data._id
    _.extend filter,
      product:product

    # Get the order if any
    order = Session.get "global.order"
    if order and not _.isEmpty order
```

```
        _.extend filter,
          order:order

      @subscribe "product", filter

...
```

File scope variables

These are the types of variables you will want to use most of the time. What are they? They are regular variables! This means that they are not reactive, so watch out. Regular variables are scoped for the file only, so you do not need to worry about interfering with the variables that are named the same in other files.

These variables are useful for things such as lists that you know will not change and that you will not need anywhere else in the application. To use one, simply type out a variable outside your Meteor-specific functions.

The ReactiveVar variables

The ReactiveVar variables are reactive variables that are not available to the console in the way that session variables are. This makes them great for things that are reactive and that we want to keep relatively difficult to access for the user from the console.

If at any point you are creating a highly reactive interface, you should fill it up with these variables so that you do not have to worry about managing them. These variables cleared automatically if there is a hot code reload or a page refresh.

To create a reactive variable, you simply define a variable with its constructor:

```
reactive_variable = new ReactiveVar(<optional-default-value>)
```

This variable will expose setter and getter functions. This means that you use the `.get` command to see the value of the variable and the `.set` command to change it:

```
# Get the value of the variable
reactive_variable.get()

# Set the value of the variable
reactive_variable.set "hello"
```

Let's use the ReactiveVar variables to create a rich number input interface for our quantity field. We will start by defining a route name as `order_quantity` with a `product` parameter:

```
# /orders/cart/cart_route.coffee

FlowRouter.route "/cart",
  ...

FlowRouter.route "/cart/:product/quantity",
  name:"order_quantity"
  action: ->
    FlowLayout.render "layout",
      content:"order_quantity"
```

We can wire up our product with a new button that will take us to the modify quantity view and passes to our product ID:

```
# /products/client/product.coffee

Template.created "product", ->
  ...

Template.product.events
  "click button.add-to-cart": (event) ->
    ...

  "click button.modify-quantity": ->
    FlowRouter.go "order_quantity",
      product:@_id

//- /products/client/product.jade

template(name="product")
  div.col-xs-12.col-sm-4.col-md-3
    article#product
      h5 {{name}}

      div.offer
        span.price {{format.money price}}

      button.add-to-cart.btn.btn-block.btn-primary Add to Cart

      button.modify-quantity.btn.btn-block.btn-info Quantity
```

Now we need to design a responsive number pad. First, let's add a global `vertical-align` class to our `/_globals/client/main.styl` directory so that we can easily align things vertically:

```
// /_globals/client/main.styl

.vertical-align
  position:relative
  transform:translateY(-50%)
  top:50%
```

We will use Jeet to control this design since it requires more specific control of our DOM elements. Let's start by creating the layout of the page in our `order_quantity` template:

```
//- /orders/cart/client/order_quantity.jade

template(name="order_quantity")
  div#order_quantity.template
    section#number
      h1.text-center.vertical-align {{total}}

    section#number-pad
      each numbers
        div.number
          h2.text-center.vertical-align {{number}}

      div.delete
        p.text-center.vertical-align
          i.fa.fa-undo.fa-2x

      div.add-to-cart
        p.text-center.vertical-align
          i.fa.fa-check.fa-2x
```

Notice that we are using the `vertical-align` class on the element that we want to align vertically. We are going to make each of these elements a large button. Also, we will create a `numbers` helper to write out our numbers to the number pad, and the `total` helper will be our reactive variable. Let's have a look at our styles before we create our helpers:

```
// /orders/cart/client/order_quantity.styl

@import "_globals/client/bootstrap/custom.bootstrap.import.styl"
@import "jeet" // Add jeet
```

```
#order_quantity
  overflow:hidden
  background:$brand-primary
  color:white

  section#number
    height:50%
    h1
      margin:0

  section#number-pad
    height:50%
    .number, .delete, .add-to-cart
      cursor:pointer
      height:25%
      col(1/3,gutter:0,cycle:3)
      h2, p
        margin:0
```

In this example, we have passed the parameters to Jeet: `gutter` and `cycle`. Gutter makes sure that there are no margins, while cycle makes sure that the elements can only make rows with a maximum of 3 elements per row. We have made each section take up exactly 50 percent of the height of the screen but ensured that scrolling wouldn't happen by setting `overflow` to hidden. Knowing that we are always going to have four rows of numbers, we set our height for each number to 25 percent.

Now let's put our `ReactiveVar` variable knowledge to work. First, we set up our variable and attach it to the template instance using this:

```
# /orders/cart/client/order_quantity.coffee

# Attach a reactive variable to the instance
# this variable controls our total
Template.created "order_quantity", ->
  @total = new ReactiveVar()
```

Let's build the `numbers` helper and the helper that will render our (`total`) reactive variable:

```
# /orders/cart/client/order_quantity.coffee

...

Template.order_quantity.helpers
  # Create a list of numbers for the number pad
```

```
  "numbers": ->
    _.map [1,2,3,4,5,6,7,8,9,0], (v,k) ->
      number:String v

  # Get the reactive variable
  # this will automatically update when the variable changes
  "total": ->
    Template.instance().total.get()
```

We need to handle the events to update our `ReactiveVar` variable:

```
# /orders/cart/client/order_quantity.coffee

...

Template.order_quantity.events
  # Concatenate numbers to make it work like a number pad
  "click .number": (event,i) ->
    total = i.total.get()

    if total
      new_total = "#{total}#{@number}"
    else
      new_total = "#{@number}"

    i.total.set new_total

  # Remove last number from string
  "click .delete": (event,i) ->
    total = i.total.get()

    if total
      i.total.set total.slice 0,-1

  "click .add-to-cart": (event,i) ->
    # Get the session variable
    order_id = Session.get "global.order"
    order = Orders.findOne order_id

    # Get the total
    total = i.total.get()
    unless total
      return
    else
```

```
  total = Number total

# Get the product with the ID from the router
product = Products.findOne FlowRouter.current().params.product

# Insert Order if it doesn't exist
unless order
  order_id = Orders.insert
    status:"new"
    total_products:0
    subtotal:0
    total:0
else
  order_id = order._id

# Set the session variable for future reference
Session.setPersistent "global.order",order_id

# Find the order
order = Orders.findOne order_id

# Check for details on this product
detail = OrderDetails.findOne
  product:product._id
  order:order._id

if detail
  # Increase by one if the details exist
  OrderDetails.update detail._id,
    $inc:
      quantity:total

  Orders.update order_id,
    $inc:
      total_products:1
      subtotal:product.price * total
      total:product.price * total
else
  # Insert if details do not exist
  OrderDetails.insert
    quantity:total
    product:product
    order:order._id
```

```
        Orders.update order._id,
          $inc:
            total_products:1
            subtotal:product.price * total
            total:product.price * total

      FlowRouter.go "products"
```

If you take a good look at how the `add-to-cart` event works, you will notice that it is almost identical to the event that we wrote for the products template. We are effectively repeating ourselves at this moment, but we will make our code less repetitive through `Meteor.methods` later.

Notice how we are using ReactiveVar the same way as we would use a session variable with the difference that we are not exposing the variable to the console. This helps to make the interface a bit more secure.

Let's not forget to add our publishers for this view to work, we will need `Products`, `Orders`, and `OrderDetails`:

```
# /orders/cart/client/order_quantity.coffee

Template.created "order_quantity", ->
  @total = new ReactiveVar()

  @autorun =>
    @subscribe "order_quantity",
      product:FlowRouter.current().params.product
      order:Session.get "global.order"

# /orders/cart/server/order_quantity_pub.coffee

Meteor.publish "order_quantity", (ops={}) ->
  if ops.product and not _.isEmpty ops.product
    @relations
      collection:Products
      filter:
        _id:ops.product

    @relations
      collection:Orders
      filter:
        _id:ops.order
        status:"new"
      mappings:[
```

```
            {
              key:"order"
              collection:OrderDetails
              filter:
                product:ops.product
            }
          ]

      @ready()
```

We learned how to build a custom form using ReactiveVar variables. The topic covers ReactiveVar and touches on Jeet. We learned how to use reactive variables to create rich UX.

Forms

So far, our buttons and forms have been highly inefficient because they are not easy to repeat, and therefore, not easy to maintain. There are two ways that we can use to make our code easier to manage:

- Meteor Methods
- Autoform

With Meteor Methods, we can easily create **repeatable** and **secure** functions while autoforms can identify the structure of a collection and generate form elements from it. This provides a decent layer of security without much effort as well. Autoform and Meteor Methods can be used together as well for further security.

The autoform method can be implemented through the `aldeed:autoform` and `aldeed:collection2` packages.

Meteor Methods

Meteor Methods can be used on the client or the server. They are created by using the `Meteor.methods(<object-of-functions>)` function and run with the `Meteor.call(<function-name>, <callback function>)` function. Running a meteor method from the client causes 2 versions of the same function to run. One runs on the server and manipulates data while the other runs on the client and simulates data manipulation. This is what we call a Method Stub. Let's look at some example code to understand this concept:

```
# Define your method CLIENT SIDE
Meteor.methods
```

```
  say_hello: ->
    console.log "hello"

# Define your method SERVER SIDE
Meteor.methods
  say_hello: ->
    console.log "I don't want to say hello"

# Run the function CLIENT SIDE
Meteor.call "say_hello"
```

Once you run the function, the Meteor Method activates both the server-side and client-side functions. So the server in this case will log "I don't want to say hello" to the console while the client will log "hello". This is a special feature that Meteor provides to help validate code or run other functions in parallel on the server without exposing sensitive data to the client.

Still, it is important to note that the client-side method will run immediately while the server-side method will take longer. The client-side code should, therefore, be used for simulation while the server-side code should be used for validation.

Let's define and use a Meteor Method that runs on both the client and the server. Let's start by redefining the `add-to-cart` events that we had programmed earlier:

```
# /orders/cart/cart_methods.coffee

Meteor.methods
  "cart.add-to-cart": (ops={},callback) ->
    #ops
      # order
      # product
      # quantity

    order = Orders.findOne ops.order
    product = Products.findOne ops.product

    # Insert Order if it doesn't exist
    unless order
      order_id = Orders.insert
        status:"new"
        total_products:0
        subtotal:0
        total:0
    else
      order_id = order._id
```

```
# Set the session variable for future reference
if Meteor.isClient
  Session.setPersistent "global.order",order_id

# Find the order
order = Orders.findOne order_id

# Check for details on this product
detail = OrderDetails.findOne
  product:product._id
  order:order._id

if detail
  # Increase by one if the details exist
  OrderDetails.update detail._id,
    $inc:
      quantity:ops.quantity

  Orders.update order._id,
    $inc:
      total_products:ops.quantity
      subtotal:product.price * ops.quantity
      total:product.price * ops.quantity
else
  # Insert if details do not exist
  OrderDetails.insert
    quantity:ops.quantity
    product:product._id
    order:order._id

  Orders.update order._id,
    $inc:
      total_products:ops.quantity
      subtotal:product.price * ops.quantity
      total:product.price * ops.quantity

# Run the callback function if it exists
callback and callback(null, true)
```

Notice that we have modified the way that we are setting our persistent session variable. The server does not have a session object, so we need to make sure that this snippet of code only runs on the client using `Meteor.isClient`. We have modified the rest of our code to accept quantity, a product ID, and an order ID, which is all we need to be able to manage both the events.

Also, we added a line at the end of the method to check for a callback function and run it, if it exists. Observe how we have accepted a function as one of our parameters in the beginning of the function. To follow in Meteor's footsteps, we return `null` as the error object and `true` as the result after the function has completed. Now let's replace the events:

```
# /orders/cart/client/order_quantity.coffee

# Attach a reactive variable to the instance
# this variable controls our total
Template.created "order_quantity", ->
  ...

Template.order_quantity.helpers
  ...

Template.order_quantity.events
  ...

  "click .add-to-cart": (event,i) ->
    # Get the total
    total = i.total.get()
    unless total
      return
    else
      total = Number total

    Meteor.call "cart.add-to-cart",
      order:Session.get "global.order"
      product:FlowRouter.current().params.product
      quantity:total
      (error,r) ->
        if not error
          FlowRouter.go "products"

# /products/client/product.coffee

Template.created "product", ->
  ...

Template.product.events
  "click button.add-to-cart": (event) ->
    Meteor.call "cart.add-to-cart",
      order:Session.get "global.order"
```

```
        product:@_id
        quantity:1
        (error,r) ->
          if not error
            FlowRouter.go "products"
```

Meteor Methods are great to create rich and easily repeatable user interfaces, but you will eventually find yourself repeating certain patterns if you are only updating a single collection. This is where autoforms come in.

Autoform

The `autoform` plugin was installed together with the `collection2` plugin. Both of these work together to quickly create forms that support error handling for you. Documentation on these plugins is vast, but it is necessary to read this to make your views safe and easy to build. You can find this documentation at `https://github.com/aldeed/meteor-autoform`. We are going to cover a simple pattern to show you how to easily manipulate autoforms.

To use autoform, you need to first create a `collection2` schema for your collection. This defines the rules that autoform is going to use to validate and generate our input fields. Let's populate our products schema:

```
# /_globals/lib/collections/products/products_collection.coffee

@Products = new Mongo.Collection "products"

Products.attachSchema new SimpleSchema
  name:
    type:String
    label:"Name"

  description:
    type:String
    label:"Description"
    optional:true

  sku:
    type:String
    label:"SKU"
    optional:true

  price:
    type:Number
    label:"Price"
```

In this snippet of code, we are defining every field within a `new SimpleSchema` constructor, then we have attached the schema to the collection using the `.attachSchema` function. The `SimpleSchema` constructor takes an object where the first key defines the name of the field and the object within that key defines the way the field will behave. We will dive into this further in the next chapter.

Attaching a schema to our collections makes sure that the client-side console commands cannot add information to the database that is not a part of the schema. Let's reset our project to make sure our new products adapt to the new schema:

```
meteor reset
```

Now we can create a simple insert form for our products collection using autoforms. Let's create the template and route:

```
//- /products/client/create_product.jade

template(name="create_product")
  h3.text-center create product

  div.container
    div.row
      div.col-xs-12
        +autoForm collection="Products" type="insert" id="insert_
product" preserveForm="true"
          +afQuickField name="name" autocorrect="off"
autocomplete="off"
          +afQuickField name="price"
          +afQuickField name="description"
          +afQuickField name="sku"

          button.btn.btn-block.btn-primary Add Product

# /products/products_route.coffee

FlowRouter.route "/",
  ...

FlowRouter.route "/products/create",
  name:"create_product"
  action: ->
    FlowLayout.render "layout",
      content:"create_product"
```

Notice that we do not need to write a controller to make this work. There are two critical components that we need to understand in this case: `autoForm` and `afQuickField`.

When we use `autoForm` with a collection, we have to declare the name of the collection using the `collection` parameter. Also, we need to define whether the form is doing an insert, update, or method call using the `type` parameter, and finally, we give the form an `id` that will be used as the HTML `id` attribute for the form. Additionally, we pass the `preserveForm` parameter to make sure that the data persists through hot code reloads. There are several other parameters that autoform takes, but these are the most commonly used.

The `afQuickField` component specifically creates a `bootstrap3` input field that can easily handle errors. Notice that we can add HTML attributes (such as `autocorrect` and `autocomplete`) to these components as well as parameters. There are a variety of ways to customize these inputs but we recommend only two: modifying the CSS directly or leveraging `afFieldInput` and `afFieldIsInvalid` to build new elements.

Let's customize our `price` field a little bit more:

```
    div.form-group(class="{{#if afFieldIsInvalid name='price'}} has-
error has-feedback {{/if}}")

      label.control-label {{afFieldLabelText name="price"}}

      +afFieldInput name="price" validation="submit"

      if afFieldIsInvalid name="price"
        span.glyphicon.glyphicon-certificate.form-control-feedback
        span.help-block {{afFieldMessage name="price"}}
```

By doing this, we are very easily adding a `glyphicon` to our field using bootstraps' classes, but this can be useful for adapting our forms to any framework, if needed.

There is still a bit of an issue with this form though, people input prices in dollars instead of cents, but our schema only accepts cents. Let's transform this data before it is submitted using autoform's **hooks**:

```
# /products/client/create_product.coffee

AutoForm.addHooks "insert_product",
  formToDoc: (product) ->
    product.price = product.price * 100

    product
```

```
    docToForm: (product) ->
      product.price = product.price / 100

      product
```

Autoform has a long list of hooks, but by far the most useful one for transformations is `formToDoc`. This function runs every time the form is converted to a document for processing, this includes before the errors are processed. We do not use the "before hook" because this hook will run after processing errors. Also, we are using a `docToForm` hook to make sure that our `price` field stays intact after a code reload. Autoform has the following hooks:

```
before:
  insert:
  update:
  update-pushArray:
  method:
  method-update:
  normal:
after:
  insert:
  update:
  update-pushArray:
  method:
  method-update:
  normal:
onSubmit:
onSuccess:
onError:
formToDoc:
formToModifier:
docToForm:
beginSubmit:
endSubmit:
```

With this, we can quickly create forms and modify them to the way we like while preserving usability. Stay away from fields that require arrays! They are difficult to manage in our application and can always be expressed as a new collection in our models.

Loading data

The performance of our frontend can suffer if we do not load data correctly because it can make the DOM redraw quickly multiple times. If the calculations are complex, then we need to wait for all our data to be available and calculate everything in one single sweep. If we do not wait for data, then calculations will run as the data is received, which in turn will cause the DOM to redraw multiple times very quickly.

To solve this issue, we can easily check whether our subscribers are ready and show a loading symbol until they are.

Designing the loading indicator

We start by creating a loading template. Let's use font awesome to handle our animations and make sure that we can easily change the color of the loader:

```
//- /loader/loader.jade

template(name="loader")
  div#loader.template
    div(class="{{color_class}}").vertical-align.text-center
      i.fa.fa-5x.fa-cog.fa-spin

// /loader/loader.styl

#loader
  height:100%
```

In this case, we are using the `fa-spin` class to make the cog spin and adding a `color_class` helper that will take bootstraps' `text-` classes to define colors. Also, we made sure that the loader fills the contents of whatever is holding it so that everything aligns nicely when we use it.

Implementing the loading indicator

Let's implement the loading indicator for our list of products. Due to the way we are going to implement this, we could easily have our interface show the loading indicator for each product or the list of products. It will be more efficient to control the latter.

So let's modify our products template:

```
//- /products/client/products.jade

template(name="products")
```

```
div#products.template
  header#promoter
    ...

  section#features
    ...

  section#featured_products
    div.container
      div.row
        if Template.subscriptionsReady
          each products
            +product
        else
          div(style="height:160px;")
            +loader color_class="text-primary"

  br
```

Meteor core exposes a special `Template.subscriptionsReady` helper that checks whether the subscriptions made in that template instance are all in the ready state. This is one of the hidden advantages of using `@subscribe` from our `Template.created` function instead of `Meteor.subscribe`.

In this case, we only run the `each` iteration on the `products` variable until all subscriptions are ready. Before this happens, we render our `loader` component in our primary color.

The advantage to this pattern is that we can control exactly where we want our loader to be and when we want it to appear.

Animations and transitions

Animation for the web is a vast field that is still growing and can get highly complex if you want it to be. At the end of the day though, efficient animations are what make a real difference. How do you make an animation efficient? You keep it simple, and you let the browser animate it for you. Stay away from using JavaScript to identify positions and change colors. Animations that are heavy on JavaScript will always have terrible performance. This poor performance happens because JavaScript cannot run effectively in all devices, so multiple changes to the DOM occur at the maximum allowed speed of the device without using video memory.

Canvas is an HTML element that is designed to render changes using JavaScript and WebGL. Animating anything through canvas will boost performance, but it is meant more for applications that want to do graphically intensive work such as 3D rendering or sprite animations for a game. We will not cover canvas in this book because you will probably never need it.

Ignoring the power of canvas leaves us with two CSS tools to make rich animations: `animation` and `transition`. The `animation` property is meant for complex animations that require keyframes, while the `transition` property is meant for simple change-of-state animations. Let's look at an example of each to first understand how they work, then we will apply them to Meteor.

Animating with CSS

The `transition` property is the easiest property to understand. You basically define which property you are going to animate and how, within a CSS property:

```
.box
  transition: <CSS property> <duration> <timing function> <delay>
```

It's important to realize that because of **Stylus Nib** (another useful Stylus plugin included in this package), we do not have to worry about vendor prefixes for this particular property. Nib will handle this for us automatically. By adding the transition property to the `box` class, we are telling the browser to animate any change that occurs on the defined property. How do these changes occur? With new classes. Let's look at a small example:

```
.box
  transition: opacity 300ms ease-in
  opacity: 0
  .in
    opacity: 1
```

With this set of CSS rules, we are making the box invisible until the `.in` class is dynamically added. Once the class is added, then the DOM element will fade in. The available timing functions are: `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `step-start`, and `step-end`.

The `animation` property is a bit more complex because it does not require a dynamic class to activate the animation:

```
.box
  animation:
    <animation name>
    <duration>
```

```
      <timing function>
      <delay>
      <direction>
      <iteration count>
      <fill mode>
      <play state>

  @keyframes <animation name>
    0%
      background:blue
    50%
      background:green
    100%
      background:red
```

As you can see, the difference between a transition and an animation is the level of control you can have over the animation. By using an `animation name` and the `@keyframes` selector, we can control at which point of the animation we want our properties to change and to what. You can change the number on the keyframe to whatever number you want.

Notice that the `animation` property has a few new parameters: `iteration count`, `fill mode`, and `play state`. The `iteration count` parameter defines how many times an animation is going to play; you can set this to infinite or a specific number. The `fill mode` defines the state at which you want the element to be when it is not animating: `forwards` sets values to the last defined keyframe, `backwards` to the first defined keyframe, both sets to `first` and `last`, and `none` will do nothing (default). The `play state` controls whether an animation is playing or not (paused or running).

We can use a second CSS selector to control our `play state`, if we need to in the same way that we did for our `transition` property.

Executing animations in Meteor

In Meteor, it is very easy to control the classes that are rendered on the DOM using helpers, so we can quickly inject a class that will trigger our CSS animation. We will have issues when we want to trigger animations on elements that do not exist in the DOM.

For example, once our products have loaded, they just pop in. What if we want to make the list fade in and the loader fade out instead? This is where Meteor's hidden `@_uihooks` function comes into play.

At the time of writing, this feature is in beta and is still a bit buggy. The best way to handle all the issues that this is causing is through the `percolate:momentum` package. This avoids the need to understand the issues that `uihooks` currently has and how to get around them.

The `momentum` package is designed to intercept Blaze by using the `@_uihooks` function that is made available in the context of the `Template.rendered` function. It is very easy to use and comes packaged with VelocityJS, a jQuery plugin that handles transitions somewhat effectively. We can use a combination of both to help keep things simple.

To customize momentum, we need to register momentum plugins that handle the three hooks that can occur in our DOM: `insertElement`, `removeElement`, and `moveElement`.

`insertElement`	Occurs when a DOM element is created
`removeElement`	Occurs when a DOM element is destroyed
`moveElement`	Occurs when a DOM element is moved from it's DOM location (sorting can cause this to be triggered)

Let's build the plugin named `fade-fast` that will control the fading in and out of our loading indicator of our products:

```
# /_globals/client/momentum/fade-fast.coffee

Momentum.registerPlugin 'fade-fast', (options) ->
  insertElement: (node, next) ->
    $(node)
      .addClass "animate opacity invisible"
      .insertBefore(next)

    Meteor.setTimeout ->
        $(node).removeClass "invisible"
      ,250

  removeElement: (node) ->
    $(node).velocity opacity:0, 250, "easeOut", ->
      $(this).remove()
```

We use the `node` variable to modify the elements that we are going to inject into the DOM, and we always use `insertBefore` with the `next` variable as a parameter to render. Notice that in this case, when we insert, we are adding three classes before the items are rendered to the DOM, and then after 250 milliseconds, we remove the invisible class.

When an element is removed, we will use `velocity` to modify `opacity` and bring this down to `0` in exactly `250` milliseconds. Velocity takes in a callback command that we will use to finally remove the element from the DOM.

Now we can add our styles and place the helper inside our view:

```
//- /products/client/products.jade

  ...

    section#featured_products
      div.container
        div.row
          +momentum(plugin="fade-fast")
            if Template.subscriptionsReady
              each products
                div.col-xs-12.col-sm-4.col-md-3
                  +product
            else
              div(style="height:160px;")
                +loader color_class="text-primary"

// _globals/client/main.styl

.vertical-align
  ...

.animate
  &.opacity
    transition: opacity 500ms
    opacity:1

    &.invisible
      opacity:0
```

Notice that we have added a `momentum(plugin="fade-fast")` component that defines which plugin to use around the area that is going to render elements with animations. Then we create the classes that will use CSS transition to make each DOM element fade in.

SEO

Meteor has one big problem. It does not support server-side rendering out of the box. This means that robots don't know how to parse our pages because the server does not render the page, the client does! There are many ways to solve this issue. You can try using `meteorhacks:ssr` to enable server-side rendering or you can let a service automate this for you for free. We are going to use a service to keep things simple.

Prerender.io

Meet prerender.io, the easiest way to get your site parsed. Prerender.io is mostly a free service that understands exactly how to parse your page. It basically navigates to your `webapp` and parses each page, and then when a bot hits our `webapp`, our server will fetch the parsed page from prerender.io and respond with this instead.

The service is free for the webapps that have a minimal number of pages that get hit by bots. The more dynamic and the more pages you have, the higher the price will be, but this is unlikely unless you are building a highly dynamic public web page. Even then, the `prerender.io` project can be downloaded and set up personally (so it's still free).

We have installed the `dfischer:prerenderio` package to easily set up the service. The only thing that is left is for us is to authorize. First, go to `www.prerender.io` and open an account, then click the **INSTALL TOKEN** tab on their navbar, and copy your token. Now paste your token under the configuration file:

```
# /_globals/server/prerenderio.coffee

prerenderio.set "prerenderToken","<yourtoken>"
```

Simple! Your site is now crawl-able. Now let's configure our router to work with prerender and add a 404 page:

```
# /router/config.coffee

if Meteor.isClient
  Template.created ->
    except = [
      "Template.__dynamicWithDataContext"
      "Template.__dynamic"
      "Template.layout"
      "Template.layout"
      "body"
    ]
```

```
    unless _.contains except, @view.name
      window.prerenderReady = false

      if @subscriptionsReady()
        window.prerenderReady = true

FlowRouter.notFound =
  action: ->
    FlowLayout.render "layout",
      content:"not_found"
```

Notice that we are creating a global `Template.created` hook function and setting `prerenderReady` to `false` by default. Then, we check whether our subscriptions are ready using `@subscriptionsReady()` and set `prerenderReady` to `true`. It's important to understand that `@subscriptionsReady()` only checks for subscriptions made through `@subscribe` within the template!

Also, notice that we are creating a list of exceptions. This is to ensure that we only check for subscribers on templates that need them.

This pattern ensures prerender does not cache the page until it has actually loaded all the data. Why not use `FlowRouter` global triggers? Sadly, these triggers do not have access to the template instance that is being rendered (since there can be many). If you need more control, you can set this per route/template.

Let's work on our 404 page template:

```
//- /router/client/not_found.jade

template(name="not_found")
  div#not_found.template
    div.text-center.vertical-align
      h3 404!
      h5 Page not found!
```

To make sure our 404 page prints out the correct 404 response, we are going to have to use `Meta`.

Using Meta

To control prerender, we need to use the `yasinuslu:blaze-meta` package. This package basically injects a template to our `head` HTML tag so that we can easily set tags. Prerender uses tags to identify whether the page should respond with a 404. We can use Meta to set our title, description, keywords, and robots properties as well. Let's start by making our 404 page respond with a 404:

```
# /router/client/not_found.coffee

Template.rendered "not_found", ->
  Meta.set [
    {
      name:"name"
      property:"prerender-status-code"
      content:"404"
    }
    {
      name:"name"
      property:"robots"
      content:"noindex, nofollow"
    }
  ]
```

The `Meta` object has only four functions: `set`, `setTitle`, `unset`, and `config`. It is unlikely that you will use `unset` because metatags do not persist through routes. Both the `set` and `unset` functions require an array of objects, and each object requires all the three keys: `name`, `property`, and `content`.

In the preceding example, we are setting `prerender-status-code` to `404` and `robots` to `noindex, nofollow`. With this, we ensure that our 404 page in fact returns a 404 error and that crawlers do not index the page.

Also, we need to configure `Meta` so that the name of the page is always correct:

```
# /router/config.coffee

if Meteor.isClient
  Meta.config
    options:
      title:"Crashing Meteor"
      suffix:""

  ...
```

Since `Meta` is controlling our `title` tag, we need to remove that tag from `layout.jade`.

Schema.org

Now that our site is crawl-able, we want our products to shine for search engines. Let's upgrade our products to enable Rich Snippets. Rich Snippets follow schema.org standards to build structured data for search engine crawlers such as Google to interpret. With this information, search engines can present your data on their site more prominently, make searches more effective, and even use it in other services. Following their guidelines is a good idea because Google uses these guidelines for its crawler.

To do this, all we need to do is set as many schema.org's properties as possible on our product page. Visit `http://schema.org/Product` for a full list of the available properties. The properties that are most used are:

Property	Type	Description
`name` (required)	`Text`	The name of the product
`image`	`URL`	The URL of the product
`description`	`Text`	The description of the product
`offers`	`OFFER OBJECT`	This includes multiple parameters that define the price of the object
`price` (required)	`Number`	The price of the product
`priceCurrency` (required)	`Text (ISO 4217) [ex: USD]`	The currency of the price

Now let's put this to practice using the `MicroData` spec:

```
//- /products/client/product.jade

template(name="product")
  article#product(itemscope itemtype="http://schema.org/Product")
    h5(itemprop="name") {{name}}

    div.offer(itemprop="offers" itemscope itemtype="http://schema.org/
Offer")
      meta(itemprop="priceCurrency" content="USD")
      span.price(itemprop="price") {{format.money price}}

    button.add-to-cart.btn.btn-block.btn-primary Add 1 to Cart

    button.modify-quantity.btn.btn-block.btn-info Add more to Cart
```

As you can see, to use the `MicroData` spec, you need to add a variety of custom attributes to your view. For the most part, anytime you are going to declare what a particular thing is, you need to express this via the `itemscope` and `itemtype` attributes. When we want to declare properties under this declaration, we need to make sure our DOM elements are children of this declaration and that the property is declared through the `itemprop` attribute.

Summary

This chapter covered several useful patterns that make the implementation of rich and minimal interfaces easy. First, we learned how to use the Twitter Bootstrap framework together with Jeet and Rupture; this improved the way we organize our DOM. Also, we learned how to make super helpers—functions that we can call in both views and controllers.

We took a dive into the different kinds of variables that we can use to keep our code safe and to create rich interfaces. Also, we learned how to use Meteor Methods and autoform to easily save information. We covered how to implement a simple loading indicator and how to get animations working in Meteor. To finish up, we learned how to implement a little SEO using prerender.io and the `Meta` package.

The next chapter will cover application-wide patterns. *Chapter 4*, *Application Patterns* will teach us how to filter and page through collections, how to fully secure our application, and how to integrate with external APIs.

4
Application Patterns

This chapter will cover application-wide patterns that share server- and client- side code. With these patterns, your code will become more secure and easier to manage. You will learn the following topics:

- Filtering and paging collections
- Security
- External API

Filtering and paging collections

So far, we have been publishing collections without thinking much about how many documents we are pushing to the client. The more documents we publish, the longer it will take the web page to load. To solve this issue, we are going to learn how to show only a set number of documents and allow the user to navigate through the documents in the collection by either filtering or paging through them.

Filters and pagination are easy to build with Meteor's reactivity.

Router gotchas

Routers will always have two types of parameters that they can accept: query parameters, and normal parameters. Query parameters are the objects that you will commonly see in site URLs followed by a question mark (`<url-path>?page=1`), while normal parameters are the type that you define within the route URL (`<url>/<normal-parameter>/named_route/<normal-parameter-2>`). It is a common practice to set query parameters on things such as pagination to keep your routes from creating URL conflicts.

A URL conflict happens when two routes look the same but have different parameters. A products route such as `/products/:page` collides with a product detail route such as `/products/:product-id`. While both the routes are differently expressed because of the differences in their normal parameter, you arrive at both the routes using the same URL. This means that the only way the router can tell them apart is by routing to them programmatically. So the user would have to know that the `FlowRouter.go()` command has to be run in the console to reach either one of the products pages instead of simply using the URL.

This is why we are going to use query parameters to keep our filtering and pagination stateful.

Stateful pagination

Stateful pagination is simply giving the user the option to copy and paste the URL to a different client and see the exact same section of the collection. This is important to make the site easy to share.

In *Chapter 2*, *Publish and Subscribe Patterns* we had created a products publisher and subscriber to elaborate on our publishers. Now we are going to understand how to control our subscription reactively so that the user can navigate through the entire collection.

First, we need to set up our router to accept a page number. Then we will take this number and use it on our subscriber to pull in the data that we need. To set up the router, we will use a `FlowRouter` query parameter (the parameter that places a question mark next to the URL).

Let's set up our query parameter:

```
# /products/client/products.coffee

Template.created "products", ->
  @autorun =>
    tags = Session.get "products.tags"
    filter =
      page: Number(FlowRouter.getQueryParam("page")) or 0

    if tags and not _.isEmpty tags
      _.extend filter,
        tags:tags

    order = Session.get "global.order"
    if order and not _.isEmpty order
```

```
        _.extend filter,
          order:order

      @subscribe "products", filter

  Template.products.helpers
    ...

    pages:
      current: ->
        FlowRouter.getQueryParam("page") or 0


  Template.products.events
    "click .next-page": ->
      FlowRouter.setQueryParams
        page: Number(FlowRouter.getQueryParam("page")) + 1

    "click .previous-page": ->
      if Number(FlowRouter.getQueryParam("page")) - 1 < 0
        page = 0
      else
        page = Number(FlowRouter.getQueryParam("page")) - 1

      FlowRouter.setQueryParams
        page: page
```

What we are doing here is straightforward. First, we extend the filter object with a `page` key that gets the current value of the page query parameter, and if this value does not exist, then it is set to `0`. `getQueryParam` is a reactive data source, the `autorun` function will resubscribe when the value changes. Then we will create a helper for our view so that we can see what page we are on and the two events that set the page query parameter.

But wait. How do we know when the limit to pagination has been reached? This is where the `tmeasday:publish-counts` package is very useful. It uses a publisher's special function to count exactly how many documents are being published.

Let's set up our publisher:

```
  # /products/server/products_pub.coffee

  Meteor.publish "products", (ops={}) ->
    limit = 10
    product_options =
```

```
      skip:ops.page * limit
      limit:limit
      sort:
        name:1

    if ops.tags and not _.isEmpty ops.tags
      @relations
        collection:Tags
        ...
            collection:ProductsTags
            ...
                collection:Products
                foreign_key:"product"
                options:product_options
                mappings:[
                  ...
                ]

    else
      Counts.publish this,"products",
        Products.find()
        noReady:true

      @relations
        collection:Products
        options:product_options
        mappings:[
          ...
        ]

    if ops.order and not _.isEmpty ops.order
      ...

    @ready()
```

To publish our counts, we used the `Counts.publish` function. This function takes in a few parameters:

```
Counts.publish <always this>,<name of count>, <collection to count>,
<parameters>
```

Note that we used the `noReady` parameter to prevent the `ready` function from running prematurely. By doing this, we generate a counter that can be accessed on the client side by running `Counts.get "products"`. Now you might be thinking, why not use `Products.find().count()` instead? In this particular scenario, this would be an excellent idea, but you absolutely have to use the `Counts` function to make the count reactive, so if any dependencies change, they will be accounted for.

Let's modify our view and helpers to reflect our counter:

```
# /products/client/products.coffee

...

Template.products.helpers
  pages:
    current: ->
      FlowRouter.getQueryParam("page") or 0

    is_last_page: ->
      current_page = Number(FlowRouter.getQueryParam("page")) or 0

      max_allowed = 10 + current_page * 10
      max_products = Counts.get "products"

      max_allowed > max_products

//- /products/client/products.jade

template(name="products")
  div#products.template
    ...
      section#featured_products
        div.container
          div.row
            br.visible-xs
            //- PAGINATION
            div.col-xs-4
              button.btn.btn-block.btn-primary.previous-page
                i.fa.fa-chevron-left

            div.col-xs-4
              button.btn.btn-block.btn-info {{pages.current}}
```

```
div.col-xs-4
  unless pages.is_last_page
    button.btn.btn-block.btn-primary.next-page
      i.fa.fa-chevron-right

div.clearfix
br

//- PRODUCTS
+momentum(plugin="fade-fast")
              ...
```

Great! Users can now copy and paste the URL to obtain the same results they had before. This is exactly what we need to make sure our customers can share links. If we had kept our page variable confined to a `Session` or a `ReactiveVar`, it would have been impossible to share the state of the webapp.

Filtering

Filtering and searching, too, are critical aspects of any web app. Filtering works similar to pagination; the publisher takes additional variables that control the filter. We want to make sure that this is stateful, so we need to integrate this into our routes, and we need to program our publishers to react to this. Also, the filter needs to be compatible with the pager. Let's start by modifying the publisher:

```
# /products/server/products_pub.coffee

Meteor.publish "products", (ops={}) ->
  limit = 10
  product_options =
    skip:ops.page * limit
    limit:limit
    sort:
      name:1

  filter = {}

  if ops.search and not _.isEmpty ops.search
    _.extend filter,
      name:
        $regex: ops.search
        $options:"i"
```

```
if ops.tags and not _.isEmpty ops.tags
  @relations
    collection:Tags
    mappings:[
        ...
        collection:ProductsTags
        mappings:[
            collection:Products
            filter:filter
            ...
    ]

else
  Counts.publish this,"products",
    Products.find filter
    noReady:true

  @relations
    collection:Products
    filter:filter
    ...

if ops.order and not _.isEmpty ops.order
  ...

@ready()
```

To build any filter, we have to make sure that the property that creates the filter exists and `_.extend` our `filter` object based on this. This makes our code easier to maintain. Notice that we can easily add the filter to every section that includes the `Products` collection. With this, we have ensured that the filter is always used even if tags have filtered the data. By adding the filter to the `Counts.publish` function, we have ensured that the publisher is compatible with pagination as well.

Let's build our controller:

```
# /products/client/products.coffee

Template.created "products", ->
  @autorun =>
    ops =
      page: Number(FlowRouter.getQueryParam("page")) or 0
      search: FlowRouter.getQueryParam "search"
    ...
```

```
    @subscribe "products", ops

Template.products.helpers
  ...

  pages:
    search: ->
      FlowRouter.getQueryParam "search"

    ...

Template.products.events
  ...

  "change .search": (event) ->
    search = $(event.currentTarget).val()

    if _.isEmpty search
      search = null

    FlowRouter.setQueryParams
      search:search
      page:null
```

First, we have renamed our `filter` object to `ops` to keep things consistent between the publisher and subscriber. Then we have attached a `search` key to the `ops` object that takes the value of the search query parameter. Notice that we can pass an undefined value for `search`, and our subscriber will not fail, since the publisher already checks whether the value exists or not and extends filters based on this. It is always better to verify variables on the server side to ensure that the client doesn't accidentally break things. Also, we need to make sure that we know the value of that parameter so that we can create a new `search` helper under the `pages` helper. Finally, we have built an event for the search bar. Notice that we are setting query parameters to `null` whenever they do not apply. This makes sure that they do not appear in our URL if we do not need them.

To finish, we need to create the search bar:

```
//- /products/client/products.jade

template(name="products")
  div#products.template
    header#promoter
      ...
```

```
    div#content
      section#features
        ...

      section#featured_products
        div.container
          div.row
            //- SEARCH
            div.col-xs-12
              div.form-group.has-feedback
                input.input-lg.search.form-control(type="text"
placeholder="Search products" autocapitalize="off"
autocorrect="off" autocomplete="off" value="{{pages.search}}")
                span(style="pointer-events:auto;
cursor:pointer;").form-control-feedback.fa.fa-search.fa-2x

            ...
```

Notice that our search input is somewhat cluttered with special attributes. All these attributes ensure that our input is not doing the things that we do not want it to for iOS Safari. It is important to keep up with nonstandard attributes such as these to ensure that the site is mobile-friendly. You can find an updated list of these attributes here at `https://developer.apple.com/library/safari/documentation/AppleApplications/Reference/SafariHTMLRef/Articles/Attributes.html`.

Security

Many packages secure certain parts of the Meteor stack, but even then, you cannot rely on these packages completely. Also, you have to be very careful about the packages you choose! Some packages might intercept core functions to funnel information out of your application. This means that you should always have a look at the source code of the package before you install it.

This topic is usually an oversight to novice Meteor developers, and yet it is one of the most important topics to know about. To secure our webapp we need to:

- Define roles (set distinctions between users)
- Define schemas for every collection (limit how they can modify fields)
- Define deny rules (limit who can modify fields)
- Use methods, when necessary, to check parameters (ensure complex security when needed)
- Set browser policies

Roles

Using roles, just about every web application creates a distinction between users and what they are allowed to do. To help us manage roles easily, we have installed the `alanning:roles` package. With this package, we are going to control who hits our routes and who may modify our collections.

This package makes the `Roles.userIsInRole` function available, which uses a `roles` collection to make sure that the user is in the correct role:

```
Roles.userIsInRole <user-id OR user-object>, [<list of allowed
roles>], <group>
```

Suppose you need to check whether a user is an `admin` or a `manager` before they access a certain function. For this, you will simply do the following:

```
if Roles.userIsInRole Meteor.userId(), ["admin","manager"]
  # allow
```

Let's add an `admin` role to our application. We can start by creating an initialization file that will automatically build our admin user:

```
# /_globals/server/initial_setup.coffee

Meteor.startup ->
  # Users
  if Meteor.users.find().count() is 0
    user = Accounts.createUser
      email:"you@email.com"
      password:"1234"

    Roles.addUsersToRoles user,["admin"]
```

Notice that we are using the `Roles.addUsersToRoles` function to set the new user's role to `admin`, and this takes place on the server side. Always set user roles on the server side. Next, let's build a login route that can be accessed only if you are not already logged in:

```
# /login/login_route.coffee

FlowRouter.route "/login",
  name:"login"
  triggersEnter:[RT.non_user_only]
  action: ->
    BlazeLayout.render "layout",
      content:"login"
```

To make sure roles run at the right moment when our user first hits the site, we need to make sure that `FlowRouter` activates after roles have been loaded. To do this we use `FlowRouter.initialize()` and `FlowRouter.wait()`.

```
# /_globals/router/config.coffee

if Meteor.isClient
  BlazeLayout.setRoot 'body'

  FlowRouter.wait()
  Meteor.startup ->
    # Initialize roles before FlowRouter
    Tracker.autorun (computation) ->
      if Roles.subscription.ready() and not FlowRouter._initialized
        FlowRouter.initialize()
        computation.stop()
```

The `FlowRouter.route` function accepts a `triggersEnter` parameter and a `triggersExit` parameter. This can be used to redirect the user based on roles. Both these parameters are arrays of functions so that many triggers can be added to each route. To make it easier for us, we are going to create a dictionary of triggers under a global `RT` object. Notice that we do not execute the function within the triggers array, so we do not include parentheses.

Let's start by moving the `/router` folder to the `/_globals` folder. This will make sure that the `RT` object is the first thing that is defined. After doing this, we should define two triggers:

```
# /_globals/router/triggers.coffee

@RT =
  non_user_only: (context,redirect) ->
    if Meteor.userId()
      if context and context.oldRoute
        redirect context.oldRoute.path
      else
        redirect "/"

  admin_only: (context,redirect) ->
    if not Roles.userIsInRole Meteor.userId(),["admin"]
      if context and context.oldRoute
        redirect context.oldRoute.path
      else
        redirect "/"
```

Notice that when `FlowRouter` calls these functions, it will include a `context` object and a `redirect` function. The `context` object has information about the route that we are trying to connect to and our previous route, while the `redirect` function is used to redirect a user. In this case, we have attempted to redirect to the previous route if it exists, if it does not exist, then we redirect to root.

Now, let's add the `admin_only` trigger to every route except `products` and `login`:

```
# /_globals/router/triggers.coffee

@RT =
  non_user_only: (context,redirect) ->
    ...

  admin_only: (context,redirect) ->
    ...

FlowRouter.triggers.enter [RT.admin_only],
except:["products","login","cart","order_quantity"]
```

We can easily create a global trigger that does not apply to the `products` nor the `login` routes using the `FlowRouter.triggers.<enter or exit>` function. We don't have to worry about including our `404` route since by default; it does not run triggers.

To finish up, let's build a custom login page:

```
//- /login/client/login.jade

template(name="login")
  div#login.template
    div.vertical-align.container
      div.row
        div.col-xs-12.col-sm-6.col-sm-offset-3
          form.login
            div.form-group
              label Email
              input.email.input-lg.form-control.text-
center(type="text" placeholder="email" value="{{email}}"
autocapitalize="off" autocorrect="off" autocomplete="off")

            div.form-group
              label Password
              input.password.input-lg.form-control.text-
center(type="password" placeholder="password")
```

```
            if error
              div.row
                div.col-xs-12
                  div.alert.alert-warning {{error}}

            button.login.btn.btn-block.btn-primary.btn-lg Log In

# /login/client/login.coffee

Template.created "login", ->
  @error = new ReactiveVar false

Template.login.events
  "submit .login": (event,i) ->
    event.preventDefault()
    email = $(".email").val()
    pw = $(".password").val()

    # Check Email
    if email and not _.isEmpty email.trim()
      email = email.replace /\s/g,""
      email = email.trim().toLowerCase()
    else
      i.error.set "Email is invalid"
      return

    # Check Password
    if not pw or _.isEmpty pw
      i.error.set "Password is invalid"
      return

    i.error.set false

    Meteor.loginWithPassword email, pw, (error) ->
      if not error
        i.error.set false
        $("input").val ""
        FlowRouter.go "dashboard"
      else
        i.error.set error.reason

Template.login.helpers
  "error": ->
    Template.instance().error.get()
```

Here, we have used Meteor's core `Meteor.loginWithPassword` function to log in, and we have kept a track of the errors using a `ReactiveVar` variable.

Collection2

Suppose a malicious user visits the site and quickly identifies one of our collections in the browser console. They are going to update one of the products by calling the `Products.update` function. Because our site is insecure, they can successfully call something like this:

```
# Malicious User
Products.update("productid",{$set:{you:"have been modified"}})
```

This will successfully create a field that should not exist for that particular product!

With the `aldeed:collection2` package, we are going to secure our collections by white-listing fields. This makes sure that the allowed users can only set acceptable values on our collections and that these values meet certain criteria.

A `collection` field can take the following parameters:

Parameter	Use
`type`	This defines the type of the value. This can be any JavaScript primitive: `String`, `Number`, `Boolean`, `Date`, `Object`, or an array of primitives such as `[Object]`, `[String]`.
`decimal`	This is only available if `type:Number`. This defines whether a number is a decimal. This can be `true` or `false`.
`optional`	This defines whether the field is required on insert. This can be `true` or `false`.
`regEx`	This checks whether the string matches the defined `regEx` expression. This can be any `regEx` expression such as `/^[A-Z]$/`.
`allowedValues`	This checks whether the string matches any of the values in an array. This can be only an array of strings such as `["Green","Blue"]`.
`blackbox`	This allows any combination of values and objects to be placed as values. This can be only `true` or `false`.
`denyUpdate`	This defines whether the field can be updated. This can be only `true` or `false`.
`denyInsert`	This defines whether the field can be inserted. If this field is set to `true`, `optional:true` must be set as well. This can be only `true` or `false`.

Parameter	Use
`autoValue`	This defines the value that the field will take during an operation. This can be a `function` only.
`custom`	This defines the custom function that will validate the field. This can be a `function` only.
`unique`	This defines whether the value of the field should be unique. This can be only `true` or `false`.

With these parameters, it is easy to lock down every collection including the `Meteor.users` collection and increase the security of our app drastically.

As we will save all our numerical data in hundreds, we will never use the `decimal` parameter. You should always avoid decimals regardless of whether you are dealing with money or not.

Both the `autoValue` and `custom` parameters expose variables and functions within their context that are critical to understand:

Context variables and functions	Use
`this.isInsert`	Boolean. This checks whether the field is being inserted.
`this.isUpdate`	Boolean. This checks whether the field is being updated.
`this.upsert`	Boolean. Check whether the field is being upserted.
`this.userId`	String. This checks the current `userId`. Returns `undefined`, if it does not exist and is `null` for all server-initiated functions.
`this.isFromTrustedCode`	Boolean. This checks whether the field is being modified by the server-side code.
`this.isSet`	Boolean. This checks whether the field is being modified.
`this.value`	Anything. If `this.isSet`, then this will be the value of the field.
`this.operator`	String. If `this.isSet` and `this.isUpdate`, then this will be the operator that modifies the value (`$pull`, `$push`, `$addToSet`, `$set`, and so on).
`this.field("<field-name>")`	This is a function that returns an object. This gets the object representation of a field that is being modified. From this object, you can use `isSet`, `value`, and `operator` to get more information. For example, `this.field("name").value` will return the value of the field name, if it is set.

With these functions, we can add custom validators and custom automatic values depending on the state of the modification. Also, we can check what the values of other fields are and react to them if we need to. If your validation requires complex queries with other collections, DO NOT RELY ON THIS PACKAGE!. This tool is strictly to control collection-specific values and not relationships. Remember that not all our collections are available on the client side, and therefore, are incapable of being validated properly. We will address these kinds of validations in the next topic.

Let's secure all our collections. We will only show the `Orders` and `OrderDetails` collections schemas:

```
# /_globals/lib/collections/orders/orders_collection.coffee

@Orders = new Mongo.Collection "orders"

Orders.attachSchema new SimpleSchema
  status:
    type:String
    allowedValues:["new","pending","complete"]

  total_products:
    type:Number

  subtotal:
    type:Number

  tax_total:
    type:Number
    optional:true

  total:
    type:Number

  date_created:
    type:Number
    autoValue: ->
      if @isInsert
        return Date.now()
      if @isUpsert
        $setOnInsert:Date.now()

@OrderDetails = new Mongo.Collection "order_details"
```

```
OrderDetails.attachSchema new SimpleSchema
  order:
    type:String

  product:
    type:String

  price:
    type:Number

  quantity:
    type:Number

  subtotal:
    type:Number

  tax:
    type:Object
    optional:true

  "tax.rate":
    type:Number

  "tax.amount":
    type:Number

  total:
    type:Number
```

As you can see, the pattern for white-listing a field within our collection is a simple object that contains the `Collection2` parameters. Notice that we can define rules for subobjects using MongoDB's dot notation. This means that we can set rules for arrays and object arrays in the same fashion as well:

```
people:
  type:[Object]

"people.$.name":
  type:String

"people.$.age":
  type:Number
```

Still, you do not have to set complex rules often since they are clear indicators that a new collection should be created.

Notice that we are not using any complex custom validation in our schema besides `name` and `type`. Why? Should we check whether the user is an admin, or whether the modification originated from the server-side code, or whether the order belongs to the customer?

While we could start adding a few lists of checks for each field, the true source of the problem stems from allowing users to modify our collection directly from the console. To fully secure our orders, we need to modify the allow/deny rules and use trusted code.

It is important to understand that these schemas are used by server-side code too. This makes sure that neither the client nor the server can mess up our keys.

Deny rules

Now that we know the fields that will be in our collections, we need to make sure that the server allows the right people to modify these collections. To do this, we first need to understand exactly how the allow/deny rules work.

Meteor has two core functions that control whether a collection modification is allowed or not: `Meteor.allow` and `Meteor.deny`.

The `Meteor.allow` functions allow a collection to be modified as soon as one of the rules resolves to `true`. Also, this means that the other allow rules are not evaluated! Knowing this, some developers cram logic into a single allow rule that could easily fail. This is bad practice because the code will be difficult to maintain.

On the other hand, the `Meteor.deny` functions will always run and trump a `Meteor.allow` rule that resolves to `true`. To manage our deny rules effectively, we are going to use the `ongoworks:security` package. With this package, we can easily build reusable and readable rules to be set on our collections.

Let's begin by removing all the allow rules from our project. Then we can set some rules for our `Orders` collection:

/_globals/lib/collections/orders/server/orders_permissions.coffee

```
Meteor.startup ->
  # Admin may only modify status
  Orders.permit "update"
    .ifLoggedIn()
```

```
  .ifHasRole "admin"
  .onlyProps "status"
  .apply()
```

With this simple rule, we have denied anything that does not match this rule. This package integrates directly with our `roles` package, so we can easily use the `ifHasRole` function. This rule makes sure that only an admin user can update an order's `status` field via the console. To make sure that the rule is applied, we have used the `apply` function.

There are three things that we need to understand about the deny rules: logic, integrated functions, and custom functions.

Logic in rules decides whether they act as ANDs or ORs. If we have multiple functions inside a rule, then we define AND rules. The function that we have defined for our orders is an AND rule because this checks `ifLoggedIn` AND `ifHasRole` AND `onlyProp` before allowing a modification to pass. If we want to create an OR rule, we simply create a new rule. Let's try this:

```
Orders.permit "update"
  .ifLoggedIn()
  .ifHasRole "admin"
  .onlyProps "status"
  .apply()

Orders.permit ["insert","remove"]
  .never()
  .apply()
```

Here, we are stating that a user may `update` `ifLoggedIn` AND `ifHasRole` AND `onlyProp` OR never `insert/remove`.

The package comes with a handful of integrated functions to help in applying the rules:

Function	Use
`never()`	This prevents DB operations
`ifLoggedIn()`	This allows DB operations if logged in
`ifHasUserId(<user-id>)`	This allows DB operations if user ID is a particular string
`ifHasRole(<role string>)` `ifHasRole({role:<role string>,group:<group>})`	This allows DB operations if user belongs to a particular role

Function	Use
`onlyProps(<string or array of strings>)`	This allows DB operations on certain top-level fields only (this will not identify arrays and subobjects)
`exceptProps(<string or array of strings>)`	This allows DB operations on all top-level fields except these (this will not identify arrays and subobjects)

Custom deny rules

While the `security` package functions are useful, you may find yourself needing custom functions for more precise handling of your security. To build a custom function, you need to use the `Security.defineMethod` function:

```
Security.defineMethod <function name>,
  transform:<function>
  deny: <function (type, args, userid, doc, fields, modifier)>
```

This function accepts two parameters: `transform` and `deny`. The `transform` function allows the modification of fields before they enter the `deny` function, while the `deny` function is an extended version of `Meteor.deny`. The `deny` function passes several arguments that include information about the document being modified and the user. These arguments are: `type`, `arguments`, `userId`, `document`, `fields`, and `modifier`. The last two arguments (`fields` and `modifier`) are only passed if `type` is equal to `update`.

Let's add a custom function:

```
# /_globals/server/security.coffee

Security.defineMethod "ifUserIsOwner",
  deny: (type,args,user,doc) ->
    user isnt (doc.user or doc._id)
```

Here, we are defining an `ifUserIsOwner` function that checks whether the ID of the currently logged in user is equal to the `user` field or the `_id` field on the modified document.

Notice that the logic of the rule denies a database operation if the user is not the owner of the document.

Now we can use this rule to secure our `users` collection:

```
# /_globals/server/security.coffee

Security.defineMethod "ifUserIsOwner",
  ...

Security.permit(["update"]).collections([Meteor.users])
  .ifUserIsOwner()
  .onlyProps ["emails"]
  .apply()

Security.permit(["insert","update","remove"]).collections([Meteor.
users])
  .ifHasRole "admin"
  .apply()
```

Notice that we are attaching rules to the `Meteor.users` collection in a different way. We do this because the `Meteor.users` collection is a special collection that is initialized in a different way than the rest of our collections, with this we have ensured that the rules are properly attached.

In this example, we are allowing our users freely to modify the `email` field from the console, and only the admin user can modify all users from the console.

However, now that we have locked the code from the client, how are we supposed to make things work? Running code directly from an event on the client will fail because the code is untrusted. We need to build a trusted code that will handle changes to the database. The solution is simple: Meteor methods.

The Meteor methods – round 2

We have already covered how `Meteor.methods` work, but we have not discussed the difference between trusted and untrusted code.

Trusted code can modify multiple documents at once by setting `multi` to `true` and can use an arbitrary Mongo selector to find the documents to modify. It bypasses any access control rules set up by allow and deny. Trusted code includes all server code and `Meteor.methods`.

Untrusted code can modify only a single document at once that is specified by its `_id`. The modification is allowed only after checking any applicable allow and deny rules. Untrusted code cannot perform upserts. Untrusted code includes client code such as event handlers and the console.

This means that whenever we directly modify a collection on the client, we are running untrusted code that is limited by our deny rules. Knowing this, it becomes obvious that most code, especially complex code with relationships, should run on a Meteor method.

But wait. Can a user modify the code of the method on the client side? They most certainly can, but remember that the code runs in a stub. The stub makes sure that the correct code runs on the server, while the client-side code updates the collections temporarily until the server responds. This is what Meteor calls **Optimistic UI**.

So, if someone tampers with the client-side version of our Meteor method, the server-side version will still run normally and the UI will update correctly.

It's important to understand though that `Meteor.methods` will still be bound to the rules we had set on `Collection2`, which is a great feature inside a team environment where not everyone understands the structure of all data models.

So where should we use untrusted code? The answer depends greatly on your application, but for the most part, you will want everything to run on trusted code because it is easier to maintain and safer. Untrusted code should be used mostly to control things that the database is not connected to, or information that is allowed to be just about anything that the user wants.

In order to use `Meteor.methods` properly, we need to run validation on `Meteor.methods` as well. We do this using Meteor's core `check` package. Let's upgrade our `cart.add-to-cart` method:

```
# /orders/cart/cart_methods.coffee

Meteor.methods
  "cart.add-to-cart": (ops={}) ->
    # Validate data
    check ops,
      order:Match.Optional(Match.OneOf(String,null))
      product:String
      quantity:Number

    ...

    # Insert Order if it doesn't exist
    unless order
      ...
    else
      # Validate order status
```

```
    if order.status isnt "new"
      throw new Meteor.Error 405, "Not Allowed"

    order_id = order._id

  ...
```

First, we have used the `check` function to validate the structure of the `ops` object and to ensure that each key inside the object matches the correct type of data primitive. When the validation fails, the function will automatically stop and return a **404 Match Failed** error to the client.

Next, we have checked the status of the order. If the order is not new, then we throw a `Meteor.Error`. This will short circuit the function as well and return the `error` object to the client. When you are settings errors inside the Meteor methods, you will always use `throw new Meteor.Error(<error message>)` to communicate the error.

We do not need to worry about doing anything else with our errors. We will see how to keep track of our application errors in a different topic.

The `check` function takes only two variables:

```
check <value>, <pattern>
```

The `value` parameter takes the variable that will be analyzed, while the `pattern` parameter takes the validator. The pattern can be as simple as a JavaScript primitive such as `String`, `Number`, and `Boolean`, an array of validators, an object of validators, or as complex as a function. The pattern simply needs to return `true` for the validation to pass.

Meteor includes a handful of useful `pattern` functions through the `Match` object:

Pattern function	Use
`Match.Any()`	This allows any value to pass validation.
`Match.Integer()`	This allows any 32-bit integer. `Infinity` and `NaN` are not allowed.
`Match.ObjectIncluding(<object>:<pattern>)`	This allows an object to include key/value pairs that are not defined in the object. The example that we used does not allow other key/value pairs from entering the method.
`Match.Optional(<second-pattern>)`	This allows a value to be `undefined`. If the value is defined, then the `second-pattern` will be evaluated instead.

Pattern function	Use
`Match.OneOf(<pattern1>, <pattern2>,...)`	This allows a value to pass if it matches any of the defined patterns.
`Match.Where(<function(value){}>)`	This runs `function` and passes the defined value as the first parameter. If the function returns `true`, then validation passes.

Now this is all grand and makes us feel much more secure, but the fact of the matter is that we are sharing our server-side logic on the client. This means that we absolutely cannot include sensitive data inside a Meteor method. If we do want to pass sensitive data, it is best to store it in a server-side variable and call it, or if you want to go an extra mile, you can separate the client-side method from the server using folders.

Managing the wait time

The first chapter gave us a brief explanation of the impact that blocking can have on our client. To summarize this, if a function is waiting for a third-party or doing something that takes long, you should unblock the function. However, what impact can this have?

Remember that methods are placed on a conveyor belt. When we unblock a function, we place the method in a different conveyor belt where we cannot put other methods. This means that if another method depends on the unblocked method to complete, there can be a serious issue because the method can run before, during, or after the unblocked method.

Notice that our `cart.add-to-cart` method does not have an unblock function. This is to make sure that the server adds items to the cart in the same order as the client. So what do we do if we have something inside the method that can be placed in a separate conveyor belt? Can there be something that can happen in parallel and the user does not have to wait to continue?

`Meteor.defer(<function>)` is a special, undocumented function that can take a particular snippet of code and run it in a separate conveyor without blocking the function from which it was called. Suppose we want to notify the admin every time a new order has been created, as shown here:

```
# /orders/cart/cart_methods.coffee

Meteor.methods
  "cart.add-to-cart": (ops={}) ->
```

```
# Validate data
...

# Insert Order if it doesn't exist
unless order
  ...
  if Meteor.isServer
    Meteor.defer ->
      Email.send
        to:"you@email.com"
        from:"me@email.com"
        subject:"New Customer!"
        text:"Someone has created a new order"

...
```

In this example, we have wrapped the `Email.send` function in a `Meteor.defer` function to make the e-mail run in parallel. This benefits the performance of your code by running deferred functions in parallel and focusing on producing only the results that matter to the user. By doing this, we ensure that the e-mail, a process that we know takes a long time to finish, does not block the server as well.

Browser policy

Now that we can secure our collections and functions, we need to protect our application as a whole. We can achieve this protection using the `browser-policy` package. Let's install it now:

```
meteor add browser-policy
```

So what exactly does this package do? By adding this, we are opening an access to a series of configuration options that will help us to set our application headers and content security policies to protect from cross-site scripting and data injection attacks.

These kinds of attacks are often used to steal your data (data theft), change the way your site looks (website defacement), and distribute malware. We definitely want to avoid all these attacks.

So how does this work? By adding the package, we have already protected our application against many attacks by default, but we need to be able to control this. To do this, the package exposes two objects each with a particular set of functions: `BrowserPolicy.framing` and `BrowserPolicy.content`. Both these functions must be set in the server.

Framing

With `BrowserPolicy.framing`, we can control whether our web application can be rendered inside an iframe or not. We have three functions to control this:

Function	Use
`*.framing.disallow()`	This will never render in an iframe regardless of the origin.
`*.framing.restrictToOrigin(origin)`	This will only render in iframes created by the specified origin. This may take only one `origin` string and may be called only once. Also, it is not fully supported in WebKit.
`*framing.allowAll()`	This will render in any iframe.

You will find yourself using `BrowserPolicy.framing.disallow()` most of the time, but it is important to understand that you will still be able to iframe your app if origins are the same.

Content

With `BrowserPolicy.content`, we can control exactly how the content will be loaded into our web application. We have many functions for this. However, we will actually end up using only a few. The functions are:

Function	Use
`*.content.allowInlineScripts()`	This allows DOM script tags to run. `DEFAULT`.
`*.content.disallowInlineScripts()`	This does not allow DOM script tags to run.
`*.content.allowEval()`	This allows the building of JavaScript from strings by using the `eval` function.
`*.content.disallowEval()`	This does not allow the building of JavaScript from strings by using the `eval` function. `DEFAULT`.
`*.content.allowInlineStyles()`	This allows inline styles and style DOM elements to run. `DEFAULT`.
`*.content.disallowInlineStyles()`	This does not allow inline styles and style DOM elements to run.

There are more! The next set of functions defines a white-list of content types and how they are allowed to load. `ContentType` can take the value of: `Script`, `Object`, `Image`, `Media`, `Font`, `Frame`, and `Connect`.

Function	Use

`*.allow<ContentType>Origin(origin)`
`*.allowScriptOrigin(origin)`
`*.allowObjectOrigin(origin)`
`*.allowImageOrigin(origin)`
`*.allowMediaOrigin(origin)`
`*.allowFontOrigin(origin)`
`*.allowFrameOrigin(origin)`
`*.allowConnectOrigin(origin)`
	This allows the `ContentType` to be loaded from the `origin` string. This function can be called multiple times and has a support for wildcards. If a protocol (`http` / `https`) is not specified, then BOTH are allowed.

`*.allow<ContentType>DataUrl()`	This allows the `ContentType` to be loaded from a `data:` URL. This will allow base64-encoded images to render.
`*.allow<ContentType>SameOrigin()`	This allows the `ContentType` to be loaded from the same origin as the webapp.
`*.disallow<ContentType>()`	This does not allow `ContentType` to be loaded.
`*.allowSameOriginForAll()`	This allows all types of content to be loaded from the same origin as the webapp.
`*.allowDataUrlForAll()`	This allows all types of content to be loaded from a `data:` URL.
`*.allowOriginForAll(origin)`	This allows all types of content to be loaded from the specified `origin`.
`*.disallowAll()`	This does not allow any type of content to be loaded from anywhere.

Depending on what we want our application to achieve, we may have to tweak our rules. Let's configure our security policy to the recommended options:

```
# /_globals/server/security.coffee

Meteor.startup ->
  # Prevent webapp from loading on an iframe
  BrowserPolicy.framing.disallow()
```

```
  # Prevent inline scripting
  BrowserPolicy.content.disallowInlineScripts()

  trusted_sites = [
    '*.google-analytics.com'
    '*.mxpnl.com'
    'placehold.it'
    'placeholdit.imgix.net'
  ]

  _.each trusted_sites, (trusted_site) ->
    BrowserPolicy.content.allowOriginForAll
"https://#{trusted_site}"
```

External APIs

Now that we know how to secure our application, we need to understand how to keep external data sources up to date. There are two patterns that we can use to ensure that the information on our server is recent: **synchronization** and **webhooks**.

Synchronization

Synchronization will, basically, get data from our source continuously and refresh the database. This type of technique is useful when we need to save information from our data source and use that information to produce analytical data via the aggregation framework.

To keep our servers synchronized, we need to make sure that the process that fetches information does not block the server. We can ensure that this happens using nonblocking functions like `Meteor.setInterval`.

Let's synchronize with Stripe. First, we will need to create a collection to capture payments, then we will have to set permissions and our Stripe secret, and finally, we will build the HTTP `GET` function:

```
# /_globals/lib/collections/stripe/payments_collection.coffee

@Payments = new Mongo.Collection "payments"

# /_globals/lib/collections/stripe/server/payments_permissions.coffee

Meteor.startup ->
  # Nobody may modify
  Payments.permit ["insert","remove","update"]
```

```
    .never()
    .apply()

# /_globals/server/stripe.coffee

@Stripe =
  secret:"secret-key"
  publishable:"publishable-key"

# /stripe/server/payments.coffee

_.extend Stripe,
  get_payments: (ops={}) ->
    params =
      limit:100

    if ops.starting_after_id
      _.extend params,
        starting_after:ops.starting_after_id

    HTTP.get "https://api.stripe.com/v1/charges",
      headers:
        "Authorization":"Bearer #{Stripe.secret}"
      params:params
      (error,result) ->
        if not error
          _.each result.data?.data, (charge) ->
            Payments.upsert _id:charge.id,
              $set:charge

          if result.data.has_more
            last = _.last result.data.data
            Stripe.get_payments
              starting_after_id:last.id

        else
          throw new Meteor.Error error

Meteor.setInterval Stripe.get_payments,3600000
```

Notice that when we created our collection, we did not give it a schema. We did this because we wanted to make sure that our collection is flexible in case the data from our endpoint changes. To secure our collection, we need to make sure that nobody can modify this in any way except for trusted code. Then we have created a configuration object that is going to hold our Stripe secret and public key.

Also, we have created a `/stripe/server/payments.coffee` directory. Here, we have added the `get_payments` function to the `Stripe` object that we had defined in our `_globals` directory. To have this work properly, we had to pass a `params` object to control the way we are requesting data from Stripe's servers. Expect to pass `params` for every `GET` request you build since this controls the endpoints' `pagination`. Stripe explains in their documentation that we can get the next page of data by first checking whether there is more data via the `has_more` key and then passing in the last object ID from which to start via the `starting_after` parameter.

At the end of this all, we have used the `Meteor.setInterval(<function>,<delay in milliseconds>)` function to make sure that the function runs every hour. Are we done yet? Not quite. While this code will definitely populate our `Payments` collection, it could potentially crash our server as well. Why? The request always queries from the beginning of time up to today, which makes the server progressively take longer. Another important issue is that if there is one `GET` request being processed and it has not finished within the hour, another request could start in parallel and consume even more resources.

We can prevent this by controlling our intervals and limiting the time frame our interval uses to get the data.

First, let's make sure that we only have one interval running at a time:

```
# /stripe/server/payments.coffee

_.extend Stripe,
  payments:
    get: (ops={}) ->
      if not Stripe.payments.is_running
        Stripe.payments.is_running = true

        params =
          limit:100

        if ops.starting_after_id
          _.extend params,
            starting_after:ops.starting_after_id
```

```
        HTTP.get "https://api.stripe.com/v1/charges",
          headers:
            "Authorization":"Bearer #{Stripe.secret}"
          params:params
          (error,result) ->
            if not error
              _.each result.data?.data, (charge) ->
                Payments.upsert _id:charge.id,
                  $set:charge

              if result.data.has_more
                last = _.last result.data.data
                Stripe.payments.get
                  starting_after_id:last.id
              else
                Stripe.payments.is_running = false

            else
              Stripe.payments.is_running = false
              throw new Meteor.Error error

    set_interval: ->
      Meteor.setInterval Stripe.payments.get,360000

    is_running:false

Stripe.payments.set_interval()
```

Notice that we are keeping payments together under a `payments` object now. Then we have simply set and checked a `is_running` Boolean key to see whether the process is running or not. Now if we were to reduce the interval to 1 millisecond, it would only GET from the external API after the previous request is done processing.

Now, we can use the `starting_after` parameter to ensure that we only get the latest information. To do this, we will have to use the `moment` function to filter data by time and get the latest payment information:

```
# /stripe/server/payments.coffee

_.extend Stripe,
  payments:
    get: (ops={}) ->
      if not Stripe.payments.is_running
        Stripe.payments.is_running = true
```

```
          params =
            limit:100

          if ops.starting_after_id
            _.extend params,
              starting_after:ops.starting_after_id
          else
            date_after =
moment().utc().startOf("day").subtract(10,"days").unix()
            latest_payment = Payments.findOne
created:$gte:date_after,
              sort:
                created:1

            if latest_payment
              _.extend params
                starting_after:latest_payment.id
          ...
```

In this example, we have simply used `moment` to identify a Unix timestamp from 10 days ago. The `moment` object is made available using the `momentjs:moment` package. Notice that we are using the `utc()` function to set `startOf("day")` consistently between our development and production environments. Then we have queried the server, and if the payment exists, we will use the ID of the payment as our `starting_after` parameter.

It is important to understand that we can make this query easily only because the information that Stripe sends includes a Unix timestamp. Not every API has a Unix timestamp. Most likely, you will end up inserting or transforming their data to fit your own. This can be easily addressed by extending their response when needed.

Webhooks

Webhooks are the way through which other servers can communicate directly with ours. They, basically, make a `POST` request to one of our endpoints to inform our server that something has happened.

In the case of Stripe, we are going to add an endpoint that will catch all of Stripe's charge webhooks. To do this, we will use the `nimble:restivus` package.

`Restivus` is an excellent package that makes it easy to maintain a RESTful API with version control and user authentication. It runs only on the server side for security.

First, we need to create an instance of `Restivus`. This instance will hold the routes for the first version of our endpoint:

```
# /_globals/server/stripe.coffee

@Stripe =
  secret:"secret"
  publishable:"public"
  hooks:
    v1:new Restivus
      apiPath:"stripe"
      version:"v1"
```

Here, we have attached our new server-side endpoint to the global `Stripe` object under the `hooks` and `v1` keys. This will make it easy to create new routes for version 1 anywhere in our server. The `Restivus` instance takes in a handful of parameters, the ones that you will end up using the most are:

Parameter	Use
`apiPath`	String defines the parent route of all the endpoints. If we define `apiPath` as `"stripe"` and a route as `"charge"`, then the path for the route will be `ROOT_URL/stripe/charge`.
`version`	String defines the version number of the API and adds it to the parent route. If we define `version` as `"v1"`, then all the routes will take this form: `ROOT_URL/<apiPath>/v1/<route>`.
`enableCors`	Boolean sets whether the route is accessible from external domains. Default: `true`.

Now let's create an endpoint that updates the `Payments` collection. To create an endpoint, we need to define a route on our `Restivus` instance. To do this, we simply call the global object that we had defined and use the `addRoute` function:

```
# /stripe/server/endpoints/charges.coffee

Meteor.startup ->
  Stripe.hooks.v1.addRoute "charge",
    post: ->
      payment = @request.body.data?.object
      if payment
        Payments.upsert _id:payment.id,
          $set:payment

      @done()
```

Now our endpoint can catch Stripe's webhook notifications. Notice that the information that the `POST` request contains is located in the `@request.body` object. After analyzing this object, we can see that the object that Stripe is sending. In this case, Stripe sends an object with a `data` key, which in turn contains an `object` key that holds the information of the payment. If you want to see the information that Stripe is responding with, simply log in to the console.

Also, we are returning a `@done()` function, which notifies Stripe that the request was processed. This makes sure that Stripe does not have to make a second attempt to notify our server.

The `addRoute` function can handle all types of HTTP requests including `get`, `put`, `delete`, `patch`, and `options`. The endpoints have these variables in their context to help handle requests:

Context variable	**Use**
`this.user`	The `Meteor.user` object after authentication passes.
`this.userId`	The `Meteor.userId` string after authentication passes.
`this.urlParams`	Parameters defined in the URL string: `ROOT_URL/stripe/v1/charge/:more` To access the `more` parameter, you would call `@urlParams.more`.
`this.queryParams`	Query parameters defined in the URL string: `ROOT_URL/stripe/v1/charge?more=data` To access the `more` parameter, you would call `@urlQueryParams.more`.
`this.bodyParams`	The body of the request. This is the equivalent to `@request.body`.
`this.request`	A NodeJS request object.
`this.response`	A NodeJS response object.
`this.done()`	This function must be called after handling a response.

Now, we can set up webhooks from Stripe by going to their settings page and creating a webhook. Make sure to point their URL to the one that we built.

We can send a couple of test hooks, too, to make sure that everything works as intended. By using both the synchronization pattern and the webhooks pattern, we are able to keep our data up to date. You should always use both patterns if the API allows you to. Why? This is because their webhooks server could potentially fail.

Summary

This chapter covered three important things: how to control the amount of data that we publish, how to secure our app, and how to better integrate with an external API. We learned a pattern to build pagination that functions with filters as well. Then we learned how to build user roles and schemas to secure access to our application better. Next, we understood the limitations of allow/deny rules and addressed these limitations by writing effective deny rules. We quickly realized these deny rules were blocking functions from all the event handlers because they are insecure. To get around this limitation, we learned how to build trusted code. Towards the end, integrating Stripe taught us how to create a nonblocking synchronization function and how to use `restivus` to catch incoming messages from external servers.

The next chapter is going to cover the basics of how to test and maintain our code. With the next chapter, we will be able to share our code without fearing that someone else will break it.

5
Testing Patterns

This chapter will cover testing patterns that ensure our code is easy to maintain. With these patterns, you will learn how to implement regression testing—a method to identify if the new code breaks the old code before the code goes into production. Building tests is crucial to maintain code and collaborate with others. You will learn the following topics:

- Behavior tests
- Unit tests

Testing in Meteor is still under heavy development, but the functions that we will cover are basic and are unlikely to change.

Behavior tests

Behavior tests are known as end-to-end tests too. The purpose of a behavior test is simple: it makes sure that a feature of the project is working. A feature refers to the business logic behind the application. For example, a feature of our current project is viewing a list of products in our landing page. Another feature is the ability to add a variable quantity of products to the order.

To run behavior tests in Meteor, we essentially need to build a robot that can visit our website and try to make these features work. While this sounds complex, the Meteor Velocity project simplifies much of this when paired with Cucumber.

Velocity is a project that sets the stage for other testing frameworks to use. It works by creating mirrors of the project where tests can run.

Cucumber is a specification-by-example testing framework. It is meant to describe application features in plain English before they are programmed. In a collaboration setting, these tests are the ones that matter most because they make sure that the application is behaving as expected. We will use the Cucumber package now because it is one of the only testing frameworks that support behavior tests that are currently available to Meteor.

To build tests, we first need to install the `xolvio:cucumber` package:

```
meteor add xolvio:cucumber
```

This package automatically installs Velocity and all other required packages. Now run the Meteor command to start up the server. The following two things are going to happen:

- A new browser window will open if there are any tests to run
- The current project will have a dot on the top-right side

The new browser window is what we call a **mirror** and is the client that is going to run all the behavior tests that we define. While this is great, it is annoying to have a second browser window open and close every time we run the Meteor command.

To get rid of the second browser window, we can use `phantomjs` to run tests. To do this, we will simply run the Meteor command with some settings:

```
SELENIUM_BROWSER=phantomjs meteor
```

PhantomJS is a headless browser. A headless browser is a web browser (such as Safari and Chrome) without a **Graphical User Interface** (**GUI**). In other words, it's a browser for robots, and this is exactly what our robot needs to run our tests.

Let's make a custom Meteor alias with this command. Open the `~/.bash_profile` directory with your favorite text editor. If you use Sublime Text 3, you can run the following command:

```
sublime ~/.bash_profile
```

In this file, add this line to the end of the document:

```
# ~/.bash_profile

alias devmeteor='SELENIUM_BROWSER=phantomjs meteor'
```

Now completely quit your terminal, and then open it again. Go to your project and run the `devmeteor` command to start your project.

Go to the project URL. Here, you will notice a blue circle on the top-right corner of the project. Click this to reveal the velocity test dashboard. Whenever tests fail, here you will see why they failed. All tests are rerun every time there is a change in the code.

Let's start by building our first behavior test. In this test, we are going to check whether we can add items to the order. To do this, we need to program two key components:

- Features
- Steps

Steps are, basically, snippets of code in JavaScript or CoffeeScript that interpret the **feature**. Step snippets are automatically generated after writing a feature. Therefore, we need to start by writing the description of the feature.

All our tests are going to be saved under the `/tests` directory. Behavior tests will always go under the `/cucumber` folder. At the time of this writing book, this has been mandatory because Velocity runs under these special directories only:

```
# /tests/cucumber/features/cart/add to cart.feature

Feature: Add to Cart
```

```
As a customer
I want to add items to my cart
So I can checkout

Background:
  Given I am an anonymous customer

@dev
Scenario: Add one to cart
  When I navigate to "/"
  And I click on the first button with class ".add-to-cart"
  Then I should have one product with quantity 1 in my cart
```

Notice that the file ends in `.feature` and is located under the `/features` directory. Placing files under the `features` directory is mandatory as well for the testing suite to be able to identify your feature files.

This file is written in the Gherkin language. If you want to add comments, you can use the pound sign (#) just we do in CoffeeScript. The `Feature` keyword describes the feature; this can be anything as long as it helps you to identify what the feature is. After this, we can see three lines that describe the purpose of the feature. This description can also be anything you want since this does not affect the test, but it is common to use the following syntax to help identify whether the feature is useful or not:

```
As a [role]
I want [feature]
So that [benefit]
```

Next, we will find the `Background` and `Scenario` keywords. Both are a list of actions that lead to a result. For each feature, there can be multiple scenarios that test the different parts of the feature. The `Background` keyword defines a list of actions to be taken before running the ones under the `Scenario` keyword. Try to keep backgrounds short and simple, and keep in mind that they will run for every scenario.

Also, notice that we have an `@dev` keyword directly above the `Scenario` keyword; this keyword controls where `Scenario` is going to run. If you do not include a keyword, it will only run when you run `meteor --test` in your terminal. If you include `@dev`, the test will run every time you make changes to your web application. Also, you can include `@ignore` to ignore the test completely.

The `Given`, `When`, `And`, and `Then` keywords are commands that run the steps in our test code:

- `Given`: The purpose of `Given` is to put the app in a known state before any interaction occurs.
- `When`: The purpose of `When` is to describe the key action the user performs.
- `And`: The purpose of `And` is to write scenarios more fluently. They, basically, substitute `And` with the previous keyword.
- `Then`: The purpose of `Then` is to observe and evaluate outcomes. This phrase will always make sure that the system has produced something.

Now that we have programmed our first feature, let's generate the steps that will execute the feature. Run your alias command:

```
devmeteor
```

Once the server has started, Velocity will produce a command you can use to tail the projects' `cucumber.log`. Open up a separate terminal without deactivating Meteor and run that command. It should look something like this:

```
tail -f /Users/YOU/pathtoyourproject/online_shop/.meteor/local/log/
cucumber.log
```

Now you will see something like this:

```
@dev
  Scenario: Add one to cart                                   # features/cart/Add to cart.feature:13
    Given I am an anonymous customer                          # features/cart/Add to cart.feature:10
    When I navigate to "/"                                    # features/cart/Add to cart.feature:14
    And I click on the first button with class ".add-to-cart" # features/cart/Add to cart.feature:15
    Then I should have one product with quantity one in my cart # features/cart/Add to cart.feature:16

1 scenario (1 undefined)
4 steps (4 undefined)

You can implement step definitions for undefined steps with these snippets:

this.Given(/^I am an anonymous customer$/, function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.When(/^I navigate to "([^"]*)"$/, function (arg1, callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.When(/^I click on the first button with class "([^"]*)"$/, function (arg1, callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.Then(/^I should have one product with quantity one in my cart$/, function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});
```

If you don't, you can restart the Meteor project without leaving `cucumber.log`. Here, we can see the JavaScript version of the steps that the feature is expecting. Let's rewrite these in CoffeeScript under the special `/step_definitions` directory. This directory must always be a sibling to the related `.feature` files.

Copy the code snippets and create a new file under the `/tests/cucumber/features/cart/step_definitions/steps.coffee` directory. We are going to convert these snippets to CoffeeScript, since this is what we have been working with all along. You can do this quickly at `http://js2.coffee`.

Notice that each function passes a `callback` variable at the end and calls the `pending()` function. When tests run, this indicates that the function has not been built yet and appears as pending in the Cucumber log. We can remove the `callback` variable after we have finished working with the function. Notice that arguments are passed as well.

Running tests with this code will not work; we need to initialize the test properly. To do this, we begin by using CoffeeScripts' `do` function and attaching the commands to the `module.exports` function. You need to do this for every test file:

```
# /tests/cucumber/features/cart/step_definitions/steps.coffee

do ->
  'use strict'

  module.exports = ->
    @Given ...
    @When ...
    @Then ...
```

Now that the tests are actually running, we can dive deeper. The first function that we will address is `Given`. `Given` needs to make sure that the user is completely new and has no active order. To do this, we are going to use **fixtures**. Fixtures are `Meteor.methods` exclusively available in testing. We can use these methods to clear our database or add seed data. Let's start by making a fixture that ensures that the user is anonymous and another fixture that clears all the orders:

```
# /tests/cucumber/fixtures/cart_fixtures.coffee

do ->
  'use strict'

  Meteor.methods
    "anonymous_user": ->
```

```
        if @userId
          @setUserId null

      "reset_orders": ->
        Orders.remove {}
        OrderDetails.remove {}
```

Now, we can call these methods in the `Given` function (or any other function) using the `@server.call` function:

```
# /tests/cucumber/features/cart/step_definitions/steps.coffee

do ->
  'use strict'

  module.exports = ->
    @Given /^I am an anonymous customer$/, ->
      @server.call "reset_orders"
      @server.call "anonymous_user"
```

Notice that we are using `@server`, but we can use `@client` too since this is a call to `Meteor.method`. Now let's program `When`:

```
# /tests/cucumber/features/cart/step_definitions/steps.coffee

do ->
  'use strict'
  ...

  module.exports = ->
    url = require('url')
    @Given /^I am an anonymous customer$/, ->
      ...

    @When /^I navigate to "([^"]*)"$/, (path) ->
      @browser
        .url url.resolve(process.env.ROOT_URL, path)

    @When /^I click on the first button with class "([^"]*)"$/,
    (button) ->
      @browser
        .waitForExist "body *"
        .waitForVisible ".product"
        .element ".product:nth-child(1) #{button}"
        .click()
```

The `@browser` object gives access to an instance of webdriver. This means that we can use the webdriver functions to simulate user clicks, inspect elements, and browse the Web. Notice that we can require NPM modules through the `require` function as well. In this case, we will require the `url` module to help identify routes.

The first `When` function requires a single argument defined by the regular expression. In this case, it is the `path` variable. Arguments will always be listed one by one as they are defined in the regular expression, and they will end with the callback function (which we do not need to use): `(arg1, arg2, arg3, callback) ->`.

The second `When` function needs to wait for the DOM to load before acting on it. To do this, we will use the `waitForExist` and `waitForVisible` functions. If the elements do not render, then the test will fail due to a timeout in this function. Remember that if you ever want to see what the test is doing, you can browse to your mirror.

You can find a full list of the available webdriver functions at `http://webdriver.io/api.html`, but here is a list of the ones you will use the most:

Function	**Use**
`waitForExist(selector[,timeout,reverse])`	Default timeout: `500`, reverse: `false`. This waits for an element to render on the DOM. Setting the `reverse` flag to `true` will instead wait for the element to stop existing.
`waitForVisible(selector[,timeout,reverse])`	Default timeout: `500`, reverse: `false`. This waits for an element to be visible (checks that the display CSS property is not set to any, that the element is not outside the viewport, and that the `opacity` CSS property is not set to `0`). Setting the `reverse` flag to `true` will instead wait for the element to be invisible.

`click(selector)`
`doubleClick(selector)`
`leftClick(selector)`
`rightClick(selector)`
	This clicks an element. Can take in a CSS selector.

`setValue(selector,values)` `addValue(selector,values)`	This sends a sequence of keystrokes to an element. Can use unicode characters as well to simulate things such as backspace and arrow keys. The `addValue` function will append to a set value.

Function	Use
`getText(selector)` `getValue(selector)`	This gets the node text or input value.
`getCssProperty(selector,property)` `getAttribute(selector,attribute)`	This gets data on a CSS property or DOM element attribute. The `property` variable will return an object instead of a string.
`then(function(valueFromGet))`	This uses data obtained from any `get` function. The first parameter is always the value of the `get` function.

Now let's build our `Then` function and use **Chai** to evaluate that everything proceeded as expected:

```
# /tests/cucumber/features/cart/step_definitions/steps.coffee

do ->
  'use strict'

  module.exports = ->
    ...

    @Then /^I should have one product with quantity one in my
    cart$/, ->
      @browser
        .waitForVisible ".order_detail"
        .getText ".order_detail:nth-child(1) .quantity"
        .then (quantity) ->
          expect quantity
            .to.equal "1"
```

Here, we have used the `then` function to work with the value from the `.quantity` node. We use Chai to check whether the value obtained is correct. The list of Chai functions is long, and you will find yourself using most of them. They are easy to guess though! You can find all of these functions here: `http://chaijs.com/api/bdd`.

Unit tests

Unit tests are much easier to build than behavior tests. These tests make sure that only a section of the web application is working correctly, such as `Meteor.method` or a template helper.

Unit tests make it quicker to find bugs in broken behavior tests. They should be used mostly in parts that you know can break easily, such as a publisher or a particular helper.

To run unit tests, we are going to use the `sanjo:jasmine` package:

```
meteor add sanjo:jasmine
```

Now create two directories: `/jasmine/client/integration` and `/jasmine/server/integration`. These are the special directories where `jasmine` runs tests. Let's build a quick test for the `products` publisher:

```
# /tests/jasmine/client/integration/publishers/products_pub_test.
coffee

do ->
  'use strict'

  describe "Products publisher", ->
    it "should return product data", ->
      # SETUP
      subscription = Meteor.subscribe("products")

      if subscription.ready()
        # EXECUTE
        product = Products.findOne()

        # VERIFY
        expect product
          .not.toBeUndefined()
```

Jasmine is simple. First, you use the `describe` function to describe the object of the feature, and then you use the `it` function to explain each part of the feature that should work. It is common to divide the evaluating function into three blocks: `SETUP`, `EXECUTE` and `VERIFY`. At setup, we make sure that everything is ready for the test to run, then we execute a series of functions, and finally, we use the Chai expressions to verify whether the test passes or fails.

Unit tests are great for testing peculiarities in your code that users might not see reflected visually. Have a look at the Jasmine documentation at `http://jasmine.github.io/2.3/introduction.html` for a collection of great examples.

Summary

In this chapter, we learned how to build simple tests for our web application. In addition, we learned that they are a crucial part of the development process in order to maintain the application in both a team and nonteam setting. Behavior tests are the tests that make sure that the application is running all the features as expected, while unit tests are the tests that make sure that only particular weak spots are running as expected. Be careful with testing. While it is important to keep tests active for maintenance, it is more important to focus on the programming of the product. If you do not have time to write a full behavior test, write at least one unit test on a function that is critical for the web application to work.

In the next chapter, we are going to cover how to deploy our web application to a production quality server and how to identify errors produced by the application easily, once the application is running.

6
Deployment

This chapter will cover the steps needed to make our web application live. Also, you will learn how to set up an SSL certificate and how to track errors occurring in production actively. This chapter will cover the following topics:

- Setting up Modulus.io
- Setting up Compose.io
- Automatic error tracking
- Setting up an SSL certificate

Setting up Modulus

Modulus is currently the best place to host a Meteor project. Why? It is easy to set up and maintain. While there are several other services where we can obtain the same results, every single one of them requires more expertise with servers and a considerable amount of time on your end. We are not developing a server. We are developing an application.

Modulus.io provides support for sticky sessions, web sockets, and free SSL endpoints.

Keep in mind that Meteor's Galaxy hosting service is going to be released soon and will undoubtedly become the best place to host a Meteor web application. Until then, Modulus is the way to go.

Let's start by creating a free account at `http://modulus.io`:

Next, you need to install the `demeteorizer` tool and the `modulus CLI` tool. Run the following commands in your terminal:

```
npm install -g demeteorizer
npm install -g modulus
```

This did not work! If your `npm install` command fails, then you need to install `node` and `npm`.

To make sure that we install `npm` properly, we need to add `homebrew` first. Install this by running this command:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once this is done, run this command:

```
brew install node
```

When this command finishes installing everything, we will have both `node` and `npm` available globally from our command line.

Now we can install both `demeteorizer` and `modulus` easily:

```
npm install -g demeteorizer
npm install -g modulus
```

Excellent. Now we have all the tools that we need. Let's create a new project (if you want to feel like a pro, you can use the CLI too):

Make sure to select **Node.JS** as the runtime environment for the project and set the size of memory of the servo to **192MB**. If you find there is a lot of traffic coming in, you can increase the memory of the server at any time to scale your application easily.

Once you have created your servo, go to the **ADMINISTRATION** tab:

Now copy the URL that has been generated for the project found next to the **Your Project URL** text:

Scroll all the way to the end of the page to the **ENVIRONMENT VARIABLES**. The first thing that we need to do is set the `ROOT_URL` environment variable. Paste the project URL here. Make sure it has the `https` protocol. Here, we are taking advantage of Modulus' secure SSL endpoint. The `wizonesolutions:canonical` package will make sure that all routes will hit our `ROOT_URL`:

Next, we need to set up our database. We cannot deploy this yet because the application is not aware of where it is supposed to save information.

Setting up Compose

Although Modulus.io offers MongoDB hosting, Modulus does not grant access to Meteor's **oplog tailing** feature and does not have any support for multiple replica sets. Both these features are must-haves for production and this is why.

A MongoDB Replica Set is an exact copy of your database. When you first create a database in Compose, you are automatically given a primary Replica Set with a secondary Replica Set to support it. Since failures happen, the secondary Replica Set exists to replace the primary replica set immediately when the primary fails. In the database world, this is known as data redundancy.

This layer of security adds an interesting problem; how do the secondary databases know about the changes that are happening to the primary? The operations log, or oplog for short, is a special collection that Mongo uses to record all the changes that are being made to the database. These changes are read by the Replica Sets to reflect all necessary changes.

Great, now let's understand how Meteor uses the oplog. Meteor's default `poll` and `diff` method of watching changes in the database is slow. It works by comparing changes between the database and the client every 10 seconds. This creates multiple unnecessary hits on your database by default and is not fast (since a change can occur 9 seconds before Meteor looks for changes) as well. To make Meteor perform better, the Meteor team tapped into Mongo's oplog. By listening for changes in Mongo's oplog, Meteor knows exactly when and which changes to push to the client. This is called oplog tailing.

Oplog tailing drastically improves Meteor's reactivity performance by effectively tailing Mongo's operations log. It is guaranteed that a production app will not run smoothly without taking advantage of this feature.

Create your account with `compose.io`:

Now create a new MongoDB:

Now click the **Users** tab. Here, we need to add a user that will have access to the database and the oplog. This is the user that Modulus will use. In the following example, we are setting the user to `root`, the password to `root`, and `oplog access` to `true`:

Click on the **Admin** tab. Copy the database URI that you see displayed under **Replica Set URI**:

Notice the `<user>` and `<password>` sections of the string. We are going to replace these fields with the user credentials that we had defined previously.

Go back to Modulus.io and visit the project's **ADMINISTRATION** page. Scroll to the **ENVIRONMENT VARIABLES** section. Here, we need to add the URI that we had just copied in to two different variables: `MONGO_URL` and `MONGO_OPLOG_URL`.

The `MONGO_URL` will look like this:

```
mongodb://root:root@candidate.1.mongolayer.com:10197,candidate.13.
mongolayer.com:10810/online_shop
```

Notice that you can remove the query parameters (anything after and including the question mark).

The `MONGO_OPLOG_URL` will look like this:

```
mongodb://root:root@candidate.1.mongolayer.com:10197,candidate.13.
mongolayer.com:10810/local?authSource=online_shop
```

Notice that we have modified the information after the trailing slash to `/local?authSource=online_shop` where `online_shop` is the name of the database.

Excellent! This is the last time we will have to configure anything for our project's deployment. Now we can use the `modulus` CLI to deploy.

Go to your terminal and run the following command to log in to your user:

```
modulus login
```

If you use your GitHub account to create your Modulus account, then pass the `github` flag:

```
modulus login --github
```

Follow the instructions on the command line. Once you are logged in, you will need to make sure that you are in your Meteor project's `root` directory (where you run the Meteor command). From here, run the following command to deploy to your new server:

```
modulus deploy
```

Modulus will ask which project you want to deploy to and begin. What does Modulus do? It first identifies that the project is a Meteor project, then it runs **demeteorizer** to convert the project into a common Node.js app. This app is then deployed to the server and automatically initiated.

Setting up Kadira

Perfect! Now that we can deploy production quality web applications, we need to understand how to identify issues with them. This is where `kadira.io` comes into the picture. Kadira is a Meteor-specific performance and error-monitoring tool. It will collect `Meteor.Errors` triggered on both the client and the server. It will also show performance data for publishers and subscribers.

Kadira does this and more, and the starting plan is completely free. Let's begin by signing up. After you have signed up, you need to create a new app:

Decide a name for the app, and enter the same in the field, and click on **Create App**:

Once created, you will see a view like this:

To finish, you will need to copy the code in step 2 and install the `meteorhacks:kadira` and `meteorhacks:zones` packages:

```
meteor add meteorhacks:kadira
meteor add meteorhacks:zones
```

The `meteorhacks:zones` package improves the description of the errors from the client side. Kadira will take advantage of this package automatically. It is important to note that the `meteorhacks:zones` package is optional because it is still in active development and can cause strange behavior in Meteor. Now, let's create a configuration file for Kadira:

```
# /_globals/server/kadira.coffee

if process.env.NODE_ENV is "production"
  Kadira.connect 'appId', 'secret'
```

We are done! With this simple configuration, we can now easily track application errors and more on Kadira:

You will notice the **Errors** tab on the top-left side of the screen. This tab will show us a list of all the errors that will occur in our application. Have a look at `meteorhacks` and their academy to learn more about optimizing your Meteor web application.

Setting up an SSL certificate

SSL, or **Secure Sockets Layer**, is a technology that creates an encrypted connection between clients and the server. This is necessary if we want to ensure that the data transferred to our server is encrypted; this includes data such as credit card information.

Setting up SSL can be painful because it requires some command-line knowledge. We like to buy our SSL certificates from `https://www.namecheap.com/` because they are cheap and they get the job done.

The cheapest SSL certificate that you can get is **PositiveSSL**; you can find the offering at this endpoint: `https://www.namecheap.com/security/ssl-certificates/single-domain.aspx`.

After buying the certificate, you will need to generate a **Certificate Signing Request** (**CSR**). Let's do this. You will be first redirected to your **Purchase Summary**. Click on **Manage**:

Now click **Activate Now** and leave the window open:

Next, open up a terminal at your project directory, and create a `/.csr` directory by running:

```
mkdir .csr
cd .csr
```

Now, let's use `openssl` to create our CSR. Run this command:

```
openssl req -nodes -newkey rsa:2048 -keyout private.key -out server.csr
```

This will give you a prompt for your site contact information; all the information fields must be filled out with your company or your information. The most important field is the **Common name** field, which must be `www.yourdomain.com`. Including the `www` will secure both `www.yourdomain.com` and `yourdomain.com`.

The command has created a `private.key` file and a `server.csr` file; you can check this by running:

```
ls
```

Keep `private.key` somewhere safe! You will need this later. Now open the `server.csr` file with any text editor such as Sublime, Atom, or, if you prefer something more basic, you can use nano or vim. You can view the contents of the current directory folder by running:

```
open .
```

Copy all the text in this file. This should look like this:

```
-----BEGIN CERTIFICATE REQUEST-----
A bunch of letters and numbers
-----END CERTIFICATE REQUEST-----
```

Go to namecheap, and change the select box to **Other** and paste the CSR to namecheap's **Enter csr** textarea field. This should look like this:

Now fill out the approver information:

Make sure that the e-mail you select is active and under your control! If you cannot receive e-mails from the e-mail that you have selected, you will not be able to apply the SSL certificate to your webapp! In the next screen, simply submit your order and wait for the **SSL Certificate Validation** e-mail to arrive:

SSL Certificate Validation

Domain Control Validation for:www.roc[redacted]

Dear **lep**[redacted]

We have received a request to issue an SSL certificate for:
Domain: **www.roc**[redacted]
Subject:
[redacted]

To permit the issuance of the certificate please browse here and enter the following "validation code":

[redacted]

**********PLEASE NOTE CHOOSING THE OPTION BELOW WILL REJECT THE CERTIFICATE**********
If neither you nor a trusted colleague made this request for
a certificate then you can reject it by browsing to Reject

Note! Rejecting the certificate at this stage will affect the whole order including the original certificate before the reissue attempt. It will not be possible to undo this action.

Kind Regards,

Comodo Security Services

Support Telephone: +1.888.266.6361 / +1.703.581.6361
Support Website: http://support.comodo.com/
Validation Docs Fax: US and Canada +1.866.831.5837 / Worldwide +1.801.303.9291

We now operate a registration-based system for support.
Please submit your ticket at the **support website**.
Please do not reply to this email as this email address is not monitored.

Copy your validation code, and click on the **here** link in the e-mail. This will take you to another site where you need to paste the validation code you copied and click on **Next**:

Close the window, and wait until you receive a new e-mail with a ZIP file. Download the ZIP file and unzip it. This will contain four files: `AddTrustExternalCARoot.crt`, `COMODORSAAddTrustCA.crt`, `COMODORSADomainValidationSecureServerCA.crt`, and `www_yourdomain_com.crt`.

Now, we need to use these files to create a Certificate Authority Bundle for Modulus. This is, basically, a concatenated version of all our certificates. Generate this by issuing this command in your command line where the certificates are located:

```
cat www_yourdomain_com.crt COMODORSADomainValidationSecureServerCA.crt
COMODORSAAddTrustCA.crt AddTrustExternalCARoot.crt > certificate_bundle.
crt
```

This command will produce a new `certificate_bundle.crt` directory with all the certificates concatenated.

Now browse to the Modulus.io administration page; make sure that you have already pointed your custom URL to your instance of Modulus by adding your domain name to the custom domains list. Click on the plus icon:

Now open the `private.key` file that was generated together with the CSR, and copy the entire text inside the file. Paste the information to the **Private Key** textarea. Then open the `certificate_bundle.crt` file, copy the information here, and paste it to the **Certificate** textarea:

Make sure to point your certificate to the `www` domain and the `non-www` domain.

Now to complete the process, we need to make sure that the `wizonesolutions:canonical` package routes traffic to the secured domain. Scroll to the **ENVIRONMENT VARIABLES** section, and replace `ROOT_URL` with your domain starting with the `https` protocol. Your `ROOT_URL` should look like this: `https://yourdomain.com`.

You might need to restart your server to make sure that all your settings take effect. Now go to your site, and you will see that you are automatically directed to a properly secured version of your site!

Summary

This chapter is straightforward. We learned how to deploy our application to a production server hosted by modulus.io. Also, we learned how to set up our Modulus project to a production quality database server provided by Compose. We chose Compose over Modulus because we can set up oplog tailing for Meteor through it. To help us track application errors, we installed Kadira. Also, we learned how to set up an SSL certificate for our server to secure our site further. With this knowledge, we can build production quality web applications.

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Building Single-page Web Apps with Meteor, Fabian Vogelsteller*
- *Meteor Cookbook, Isaac Strack*
- *Meteor Design Patterns, Marcelo Reyna*