Community Experience Distilled

Getting Started with Meteor.js JavaScript Framework

Second Edition

Learn to develop powerful web applications in minutes with Meteor

Isaac Strack

[PACKT] PUBLISHING open source* community experience distilled

Getting Started with Meteor.js JavaScript Framework

Second Edition

Learn to develop powerful web applications in minutes with Meteor

Isaac Strack

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Meteor.js JavaScript Framework

Second Edition

First published: December 2012

Second edition: June 2015

Production reference: 2290615

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-554-7

www.packtpub.com

Credits

Author
Isaac Strack

Reviewers
Netanel Gilad
Flávio Juvenal da Silva Junior
Arthur Pham

Commissioning Editor
Veena Pagare

Acquisition Editors
Subho Gupta
James Jones

Content Development Editor
Anish Sukumaran

Technical Editor
Menza Mathew

Copy Editor
Jasmine Nadar

Project Coordinator
Izzat Contractor

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Jason Monteiro

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Isaac Strack is a design technologist and STEM education advocate, currently working as a solutions consultant for Adobe Systems. With more than 15 years of experience in management information systems and web and creative technologies, Isaac has a strong background in modern web application development. He is the author of the Packt Publishing book *Meteor Cookbook* and the Packt Publishing video series *Learning Meteor Application Development*; he also assisted recently as a technical reviewer for another Packt Publishing book named *Building Single-page Web Apps with Meteor*. He holds a patent for online fraud detection and is a co-captain of the Salt Lake City Meteor Meetup group. He is an experienced lecturer/speaker. Isaac regularly mentors others at boot camps, training events, and conferences, such as UtahJS, DevMountain Meteor Day, NMC Summer Conference, Adobe workshops/events, and the Consumer Electronics Show (CES).

A huge thank you to my family, especially my mom, who have loved me despite instead of because. A shout out to my amazing daughters, without whom I wouldn't be long for this world. Sunshine, Monkey, Boogers, Pig, this one is for you.

About the Reviewers

Netanel Gilad is an enthusiastic developer with expertise in web development. He loves to learn everything, from new web development frameworks to setting up a Continuous Integration environment to creating the ultimate application architecture. He has led a team and worked on a mission-critical C2 web application with high-performance requirements and an emphasis on UX. Netanel is currently working on multiple web-related projects and is a coauthor of the popular Meteor package angular-meteor, which brings the worlds of Angular and Meteor together.

Flávio Juvenal da Silva Junior is a Brazilian software developer. He works at Vinta Software Studio (`http://www.vinta.com.br`), a software shop that uses state-of-the-art tools such as Django or Meteor to build web and mobile products from the backend to UX.

He believes that programming is a mix of art and engineering; therefore, the programmer has the right to choose the best tools (such as Meteor) to transform clients' expectations into elegant solutions.

Arthur Pham has been working for Thomson Reuters as a lead quantitative engineer since 2006. He has spent many years designing and implementing derivative pricing models and still loves to learn new programming languages such as F#, C++, Python, Flex/ActionScript, C#, Ruby, and JavaScript.

He currently lives in New York, USA, and can be contacted on twitter at `@arthurpham`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

We live in amazing times. Advances in medicine, communication, physics, and all other scientific fields provide us with opportunities to create things that were literally impossible to create only a short while ago.

Yet, we aren't easily amazed. Moore's law has not only affected how fast our computers are, it has significantly increased our expectations as well. We've come to expect wondrous advances, and therefore, what was once amazing has become…well…expected. It's a rare thing, indeed, to find something that takes us by surprise—something that renews that childhood sense of wonder we all secretly want back because it was stolen from us.

Well, get ready to regain some of that wonder. A dedicated group of computer scientists, who were determined to make something wondrous, have created a new JavaScript framework called Meteor. You may be thinking, "A new JavaScript framework? That's nothing special." And, if that's all Meteor is, you'd be correct. However, fortunately for you, that's not the end of the story.

Meteor is a reactive, simple, and powerful application platform, capable of producing sophisticated, robust web and mobile applications with just a few lines of code.

In the context of modern web applications, it is state-of-the-art. Using established, proven development design patterns, Meteor takes all the mundane parts of building an app and does them all for you. Therefore, you get to focus on building a solid application without getting bogged down with the usual time-wasting activities, such as writing yet another database interface or learning a new templating engine.

And the best part is, it's simple to learn, amazingly simple! You will see an application come to life right before your eyes, and when you look back at the number of lines of code it took to create and compare it to the traditional methods of development, you may actually find yourself saying "wow" or "how did they do that?"

This book will walk you through the major features of Meteor and show you how to create an application from scratch. By the end of the book, you will have created a working, useful application, and you will have a solid understanding of what makes Meteor different. This may sound like hyperbole, but if you're open to the idea that something innovative and unexpected can qualify as amazing, then prepare to be amazed!

What this book covers

Chapter 1, *Setup and Installation*, gets you up and running with Meteor in just a few minutes, and you'll see how quickly and easily you can build a fully functional and useful application.

Chapter 2, *Reactive Programming…It's Alive!*, teaches you all about reactive programming, and how you can leverage reactivity in Meteor to create amazing responsive applications.

Chapter 3, *Why Meteor Rocks!*, helps you to gain an understanding of the design patterns that Meteor uses and see examples of these powerful patterns in action.

Chapter 4, *Templates*, teaches you about Meteor Templates in depth and uses templates to lay the groundwork for your Lending Library application.

Chapter 5, *Data – Meteor Style!*, helps you to discover how Meteor handles data, making an enterprise-level application incredibly simple and robust. Implement Meteor's data handling quickly and effectively in your application.

Chapter 6, *Application Structure – Client, Server, and Public (oh my!)*, shows you what changes you can make to the default configuration to make your application more secure, extensible, and user-friendly.

Chapter 7, *Packaging and Deploying*, helps you to become an expert on Meteor's packaging system, including how to include many popular third-party frameworks. You will learn how to deploy a Meteor application to your developer, testing, and production environments.

What you need for this book

To run the examples in the book, the following software will be required:

- Operating systems:
  - Mac OS X 10.7 (Lion) and above versions
  - Linux x86 or x86_64 architectures
  - Windows 7 and above versions
- Meteor:
  - Meteor version 1.1 or above

The following table will guide you to sites that contain more information:

#	Software Name	URL
1	Mac OS X	`http://www.apple.com`
2	Linux	`http://www.debian.org` `http://www.redhat.com`
3	Windows	`https://www.microsoft.com`
4	Meteor	`https://www.meteor.com/install`

Who this book is for

This book is for an application developer, designer, or an analyst with a decent understanding of HTML and JavaScript who wants to learn about Meteor and the new movement inside the JavaScript community towards full-stack web and mobile applications.

If you are looking for a step-by-step approach to understand how and when to use one of the most popular and innovative application development frameworks, this book is for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `WebElement` class also supports find methods to find child elements."

A block of code is set as follows:

```
<form name="loginForm">
    <label for="username">UserName: </label> <input type="text"
        class="username" /></br>
    <label for="password">Password: </label> <input
        type="password" class="password" /></br>
    <input name="login" type="submit" value="Login" />
</form>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
//Locate all the Checkbox which are checked by calling jQuery
    find() method.
//find() method returns elements in array
List<WebElement> elements = (List<WebElement>)
    js.executeScript("return jQuery.find(':checked')");
```

Any command-line input or output is written as follows:

```
mvn clean test
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Right-click to open the pop-up menu and select the **Inspect element** option."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through 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 on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, 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 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.

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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, 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.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Setup and Installation

Under the hood, Meteor is really just a bunch of files and scripts, which are designed to make the building of a web application easier. That's a terrible way to describe something so elegant, but it helps us to better understand what we're using.

After all, Mila Kunis is really just a bunch of tissue wrapped around bone, with some vital organs inside. I know you hate me now for that description, but you get the point. She's beautiful. So is Meteor. But it doesn't do us any good to just leave it at that. If we want to reproduce that type of beauty on our own, we have to understand what's really going on.

So, files and scripts… We're going to walk you through how to get the Meteor package properly installed on your Linux or Mac OS X system, and then see the package of files and scripts in action.

Windows is now officially supported (Yay!) so you can follow along using the new Windows installation if you would like. Information can be found at `https://www.meteor.com/install`.

In this chapter, you will learn the following topics:

- Downloading and installing Meteor via curl
- Loading an example application
- Making changes and watching Meteor in action

Installing using curl

There are several ways to install a package of files and scripts. You can manually download and transfer files, you can use a pretty installation wizard/package with lots of **Next** buttons, or you can do what *real* developers do and use the command line. It puts hair on your chest. Which, now that I think about it, may not be a very desirable thing. Okay, no hair; we lied. But still, you want to use the command line, trust us. Trust the people that just lied to you.

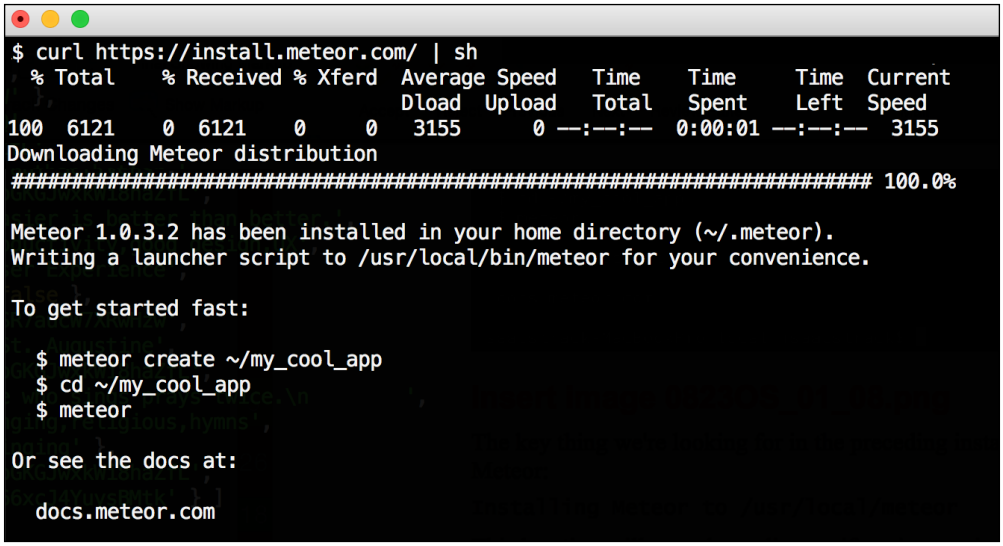
`curl` (or cURL if you want to get fancy) is a command-line tool used to transfer files and run scripts using standard URL locations. You probably already knew that, or you probably don't care. Either way, we've described it and we're now moving on to using it.

Open a terminal window or the command line, and enter the following command:

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

This will install Meteor on your system. `curl` is the command to go and fetch the script. `https://install.meteor.com` is the URL/location of the script, and `sh` is, of course, the location of the script interpreter "Shell", which will run the script.

Once you've run this script, assuming you have an Internet connection and the proper permissions, you will see the Meteor package downloaded and installed:



The key thing that we're looking for in the preceding installation text is the launcher script location:

```
Writing a launcher script to /usr/local/bin/meteor
```

This location could vary depending on whether you're running this script in Linux or Mac OS X, but it puts Meteor into a location where you can then access the Meteor script from anywhere else. This will become important in a minute. For now, let's see what kind of friendly message we get when the Meteor installation is finished:

```
To get started fast:

  $ meteor create ~/my_cool_app
  $ cd ~/my_cool_app
  $ meteor

Or see the docs at:

  docs.meteor.com
```

Great! You've successfully installed Meteor, and you're on your way to create your first Meteor web application!

You should bookmark `http://docs.meteor.com`, an invaluable reference moving forward.

Loading an example application

The wonderful people at Meteor have included several example applications, which you can quickly create and play with; these help you to get a better idea of what Meteor is capable of.

We want to use the simplest possible example, just to get an idea of how Meteor works, so we will be creating the `leaderboard` example. We'll be using the command line again. This is awesome news if you still have it open! If not, open a terminal window and follow these steps.

Selecting your file's location

So that we can remember where they are later, we'll put all the files for this book in the `~/Documents/Meteor` folder. We will create this folder as follows:

```
$ mkdir ~/Documents/Meteor
```

Now, we need to get to that directory. Use the following command:

```
$ cd ~/Documents/Meteor
```

Loading the example application

We can now use the Meteor `create` command with the `--example` parameter to create a local copy of the `leaderboard` example application:

```
$ meteor create --example leaderboard
```

As with the Meteor installation itself, the `create` command script has a friendly success message:

```
leaderboard: created.
To run your new app:
   cd leaderboard
   meteor
```

There are even instructions on what to do next. How handy! Let's go ahead and do what our good command-line friend is telling us.

Starting the example application

To start up a Meteor application, we need to be in the application directory itself. This is because Meteor looks for the startup files, HTML, and JavaScript that are needed to run the application. These are all found in the application folder, so let's go there:

```
$ cd leaderboard
```

This puts us in the `~/Documents/Meteor/leaderboard` folder, and we're ready to run the application:

```
$ meteor
```

Yes, that's it. Meteor takes care of everything for us; it reads all the files and scripts, and sets up the HTTP listener:

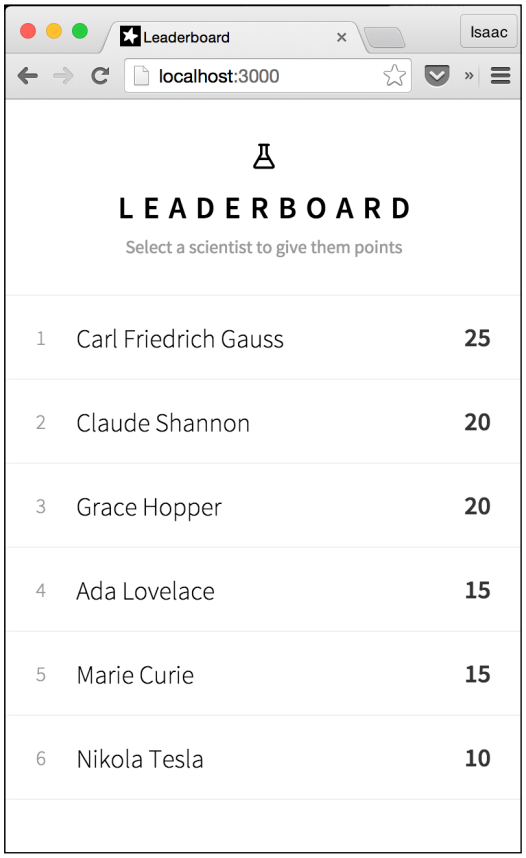
```
[[[[[ ~/Documents/Meteor/leaderboard ]]]]]


Running on: http://localhost:3000/
```

We can now take the URL we've been given (`http://localhost:3000/`) and check out the example application in a web browser.

Previewing the application

Open your favorite web browser (we'll be using Chrome, but any modern, updated browser will work) and navigate to `http://localhost:3000/`.

You should see a screen with a list containing the names of scientists, similar to the following screenshot:



You can go ahead and poke around the application a bit, if you want to. Click on Nikola Tesla's name and add 5 points to his score about 20 bajillion times, because he deserves it. Give some love to Marie Curie because she was so radioactive that she actually made up the word. Go nuts, friend!

Help! I made too many changes!

Do you fear change and want to reset the scores? No problem, we can start with a clean database instance; to do this, perform the following steps:

1. Open the command line, and press *Ctrl* + *C*.
2. This stops the running application. Now, enter the following command:

```
$ meteor reset
```

3. Restart your app, and you're good to go. Just type the following command:

```
$ meteor
```

Note that the initial scores are random, so it won't look exactly like it did before. The `meteor reset` command resets all the data collections and whatnot; so in a non-random app, the command will indeed reset the app cleanly.

Making code changes

Okay, we've got our application up and running in the browser, and we now want to see what happens when we make some code changes.

One of the best features of Meteor is reactive programming. The following extract is taken from `http://docs.meteor.com/#/full/reactivity`:

> *Meteor embraces the concept of reactive programming. This means that you can write your code in a simple imperative style, and the result will be automatically recalculated whenever data changes that your code depends on.*

This principle applies to code changes too, which means that any changes that you make to the HTML, JavaScript, or database are automatically picked up and propagated.

You don't have to restart the application or even refresh your browser. All changes are incorporated in real time, and the application reactively accepts the changes.

Let's see an example.

Changing from Leaderboard to Yay Science!

As you become more familiar with Meteor, you will come to learn that you can make changes and add files pretty much whenever you want. You don't have to link anything up, and you certainly don't have to redeploy before you can see the results. You get to just play around, build wonderful things, and let Meteor take care of all the crunchy stuff.

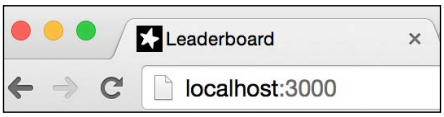
To see what we mean, let's change the title of this application from **Leaderboard** to **Yay Science!** because, well, yay science!

First, make sure that the application is up and running. You can do this by having an open browser window that is pointing to `http://localhost:3000/`. If the app is running, you'll see your `leaderboard` application. If your application isn't running, make sure to follow the steps previously given in the *Starting the example application* section.

Now, we need to open and edit the `leaderboard.html` file. With your favorite text/code editor, open the `leaderboard.html` file under the location, `~/Documents/Meteor/leaderboard/client/`, and change `title` in the `head` section using the following lines of code:

```
<head>
  <title>Yay Science!</title>
</head>
```

Go ahead and save the file, and then look at your web browser. The page will automatically update, and you'll see the title change. Earlier, it displayed the word **Leaderboard**:



However, now, after the execution of the preceding code, the title will display our spiffy new **Yay Science!** page:

This is Meteor in action! It monitors any changes to files, and when it sees that a file has changed, it tells your browser that a change has been made and that it should refresh itself to get the latest version.

Moving forward, we're going to build an application from scratch, so we don't want to make too many changes to this example application. However, we still want to stay with our new theme rather than that generic old leaderboard stuff. So, to do so, perform the following steps:

1. Back in your text editor, on about the tenth line or so, we have the title label for our leaderboard. Make the following change to the `<h1>` tag:

   ```
   <h1 class="title">Yay Science!</h1>
   ```

   Save this change, and you'll see the change reflected in your browser. The title in our page will now look like this:

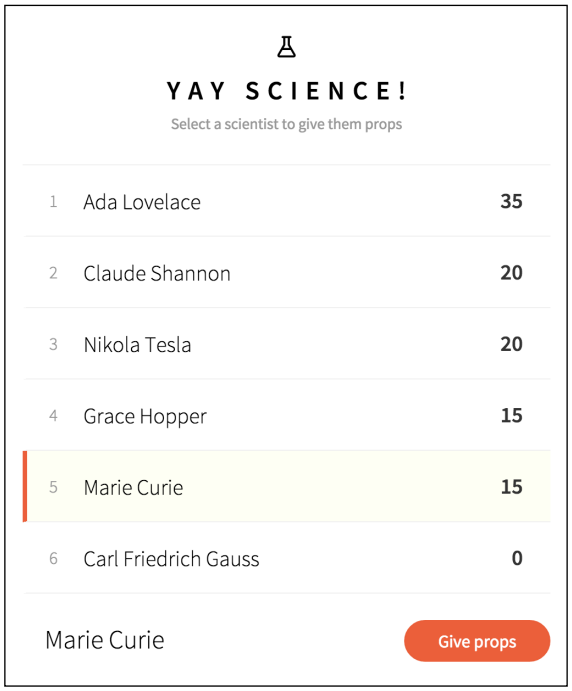
2. Likewise, we don't give "points" to scientists. They're not trained monkeys, dancing around for our amusement. They're scientists! We give them mad props instead. In the `<div>` tag just below our title, make the following text change:

   ```
   <div class="subtitle">Select a scientist to give them props</div>
   ```

   We also need to change the button text from the word `points` to the word `props`. Towards the bottom half of the file, you'll find a `<button>` tag. Change the text in that tag to the following:

   ```
   <button class="inc">Give props</button>
   ```

Save your changes, and you will see the application update almost immediately:



3. Just below the `<button>` tag, there is a message displayed if no scientist's name is selected. It currently uses the word "players". We want to change that to something a little more specific. To do this, make the following change to the `<div>` message tag:

   ```
   <div class="message">Click a name to select</div>
   ```

   Save this change, and this time, refresh your browser. Not because we need the change to take effect, but because we want to make sure no scientist is highlighted so that we can verify our message text:

Summary

Great success! In this chapter, you've successfully installed the Meteor framework, loaded an example application, and made changes to that application by becoming familiar with file changes and the reactive nature of Meteor.

You are now ready to start building your very own Meteor application, and learn more about the elegant features and advantages that come from developing with Meteor in the coming chapters.

2
Reactive Programming… It's Alive!

As we learned in *Chapter 1*, *Setup and Installation*, Meteor operates on a reactive programming model. This means that your templates aren't only concerned with displaying data, but they are also listening for changes to that data so that they can "react" to those changes. These areas of data where the templates look for changes are called **reactive contexts**.

We will now start developing a Lending Library application, lay the framework for future chapters, and use Meteor's built-in reactive model to track and propagate changes to our application to all clients that are listening.

In this chapter, you will learn about:

- Creating your first real application
- Using reactive programming to track and automatically update changes
- Exploring and testing changes made to your data from multiple browser windows

Creating the Lending Library

There are two kinds of people in this world—those who remember who they lent something to and those who buy a *lot* of stuff twice. If you're one of the people who is on a first-name basis with your UPS delivery driver, this application is for you!

Using Meteor, we're going to build a Lending Library. We'll keep track of all our stuff and who we lent it to, so that the next time we can't remember where we put our linear compression wrench, we can simply look up who we lent it to last and get it back from them.

And when that same friend says, "Are you sure you lent it to me?", you can say, "Yeah, STEVE, I'm sure I lent it to you! I see you're enjoying your digital cable, thanks to my generous lending of said linear compression wrench. Why don't you go find it so that I too can enjoy the benefits of digital cable in my own home!"

Okay, okay, maybe STEVE forgot too. Maybe he's a dirty liar, and he sold your wrench to pay for his deep-fried Oreo® habit. Either way, you'll have your very own custom Meteor app that gives you proof that you're not going crazy. And, if he did sell it for deep-fried carnival food, at least you can make him share his stash with you while you watch the game at his house.

Creating the base application

The first thing we want to do is create the base application, similarly to what we did in the first chapter, and we can then expand the base application to fit our needs:

1. Start by navigating to your applications folder. This can be anywhere, but as mentioned in our earlier chapter, we'll be working out of `~/Documents/Meteor` as our root folder:

   ```
   $ cd ~/Documents/Meteor
   ```

2. Now, we create our base folder structure for our Lending Library application:

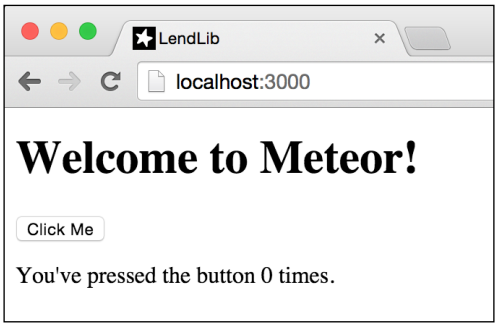
   ```
   $ meteor create LendLib
   ```

3. As usual, we'll give instructions on how to get the application up and running. Let's go ahead and try that, just to make sure that everything was created properly:

   ```
   $ cd LendLib
   $ meteor
   ```

   This navigates to the Lending Library folder under `~/Documents/Meteor/LendLib` and runs the application.

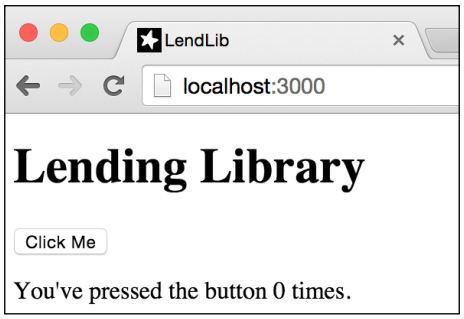
4. Open a browser and navigate to `http://localhost:3000/`. You should see the following screen:



5. **Welcome to Meteor!** is nice and all, but we are going to change this to **Lending Library**. Open the `LendLib.html` file under `~/Documents/Meteor/LendLib/` in your favorite editor. Towards the top (the sixth line or so), you'll see the HTML code snippet that's responsible for our greeting. Go ahead and change `Welcome to Meteor!` to `Lending Library`:

```
<body>
  <h1>Lending Library</h1>
```

6. Save the change. The page will refresh and will look like the following screenshot:



You may have noticed a reference to a template called `hello`, just below the title:

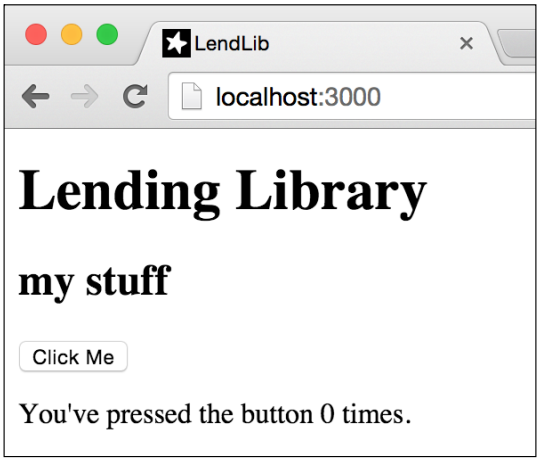
```
{{> hello}}
```

This is called the **template inclusion** because we are including a template. By adding this to the body of our code, we instruct Meteor to include the template named `hello` and append its contents inside our `<body>` tag.

7. Let's make a change in this template as well. Edit `LendLib.html` and modify the title of the `hello` template:

```
<template name="hello">
  <h2>my list</h2>
  <button>Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>
</template>
```

8. Save the change and your page will update as follows:



Creating a Collection

Okay, so you've just made a few small changes to static files, but what we really want to see is some dynamic, reactive programming, and some live HTML code!

We need to attach a data source—something that will keep track of our items. Normally, this would be quite a process indeed, but Meteor makes it easy by supporting MongoDB, and its own client-side version called **Minimongo**, out of the box.

To learn more about NoSQL databases (and specifically MongoDB, the default database used inside Meteor), you can visit the following sites:

- `http://en.wikipedia.org/wiki/NoSQL`
- `http://www.mongodb.org/`
- `http://www.packtpub.com/books/all?keys=mongodb`

Let's create our collection. Inside the `LendLib.js` file under `~/Documents/Meteor/LendLib/`, we want to add the following highlighted line of code as the first line, and then save the change:

```
lists = new Mongo.Collection("lists");

if (Meteor.isClient) {
...
```

This creates a new collection in MongoDB. Since it comes before anything else in the `LendLib.js` file, the collection is available for both the client and server to see. It is persistent, as we'll see in a moment, and once values are entered into it, they can be retrieved by any client accessing the page.

To see this persisted object, we'll need to use the console of our web page.

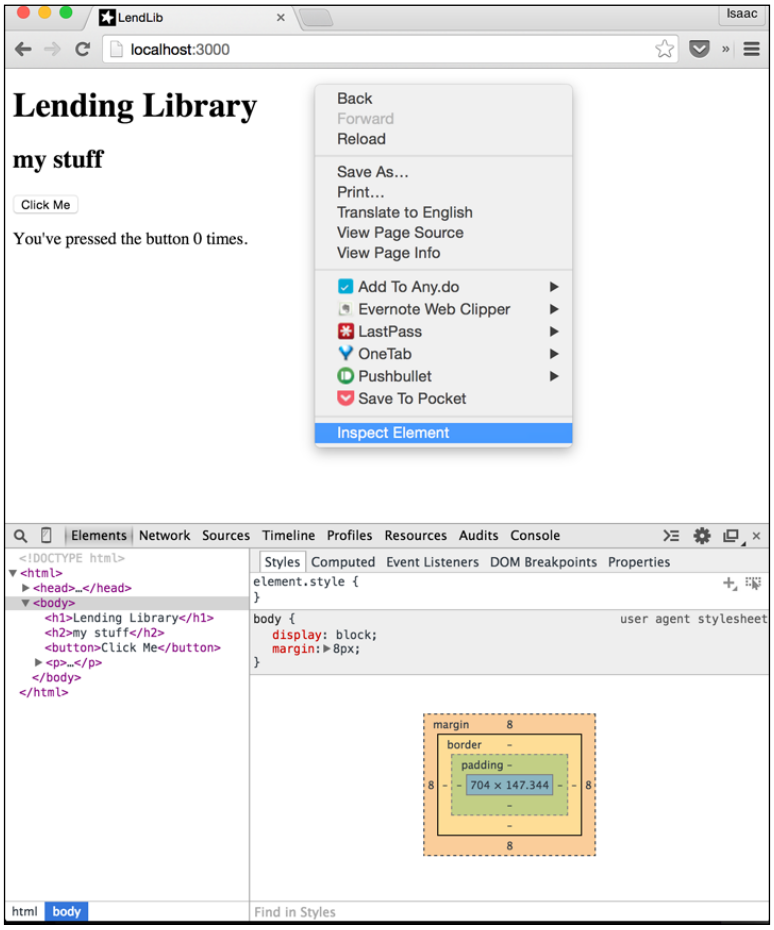
Having fun with the browser console

The **browser console** is a debugging tool available in most modern browsers by default, or as an add-on through plugins.

For a more in-depth tutorial on using the console in Chrome, check out `http://developer.chrome.com/extensions/tut_debugging.html`.

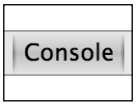
Since we're using Chrome, the console is available by default. Let's start by performing the following steps:

1. In a browser window pointing to `http://localhost:3000/`, enter the shortcut key combination *command* + *option* + *I*, or you can right-click anywhere on the page and select **Inspect Element**:



   This will open our debugging tools. We now want to get into the console.

2. Click on the **Console** icon found at the extreme right of the debugging menu bar:

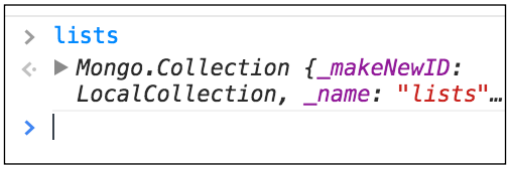


You will now have a blinking cursor, and you're ready to check out our newly minted collection!

3. Enter the following command in the console and hit *enter*:

```
> lists
```

You should get a returned object that says **Mongo.Collection**:



Adding some data

The previous output means that our changes were accepted, and we have a new persistent collection! It's empty, but let's do something about that:

1. Enter the following commands in the browser console to create a couple of sample **Categories**:

```
> lists.insert({Category:"DVDs", items: [{Name:"Mission Impossible
",Owner:"me",LentTo:"Alice"}]});


> lists.insert({Category:"Tools", items: [{Name:"Linear
Compression Wrench",Owner:"me",LentTo: "STEVE"}]});
```

After each command, you'll get a GUID (something like `ehkGMaxmjWqqXAMdq`), which is Meteor's way of telling you that the item was saved properly.

2. Being the natural skeptics that we are, we're going to check. To do so, enter the following command:

```
> lists.findOne({Category: "DVDs"});
```

You should then get an **Object** with an expandable icon next to it.

3. Click on this icon to expand it, and you should see something similar to the following screenshot:

We could similarly check for our tools collection by entering the `lists.findOne({Category:"Tools"})` command, but we don't need to. This time we'll trust that Meteor entered it correctly. We *do*, however, want to check to see whether the objects are persistent. To do so, perform the following steps:

1. Refresh the web page. Your console will clear, but the `Categories` that we entered have been saved in the persistent Meteor Collection, so we can check again to see if they're hanging around.
2. Enter the following command in the console:

   ```
   > lists.find({}).count();
   ```

   This command finds all records in the `lists` collection and gives us a total count. If everything went according to plan, you should have got back a count of **2**.

We're on our way! We've created two categories, and we have one item in each category. We've also verified that the `lists` collection is being saved after each session. Now, let's see about displaying this in our page.

Displaying collections in HTML

We're now going to see our collection come to life inside the HTML page that we created when we initialized our project. This page will use templates that are reactive, allowing us to have changes made to our Collection appear instantly, without a page refresh. These types of reactive templates, where the DOM for the page can be updated without a refresh are called **Live HTML Templates**.

To read more about Live HTML Templates, consult the Meteor documentation at `http://docs.meteor.com/#/full/livehtmltemplates`.

Let's perform the following set of steps:

1. With the `LendLib.html` file under `~/Documents/Meteor/LendLib/` still open, locate the `<body>` tag and add a new template inclusion:

```
<body>
  <h1>Lending Library</h1> {{> hello}}
  <div id="categories-container" class="container">
    {{> categories}}
  </div>
</body>
```

   This creates a new `div` element with the contents being filled by a **template partial** named categories.

2. Now, at the very bottom of the page, add the skeleton for the categories:

```
<template name="categories">
</template>
```

   This won't change the appearance of the page, but we now have a template partial where we can list our categories.

3. Let's put in our section title within the preceding lines of code:

```
<template name="categories">
<div class="title">categories</div>
</template>
```

4. And now, let's get our categories in there:

```
<template name="categories">
<div class="title">categories</div>
<div id="categories">

</div>
</template>
```

   This creates the `div` categories which we can then go through and list all of our categories. If we only had one record to deal with, the code would look like this:

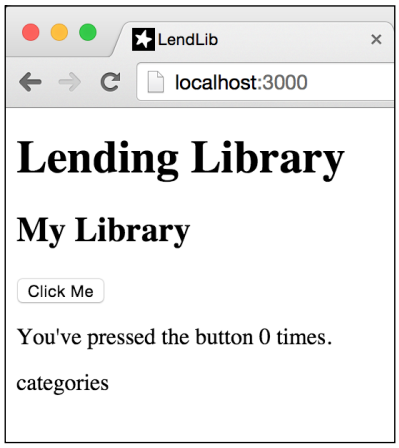
```
<div class="category">
{{Category}}
</div>
```

5. But, we need to wrap this into a loop (in this case, an `#each` statement) so that we get all the categories:

```
<template name="categories">
<div class="title">categories</div>
<div id="categories">
  {{#each lists}}
    <div class="category">
      {{Category}}
    </div>
  {{/each}}
</div>
```

   Notice that we are telling the template "for each record in the lists collection" with our `{{#each lists}}` command, and then, "display the Category" with `{{Category}}`.

6. If you save these changes and look at the web page, you will see something like the following screenshot:



This doesn't look much different. Yes, we have our header (**categories**), but where are the categories for which we just created our template?

There's one more step we need to complete for the categories to show up. Currently, the template that we just created isn't pointing towards anything. In other words, we have a `lists` collection, and we have a template, but we don't have the underlying JavaScript function that hooks them together. Let's take care of this by performing the following steps:

1. Open the `LendLib.js` file under `~/Documents/Meteor/LendLib/`, and we can see some `Template` functions:

   ```
   Template.hello.helpers({ ...

   ...

   Template.hello.events({ ...
   ```

   These code chunks hook up JavaScript functions and objects to the HTML `hello` template. Meteor's built-in `Template` object makes this possible, and we're going to follow the same pattern, that is, to hook up our `categories` template.

2. We want to declare that the `categories` template has a `lists` collection. We do this by entering the following code, just below the `Template.hello.events()` block:

   ```
     Template.hello.events({
       ...
     });
     Template.categories.helpers({
       lists : function(){
       }
     });
   }
   ```

   The `Template` declaration *must* be inside of the `if (Meteor.isClient){...}` code block so that the client will pick up the change and the server will ignore it.

3. We've now declared the `lists` collection for all the templates to use, and we can have the function return the results from a `Meteor.Collection` query. We can do that using the `find()` command:

   ```
   lists : function(){
     return lists.find({}, { sort: { Category : 1 }});
   }
   ```

This code will find every record in the `lists` collection and will sort the results by the `Category` (name). Save these changes, and you will now see a populated list of categories:

Cleaning up

We're fast approaching a working application, and we want it to look super-shiny and clean. Let's do a bit of clean up in our code and add some CSS to make things more readable. To do this, perform the following steps:

1. We don't need the greeting anymore. So, let's get rid of it. To do this, remove the following lines from `LendLib.html` and save the page:

```
<body>
  <h1>Lending Library</h1>
  {{> hello}}
  <div id="categories-container" class="container">
    {{> categories}}
  </div>
</body>

<template name="hello">
  <h2>My Library</h2>
  <button>Click Me</button>
  <p>You've pressed the button {{counter}} times.</p>
</template>
```

2. We'll keep the `Template.hello` declarations in `LendLib.js` for now, as a reference. We'll comment them out for now and remove them later when they're no longer needed:

```
/* Template.hello.helpers({
    ...
```

```
});
Template.hello.events({
  ...
});  */
```

3. Now, let's add the Twitter Bootstrap framework, which gives us a lot of style without much effort. Using a terminal window, create a `client` folder in `/LendLib/`:

```
$ mkdir ~/Documents/Meteor/LendLib/client
```

Download the latest Bootstrap framework at `http://getbootstrap.com/getting-started/` and extract the archive into the `client` folder under `~/Documents/Meteor/LendLib/`.

Since Meteor will read and use every file put into the application folder, we want to eliminate the redundant files. We don't have to worry too much about efficiency, but some things are just shameful and leaving that much extraneous code lying around is right up there with enjoying deep-fried carnival food.

So, let's perform the following set of steps:

1. Navigate to the Bootstrap folder (the name will vary) using the following command:

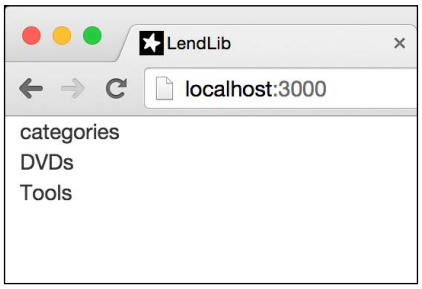
```
$ cd ~/Documents/Meteor/LendLib/client/[bootstrap]/
```

2. Once you've entered the folder, delete the unneeded files:

```
$ rm css/bootstrap-theme.c*
$ rm css/bootstrap.c*
$ rm js/npm.js
$ rm js/bootstrap.js
```

If you know what you're doing with Bootstrap, you can just copy the `fonts`, `min.js`, and `min.css` files instead of performing the preceding instructions.

After all these changes, your UI should be really clean and simple:



Let's quickly make the UI more distinct and readable. To do this, we must perform the following set of steps:

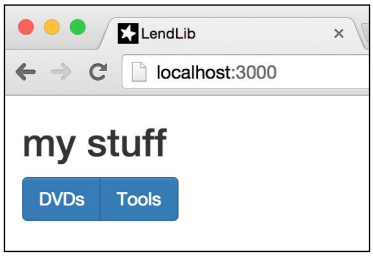
1. In `LendLib.html`, let's change our header from a `<div>` tag to an `<h2>` tag, and change the text from **categories** to **my stuff**:

```
<template name="categories">
<h2 class="title">my stuff</h2>
```

2. Let's turn `categories` into a pretty button group:

```
<div id="categories" class="btn-group">
    {{#each lists}}
     <div class="category btn btn-primary">
           {{Category}}
     </div>
{{/each}}
```

This gives us a distinct, clean-looking page:



Creating a reaction

Following the creation of our basic template and collection, and with Meteor putting our `lists` collection into the reactive context, we can now proceed to watch the reactive programming model in action.

Navigate to our Lending Library page at `http://localhost:3000/` and open the browser console window.

In the console, enter the following command:

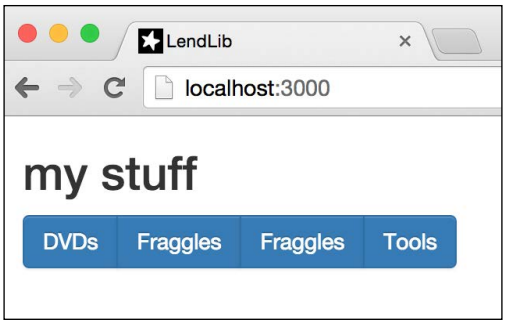
```
> lists.insert({Category:"Fraggles"});
```

You will instantly see the page update. Note that, this time, the full page didn't refresh! This is because, under the hood, Meteor is tracking changes to our reactive context (in this case, the `lists` collection) and the template is being updated immediately after a change is made.

Let's make a few more changes. To do this, enter the same `Fraggles` command again:

```
> lists.insert({Category:"Fraggles"});
```

Just as before, a new **Fraggles** button will instantly appear:



But we have too many `Fraggles` categories now. There *are* a lot of Fraggles, but unless you're some weirdo collector, you don't need *two* categories. So, let's remove them. However, we can't just erase whatever we want on the client side. This is a basic safety feature, and if we just let it happen, things could get quite chaotic. For instance, if we tried to just remove the records using the following command:

```
> lists.remove({Category:"Fraggles"});
```

...then that's quite a no-no. Meteor detects stuff like this and will give us a `403` error as follows:

There are three ways in which we can manually delete or modify records.

- The first is on the client side, by finding the ID of the record. In your browser console, run the following command:

  ```
  > lists.findOne({Category:"Fraggles"});
  ```

  This will return a single record, including an `_id` property, similar to the following screenshot:

  If we take this `_id` and execute `lists.remove()`, the extra `Fraggles` category will be removed:

  ```
  > lists.remove({_id:"Jz68JR4gBbiBcbLPt"});
  ```

- The second way is to use `meteor shell`. In a new terminal window (keep your app running!), navigate to `~/Documents/Meteor/LendLib/` and enter the following command:

  ```
  $ meteor shell
  ```

  This will open a console that is directly connected to the server so that we can run commands as if they were server code; for example:

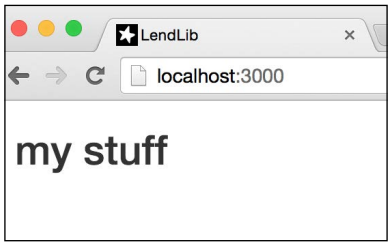
  ```
  > lists.remove({Category:"Fraggles"});
  ```

  This will remove all categories that are named `Fraggles`. The `meteor shell` command comes in handy when we need to run tests or debug, so keep this command in your pocket to use when you need it.

- The final way to delete records is how we did it in the previous chapter, with `meteor reset`. In fact, let's do that right now. Stop your application from running (*Ctrl* + *C*) and execute the following command in the terminal:

  ```
  $ meteor reset
  ```

Start Meteor again with the `meteor` command, and your application screen should be nice and clean:



It would probably be good to have a couple of categories, so let's create them really quickly. In the browser console, enter the following commands:

```
> lists.insert({Category:"Collectibles"});
> lists.insert({Category:"DVDs"});
> lists.insert({Category:"Tools"});
```

As you can see, the changes are made instantly, with no page refresh.

Multiple clients

Good things should be shared. Meteor gets this, and as we're about to see for ourselves, the reactive programming model allows us to share updates in real time across multiple clients.

With your Chrome web page still open to `http://localhost:3000/`, open a new browser tab and navigate to the same page.

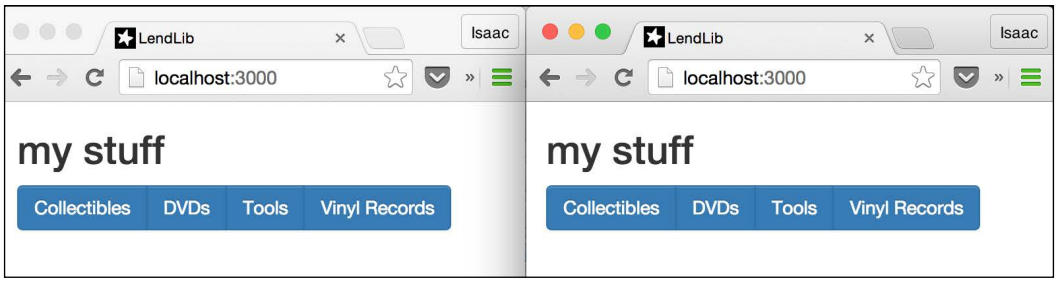
If you really want to get fancy, you can conduct this same experiment with multiple browsers (Firefox, Opera, or Safari). Each session will be live and reactive!

You now have two clients open, which is simulating the application being open by different people, at different locations, with different computers. Meteor's reactive model allows you to treat all clients in the same manner, and a change made by one will be propagated to all the others.

With your eyes on the new second browser, type the following command into the console of browser #1:

```
> lists.insert({Category:"Vinyl Records"})
```

You will notice that the change propagates to both browsers, without a page refresh:



Feel free to make any extra collections, to remove or rename them, and so on. Experiment a little and notice how these changes can be instantly made to every listening client. Meteor operates under a very powerful paradigm, and in the next chapter, you'll be able to see exactly why this is such an important and disruptive change to web application development.

Summary

In this chapter, you successfully created the framework for your new Meteor application. You saw firsthand how quickly a new project can be created, and you created database and template functionality with just a few lines of code. You saw live HTML and reactive programming in action, and you are now ready to go even deeper into the Meteor engine. You've conquered the tip of the iceberg, my friend. Take a break, have a cold one, and get ready for even more Meteor awesomeness!

3
Why Meteor Rocks!

Meteor is a disruptive (in a good way!) technology. It enables a new type of web application that is faster, easier to build, and takes advantage of modern techniques, such as **Full Stack Reactivity**, **Latency Compensation**, and **Data On The Wire**.

This chapter explains how web applications have changed over time, why that matters, and how Meteor specifically enables modern web apps through the above-mentioned techniques.

By the end of this chapter, you will have learned:

- What a modern web application is
- What Data On The Wire means and how it's different
- How Latency Compensation can improve your app experience
- Templates and Reactivity—programming the reactive way!

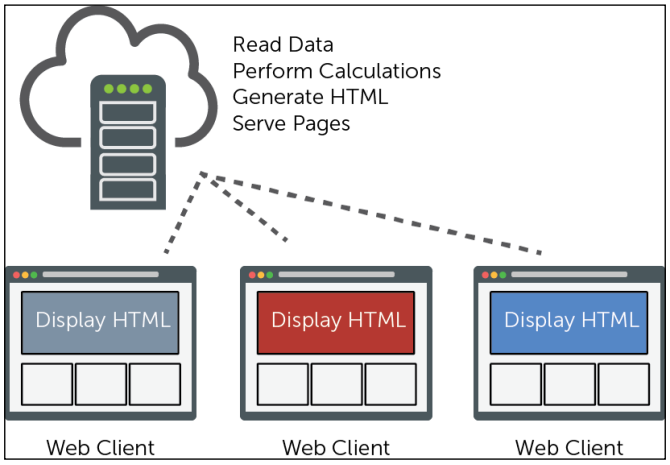
Modern web applications

Our world is changing. With continual advancements in displays, computing, and storage capacities, things that weren't even possible a few years ago are now not only possible but are critical to the success of a good application. The Web in particular has undergone significant change.

The origin of the web app (client/server)

From the beginning, web servers and clients have mimicked the **dumb terminal** approach to computing where a server with significantly more processing power than a client will perform operations on data (writing records to a database, math calculations, text searches, and so on), transform the data and render it (turn a database record into HTML and so on), and then serve the result to the client, where it is displayed for the user.

In other words, the server does all the work, and the client acts as more of a display, or a dumb terminal. This design pattern for this is called…wait for it…the **client/server** design pattern. The diagrammatic representation of the client-server architecture is shown in the following diagram:



This design pattern, borrowed from the dumb terminals and mainframes of the 60s and 70s, was the beginning of the Web as we know it and has continued to be *the* design pattern that we think of when we think of the Internet.

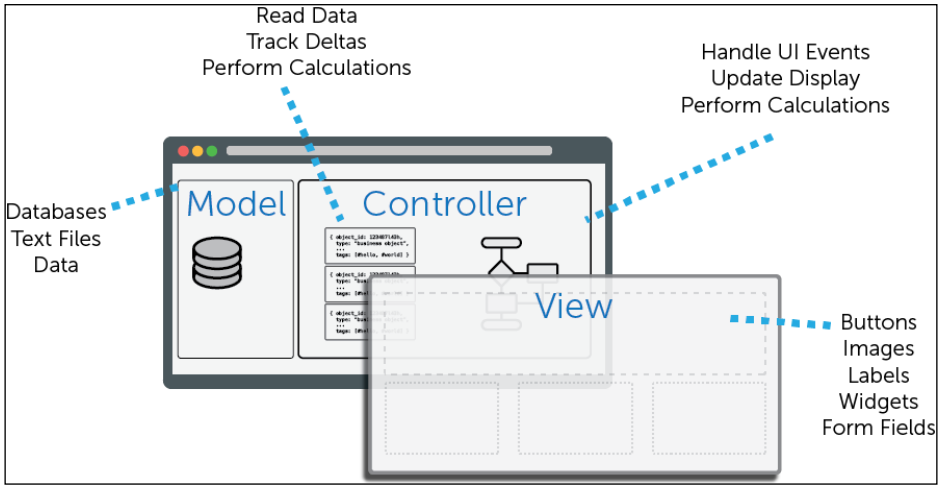
The rise of the machines (MVC)

Before the Web (and ever since), desktops were able to run a program such as a spreadsheet or a word processor without needing to talk to a server. This type of application could do everything it needed to, right there on the big and beefy desktop machine.

During the early 90s, desktop computers got even more beefy. At the same time, the Web was coming alive, and people started having the idea that a hybrid between the beefy desktop application (a **fat app**) and the connected client/server application (a **thin app**) would produce the best of both worlds. This kind of hybrid app—quite the opposite of a dumb terminal—was called a **smart app**.

Many business-oriented smart apps were created, but the easiest examples can be found in computer games. **Massively Multiplayer Online** games (**MMOs**), first-person shooters, and real-time strategies are smart apps where information (the data **model**) is passed between machines through a server. The client in this case does a *lot* more than just display the information. It performs most of the processing (or acts as a **controller**) and transforms the data into something to be displayed (the **view**).

This design pattern is simple but very effective. It's called the **Model View Controller** (**MVC**) pattern.



The **model** is essentially the data for an application. In the context of a smart app, the model is provided by a server. The client makes requests to the server for data and stores that data as the model. Once the client has a model, it performs actions/logic on that data and then prepares it to be displayed on the screen. This part of the application (talking to the server, modifying the data model, and preparing data for display) is called the **controller**.

The controller sends commands to the **view**, which displays the information. The view also reports back to the controller when something happens on the screen (a button click, for example). The controller receives the feedback, performs the logic, and updates the model. Lather, rinse, repeat!

Since web browsers were built to be "dumb clients", the idea of using a browser as a smart app back then was out of question. Instead, smart apps were built on frameworks such as Microsoft .NET, Java, or Macromedia (now Adobe) Flash. As long as you had the framework installed, you could visit a web page to download/run a smart app.

Sometimes, you could run the app inside the browser, and sometimes, you would download it first, but either way, you were running a new type of web app where the client application could talk to the server and share the processing workload.

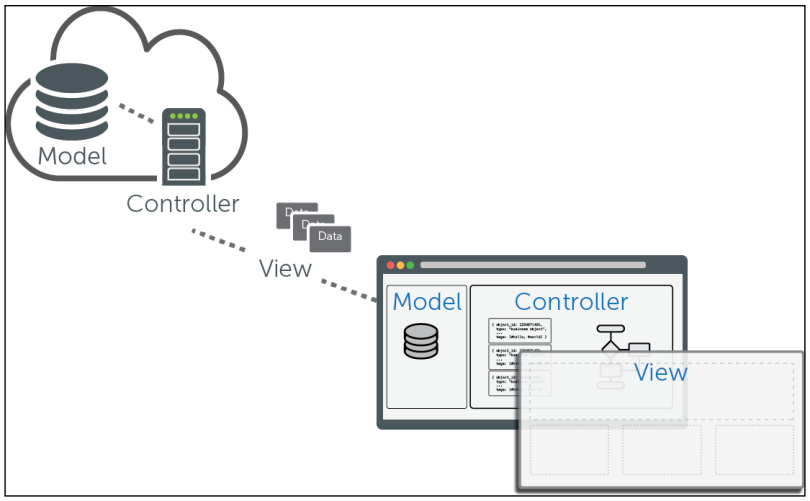
The browser grows up

Beginning in the early 2000s, a new twist on the MVC pattern started to emerge. Developers started to realize that, for connected/enterprise "smart apps", there was actually a nested MVC pattern.

The server code (controller) was performing business logic against the database (model) through the use of business objects and then sending processed/rendered data to the client application (a "view").

The client was receiving this data from the server and treating it as its own personal "model". The client would then act as a proper controller, perform logic, and send the information to the view to be displayed on the screen.

So, the "view" for the server MVC was the "model" for the client MVC.



As browser technologies (HTML and JavaScript) matured, it became possible to create smart apps that used the Nested MVC design pattern directly inside an HTML web page. This pattern makes it possible to run a full-sized application using only JavaScript. There is no longer any need to download multiple frameworks or separate apps. You can now get the same functionality from visiting a URL as you could previously by buying a packaged product.

A giant Meteor appears!

Meteor takes modern web apps to the next level. It enhances and builds upon the nested MVC design pattern by implementing three key features:

- Data On The Wire through the Distributed Data Protocol (**DDP**)
- Latency Compensation with **Mini Databases**
- Full Stack Reactivity with **Blaze** and **Tracker**

Let's walk through these concepts to see why they're valuable, and then, we'll apply them to our Lending Library application.

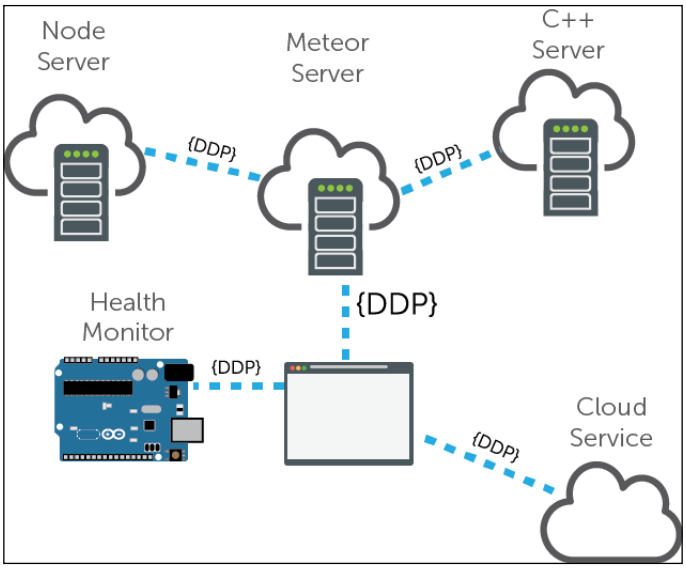
Data On The Wire

The concept of Data On The Wire is very simple and in tune with the nested MVC pattern; instead of having a server process everything, render content, and then send HTML across the wire, why not just send the data across the wire and let the client decide what to do with it?

This concept is implemented in Meteor using the Distributed Data Protocol, or **DDP**. DDP has a JSON-based syntax and sends messages similar to the REST protocol. Additions, deletions, and changes are all sent across the wire and handled by the receiving service/client/device. Since DDP uses WebSockets rather than HTTP, the data can be pushed whenever changes occur.

But the true beauty of DDP lies in the generic nature of the communication. It doesn't matter what kind of system sends or receives data over DDP—it can be a server, a web service, or a client app—they all use the same protocol to communicate. This means that none of the systems know (or care) whether the other systems are clients or servers. With the exception of the browser, any system can be a server, and without exception, any server can act as a client. All the traffic looks the same and can be treated in a similar manner.

In other words, the traditional concept of having a single server for a single client goes away. You can hook multiple servers together, each serving a discreet purpose, or you can have a client connect to multiple servers, interacting with each one differently. Think about what you can do with a system like that:

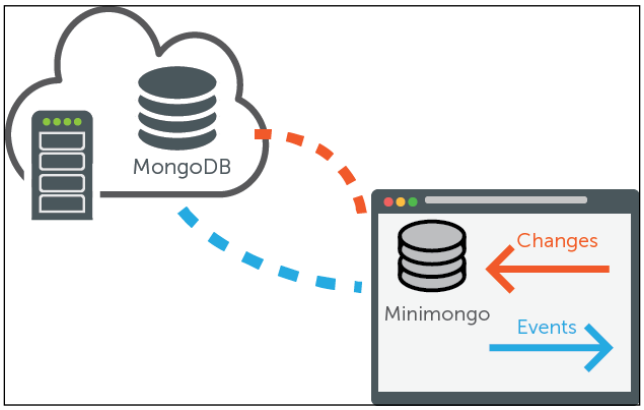


Imagine multiple systems all coming together to create, for example, a health monitoring system. Some systems are built with C++, some with Arduino, some with…well, we don't really care. They all speak DDP. They send and receive data on the wire and decide individually what to do with that data. Suddenly, very difficult and complex problems become much easier to solve. DDP has been implemented in pretty much every major programming language, allowing you true freedom to architect an enterprise application.

Latency Compensation

Meteor employs a very clever technique called **Mini Databases**. A mini database is a "lite" version of a normal database that lives in the memory on the client side. Instead of the client sending requests to a server, it can make changes directly to the mini database on the client. This mini database then automatically syncs with the server (using DDP of course), which has the actual database.

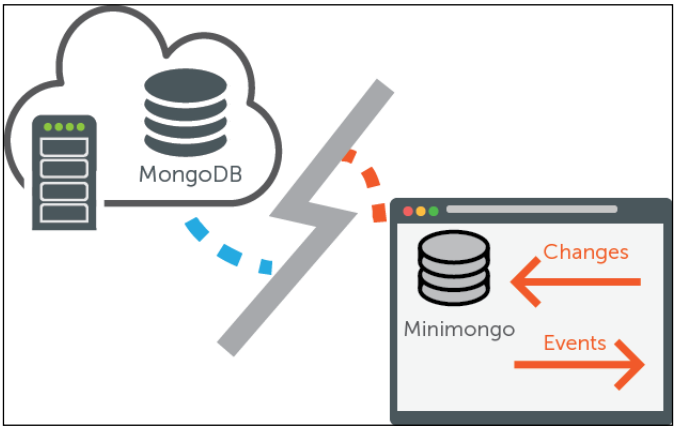
Out of the box, Meteor uses MongoDB and Minimongo:



When the client notices a change, it first executes that change against the client-side **Minimongo** instance. The client then goes on its merry way and lets the Minimongo handlers communicate with the server over DDP. If the server accepts the change, it then sends out a "changed" message to all connected clients, including the one that made the change. If the server rejects the change, or if a newer change has come in from a different client, the Minimongo instance on the client is corrected, and any affected UI elements are updated as a result.

All of this doesn't seem very groundbreaking, but here's the thing—it's all asynchronous, and it's done using DDP. This means that the client doesn't have to wait until it gets a response back from the server. It can immediately update the UI based on what is in the Minimongo instance. What if the change was illegal or other changes have come in from the server? This is not a problem as the client is updated as soon as it gets word from the server.

Now, what if you have a slow internet connection or your connection goes down temporarily? In a normal client/server environment, you couldn't make any changes, or the screen would take a while to refresh while the client waits for permission from the server. However, Meteor compensates for this. Since the changes are immediately sent to Minimongo, the UI gets updated immediately. So, if your connection is down, it won't cause a problem:



All the changes you make are reflected in your UI, based on the data in Minimongo. When your connection comes back, all the queued changes are sent to the server, and the server will send authorized changes to the client.

Basically, Meteor lets the client take things on faith. If there's a problem, the data coming in from the server will fix it, but for the most part, the changes you make will be ratified and broadcast by the server immediately.

Coding this type of behavior in Meteor is crazy easy (although you can make it more complex and therefore more controlled if you like):

```
lists = new Mongo.Collection("lists");
```

This one line declares that there is a `lists` data model. Both the client and server will have a version of it, but they treat their versions differently. The client will subscribe to changes announced by the server and update its model accordingly. The server will publish changes, listen to change requests from the client, and update *its* model (its master copy) based on these change requests.

Wow, one line of code that does all that! Of course, there is more to it, but that's beyond the scope of this chapter, so we'll move on.

To better understand Meteor data synchronization, see the *Publish and subscribe* section of the meteor documentation at `http://docs.meteor.com/#/full/meteor_publish`.

Full Stack Reactivity

Reactivity is integral to every part of Meteor. On the client side, Meteor has the **Blaze** library, which uses HTML templates and JavaScript helpers to detect changes and render the data in your UI. Whenever there is a change, the helpers re-run themselves and add, delete, and change UI elements, as appropriate, based on the structure found in the templates. These functions that re-run themselves are called **reactive computations**.

On both the client and the server, Meteor also offers reactive computations without having to use a UI. Called the **Tracker** library, these helpers also detect any data changes and re-run themselves accordingly. Because both the client and the server are JavaScript-based, you can use the Tracker library anywhere. This is defined as isomorphic or **full stack reactivity** because you're using the same language (and in some cases the same code!) on both the client and the server.

Re-running functions on data changes has a really amazing benefit for you, the programmer: you get to write code declaratively, and Meteor takes care of the reactive part automatically. Just tell Meteor how you want the data displayed, and Meteor will manage any and all data changes. This declarative style is usually accomplished through the use of **templates**.

Templates work their magic through the use of **view data bindings**. Without getting too deep, a view data binding is a shared piece of data that will be displayed differently if the data changes.

Let's look at a very simple data binding—one for which you don't technically need Meteor—to illustrate the point. Let's perform the following set of steps to understand the concept in detail:

1. In `LendLib.html`, you will see an HTML-based template expression:

```
<div id="categories-container">
     {{> categories}}
</div>
```

2. This expression is a placeholder for an HTML template that is found just below it:

```
<template name="categories">
   <h2 class="title">my stuff</h2>..
```

3. So, `{{> categories}}` is basically saying, "put whatever is in the template `categories` right here." And the HTML template with the matching name is providing that.

If you want to see how data changes will affect the display, change the `h2` tag to an `h4` tag and save the change:

```
<template name="categories">
    <h4 class="title">my stuff</h4>
```

4. You'll see the effect in your browser. (**my stuff** will become itsy bitsy.) That's view data binding at work. Change the `h4` tag back to an `h2` tag and save the change, unless you like the change. No judgement here...okay, maybe a little bit of judgment. It's ugly, and tiny, and hard to read. Seriously, you should change it back before someone sees it and makes fun of you!

Alright, now that we know what a view data binding is, let's see how Meteor uses it.

Inside the `categories` template in `LendLib.html`, you'll find even more templates:

```
<template name="categories">
  <h4 class="title">my stuff</h4>
  <div id="categories" class="btn-group">
    {{#each lists}}
      <div class="category btn btn-primary">
        {{Category}}
      </div>
    {{/each}}
  </div>
 </template>
```

Meteor uses a template language called **Spacebars** to provide instructions inside templates. These instructions are called **expressions**, and they let us do things like add HTML for every record in a collection, insert the values of properties, and control layouts with conditional statements.

The first Spacebars expression is part of a pair and is a `for-each` statement. `{{#each lists}}` tells the interpreter to perform the action below it (in this case, it tells it to make a new `div` element) for each item in the `lists` collection. `lists` is the piece of data, and `{{#each lists}}` is the placeholder.

Now, inside the `{{#each lists}}` expression, there is one more Spacebars expression:

```
{{Category}}
```

Since the expression is found inside the `#each` expression, it is considered a property. That is to say that `{{Category}}` is the same as saying `this.Category`, where `this` is the current item in the `for-each` loop. So, the placeholder is saying, "add the value of the `Category` property for the current record."

Now, if we look in `LendLib.js`, we will see the reactive values (called **reactive contexts**) behind the templates:

```
lists : function () {
  return lists.find(...
```

Here, Meteor is declaring a template helper named `lists`. The helper, `lists`, is found inside the template helpers belonging to `categories`. The `lists` helper happens to be a function that returns all the data in the `lists` collection, which we defined previously. Remember this line?

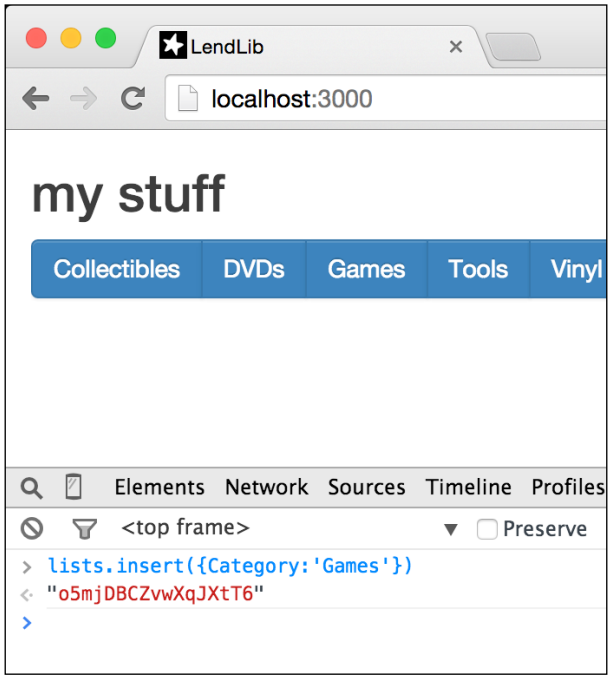
```
lists = new Mongo.Collection("lists");
```

This `lists` collection is returned by the above-mentioned helper. When there is a change to the `lists` collection, the helper gets updated and the template's placeholder is changed as well.

Let's see this in action. On your web page pointing to `http://localhost:3000`, open the browser console and enter the following line:

```
> lists.insert({Category:"Games"});
```

This will update the `lists` data collection. The template will see this change and update the HTML code/placeholder. Each of the placeholders will run one additional time for the new entry in `lists`, and you'll see the following screen:



When the `lists` collection was updated, the `Template.categories.lists` helper detected the change and reran itself (recomputed). This changed the contents of the code meant to be displayed in the `{{> categories}}` placeholder. Since the contents were changed, the affected part of the template was re-run.

Now, take a minute here and think about how little we had to do to get this reactive computation to run: we simply created a template, instructing Blaze how we want the `lists` data collection to be displayed, and we put in a placeholder. This is simple, declarative programming at its finest!

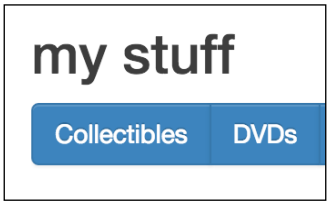
Let's create some templates

We'll now see a real-life example of reactive computations and work on our Lending Library at the same time. Adding categories through the console has been a fun exercise, but it's not a long-term solution. Let's make it so that we can do that on the page instead as follows:

1. Open `LendLib.html` and add a new button just before the `{{#each lists}}` expression:

```
<div id="categories" class="btn-group">
  <div class="category btn btn-primary" id="btnNewCat">
    <span class="glyphicon glyphicon-plus"></span>
  </div>
  {{#each lists}}
```

2. This will add a plus button on the page, as follows:



3. Now, we want to change the button into a text field when we click on it. So let's build that functionality by using the reactive pattern. We will make it based on the value of a variable in the template.
4. Add the following `{{#if...else}}` conditionals around our new button:

```
<div id="categories" class="btn-group">
  {{#if new_cat}}
  {{else}}
    <div class="category btn btn-primary" id="btnNewCat">
      <span class="glyphicon glyphicon-plus"></span>
    </div>
  {{/if}}
  {{#each lists}}
```

5. The first line, `{{#if new_cat}}`, checks to see whether `new_cat` is `true` or `false`. If it's `false`, the `{{else}}` section is triggered, and it means that we haven't yet indicated that we want to add a new category, so we should be displaying the button with the plus sign. In this case, since we haven't defined it yet, `new_cat` will always be `false`, and so the display won't change. Now, let's add the HTML code to display when we want to add a new category:

```
{{#if new_cat}}
  <div class="category form-group" id="newCat">
      <input type="text" id="add-category"
```

```
class="form-control" value="" />
    </div>
{{else}}
  ...
{{/if}}
```

There's the smallest bit of CSS we need to take care of as well. Open `~/Documents/Meteor/LendLib/LendLib.css` and add the following declaration:

```
#newCat {
  max-width: 250px;
}
```

6. Okay, so now we've added an input field, which will show up when `new_cat` is `true`. The input field won't show up unless it is set to `true`; so, for now, it's hidden. So, how do we make `new_cat` equal to `true`?
7. Save your changes if you haven't already done so, and open `LendLib.js`. First, we'll declare a `Session` variable, just below our `Meteor.isClient` check function, at the top of the file:

```
if (Meteor.isClient) {
  // We are declaring the 'adding_category' flag
  Session.set('adding_category', false);
```

8. Now, we'll declare the new template helper `new_cat`, which will be a function returning the value of `adding_category`. We need to place the new helper in the `Template.categories.helpers()` method, just below the declaration for `lists`:

```
Template.categories.helpers({
  lists: function () {
    ...
  },
  new_cat: function(){
    //returns true if adding_category has been assigned
    //a value of true
    return Session.equals('adding_category',true);
  }
});
```

Note the comma (,) on the line above `new_cat`. It's important that you add that comma, or your code will not execute.

Save these changes, and you'll see that nothing has changed. Ta-da! In reality, this is exactly as it should be because we haven't done anything to change the value of `adding_category` yet. Let's do this now:

1. First, we'll declare our `click` event handler, which will change the value in our `Session` variable. To do this, add the following highlighted code just below the `Template.categories.helpers()` block:

```
Template.categories.helpers({
  ...
});
Template.categories.events({
  'click #btnNewCat': function (e, t) {
    Session.set('adding_category', true);
   Tracker.flush();
    focusText(t.find("#add-category"));
  }
});
```

2. Now, let's take a look at the following line of code:

```
Template.categories.events({
```

   This line declares that events will be found in the `categories` template.

3. Now, let's take a look at the next line:

```
'click #btnNewCat': function (e, t) {
```

   This tells us that we're looking for a click event on the HTML element with an `id="btnNewCat"` statement (which we already created in `LendLib.html`).

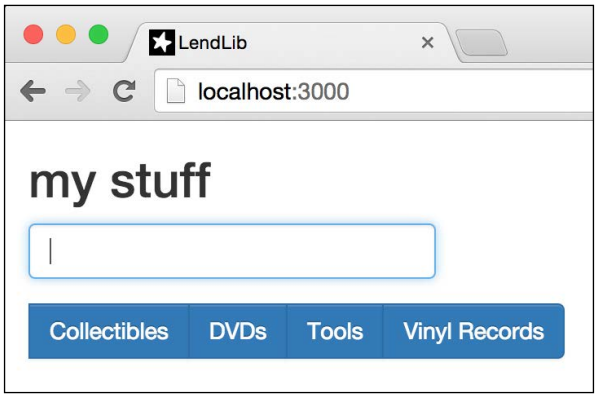
```
Session.set('adding_category', true);
Tracker.flush();
focusText(t.find("#add-category"));
```

4. Next, we set the `Session` variable, `adding_category = true`, flush the DOM (to clear up anything wonky), and then set the focus onto the input box with the `id="add-category"` expression.

5. There is one last thing to do, and that is to quickly add the `focusText().` helper function. To do this, just before the closing tag for the `if (Meteor.isClient)` function, add the following code:

```
/////Generic Helper Functions/////
//this function puts our cursor where it needs to be.
function focusText(i) {
  i.focus();
  i.select();
};
} //<------closing bracket for if(Meteor.isClient){}
```

Now, when you save the changes and click on the plus button, you will see the input box:



Fancy! It's still not useful, but we want to pause for a second and reflect on what just happened; we created a conditional template in the HTML page that will either show an input box or a plus button, depending on the value of a *variable*.

This variable is a reactive variable, called a **reactive context**. This means that if we change the value of the variable (like we do with the `click` event handler), then the view automatically updates because the `new_cat` helpers function (a reactive computation) will re-run. Congratulations, you've just used Meteor's reactive programming model!

To really bring this home, let's add a change to the `lists` collection (which is also a reactive context, remember?) and figure out a way to hide the `input` field when we're done.

First, we need to add a listener for the `keyup` event. Or, to put it another way, we want to listen when the user types something in the box and hits *Enter*. When this happens, we want to add a category based on what the user typed. To do this, let's first declare the event handler. Just after the `click` handler for `#btnNewCat`, let's add another event handler:

```
'click #btnNewCat': function (e, t) {
  ...
},
'keyup #add-category': function (e,t){
  if (e.which === 13)
  {
    var catVal = String(e.target.value || "");
    if (catVal)
```

```
        {
          lists.insert({Category:catVal});
          Session.set('adding_category', false);
        }
      }
    }
```

We add a "," character at the end of the first `click` handler, and then add the `keyup` event handler. Now, let's check each of the lines in the preceding code:

This line checks to see whether we hit the *Enter/Return* key.

```
if (e.which === 13)
```

This line of code checks to see whether the input field has any value in it:

```
var catVal = String(e.target.value || "");
if (catVal)
```

If it does, we want to add an entry to the `lists` collection:

```
lists.insert({Category:catVal});
```

Then, we want to hide the input box, which we can do by simply modifying the value of `adding_category`:

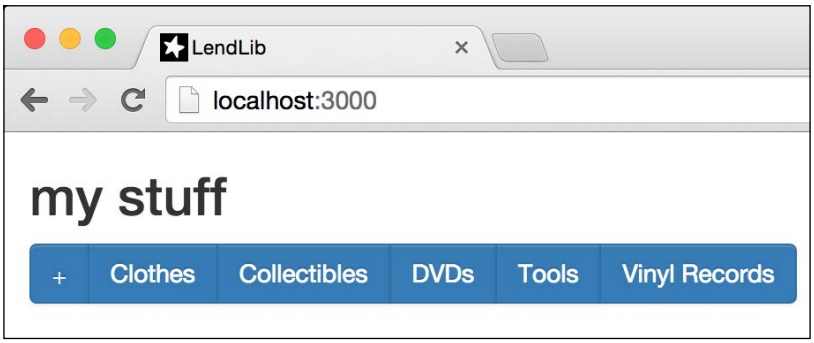
```
Session.set('adding_category', false);
```

There is one more thing to add and then we'll be done. When we click away from the `input` box, we want to hide it and bring back the plus button. We already know how to do this reactively, so let's add a quick function that changes the value of `adding_category`. To do this, add one more comma after the `keyup` event handler and insert the following event handler:

```
'keyup #add-category': function (e,t){
  ...
},
'focusout #add-category': function(e,t){
    Session.set('adding_category',false);
}
```

Save your changes, and let's see this in action! In your web browser on `http://localhost:3000`, click on the plus sign, add the word `Clothes`, and hit *Enter*.

Your screen should now resemble the following screenshot:



Feel free to add more categories if you like. Also, experiment by clicking on the plus button, typing something in, and then clicking away from the input field.

Summary

In this chapter, you learned about the history of web applications and saw how we've moved from a traditional client/server model to a nested MVC design pattern. You learned what smart apps are, and you also saw how Meteor has taken smart apps to the next level with Data On The Wire, Latency Compensation, and Full Stack Reactivity. You saw how Meteor uses templates and helpers to automatically update content, using reactive variables and reactive computations. Lastly, you added more functionality to the Lending Library. You made a button and an input field to add categories, and you did it all using reactive programming rather than directly editing the HTML code. In the next chapter, we'll really get to work and add all kinds of templates and logic, bringing our Lending Library to life!

4
Templates

We've gotten just a taste of templates so far and are now ready to dive in to create a working application using reactive programming. This chapter will take us through the template system in depth and show us how to implement display logic and take care of data flow.

In this chapter, you will complete the following tasks:

- Complete the Lending Library core functionality
- Create multiple templates and template logic
- Add, delete, and update entries in the data model
- See reactivity in action and use it in your application

A new HTML template

We've already created our categories through the use of the `categories` template. Now, we want to take this to the next level and display the actual items that we may want to let people (except STEVE!) borrow so that when we click on a category, we get a list of items.

Let's use that terminology. We need a place to display a list. So, let's modify our `LendLib.html` file at `~/Documents/Meteor/LendLib/` just a bit at the top:

```
<body>
  <div id="lendlib" class="container">
    <div id="categories-container" class="container">
      {{> categories}}
    </div>
    <div id="list">
      {{> list}}
```

```
      </div>
    </div>
</body>
```

We did two things by adding this code:

1. We wrapped the `div` element with `id="categories-container"` inside the `div` element called `lendlib`, and we moved `class="container"` to the new `div` element. This is for stylistic purposes so that our list will more or less line up with the `categories` template.
2. We added a `div` element with `id="list"` just below it and also added a call to a new template: `{{> list}}`. This is our template/placeholder for the list of items, which we'll see shortly in the sections that follow.

And, there you have it. We've created a very easy-to-maintain structure with definite boundaries in the document. We know where our categories are going to go, and we know where our list of items is going to go.

Now, let's see about the `list` template itself. Not so simple, but still not bad. At the very end of `LendLib.html`, below the closing `</template>` tag for our `categories` template, place the following code:

```
<template name="list">
  <div id="lending_list" class="list-group">
    {{#each items}}
      <div class="lending_item list-group-item">
        <button type="button"
          class="close delete_item" id="{{Name}}">x</button>
        {{Name}}
        {{#if lendee_editing}}
          <input type="text" id="edit_lendee"
          class="span-2" value=""/>
        {{else}}
          <div class="lendee label {{LendClass}}">
            {{Lendee}}
          </div>
        {{/if}}
      </div>
    {{/each}}
    {{#if list_selected}}
      <div class="list-group-item" id="btnAddItem">
        <span class="glyphicon glyphicon-plus"></span>
        {{#if list_adding}}
          <input class="span-4" id="item_to_add"
           size="32" type="text">
```

```
        {{/if}}
      </div>
    {{/if}}
  </div>
</template>
```

Let's go through this step by step so that we understand what each line does.

```
<template name="list">
  <div id="lending_list" class="list-group">
    {{#each items}}
```

Here, we declare the `<template>` tag with the name `list` to match the call to the `list` template that we made in the body. We create a Bootstrap `list-group` and give it an `id` value so that we can refer to it later if need be.

Then, we start a templated `each` statement. This time, we're going to iterate through `items`. We haven't created the Meteor `items` template just yet, but we'll get there soon enough.

```
      <div class="lending_item list-group-item">
        <button type="button"
          class="close delete_item">x</button>
        {{Name}}
```

Now, under the `each` statement, we create a `div` element and give it two class names. The `lending_item` class name is added so that we can refer to it in our reactive templates. The `list-group-item` class name is for Bootstrap so that it will display all nice and pretty.

Next, we create a `button` that we can use should we choose to delete the item. There are also two class names on this one. The `close` class is added for Bootstrap, and `delete_item` is added so that we can refer to it in our helpers when the time comes.

Now, just below that we have another template placeholder for `{{Name}}`. This is so that we can use the title of the item (for example, on a DVD item the title could be **Mission Impossible**) inside our display element. We'll see this in action very soon.

Now, we begin a series of conditional statements. The first conditional statement lets us edit the **lendee** (the person who borrowed the item):

```
        {{#if lendee_editing}}
          <input type="text" id="edit_lendee"
          class="span-2" value=""/>
        {{else}}
          <div class="lendee label {{LendClass}}">
            {{Lendee}}
```

```
            </div>
          {{/if}}
        </div>
      {{/each}}
```

We are first using an `if` statement to see whether our *current mode* for this `item` is `lendee_editing`. That is to say, if we wanted to edit the lendee, we would be in "lendee editing" mode, and therefore, (in our JavaScript file) `Template.list.lendee_editing` would return as `true`. If this is the case, we need a textbox; therefore, we included the `<input>` element with its associated `id`.

Alternately—and this is the default—we just want to display who the lendee is, if there is one. If there isn't one, we'll want to maybe change the color or something, but we still want to display it. So, we create a bootstrap-styled `label` in the form of a `<div>` element.

At the end of the classes' declarations, we see a template variable, `...{{LendClass}}"`. This class addition is stylistic. It will tell our CSS template whether to show it as **free** (someone can borrow it) or **lent out**. If it's green, it's free; however, if it's red, someone has borrowed it. The CSS class name used to represent the color will be determined in `LendLib.js`, by the `item.LendClass` property, which we will create shortly.

Then, we have the value inside the div, `{{Lendee}}`. This will also be a property in `LendLib.js`, as the `item.Lendee` property, and it will display either the name of the lendee or the word **free**, if no one has borrowed it.

We then have the ending `</div>` tag, and our `each` comes to an end with `{{/each}}`.

Now, we have our second `if` statement, and this one is actually a nested `if`. This one is outside the `each` statement, so it's not specific to items. This `if` statement displays either a plus (**+**) sign or a textbox in the form of an `<input>` element so that we can add items to our list:

```
      {{#if list_selected}}
        <div class="list-group-item" id="btnAddItem">
          <span class="glyphicon glyphicon-plus"></span>
          {{#if list_adding}}
            <input class="span-4" id="item_to_add"
             size="32" type="text">
          {{/if}}
        </div>
      {{/if}}
    </div>
  </template>
```

So, we see our first `if`, which is conditioned on whether we're displaying any list items. If we are, this means that we have selected a list, or that we are in the `list_selected` mode. Keeping track of this is the Blaze rendering engine's job, so `Template.list.list_selected` is found inside `LendLib.js`.

We then create a plus sign using the Bootstrap `glyphicon-plus` class.

Next is our nested (second) `if`. This `if` is checking to see whether we are making additions to the list of items. If we are, we are in the `list_adding` mode, so we'll display a textbox in the form of an `<input>` element. If not, we'll just leave the plus sign all by its lonesome.

Finally, we end our nested `{{/if}}`, our `</div>`, our parent `{{/if}}`, our `</ul>`, and our `</template>`.

Gluing it all together

In an MVC or MV* model, the controller (in this case, the Blaze rendering engine) is sometimes called the **glue**. That's because it "glues" together all the view items, such as buttons or textboxes, to the model. Blaze does this reactively, using reactive contexts and reactive computations, vastly simplifying the "gluing" process for you, the developer.

Pretty fancy explanation, eh? Well, you try to come up with a better explanation for what it does. It really does fill in the cracks and keeps the model and the view together. Someone else invented the term, not us; so, let's continue without the Judgy McJudgerson viewpoint, mmmkay?

In this section, we'll go through all the changes that need to happen inside `~/Documents/Meteor/LendLib/LendLib.js` step by step in order to glue the template and the data model together.

Displaying items

In our data collection that we created in *Chapter 2*, *Reactive Programming…It's Alive!*, we added some sample `items` when we created a couple of `lists`. We did so, if you recall, using the browser console, as follows:

```
> lists.insert({Category:"DVDs", items: [{Name:"Mission
  Impossible",Owner:"me",LentTo:"Alice"}]});
```

You'll notice that we have a hierarchy here. Every `list` in the `lists` collection has an `items` object, which is an array:

```
Items: [...]
```

We need to represent this `items` array to our HTML template, but we need a couple of extra properties so that the view knows what to do with it. Specifically, we need to do the following things:

- Return the name of the lendee or return "free" if there is no lendee (`item.Lendee`)
- Return the CSS class (red or green), depending on whether the item is lent out (`item.LendClass`)

So, we're going to get the `items` collection from the currently selected list, add the `Lendee` and `LendClass` properties, and make the template available as follows:

1. Open the `LendLib.js` file under `~/Documents/Meteor/LendLib/`.
2. Immediately after the closing `"}"` curly bracket of the `focusText()` function, add the following highlighted code:

```
function focusText(i) {
  ...
};
Template.list.helpers({
  items: function () {
    if (Session.equals('current_list', null))
      return null;
    else {
      var cats = lists.findOne({
        _id: Session.get('current_list')
      });
      if (cats && cats.items) {
        for (var i = 0; i < cats.items.length; i++) {
          var itm = cats.items[i];
          itm.Lendee = itm.LentTo ? itm.LentTo :
          "free";
          itm.LendClass = itm.LentTo ?
            "label-danger" : "label-success";
        }
        return cats.items;
      };
    };
  }
});
```

Now, we'll walk through this step by step. Consider the following code snippet:

```
items: function () {
  if (Session.equals('current_list',null))
    return null;
```

Here, we're declaring the `Template.list.items` helper and checking to see whether a `list` is selected. If a `list` is selected, the session variable, `current_list`, will have a value in it. If it's `null`, there's no reason to return anything; so, we'll just return `null`.

This is the glue at work. It is reactively reading the contents of a given category and incorporating it into the current state of the UI, depending on whether the user has selected a list.

If something is selected, we need to first find the category. We'll call it `cats` because it's shorter even though it's not strictly speaking the best naming convention. But what do we care? We're doing this for fun, and `cats` are awesome!

```
else {
  var cats = lists.findOne({_id:Session.get('current_list')});
```

We are using the MongoDB command, `findOne()`, and passing the `current_list` session parameter as `_id` in the selector/query. If something is selected, we will get a single category/list back. Let's check to make sure that we do and we also get `items`.

If nothing returns, or there are no `items` in the category, we don't really need to figure out the `Lendee` or the `LendClass`, do we? So, let's create an `if` statement and a `for` statement inside the `if` that will only get executed if we have something worth iterating over:

```
if (cats && cats.items) {
  for (var i = 0; i < cats.items.length; i++) {
    var itm = cats.items[i];
    itm.Lendee = itm.LentTo ? itm.LentTo :
    "free";
    itm.LendClass = itm.LentTo ?
      "label-danger" : "label-success";
  }
  return cats.items;
};
```

First, we check to see whether `cats` and `cats.items` are not `undefined/null`.

Next, we iterate over all the values in `items` (`items` is an array, if you recall). To make it easier, we declare the variable `itm = cats.item[i]`.

Then, we add the `Lendee` property, checking to see whether the item is lent to anyone with the `LentTo` property. If it isn't (if `LentTo` doesn't exist), we'll assign the `"free"` string instead.

Likewise, if `LentTo` exists, we declare the red bootstrap label class, `label-important`, as the `LendClass`. If the item is not lent out, we'll use the green bootstrap class, `label-success`, instead.

Finally, with our new `Lendee` and `LendClass` properties assigned, we'll return `cats.items`. We didn't save these properties to our data collection. This is because they're *not* part of the data collection. They're used by the Blaze to modify the look and feel of our UI, so we'll only need to make them available via the template helper.

Additional view states

We now need to declare the templates for all the different view states. That is, we need to add properties to our templates that will let us know what we're looking at, what we're editing, and what should be `hidden/visible`. Specifically, we need to access the state value in four situations:

- What list are we looking at? (`list_status`)
- Are we looking at a list? (`list_selected`)
- Are we adding an item to a list? (`list_adding`)
- Are we updating the lendee? (`lendee_editing`)

The `list_status` property will be attached to each category. If the category is selected (if `current_id` is set to that category's `id`), we will change the color from `btn-primary` (blue) to `btn-info` (light blue). Since it's a property on the category, we need to add it to the `Template.categories.helpers()` section. To do this, add the following code just below the `new_cat` template helper:

```
new_cat: function () {
  ...
},
list_status: function () {
  if (Session.equals('current_list', this._id))
    return " btn-info";
  else
    return " btn-primary";
}
```

The rest of the properties belong to `lists` and list `items`, so we will add the following to the `Template.lists.helpers()` method, just below the `items` template helper:

```
items: function () {
  ...
},
list_selected: function () {
  return ((Session.get('current_list') != null) && (!Session.equals(
    'current_list', null)));
},
list_adding: function () {
  return (Session.equals('list_adding', true));
},
lendee_editing: function () {
  return (Session.equals('lendee_input', this.Name));
}
```

Let's go over each of these template functions.

```
list_selected: function () {
  return ((Session.get('current_list') != null) && (
    !Session.equals('current_list', null)));
},
```

The session variable, `current_list`, can be either `undefined` or `null`. If it's `undefined`, then `Session.equals('current_list'),null)` will return `true`. So, we need to check both cases unfortunately.

```
list_status: function () {
  if (Session.equals('current_list', this._id))
    return " btn-info";
  else
    return " btn-primary";
}
```

Now, `list_status` is used to tell the category button whether it should show as selected. The easiest way to do this is via a CSS class. Bootstrap uses `btn-primary` to show white-on-blue text but that's the default look and feel. For the selected category, we will change the background color to a lighter blue, using the `btn-info` class.

In other words, for the `current_list`, we'll return `btn-info`. For all the other `lists/categories`, we'll return `"btn-inverse"` to change the CSS style.

You may be wondering about `this._id`. Now, `this`, in this instance, refers to the MongoDB record (technically the **document cursor**), and `._id` is the unique MongoDB identifier for that record. This is called the **context**, and in this case, when using the HTML template, the context is implied to be the list/category element from where the template was called.

```
list_adding: function () {
  return (Session.equals('list_adding', true));
},
```

This one's really straightforward. If the `Session` variable, `list_adding`, is `true`, we'll add it to the list. If it isn't, we won't.

```
lendee_editing: function () {
  return (Session.equals('lendee_input', this.Name));
}
```

To check and see whether we should be in the "lendee editing" mode, we'll check the `Session` variable, `lendee_input`, to see whether it has a value and if that value is the `Name` of the item on which we just clicked. Once again, this is the implied context. This time, it's not the list, it's the item. How do we know this? Because of where this function is called from. Remember the HTML code?

```
<div class="lending_item list-group-item">
    <button type="button"
    class="close delete_item">x</button>
    {{Name}}
    {{#if lendee_editing}}
```

Notice how we use `lendee_editing` in the `if` statement, right after we use `{{Name}}`. This shows us the context. The `this.Name` statement in `LendLib.js` has the same context as `{{Name}}` in `LendLib.html`. In other words, `this.Name` is referencing the same property as `{{Name}}`.

While we're here in the HTML templates, there's one change that we need to make to the HTML `categories` template. We waited until now so that the change would make sense. When you make the following code change, you'll see how the `{{list_status}}` and `{{_id}}` templates are used, and why the context for `this._id` suddenly makes sense.

Locate the following lines in `LendLib.html` (they should begin from the 27th line or so):

```
{{#each lists}}
  <div class="category btn btn-primary">
    {{Category}}
```

```
  </div>
{{/each}}
```

Change them to look like the following code snippet:

```
{{#each lists}}
  <div class="category btn {{list_status}}" id="{{_id}}">
    {{Category}}
  </div>

{{/each}}
```

Adding events

We are now going to hook up all the events. Instead of doing this in the HTML like a hobo, we'll do it using the template helpers, like a boss.

The first one takes place in the `Template.categories.events` declaration because we need to add the event that will change the `Session` variable, `current_list`. If you recall, `current_list` helps us to know whether we have a selected list (`list_selected`), and if so, which list is selected (`list_status`).

In `LendLib.js`, between the event declaration for `focusout #add-category` and the end `});` bracket for the `Template.categories.events` function, add the following code:

```
'focusout #add-category': function (e, t) {
  ...
},
'click .category': selectCategory
```

Don't forget the comma (`,`) right after the `focusout` handler block.

This adds a `click` event handler for every button with the `category` CSS class and calls the `selectCategory()` function. We'll declare this function right now.

Just after the `focusText()` function and just before the `Template.list.helpers` declaration, add the following code:

```
function selectCategory(e,t){
  Session.set('current_list',this._id);
}
Template.list.helpers({...
```

Yes, you could have put this anywhere. And yes, you could have just put a generic function in line with the `click` event declaration. So, why put it here? Because it makes our code more readable, and we need a section for all the add/delete/update calls we'll need anyway, so it fits right here.

And yes, it's very simple. It just updates the `Session` variable, `current_list` with `this._id`. The context of `this` here is the category/list, and therefore, `_id` is the MongoDB-generated ID for the record.

Alright, now that we have all the categories' events taken care of, let's work on the `items` events. At the very end of the `if (Meteor.is_client) {...` code block, just inside the closing bracket "}", add the following code:

```
Template.list.events({
  'click #btnAddItem': function (e,t){
    Session.set('list_adding',true);
    Tracker.flush();
    focusText(t.find("#item_to_add"));
  },
  'keyup #item_to_add': function (e,t){
    if (e.which === 13)
    {
      addItem(Session.get('current_list'),e.target.value);
      Session.set('list_adding',false);
    }
  },
  'focusout #item_to_add': function(e,t){
    Session.set('list_adding',false);
  },
  'click .delete_item': function(e,t){
    removeItem(Session.get('current_list'),e.target.id);
  },
  'click .lendee' : function(e,t){
    Session.set('lendee_input',this.Name);
    Tracker.flush();
    focusText(t.find("#edit_lendee"),this.LentTo);
  },
  'keyup #edit_lendee': function (e,t){
    if (e.which === 13)
    {
      updateLendee(Session.get('current_list'),this.Name,
      e.target.value);
      Session.set('lendee_input',null);
    }
```

```
      if (e.which === 27)
      {
        Session.set('lendee_input',null);
      }
    }
  });

  }//<----this is the closing bracket for if(Meteor.is_client) ----
```

Six events! It looks more monstrous than it is. As usual, let's break it down, step by step.

```
Template.list.events({
  'click #btnAddItem': function (e,t){
    Session.set('list_adding',true);
    Tracker.flush();
    focusText(t.find("#item_to_add"));
  },
```

We declare `Template.lists.events` and enumerate our events. The first one is to add an item. The button to add an item is, funnily enough, named `btnAddItem`, so all we have to do is add the declaration and then write our function.

We set `list_adding` to `true`. Because we use `Session.set()`, the change cascades through our templates. This is a reaction, or reactive programming, in action. We also call `Tracker.flush()` to ensure that the UI is cleaned up, and then, as a courtesy to our user, we will focus on the textbox (named `item_to_add`) so that our beloved user can just start typing.

```
  'keyup #item_to_add': function (e,t){
    if (e.which === 13)
    {
      addItem(Session.get('current_list'),e.target.value);
      Session.set('list_adding',false);
    }
  },
```

You can learn more about what `Tracker.flush()` does in the Meteor documentation at `http://docs.meteor.com/#/full/tracker_flush`.

The next event is based on the `keyup` event for our `item_to_add` textbox (`e.which === 13`). If we hit *Enter* or *Return*, we'll call the `addItem()` function to update the data model, and then we'll hide the textbox. How do we do this? Set `list_adding = false` of course. Again, doing it through `Session.set()` will cascade the change through our templates.

There is something else that you may have missed. Remember when we manually added a category/list in *Chapter 2*, *Reactive Programming… It's Alive!* The change was instantly reflected in the HTML DOM. The same thing is true here. When `addItem()` updates the data model, that change will trigger a template refresh for `Template.list.items`.

```
'focusout #item_to_add': function(e,t){
  Session.set('list_adding',false);
},
```

This event handler creates a situation so that, if we were to change our minds about adding an `item`, all we have to do is click away from it. If the `item_to_add` textbox triggers the `focusout` event, we'll set the `Session` variable, `list_adding`, to `false`, and the template will cascade.

```
'click .delete_item': function(e,t){
  removeItem(Session.get('current_list'),e.target.id);
},
```

Remember, when we created the little **x** button in our HTML `lists` template?

That button belongs to the CSS class, `delete_item`; when the user clicks on it, we will call the `removeItem()` function and pass `current_list` and `id` from the HTML element that was clicked, which happens to be the `Name` item (we added `id="{{Name}}"` on the 29th line of `LendingLib.html`).

```
'click .lendee' : function(e,t){
  Session.set('lendee_input',this.Name);
  Tracker.flush();
  focusText(t.find("#edit_lendee"),this.LentTo);
},
```

We now focus on the `lendee` section/button. If you recall, we set up a `<div>` element with the `lendee` CSS class in `LendingLib.html`. We are now declaring that whenever one of those `<div>` elements is clicked, we'll perform actions very similar to those we used when we wanted to add an `item` to a `list`:

1. Set the `Session` variable that controls textbox visibility to `true` (`lendee_input`).
2. Refresh the UI (`Tracker.flush()`).
3. Set the focus on the textbox (`focusText()`).

Let's consider one more event handler:

```
  'keyup #edit_lendee': function (e,t){
    if (e.which === 13)
    {
      updateLendee(Session.get('current_list'),this.Name,
      e.target.value);
      Session.set('lendee_input',null);
    }
    if (e.which === 27)
    {
      Session.set('lendee_input',null);
    }
  }
});
```

The textbox with `id="edit_lendee"` has two `keyup` conditions attached to it.

If we hit *Enter* or *Return* `(e.which === 13)`, we'll update the `LentTo` property on the data model using `updateLendee()`. We'll then hide the textbox by setting `lendee_input` to `null`. Setting `lendee_input` to `null` is a change to a reactive variable, which triggers the reactive computations and updates the UI.

If we instead decide that we don't like the changes, we can hit the *Esc* key (`e.which === 27`); in this case, we'll set `lendee_input` to `null` and let the reactive programming hide the textbox.

Model updates

We have two things left to do. We need to polish up our app with a bit of CSS (not just yet though), and we need to create the `addItem()`, `removeItem()`, and `updateLendee()` functions to which we just referred in the `events` section.

So, let's get to work! In `LendingLib.js`, just below the `selectCategory()` function, let's create the `addItem()` function:

```
function selectCategory(e,t){
  ...
};
function addItem(list_id,item_name){
  if (!item_name&&!list_id) return;
  lists.update({_id:list_id},
               {$addToSet:{items:{Name:item_name}}});
};
```

The first thing that we do is check to make sure that we have values for `item_name` and `list_id`. If we don't, we'll just `return`.

Now, we'll call the MongoDB `update()` function on the `lists` collection. `{_id:list_id}` is the selector. We tell MongoDB to find us the record with `_id = list_id`. We then tell MongoDB what kind of update we're going to perform. In this case, we'll use `$addToSet`, which will append to an array. And, what are we going to append?

`{items:...` indicates that we're updating the `items[]` array. `{Name:item_name}` is what we're adding, and it's the `Name` property. This is equivalent to saying `item.Name = item_name`.

Once we add this, as you might have guessed by now, the templates will automatically update because `lists` is a reactive variable. Stop for a second and think about this. In six lines of code, we performed an update query *and* propagated the change to our UI. Six lines! Pretty magical, isn't it?

Let's handle `removeItem()` next:

```
function addItem(list_id,item_name){
  ...
};
function removeItem(list_id,item_name){
  if (!item_name&&!list_id)
    return;
  lists.update({_id:list_id},
               {$pull:{items:{Name:item_name}}});
};
```

Wow, this looks really similar to the `addItem()` function. We are, in fact, using the same `update()` function, just with a different action this time. This time, we'll use `$pull`, which pulls an element out of the `items[]` array, where `Name == item_name`. Once again, six lines are all we need. Once we update the `lists` collection, the UI will update reactively.

Now, we tackle `updateLendee()`, which is a little more complex, but only because we are updating a property in an array rather than a direct property of an item. It's not too bad though. The only real difficulty is in finding the right item in the array. Let's begin by adding the following function just below the `removeItem()` function:

```
function removeItem(list_id,item_name){
  ...
};
function updateLendee(list_id,item_name,lendee_name){
  var l = lists.findOne({"_id":list_id ,
                         "items.Name":item_name});
  if (l&&l.items)
  {
    for (var i = 0; i<l.items.length; i++)
    {
      if (l.items[i].Name === item_name)
      {
        var updateItem = {};
        updateItem['items.'+i+'.LentTo'] = lendee_name;
        lists.update({'_id':list_id},{$set:updateItem});
        break;
      }
    }
  }
};
```

We get the `list_id`, the `item_name`, and the `lendee_name`, so we can identify the right record (we use `list_id` and `item_name` for this) and then update the `LentTo` property with the value in `lendee_name`.

To save some typing, we declare the `l` variable using the MongoDB `findOne()` function. This takes a two-part selector statement: `_id:list_id` and `"items.Name":item_name`. This selector is basically saying, *find me ONE record where the _id == list_id and the items[] array has a record where Name == item_name*.

If `l` has a value and it also has `items`, we'll go into our `for` loop. Here, we check specifically for whichever array element had `Name == item_name`. If we find one, we'll first create the change we want as an object. If, for example, the item we want to change is found in the first element in the `items` array, our `updateItem` object would look like this:

```
{ 'items.0.LentTo' : lendee_name }
```

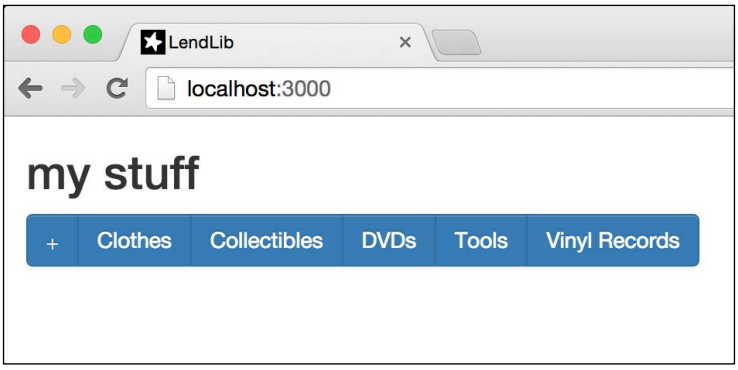
With the object created, we call the `lists.update()` method, passing the `updateItem` in with the `$set` command like this:

```
lists.update({...},{$set : updateItem});
```

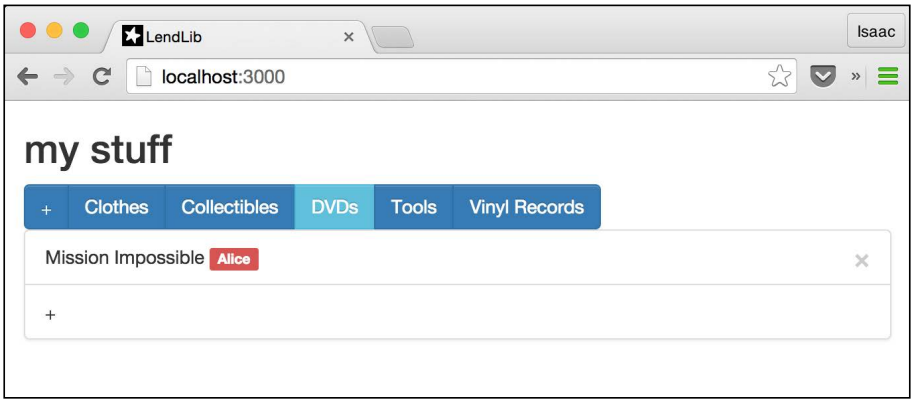
This `update()` method will update the `lists` collection. As you recall, the `lists` collection is a reactive context, and therefore, the UI will update with the changes.

Style updates

Right now, we have a working application! We haven't polished up our CSS yet, but let's go ahead and do that real quickly. Make sure your application is running (run the `meteor` command in a terminal window) and navigate to `http://localhost:3000`:



Click on **DVDs**, and you should see the one entry for **Mission Impossible**:



It's not all that bad besides some width/height issues and maybe some issues with the font sizing, but that's because Bootstrap gives us a lot of help. Let's start by slightly increasing the font size of our items. To do this, open `LendLib.css` and add the following declaration:

```
.lending_item {
  font-size:1.8rem;
}
```

Next, our `btnAddItem` button is a little too "Bruce Banner" for our liking. Let's Hulk it up a bit by making it big and green. To do this, add the following declaration to the bottom of LendLib.css:

```
#btnAddItem span{
  color:#5cb85c;
  font-size:4rem;
}
```

Finally, we will add a small creature comfort. Open `LendLib.js` and modify the following `focusText()` function from this:

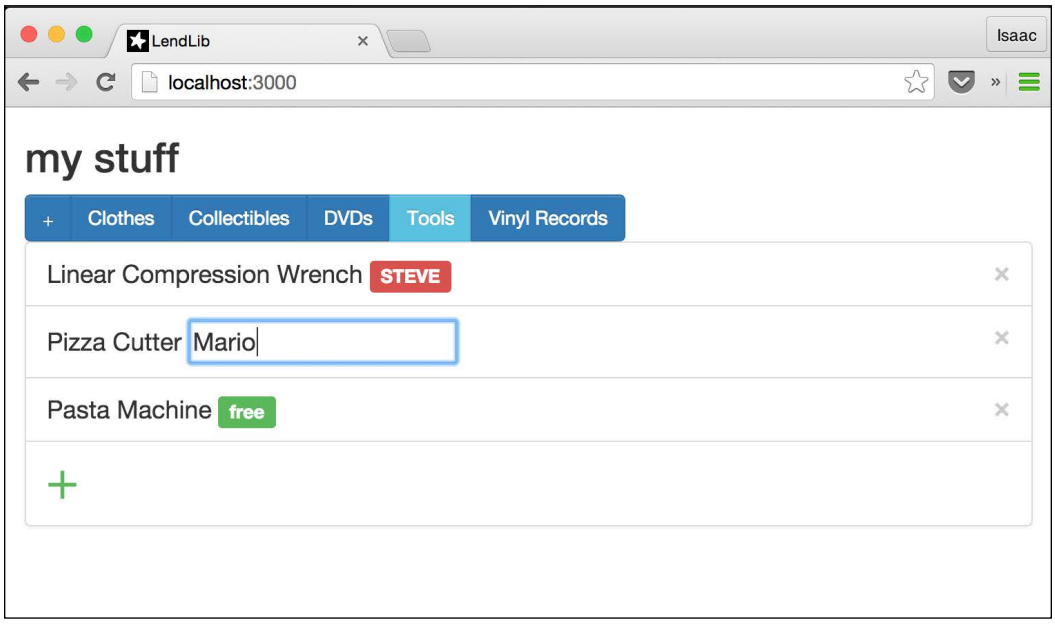
```
//this function puts our cursor where it needs to be.
function focusText(i) {
  i.focus();
  i.select();
};
```

to this:

```
//this function puts our cursor where it needs to be.
function focusText(i,val) {
  i.focus();
  i.value = val ? val : "";
  i.select();
};
```

This change just makes it so that when we go to edit something with a value already in it (such as the lendee), the value will transfer to the textbox. This makes it easier for the user to see who the current lendee is. The `val ? val : ""` conditional statement is necessary, because if `val` isn't passed or is null, **undefined** gets put into the textbox.

Save all your changes and let's preview what our app looks like:



Everything seems to be in order. Play around with your app for a bit—add some new categories, lend some stuff out to your friends (except STEVE!) and, once you're done, move on to the next chapter.

Summary

In this chapter, you completed the templates, events, and data model sections for your Lending Library application. You created statements to add, delete, and update your records and implemented UI state changes. You saw firsthand how reactive programming works and gained a solid understanding of context. You are now equipped to create an application from scratch, using the core functionality of Meteor to develop an app quickly and with robust functionality.

In the next chapter, we'll dive ever more deeply into Meteor's data caching and synching methodology and harden your application.

5
Data – Meteor Style!

We've nearly completed the **Lending Library** application, without having to worry much about how our data is being stored. Meteor's data caching and synching methodology is intentionally built to make that part of building the application as simple as possible so that, instead of spending a lot of time messing around with database connections, queries, and caching, you can concentrate on writing a great program.

However, we do want to go over the methodology, and we'll have to make sure we have a solid understanding of how Meteor handles data so that we can perform some common optimizations and build our applications even more quickly.

In this chapter, you will learn about the following topics:

- MongoDB and document-oriented storage
- Broadcasting changes—how Meteor makes your web application reactive
- Configuring publishers—how to streamline and protect your data

Document-oriented storage

Meteor uses MongoDB on the server side, and a version of MongoDB, **Minimongo**, on the client side. It's capable of using any other NoSQL/document-oriented database, but MongoDB comes by default with the Meteor installation. This feature makes your programs much simpler and easier to write and works really well for quick, lightweight data storage.

Why not use a relational database?

Traditionally, most data is stored using a relational model. The relational model, with all of its associated rules, relations, logic, and syntax, is an integral and invaluable part of modern computing. The rigid structure of a relational database, with exact requirements for each record, relation, and association, provides us with quick searches, scalability, and the potential for deep analytics.

This type of exactness, however, isn't always necessary. In the case of our Lending Library app, for example, a full-fledged relational database would be overkill. In fact, in some cases, it's more effective to have a flexible data storage system—one that you can extend quickly and without significant recoding.

For example, if you wanted to add a new property to your `list` object, it would be much simpler to just add the new property and let the database worry about it rather than having to refactor your database code, add a new column, update all of your SQL statements and triggers, and make sure that all previous records have the new property.

This is where document-oriented storage comes into play. In a document-oriented storage system, your data store is made up of a bunch of key-value-paired documents. So, what's the structure of that document? The data store doesn't really care. It could contain pretty much anything as long as each record has a unique key so that it can be retrieved.

So, in one document entry, you could have a very simple document—maybe a key-value pair:

```
{name:phone_number}
```

And then, in another document entry (in the same data store), you could have a complex object with nested arrays, nested objects, and so on:

```
{ people: [
  {firstname:"STEVE", lastname:"Scuba", phones :[
    {type:cell, number:8888675309},
    {type:home, number:8005322002}]
  },
  {firstname:...
    ...
  }]
}
```

Heck, it could be the unabridged works of William Shakespeare. It doesn't really matter. As long as the data store can take the document and assign a unique key to it, it can be stored.

As you may have guessed, the lack of structure *can* make querying, sorting, and manipulating documents less efficient. But that's okay because our primary concern is with ease of coding and development speed, not efficiency.

In addition, since our application only has a few core functions, we can quickly identify what queries we'll be using most often and optimize our document schema around that. This makes a document-oriented database actually perform *better* in some cases than a traditional relational database.

There are some pretty sophisticated document-oriented storage solutions out there that some would argue are as efficient as, or more efficient than, a standard relational database; but that discussion is beyond the scope of this book.

Given the flexible nature of a document-oriented storage system, it is perfect for making quick changes, and the foundation libraries that Meteor provides make it so that we don't have to worry about the connection or the structure. All we have to do is have a high-level understanding of how to retrieve, add, and modify the documents, and we can leave the rest to Meteor.

MongoDB

MongoDB—a play on the word "humongous"—is an open source NoSQL (Not Only SQL) database. It provides sophisticated features such as indexing, linking, atomic operations, and so on, but at its heart it is a document-oriented storage solution.

To learn more about MongoDB, visit its official site at `http://www.mongodb.org`.

Using simple commands, we can see what records (what documents) are available, convert the records into JavaScript objects, and then save the changed objects. Think of MongoDB records in the same way you would think about an actual text document:

1. Locate the document and open it for editing: (Meteor equivalent: `lists.find()`).
2. Make changes to the document (Meteor equivalent: `lists.update({})`).
3. Save the document (automatically done with the `.update()` function).

It's not as simple as that, and there's a lot of syntax that you'll need to learn if you want to consider yourself proficient in MongoDB, but you can clearly see the simple, clean document-oriented approach: find/create a record, make changes, and save/update the record to the data store.

We need to discuss one final concept to help you better understand how MongoDB works. MongoDB is called a database, but it's easier to conceptualize as a collection of documents. The collection is indexed and quickly accessible, but it's still a collection rather than a group of relational tables/entities. In just the same way you'd think about a folder on your hard drive where you keep all of your text documents, think about MongoDB as a collection of documents, all of which are accessible and can be opened, changed, and saved.

Using direct commands

To gain a better understanding of how MongoDB works, let's have some fun in the command line. Perform the following set of steps:

1. First, make sure that your application is up and running (open a terminal window, `cd` to the `~/Documents/Meteor/LendLib` directory, and execute the `meteor` command).
2. Next, open your browser and point it to `http://localhost:3000`.
3. Now, you will need to open an *additional* terminal window, `cd` to the `~/Documents/Meteor/LendLib` directory, and run the following command:

   ```
   meteor mongo
   ```

   You should see a message similar to the following screenshot:

You have now connected to the running MongoDB database for your Lending Library application. Let's poke around with a couple of commands.

1. First, let's bring up the help screen. Enter the following command and hit *Enter*:

   ```
   > help
   ```

   We'll get a list of commands and a brief explanation of what each one does. One in particular will show us even *more* commands that we can use; for example, `db.help()` will give us a list of JavaScript database-related commands.

2. Type the following into your terminal window and press *Enter*:

   ```
   > db.help()
   ```

Don't get overwhelmed by the number of possible commands. You don't need to know all of these unless you'd like to become an expert in MongoDB. You only need to know a few but having a look around never hurt anybody, so let's proceed.

As mentioned previously, documents are stored in MongoDB in a logical grouping called a collection. We can see this firsthand and take a look at our `lists` collection directly in the terminal window. Run the following set of commands to see their distinct output:

1. To see a list of all available collections, enter the following command:

   ```
   > db.getCollectionNames()
   ```

   In the response, you will find the name of your Lending Library collection—`lists`.

2. Let's go ahead and take a look at the `lists` collection. To do this, enter the following command:

   ```
   > db.getCollection('lists')
   ```

   Well, that wasn't very exciting. All we got back was `meteor.lists`.
   We want to be able to perform some queries on that collection.
   So, this time, let's assign the collection to a variable:

   ```
   > myLists = db.getCollection('lists')
   ```

   It appears that we have the same result as last time, but we have a lot more than that. We've now assigned the `lists` collection to the `myLists` variable Therefore, we can run commands in the terminal window just like we would inside our Meteor code.

3. Let's get the `Clothes` list, which probably doesn't have any items in it but still exists. To do this, enter the following command:

   ```
   > Clothes = myLists.findOne({Category:"Clothes"})
   ```

   This will return some very basic JSON. You will notice an `_id` key-value with a long number next to it, similar to this:

   ```
   "_id" : "vqWrHqMTBdYjggsco"
   ```

We didn't add `_id`; MongoDB created it for us. It's a unique key so that, if we know it, we can make changes to a document without disturbing any of the other documents. We actually use this inside our Lending Library application in multiple locations.

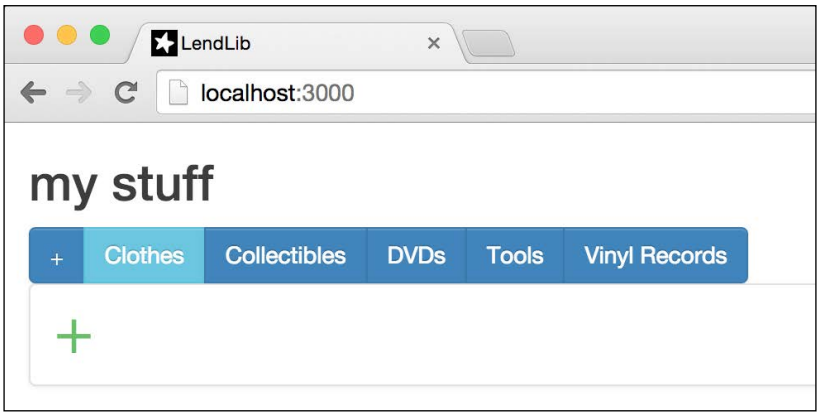
If you look inside the `LendLib.js` file by navigating to `~/Documents/Meteor/LendLib/`, you'll see the following function for adding an item to a list:

```
function addItem(list_id,item_name){
  if (!item_name&&!list_id)
    return;
  lists.update({_id:list_id}, {
    $addToSet:{items:{Name:item_name}}});
};
```

Note that when we call the `lists.update()` function, we identify which document we're going to update by the `_id`. This ensures that we're not updating multiple documents accidentally. If, for example, you were to give two `lists` the same `Category` name (for example, `DVDs`), and used `Category` as the selector (`{Category:"DVDs"}`), you would be taking action on both `lists`. If, instead, you use the `_id`, it will only update the unique document with the matching value of `_id`.

Getting back to the terminal, we now have the `myLists` variable assigned to our `lists` collection, and we've assigned the `Clothes` variable to the document in our `lists` collection that represents the `Clothes` list.

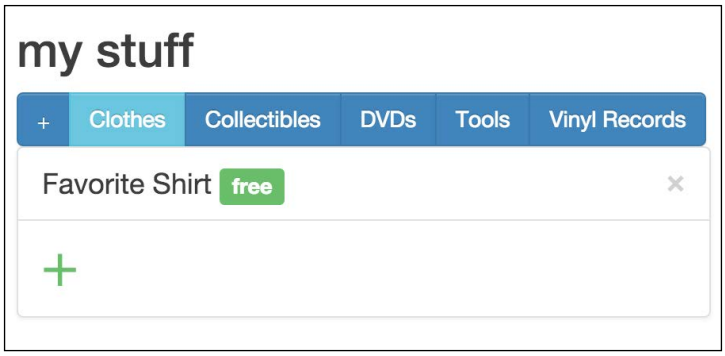
Take note of what the **Clothes** list currently looks like in our browser:



Let's go ahead and add our favorite shirt to the `Clothes` list. We'll do this directly in the terminal window. To do this, enter the following command:

```
>myLists.update({_id:Clothes._id},{$addToSet:{items:{Name:"Favorite
Shirt"}}})
```

This command updates `myLists`, using `Clothes._id` as the selector, and calls `$addToSet`, adding an item with `Name:"Favorite Shirt"`. It might take a few moments for Meteor to update, but you will soon see your favorite shirt added to the list:



If you re-run the `Clothes` assignment command as follows, you will now see that there is an `items` array, with an entry for your favorite shirt:

```
Clothes = myLists.findOne({Category:"Clothes"})
```

We can just as easily update or remove an item by using the `.update()` function with different arguments (`$pull` for removing and `$set` for updating).

For code examples, review the `removeItem()` and `updateLendee()` functions in `LendLib.js`.

For a more in-depth tutorial on MongoDB commands, visit `http://mongodb.org` and click on **Try It Out**.

Now that we've gone through some of the commands that we can implement directly, let's revisit some of our `LendLib.js` code and discuss the reactive code that tracks changes to our collection.

Broadcasting changes

As mentioned in *Chapter 3*, *Why Meteor Rocks!*, communicating changes between the client-side Minimongo and the server-side MongoDB instances happens over DDP. The requests from the client to the server happen automatically when you update a collection on the client. The server to client communication, however, uses a publish/subscribe model that, until now, we have let Meteor take care for us.

Configuring publishers

Up to this point, we have been using a Meteor package named **autopublish**. This means that we haven't had to code specific `publish` events for changes to be broadcast from the server to the client. This is great for testing, but we want to have a bit more control over what events and documents get published so that we that can improve both performance and security.

If we have a large dataset, we may not want to return the entire collection every time. If `autopublish` is being used, the entire collection will return, and this can slow things down or expose data that we don't want to expose.

Turning off autopublish

The time has come to turn off `autopublish`. Temporarily stop your Meteor application (if it's still running) by opening the terminal window from which you ran the `meteor` command. You can stop it by pressing *Ctrl* + *C*. Once it's stopped, enter the following command:

```
> meteor remove autopublish
```

This removes the `autopublish` package, which is responsible for all the automatic publishing of all events inside Meteor.

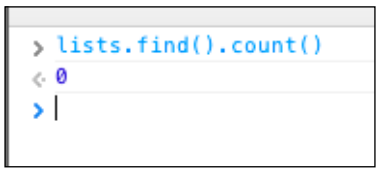
It's considered a best practice to remove `autopublish` from your project. The `autopublish` package is for developing and debugging and should be turned off when you're ready to start using your application in earnest.

By turning this off, you've effectively disabled your application from doing anything! Congratulations! You can see your amazing progress by starting up your Meteor service again (to do this, enter the `meteor` command and press *Enter*) and by opening/navigating to `http://localhost:3000`. You will then see the following screen:

The `categories/lists` are gone! You can even check the console if you like. To do this, enter the following command:

```
> lists.find().count()
```

You should see a count value of 6 or more, but you will instead see a count of 0:



What gives? Well, it's pretty simple actually. Since we removed the `autopublish` library, the server is no longer able to broadcast changes to the client.

Why, again, did we do this? What's the purpose of breaking our application? Ah! Because we want to make our application more efficient and secure. Instead of getting every record automatically, we're going to just get the records that we need and the minimum set of data fields from those records.

Listing Categories

In `LendLib.js`, inside the `if(Meteor.isServer)` block, create the following `Meteor.publish` function:

```
Meteor.startup(function () {
  Meteor.publish("Categories", function() {
    return lists.find({},{fields:{Category:1}});
  });
});
```

This tells the server to create a `Categories` publication. The server will publish this whenever a change is made to the variables found inside the function; in this case, it's `lists.find()`. Whenever a change is made that would affect the results of `lists.find()`, Meteor will trigger/update the publication.

If you noticed, the `lists.find()` call isn't empty. There's a selector, `{fields:{Category:1}}`. This selector tells the `lists.find()` call to only return the specified `fields:`. And only one field is specified—`{Category:1}`.

The preceding snippet of JSON tells the selector that we want to get the `Category` field (`1` = `true` and `0` = `false`). Since this is the only field mentioned and it's set to `1` (`true`), Meteor assumes that you want to *exclude* all other properties. If you had only fields set to `0` (`false`), Meteor would assume that you want to *include* all the other fields that you didn't mention.

For more information on the `find()` function, consult the MongoDB documentation at `http://docs.mongodb.org/manual/applications/crud/`.

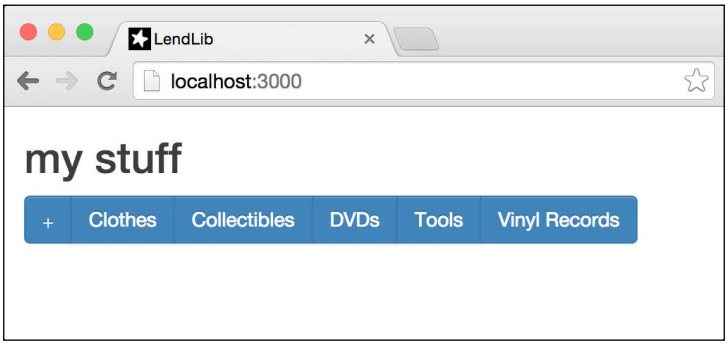
So, if you save this change, your browser will refresh and nothing will happen to the display!

Why? As you may have guessed, removing the `autopublish` library did more than get rid of the `publish` events. It also got rid of the listeners/subscribers. We don't have any subscriber set up to listen on the `Categories` publication channel. So, we need to add a subscriber event that is tuned in to the `Categories` channel.

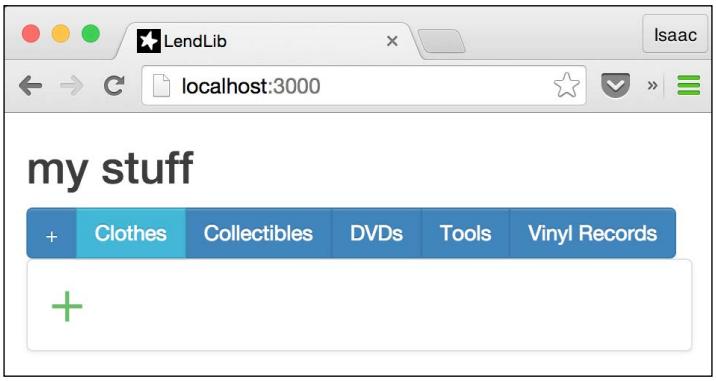
Within the `if (Meteor.isClient)` function, at the very top just inside the opening bracket, enter the following highlighted line of code:

```
if (Meteor.isClient) {
  Meteor.subscribe("Categories");
```

Save this change and you will now see your `Categories` back where they belong:



Before we celebrate, go ahead, and click on the **Clothes** category:

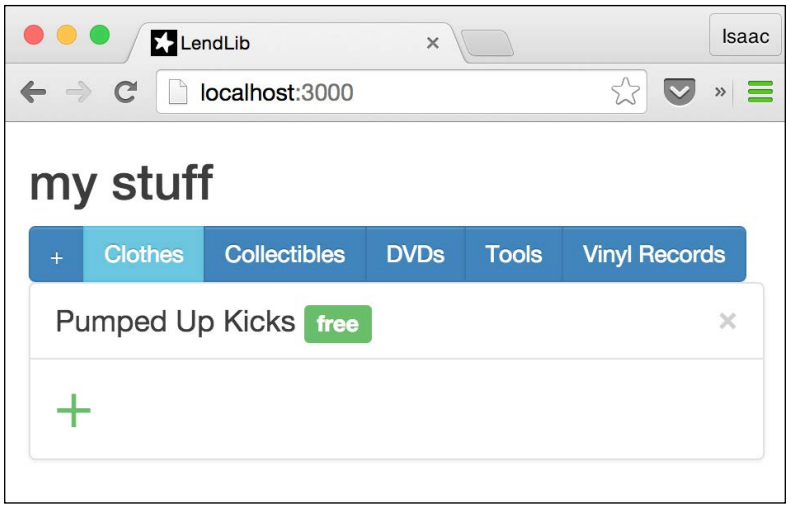


Our favorite shirt is missing! As you probably figured out by now, this is because the publish event that we set up was very specific. The only field in the `Categories` channel that is being published is the `Category` field. All the other fields, including our `items` (and therefore, our favorite shirt), are not being broadcast.

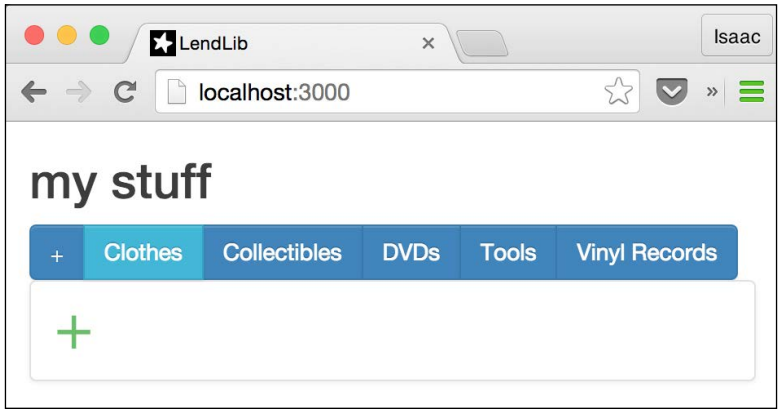
Let's double-check this. Click on the **+** button inside the **Clothes** category in your browser, type **Red Hooded Sweatshirt**, and press *Enter*. The new entry will appear for a split second, and then, it will disappear. This is because of local caching and server sync.

When you enter the new `item`, the local cache contains a copy. This `item` is temporarily visible to your client. However, when the sync with the server occurs, the server update only publishes the `Category` field; so, when the server model updates the local model, the `item` is no longer included.

Let's do one more test, just for funzies. In your terminal window, stop the Meteor service (*Ctrl* + *C*). Now, in your browser, enter another `item` in the **Clothes** category (we'll use `Pumped Up Kicks`). Since the service is stopped, no sync happens with the server, so you're using your local cache and there is your `item`:



Now, start your server again. Your client will sync with the server, and poof! Your item will be gone again:



Listing items

This is no good because we want to see our items. So, let's add items back in and grab the appropriate list of items whenever a `Category` is selected. In `LenLib.js`, just below our first `Meteor.publish()` function and inside the `if(Meteor.isServer)` block, add the following function:

```
Meteor.publish("listdetails", function(category_id){
  return lists.find({_id:category_id});
});
```

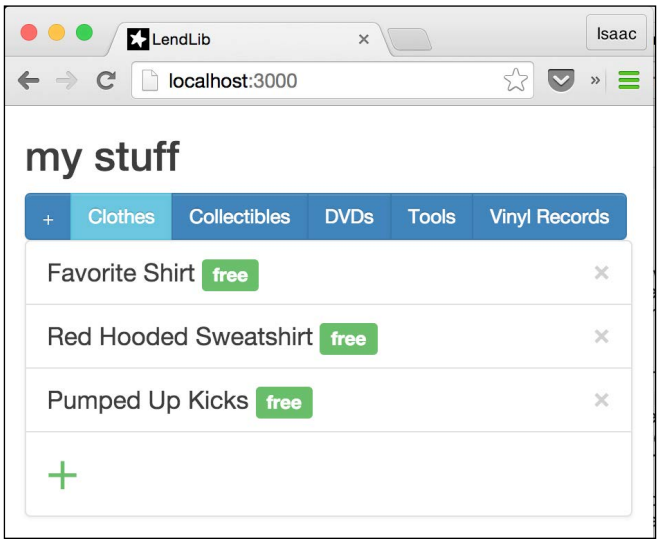
This `publish` function will publish on the `listdetails` channel. Any listener/subscriber will need to provide the `category_id` variable so that our `find()` function returns a leaner recordset.

Notice that nothing has changed in our client yet (your items still aren't visible). This is because we need to create the `subscribe` function.

Just below our first `Meteor.subscribe()` function, add the following function:

```
Tracker.autorun(function() {
  Meteor.subscribe("listdetails", Session.get('current_list'));
});
```

Save your changes and check out your swag **Clothes** collection!



Let's look under the hood here for a minute and figure out what just happened. Notice that the subscription uses `Session.get('current_list')`. This is a reactive context. The `Tracker.autorun()` method is a reactive computation. This means that, any time the `current_list` variable changes, `Tracker.autorun()` re-runs itself and the subscription is updated with the new ID found in the `current_list` variable.

If you remember from *Chapter 4*, *Templates*, we have a `click` event handler set up to listen for `Category` changes. When you click on **Clothes**, for example, an event fires. The `selectCategory()` function inside `LendLib.js` handles the event and changes our `Session` variable:

```
function selectCategory(e,t){
  Session.set('current_list',this._id);
}
```

With the `Session` variable changed, the subscription is re-run. This leads to the `lists` collection being modified (different records for a different `Category`), which then cascades to the helpers templates. Everything reacts to the changes.

In other words, the following actions take place:

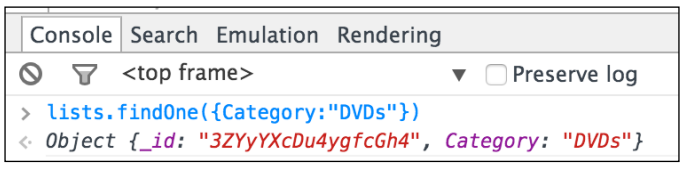
1. `Session.set()` modifies the `current_list` variable.
2. `Tracker.autorun()` re-runs, causing the `Meteor.subscribe()` function to refresh the `lists` collection.
3. Any templates or template helpers that contain the `lists` collection are re-run, and the UI is updated.

Checking your streamlined data

The display is now no different from when we started this chapter. However, underneath the display, the model is much leaner. With the **Clothes** category selected, run the following command in the browser console:

```
> lists.findOne({Category:"DVDs"})
```

Expand `Object`, and you'll see that no items are listed:

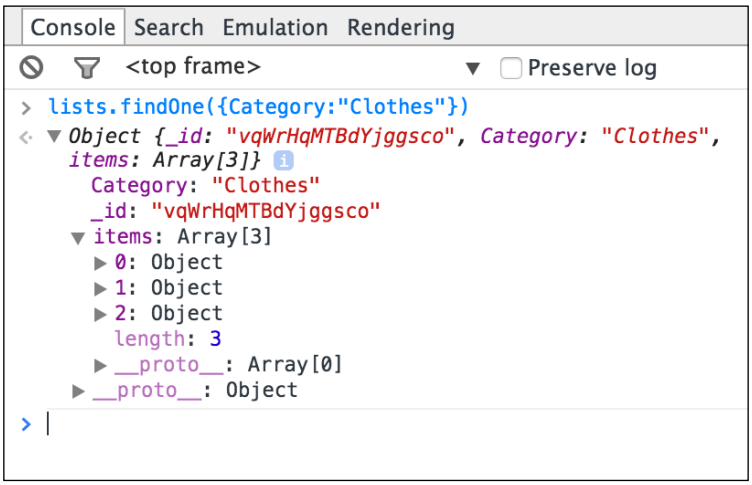


The reason why there are no items is because our `Session` variable, `current_list`, is set to `Clothes` and not `DVDs`. The `find()` function only gets the full record for the `current_list`.

Now, enter the following command in the browser console and press *Enter*:

```
> lists.findOne({Category:"Clothes"})
```

Expand `Object`, and you'll see your three items in an array:



Click around in the app, add items to categories, add new categories, and check the underlying data model that's visible on the client. You'll see that your `lists` are now much less visible; therefore, they are more secure and discreet. This probably won't be a problem for your own personal Lending Library application; however, as we expand this in the next chapter so that multiple people can use the app, the streamlined and discreet data will improve performance.

Summary

In this chapter, you learned what MongoDB is, how a document-oriented database works, and you performed direct queries in the command line. This enabled you to become familiar with Meteor's default data repository system. You also streamlined your application by removing `autopublish` and gained a firm understanding of the publish/subscribe design pattern built into Meteor.

In the next chapter, we'll really tighten up security on your app, allowing multiple users to keep track of and control their own `lists` of items. You'll also see how to further streamline your client and server code through the use of folders.

6
Application Structure – Client, Server, and Public (oh my!)

To allow you to jump right in, Meteor creates a default set of libraries, a default folder structure, and default permissions. This default configuration works great for quick development, testing, and learning as you go. It does not, however, make for a great production environment.

In this chapter, we'll go over the changes that you'll want to make to the default configuration so that your app will be performant, secure, and easier to manage. Specifically, you will learn about the following topics:

- Separating the client, server, and public files of your application
- Enabling database security and user login
- Tailoring display results to protect privacy

The client and server folders

Up to this point, we've put all of our JavaScript code in one file: LendLib.js.

Inside `LendLib.js`, we have two sections that are separated by `if` statements. The client-facing code is found inside the `if (Meteor.isClient) {...}` block, and the server-side code is found inside the `if (Meteor.isServer) {...}` block.

This structure works fine for a very simple application, but when we are writing a more complex application, or we have multiple people working on the same app, trying to share one file with conditional statements will quickly turn into a nightmare situation.

Additionally, Meteor will read any and all files in our application folders and try to apply JavaScript to both the client and the server. This makes for a sort of strange situation if we want to use a client-facing JavaScript library (for example, Twitter Bootstrap or jQuery). If we add the library to the root folder, Meteor will try to implement that file on both the client and the server. This either creates performance issues because we're loading files to the server that it doesn't need, or it produces errors because the server doesn't know what to do with display objects (the server doesn't display anything.)

Conversely, if there is server-side code in files that are accessible by both the client and the server, the client may try to implement that code, which can cause all kinds of problems, or it will at the very least make the code available to the client, which could quickly become a security issue. There are some files and code that we don't want the client to see or have access to.

Let's see an example of the client code being processed by the server, and then let's move that code to a place where only the client will try to execute it. To do this, create a new file in `~/Documents/Meteor/` called `LendLibClient.js`. Then, open `LendLib.js` and cut the entire client code block from it, which is highlighted here:

```
lists = new Mongo.Collection('lists');

[CUT ALL THIS CODE]

if (Meteor.isServer){...
```

You should have to cut well over 175 lines of code. Make sure that you get the ending } bracket!

Now, paste the code that you just cut into `LendLibClient.js` and save the changes to both files. You'll notice that this makes no visual changes to your running application. This is because Meteor is processing both files, and the `if (Meteor.isClient)` condition stops the server from executing the code.

But let's see what happens when we remove the `if` condition. In `LendLibClient.js`, remove the first line containing the `if (Meteor.isClient) {` condition. Additionally, make sure that you remove the last line, containing the ending bracket (}), all the way at the bottom. Save `LendLibClient.js` and then take a look at your console where Meteor is running.

You will see the following error message, or something similar to it:

```
app/LendLibClient.js:21
   Meteor.subscribe("Categories");
          ^
TypeError: Object #<Object> has no method 'subscribe'
    at app/LendLibClient.js:21:11
...
Exited with code: 8
Your application is crashing. Waiting for file change.
```

Removing the `if` condition has created a situation where the server part of Meteor is trying to run the client-facing code. It doesn't know what to do with it, so the app is crashing. We're going to fix the situation by using a folder structure.

If you recall, when we implemented Twitter Bootstrap, we created the `client` folder. Meteor identifies the `client` folder, and it will run any JavaScript files it finds there exclusively as client-facing code and not on the server side.

Move (cut-paste, click-drag, or using `mv`) the `LendLibClient.js` file from `~/Documents/Meteor/LendLib/` to `~/Documents/Meteor/LendLib/client/`. This will instantly fix our crashing app, and Meteor will be happy again! You'll see the following message in the console:

```
=> Modified -- restarting.
```

Since we moved `LendLibClient.js` to the `client` folder, the `if` condition is no longer needed. Due to the file location, Meteor knows that the code is only intended to be run on the client, so it doesn't try to run it on the server.

You may want to refresh your browser that is pointing to `http://localhost:3000`.

This is because you crashed the application. Repent of your evil ways and refresh your page to ensure that everything runs as it should.

Let's now do the same thing with the server code. Create a new folder named `server`. You can do this through a Finder window or directly in the command line as follows:

```
$ mkdir ~/Documents/Meteor/LendLib/server
```

We know that we should create our JavaScript file directly in the new `server` folder. However, we are also pathologically curious, and we enjoy breaking things, so we're going to create it where it can cause problems.

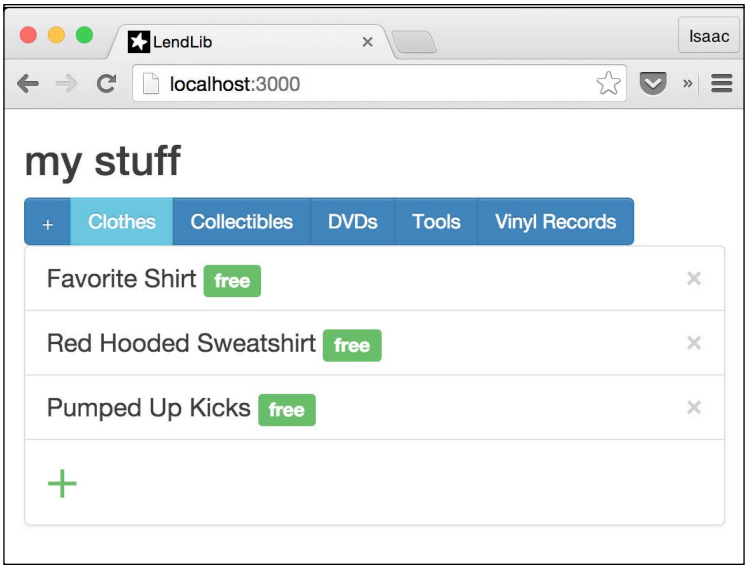
Create a new file named `LendLibServer.js` in the `LendLib` folder under `~/Documents/Meteor/`. Cut the `if (Meteor.is.server) { ... }` block from `LendLib.js`, paste it into `LendLibServer.js`, and save both files.

At this point, there should be only one line of code left in `LendLib.js`:`lists = new Meteor.Collection('lists');`

Just like when we moved the client code, nothing adverse will happen here at this point, because we still have the `if` condition. Let's remove this and let the app crashing continue!

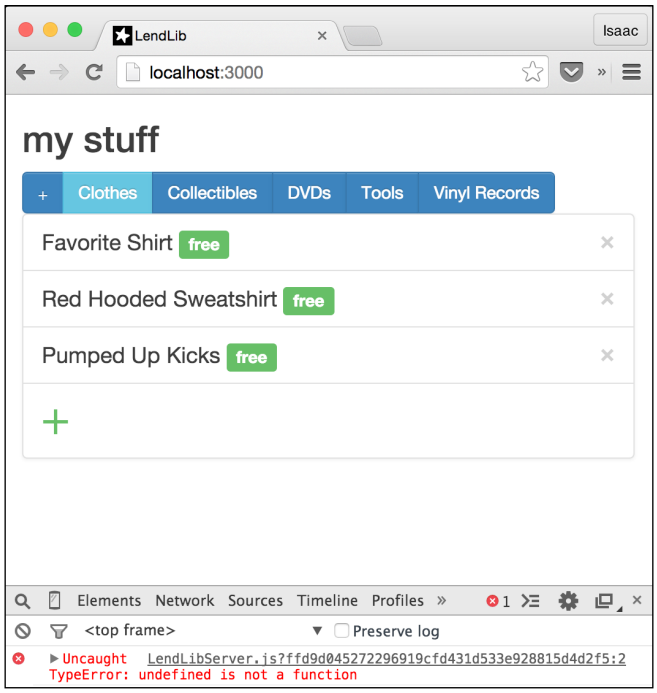
In `LendLibServer.js`, remove the first line containing `if (Meteor.isServer) {` and the last line containing the end bracket (}).

Then, save your changes and let's see the carnage!



Huh. No crashes. The app still works fine. What a letdown...

Let's check the browser console:



Yes! We did do something naughty! The reason this (unfortunately) didn't interfere with or affect the rest of the application is two-fold:

- It's the client side (browser) that threw the error, which won't affect the server application.
- The only code in `LendLibServer.js` is the server code; if this code breaks on the client side, it will be no big deal because it wasn't supposed to run on the client anyway.

The end user will never know that the error is there, but we will; so, let's fix it. Move `LendLibServer.js` to `~/Documents/Meteor/LendLib/server/`. The error will go away and all will be right again in our tiny little Meteor kingdom.

The public folder

It's pretty logical that the `client` folder will only be processed by the client, and the `server` folder will only be processed by the server. However, there's one more consideration that we need to make, and that's for **assets** (images, text/content files, and so on).

The assets are only needed at runtime. We don't depend on them for any logic or processing. So, if we can get them out of the way, the Meteor compiler can ignore them, which will speed up the processing and delivery of our application.

That's where the `public` folder comes into play. When Meteor is compiling CSS or JavaScript for both the client and the server, it ignores anything inside the `public` folder. Then, after all the compiling is done, it will use the `public` folder to access any asset it may need to deliver.

Let's add a '"Created with Meteor"' badge at the bottom of our app. The handsome and generous people at MDG have created a great new logo, and it just so happens to be available on Wikimedia. You can navigate to `http://upload.wikimedia.org/wikipedia/en/a/a4/Meteor-logo.png` and download the image.

Before we can use our downloaded image, we need to make a quick change to `LendLib.css` and create a `public` folder:

1. Open `LendLib.css` (which can be found in `~/Documents/Meteor/LendLib/` unless you moved it to the `client` folder, which is totally fine) and add the following CSS declaration:

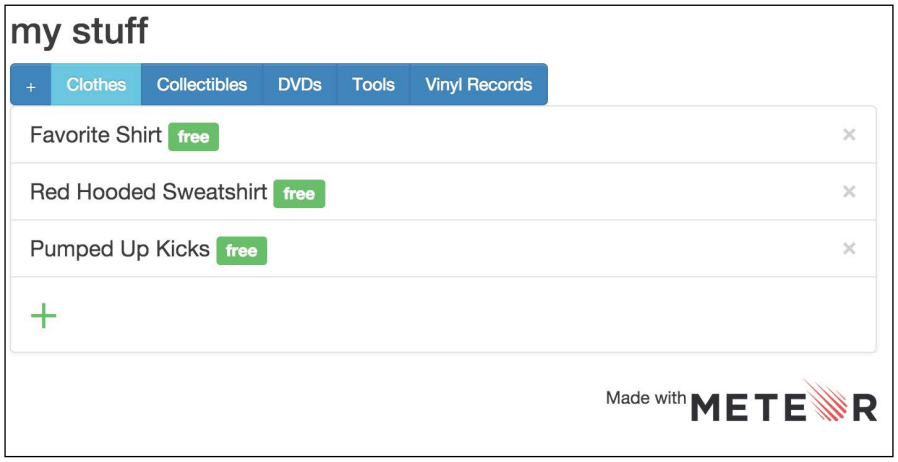
```
.madewith img{
  width:15rem;
}
```

2. Now, open `LendLib.html` and find the `<body>` element. Just before the last `</div>` tag, insert the following code:

```
<body>
  <div id="lendlib" class="container">
    ...
    <div class="madewith pull-right">
      Made with <img src="Meteor-logo.png" alt="">
    </div>
  </div>
</body>
```

Save these changes. Nothing much should have happened (yet), but we'll take care of that right now.

3. Create a folder named `public` under `~/Documents/Meteor/LendLib/`. Now, move the downloaded `Meteor-logo.png` image into the newly-created `public` folder:



You now have a snazzy '"made with Meteor"' badge in the bottom-right corner.

This file is safely tucked away in the `public` folder, so the Meteor compiler doesn't have to deal with it. However, it's still available and ready to go when it needs to be served to a client for display purposes.

Other folders exist, which have varying effects and purposes. For a full explanation, consult the Meteor documentation at `http://docs.meteor.com/#/full/structuringyourapp`.

The security and accounts

At this point, our Lending Library app does exactly what we want it to do. It keeps track of all our stuff and to whom we've lent items. However, if we were to put this app into use, there are some security issues inside the app itself that we'd have to deal with first.

First and foremost, what's to stop someone from accessing our app and erasing their name from an item they borrowed? That scumbag STEVE might just keep our linear compression wrench forever if he were so inclined, and we'd have no way of proving whether or not he still had it.

We cannot let such thievery and dishonesty go unpunished! STEVE must be held accountable! So, we need to implement security. Specifically, we need to perform two actions:

- Only allow editing in the UI by the owner of the items
- Secure the database so that changes can't be made using the web console

Removing insecure

The first step in accomplishing these two goals is to remove the `insecure` library from Meteor. By default, the `insecure` library is included so that we can go about building our application without having to worry about security until we've got our security strategy in place and written most of our code.

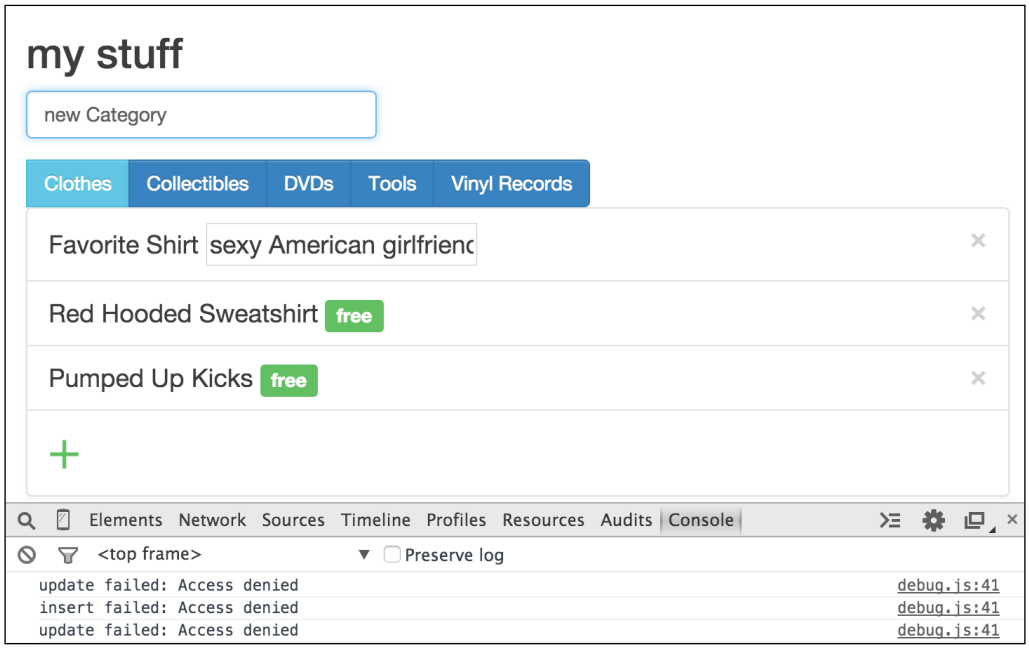
The time has come. We know what we want security-wise, so let's go ahead and get rid of that library. Stop the Meteor application (press *Ctrl* + *C* in the terminal window) and enter the following command (you need to be in the `LendLib/` directory):

```
$ meteor remove insecure
```

This will generate the following message:

```
Changes to your project's package version selections:
insecure removed from your project
insecure: removed dependency
```

Our application is now secure. In fact, it's actually too secure. Start Meteor again (type `meteor` in the terminal and press *Enter*) and navigate to our app in a browser window using `http://localhost:3000`. Once there, try to add a new item, add a lendee, or even delete an item. We'll try to lend our favorite shirt to our significant other, but nothing will happen—no deletions, no additions, and no changes. Nothing will work! If you open the browser console, you'll see that every attempt to update the database will be met with the **update failed: Access denied** message:



These messages are displayed because we removed the `insecure` package. Put another way, no client-initiated changes are allowed anymore. When the change request goes from the client to the server, there are no `allow` conditions; therefore, the request will be denied.

Adding an admin account

To re-enable the update functionality, we need to be able to create an admin account, give the admin account permissions to make changes, and provide the user with a way to recover a lost password.

We'll first need to add three built-in Meteor packages. Stop the Meteor application, and in the terminal window, enter the following command:

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

This commands will add the necessary package (and dependency packages) to our Meteor application, enabling us to administer accounts.

Meteor also has a UI package that will create the login logic for us automatically so that we don't have to write any custom accounts' UI code. Let's add this package while we're at it:

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

Now that we've added the `accounts-ui` package, we just need to quickly configure the fields to be displayed and update our HTML template. To do so, perform the following steps:

1. Open `LendLibClient.js` and append the following code to the very bottom of the file:

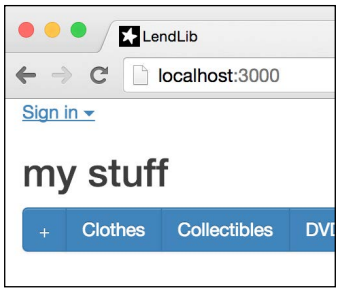
   ```
   Accounts.ui.config({
       passwordSignupFields: 'USERNAME_AND_OPTIONAL_EMAIL'
   });
   ```

   This tells the `accounts-ui` package that we want to display the `username` and `email` fields in the sign-up form, with the `email` field being optional (as we would only need it to recover a lost password).

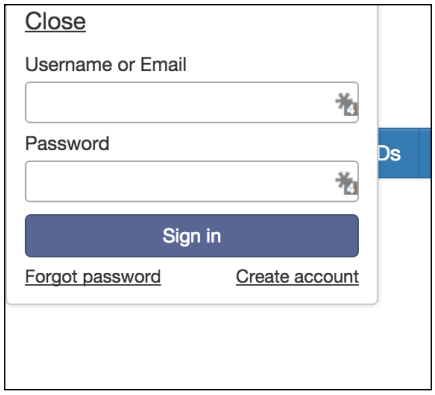
2. Now, open `LendLib.html` and enter the following code directly below the `<body>` tag:

   ```
   <body>
     <div class="container">
       {{> loginButtons}}
     </div>
   ```

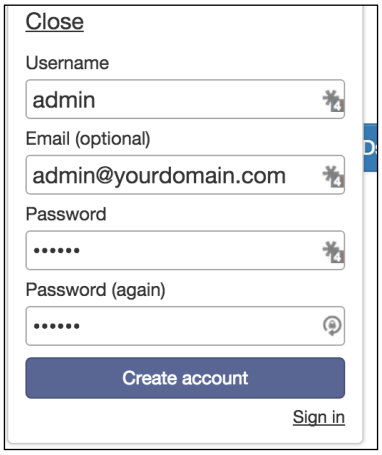
   This HTML code will add a login link and context menu box to the top left of our screen. Let's see this in action. Save all your changes, start your Meteor app, and navigate to `http://localhost:3000` in a browser. Notice the top left of the following screenshot:



3. Click on **Sign in** and then click on **Create account** in the bottom right of the pop up window:



4. Fill in the form, making sure to a username of admin and a valid email address so that you can recover your password, if needed. Enter and confirm your new password and then click on **Create account**:



You will now be logged in as admin, as shown in the following screenshot, and we can proceed with configuring permissions.

Granting admin permissions

Now that we have our admin account, let's allow the account to make any changes that are needed in the UI. At the same time, we'll remove the ability of the user to make changes in the browser console if they are not logged in as admin.

Our original `LendLibServer.js` file currently only has a `Meteor.startup()` method in it. We will add some account checking code to it, ensuring that only the admin account can make changes.

Add the following code to the very bottom of `LendLibServer.js` and save your changes:

```
/*checks to see if the current user making the request to update is
the admin user */

function adminUser(userId) {
   var adminUser = Meteor.users.findOne({username:"admin"});
   return (userId && adminUser && userId === adminUser._id);
}

lists.allow({
   insert: function(userId, doc){
     return adminUser(userId);
},
   update: function(userId, docs, fields, modifier){
     return adminUser(userId);
   },
   remove: function (userId, docs){
     return adminUser(userId);
   }
});
```

Now, at this moment, we have a very interesting situation; if you save all of your changes and refresh your browser, you'll receive an error message:

```
ReferenceError: lists is not defined
    at app/server/LendLibServer.js:15:1
    at app/server/LendLibServer.js:27:3
    at
```

We know that we created a `lists` collection. We've been using it for a while now. So, what gives? Meteor has a certain order of invocation for the files, and as a general rule, any files in a child directory are executed first. Since we moved our `LendLibServer.js` file into the `server/` subfolder, it is being executed before our `LendLib.js` file that is found in the project's root folder.

To fix this, we need to create a new subfolder:

```
$ mkdir ~/Documents/Meteor/LendLib/both
```

Now, move `LendLib.js` into the `both/` subfolder. Refresh your application, and everything will work again.

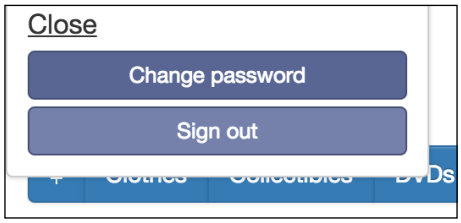
Getting back to the code, the `adminUser` function is used in multiple places; so, it makes sense to create a common function that simply checks to see whether the `userId` making the request is the same as the `_id` of the admin account.

`lists.allow` sets up the conditions upon which operations are allowed, with each operation having a function that returns `true` to allow execution and `false` to deny it. We could, for example, set the `remove` function check to always return `false` if we never wanted to let anyone (including the admin account) delete categories.

For now, we simply want to make the operations conditional on whether the admin account is logged in and is making the request, so we will set each function to `return adminUser(userId);`.

In our browser, we can now test our permissions. Add a new category (anything you'd like, but we'll add `Glassware`), add a new item, change an owner, and so on; all operations should now be allowed, provided you're logged in as admin.

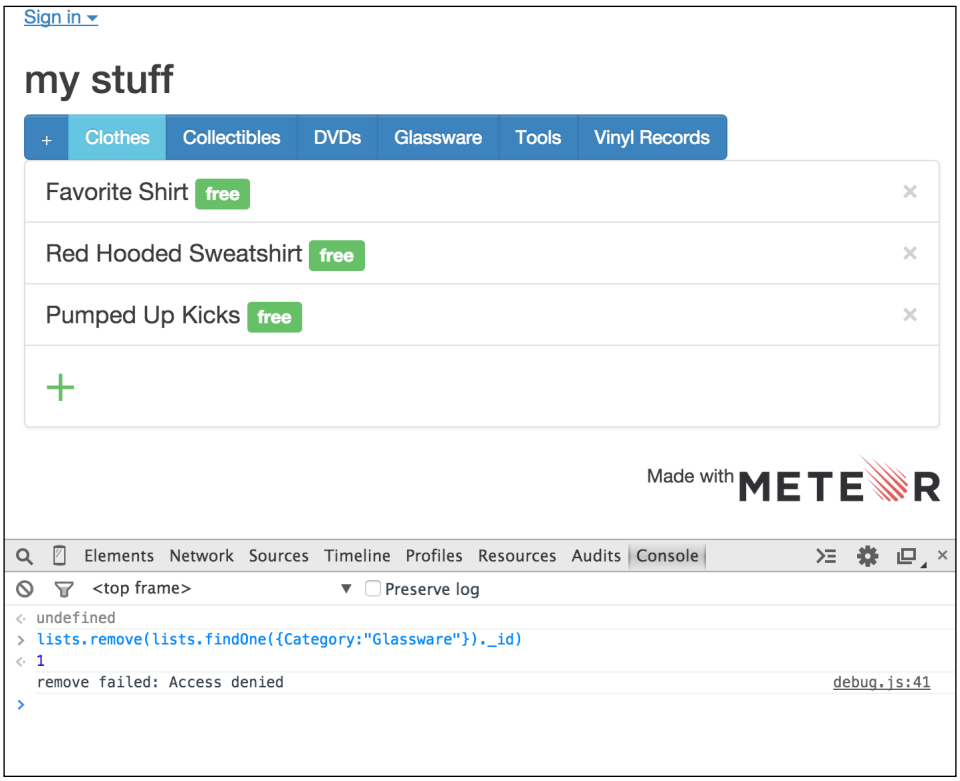
Let's make sure that the access is indeed linked to our admin account. Log out of the app by clicking on **admin** in the top-left corner and then the **Sign out** button:



Now, in the browser console, enter the following command (or equivalent for the category you added):

```
> lists.remove(lists.findOne({Category:"Glassware"})._id)
```

You will again receive an **Access denied** message:



Log back in as admin and run the command again. This time the category will be removed. By setting the permissions and permitted actions on the `lists` level using `lists.allow()`, we've made it impossible for anyone to make changes without being logged in as admin. Both the UI and the browser console are now secure from the evil machinations of one STEVE, the wrench thief!

Customizing results

There is one more thing we should take into consideration when it comes to security and usability of our app. What if we could enable multiple users to use the Lending Library, with each user only able to see the items that belong to them? If we did this, we could stop people from being able to see what kinds of things other people own, and at the same time, we could allow each person to track their own stuff. We originally set out to just create an app for ourselves, but with a little tweaking, we can let anyone use it, and they'll think we're awesome and maybe buy us lunch! Or...a new wrench!

Modifying Meteor.publish()

In preparation for multiple people using our application, we need to make sure that no one can see anyone else's stuff. This is done inside the `Meteor.publish()` declaration for **categories**. Logically, if we limit which categories can be seen by a user, this limitation will cascade to the visible **items** because items are found inside `categories`.

Open `LendLibServer.js` and modify the `find({})` block for the `Meteor.publish("Categories")` method, similar to the following code:

```
Meteor.publish("Categories", function() {
  return lists.find({owner:this.userId},{fields:{Category:1}});
});
```

Adding the `owner:this.userId` selector will check each list in our `lists` repository and return the value under `Category` for each instance where the currently logged in user is the owner of the list. Save this change, and you'll notice that all of the current categories have disappeared!

This is because the categories that we created already don't have any `owner`, and we're logged in as admin; so, the owner property (`undefined`) doesn't match. We're going to experience similar problems when we go to try and modify existing items because no `items` have any owner either.

We have several options to fix this, including manually adding the admin account as the owner, letting the admin account see all unclaimed `lists`, or we can just start with a clean slate. Since we only lent out one item (dang it, STEVE! We want our wrench back!), now is a good time to clear out our database and add back our linear compression wrench, before we forget who has it (yeah, right!)

Open a new terminal window (keep your app running) and navigate to the root folder of your app. Once there, we're going to run the `meteor shell` command:

```
$ cd ~/Documents/Meteor/LendLib
$ meteor shell
```

This will start up the server-side equivalent of the browser console. In this server console, enter the following command:

```
> lists.remove({})
```

This will remove all of our categories, and we can start over once we've added an owner for newly-created categories.

If you want to clear out all users as well, you can do this by stopping the Meteor application, then running `meteor reset` in the terminal window, and finally, restarting the Meteor application. *Be careful!* There's no warning and no takebacks!

Adding owner privileges

Adding an owner to any new categories is pretty simple. We just need to update our `lists.insert()` function and add the owner field. To do this, open `LendLibClient.js` and locate the `Templates.categories.events` declaration. Inside the event delegate for `keyup #add-category`, you will see the `lists.insert()` function call. Modify this call as follows:

```
'keyup #add-category': function (e, t) {
  ...
      lists.insert({
        Category: catVal,
        owner: Meteor.userId()
      });
  ...
},
```

Now, whenever a new list is added, instead of just adding a `Category` field, we will also add an owner field. This allows our `Meteor.publish()` code to work correctly for any new `lists` that we make.

Let's add back the **Tools** Category, enter the item **Linear Compression Wrench** and assign the Lendee as **STEVE**:

There, we're up and running! Inside each item, we now have an `owner` property. This becomes important when we enable others to create and maintain their own `lists` of items.

Enabling multiple users

Okay, everything is in place now for us to have a customized, private view of our own stuff, but currently, only the admin account can add `lists` or `items` and assign a Lendee to an item.

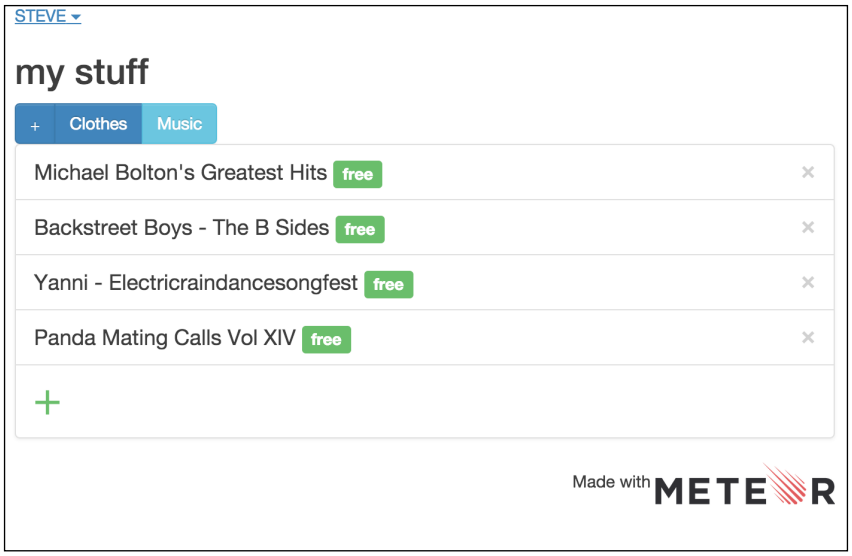
We'll fix this by going back to `LendLibServer.js` and adding some logic to check whether the currently logged-in user owns the list or is an admin. Inside `LendLib.js`, in the `lists.allow()` code block, make the following additions:

```
lists.allow({
  insert: function(userId, doc){
    return (adminUser(userId) || (userId && doc.owner === userId));
  },
  update: function(userId, docs, fields, modifier){
    return (adminUser(userId) || (userId && docs.owner ===
      userId));
  },
  remove: function (userId, docs){
    return (adminUser(userId) || (userId && docs.owner ===
      userId));
  }
});
```

Inside `insert`, we check to see whether the current `doc.owner` user is the logged-in user. In `update` and `remove`, the name of the parameter changes to `docs`, but the check is the same since we are only updating one record at a time. We, therefore, check the `docs.owner` property.

You will now want to save your changes and create a new account on `http://localhost:3000`. You can add categories and items to your heart's content. You can switch between users and add as many users and categories as you like.

You'll notice that there's no visibility from one person's lists to another, and consequently, there is no way for someone to manipulate or delete another person's lists and records. Now, when STEVE finally gets his grubby little hands on your application, he can only see his stuff (none of which is worth borrowing by the way!):



Summary

In this chapter, you learned how Meteor compiles and searches for JavaScript and CSS code, and how you can optimize that search. You also learned how to protect your server code and keep things running smoothly and efficiently. In addition, you learned how to secure your database through the use of Meteor's built-in `accounts` packages, and you closed some of the security loopholes in your application. Finally, you enabled multiple accounts, so anybody can use your Lending Library to keep track of their items, and you did this without compromising on the privacy of the end user.

In the next chapter, you will learn how to deploy a Meteor application to a production environment and learn techniques on how to start coding fast, robust, production-ready Meteor applications.

7
Packaging and Deploying

Our application is looking great. We've made it secure, easy to use, and with the addition of multiple logins, anybody can now use the Lending Library to keep track of their stuff.

In this final chapter, we'll go over Meteor's amazing package system, which will speed up future code projects, and we'll talk about the options for deploying your applications. You will learn how to:

- Add and configure third-party packages such as JQueryUI, Backbone, and Bootstrap
- Bundle your entire application so that it can be deployed
- Deploy your app using Meteor's public servers
- Deploy your app to a custom server

Third-party packages

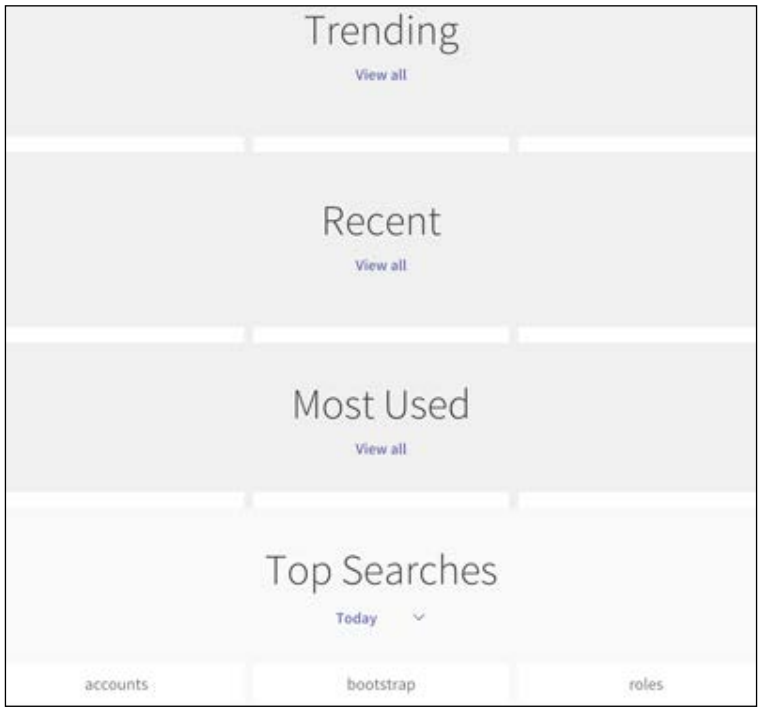
Through a thriving, growing community, Meteor is rapidly gaining packages for all major JavaScript and Preprocessing libraries. These packages are intelligent in that they contain the base JavaScript or Preprocessing libraries, but they are also configured to interact directly with the Meteor code base.

What this means for you is that adding your favorite library involves almost no effort, and you can be confident that it will work with your Meteor application.

Finding the available packages

The best way to find new packages is through a web interface called Atmosphere. If you navigate to `https://atmospherejs.com`, you will be prompted with a search box:

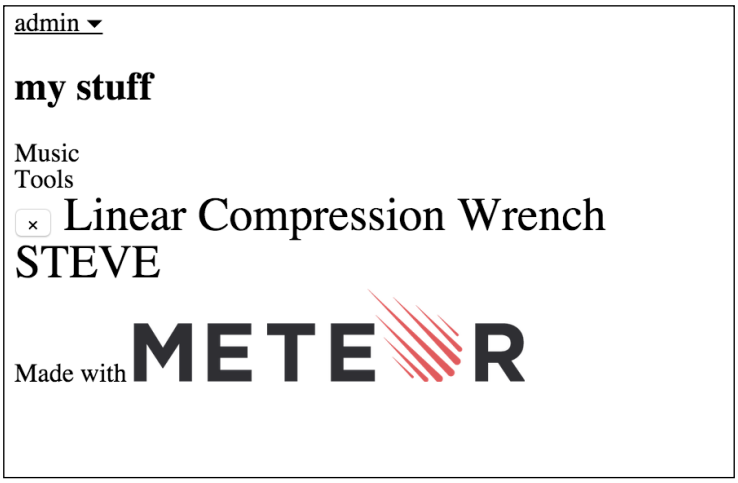
Type in the names of some popular frameworks (such as `bootstrap`, `iron router`, `react`, and so on) and the chances are that you will be able to see multiple libraries to choose from. You can explore the most popular or most used packages as well by scrolling down on the home page:



The **Trending**, **Recent**, **Most Used**, and **Top Searches** divisions as seen on the home page

As you can see, quite a few of the most popular frameworks are available, including Polymer, Angular, underscore, and… Twitter's Bootstrap! We spent a good amount of time manually downloading Bootstrap, creating the `client` folder, and extracting the Bootstrap files. That was a good exercise in how to manually install a framework, but now, we're going to learn how to install it as a package.

First, let's remove the existing Bootstrap installation. To do this, navigate to `~/Documents/Meteor/LendLib/client/` and delete the `bootstrap` directory. It doesn't matter whether or not your Meteor app is running (remember, Meteor updates dynamically!). After removing the `bootstrap` directory, either start your app and then navigate to `http://localhost:3000`, or just navigate there if your app is already running. You will see that all of our pretty formatting is gone!



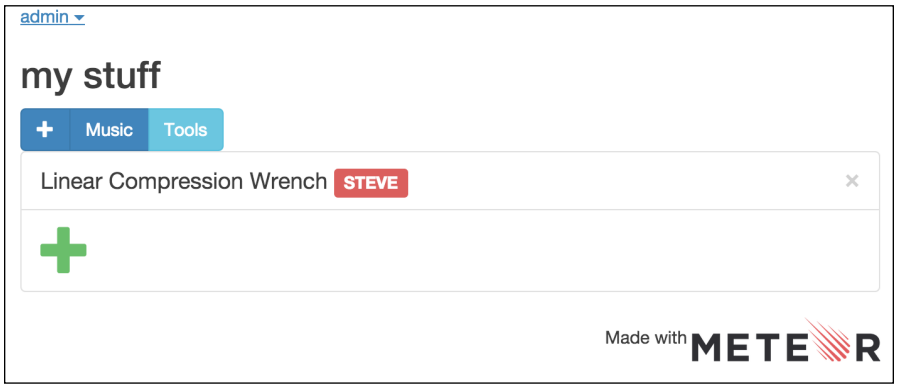
We'll now add the official Meteor bootstrap package. Again, because Meteor updates dynamically, so we don't have to stop the Meteor app unless we want to. Either open a new terminal window or temporarily stop your Meteor application, and make sure that you are in the `LendLib` folder under `~/Documents/Meteor/`. Once there, enter the following command:

```
$ meteor add twbs:bootstrap
```

You will receive a very short message, as follows:

```
Changes to your project's package version selections:

twbs:bootstrap  added, version 3.3.2


twbs:bootstrap: Bootstrap (official): the most popular HTML/CSS/JS fra
mework for responsive, mobile first projects
```

If you used a second terminal window, just head to your browser (you don't even have to refresh the page). If you stopped your Meteor application, start it up again and navigate to `http://localhost:3000`. You will be able to see that the Bootstrap formatting is now back, and everything is back to normal:



This is literally all there is to it. By using a single command in your terminal, you can add a library or framework to your project without having to worry about linking, downloading, making sure that the files are in the right location, and so on. Just run `meteor add` and off you go!

You can get a list of the packages that you're already using by entering the following command in the terminal:

```
meteor list
```

Since the Meteor community is adding packages so rapidly, it's a good idea to stay current with your Meteor installation. From time to time, run the following command in the terminal:

```
$ meteor update
```

If you're on the latest version, it will tell you so, and what version you're running. If there's a new version, it will download and install it for you and update any packages as well.

Bundling your application

In usual Meteor fashion, bundling your application so that it can be deployed is incredibly simple. Stop your Meteor application if it's running, make sure you are in your application folder (for the Lending Library, this is `LendLib` folder under `~/Documents/Meteor/`), and enter the following command in the terminal:

```
$ meteor bundle ~/Documents/Meteor/builds/
```

This will take a little bit to run, but when it's finished, you'll have a `LendLib.tar.tgz` tarball in the `builds/` folder, and you'll be ready to deploy it wherever you like. This is a complete Node package/bundle. The machine to which you deploy this bundle only needs to have **Node.js** and **MongoDB** installed on it. Everything else that you need is included in the bundle.

Deploying your application to Meteor's servers

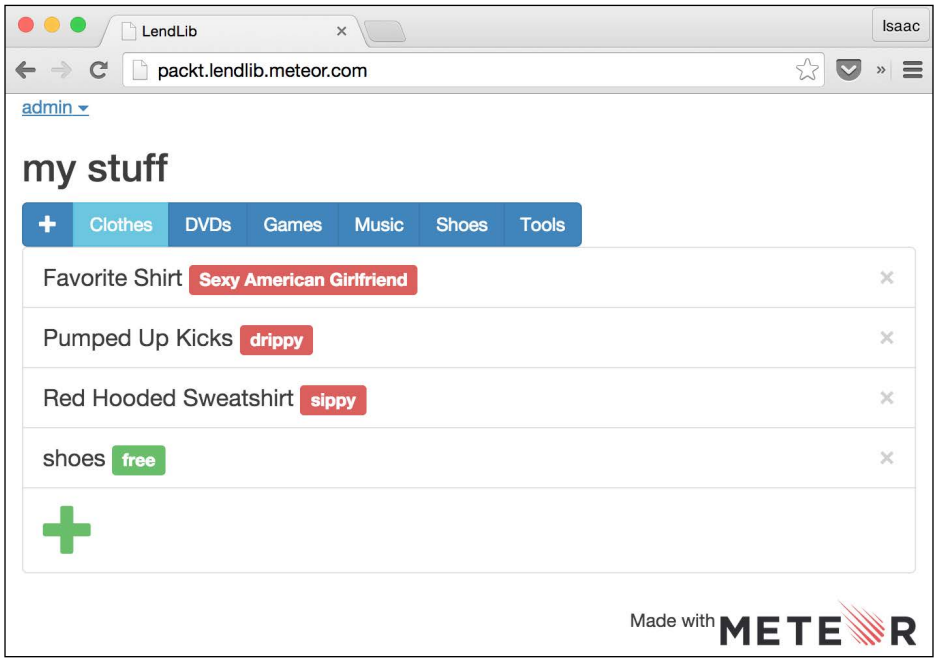
The folks at Meteor have taken deployment one step further - above and beyond what you'd expect from even a paid product, much less a free one. Meteor allows you to deploy your application directly on their deployment servers. Pick a name for your app (we'll use `packt.lendlib`, but you'll need to come up with your own) and enter the following command in the terminal:

```
$ meteor deploy [your app name].meteor.com
```

So, in our case, we entered `meteor deploy packt.lendlib.meteor.com`. The console will give you status updates as it bundles, uploads, and deploys your application. Once it's finished, it will give you a message similar to this:

```
Now serving at [your app name].meteor.com
```

If you navigate to that URL in your browser (for example `http://packt.lendlib.meteor.com`), you will see that your application has been deployed and is running!



You will probably want to create the admin login before you start using the application, or to tell others about it. You don't want sneaky STEVE to have control of your app!

Updating Meteor's servers

What if you make changes, or you had a bug, and you want to update the code for your app on Meteor's servers? As you probably guessed, this is super simple. Just re-run your deploy command:

```
$ meteor deploy [your app name].meteor.com
```

This not only updates your app, but it preserves your data as well so that you don't have to start from scratch if you've entered a lot of information already. Pretty slick, right? The people at Meteor really know what makes developing enjoyable, and they've gone out of their way to provide an environment where you can just code, play, and receive instant feedback on your application.

Using your own hostname

But wait, there's more! You can even use one of your own hostnames with an app that you deploy on Meteor's servers. Set up a `CNAME` pointing to `origin.meteor.com` using your host provider, and you can then deploy your app to that hostname. For example, if we had the `meteor.too11.com` subdomain pointing as a `CNAME` to `origin.meteor.com`, we would run the following command in the terminal:

```
$ meteor deploy meteor.too11.com
```

If your `CNAME` is set up properly, the app will deploy just as it would if you were to deploy directly to `[your app name].meteor.com` and will be available with your customized hostname!

Check with your host provider about setting up a CNAME route. It varies from provider to provider, but it's pretty easy to do.

Deploying your application to a custom server

At the time of writing, deploying a Meteor application to either a hosting service or to your own personal machine is a pretty hefty task. There are versioning issues with deployment, and most hosting services are still in the early stages of supporting Meteor bundles.

That being said, Meteor is working on a really slick deployment service (nicknamed **Galaxy**), and we have a pretty great third-party plugin available with us called Meteor Up (MUP), which was developed by the incomparable Arunoda Susiripala at `http://meteorhacks.com`. Let's walk you through deploying our Meteor application to a custom server using MUP.

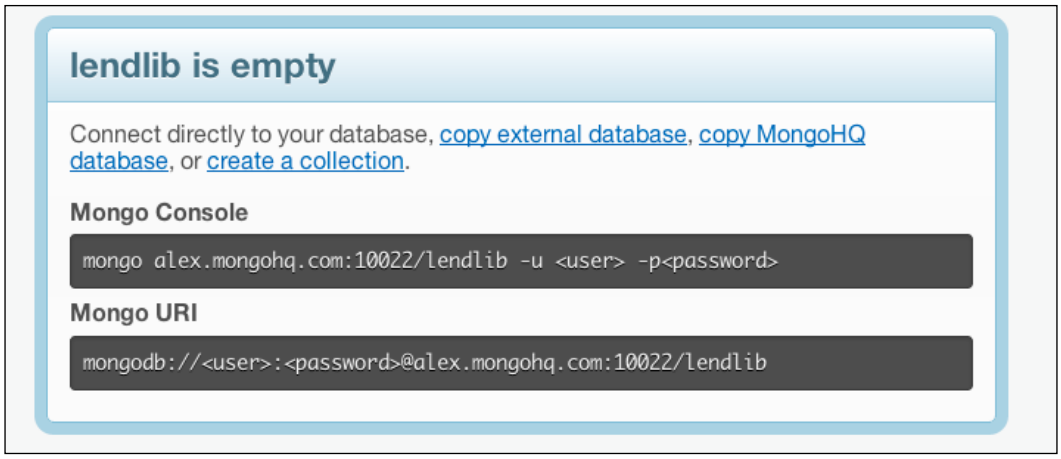
The server setup

The server from which you'll be hosting your application needs to have the latest *stable* version of Node.

To Install Node.js, go to `http://nodejs.org/` and follow the instructions for either Linux or Mac OS X installation.

MUP will install the correct MongoDB version that we need on our remote server, so we don't need to worry about that, at least not for testing purposes.

Alternatively, you could set up MongoDB on a remote server or use a hosting service such as MongoHQ (`https://www.mongohq.com`). If you do use a remote service, set up a new database and note the URI that you'll need. An example from MongoHQ is shown here:



Installing and configuring MUP

You will need Meteor Up installed on your client machine, and that's super easy to do, so let's make it happen!

In a terminal window, execute the following command:

```
$ npm install -g mup
```

Note that you may need to use the `sudo` command, depending on your user permission settings. The preceding command installs the MUP module globally, so we can use it wherever we like. After the installation is finished, make sure that your application is stopped and that you are in the root folder of your project (`~/Documents/Meteor/LendLib`). We will now initialize MUP in our project. In the terminal, execute the following command:

```
$ mup init
```

This will generate a configuration file named `mup.json` in the root directory. Go ahead and open the file. You will see the name of the server to which you'd like to deploy your app at the top. You will also see a username / password / pem properties section. These fields are necessary so that MUP can log into your remote hosting server and configure it properly.

You will need to fill in the fields with your SSH login information. If the address for your remote server were `http://myserver.com`, you would enter the following code snippet:

```
{
      "host": "myserver.com",
      "username": "root",
      "password": "SUPERHARDTOGUESSpassword"
      // or pem file (ssh based authentication)
      //"pem": "~/.ssh/id_rsa"
}
```

The next section asks us whether we want to set up MongoDB and Node. If your remote server is already running these, there's no need to install them now; so, you can turn the values to `false`. If you leave the values as `true`, it will re-install both Node and MongoDB. It's up to you. Here, we will leave the values as true:

```
  "setupMongo": true,

  "setupNode": true,

  "nodeVersion": "0.10.33",
```

PhantomJS is also a necessary component. During the first deployment of the app, we will leave the `setupPhantom` property set to `true` so that PhantomJS will be installed. However, on subsequent deployments, it's entirely up to you. Here's how we used it for the first run:

```
"setupPhantom": true,
```

Since the name of our app is Lending Library, we will be changing the `appName` property to the following:

```
"appName": "LendLib",
```

The next property tells MUP where to look on your client machine for the source files so that they can be bundled and uploaded to your remote server. In our case, we would enter the following line of code:

```
"app": "~/Documents/Meteor/LendLib",
```

The last property is the `env` property, which specifies how to reach your app once it's deployed. Usually, the `env.ROOT_URL` property will be set to the `localhost` address: `http://localhost`. We can then specify a `PORT` property if we like. If our `ROOT_URL` is set to `http://myserver.com` and the `PORT` property is set to `8080`, our declaration would look like this:

```
"env": {
  "ROOT_URL": "http://myserver.com",
  "PORT": 8080
},
```

If you want to deploy your app using port `80`, you must have root privileges on the remote server, for security reasons.

Deploying your app using MUP

With our `mup.json` file in place, it's time to initialize our remote server and let it install the software as needed. In the terminal window, enter the following command:

```
$ mup setup
```

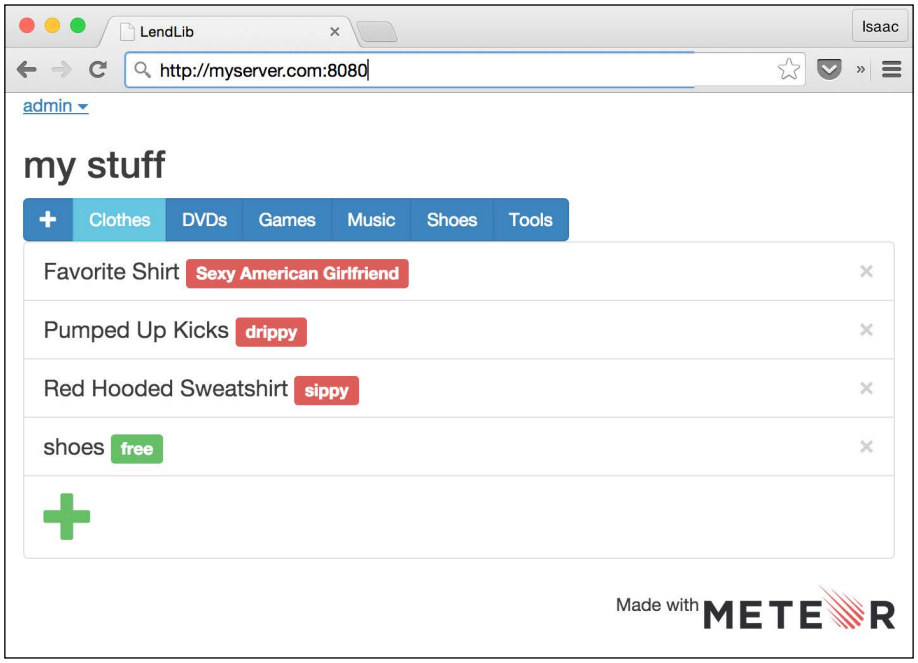
This will install MongoDB, Node, and PhantomJS on the remote server. It will also configure our remote server environment and install some helper NPM packages, such as `upstart`.

Once the `mup setup` command completes, we will be ready to deploy your 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 the app is deployed, we can navigate in a browser to your app's URL and check out your brand new, fully deployed app, which will look similar to the following screenshot:



As we rushed through the basics of using MUP, there were several topics that we didn't cover, including setting up a reverse-proxy, load balancing, and DNS entries. We are making the assumption that if you want to deploy your app manually, you have some prior experience with setting up your server environment properly.

All MUP configuration settings and more detailed instructions for a MUP installation can be found at `https://github.com/arunoda/meteor-up`.

You can go even further with deployment, such as setting up environment variables, running your app as a daemon/service, or even using remote servers to publicly host your application. What we've done so far should be sufficient to get you started and be well on your way to use Meteor in a production environment.

Summary

You made it! In no time at all, you built a working application and, hopefully, had fun doing it! You now know how to build an application in Meteor from the ground up. You understand the design patterns and development principles behind Meteor, and you can tailor, optimize, and secure your application to do anything you want. You can also deploy Meteor to multiple environments. You are well on your way to writing productive, efficient, and rock-solid web applications.

The very best way to gain more knowledge about Meteor is to use it. We suggest using Meteor in your upcoming development projects, contributing to the Meteor community through code contributions, answering questions on the forums, and making tutorials of your own.

Meteor is a breakthrough technology, which is gaining more and more momentum. You now have the knowledge and experience that you need to use this breakthrough technology in your personal and professional development projects, keeping you well ahead of the curve and making your life as a developer that much more rewarding.

Index

S

T

V

W

Thank you for buying
Getting Started with Meteor.js JavaScript Framework
Second Edition

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

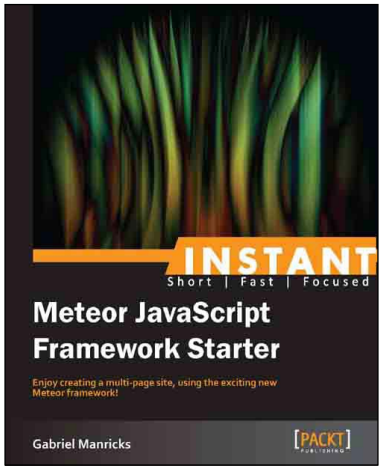
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

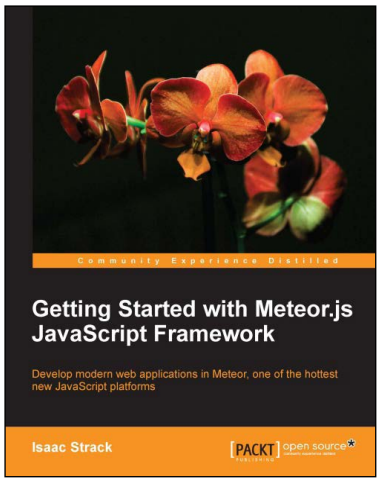


Instant Meteor JavaScript Framework Starter

ISBN: 978-1-78216-342-8 Paperback: 78 pages

Enjoy creating a multi-page site, using the exciting new Meteor framework!

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create multi-page Meteor sites.
3. Learn best practices for structuring your app for maximum efficiency.



Getting Started with Meteor.js JavaScript Framework

ISBN: 978-1-78216-082-3 Paperback: 130 pages

Develop modern web applications in Meteor, one of the hottest new JavaScript platforms

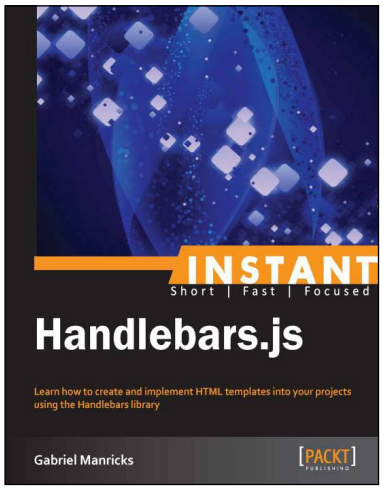
1. Create dynamic, multi-user web applications completely in JavaScript.
2. Use best practice design patterns including MVC, templates, and data synchronization.
3. Create simple, effective user authentication including Facebook and Twitter integration.

Building Single-page Web Apps with Meteor

ISBN: 978-1-78398-812-9 Paperback: 198 pages

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

1. Create a complete web blog from frontend to backend that uses only JavaScript.
2. Understand how Web 2.0 is made by powerful browser-based applications.
3. Step-by-step tutorial that will show you how fast, complex web applications can be built.



Instant Handlebars.js

ISBN: 978-1-78328-265-4 Paperback: 62 pages

Learn how to create and implement HTML templates into your projects using the Handlebars library

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Create and display templates on your pages.
3. Extend Handlebars with custom helpers.