



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Getting Started with Meteor.js JavaScript Framework

Develop modern web applications in Meteor, one of the hottest new JavaScript platforms

Isaac Strack

[PACKT] open source*
PUBLISHING community experience distilled

Getting Started with Meteor.js JavaScript Framework

Develop modern web applications in Meteor, one of the hottest new JavaScript platforms

Isaac Strack



BIRMINGHAM - MUMBAI

Getting Started with Meteor.js JavaScript Framework

Copyright © 2012 Packt Publishing

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

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

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

First published: December 2012

Production Reference: 1201212

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

ISBN 978-1-78216-082-3

www.packtpub.com

Cover Image by Asher Wishkerman (wishkerman@hotmail.com)

Credits

Author

Isaac Strack

Project Coordinator

Amigya Khurana

Reviewers

Arturas Lebedevas

Gabriel Manricks

Proofreader

Chris Smith

Acquisition Editor

Wilson D'souza

Indexer

Monica Ajmera Mehta

Commissioning Editor

Ameya Sawant

Graphics

Aditi Gajjar

Technical Editors

Veronica Fernandes

Dipesh Panchal

Lubna Shaikh

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Author

Isaac Strack, as a Design Technologist for Adobe Systems, actively researches, develops, and contributes to emerging device and Internet technologies, incorporating these new technologies into the Adobe Digital Media and Digital Marketing product lines. He is on the board of directors for the Wasatch Institute of Technology, a computer science high school located in Utah that is changing the face of education through an Agile-based teaching methodology, which emphasizes real-life technology skills and STEM education.

Isaac worked for the Service Technologies group at eBay for over 11 years, where he was on the forefront of AJAX, .NET, and web-related technologies. While at eBay, he earned a web technology patent, and is one of the original developers of the Listing Violation Inspection System (LVIS), used to monitor and regulate auctions and member-to-member transactions.

Isaac has a passion for technology and design, and conveys that passion through his contributions online and in his local community. Despite his experiences to the contrary, he's still naive enough to believe what Steve Jobs said, "If you have a good idea and a little moxie, you can change the world."

I want to thank my four wonderful daughters, for teaching me what true, unconditional love is, and for making me feel young and happy, even on cold winter days. I want to thank my wife, Kirsten, for encouraging me to never give up on my stupid, stupid dreams, and for being so supportive and sacrificing during the making of this book. I'm grateful to my employer, Adobe Systems, and my manager, Joel Den Engelsen, who continually support me, and have given me my dream job. Lastly, I want to thank my Heavenly Father, for my talents and blessings, and for the love/passion I have for learning new, amazing things. I truly am better than I deserve, and I am grateful for the peace in my heart, despite my best efforts to ruin everything.

About the Reviewers

Arturas Lebedevas is a Software Developer who has been working on various projects in both Lithuania and Ireland. Previously, he was the co-founder and CTO of an Irish legal startup LawSimply, where he used Node.js extensively along with MongoDB.

Currently he is doing software consultancy focusing mainly on using Meteor framework, and has been an active member of the Meteor framework community contributing to Stack Overflow.

I would like to thank to my mother who supports me in all my decisions.

Gabriel Manricks is a Software/Web Developer born in Montreal, Canada. He learned his first programming language at the age of 12 (C++), and went on to graduate in programming science.

In addition to programming, Gabriel's hobbies include electronics and crafts; basically anything involving taking things apart, seeing how they work, and putting them back together.

Currently Gabriel is a Staff Writer for NetTuts+, where he enjoys learning and teaching cutting-edge web technologies.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Setup and Installation	7
Installing with curl	7
Loading an example application	9
Selecting your file location	9
Loading the example application	10
Starting the example application	10
Previewing the application	10
Help! I made too many changes!	11
Making code changes	12
Changing from todos to items	12
Summary	14
Chapter 2: Reactive Programming... It's Alive!	15
Creating the Lending Library	15
Creating the base application	16
Creating a collection	18
Fun with the browser console	19
Adding some data	20
Displaying collections in HTML	21
Cleaning up	25
Creating a reaction	28
Multiple clients	29
Summary	30
Chapter 3: Why Meteor Rocks!	31
Modern web applications	31
The origin of the web app (client/server)	31
The rise of the machines (MVC)	32
The browser grows up (MVVM)	33

A giant Meteor appears!	35
Cached and synchronized data (the model)	35
Templated HTML (the view)	37
Meteor's client code (the View-Model)	39
Let's create some templates	40
Summary	46
Chapter 4: Templates	47
A new HTML template	47
Gluing it all together	51
Our items View-Model	51
Additional view states	54
Adding events	57
Model updates	61
Style updates	64
Summary	67
Chapter 5: Data, Meteor Style!	69
Document-oriented storage	69
But why not use a relational database	70
MongoDB	71
Using direct commands	72
Broadcasting changes	75
Published events	75
Configuring publishers	76
Turning off autopublish	77
Listing categories	78
Listing items	81
Checking your streamlined data	82
Summary	84
Chapter 6: Application and Folder Structure	85
Client and server folders	85
Public folder	89
Security and accounts	91
Removing insecure	91
Adding an admin account	92
Granting admin permissions	95
Customizing results	98
Modifying Meteor.publish()	98
Adding owner privileges	99
Enabling multiple users	100
Summary	102

Chapter 7: Packaging and Deploying	103
Third-party packages	103
Listing available packages	103
Bundling your application	105
Deploying to Meteor's servers	106
Updating Meteor's servers	107
Using your own hostname	107
Deploying to a custom server	107
Server setup	108
Deploying your bundle	108
Optional – different platform	109
Running your application	109
Summary	111
Index	113

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.

And yet, we aren't easily amazed. We've come to expect wondrous advances, and therefore what was once amazing becomes...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, prepare to regain some of that wonder. A dedicated group of computer scientists who are determined to make something wondrous have created a new JavaScript platform called Meteor. You may be thinking, "A new JavaScript platform? That's nothing special." And if that's all Meteor was, you'd be correct, but 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 applications with just a few lines of code.

In the context of web applications, it is state-of-the-art. Using established, proven development design patterns, Meteor takes all the difficult and mundane parts of building a web application and does them all for you. You get to focus on building a solid application with all the latest innovations such as reactive programming, templates, plugins, and client-side caching/synchronization. You get to do all of this without getting bogged down in 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. It 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 shows how quickly and easily you can build a fully functional, 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 Meteor uses, and shows examples of these powerful patterns in action.

Chapter 4, Templates, teaches you about Meteor templates in depth, and how to use 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. It also helps you to implement Meteor's data handling quickly and effectively in your application.

Chapter 6, Application and Folder Structure, shows 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. Learn how to deploy a Meteor application to your development, testing, and production environments.

What you need for this book

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

- Operating System:
 - Mac: OS X 10.6 and above (<http://www.apple.com>)

- Linux: x86 or x86_64, Debian (<http://www.debian.org>) and Red Hat-based systems (<http://www.redhat.com>)
- Meteor: Version 0.5.0 or above (<http://docs.meteor.com/#quickstart>)

Who this book is for

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

If you are looking for a step-by-step approach to understanding how and when to use one of the latest and most innovative web technologies in your application development projects, 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: "We've already created our categories through the use of the `categories` template."

A block of code is set as follows:

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

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


```
<body>
  <div id="lendlib">
    <div id="categories-container">
```


```
    {{> categories}}  
  </div>  
  <div id="list">  
    {{> list}}  
  </div>  
</div>  
</body>
```

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

```
> meteor remove autopublish
```

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

 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 via 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

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, designed to make the building of a web application easier. That's a terrible way to describe something so elegant, but it helps us 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 through how to get the Meteor package properly installed on your Linux or Mac OS X system, and then see that package of files and scripts in action. Note that Windows support is coming, but as of the time of this writing, only the Linux and Mac versions are available.

In this chapter, you will learn:

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

Installing with 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; I lied. But still, you want to use the command line, trust me. Trust the person 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:

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



Downloading the example code

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

This 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 `/bin/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 download and install:

```
isaacstrack-MacBookPro:local isaacstrack$ curl https://install.meteor.com | /bin/sh
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left   Speed
100 5239    0 5239    0    0  5112      0 --:--:--  0:00:01 --:--:-- 7347
Installing Meteor to /usr/local/meteor
... downloading
##### 100.0%

Meteor installed! To get started fast:

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

Or see the docs at:

docs.meteor.com

isaacstrack-MacBookPro:local isaacstrack$
```

The key thing we're looking for in the preceding installation text is the location of Meteor:

Installing Meteor to `/usr/local/meteor`

This location will vary depending on if you're running this 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:

Meteor installed! 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 creating your first Meteor web application!



You should bookmark `http://docs.meteor.com` as 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, helping you get a better idea of what Meteor is capable of.

For the application we will build, the `todos` example is the closest fit, so we'll go ahead and build off of that example. We'll be using the command line again, so awesome news if you still have it open! If not, open a terminal window, and follow these steps.

Selecting your file location

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

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

Now, we want to be in that directory:

```
$ 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 `todos` example application:

```
$ meteor create --example todos
```

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

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

How handy, there are even instructions on what to do next! 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 is looking for the startup files, HTML, and JavaScript needed to run the application. Those are all found in the application folder, so let's go there:

```
$ cd todos
```

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

```
$ meteor
```

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

```
[[[[[ ~/Documents/Meteor/todos ]]]]]
```

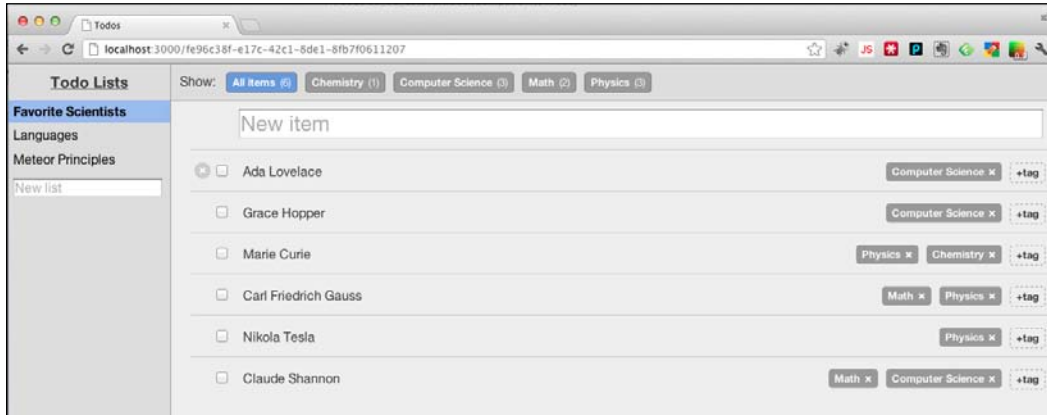
```
Running on: http://localhost:3000/
```

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 the following screen, with a few todo lists already added:



You can go ahead and poke around the application if you'd like. Add a new item to a list, change lists, add a new tag, or mark items as complete. Go nuts, friend! Any changes we make in the future won't match exactly what you will have on your screen if you make a lot of changes, but you'll be able to follow along just fine.

Help! I made too many changes!

Do you fear change, and want your screens to look exactly like our sample screens? No problem, just start with a clean instance.

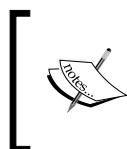
1. At the command line:
`Ctrl + C`
2. This stops the running application. Now go up one directory:
`$ cd ..`
3. Remove the todos application:
`$ rm -R todos`
4. Create the todos example application again:
`$ meteor create --example todos`
5. Change to the new directory, start Meteor, and you're good to go:
`$ cd todos`
`$ meteor`

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 and hot code pushes.

The following is from <http://docs.meteor.com/#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.

Put even more simply, this means that any changes 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 those changes.

Let's see an example.

Changing from todos to items

As we learn the ins and outs of Meteor, we want to build a working application: something useful, and complex enough so that we can experience all the major features of Meteor. We will be building a Lending Library, where we can keep track of what items we have (for example, Mad Men Season 1), organize these items into categories (for example, DVDs), and keep track of the people to whom we have lent the items.

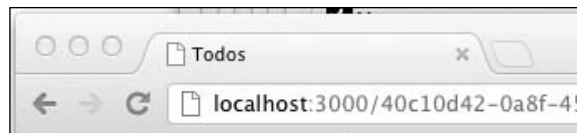
To see the beginnings of this, let's change the lists of *todos* to lists of *items*, and let's change the word *list* to *category*, because that sounds much more awesome.

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

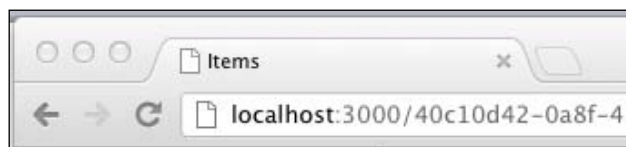
Now, we need to open and edit the `todos.html` file. With your favorite text/code editor, open `~/Documents/Meteor/todos/client/todos.html`.

1. Change `title` in the head section:

```
<head>
  <title>Items</title>
</head>
```
2. Go ahead and save the file, and look at your web browser. The page will automatically refresh, and you'll see the title change from **Todos**:



The title will now display the word **Items**:



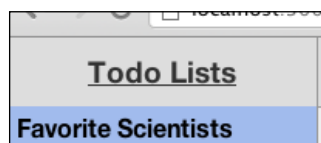
This is Meteor in action! It's monitoring any changes to files, and when it sees that a file has changed, it's telling your browser that a change has been made, and that it should refresh itself to get the latest version.

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 at least clean up the other visible references to `todo` and `list`.

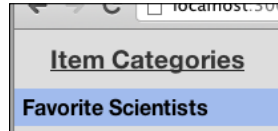
1. Back in your text editor, make the following change to the `<h3>` tag (located approximately around line 20):

```
<template name="lists">
  <h3>Item Categories</h3>
```

Save this change, and you'll see the change reflected in your browser. The left header originally displayed the following text:

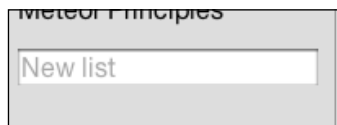


It will now have changed to the following:



2. We need to deal with one more area, and we've successfully turned our todos application into an items application.

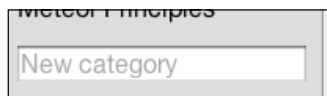
If you noticed, in the bottom of the Categories list, the open box currently says **New list**:



We need to change this to say **New category** instead. Make the following code change on line 39:

```
<div id="createList">
  <input type="text" id="new-list" placeholder="New category" />
</div>
```

3. Save your changes, and check your work:



Summary

Great success! In this chapter, you've successfully installed the Meteor framework, loaded an example application, and made changes to that application, 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 of the elegant features and advantages that come from developing with Meteor.

2

Reactive Programming... It's Alive!

As you learned in *Chapter 1, Setup and Installation*, Meteor operates on a reactive programming model. This means that your client/browser isn't only concerned with displaying data, but it's also listening for changes to that data, so that it can "react" to those changes. These areas of data, where your browser looks for changes, are called **reactive contexts**.

We will start our Lending Library application in earnest, laying the framework for future chapters, and using Meteor's built-in reactive contexts to track and propagate changes to our application to all clients who 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 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 that are 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 last lent it to, and go get it back from them.

And when that same friend says, "are you sure you lent it to me?" we 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 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 Twinkies® habit. Either way, you've got your 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, which we can then expand to fit our needs.

1. Start by navigating to your applications folder. This can be anywhere, but as mentioned, 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
```

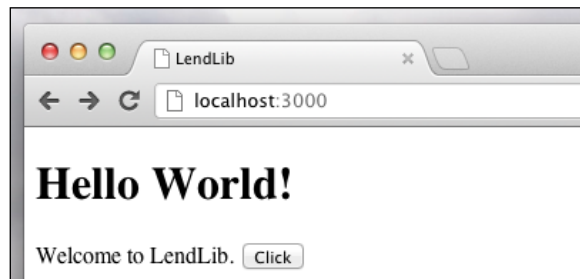
3. As usual, we'll get instructions on how to get the application up and running. Let's go ahead and try that, just to make sure that everything was created properly:

```
$ cd LendLib
```

```
$ meteor
```

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

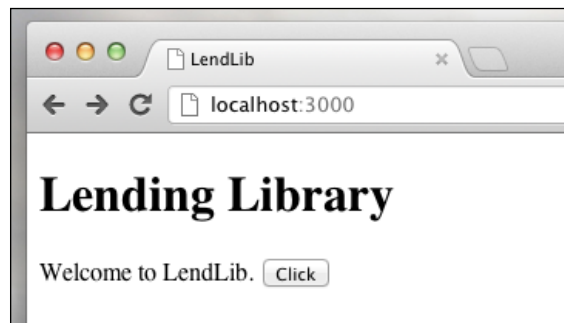
4. Open a browser and navigate to `http://localhost:3000/`. You should see the following screen:



5. Hello World just isn't going to cut it, so let's change that to Lending Library. Open `~/Documents/Meteor/LendLib/LendLib.html` in your favorite editor. Towards the top (line 9 or so), you'll see the template HTML code snippet that's responsible for our greeting. Go ahead and change `Hello World` to `Lending Library`:

```
<template name="hello">
  <h1>Lending Library</h1>
  {{greeting}}
  <input type="button" value="Click" />
</template>
```

6. Save that change, and the page will refresh:



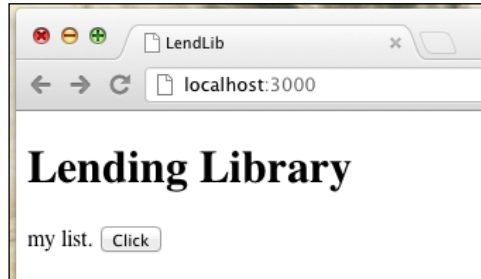
The welcome message wasn't located in the HTML file, however. If you noticed, it's found in a template function called `greeting`:

```
{{greeting}}
```

7. Let's change that as well. Open `~/Documents/Meteor/LendLib/LendLib.js` and make the following change to the `greeting` template function:

```
if (Meteor.isClient) {
  Template.hello.greeting = function () {
    return "my list.";
  };
}
```

8. Save the change, and your page will update:



Creating a collection

Okay, 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!

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, supporting Minimongo (a light version of MongoDB) 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 `LendLib.js`, we want to add the following as the first line, and then save the change:

```
var lists = new Meteor.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.

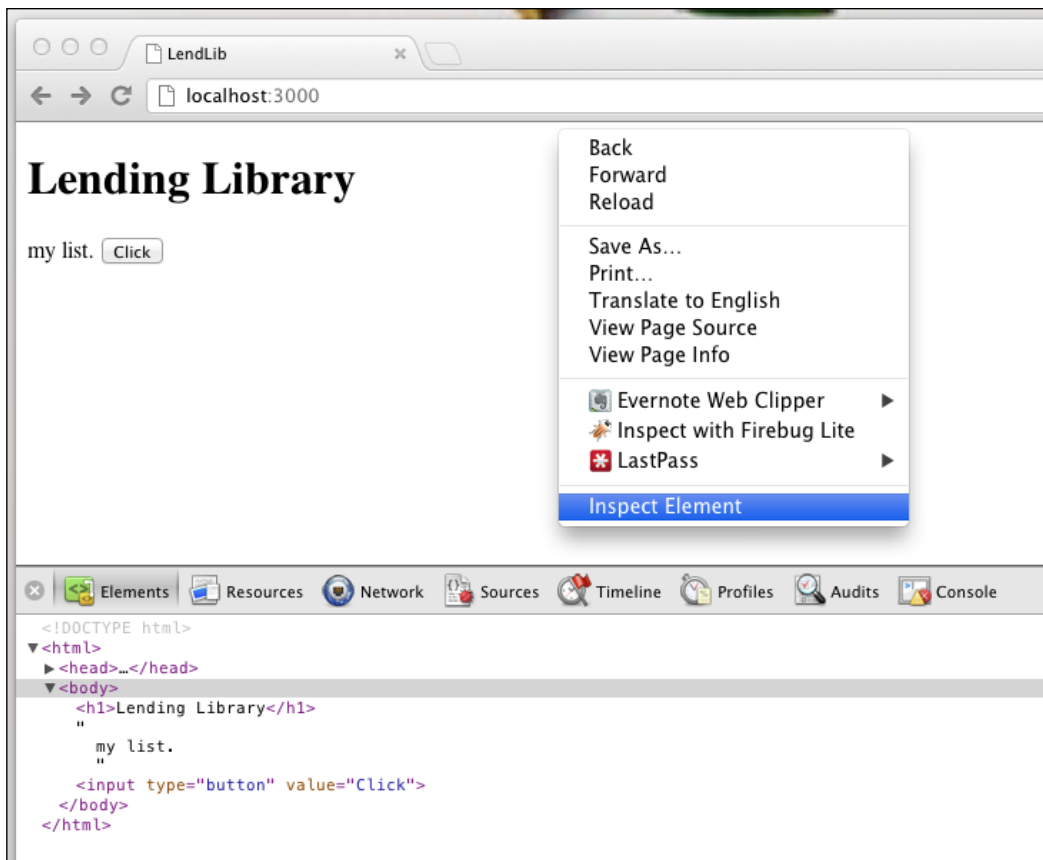
To see this persisted object, we'll need to use the console for our web page.

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.

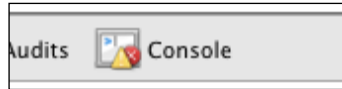
 For a more in-depth tutorial on using the console in Chrome, check out http://developer.chrome.com/extensions/tut_debugging.html.

1. Since we're using Chrome, the console is available by default. 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:

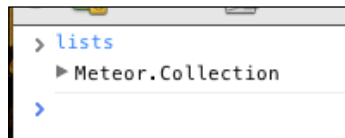


You will now have a blinking cursor, and you're ready to check for our newly-minted collection!

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

```
> lists
```

You should get a returned object that says Meteor Collection:



Adding some data

This means that our changes were accepted, and we have a new persistent collection! It's blank, 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 `f98c3355-18ce-47b0-82cc-142696322a06`), which is Meteor's way of telling you that the item was saved properly. Being the natural skeptics that we are, we're going to check this.

2. Enter the following command:

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

You should get back an object, with an expandable icon next to it.

3. Click on that icon to expand, and you should have the following:

```
> lists.findOne({Category:"DVDs"})
▼ Object
  Category: "DVDs"
  _id: "81219539-86bd-47d4-806a-9010018f23b0"
  ▶ items: Object
  ▶ __proto__: Object
>
```

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

Refresh the web page. Your console will clear, but the categories we entered have been saved in the persistent Meteor Collection, so we can check again to see if they're hanging around.

4. 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 gotten 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 from session to session. Now, let's see how to display this in our page.

Displaying collections in HTML

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



To read more about Live HTML, consult the Meteor documentation at the following URL:
<http://docs.meteor.com/#livehtml>

1. With ~/Documents/Meteor/LendLib/LendLib.html still open, locate the body tag, and add a new **template** declaration:

```
<body>

  {{> hello}}

  <div id="categories-container">

    {{> categories}}

  </div>
</body>
```

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

2. Now, at the very bottom of the page, let's add the skeleton for the categories template partial:

```
<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:

```
<template name="categories">

  <div class="title">my stuff</div>

</template>
```

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

```
<template name="categories">

  <div class="title">my stuff</div>

  <div id="categories">

    </div>

</template>
```

This creates the `categories` `div`, where 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 we get all the categories:

```
<template name="categories">

  <div class="title">my stuff</div>

  <div id="categories">

    {{#each lists}}

      <div class="category">

        {{Category}}

      </div>

    {{/each}}

  </div>

</template>
```

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. Save these changes, and look at the web page:



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

There's one more step we need to complete in order for the categories to show up. Currently, the template we just created isn't pointed 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 that.


In `~/Documents/Meteor/LendLib/LendLib.js` we can see some `Template` functions:

```
Template.hello.greeting = function () {  
  ...  
  
  ...  
  
  Template.hello.events = { ...
```

These code chunks are hooking 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 to hook up our `categories` template.

7. We want to declare to any listening client that the `categories` template has a `lists` collection. We do this by entering the following code, just below the `Template.hello.events = { ... }` code block:

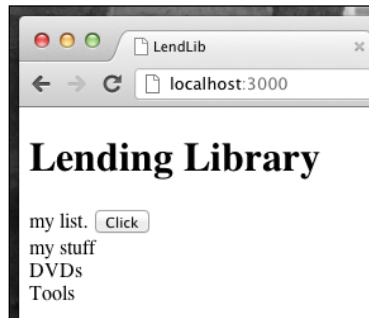
```
Template.hello.events = {  
  ...  
};  
  
Template.categories.lists = function () {  
};
```

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

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

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

This code will find every record in the `lists` collection, and will sort the results by `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 cleanup in our code, and add some CSS to make things more readable:

1. We don't need the greeting anymore. Let's get rid of that. Remove the following highlighted lines from `LendLib.html` and save the page:

```
<body>

  {{> hello}}

  <div id="categories">

    {{> categories}}

  </div>

</body>

<template name="hello">

  <h1>Lending Library</h1>

  {{greeting}}

  <input type="button" value="Click" />

</template>
```

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

We'll want to 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.greeting = function () {

  ...
};

Template.hello.events = {

  ...
};

*/
```

2. Now, let's add the Twitter Bootstrap Framework, which gives us a lot of style without much effort:

1. Using a terminal window, create a `client` folder in `/LendLib/`:

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

2. Download the latest Bootstrap framework at <http://twitter.github.com/bootstrap/assets/bootstrap.zip> and extract the archive into the `~/Documents/Meteor/LendLib/client` folder.


Because Meteor will read and use every file put in to 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 the *Twilight* movies.

3. Navigate to the bootstrap folder:

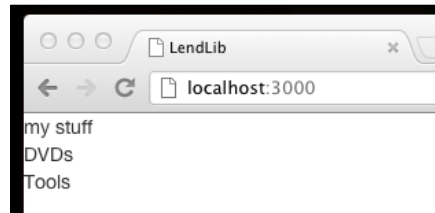
```
$ cd ~/Documents/Meteor/LendLib/client/bootstrap
```

4. Delete the unneeded files:

```
$ rm js/bootstrap.js
$ rm css/bootstrap.css
$ rm css/bootstrap-responsive.css
```

[ If you know what you're doing with Bootstrap, you can just copy the `images`, `min.js`, and `min.css` files over instead of following the previous instructions.]

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



3. Let's quickly make it more distinct and readable. In `LendLib.html`, let's change our header from a `div` tag to an `h2` tag:

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

```
<h2 class="title">my stuff</h2>
```

4. And let's turn categories into a pretty button group:

```
<div id="categories" class="btn-group">
```

```
  {{#each lists}}
```

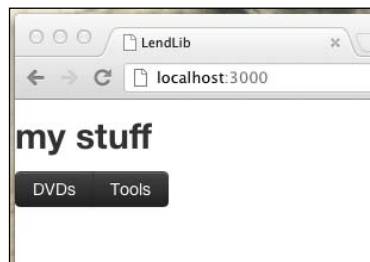
```
    <div class="category btn btn-inverse">
```

```
      {{Category}}
```

```
    </div>
```

```
  {{/each}}
```

This gives us a distinct, clean-looking page:



Creating a reaction

With our basic template and collection created, 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:

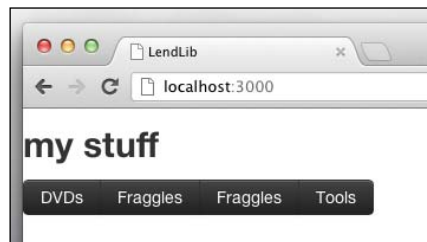
```
> lists.insert({Category:"Fraggles"});
```

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

Let's make a few more changes. Enter the same `Fraggles` command again:

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

Just as before, a new **Fraggles** button instantly appears:



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:

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

This command finds any records where `Category = "Fraggles"` and deletes them.

It would probably be better to add a single collection entry for all our collectibles, so let's do that instead:

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

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.



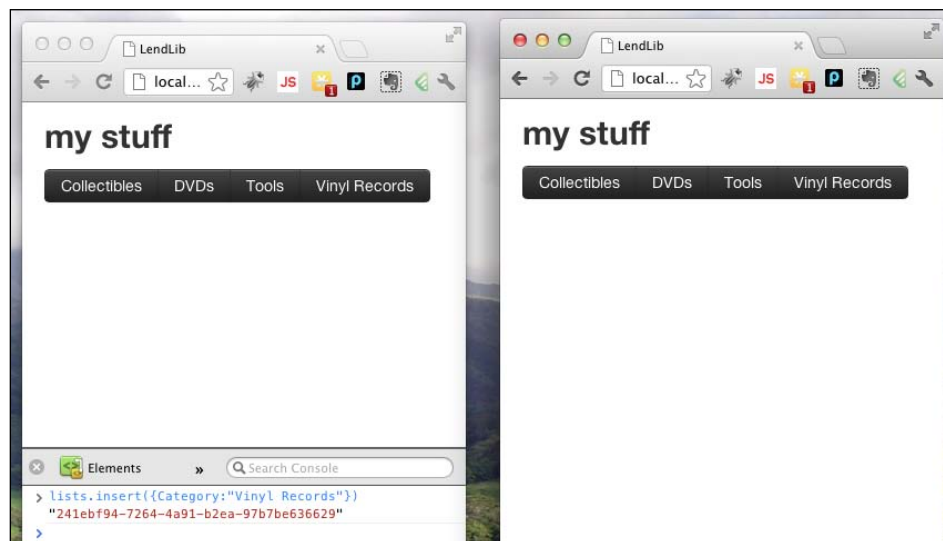
If you really want to get fancy, you can conduct this same experiment with multiple browsers (Firefox, Opera, or Safari) – each session is live and reactive!

You now have two clients open, which are simulating the application being opened by different people, at different locations, with different computers. Meteor's reactive model allows you to treat all clients the same, 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 on browser #1:

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

You will notice that the change propagates to *both* browsers, and again without the page refreshing:



Feel free to make any extra collections, remove or rename, 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, we'll be able to see exactly why this is such an important and disruptive change to web application development.

Summary

In this chapter you've successfully created the framework for your new Meteor application. You've seen firsthand how quickly a new project can be created, and you've created some major database and template functionality, with just a few lines of code. You've seen 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, using the **Model View View-Model (MVVM)** design pattern.

This chapter explains how web applications have changed, why it matters, and how Meteor specifically enables modern web apps through MVVM.

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

- What a modern web application is
- What MVVM means, and how it's different
- How Meteor uses MVVM to create modern web applications
- Templating inside of Meteor – starting to use MVVM

Modern web applications

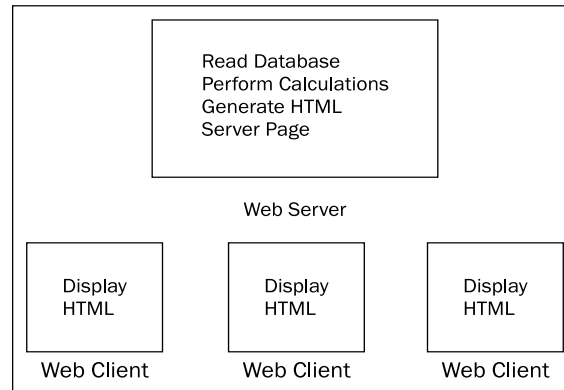
Our world is changing.

With continual advancements in display, computing, and storage capacities, what wasn't possible just a few years ago is now not only possible, but 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 into a readable format (turn a database record into HTML, and so on), and then serve the result to the client, where it's displayed for the user.

In other words, the server does all the work, and the client acts as more of a display, or dumb terminal. The design pattern for this is called...wait for it...the **client/server** design pattern:



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 we think of, when we think of the Internet.

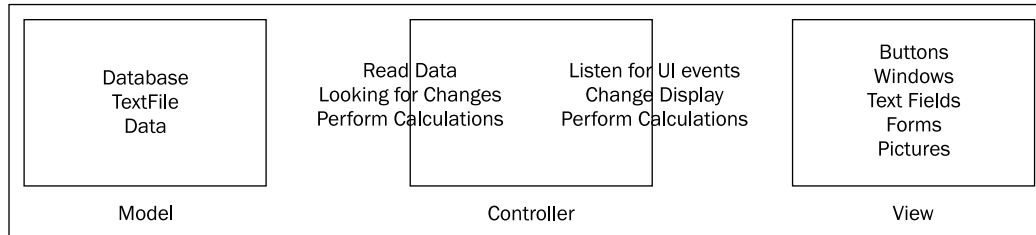
The rise of the machines (MVC)

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 faster and better. Even more and more beefy. At the same time, the Web was coming alive. 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**.

There were many business-oriented smart apps created, but the easiest examples are 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 **controls**) 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 all the data. In the context of a smart app, the model is provided by a server. The client makes requests for the model from the server. Once the client gets the model, it performs actions/logic on this data, and then prepares it to be displayed on the screen. This part of the application (talk to the server, modify the data model, and prep data for display) is called the **controller**. The controller sends commands to the view, which displays the information, and reports back to the controller when something happens on the screen (a button click, for example). The controller receives that feedback, performs logic, and updates the model. Lather, rinse, repeat.

Because web browsers were built to be "dumb clients" the idea of using a browser as a smart app was out of the 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, sometimes you could download it first, but either way, you were running a new type of web app, where the application could talk to the server and share the processing workload.

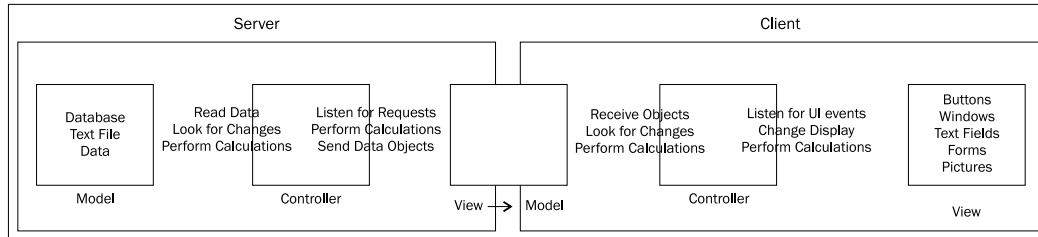
The browser grows up (MVVM)

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 (controller) was performing business logic on the database information (model) through the use of business objects, and then passing that information on to a client application (a "view").

The client was receiving this information 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.

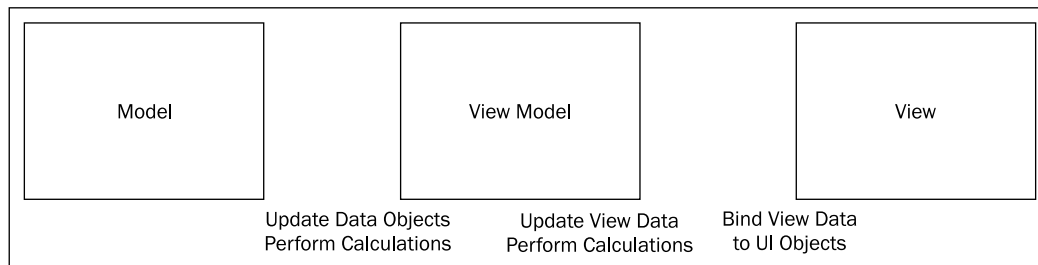
So, the "view" for the server MVC was the "model" for the second MVC.



Then came the thought, "why stop at two?" There was no reason an application couldn't have *multiple* nested MVCs, with each view becoming the model for the next MVC. In fact, on the client side, there's actually a good reason to do so.

Separating actual display logic (such as "this submit button goes here" and "the text area changed value") from the client-side object logic (such as "user can submit this record" and "the phone # has changed") allows a large majority of the code to be reused. The object logic can be ported to another application, and all you have to do is change out the display logic to extend the same model and controller code to a different application or device.

From 2004-2005, this idea was refined and modified for smart apps (called the **presentation model**) by Martin Fowler and Microsoft (called the **Model View View-Model**). While not strictly the same thing as a nested MVC, the MVVM design pattern applied the concept of a nested MVC to the frontend application.



As browser technologies (HTML and JavaScript) matured, it became possible to create smart apps that use the MVVM design pattern directly inside an HTML web page. This pattern makes it possible to run a full-sized application directly from a browser. No more downloading multiple frameworks or separate apps. You can now get the same functionality from visiting a URL as you previously could from buying a packaged product.

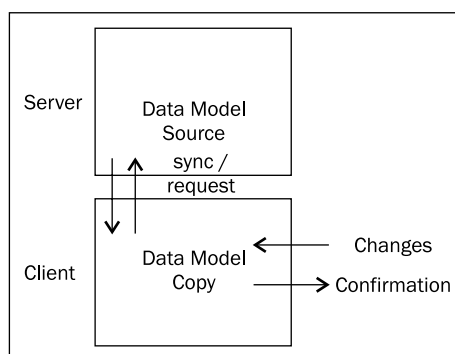
A giant Meteor appears!

Meteor takes the MVVM pattern to the next level. By applying templating through `handlebars.js` (or other template libraries) and using instant updates, it truly enables a web application to act and perform like a complete, robust smart application.

Let's walk through some concepts of how Meteor does this, and then we'll begin to apply this to our Lending Library application.

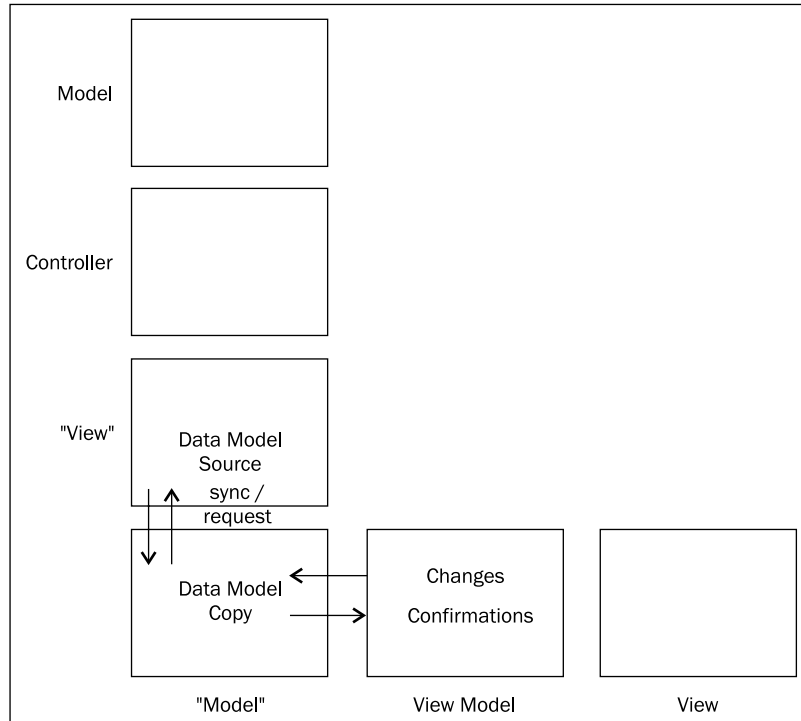
Cached and synchronized data (the model)

Meteor supports a cached-and-synchronized data model that is the same on the client and the server.



When the client notices a change to the data model, it first caches the change locally, and then tries to sync with the server. At the same time, it is listening to changes coming from the server. This allows the client to have a local copy of the data model, so it can send the results of any changes to the screen quickly, without having to wait for the server to respond.

In addition, you'll notice that this is the beginning of the MVVM design pattern, within a nested MVC. In other words, the server publishes data changes, and treats those data changes as the "view" in its own MVC pattern. The client subscribes to those changes, and treats the changes as the "model" in its MVVM pattern.



A code example of this is very simple inside of Meteor (although you can make it more complex and therefore more controlled if you'd like):

```
var lists = new Meteor.Collection("lists");
```

What this one line does is declare 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, and listen to change requests from the client, and update *its* model (its master copy) based on those 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/#publishandsubscribe>.

Templated HTML (the view)

The Meteor client renders HTML through the use of templates.

Templates in HTML are also called **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.

The HTML code has a placeholder. In that placeholder different HTML code will be placed, depending on the value of a variable. If the value of that variable changes, the code in the placeholder will change with it, creating a different view.

Let's look at a very simple data binding – one that you don't technically need Meteor for – to illustrate the point.

In `LendLib.html`, you will see an HTML (Handlebar) template expression:

```
<div id="categories-container">

  {{> categories}}

</div>
```

That expression is a placeholder for an HTML template, found just below it:

```
<template name="categories">

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

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 change the display, change the `h2` tag to an `h4` tag, and save the change:

```
<template name="categories">

  <h4 class="title">my stuff</h4>...
```

You'll see the effect in your browser ("my stuff" become itsy bitsy). That's a template – or view data binding – at work! Change the `h4` back to an `h2` and save the change.

Unless you like the change. No judgment 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 them.

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

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

The first Handlebars 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, make a new `div`) for each item in the `lists` collection. `lists` is the piece of data. `{{#each lists}}` is the placeholder.

Now, inside the `#each lists` expression, there is one more Handlebars expression.

```
{{Category}}
```

Because this is found inside the `#each` expression, `Category` is an implied property of `lists`. 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 `this.Category` here."

Now, if we look in `LendLib.js`, we will see the values behind the templates.

```
Template.categories.lists = function () {
  return lists.find(...
```

Here, Meteor is declaring a template variable named `lists`, found inside a template called `categories`. That variable happens to be a function. That function is returning all the data in the `lists` collection, which we defined previously. Remember this line?

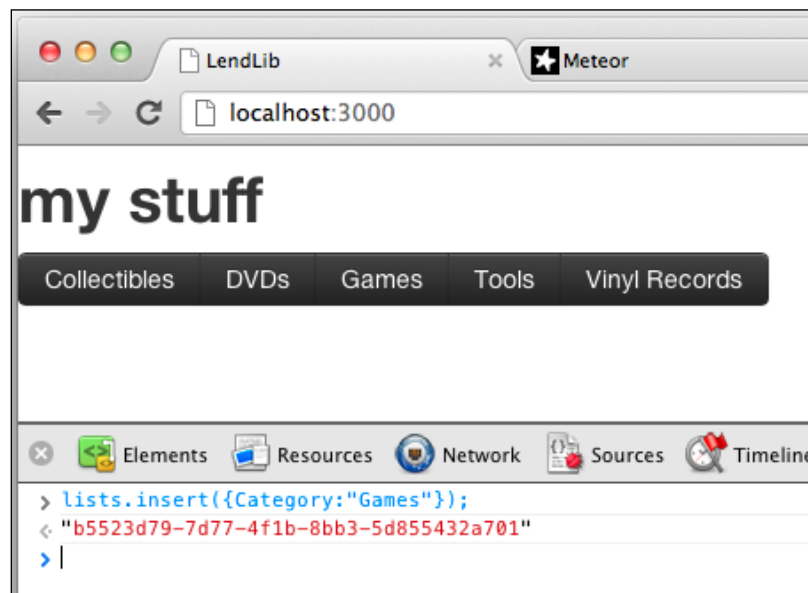
```
var lists = new Meteor.Collection("lists");
```

That `lists` collection is returned by the declared `Template.categories.lists`, so that when there's a change to the `lists` collection, the variable 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 model). The template will see this change, and update the HTML code/placeholder. The `for each` loop will run one additional time, for the new entry in `lists`, and you'll see the following screen:



In regards to the MVVM pattern, the HTML template code is part of the client's view. Any changes to the data are reflected in the browser automatically.

Meteor's client code (the View-Model)

As discussed in the preceding section, `LendLib.js` contains the template variables, linking the client's model to the HTML page, which is the client's view. Any logic that happens inside of `LendLib.js` as a reaction to changes from either the view or the model is part of the View-Model.

The View-Model is responsible for tracking changes to the model and presenting those changes in such a way that the view will pick up the changes. It's also responsible for listening to changes coming from the view.

By changes, we don't mean a button click or text being entered. Instead, we mean a change to a template value. A declared template is the View-Model, or the *model for the view*.

That means that the client controller has its model (the data from the server) and it knows what to do with that model, and the view has its model (a template) and it knows how to display that model.

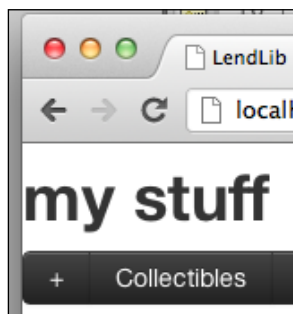
Let's create some templates

We'll now see a real-life example of the MVVM design pattern, 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 -t term solution. Let's make it so we can do that on the page instead.

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-inverse" id="btnNewCat">&plus;</div>
  {{#each lists}}
```

This will add a plus button to the page.



Now, we'll want to change out that button for a text field if we click on it. So let's build that functionality using the MVVM pattern, and make it based on the value of a variable in the template.

Add the following lines of code:

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

The first line `{{#if new_cat}}` checks to see if `new_cat` is true or false. If it's false, the `{{else}}` section triggers, and it means we haven't yet indicated 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 be false, and so the display won't change. Now let's add the HTML code to display, if we want to add a new category:

```
<div id="categories" class="btn-group">
  {{#if new_cat}}
    <div class="category">
      <input type="text" id="add-category" value="" />
    </div>
  {{else}}
    <div class="category btn btn-inverse"
      id="btnNewCat">&plus;</div>
  {{/if}}
  {{#each lists}}
```

Here 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, so for now it's hidden. So how do we make `new_cat` equal true?

Save your changes if you haven't already, and open `LendingLib.js`. First, we'll declare a `Session` variable, just below our `lists` template declaration.

```
Template.categories.lists = function () {
  return lists.find({}, {sort: {Category: 1}});
};
// We are declaring the 'adding_category' flag
Session.set('adding_category', false);
```

Now, we declare the new template variable `new_cat`, which will be a function returning the value of `adding_category`:

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

```
// This returns true if adding_category has been assigned a value
//of true

Template.categories.new_cat = function () {
  return Session.equals('adding_category',true);
};
```

Save these changes, and you'll see that nothing has changed. Ta-daaa!

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 that now.

First, we'll declare our click event, which will change the value in our Session variable.

```
Template.categories.new_cat = function () {
  return Session.equals('adding_category',true);
};
Template.categories.events({

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

    Session.set('adding_category', true);

    Meteor.flush();
    focusText(t.find("#add-category"));
  }
});
```

Let's take a look at the following line:

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

This line is declaring that there will be events found in the category template.

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

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

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

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

Meteor.flush();
focusText(t.find("#add-category"));
```

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

One last thing to do, and that is to quickly add the helper function `focusText()`. Just before the closing tag for the `if (Meteor.isClient)` function, add the following code:

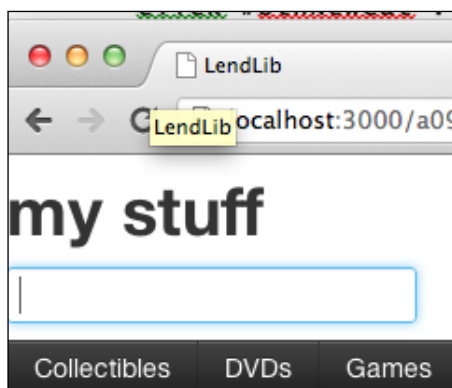
```
/////Generic Helper Functions/////

//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'll see the following 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*.

That variable belongs to the View-Model. That is to say that if we change the value of the variable (like we do with the click event), then the view automatically updates. We've just completed an MVVM pattern inside a Meteor application!

To really bring this home, let's add a change to the `lists` collection (also part of the View-Model, 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 that happens, we want to have a category added, based on what the user typed. First, let's declare the event handler. Just after the `click` event for `#btnNewCat`, let's add another event handler:

```
focusText(t.find("#add-category"));
},
'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 `" , "` at the end of the click function, and then added the `keyup` event handler.

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

This line checks to see if we hit the *Enter*/return key.

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

This checks to see if the input field has any value in it.

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

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

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

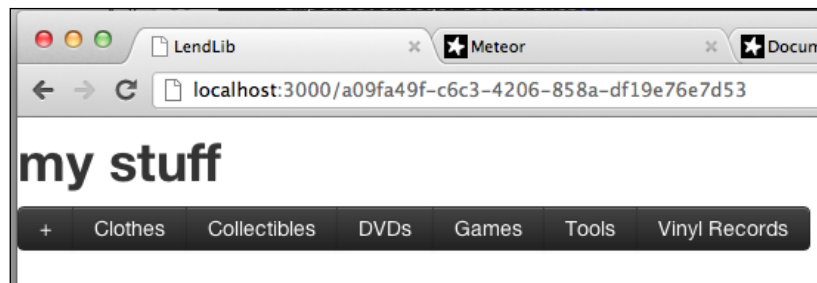
Then we want to hide the input box, which we can do by simply modifying the value of `adding_category`.

One more thing to add, and we're all done. If we click away from the input box, we want to hide it, and bring back the plus button. We already know how to do that inside the MVVM pattern by now, so let's add a quick function that changes the value of `adding_category`. Add one more comma after the `keyup` event handler, and insert the following event handler:

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



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

Summary

In this chapter you've learned about the history of web applications, and seen how we've moved from a traditional client/server model to a full-fledged MVVM design pattern. You've seen how Meteor uses templates and synchronized data to make things very easy to manage, providing a clean separation between our view, our view logic, and our data. Lastly, you've added more to the Lending Library, making a button to add categories, and you've done it all using changes to the View-Model, rather than directly editing the HTML. 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, creating a working application using the MVVM design pattern. This chapter will take us through the template system in depth, and show us how to implement display logic, add design considerations to a model (create the View-Model), and take care of data flow.

In this chapter, you will complete the following tasks:

- Completing the Lending Library core functionality
- Creating multiple templates and template logic
- Adding, deleting, and updating entries in the data model
- Seeing reactivity in action and using 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 when we click on a category we should get a **list of items**.

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

```
<body>
  <div id="lendlib">
    <div id="categories-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 of the `div` called `lendlib`. This is for stylistic purposes, so that our `list` will more or less line up with the `categories` template.
2. We added a `div` with `id="list"` just below it, and 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">
  <ul id="lending_list">
    {{#each items}}
      <li class="lending_item alert">
        <button type="button" class="close delete_item"
          id="{{Name}}">x</button>

        {{Name}}

        {{#if lendeer_editing}}
          <input type="text" id="edit_lendeer" class="span2
            pull-right" value=""/>
        {{else}}
          <div class="lendeer pull-right label {{LendClass}}">
            {{Lendeer}}</div>
        {{/if}}
      </li>
    {{/each}}
    {{#if list_selected}}
      <li class="alert-success" id="btnAddItem">&plus;
        {{#if list_adding}}
          <input class="span4" id="item_to_add" size="32"
            type="text">
        {{/if}}
      </li>
    {{/if}}
  </ul>
</template>
```

Let's go through this step by step, so we understand what each line does:

```
<template name="list">
  <ul id="lending_list">
    {{#each items}}
  ...
```

Here we declare the HTML `<template>` with the name "list", to match the call to the list template we made in the body. We create an unordered list `` and give it an `id` so 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.

```
    <li class="lending_item alert">
      <button type="button" class="close delete_item"
        id="{{Name}}">x</button>
      {{Name}}
```

Now, under the `each` statement, we create an `` element and give it two class names. The `lending_item` class name is added so that we can refer to it in our View-Model (Meteor template code). The `alert` class name is for Bootstrap, so it will display all nice and pretty.

Next, we create a `button` that we can use, should we choose to delete the item. Notice, that we give it an ID `id="{{Name}}"`. This will be read from the `items` View-Model, and will make our jobs much easier in the future, should we want to delete the item from our `items` collection. There are also two class names on this one. `close` is added for Bootstrap, and `delete_item` is added so that we can refer to it in the View-Model 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 has to do with when we want to edit who is borrowing our item, or the **lender**:

```
    {{#if lender_editing}}
      <input type="text" id="edit_lender" class="span2
        pull-right" value=""/>
    {{else}}
      <div class="lender pull-right label {{LenderClass}}">
        {{Lender}}</div>
    {{/if}}
  </li>
{{/each}}
```

We are first using an `if` statement to see if 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 `true`. If this is the case, we need a textbox, hence the inclusion of 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, we'll want to maybe change the color or something, but we still want it displayed. So, we create a Bootstrap-styled `label` in the form of a `<div>` element.

At the end of the class 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 as "lent out." If it's green, it's free, if it's red, someone has borrowed it. And what CSS class name is 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 is also a property in `LendLib.js`, as the `item.Lendee` property, and it will either display the name of the `lendee`, or "free" if no one has borrowed it.

We then have the ending `` 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 of the `each` statement, so it's not specific to items. This `if` statement displays either a light-green bar with a `+` sign, or a textbox in the form of an `<input>` element, so that we can add items to our list:

```
    {{#if list_selected}}
      <li class="alert-success" id="btnAddItem">&plus;
        {{#if list_adding}}
          <input class="span4" id="item_to_add" size="32"
            type="text">
        {{/if}}
      </li>
    {{/if}}
  </ul>
</template>
```

So we see our first `if`, which is conditioned on whether we're even displaying any list items. If we are, that means we have a list selected. Or rather, we are in the `list_selected` mode. Keeping track of this is part of the View-Model's job, so `Template.list.list_selected` is found inside `LendLib.js`.

We then create an `` element, style it green with the Bootstrap `alert-success` class, and add the `+` sign.

Next is our nested (second) `if`. This one is checking to see if we are adding to the list of items. If we are, we are in `list_adding` mode, so we'll display a textbox in the form of an `<input>` element. If not, we'll just leave the pretty light-green box with just the `+` sign in it.

Finally, we end our nested `if`, our ``, our parent `if`, our ``, and our `</template>`.

Gluing it all together

The View-Model (MVVM) or controller (MVC) or presenter (MVP) is considered the glue of an MV* application model. That's because it "glues" together all the view items, such as buttons or textboxes, to the model.

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 keep the model and the view together. Someone else invented the term, not us, so let's continue without the Judgy McJudgerson viewpoint, mmmk?

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

Our items View-Model

In our data model that we created in *Chapter 2, Reactive Programming...It's Alive!*, we added some sample items when we created a couple of the 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 in there. 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 the view knows what to do with it. Specifically, we need to:

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

So, we're going to get the items collection from the currently selected list, add the Lendeer and LendClass properties, and make the template available.

Open ~/Documents/Meteor/LendLib/LendLib.js.

Immediately after the closing } curly bracket for the function focusText (...), add the following code:

```
}; //<-----This is the end tag for focusText() -----

Template.list.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 d = cats.items[i];  d.Lendeer = d.LentTo ? d.LentTo :
          "free"; d.LendClass = d.LentTo ?
          "label-important" : "label-success";
      }
      return cats.items;
    }
  }
};
```

We'll walk through this step-by-step.

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

Here we're declaring the Template.list.items function, and checking to see if 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 View-Model at work. It is reading the contents of a given category, and incorporating the current state of the UI, according to whether the user has selected a list. This is the glue at work.

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 we do, and that 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 d = cats.items[i];
    d.Lendee = d.LentTo ? d.LentTo : "free";
    d.LendClass = d.LentTo ? "label-important" : "label-success";
  };
  return cats.items;
};
};
```

First, we check to see if `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 `d = cats.item[i]`.

Now we add the `Lendee` property, checking to see if the item is lent to anyone with the `LentTo` property. If it isn't (if `LentTo` doesn't exist), we'll assign the string `"free"` 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 model. That's because they're *not* part of the model. They're used by the view, so we'll only make them available via the View-Model template.

Additional view states

We now need to declare the templates for all the different view states. That is, we need to add properties to the View-Model/session 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:

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

Add the following code, just below our newly-created items template/function in `LendLib.js`:

```
        return cats.items;
    };
};
}; // <---- ending bracket for Template.list.items function ----

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

Template.categories.list_status = function(){
    if (Session.equals('current_list',this._id))
        return "";
    else
        return " btn-inverse";
};

Template.list.list_adding = function(){
```

```

    return (Session.equals('list_adding',true));
};

Template.list.lendee_editing = function(){
    return (Session.equals('lendee_input',this.Name));
};

```

Let's go over each of these template functions.

```

Template.list.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, `Session.equals('current_list',null)` will return true. So we need to check both cases, unfortunately.

```

Template.categories.list_status = function(){
    if (Session.equals('current_list',this._id))
        return "";
    else
        return "btn-inverse";
};

```

`list_status` is used to tell the category button whether it should show as selected. The easiest way to do that is via a CSS class. Bootstrap uses `btn-inverse` to show white-on-black text, but that's our default button look and feel. So, because we are using the exact opposite color scheme, we'll use Bootstrap's regular black-on-white appearance for the selected category.

In other words, for the `current_list`, we'll return "" (default button look and feel). For all the other lists/categories, we'll return `"btn-inverse"`, to change the CSS style.

You may be wondering about `this._id`. `this`, in this instance, refers to the MongoDB record (technically 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.

```

Template.list.list_adding = function(){
    return (Session.equals('list_adding',true));
}

```

This one's really straightforward. If the `Session` variable `list_adding` is true, we're adding to the list. If it isn't, we're not.

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

To check and see if we should be in lendee editing mode, we'll check the `Session` variable `lendee_input` and see a if it has a value, and b if that value is the `Name` of the item we just clicked on. 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?

```
<li class="lending_item alert">
  <button type="button" class="close delete_item"
    id="{{Name}}">x</button>
  {{Name}}

  {{#if lendee_editing}}
```

Notice how we use `lendee_editing` in the `if` statement, right after we use `{{Name}}`. This is showing us the context. `this.Name` 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 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 templates `{{list_status}}` and `{{_id}}` are used, and why the context for `this._id` suddenly makes sense.

Locate the following lines in `LendLib.html` (should be around line 27 or so):

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

{{/each}}
```

And change it 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 (our view), we'll do it in the template declarations (our View-Model).

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 know whether we have a selected list (`list_selected`) and what list that is (`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:

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



Don't forget the comma (,) right after the `'focusout ... function(e,t) { ... }` code block.

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

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

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

Template.list.items = function () {
    ...
```

Yes, you could have put this anywhere. Yes, you could have just put a generic function inline 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, put the following code:

```
Template.list.lendee_editing = function() {
  ...
}

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

```

We declare `Template.lists.events`, and enumerate our events. The first one is for adding 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 `Meteor.flush()` to ensure the UI is cleaned up, and then, as a courtesy to our user, we focus the textbox (named `item_to_add`) so 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 `Meteor.flush()` does in the Meteor documentation at http://docs.meteor.com/#meteor_flush.


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

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

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

The trigger creates a situation so that if we change our minds about adding an item, all we have to do is click away from it. If the textbox `item_to_add` 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  button in our HTML `lists` template? That button belongs to the CSS class `delete_item`, and when the user clicks on it, we will call the `removeItem()` function, passing the `current_list`, and the `id` from the HTML element that was clicked, which happens to be the item `Name` (we added `id="{{Name}}"` on line 39 of `LendingLib.html`).

```
'click .lendee' : function(e,t){
  Session.set('lendee_input', this.Name);
  Meteor.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 CSS class `lendee` in `LendingLib.html`. We are now declaring that whenever one of those `<div>` elements is clicked, we'll perform actions very similar to 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 (`Meteor.flush()`).
3. Set focus on the textbox (`focusText()`).

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`. Keep in mind that updating the data model, and setting a `Session` variable will cause the templates to refresh (reactive programming again).

If we instead decide we don't like the changes, we're going to hit the *Esc* key (`e.which === 27`) in which 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 take care of making our app pretty (not just yet, though), and we need to create the `addItem()`, `removeItem()` and `updateLendee()` functions that we just made references to in the `events` section.

So let's get to work! In `LendingLib.js`, in the `helpers` section (just above `Template.lists.items`, on line 68 or so) let's add our `addItem()` function:

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

Template.list.items = function () {
  ...
```


`addItem()` takes two arguments: `list_id` and `item_name`.

`list_id` is used in the selector (the query) portion of the update statement, and `item_name` contains the value to be added to the `item.Name` property of the new item to be added.

But first, we need to check to make sure 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're telling MongoDB to find us the record with an `_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 a change was made to the data model. Stop for a second and think about that. 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 removeItem(list_id,item_name){
  if (!item_name&&!list_id)
    return;
  lists.update({_id:list_id},
    {$pull:{items:{Name:item_name}}});
}
```

```
Template.list.items = function () {
  ...
}
```

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 is all we need. The data model and the UI will both update automatically.

Now we tackle `updateLendee()`, which is a little more complex, but only because Meteor actually uses `minimongo`, which is a pared-down version of MongoDB, and lacks cursor variable support. That means that instead of using something like `items.$.Name`, where `$` is the cursor location, we instead need to go through the `items[]` array, update values, and then call an update where we replace the entire `items[]` array with our updated one. Here's how we do it for `LendLib.js`:

```
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)
      {
        l.items[i].LentTo = lendee_name;
      }
    }
    lists.update({"_id":list_id},{ $set:{"items":l.items}});
  }
};
```

```
Template.list.items = function () {
  ...
```

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 that), and then update the `LentTo` property with the value in `lendee_name`.

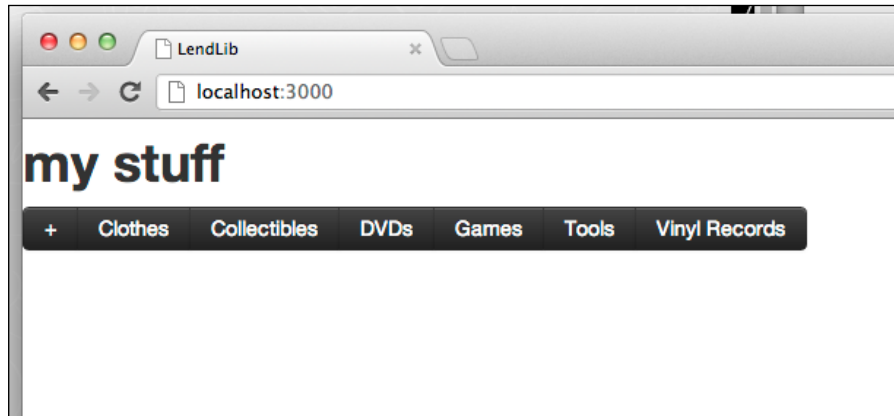
To save some typing, we declare variable `l`, using the MongoDB `findOne()` function. This takes a two-part selector statement: `_id:list_id` and `"items.Name":item_name`. This selector is basically saying "find me *one* record where the `_id == list_id` and the `items[]` array has a record where `Name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

If `l` has a value, and `l` 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 set the `LentTo` property to `lendee_name`.

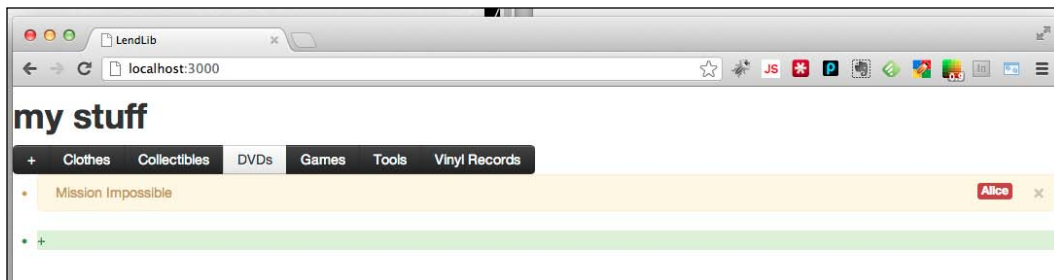
Once we're done with the `for` loop, we'll call the MongoDB `update()` function, and use the `$set` action to replace the old `items[]` array with the new one that is `{"items":l.items}`. Automatic updates happen again, and our UI (view) and data document (model) are in sync again.

Style updates

Right now, you can run the application. It will be a visual train wreck, because we haven't set any CSS styles yet, but let's go ahead and do that real quick. We'd all stare at a train wreck anyway, just admit it! Make sure your application is running (> meteor in the console) 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, but that's because we are using Bootstrap. Let's go ahead and fix the remaining UI issues.

First, we have one final change in `LendLib.js`. Change the `focusText()` function (located at about line 55):

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

It should now be:

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

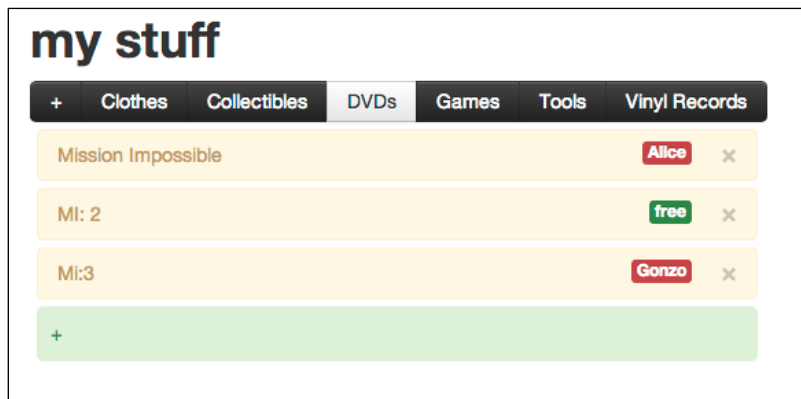
This change just makes it so that when we go to edit something with a value already in it (like the lender), the value will transfer to the textbox. This makes it easier for the user to see who the current lender is. The `val ? val : ""` conditional statement is necessary, because if `val` isn't passed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

We now want to update the CSS for all the other visual idiosyncrasies. We won't be going over the CSS here, as there are probably much better ways to deal with it, and we're not experts on CSS. So just add the following to `~/Documents/Meteor/LendLib/LendLib.css`, and save the changes:

```
/* CSS declarations go here */
#lendlib{
  width:535px;
  margin:0 auto;
}
#categorybuttons{
  width:100%;
}
#lending_list{
  list-style:none;
  margin:0;
  padding:0;
}
#lending_list li{
  list-style:none;
  margin:5px;
}
#lending_list li.lending_item:hover{
  background-color:#fc6;
  color:#630;
}
#lending_item{
  vertical-align:middle;
}
#add_item{
```

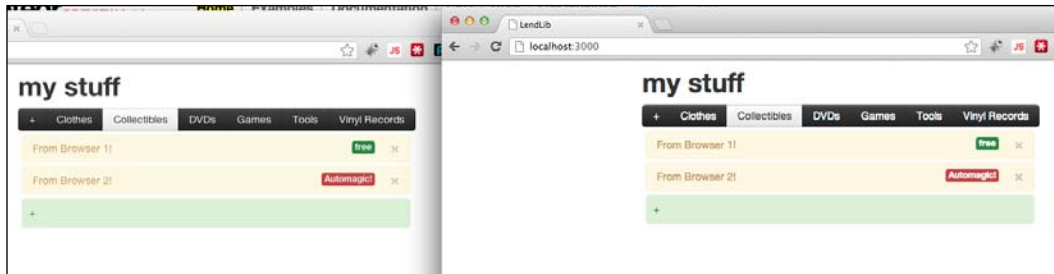
```
padding-left:5px;
}
#btnAddItem{
padding:10px;
border-radius:5px;
}
#btnAddItem:hover{
background-color:#B7F099;
}
#edit_lendee{
padding-top:0;
margin-top:-2px;
}
```

If you kept `http://localhost:3000` open, your browser will automatically refresh. If you didn't, open it backup (make sure Meteor is running) and observe the results:



Don't forget to test all your new functionality! Add new categories, add items to a list, change the lendeer, delete the lendeer, delete the items, and so on, and just get a feel for how fast and clean the updates to the model are.

Now, open two browsers, both pointing to `http://localhost:3000`. You'll notice that changes you make in one browser are reflected in the other one as well! Just as before, Meteor is taking care of the data model synching between the client and server, and any change on one client is propagated to the other clients by the server.



Once you get as many items and lists created as you'd like, move on to the next chapter.

Summary

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

We do want to go over the methodology, however, and make sure we have a solid understanding of how Meteor handles data, so 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 a version of MongoDB (minimongo), to store all of the data from your models. 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.

But why not use a relational database

Traditionally, 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 possibility for deep analytics.

That type of exactness, however, isn't always necessary. In the case of our Lending Library, 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.

That's 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. 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 that 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 those documents less efficient. But that's okay, because our primary concern is with ease of coding and development speed, not efficiency.


In addition, because 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 or even more so 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 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 those 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, and atomic operations, but at its heart it is a document-oriented storage solution.

[ To learn more about MongoDB, visit the official site <http://www.mongodb.org>.]

Using simple commands, we can see what records (what documents) are available, convert those records into JavaScript objects, and then save those changed objects. Think of MongoDB records as 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 you'll need to learn if you want to consider yourself an expert 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. It 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. Just like 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 able to be "opened", changed, and saved.

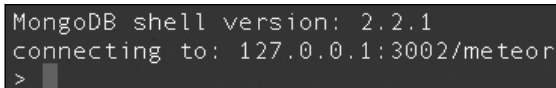
Using direct commands

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

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). Next, open a browser to `http://localhost:3000`.
2. Now, you will want 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:



```
MongoDB shell version: 2.2.1
connecting to: 127.0.0.1:3002/meteor
>
```

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
2. You'll get a list of commands, and a brief explanation of what each one does. One in particular will show us even *more* commands we can use: `db.help()`. This will give us a list of database-related commands. 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 on MongoDB. You only need to know a few, but having a look around never hurt anybody, so let's proceed.

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

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

4. In the response you will find the name of your Lending Library collection: `lists`. Let's go ahead and take a look at the `lists` collection. Enter the following:

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

5. 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 variable `myLists`. Therefore, we can run commands in the terminal window just like we would inside our Meteor code.

6. Let's get the `Clothes` list, which currently doesn't have any items in it, but still exists. Enter the following command:

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

This will return some very basic JSON. If you look closely, you'll be able to see the empty items array, represented as `"items" : []`. You'll also notice an `_id` key-value, with a long number next to it, similar to the following:

```
"_id" : "520e4f45-8469-47b9-8621-b41e60723de0",
```

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

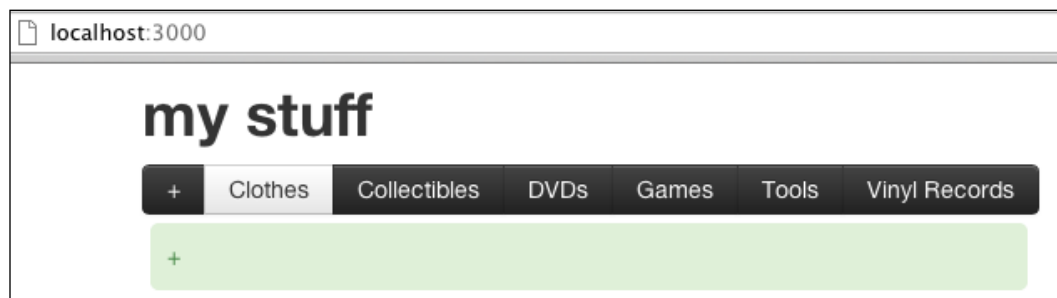
If you look inside `~/Documents/Meteor/LendLib/LendLib.js`, you'll see the following function for adding an item to a list:

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

Notice 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 the category as the selector (`{Category: "DVDs"}`), you would be taking action on both category lists. If, instead, you use the `_id`, it will only update the unique document with that matching `_id`.

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

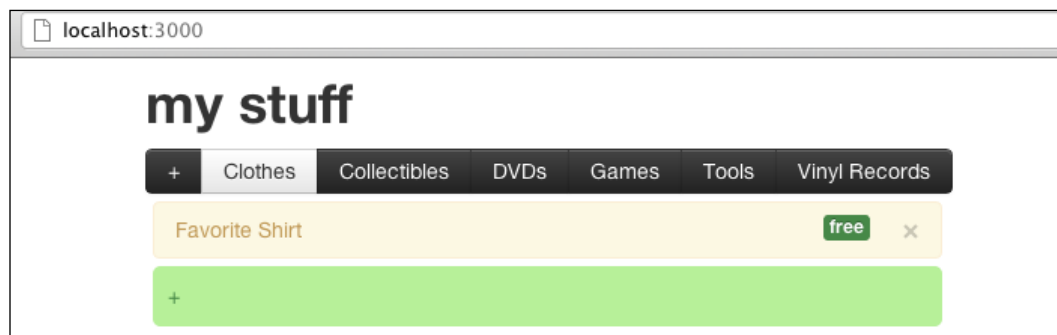
Take note of what the `Clothes` list currently looks like in our browser.



Let's go ahead and add our favorite shirt to the `Clothes` list. We'll do this directly, in the terminal window. 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 the `Name: "Favorite Shirt"`. It will take a few seconds for Meteor to update, but you will soon see your favorite shirt now added to the list.



If you re-run the `Clothes` assignment command `Clothes = myLists.findOne({Category:"Clothes"})` you will now see that the `items` array has an entry for your favorite shirt.

We can just as easily update or remove an item, using the `.update()` function with different arguments (`$pull` for removing, `$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 we can implement directly, let's revisit some of our `LendLib.js` code, and discuss the reactive code that tracks changes to our collection.

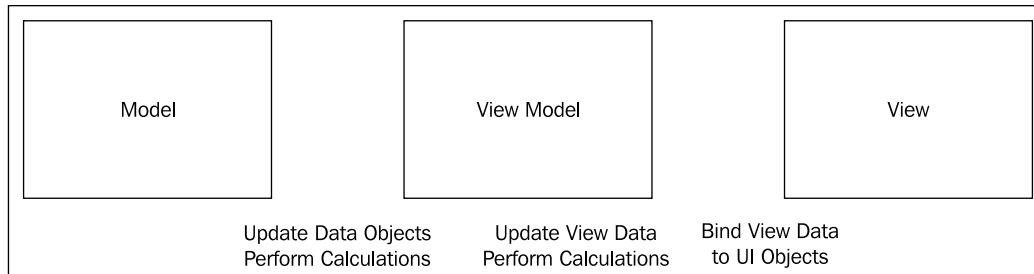
Broadcasting changes

Using a publish/subscribe model, Meteor is constantly looking for changes to collections and to `Session` variables. When changes are made, a change event is broadcast (or published). Callback functions are listening (or subscribed) to the events being broadcast, and the code in a function is activated when the specific event it's subscribed to is published. Alternatively, the data model can be directly bound to portions of the HTML/Handlebars templates, so that when a change occurs, the HTML is re-rendered.

Published events

So, when is an event published? As mentioned previously, events are broadcast when there's a change in the model. In other words, when a collection or a variable is modified, Meteor publishes the appropriate change event. If a document is added to a collection, an event is triggered. If a document already in a collection is modified and then saved back into the collection, an event is triggered. Finally, if a `Session` variable is changed, an event is triggered. Functions and templates are listening (subscribed) to the specific events, and will process the change in the data appropriately.

If you recall from *Chapter 3, Why Meteor Rocks!*, this is the Model View View-Model pattern at work. In a reactive context, functions and templates react to changes in the model. In turn, actions from the view will create changes to the model, through the View-Model logic:



Meteor's MVVM is a clean, concise development pattern:

1. Set up subscribers to model changes (model = collections, documents, and `Session` variables).
2. Create logic to handle view events (view events = button click, text input, and so on).
3. Change the model, when the logic calls for it (changes = published events).

Around and around it goes, with a click on a button causing a model change, which then triggers an event, listened to by a template. This updates the view based on the model change. Lather, rinse, repeat.

Configuring publishers

Up to this point, we have been using `autopublish`. Meaning, we haven't had to code specific publish events for any events or collections. This is great for testing, but we want to have a bit more control over what events and documents get published, so we 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 that can slow things down, or it can expose data we don't want to have exposed.

Turning off autopublish

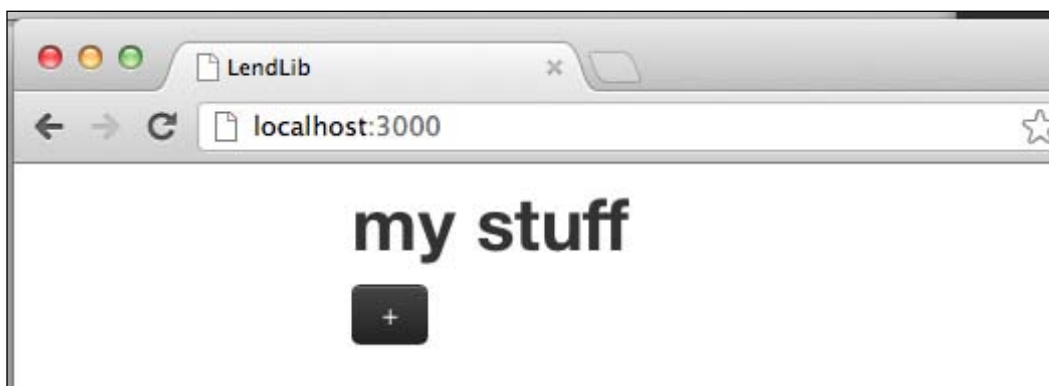
The time has come to turn off `autopublish`. Temporarily stop your Meteor application (if it's still running) by opening the terminal window you ran the `meteor` command from. You can stop it by pressing `Ctrl + C`. Once it's stopped, enter the following command:

```
> meteor remove autopublish
```

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

 It's considered a best practice to remove `autopublish` from your project. `autopublish` 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 made your application do nothing! Congratulations! You can see your amazing progress by starting up your Meteor service again (enter the `meteor` command and press `Enter`), and opening/navigating to `http://localhost:3000`. You will see the following screenshot:



The categories/lists are gone! You can even check in the console if you'd like. Enter the following command:

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


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

```
> lists.find().count()
< 0
> |
```

What gives? Well, it's pretty simple, actually. Because we removed the `autopublish` library, the server is no longer broadcasting any changes to our model.

Why again did we do this? What's the purpose of breaking our application? Ah! Because we want to make our application more efficient. Instead of getting every record automatically, we're going to instead just get the records 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.publish("Categories", function() {
  return lists.find({}, {fields:{Category:1}});
});
```

This tells the server to publish a "Categories" event. It 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/publish an event.

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

This snippet of JSON is telling the selector that we want to get the `Category` field (`1` = true, `0` = false). Because that 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 any fields set to `0` (false), Meteor would assume that you want to *include* all the other fields you didn't mention.



For more information on the `find()` function, consult the MongoDB documentation at <http://www.mongodb.org/display/DOCS/Advanced+Queries>.

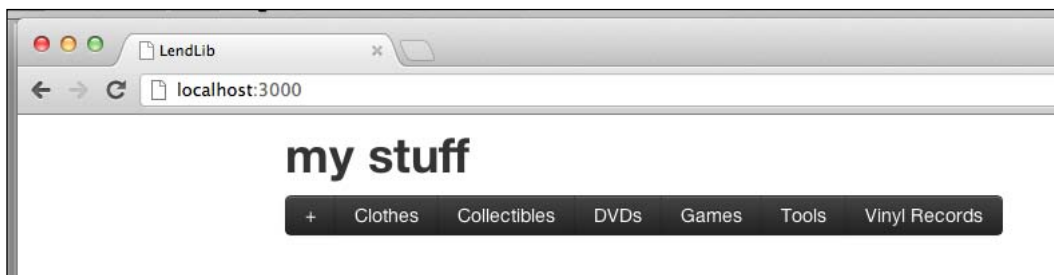
So, if you save this change, your browser will refresh and... nothing happens to the display!

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` event channel. So we need to add a subscriber event that is tuned in to the `Categories` channel.

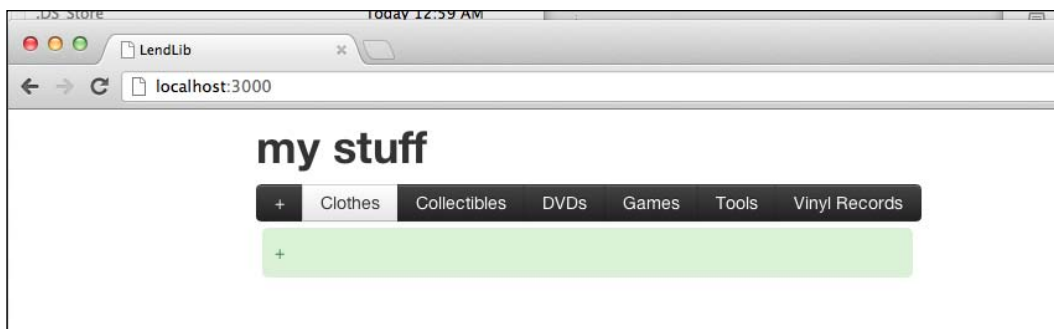
Inside the `if (Meteor.isClient)` function, at the very top, just inside the opening bracket, enter the following line of code:

```
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.

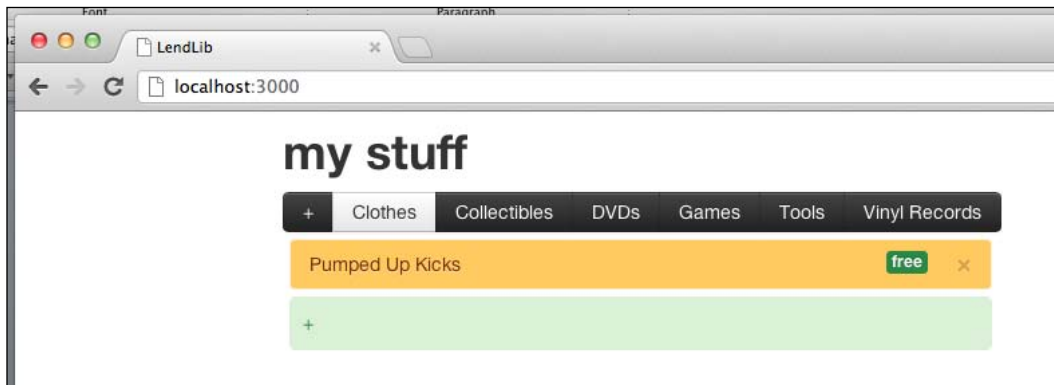


Our favorite shirt is missing! As you probably figured out by now, this is because the `publish` event we set up was very specific. The only field in the `Categories` channel being published is the `Category` field. All the other fields, including our `items` (and therefore our favorite shirt) are not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

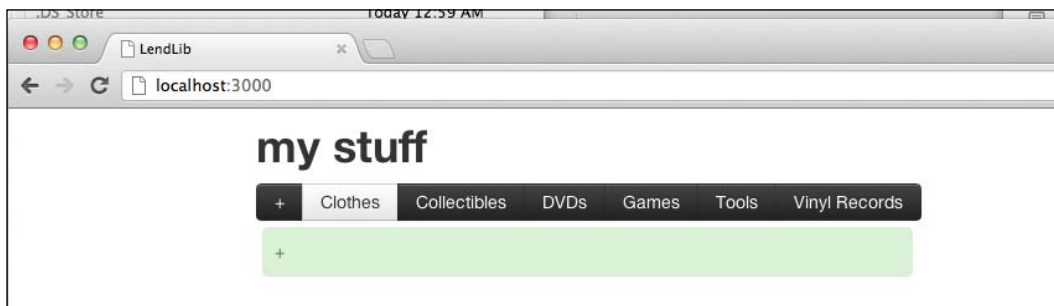
Let's double-check this. Click on the **+** button inside of the **Clothes** category in your browser, type in `Red Hooded Sweatshirt`, and press `Enter`. The new entry will appear for a split second, and then it will disappear. This is because of local caching and server sync.

When you enter the new item, the local cache contains a copy. That 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.

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`). Because 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 back up. Your client will sync with the server, and poof! your item is 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.cs`, just below our first `Meteor.publish()` function 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 provide the variable `category_id`, so that our `find()` function returns a leaner recordset.

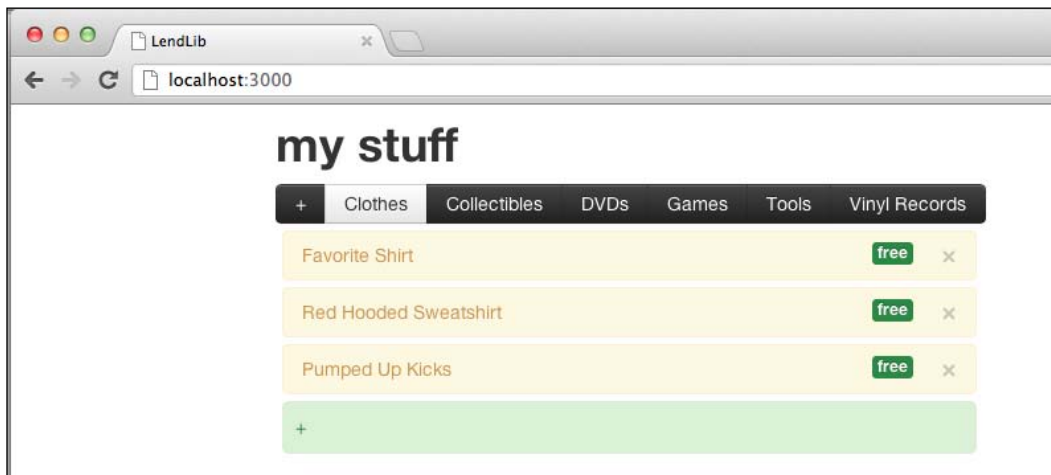
Notice that nothing has changed in our client yet (your items still aren't visible). That's because we need to create the subscribe function.

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

```
Meteor.subscribe("Categories");

Meteor.autosubscribe(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')`. That is the variable that gets passed into the publish function. In other words, the value inside of the Session variable `current_list` will be used as `category_id` in the `find()` function's selector.

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, and the `selectCategory()` function inside `LendLib.js` handles the event, and changes our Session variable.

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

That `Session.set()` triggers a publish event. We wrapped the `Meteor.subscribe()` function for the "listdetails" channel inside of a `Meteor.autosubscribe()` function. We did this because the `Session.set()` event will trigger `Meteor.autosubscribe()`, and we have a `Meteor.subscribe()` function in there, specifically for the "listdetails" channel.

In other words:

1. `Session.set()` triggers an event.
2. `Meteor.subscribe()` listens to that event because it uses the Session variable.
3. Meteor resets the subscription listener on the "listdetails" channel (because it's wrapped inside `Meteor.autosubscribe()`).
4. Meteor sees that new subscription listener and fires an initial event.
5. `Meteor.subscribe()` function picks up that event, passes in the `category_id` variable, and the UI refreshes because of the model change.

Checking your streamlined data

The display is now no different than when we started this chapter. But 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 the object, and you'll see that there are no items listed.

```
> lists.findOne({Category:"DVDs"})
< ▼ Object
  Category: "DVDs"
  _id: "51e0c3ce-d166-44fd-805b-683c24a25585"
  ▶ __proto__: Object
```

The reason there are no items is because our Session variable `current_list` is set to `Clothes`, 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 the object, and you'll see your three items in an array.

```
> lists.findOne({Category:"Clothes"})
< ▼ Object
  Category: "Clothes"
  _id: "6b51e14f-2b95-4569-936e-38fbf860488f"
  ▼ items: Array[3]
    ▶ 0: Object
    ▶ 1: Object
    ▶ 2: Object
    length: 3
    ▶ __proto__: Array[0]
  ▶ __proto__: Object
```

Click around, 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, and therefore more secure and discreet. This probably won't be a problem for your own personal Lending Library application, but as we expand this out in the next chapter, so that multiple people can use it, streamlined and discreet data will really improve performance.

Summary

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

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

6

Application and Folder Structure

To allow you to jump right in, Meteor creates a default set of libraries, 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 changes 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:

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

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, 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.

That 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 sort of a 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 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 accessible to both the client and server, the client may try to implement that code, which can cause all kinds of problems, or will at the very least make the code available to the client, which could quickly become a security issue. There simply 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. Create a new file in `~/Documents/Meteor/` called `LendLibClient.js`. Open `LendLib.js` and cut the entire client code block from it indicated by the following highlighted code:

```
var lists = new Meteor.Collection("lists");

if (Meteor.isClient) {
  ...
}

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



You should have cut about 186 lines of code. Make sure you get the closing `}` bracket!

Now paste the code you just cut into `LendLibClient.js`, and save the changes to both files. You'll notice that this made no visual changes to your running application. That's because Meteor is processing both files, and the `if` 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 you remove the last line, containing the closing bracket `()` for the `if` condition. Save `LendLibClient.js` and then go 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: 1
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 folder structure.


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

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

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

Because we moved `LendLibClient.js` to the `client` folder, the `if` condition is no longer needed. Because of 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 will want to refresh your browser pointing to `http://localhost:3000`.
This is because you crashed the application. Repent of your evil ways, and refresh your page.

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

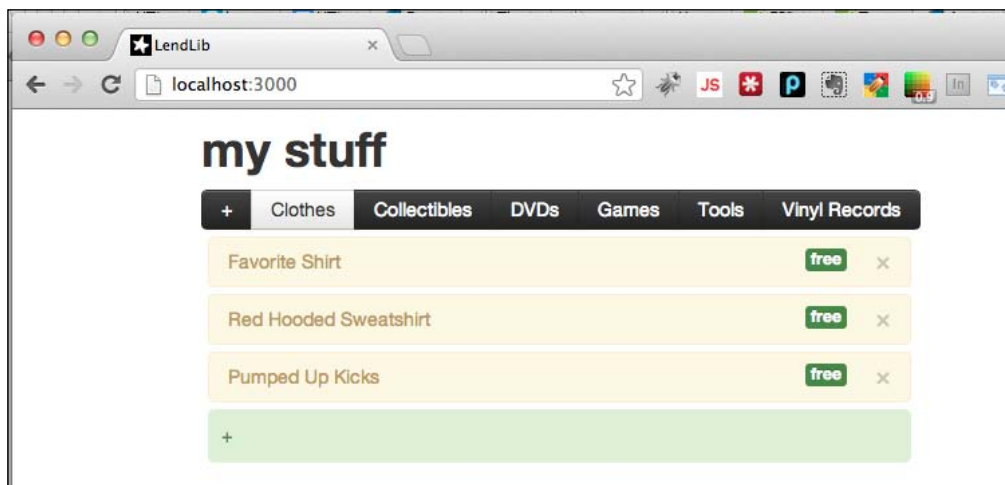
Create a new file named `LendLibServer.js` in the `~/Documents/Meteor/LendLib` folder. Cut the `if (Meteor.is.server) { ... }` block from `LendLib.js`, paste it into `LendLibServer.js`, and save both files.

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

As with the move of the client code, nothing adverse will happen at this point, because we still have the `if` condition. Let's remove that, and let the app crashing continue!

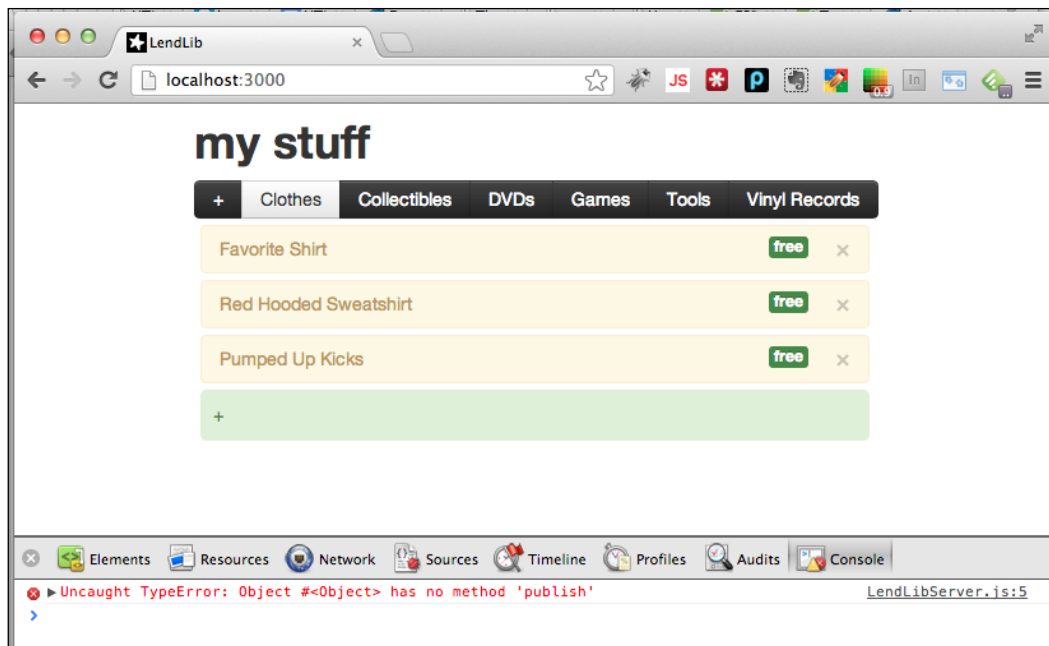
In `LendLibServer.js`, remove the first line, containing `if (Meteor.isServer) {` and remove the last line, containing the closing bracket `}`.

Save your changes, and let's see the carnage!



Huh. No crashes. The app still works fine. What a let down...

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 twofold:

- It's the client side (browser) that threw the error. That won't affect the server application.
- The only code in `LendLibServer.js` is the server code. If that code breaks on the client, 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.

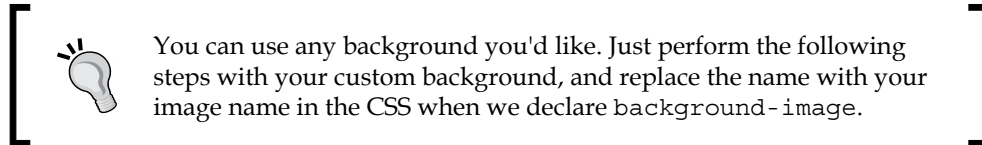
Public folder

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. But there's one more consideration we need to make, and that's for **assets** (images, text/content files, and so on).

The assets are only needed in runtime. We don't depend on them for any logic or processing, and so if we can get them out of the way, the Meteor compiler can ignore them, which speeds 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 of `public`. Then, when all the compiling is done, it will use the `public` folder to access anything it may need to deliver.

Let's add a background image to our application. The handsome and generous fellas over at subtlepatterns.com have quite a few to choose from, and they're all free, so we'll pick one from there. We'll use Texturetastic Gray, because it seems to fit our theme. Navigate to <http://subtlepatterns.com/texturetastic-gray/> and download the image.

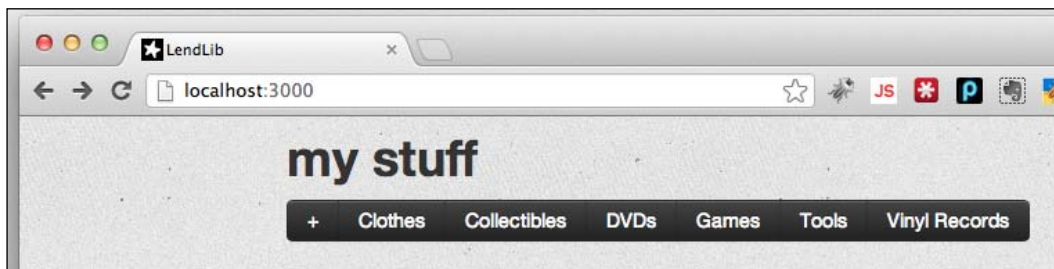


Before we can use our downloaded background, we need to make a quick change to `LendLib.css`, and create a `public` folder.

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

```
body {  
  background-image: url(/texturetastic_gray.png);  
}
```

Save this change. Nothing will happen (yet) but we'll take care of that right now. Create the folder `~/Documents/Meteor/LendLib/public`. Now, open the downloaded zip folder `texturetastic_gray.zip` and copy `texturetastic_gray.png` from the zipped folder to our newly created `public` folder:



The background has changed to your background, and we now have a snazzier interface!

This file is safely tucked away in the `public` folder, so the Meteor compiler doesn't have to deal with it, but 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/#structuringyourapp>.

Security and accounts

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

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 he still had it or not.

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 most of our code written.

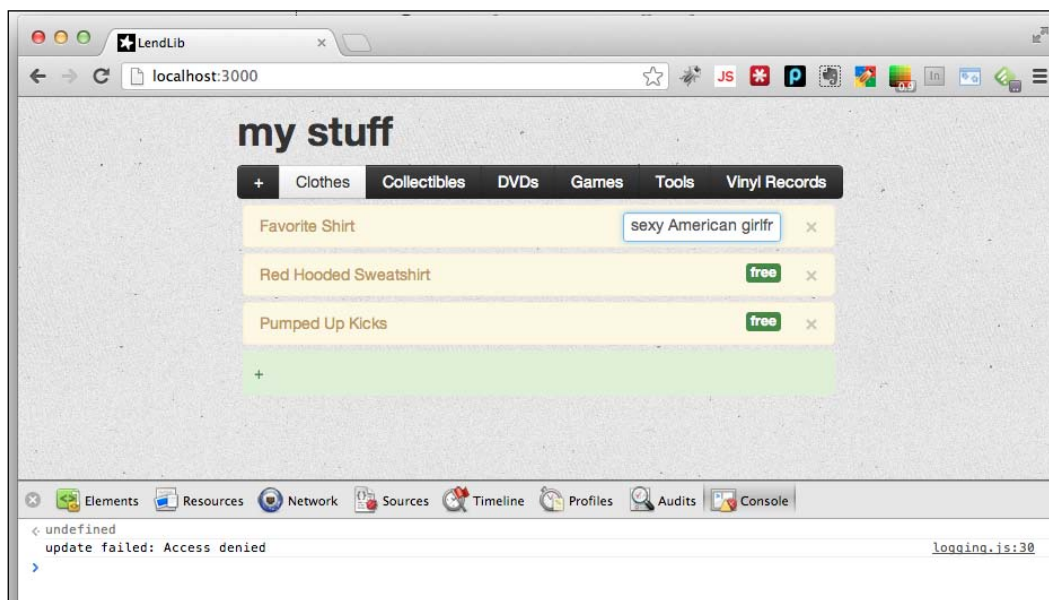
The time has come, we know what we want security-wise, so let's go ahead and get rid of that library. Stop the Meteor application (press `Ctrl + C` in the terminal window) and enter the following command (you need to be in the `LendLib` directory):

```
>meteor remove insecure
```

This will generate the following message:

```
insecure: removed
```

Our application is now secure. 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 you're 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 sexy American girlfriend, but nothing will happen; no deletions, no additions, no changes. Nothing is working now! If you open the browser console, you'll see that every attempt to update the database is being met with the message **update failed: Access denied**:



This message is occurring because we disabled the insecure package. Put another way, no anonymous changes are allowed anymore. Because we don't yet have a login account, all of our requests are anonymous, and will therefore fail.

Adding an admin account

To re-enable update functionality, we need to be able to create an admin account, give the admin account permissions to make changes, and provide the user 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 three commands:

```
$ meteor add accounts-base
$ meteor add accounts-password
$ meteor add email
```

These commands will add the necessary packages to our Meteor application for 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 that 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. 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 (we need it to recover a lost password).

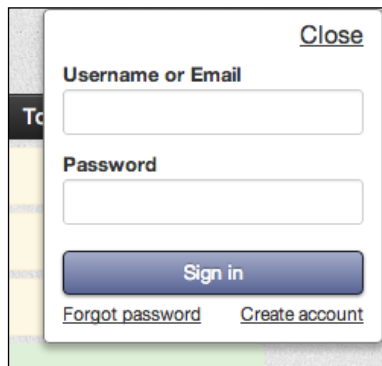
Now open `LendLib.html` and enter the following code directly below the `<body>` tag:

```
<body>
  <div style="float: right; margin-right:20px;">
    {{loginButtons align="right"}}
  </div>
  <div id="lendlib">
```


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

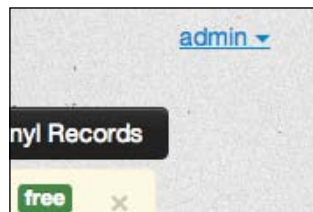


Click on **Sign in** and then click on **Create account** in the bottom right of the popup window:



Fill in the create account form, making sure to enter a username for admin, and a valid e-mail address, so that you can recover your password if needed. Enter and confirm your new password, and click on **Create account**:

You will now be logged in as admin, 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 needed in the UI, while at the same time removing the ability to make changes in the browser console, if the admin account is not logged in.

Our original `LendLib.js` file currently has only one line of code 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 `LendLib.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);
    }
  });
```

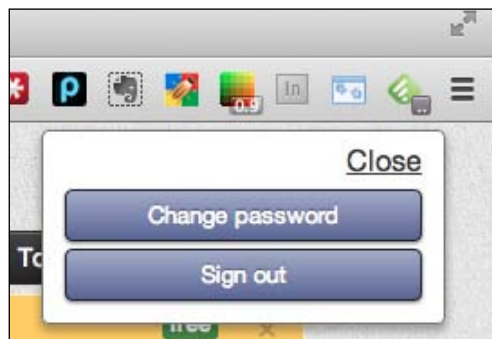
The `adminUser` function is used in multiple places, so it makes sense to create a common function, which simply checks to see if the `userId` making the request is the same as the `_id` of the admin account.

`lists.allow` sets up the conditions upon which operations are allowed, with each operation having a function that returns `true` to allow and `false` to deny. 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 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.

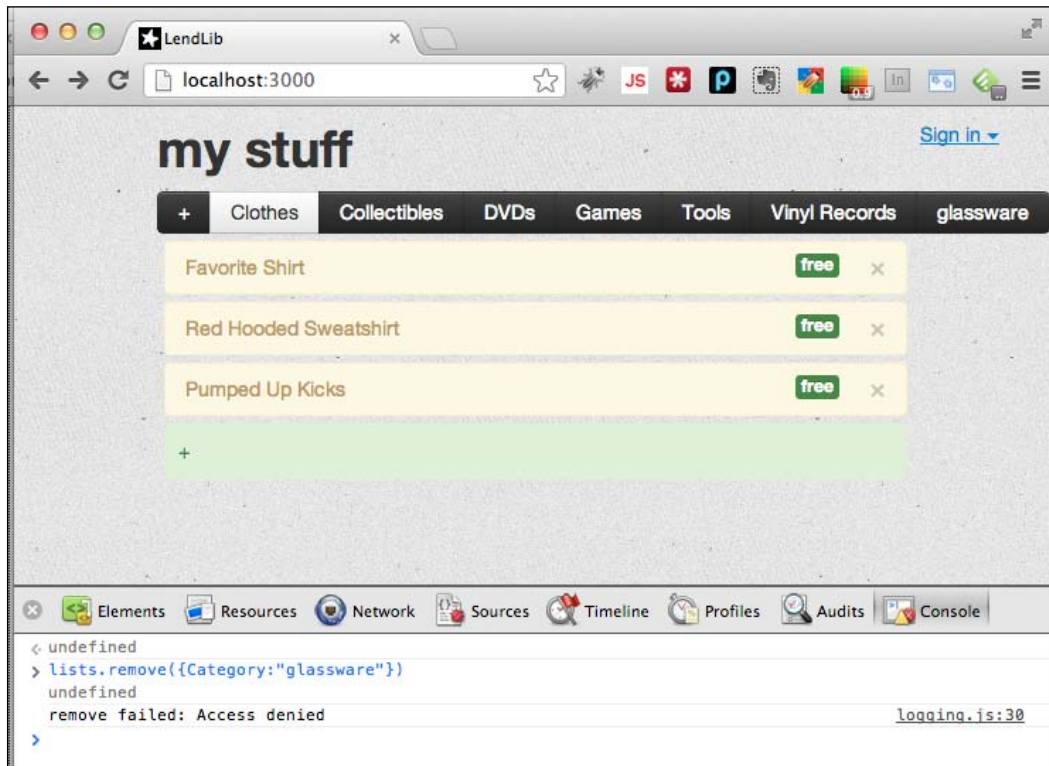
Let's make sure that the access is indeed linked to our admin account. Log out of the app by clicking on **admin** in the top right corner, and clicking on the **Sign out** button:



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

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

You will get 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 allowed actions on the lists level, using `lists.allow()`, we've made it impossible for someone to make changes without being logged in as admin. Both the UI and the browser console are now secure from the evil machinations of STEVE, the wrench thief!

Customizing results

There is one more consideration we should make 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!

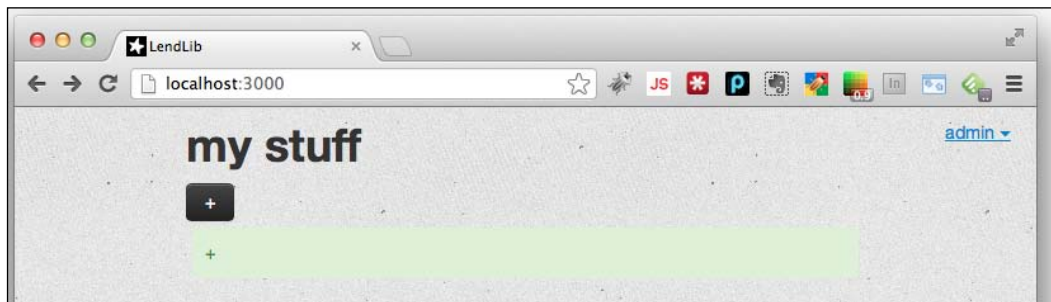
Modifying Meteor.publish()

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 the categories that can be seen by a user, that limitation will cascade to the visible items, because items are found inside categories.

Open `LendLibServer.js`, and modify the `find({})` block, found approximately around line 6:

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

Adding the selector `owner:this.userId` will check each list in our `lists` repository, and return the 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 disappeared!



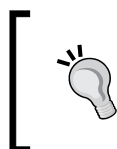
That's because the lists we already created don't have any owner, and we're logged in as admin. We're going to experience similar problems when we try to modify existing items, because no lists have any owner.

We have several options of how to fix this, including manually adding the admin account as the owner, letting the admin account see all unclaimed lists, or just starting with a clean slate. Since we only have one item lent out (dangit, 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!).

In your browser console, while logged in as admin, enter the following command:

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

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



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

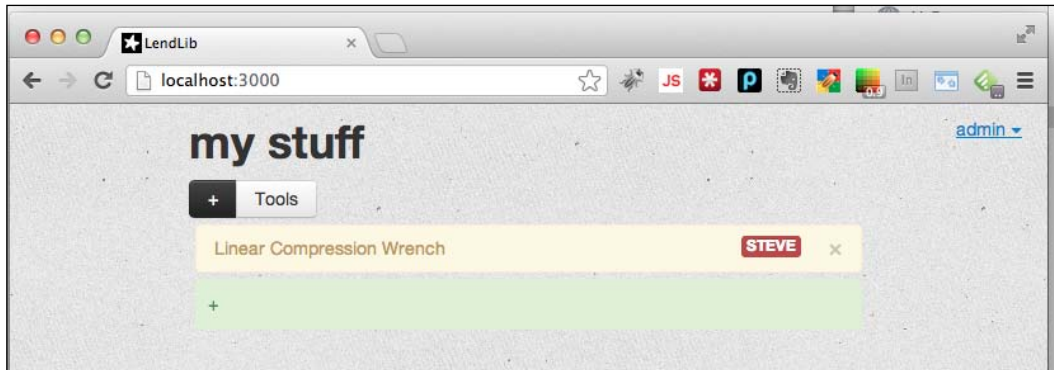
Adding owner privileges

Adding an owner to any new category is pretty simple. We just need to update our `lists.insert()` function, and add the owner field. 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 that call as follows:

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

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

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



There, we're back up and running, and hidden in each list, we now have an owner property. This becomes important when we enable others to create and maintain their own lists.

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 that by going back to `LendLib.js`, and adding some logic to check if either 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:

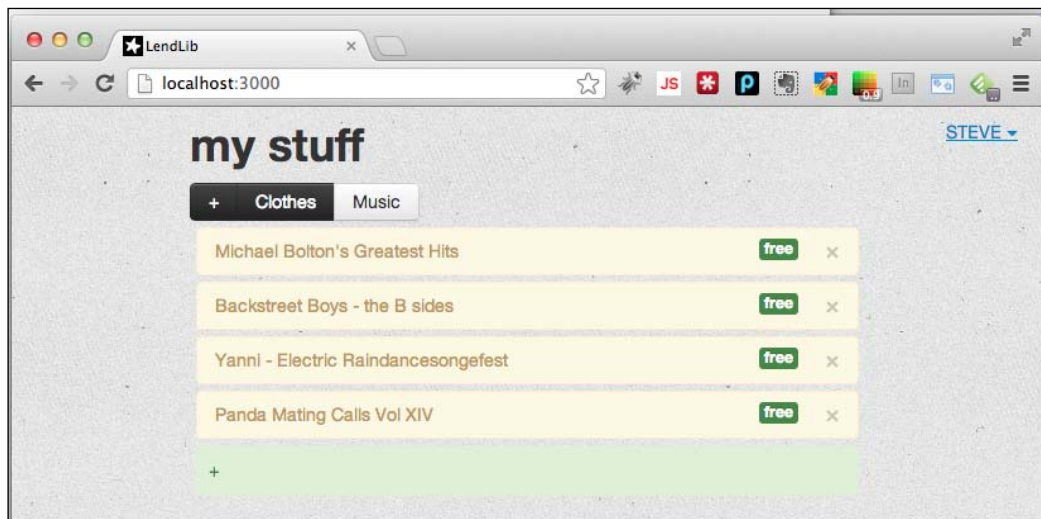
```
lists.allow({
  insert: function(userId, doc) {
    return adminUser(userId) || (userId && doc.owner === userId);
  },
  update: function(userId, docs, fields, modifier) {
    return adminUser(userId) ||
      _.all(docs, function(doc) {
        return doc.owner === userId;
      });
  },
});
```

```
remove: function (userId, docs){  
  return adminUser(userId) ||  
    _.all(docs, function(doc) {  
      return doc.owner === userId;  
    });  
}  
});
```

Inside `insert`, we check to see if the current `doc.owner` is the logged in user. In `update` and `remove`, we iterate through all the records to be updated (using `_.all()`) and check if the `doc.owner` is the logged in user.

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

You'll notice that there's no visibility from one person's lists to another, and consequently 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've learned how Meteor compiles and searches for JavaScript and CSS code, and how to optimize that search. You've learned how to protect your server code, and keep things running smoothly and efficiently. You've learned how to secure your database, through the use of Meteor's built-in Accounts packages, and you've closed all the major security loopholes in your application. Finally, you've enabled multiple accounts, so anybody can use your Lending Library to keep track of their items, and you've done so without compromising on privacy for the end user.

In the next chapter you will learn how to deploy a meteor application to a production environment, and learn techniques to start coding fast, robust, and 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, now anybody can 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 options for deploying your applications. You will learn how to:

- Add and configure third-party packages, such as jQuery, 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

Meteor is rapidly adding packages for the major JavaScript and preprocessing libraries. These packages are intelligent, in that they not only 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.

Listing available packages

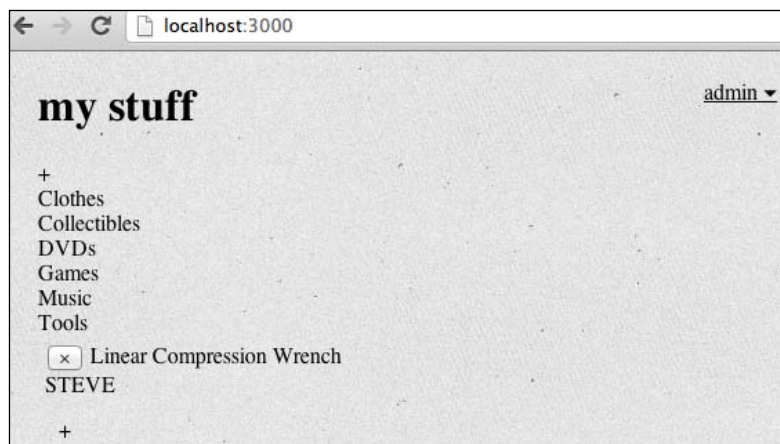
To see a list of the available packages, with a brief description, simply enter the following command into the terminal, and press *Enter*:

```
$ meteor list
```

This will give you a listing of all the packages available with the version of Meteor that you have installed.

As you can see, there are quite a few of the most popular frameworks available, including jQuery, Backbone, 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. Navigate to `~/Documents/Meteor/LendLib/client/` and delete the `bootstrap` directory. It doesn't matter if your Meteor app is running or not (remember, Meteor updates dynamically!). Either start it up and then navigate to `http://localhost:3000` or just navigate there if it 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, 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 you are in the `~/Documents/Meteor/LendLib` folder. Once there, enter the following command:

```
$ meteor add bootstrap
```

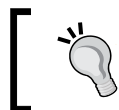
You will receive a very short message:

```
bootstrap: UX/UI framework from Twitter
```

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:



That's literally all there is to it. Using a single command in your terminal, you can add a library or framework to your project, without having to worry about linking, downloading, and making sure that the files are in the right location. 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 --using`.

Because Meteor 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.

Bundling your application

In usual Meteor fashion, bundling your application so 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 it is `~/Documents/Meteor/LendLib`) and enter the following command in the terminal:

```
$ meteor bundle lendlib.tgz
```

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

Deploying to Meteor's servers

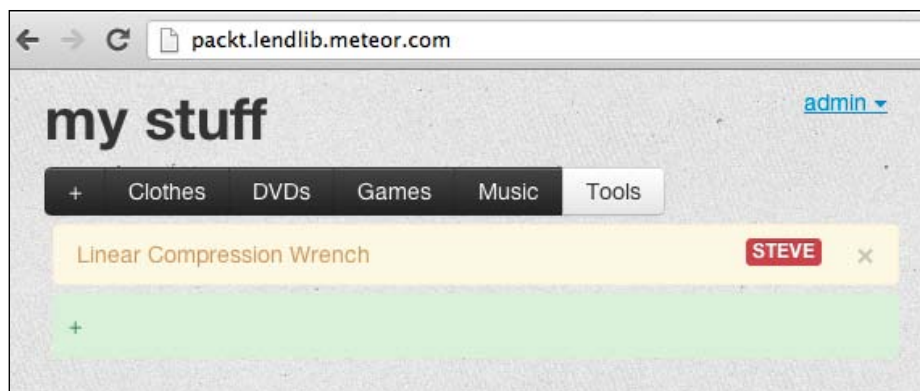
The folks at Meteor have taken deployment one step further, above and beyond what you'd expect from even a paid product, much less a free one. Meteor allows you to deploy your application directly on their deployment servers. Pick a name for your app (we'll use `packt.lendlib`, but you'll need to come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

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 your application deployed and running!



 You will probably want to create the `admin` login before you start using the application, or telling 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 the Meteor 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 also preserves your data, so 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 the Meteor servers. Set up a CNAME pointing to `origin.meteor.com` using your host provider, and you can then deploy to that hostname. For example, if we had the subdomain `meteor.tool1.com` pointing as a CNAME to `origin.meteor.com`, we would run the following command in the terminal:

```
$ meteor deploy meteor.tool1.com
```

If your CNAME is set up properly, this 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 on setting up a CNAME route. It varies from provider to provider, but it's pretty easy to do.]

Deploying to a custom server

At the time of this 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.

With that said, we'll walk through deploying a Meteor application to a custom server, and leave exploring hosting services (such as Heroku or AppFog) up to you.

Server setup

The server you'll be hosting your application from needs two things:

- Node.js, version 0.8 or later
- MongoDB (latest version)

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

To install MongoDB, visit <http://docs.mongodb.org/manual/installation/> and follow the instructions for your corresponding OS. Once it is installed, make sure to set up a database and name it `lendlib`.

Once these two products are installed and configured, you will have a default location for your MongoDB. This will be something like `mongodb://localhost:27017/lendlib`. You'll need that URI in a future step, so make sure to write it down or have it available for reference.

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



Deploying your bundle

If you recall, we created a tarball earlier in this chapter. We now need to extract that tarball, and make a couple of modifications, then we're ready to turn on our app. Copy `lendlib.tgz` to your server, in the directory you'll use for deployment (for example `~/Sites/LendLib`). Once `lendlib.tgz` is in the right place, extract the tarball with the following command:

```
$ tar -zxvf lendlib.tgz && rm lendlib.tgz
```

This will extract the tarball, and you'll have a new folder named `bundle`.

Optional – different platform

If the machine you developed the application on is different than the machine you are deploying to, you will need to rebuild the native packages. To do this, enter the `node_modules` directory:

```
$ cd bundle/server/node_modules
```

Once there, remove the `fibers` directory:

```
$ rm -r fibers
```

Now rebuild `fibers`, using `npm`:

```
$ npm install fibers
```

This will install the latest `fibers` version, specific for the platform you're deploying to. You do not need to do this if the `dev` machine and `deploy` machine are running the same platform.

Running your application

Now that your bundle is properly extracted, you're ready to turn on your application. You will need the following information to start up your app:

- The root URL (for example, `http://lendlib.mydomain.com` or `http://localhost`)
- The port you want to run the app from (for example, `80`)
- Your MongoDB URI (for example, `mongodb://<user>:<password>@alex.mongohq.com:10022/lendlib`)

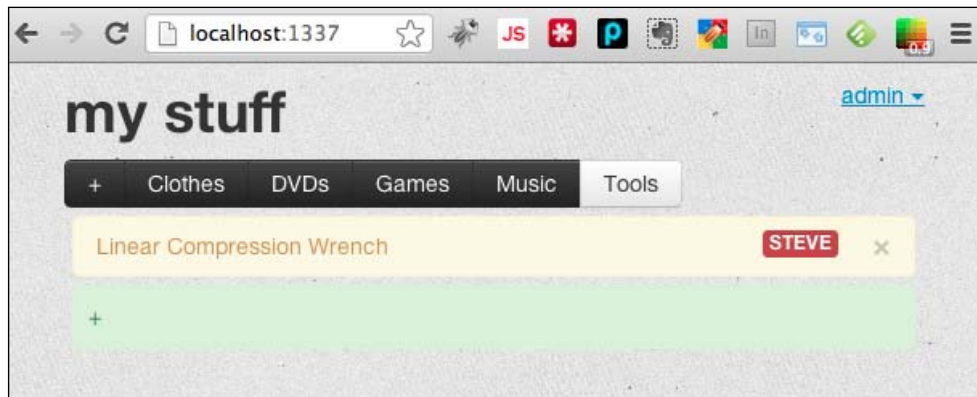
Once you've made your decisions and gathered this information, start Node.js for your application. Navigate to your `root` folder (ours is `~/Sites/LendLib`) and enter the following:

```
$ PORT=80 ROOT_URL=http://localhost MONGO_
URL=mongodb://<user>:<password>@alex.mongohq.com:10022/lendlib node
bundle/main.js
```


Let's break this down:

- `PORT` sets the port variable, so that NodeJS knows what port you'd like to serve the application form
- `ROOT_URL` sets the `rootUrl` variable, so NodeJS knows what hostname requests are meant for your application
- `MONGO_URL` tells NodeJS where it can find our MongoDB
- `node` is the invoking command
- `bundle/main.js` is the starting JavaScript file invoked by NodeJS

If all your information is correct, the app will run, and you'll be able to test it using a browser:



You can go even further into deployment, such as setting up environment variables, running your app as a daemon/service, and even using remote servers to publicly host your application. What we've done this far should be sufficient to get you started, and well on your way to using Meteor in a production environment.

Summary

You are now a Meteor expert! Seriously. You know how to build an application in Meteor from the ground up. You understand the design patterns and database 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.

Because Meteor is so new, there are very few people that possess as much working knowledge about Meteor as you now possess. That, by definition, makes you an expert. Now it is up to you to apply that expertise. Suggest using Meteor in your upcoming development projects, contribute to the Meteor community through code contribution, answering questions on the forums, and making tutorials of your own.

Meteor is a breakthrough technology, gaining more and more momentum, and you now have the knowledge and experience 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

Symbols

`$addToSet` 74
`$pull` 75
`$set` 75

A

`accounts-ui` package 93
`addItem()` function 59, 62
admin account
 adding 92, 94
admin permissions
 granting 95, 96
`adminUser` function 96
`AppFog` 107
application
 background image, adding 90
 bundling 105, 106
 deploying, to custom server 107
 deploying, to Meteor servers 106
 running 109
autopublish
 turning off 77

B

beefy desktop application (fat app) 32
bootstrap directory 104
browser console 19
bundle
 deploying 108
`bundle/main.js` 110

C

callback functions
 using 75
categories
 about 38
 listing 78-80
Categories event channel 79
categories template 47, 48
changes
 broadcasting 75
Chrome
 console, using 19
 web page 29
client folder 85-89
client/server design pattern 32
code changes
 creating 12
connected client/server application
 (thin app) 32
Console icon 20
context 55
controller 33
controller (MVC) 51
create command 10
curl
 installing, with Meteor 7-9
cURL. *See* curl
`current_list` session parameter 53
custom server
 deploying to 107, 108

D

deploy command 107
doc.owner 101
document-oriented storage
 and MongoDB 69-72
 relational database, need for 70, 71
dumb terminal 31

E

each statement 49, 50
events
 adding 57-61
 published 75, 76
example application
 file location, selecting 9
 loading 9, 10
 previewing 10, 11
 starting 10

F

fibers directory
 removing 109
find({}) block 98
find() command 24
find() function 78, 83
findOne() function 63
focusText() function 64
for each loop 39
for each statement 38
Fraggles command 28

H

Heroku 107
HTML
 collections, displaying 21
 template 47

I

if condition 86
if (Meteor.isClient) function 43
if (Meteor.isServer) {...} block 85
if statement 85
insecure library

 removing, from Meteor 91, 92
item.LendClass property 50
item.Lendee property 50
items
 listing 81, 82

K

keyup event 44

L

LendClass properties 54
Lendee properties 49, 54
lending_item class name 49
Lending Library
 base application, creating 16, 17
 browser console 19, 20
 building 15
 code, cleaning up 25-27
 collection, creating 18
 collections, displaying in HTML 21-25
 creating 15, 16
 data, adding 20, 21
 MongoDB 18
 multiple clients 29, 30
 page, URL 28
lendlib 48
LendLibClient.js 86
LendLib folder 106
lendlib.tgz 108
LentTo property 63
lists.allow() code block 97, 100
lists collection 21, 24, 28, 39
lists data model 36
lists.find() 78
lists.insert() function 99
list_status 55
lists.update() function 74
Live HTML 21

M

Massively Multiplayer Online games
 (MMOs) 32
Meteor
 about 7, 31
 cached and synchronized data (the model)

- 35, 36
- client code 39
- client code (the View-Model) 39, 40
- create command 10
- example application 9
- installing, with curl 7-9
- Templated HTML (the view) 37, 39
- third-party packages 103
- todos example application 10
- URL, for installing 8
- Meteor.Collection query** 24
- Meteor data synchronization**
 - URL 37
- Meteor.flush()** 59
- Meteor.publish()** code 99
- Meteor.publish()** function
 - about 81, 98
 - modifying 98, 99
- Meteor's servers**
 - deploying to 106
 - hostname, using 107
 - updating to 107
- Meteor.subscribe()** function 81
- Minimongo.** *See* MongoDB
- model** 33, 35, 36
- model updates** 61, 63
- Model View Controller (MVC) pattern** 33
- Model View View-Model.** *See* MVVM
- modern web applications**
 - about 31
 - browser grows up (MVVM) 33, 34
 - machines (MVC), rise 32
 - origin 31, 32
- MongoDB**
 - about 18, 69
 - and document-oriented storage 69-72
 - direct commands, using 72-74
 - URL 71
 - URL, for installing 108
 - working 72-74
- MongoDB documentation**
 - URL 78
- MongoDB URI** 109
- MongoHQ**
 - URL, for installing 108

- MONGO_URL** 110
- multiple users**
 - enabling 100
- MVC** 32
- MVVM** 31, 33, 76
- N**
- node** 110
- Node.js**
 - URL, for installing 108
- node_modules** directory 109

- O**
- owner privileges**
 - adding 99, 100

- P**
- PORT** 110
- presentation model** 34
- presenter (MVP)** 51
- public folder** 89-91
- publishers**
 - configuring 76
- publish/subscribe model**
 - using 75

- R**
- reactive contexts** 15
- relational database**
 - need for 70
- removeItem()** function 60, 61, 75
- ROOT_URL** 110

- S**
- security**
 - implementing 91
- selectCategory()** function 82
- server folder** 85-89
- Session.set()** triggers 59, 82
- Session variable** 41, 55, 60, 75
- Sign out button** 96
- style updates** 64-67
- subscribe function** 81

T

template

- creating 40-45
- HTML template 47

Template functions 24

Template.list.items function 52

third-party packages

- listing 103, 104

this.Category 38

this submit button 34

todos list

- changing, to items list 12-14

U

updateLendee() function 61, 75

update() function 62

V

view 34, 37, 38

view data bindings 37

View-Model 39, 40, 51-53

View-Model (MVVM) 51

view states 54

W

web page, chrome 29



Thank you for buying Getting Started with Meteor.js JavaScript Framework

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: 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 licences, 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, 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.



Twitter Bootstrap Web Development How-To

ISBN: 978-1-849518-82-6

Paperback: 68 pages

A hands-on introduction to building websites with Twitter Bootstrap's powerful front-end development framework

1. Conquer responsive website layout with Bootstrap's flexible grid system
2. Leverage carefully-built CSS styles for typography, buttons, tables, forms, and more
3. Deploy Bootstrap's jQuery plugins to create drop-downs, switchable tabs, and an image carousel



Ext JS 4 Web Application Development Cookbook

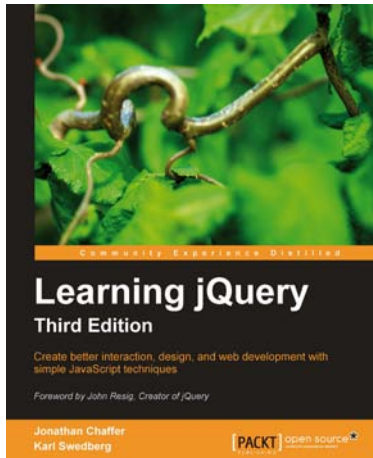
ISBN: 978-1-849516-86-0

Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

Please check www.PacktPub.com for information on our titles



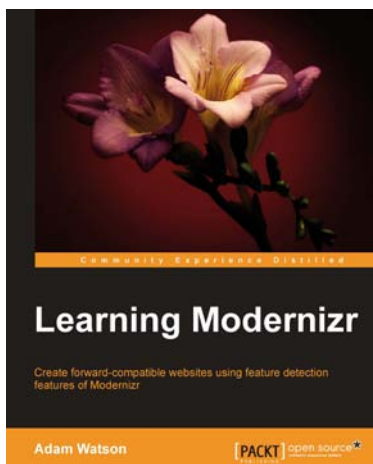
Learning jQuery

ISBN: 978-1-849516-54-9

Paperback: 428 pages

Create better interaction, design, and web development with simple JavaScript techniques

1. Create interactive elements for your web designs
2. Learn how to create the best user interface for your web applications
3. Use selectors in a variety of ways to get anything you want from a page



Learning Modernizr

ISBN: 978-1-782160-22-9

Paperback: 130 pages

Create forward-compatible websites using feature detection features of Modernizr

1. Build a progressive experience using a vast array of detected CSS3 features
2. Replace images with CSS based counterparts
3. Learn the benefits of detecting features instead of checking the name and version of the browser and serving accordingly

Please check www.PacktPub.com for information on our titles

