# Mastering MeteorJS Application Development

MeteorJS makes full-stack JavaScript application development simple – Learn how to build better modern web apps with MeteorJS, and become an expert in the innovative JavaScript framework

Jebin B V

[PACKT] open source*
PUBLISHING     community experience distilled

# Mastering MeteorJS Application Development

MeteorJS makes full-stack JavaScript application development simple – Learn how to build better modern web apps with MeteorJS, and become an expert in the innovative JavaScript framework

**Jebin B V**

# Mastering MeteorJS Application Development

# Credits

**Author**
  Jebin B V

**Project Coordinator**
  Shweta H Birwatkar

**Reviewer**
  Ethan Escareño Rosano

**Proofreader**
  Safis Editing

**Commissioning Editor**
  Veena Pagare

**Indexer**
  Hemangini Bari

**Acquisition Editor**
  Indrajit Das

**Production Coordinator**
  Shantanu N. Zagade

**Content Development Editor**
  Preeti Singh

**Cover Work**
  Shantanu N. Zagade

**Technical Editor**
  Saurabh Malhotra

**Copy Editor**
  Trishya Hajare

# About the Author

**Jebin B V** is fond of JavaScript and anything related to JavaScript excites him. He is a front-end developer who has experience of full-stack development and also with stacks such as LAMP. Right from the beginning of his career, he has worked as a founding engineer for various startups.

Initially, he started his career as a PHP developer where he developed Web applications using PHP and its frameworks such as YII, Zend, Symphony, and WordPress. Later, he saw the growing potential of JavaScript and decided to be a JavaScript developer. He self-taught JavaScript and its concepts, with which he moved to work as a full-time JavaScript developer for a revolutionary big data product called DataRPM. When he was a part of DataRPM, he developed a significant part of the product that helped the product to grow rapidly.

At DataRPM, he nourished himself to be a challengeable JavaScript developer who can build a whole product all alone. In a short period, he learned and mastered JavaScript's quirks and solved many problems to scale the product. With JavaScript, he also learned how to design modules using proper design patterns, structuring the codebase, and maintaining the code discipline.

Along with development, he loves to teach. He always strives to share knowledge. Whenever he finds a solution to a peculiar problem, he calls up the team to let them know how he solved it. Not a single day of his life goes by without reading, and the major part of his reading is about JavaScript and its ecosystem. The routine of his professional life is reading about code, writing code, and teaching to code better.

Carrying all these experiences, he moved to another startup where he built, all alone, the web version of the instant messaging application, Avaamo. The web version was developed and launched in less than three months, which is now consumed by all premium users, and there are also quite a few firms who run their businesses in Avaamo Web.

Other than JavaScript, the only other thing he is very passionate about is bodybuilding. He does weight training and calisthenics on his preferable schedules. He is very concerned about pollution and thus he commutes by bicycle to work every day. He has a very good sense of innovation and user experience, which is driving him to build products that can solve day-to-day problems, again using JavaScript and open source tools and frameworks such as MeteorJS.

# About the Reviewer

**Ethan Escareño Rosano**'s very first step in the programming world was to try IOS developing. He then tried JAVA and finally fell in love with JavaScript. As many developers, he too had to learn everything on his own. At some point, he quit his last job and decided, along with his friend, Javier, to start coding his first app and his first start-up, which was "Pa'donde", a web app that promised to empower the Mexican commerce of restaurants. "Pa'donde" won a contest called "Possible", but due to some disorganization, they couldn't present the project. After this, he worked for some start-ups, where he met George. Currently, he's still working with his friend, Javier, and his girlfriend, Katya. He is also working as a CTO for a start-up called Dobox.

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

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

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

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



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

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

## Why subscribe?

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

## Free access for Packt account holders

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

# Table of Contents

# Preface

Web is inevitably one of the core reasons for the advancements that we experience today almost everywhere. Though the development of Web and its content has been happening for quite a long period of time, the current decade is very significant, especially for JavaScript. When people started writing JavaScript in servers, the language became truly universal. Apart from Web, JavaScript has found its way into IoT devices too, which is considered to be the most opportune.

The potential and traction of JavaScript has brought countless developers into developing JavaScript-based applications, frameworks, and utilities. Even after evolving so much, JavaScript application development is deficient in certain areas. Developers are spending time on doing repetitive tasks, such as data fetching, wiring them to views, posting data back to servers to persist, and so on. Moreover, it is required to speed up the data transfer that is slow in the case of HTTP and HTTPS. Keeping all these traditional problems in mind, a bunch of developers developed a solution called MeteorJS.

MeteorJS provides most of the things that a developer would have to do repetitively, out of the box. The developers need to concentrate mostly on business logic rather than spending time on the basic data fetch and transfers, optimizations for network latency, syncing of data across devices, and reactivity.

There are already plenty of developers and organizations using MeteorJS in production. Many are experimenting with MeteorJS to make it the de facto framework for their future work. This book is written with the intention to guide those who are experimenting with MeteorJS to develop their future applications.

The best part of the book is that it doesn't just cover Web application development. It helps to write maintainable MeteorJS applications and deploy them to production. In short, the book aims at guiding the developers to develop production-ready, mobile-compatible, and horizontally scalable MeteorJS applications.

# What this book covers

*Chapter 1*, *Building a MeteorJS Web Application*, provides an introduction to developing a Web application using MeteorJS. Readers will develop a multipage, multilayout application in this chapter, which gives enough insight about MeteorJS components and routes.

*Chapter 2*, *Developing and Testing an Advanced Application*, helps you rebuild the same application as in the previous chapter, but using a generator and other advanced packages to ensure the app is of good quality. Every possible way of debugging the entire application and testing the code is discussed in this chapter.

*Chapter 3*, *Developing Reusable Packages*, shows that packages are very important blocks for any MeteorJS app. This chapter shows the reader, with a typical example, how to develop and test custom packages and also provides the steps to distribute them for community use.

*Chapter 4*, *Integrating Your Favorite Frameworks*, guides the readers to use Angular.js and React.js with MeteorJS. MeteorJS has its own view layer managed by Blaze. However, many developers want to use their favorite frontend framework instead of Blaze. How powerfully d3.js can be used with MeteorJS is demonstrated with examples in this chapter.

*Chapter 5*, *Captivating Your Users with Animation*, shows how animations improve the user experience to a great extent. With all the in-built reactivity of MeteorJS views, many developers struggle to find ways to incorporate animations. This chapter walks you through creating soothing animations with a lot of examples.

*Chapter 6*, *Reactive Systems and REST-Based Systems,* helps us understand the reactivity of MeteorJS to its depths and the precautions needed to handle reactivity. Also, this chapter discusses how to use MeteorJS as a REST-based system for consuming API.

*Chapter 7*, *Deploying and Scaling MeteorJS Applications*, teaches you to deploy, monitor, and scale MeteorJS applications, as MeteorJS is not so familiar in terms of deployment.

*Chapter 8*, *Mobile Application Development*, helps you understand that one of the most important features of MeteorJS is to write once and build for multiple platforms. Developers can write code that can be ported as a mobile application in MeteorJS. This chapter will guide you to develop an app for a mobile using MeteorJS.

*Chapter 9*, *Best Practices*, *Patterns, and SEO*, discusses various best practices to design, develop, and maintain MeteorJS applications, and also the best patterns to follow in order to organize the code and structure modules. This chapter also guides you to make the application search engine friendly to improve the sites ranking. With this chapter, readers will get to know where to find anything related to MeteorJS.

# What you need for this book

You will need the following things to understand the content of this book:

- Node.js
- NPM
- MeteorJS
- iron-cli
- MeteorJS hosting platform
- Cordova, iOS, and Android devices

# Who this book is for

This book is for developers who want to develop MeteorJS applications in a mature and maintainable way. The readers are expected to know the basics of MeteorJS such as the core principles, templates, server and client code positioning, and basic directory structuring. A little knowledge about querying MongoDB will help very much to understand data fetching from MongoDB. It is assumed that the reader has developed small example applications with MeteorJS.

# Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create the `bookTravel` directory in the client and add `bookTravel.html`."

A block of code is set as follows:

```
data: function() {
  templateData = {
    _id: this.params._id,
```

```
        bus: BusServices.findOne({_id: this.params._id}),
        reservations: Reservations.find({bus:
        this.params._id}).fetch(),
        blockedSeats: BlockedSeats.find({bus:
        this.params._id}).fetch()
    };
    return templateData;
}
```

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

```
Router.route("/book/:_id", {
  name: "book",
  layoutTemplate: "createTravelLayout",
  template: "bookTravel",
  waitOn: function () {
    Meteor.subscribe("BlockedSeats", this.params._id);
    Meteor.subscribe("Reservations", this.params._id);
  },
```

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

**iron add velocity:html-reporter**

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the **Cart** division in the top-right."

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

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

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

# Customer support

Now that you are the proud owner of a Packt 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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

# 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 could 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/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

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

# Piracy

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

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

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

# Questions

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

# 1
# Building a MeteorJS Web Application

The need for omni-presence has increased dramatically and the Web is the primary means of being really omni-present. This has led to tremendous advancements in the technology that subsequently gave us a window to the ever-growing Web, which is browsers. To develop something interactive in the browser, we end up with the ultimate language, JavaScript. Though it is the most underestimated and misinterpreted language, it has gained enormous value in the last decade. The rise of libraries and frameworks, such as jQuery, YUI, Backbone.js, Angular.js, Ember.js, and so on, have transformed the way applications are developed today. It didn't stop there, and the language found its space in the server as well, with the introduction of Node.js. Now, the language also manages to find a warm welcome in IoT space, with the introduction of Tessel.io, Windows 10, and so on. This is a better time than ever to become a JavaScript developer.

There is a trend where backend, that is, the data access layer, is developed with other languages, while the whole app is rendered and managed using one of the MV* JavaScript frameworks in the browser itself. With Node.js, JavaScript applications started becoming isomorphic. Node.js is very popular because of the default asynchronous behavior. Frameworks such as Express.js helped to create isomorphic applications.

Still something was missing. Even after all these improvements, we developed applications with a lot of redundancies in terms of code. For example, data fetching, data binding, view to model reactivity, and so on, are not really that efficient. So, a group of developers gathered around and found a powerful solution called **MeteorJS**. This book is about mastering your skill to develop applications using MeteorJS.

In this chapter, we will learn the following parts of MeteorJS by developing an application:

- MeteorJS internals and working principles
- How to build a customized login solution
- How to create routes, templates, and layouts
- Handling forms from the template handlers
- Persisting the data to a database
- Data handling between the client and server and reactive variables
- How to use multiple layouts in the application

# An overview of MeteorJS

As I have mentioned earlier, MeteorJS is an open source isomorphic framework that is built using JavaScript that runs on Node.js. The beauty of the framework lies in the core principles of the framework. A truly modern application needs to be highly reactive. To be reactive, the existing stack is not that great. HTTP is slow because of the latency in handshaking on every request. The databases that we use are not reactive. HTML needs to be updated as and when the data changes, which is an overhead for developers. Also, the updated data must be transferred to all the clients without a refresh or manual intervention. MeteorJS provides a one-stop solution for all these problems and needs.

To master something, it is not enough to know how to use it, but also it is absolutely necessary to know the internal working of the thing. In our case, it is really important to know the working principles of MeteorJS to master it.

MeteorJS is built using a bunch of packages that can be used independently in one of your projects if needed. Let's take a deeper look into these packages.

A typical MeteorJS application has three parts: the server, the communication channel, and the client. Once a server is connected to a client, there is a socket introduced between the client and the server. Any data transfer that happens between the server and the client is through this socket.

# Server

The server is where MeteorJS is installed on top of Node.js. MeteorJS, on the server, is connected to MongoDB that is the default database for the framework.

# MongoDB

MongoDB is a NoSQL database. Each record is a document and the set of documents is called a collection that is equivalent to a table in a SQL database. As you may know, MongoDB is an in-memory JSON-based database, which means it is extremely fast in favorable conditions. Usually, MongoDB can have operation logs, which is called **oplog**. The oplog has the database operations happening with time. This is used in the case of making the replica (slave) of the primary database. The operations that happen in the primary database are copied to the secondary databases asynchronously.

Though MongoDB is not a reactive database, **Livequery**, which is a part of MeteorJS, does some work internally to get the updates of the database periodically. Livequery can connect to the database and set triggers on certain conditions as required. In the case of MongoDB, triggers are not supported. So, the Livequery depends on oplog if enabled, or else it will poll the database at a particular interval. When oplog is enabled, which should be the case for production, MeteorJS observes the oplog and intelligently does the transaction. If oplog is not enabled, meteor polls the database, computes the diff, and then sends the changed data to a client. Livequery can guess when to poll the database as all the write operation to the database go via Livequery.

# Publish/Subscribe

A very important part of MeteorJS is this good old design pattern. By default, the entire Mongo database is published to the client from the server. However, it is not good for production to auto-publish all the data to the client. Instead, the client can subscribe to the required data that is published by the server. The subscriber will automatically be updated whenever there is a change in the subscribed data:

```
/* Publishing from server. */
if (Meteor.isServer) {
  Meteor.publish("tasks", function () { //Registering "tasks"
  publication
    return Tasks.find();
  });
}
/* Subscribing from client */
Meteor.subscribe("tasks");
```

# Communication channel

In the realm of publish and subscribe, there should be a way to transfer the subscribed data. MeteorJS uses a protocol called **Distributed Data Protocol** (**DDP**). To define DDP, it is simply a REST over Web socket. It is a socket implementation that can transfer JSON data to and fro (duplex). MeteorJS uses Socket.io internally to establish a socket connection between the client and the server. However, neither the client nor the server knows to whom they are talking to. All they know is to talk DDP over the socket.

DDP is human-readable and one can literally see what is transferred via DDP using the package Meteor DDP analyzer. Over DDP, there will be either a message transfer or procedure calls. You can use DDP not only with MeteorJS, but also with any other languages or projects that can support socket. It is a common standard protocol that gives a great way to pair with any other DDP-consuming implementation, if required. Sockets reduce latency in a very high rate than HTTP, because of which it is very much suitable for reactive applications.

# Client

Let's say the server is ready with the data. How does the client keep all this data so that it can be reactive? Also, who is doing the magic of refreshing the views when the data changes?

Modern apps try their best to solve two things as intelligently as possible. One is latency compensation and another is reactivity. MeteorJS does that quiet powerfully using the following implementations.

## MiniMongo

Being a developer, if you are implementing a table that can be sorted, filtered, and paginated, what will you do to make it faster? Won't you fetch the data and keep it in browser memory in the form of multi-dimensional array, apply all the operations on the cached array, and update the table as and when required? The same is the case for MeteorJS with little advancement in the cache implementation. Instead of using a plain object or array, MeteorJS creates a cache in the browser called **MiniMongo**, which is again a simplified client memory database. The highlight is that you can query it in the way you query the MongoDB that enables you to use the same query both in the client and the server.

Whenever there is change in MongoDB, the server sends the difference to the client and that data is stored in MiniMongo. At any instance, MeteorJS tries to keep both the MongoDB in sync.

# Tracker

Now, the data is with the client. Let's call this model. In a MV* framework, we have the views bound to models to auto-update the views as the model changes. In Backbone.js, you have to do it explicitly. However, in Angular.js, it is taken care of by the framework itself with the help of $ scope and digest cycles. How does MeteorJS handle data changes? With the help of Trackers. Trackers create observers for everything you need to track. By default, MeteorJS has enabled a tracker on certain data sources, such as database cursors and session variables. You can even have a custom variable to be tracked using the tracker.

# Blaze

Blaze is a templating engine that is reactive because of the tracker. Blaze plays the magical part of reactivity by binding the data to the templates. An important point to note is that Blaze is declarative, which means you just have to tell Blaze what to do when the data changes, and need not say how to do it. With the help of the tracker, Blaze keeps a track of model changes and reacts to the change. The default templates are spacebars. This is a variant of Handlebar's templating engine. You can use Jade as well. Blaze is again intelligent to compute the diff of what needs to be updated. It doesn't update all the template until it is necessary. Blaze handles the user interactions and thereby makes a call to the server, if absolutely needed.

# Additional information

Developers can use MeteorJS ecosystem, which has a lot of packages to use in the application. Iron router, masonry, auto-form, simple schema, and twitter bootstrap are a few important packages for application development. Being a Node.js-based framework, developers can harness the power of Node.js ecosystem as well. You can also use NPM packages in the MeteorJS application.

MeteorJS does hot code deployment, which means without restarting the application, the code is deployed and the client will see the changes without completely refreshing the browser.

> MeteorJS has just reached 1.x. There are many new features and implementations yet to come such as drivers for different databases, support of various front-end frameworks, and so on. However, basically, MeteorJS is designed in a way to accommodate anything just by small integration work. If you really want to see if this is true, check their source in GitHub (`https://github.com/meteor/meteor/tree/devel`).

*It was said, "To know the truth, return to the source".*

If you are interested in learning more about the framework's internals, I would suggest take a look at the source code that will help you learn a lot of new things.

# Developing a bus reservation application

Long story, short—MeteorJS is awesome. Let's take a look at the awesomeness of MeteorJS by developing an application.

By developing this application, you will learn about MeteorJS login, routing, using multiple layouts based on route, form handling, database operations, publishing and subscribing data, custom reactive data sources, and server calls. By the end, you will see the reactivity of the framework in action.

To understand and experience MeteorJS, we are going to build a bus reservation application. Let's define what we are going to develop and then get our hands dirty:

- Develop and enable account creation and login
- Create bus services
- Create a landing page that has the list of buses available
- Develop a search section besides the listing so that the users can reach their appropriate bus for booking
- Create a reservation page where users can block and reserve the seats

To keep the application simple, a lot of details are omitted. You can implement them on your own later.

> This is not the professional way to build MeteorJS. With this application, you will get started and in the upcoming chapters, you will learn how to develop apps like a pro.
>
> Basic prerequisite is that Meteor must be installed. You should know how to create an application and add or remove packages, and also know a little about routes, mongo, and collections.

Let's start from scratch. Create a MeteorJS application using the `create` command (`meteor create BookMyTravel`) and remove all the default `.html`, `.css`, and `.js` files. Create the following directories: `assets`, `client`, `commons`, and `server`. Remove the `insecure` (`meteor remove insecure`) and `autopublish` (`meteor remove autopublish`) packages. Add the twitter bootstrap (`meteor add twbs:bootstrap`) package that will help us with layout and designing. Add the Moment.js (`meteor add momentjs:moment`) package for data manipulation.

As our application is not a single page application, routes are required to navigate between pages. For routing purposes, we'll use the famous `iron-router` package. Add the iron-meteor package to the application by running the `meteor add iron:router` command. Create the `routes.js` file inside the commons directory and add the following code:

```
Router.configure({
  notFoundTemplate: 'notFound',   //template with name notFound
  loadingTemplate: 'loading'      //template with name loading
});
Router.onBeforeAction('loading');  //before every action call show
  loading template
```

Define these two templates in an HTML file of your choice as follows:

```
<template name="notFound">
  <div class="center">You are lost</div>
</template>
<template name="loading">
  Loading...
</template>
```

Here, we specify the global loading template and the page-not-found template. If you only have one layout template for the entire application, you can add it here. This configuration is optional and you can create those templates as per your need. If you configure these options, it is mandatory to create these templates. This configuration will act as a global configuration. For more details, take a look at the `iron-router` package documentation (`https://github.com/iron-meteor/iron-router`).

Since our application is going to be route-driven, which is a common trait of large non-singe-page-applications, we have to define routes for each navigation. This `iron-router` exposes the `Router` object into which we have to define (map) your routes.

In each route, you can provide `path` as the first parameter that is the actual route, an object as the second parameter that can have `name` that is useful for named navigations, `template` that is the actual view, `layoutTemplate` that is optional and is a container for the template mentioned earlier, and `yieldTemplates` that allows you to render multiple templates into the layout specified. There are still a lot of other options we can configure. However, these are the predominant ones. The example for this is as follows:

```
//path is / which is the landing page
Router.route("/", {
  //name is "home"
  name: "home",
  //on route / the layout template will be the template named
  "homeLayout"
  layoutTemplate: "homeLayout",
  //on route / template named "home" will be rendered
  template: "home",
  //render template travelSearch to search section of the layout
  template.
  yieldRegions: {
    travelSearch: {to: "search"}
  }
});
```

Our application will use multiple layouts based on the routes. We will use two different layouts for our application. The first layout (`homeLayout`) is for the landing page, which is a two-column layout. The second layout (`createTravelLayout`) is for travel (bus service) creation and for the reservation page, which is a single-column layout. Also, define the loading and the `notFound` templates if you had configured them.

# Accounts

I am not going to explain much about account (signin/signup). MeteorJS comes, by default, with accounts and the `accounts-ui` package that gives us instant actionable login templates. Also, they provide third-party login services such as Google, Facebook, Twitter, GitHub, and so on. All of these can be made available just by configurations and less amount of code.

Still, they do not suffice for all of our needs. Clients might need custom fields such as the first name, gender, age, and so on. If you don't find the `accounts-ui` package to serve your purpose, write your own. MeteorJS provides extensive APIs to make logging in smooth enough. All you need to do is understand the flow of events. Let us list down the flow of events and actions for implementing a custom login.

# Signup

Create your own route and render the sign up form with all the desired fields. In the event handler of the template, validate the inputs and call `Account.createUser` (`http://docs.meteor.com/#/full/accounts_createuser`) with the e-mail ID and password. The additional user information can go into the profile object. Also, if required, you can change the profile information in the `Account.onCreateUser` callback. You can use `Accounts.config` (`http://docs.meteor.com/#/full/accounts_config`) to set certain parameters such as sending e-mail verification, setting restrictions to account creation (unconditionally or conditionally), login expiration, and secret keys. Obviously, we need to send a verification link to the user by e-mail on signup. Add the e-mail package to the application and provide the SMTP details at the server-side (`http://docs.meteor.com/#/full/email`) as follows:

```
Meteor.startup(function () {
  smtp = {
    username: '',   // eg: bvjebin@meteorapp.com
    password: '',   // eg: adfdouafs343asd123
    server:  '',  // eg: mail.gmail.com
    port: <your port>
  }
  process.env.MAIL_URL = 'smtp://' +
  encodeURIComponent(smtp.username) + ':' +
  encodeURIComponent(smtp.password) + '@' +
  encodeURIComponent(smtp.server) + ':' + smtp.port;
});
```

If you are using the default e-mail verification, which is good to use, you can customize the e-mail templates by adding the following code to the server that is self-explanatory:

```
Meteor.startup(function() {
  Accounts.emailTemplates.from = 'Email Support
  <support@bookMyTravel.com>';
  Accounts.emailTemplates.siteName = 'Book My Travel';
  Accounts.emailTemplates.verifyEmail.subject = function(user) {
    return 'Confirm Your Email Address';
  };
  /** Note: if you need to return HTML instead, use .html instead
  of .text **/
  Accounts.emailTemplates.verifyEmail.text = function(user, url) {
    return 'click on the following link to verify your email
    address: ' + url;
  };
});
```

**[ 9 ]**

When the verification link is visited by the user, callbacks registered with the `Accounts.onEmailVerificationLink` method will be called. If you want to prevent auto-login, call the `Account.createUser` method in a server rather than in a client. The `Accounts.validateNewUser` method can be used to register callbacks, which will validate the user information. Throwing an error from this callback will stop user creation.

## Signin

The `Meteor.loginWithPassword` method (`http://docs.meteor.com/#/full/meteor_loginwithpassword`) needs to be called if you have a custom login form. There are helpers such as `Accounts.validateLoginAttempt`, `Accounts.onLogin`, and `Accounts.onLoginFailure` to perform various actions in the middle via callbacks, if needed. Once logged in, `Meteor.user()` and `Meteor.userId` will have the user information. To check whether the user is logged in or not, you can use `if(Meteor.userId)`. In the `Account.onLogin` method, we can register a callback that will navigate to a desired route on successful login.

The accounts package also provide various methods such as `changePassword`, `forgotPassword`, `sendResetPasswordEmail`, `resetPassword`, `setPassword`, and `onResetPasswordLink` that completes the accounts implementation. One can make use of these methods to customize the login as required.

I hope all these details help you in creating a custom account management module.

## Creating a bus service

Though this section is not going to be our landing page, we will develop the bus service creation part first, which will give us enough data to play around the listing section.

While developing a server-based application, we can start with routes, then the models, followed by the interfaces, and, lastly, the server calls. Thinking in this order will give us a fair idea to reach our goal.

Let's define a route. The route name is going to be `createTravel`. The URI or path is `/create-travel`, the layout can be `createTravelLayout` and the template can be `createTravel`. The route will look like the following code snippet; copy it to `routes.js.Router.route`:

```
("/create-travel", {
  name: "createTravel",
  layoutTemplate: "createTravelLayout",
  template: "createTravel"
});
```

Now, we need to define our collections. In the first place, we need a collection to persist our travel service (bus services).

Create a file, `collections.js`, in the commons directory so that we can access this collection both in the server and client. This is a big advantage of isomorphic applications. You don't have to define collections in two places. Place the following snippet in the `collections.js` file:

```
BusServices = new Meteor.Collection("busservice");
```

Mind the global variable `BusServices` that has to be global so that it can be accessed across the application. Using a global variable is bad practice. Still, we have to live with it in the case of MeteorJS. Where it is avoidable, avoid it.

MeteorJS will create the `busservice` collection in the database on the first insertion. We get a handle to this collection using the `BusServices` variable. It's time to decide all the fields we need to persist in the collection. We will have `_id` (auto-generated by MeteorJS), `name`, `agency`, `available_seats`, `seats`, `source`, `destination`, `startDateTime`, `endDateTime`, `fare`, `createdAt`, and `updatedAt`.

You can add whatever you feel that should be present. This part helps us to create the UI to get the user inputs. Let's create a form where the user inputs all these details.

As mentioned in the route, we need a layout template and a view template to display the form in the client. Create a directory with the name `createTravel` in the client directory and add a layout file `createTravelLayout.html`. Our layout will be as follows:

```html
<!-- name attribute is the identifier by which templates are
identified -->
<template name="createTravelLayout">
  <div class="create-container">
    <header class="header">
      <h1>{{#linkTo route="home"}}BookMyTravel{{/linkTo}}</h1>
      <ul class="nav nav-pills">
        <li>{{#linkTo route="home"}}List{{/linkTo}}</li>
      </ul>
    </header>
    <section class="create-container__section">
      {{> yield}}
    </section>
    <footer class="footer">Copyright @Packt</footer>
  </div>
</template>
```

One important code in the template is `{{>yield}}`. This is a built-in helper/ placeholder where the actual view template will be placed, which means the `createTravel` template will be placed in `{{>yield}}` as a part of this layout.

Create the view template file, `createTravel.html`, in the same directory as the layout and paste the following code:

```
<template name="createTravel">
  <div class="row col-md-6 col-md-offset-3 top-space">
        <div class="col-md-12 well well-sm">
            <form action="#" method="post" class="form"
            id="signup-form" role="form">
              <div class="error"></div>
              <input class="form-control" name="name"type="text"
              required />
          <input class="form-control" name="agency"required />
          <input class="form-control" name="seats"type="number"
          required />
            <div class="row">
                <div class="col-xs-6 col-md-6"><input class="form-
                control" name="startpoint"type="text" required
                /></div>
                  <div class="col-md-6"><input class="form-
                  control" name="endpoint" type="text" required
                  /></div>
              </div>
              <div class="row">
                  <div class="col-md-3"><input class="form-control"
                  name="startdate" type="date" required /></div>
      <div class="col-md-3"><input class="form-control"
      name="starttime" type="time" required /></div>
      <div class="col-md-3"> <input class="form-control"
      name="enddate" type="date" required /></div>
      <div class="col-md-3"><input class="form-control"
      name="endtime" type="time" required /></div>
                  </div>
                  <input class="form-control" name="fare"
                  type="number" required />
                  <button class="btn btn-lg btn-primary btn-block"
                  type="submit">Create</button>
              </form>
          </div>
      </div>
  </template>
```
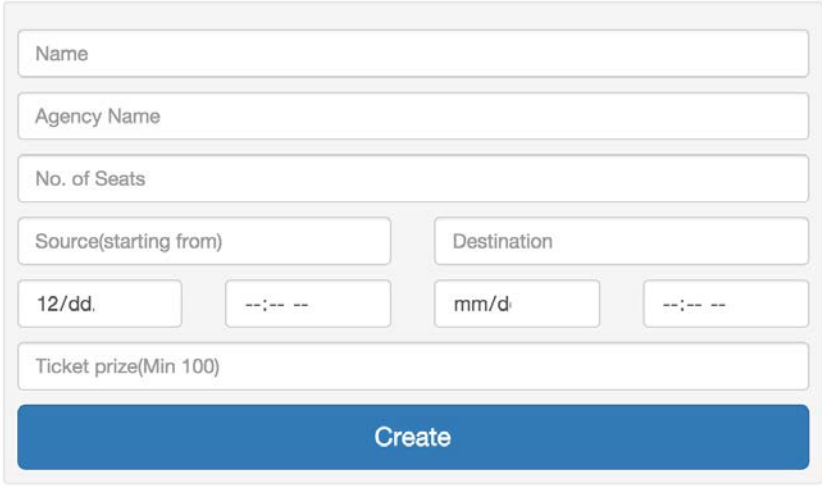
We are almost there. We need to see how this looks. Start the meteor server using the `meteor` or `meteor -p <port number 3001>` command. Navigate to `localhost:3000/create-travel` in your browser.

You will see the form, but the layout is broken. Some styles are needed. Create a file, `styles.css`, in `assets` directory and add the following styles to it. I am using a flex box for the layout, along with a twitter bootstrap:

```css
body { height: 100vh; display: flex;}
.header, .footer {
  flex: 0 1 auto;
  height: 60px;
  border-top: 1px solid #ccc;
  background: #ddd;
  display: flex;
  align-items: center;
  justify-content: space-between;
  padding: 0 20px;
}
.header {border-bottom: 1px solid #aaa;}
.header h1 { margin: 0; }
.footer {height: 40px; text-align: center; justify-content:
  center;}
.home-container, .create-container {width: 100%;display:
  flex;flex-direction: column;}
.home-container__section, .create-container__section {display:
  flex;flex: 1 1 auto;overflow: auto;}
.home-container__section__left {flex: 1 1 auto;box-shadow: inset
  0px 0px 4px 1px #ccc;}
.main {overflow: auto;}
.bus-list {margin: auto;}
.bus-list__header {background-color: #ddd;height: 45px;}
.bus-list__row {border-bottom: 1px solid #ccc;height: 50px;}
.bus-list__row-empty {padding: 20px;}
.bus-list__row__col {text-align: center;border-right: 1px solid
  #fff;height: 100%;display: flex;justify-content:
  center;align-items: center;
}
.bus-list__row__col.last {border: 0;}
.bus-list__body {background-color: #efefef;}
.accounts-container__row {  margin-top: 7em; }
.busView {display: flex;flex-direction: column;padding: 20px 0;}
.busView__title {flex: 0 1 auto;height: 50px;}
```

```
.busView__seats {margin: 0 auto;}
.busView__left, .busView__right {border: 1px solid #ccc;}
.busView__book {padding-top: 2em;}
.busView__seat {text-align: center;vertical-align: middle;height:
  25px;width: 25px;border: 1px solid #ccc;margin: 13px;cursor:
  pointer;display: inline-block;}
.busView__seat.blocked {background-color: green;}
.busView__seat.reserved {background-color: red;}
.busView__divider {display: inline-block;}
.busView__divider:last-child {display: none;}
.top-space {margin-top: 5em;}
.error {color: red;padding-bottom: 10px;}
.clear {clear: both;}
.form-control { margin-bottom: 10px; }
```

This has all the necessary styles for the whole application. Visit the page in the browser and you will see the form with styles applied and layout fixed, as shown in the following image. MeteorJS refreshes the browser automatically when it detects a change in the files:



The last part of the create section is persistence. We have to collect the input on submit, validate it, and call the server to persist it. We should try to avoid direct database insertions from the client.

To collect data from the client, we will create a helper file, `createTravel.js`, in the `createTravel` directory and add the following code to it:

```
Template.createTravel.events({
  "submit form": function (event) {
    event.preventDefault();
    //creating one object with all the properties set from user
    input
    var busService = {
      name: event.target.name.value,
      agency: event.target.agency.value,
      seats: parseInt(event.target.seats.value, 10),
      source: event.target.startpoint.value,
      destination: event.target.endpoint.value,
      startDateTime: new Date(event .target.startdate.value+"
      "+event.target.starttime.value),
      endDateTime: new Date(event .target.enddate.value+"
      "+event.target.endtime.value),
      fare: event.target.fare.value
    };
    //Checking if start time is greater than end time and throwing
    exception
    if(busService.startDateTime.getTime() >
    busService.endDateTime.getTime()) {
      $(event.target).find(".error").html("Start time is greater
      than end time");
      return false;
    }
    //Server call to persist the data.
    Meteor.call("createBusService", busService, function(error,
    result) {
      if(error) {
        $(event.target).find(".error").html(error.reason);
      } else {
        Router.go("home");
      }
    });
  }
});
```

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

MeteorJS provides the `Template` global variable that holds the template objects in the page. So far, we have two templates, the `createTravel` and `createTravelLayout` templates. One can add events and helpers to these templates using these objects. If you look at the preceding code snippet, we are attaching a submit handler to the form we created. One can refer any template using the name of the templates. Everything else is pretty straightforward. By default, jQuery is available inside the template helpers, and if you wish, you can use it for DOM data retrieval.

In the submit handler, all we do is, collect the filled form data and pack it in an object. You can validate if you need it right here. There is a validation which checks for the start time to be greater than the end time and stops proceeding to call the server. The rest of the fields are validated by HTML5 form attributes.

The important part of the preceding code snippet is the last few lines, which is the call to the server. Now is the time to create the server handler.

Create a file, `createTravel.js`, in the `server` directory and add the following code snippet to the file:

```
Meteor.methods({
  createBusService: function(busService) {
    if(!busService.name) {
      throw new Meteor.Error("Name cannot be empty");
    }
    if(!busService.agency) {
      throw new Meteor.Error("Agency cannot be empty");
    }
    if(!busService.seats) {
       throw new Meteor.Error("Seats cannot be empty");
    }
    busService.createdAt = new Date();
    busService.updatedAt = null;
    busService.available_seats = parseInt(busService.seats, 10);
    BusServices.insert(busService);
  }
});
```

We have created a server method called `createBusService`, which takes the `busService` object, does some validation, and then adds `createdAt`, `updatedAt` and `available_seats`. Finally, it inserts the objects to the database. The `BusServices` object is the collection variable we created sometime back, if you remember.

It is always good to do the validation at the server end as well. This is because, at the developer front, it is always said, not to trust the client. They can modify a client-side validation easily and make the client post the irrelevant data. As developers, we have to do all the necessary validations at the server end.

This server method is called from the client in the submit handler using `Meteor.call` with three arguments: the server method name, parameters to the server, and callback.

The callback is called with two parameters: error and result. If there is an error, result is undefined; if the result is present, the error is undefined. One can do post actions based on these parameters inside the callback; for example, in our case, we navigate to the home route if all went well, or else we show the error to the user at the top of the form.

Try filling the form now and check whether everything is fine. If the form data is inserted to the database, you will be taken to `localhost:3000/`. Here, if you have configured `notFoundTemplate` in the router, it will be rendered. If not, you will see the exception:

**Oops, looks like there's no route on the client or the server for url: "http://localhost:3000/"**.

The reason for this is that we haven't yet defined or mapped the `/` route to any template so far. How to verify that the data is saved to the database?

Go to your project terminal and run the `meteor mongo` command. This will log you into the mongo database console. Run the `db.busservice.find().pretty()` query. This will show all the inserted data in the mongo console.

# List and search

In this section of the application, we will show the list of buses available with their details and also we are going to have a reactive search for the list.

Let's start with a route for the list. Add the following to the `routes.js` file in the commons directory after the `createTravel` route, which we created earlier:

```
Router.route("/", {
name: "home",
layoutTemplate: "homeLayout",
template: "home",
yieldRegions: {
  travelSearch: {to: "search"}
}
});
```

This is the home route. When you hit `localhost:3000/`, you know what will happen. Pretty much easy to remember, right?

Under the client directory, create a subdirectory called `home`. The directory name has nothing to do with the route. This directory will have the files to display the list of bus services. Let's create `homeLayout.html`, `home.html`, and `homeHelper.js`.

In `homeLayout.html` file, add the following code:

```
<template name="homeLayout">
  <div class="home-container">
    <header class="header">
      <h1>{{#linkTo route="home"}}Booking{{/linkTo}}</h1>
      <ul class="nav nav-pills">
        <li>
          {{#linkTo route="createTravel"}}Create{{/linkTo}}
        </li>
      </ul>
    </header>
    <section class="home-container__section">
      <div class="home-container__section__left container-fluid">
        {{> yield region="search"}}
      </div>
      <div class="main">
        {{> yield}}
      </div>
    </section>
    <footer class="footer">Copyright @Booking</footer>
  </div>
</template>
```

In `home.html` file, add the following two templates (list and search):

```
<template name="home">
  <div class="container bus-list">
    <div class="row bus-list__row bus-list__header">
      <div class="bus-list__row__col col-md-3">Bus</div>
      <div class="bus-list__row__col col-md-1">Available
      seats</div>
      <div class="bus-list__row__col col-md-1">Start point</div>
      <div class="bus-list__row__col col-md-1">End point</div>
      <div class="bus-list__row__col col-md-2">Start time</div>
      <div class="bus-list__row__col col-md-2">Reaching time</div>
      <div class="bus-list__row__col col-md-1">Fare</div>
```

```
        <div class="bus-list__row__col last col-md-1">Book</div>
      </div>
      <div class="row bus-list__body">
      {{#if hasItem}}
        {{#each list}}
        <div class="bus-list__row">
        <div class="bus-list__row__col col-md-3">{{name}}<br
        />{{agency}}</div>
        <div class="bus-list__row__col col-md-
        1">{{available_seats}}/{{seats}}</div>
        <div class="bus-list__row__col col-md-1">{{source}}</div>
        <div class="bus-list__row__col col-md-
        1">{{destination}}</div>
        <div class="bus-list__row__col col-md-2">{{humanReadableDate
        startDateTime}}</div>
        <div class="bus-list__row__col col-md-2">{{humanReadableDate
        endDateTime}}</div>
        <div class="bus-list__row__col col-md-1">{{fare}}</div>
        <div class="bus-list__row__col last col-md-1"><a
        href="/book/{{_id}}">Book</a></div>
        </div>
        <div class="clear"></div>
         {{/each}}
      {{else}}
        <div class="row bus-list__row bus-list__row-empty">
        <div class="bus-list__row__col last col-md-12
        text-center">No buses found</div>
        </div>
      {{/if}}
      </div>
    </div>
</template>
<template name="travelSearch">
  <div class="col-xs-12 col-sm-12 col-md-12 text-center top-space
  well well-sm">
  Search
  </div>
  <div class="col-xs-12 col-sm-12 col-md-12 well well-sm">
      <div class="form" id="signup-form">
      <div class="error"></div>
      <input class="form-control" name="startpoint"
      placeholder="Source(starting from)" type="text" required="">
      <input class="form-control" name="endpoint"
      placeholder="Destination" type="text" required="">
```

```
        <input class="form-control" name="startdate"
        placeholder="Date" type="date" required="">
        <input class="form-control" name="fare" placeholder="Max
        prize" type="number" required="">
        </div>
    </div>
</template>
```

The last thing to add is the helper file. Create `homeHelper.js` and add the following code:

```
Template.home.helpers({
  list: function() {
    return BusServices.find();
  },
  hasItem: function() {
    return BusServices.find().count();
  },
  humanReadableDate: function (date) {
    var m = moment(date);
    return m.format("MMM,DD YYYY HH:mm");
  }
});
```

Previously, we have attached events using the `Template` object. Now, we have helpers with which you can pass the customized data to the template. Visit the browser and you will find the empty list and the search form.

Wait, we have data in our database. Why didn't it show up in the list? Here comes the data access pattern that we should follow. By default, MeteorJS doesn't send the data to the client when there is no `autopublish` package. This is a good thing too. When we create a large application, we might not need to send all the database data to the client. The client will be interested in only a few, so let's play with that few.

MeteorJS provides the `publish` and `subscribe` methods to publish the required data from a server and subscribes those publications from a client. Let us use these methods to get the data.

In `createTravel.js` file at the server directory, add the following code:

```
Meteor.publish("BusServices", function () {
    return BusServices.find({}, {sort: {createdAt: -1}});
});
```

With this piece of code, the server publishes the `busservices` collection sorted by the `createdAt` date with the `BusServices` identifier.

In the client, to subscribe this publication, add the following line at the top of the `homeHelper.js` file:

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

After this addition, you will see that the list has the trips that you created earlier, as shown in the following screenshot. Now, go create some more travels that we will use for search:

| Search | Bus | Available seats | Start point | End point | Start time | Reaching time | Fare | Book |
|---|---|---|---|---|---|---|---|---|
| | SUper Royal Travels | 30/30 | Bangalore | Kolkatta | Aug,02 2015 02:01 | Aug,03 2015 01:00 | 3000 | Book |
| Source(starting from) | Royal Royal Travels | 40/40 | Delhi | Mumbai | Jul,02 2015 01:00 | Jul,03 2015 02:01 | 2000 | Book |
| Destination | | | | | | | | |
| mm/dd/yyyy | | | | | | | | |
| Max prize | | | | | | | | |

Also, add the following event handler to `homeHelper.js` file:

```
Template.travelSearch.events({
  "keyup input": _.debounce(function(e) {
    var source = $("[name='startpoint']").val().trim(),
      destination = $("[name='endpoint']").val().trim(),
      date = $("[name='startdate']").val().trim(),
      fare = $("[name='fare']").val().trim(),
      search = {};
    if(source) search.source = {$regex: new RegExp(source),
    $options: "i"};
    if(destination) search.destination = {$regex: new
    RegExp(destination), $options: "i"};
    if(date) {
      var userDate = new Date(date);
      search.startDateTime = {
        $gte: userDate,
          $lte: new Date(moment(userDate).add(1,
          "day").unix()*1000)
      }
    }
    if(fare) search.fare = {$lte: fare};
    BusServices.find(search, {sort: {createdAt: -1}});
  }, 200)
});
```

This is a text box event handler that is debounced by `200` ms for improving performance. The handler collects the search field's data and accumulates it into an object and queries the collection. Do you see any change in the list when you search? It won't, and that is where we get things wrong. Although we have subscribed the `busservice` collection, MiniMongo holds the data from the server. From one template, when you query the collection, the result doesn't update an other template. We are not changing the subscription itself, instead just the local query. Then, how do we make things happen?

MeteorJS has some data sources that are reactive, by default. For example, database cursors and session variables. However, we need more, don't we? We need custom variables to be reactive so that we can also do the magic. MeteorJS' core team developers have thought about it and provided us with a simple package called `reactive-var`.

Add the `reactive-var` package to the application using the `meteor add reactive-var` command. The logic behind reactive variables is simple—when the value changes, all the instances including the templates will get them immediately.

Simple example of reactive variables is as follows:

```
var reactVar = new ReactiveVar(2); //2 is default value that can
  be set in the constructor parameter.
reactVar.set(4); //will update the value of all instance where the
  reactVar variable is used.
```

Let's use it in our application. In `homeHelper.js`, add the following code snippet before the `Template.home.helpers` method:

```
var busServicesList = new ReactiveVar([]);
Template.home.onCreated(function() {
  busServicesList.set(BusServices.find({}));
});
```

This initializes the reactive variable `busServicesList` with an empty array and then sets the complete `busservices` collection when the `home` template' `onCreated` callback is called. We will use this reactive variable in the templates, instead of the actual collection query cursor. Change the `list` method in the template helpers to the following:

```
list: function() {
  return busServicesList.get();
},
hasItem: function() {
  return busServicesList.get().count();
},
```

Whenever there is a search, we have to update this reactive variable, which will instantly update the template. It is that simple.

Go to the events handler of the search template and replace `BusServices.find(search, {sort: {createdAt: -1}});` with `busServicesList.set(BusServices.find(search, {sort: {createdAt: -1}}));`.

Perform a search and see the update instantly. Pat yourself on the back. You have accomplished a big job.

This isn't the only approach to implement a search. You can add a route-based implementation, which will subscribe to collection every time you change the route, based on search parameters. However, that isn't efficient because the client has all the data, but still we are asking the server to send the data based on the search parameter.

# Reservation

We have reached the last part of the application. We have to allow the user to block or reserve seats in the bus. Also, these actions must be instantaneous to all users, which means both blocking and reservation should reflect in all the users' browsers immediately so that we don't have to manually resolve users' seat selection conflicts. Here, you will see the power of MeteorJS' reactivity:

As usual, we will create a route. Add the following code snippet to `routes.js` as done earlier:

```
Router.route("/book/:_id", {
  name: "book",
  layoutTemplate: "createTravelLayout",
  template: "bookTravel",
  waitOn: function () {
    Meteor.subscribe("BlockedSeats", this.params._id);
    Meteor.subscribe("Reservations", this.params._id);
  },
  data: function() {
    templateData = {
      _id: this.params._id,
      bus: BusServices.findOne({_id: this.params._id}),
      reservations: Reservations.find({bus:
      this.params._id}).fetch(),
      blockedSeats: BlockedSeats.find({bus:
      this.params._id}).fetch()
    };
```

```
        return templateData;
    }
  });
```

Hope you guessed what we are up to. On each record in the list of the listing page, we have a link, which on click will hit this route and the relevant seating layout will appear for the user to block or reserve the seats. What are those new properties in the route? The `waitOn` property keeps the template rendering to wait until the subscription is completed. We do this because subscriptions are asynchronous. We pass the `_id` attribute of the bus service to the route and this is passed to the subscription. Similarly, the `data` property is the place where we can prepare the data that needs to be passed to the templates. Here, we prepare bus details, reservations of the selected bus, and seats that are blocked in this bus; then, send them to the template.

Where will we store all the reservation data? For this, we need a collection. So, let's go to `collections.js` and add the following:

```
Reservations = new Meteor.Collection("reservations");
```

This collection holds seats for reservation. What about blocking? Let's have a collection for that too. Add the following line to the `collections.js` file:

```
BlockedSeats = new Meteor.Collection("blockedSeats");
```

Create the `bookTravel` directory in the client and add `bookTravel.html.` file Add the following template code into the file. As you have guessed, we are reusing the same `createTravelLayout` template as a layout for this interface:

```html
<template name="bookTravel">
  <div class="container busView">
    <div class="row text-center busView__title">{{bus.name}}
    <br />{{bus.agency}}</div>
    <div class="row col-md-4 busView__seats">
      <div class="col-md-12 busView__left">
      {{#each seatArrangement}}
        <div class="col-md-12 row-fluid">
        {{#each this}}
        <div id="seat{{this.seat}}" class="busView__seat
        {{blocked}} {{reserved}}">{{this.seat}}</div>
        {{#if middleRow}}
          <div class="busView__divider col-md-offset-3"></div>
        {{/if}}
        {{/each}}
        </div>
      {{/each}}
```

```
        </div>
      </div>
      <div class="row text-center busView__book"><button id="book"
      class="btn btn-primary">Book My Seats</button></div>
    </div>
</template>
```

This template will draw seats in rows and columns based on the total seats stored in the `busservices` collection document. The idea is to get the data of the interested bus service, reservations made so far for the same bus, and seats blocked at the moment for the same bus. Once we get all the data, we draw the seating layout with the blocked and reservation information.

We need a few helpers and event handlers to get this entire stuff done. Create `bookTravelHelper.js` inside the `bookTravel` directory and add the following code:

```
Template.bookTravel.helpers({
  seatArrangement: function() {
    var arrangement = [],
      totalSeats = (this.bus || {}).seats || 0,
      blockedSeats = _.map(this.blockedSeats || [], function(item)
      {return item.seat}),
      reservedSeats = _.flatten(_.map(this.reservations || [],
      function(item) {return _.map(item.seatsBooked,
      function(seat){return seat.seat;});})),
      tmpIndex = 0;
      Session.set("blockedSeats", this.blockedSeats);
    arrangement[tmpIndex] = [];
    for(var l = 1; l <= totalSeats; l++) {
      arrangement[tmpIndex].push({
        seat: l,
        blocked: blockedSeats.indexOf(l) >= 0 ? "blocked" : "",
        reserved: reservedSeats.indexOf(l) >= 0 ? "reserved" : "",
      });
        if(l % 4 === 0 && l != totalSeats) {
          tmpIndex++;
          arrangement[tmpIndex] = arrangement[tmpIndex] || [];
        }
      }
    return arrangement;
  },
  middleRow: function () {
    return (this.seat % 2) === 0;
  }
});
```

```
Template.bookTravel.events({
  "click .busView__seat:not(.reserved):not(.blocked)": function
  (e) {
    e.target.classList.add("blocked");
    var seat = {
      bus: Template.currentData().bus._id,
      seat: parseInt(e.target.id.replace("seat", ""), 10),
      blockedBy: ""
    };
    Meteor.call("blockThisSeat", seat, function(err, result) {
      if(err) {
        e.target.classList.remove("blocked");
      } else {
        var blockedSeats = Session.get("blockedSeats") || [];
        blockedSeats.push(seat);
        Session.set("blockedSeats", blockedSeats);
      }
    });
  },
  "click #book": function() {
    var blockedSeats = Session.get("blockedSeats");
    if(blockedSeats && blockedSeats.length) {
      Meteor.call("bookMySeats", blockedSeats, function (error,
      result) {
      if(result) {
          Meteor.call("unblockTheseSeats", blockedSeats,
          function() {
            Session.set("blockedSeats", []);
          });
        } else {
          alert("Reservation failed");
          console.log(error);
        }
      });
    } else {
      alert("No seat selected");
    }
  }
});
```

The helper method `seatArrangement` will aggregate the reservation and the blocked seats data along with the seat information in a way which will be easy to render. The `middleRow` helper method is used to do a small modulus operation to have a gap between the second and the third column.

The event handler on each seat will call the server to persist the blocking action. Clicking on the book button will call the server to reserve the blocked seats.

Let's get into the server section. We have to publish both the newly created collections to the client and also add a method that the client is calling to persist the data.

Create the `reservations.js` file in the server directory and add the following code:

```
Meteor.methods({
  /**
    seatsBooked: [{seat: #}]
    bus
    createdAt
    updatedAt
  **/
  bookMySeats: function(reservations) {
    var insertRes = reservations.map(function(res) {
      return {
        seat: res.seat
      }
    });
    return Reservations.insert({
      bus: reservations[0].bus,
      seatsBooked: insertRes,
      createdAt: new Date(),
      updatedAt: null
    }, function (error, result) {
      console.log("Inside res insert", arguments);
      if(result) {
        BusServices.update({_id: reservations[0].bus}, {
          $set: {
            updatedAt: new Date()
          },
            $inc: {
            available_seats: -insertRes.length
          }
        }, function() {});
      }
    });
  }
});
Meteor.publish("Reservations", function (id) {
    return Reservations.find({bus: id}, {sort: {createdAt: -1}});
});
```

Similarly, create the `bookTravel.js` file and add the following code:

```
Meteor.methods({
  blockThisSeat: function(seat) {
    var insertedDocId;
    seat.createdAt = new Date();
    seat.updatedAt = null;
    BlockedSeats.insert(seat, function(error, result) {
      if(error) {
        throw Meteor.Error("Block seat failed");
      } else {
        insertedDocId = result;
      }
    });
    Meteor.setTimeout(function() {
      BlockedSeats.remove({_id: insertedDocId});
    }, 600000);// 10 mins
  },
  unblockTheseSeats: function(seats) {
    seats.forEach(function (seat) {
      BlockedSeats.remove({_id: seat._id});
    });
  }
});

Meteor.publish("BlockedSeats", function (id) {
  return BlockedSeats.find({bus: id});
});
```

If you look at the event handlers that we created for the `bookTravel` template, you will find these method calls. All they do is persist data. Also, a blocked seat will be released after 10 minutes and you can see that happening in the `blockThisSeat` server method. A timer is registered on each call. Let us see things in action.

Open the same booking page in another browser. You will find the seat arrangement and reservation data, if any, as shown in the following image:
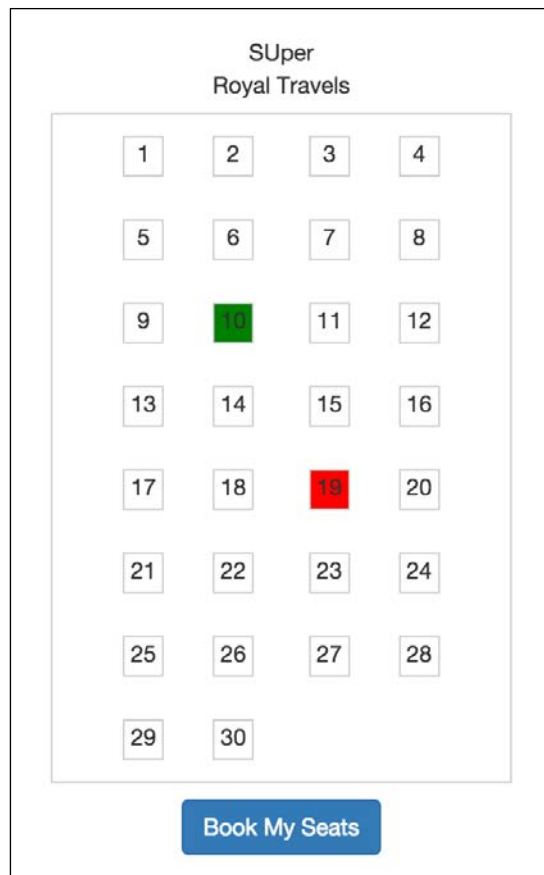
Reserve or block some seats and visit the page in the other browser. You will see the changes instantly appearing here. Also, our event handler will not allow the user on any end to choose seats that are reserved or blocked. This is the actual power of MeteorJS. Instant reactivity on any data change to all clients without any special effort from the developer will drastically reduce your development effort.

# Summary

I hope you have enjoyed this chapter. There is a lot of scope to improve the application in terms of features. Go play around and implement additional features and get your hands dirty. I will leave it to your imagination. This chapter has come to an end. Let's summarize what we have learned so far. MeteorJS is built by integrating various packages. MeteorJS employs various components on the server, client, and channel to build the applications. MeteorJS provides extensive and flexible APIs to create customized logins. We can define named routes, and thereby associate templates and layouts with the route. We have also learned how to use multiple layouts in the application. We also learned to code database operations, query for search operations, sever side method calls, and custom reactive variables to make the application more lively and reactive.

What we have learned so far is good. However, there is a lot we can improve in this whole process. In the next chapter, we'll learn how to develop MeteorJS application like a pro.

# 2
# Developing and Testing an Advanced Application

The popularity of frameworks such as Ruby On Rails is because of two reasons. One is the ecosystem where one can find plugins and utilities to solve common problems and other is the scaffolding. MeteorJS has also earned countless contributors, which ultimately created a wonderful ecosystem. Also, as an added advantage, developers can use the Node.js ecosystem. Developers can find a lot of useful MeteorJS packages in the MeteorJS ecosystem, which is called atmosphere. Visit atmosphere (`https://atmospherejs.com/`) once and check the enormous amount of packages available for development.

Scaffolding enables anyone to kick-start a big application in minutes. Frameworks such as Ruby On Rails, YII, and Zend have mature scaffolding, which helps anyone with less knowledge about the framework to get started easily. Scaffolding reduces a lot of work and, therefore, developers can use their precious time in developing the application logic rather than spending time on less logical work, such as creating and managing the views, collections, and routes. Although there is no in-built scaffolding in MeteorJS, there are tools that can help to scaffold your applications.

Moreover, if you have worked with any of the previously mentioned frameworks or even with Backbone.js, and Ember.js, you will find that there is a separate entity called **Model**; you define the proper structure that is very similar to or the same as the columns in the database. The important reason behind this is maintainability. It gives a proper shape to the data so that it can be referenced without any chaos in the future. In MeteorJS, we have no built-in way to structure the data. However, there are packages that will help us to define the schema, validate the data before insertion, perform type checking, and define authorization rules to perform database-level operations.

Precisely, in this chapter, we will learn the following by redoing the same application from the previous chapter:

- Scaffolding in MeteorJS
- Collections and schema management
- Securing database operations at schema level
- Form creation and validation based on schema
- Debugging the application
- Testing the MeteorJS application

# Scaffolding in MeteorJS

As I have mentioned earlier, there is no built-in scaffolding in MeteorJS. However, there are tools that help us scaffold our application. Although there are a few tools to serve the purpose, we will look into `iron-cli` (`https://github.com/iron-meteor/iron-cli`), which is a scaffolding tool from the makers of the popular package `iron-router`. It is a Node.js module and is still growing; so, we have to watch out for the releases carefully. The `iron-cli` must be installed from NPM using the following command:

```
npm install -g iron-meteor
```

Let me tell you what exactly `iron-cli` does. It provides command-line utilities to scaffold. Instead of using MeteorJS commands directly such as `meteor create myapp`, we will use the `iron-cli` commands to generate the application. The `iron-cli` wraps the MeteorJS application inside it and provides us the ability to manage the application. Let's try it out.

After installing `iron-meteor` using the preceding NPM command, try `iron help` in the terminal. It should show the list of commands available to run the generator options, as shown in the following screenshot:

```
jebin@Jebins-MacBook-Pro:[server][staging]$ iron help

A command line scaffolding tool for Meteor applications.

Usage: iron <command> [<args>] [<opts>]

Examples:
  > iron generate:scaffold todos
  > iron generate:view todos/todo_item

The default command will run your meteor application.

Commands:
```

| build | Build your application into the build folder. |
| create | Create a new iron meteor project. |
| generate | Generate different scaffolds for your project. |
| help | Get some help. |
| init | Initialize your project structure. |
| migrate | Migrate to the new iron project structure. |
| run | Run your app for a given environment. |

The `iron-cli` actually proxies most of the meteor commands and so we will use commands such as `iron add <package>`, `iron remove <package>`, `iron run`, `iron list`, and so on. These commands will internally call the MeteorJS commands. The reason for this proxying is to keep track of the application changes so that iron-cli can take full control of the application.

Let's start recreating the same application that we did in the previous chapter, but using all these advanced techniques. Recreating the same application will help you relate things and understand how much we have improved things.

# Recreating the travel booking application

Create a new MeteorJS application using the `iron create BookMyTravel2` command. Remove the `autopublish` and `insecure` packages using the `iron remove <package name>` command. Visit the directories and files created inside the `BookMyTravel2` directory. As I said, iron-cli wraps the MeteorJS app inside itself and in this case, it is the `app` directory. The `app` directory is the actual MeteorJS app. If you want to run any MeteorJS-specific commands, you have to go into the `app` directory and run the commands. Both the mongo database and the `.meteor` directory reside inside the `app` directory. The unfortunate thing is that this scaffolding is tightly coupled to `iron-router`. We have to do some rework if we wish to use any other routing solution such as `FlowRouter`. However, this is not a big deal if you know what to change and where to change. For now, let' stay on track and build the application with `iron-router` itself.

Run the `iron run` command in the terminal and it will start the application. If you watch the directories and files created, you will find that the home page-related scaffolding has been done already. Route, controller, action, layout, and the template for home are created in different directories. Let's get familiar with those directories. You can ignore the `.iron` directory.

At the root level of the application, there are the `app`, `bin`, `build` and `config` directories. The `app` directory is the developer's concern. The `bin` and `build` directories are for iron-cli's internal work; and the `config` directory is where we can define environmental variables and other platform settings.

# The app directory

The `app` directory has sub-directories and files where we do the real coding. We can find sub-directories such as `client`, `lib`, `packages`, `private`, `public`, and `server`, inside the `app` directory.

# Client

As you know, the `client` directory is where we do the client-side stuff. The `client` directory has the collections, `lib`, `stylesheets`, and `templates` directory. You can make out for yourself what these directories are meant for. Inside the `template` directory, there is a `home` directory where the `home` template, `home` template handlers, and CSS are present. You will see the skeletons inside those files. Similarly, the `layouts` directory has `master_layout` that has a master layout template and its handlers. We can create various templates inside this `layouts` directory and use them in the application. It is well organized, so anyone who searches for layouts will just have to look into this directory. The `shared` directory has a `loading` template and a `notFound` template. We can keep any template or a related piece of code that can be shared across the application inside this directory.

# lib

The files under this section will be run both in the server and the client. The `iron-cli` creates the `collections` and `controllers` directory here. When we use `iron-cli` to generate collections, it will create collections inside the `collections` directory under `lib`. The `controllers` directory will have to hold all the generated controllers. These controllers are extended from `RouterController`, which is a part of `iron-router`. These controllers are invoked from the router handlers. Inside the `controllers` directory, we can see that `home_controller.js` is already created for us by the `iron-cli`. If you look at it, you can understand for yourself as to what is going on. The `routes.js` file is connected to the controllers. In each route, we can specify which method of which controller to invoke when the route is visited. As per the `routes.js` file and the `home` route, when one hits `localhost:3000/`, it will call the `action` method in `HomeController`. Before the `action` executes, the `subscriptions` method will be invoked. Finally, `methods.js` is the file where we can define methods that can be used both in the server and the client.

# Private and public packages

The `packages` directory is for the custom packages that we create for our application. We will cover this in greater detail in an upcoming chapter. The `private` directory is an asset directory for the server and the `public` directory is an asset directory for the client.

# Server

Here, we have the `collections`, `controllers`, and `lib` directories. Server-only collections will reside in `collections`, server-only controllers will be in `controllers`, and other code can live in the `lib` directory. The `bootstrap.js` file has a hook for the `startup` callback. If some code needs to be executed during the application startup, we can put it inside this file. The `methods.js` file is where we put all server-only methods. Finally, the `publish.js` file is where we put all the publish registration-related code.

Now, we are familiar with the directory structure that `iron-cli` has created for us. The last thing that we need to know to kick-start scaffolding is generators. The `iron-cli` provides generators with which we can create skeleton code for our application anytime. The following are the generators that are available:

```
iron g:scaffold <entity>
```

```
iron g:template <optional-directory/template>
```

```
iron g:controller <optional-directory/controller-name> --where
  "<server/client>"
```

```
iron g:route <route name>
```

```
iron g:collection <collection name>
iron g:publish <publication name>
iron g:stylesheet <stylesheet name>
```

The `iron g:scaffold` command will generate collection, templates, handlers, controller, route, and publication. If you wish to do it in a customized manner, you can use other commands to generate specific components. For example, in our application, we need to create a collection called reservations, but we don't need the controller, route, and templates for that collection. So, we can just use the iron `g:collection reservations` command that only creates a collection inside `lib/collections`.

# Generators for the application

The time has come to use the generators. We will follow the same flow as the previous chapter. We will develop the create travel section first and then the listing, followed by the search and finally the reservation.

## Creating travel

Run the `iron g:scaffold createTravel` command and you will see the following files created and updated:

```
created app/lib/collections/create_travel.js
created app/client/templates/create_travel
created app/client/templates/create_travel/create_travel.html
created app/client/templates/create_travel/create_travel.js
created app/client/templates/create_travel/create_travel.css
created app/lib/controllers/create_travel_controller.js
updated ../BookMyTravel2/app/lib/routes.js
updated ../BookMyTravel2/app/server/publish.js
```

The generator has created a collection (`create_travel.js`) and a publication (in `publish.js`), which are not needed. We can ignore them. However, we will use the other components generated. Along with these files, we need a different layout for the creation screen. Let's create one by adding the `create_travel` directory inside the `layouts` directory. Add `create_travel_layout.html` inside the directory and add the following code to it:

```html
<template name="CreateTravelLayout">
  <div class="create-container">
      <header class="header">
```

```
<h1>
  {{#linkTo route="home"}}
    BookMyTravel2
  {{/linkTo}}
</h1>
<ul class="nav nav-pills">
  <li>{{#linkTo route="home"}}List{{/linkTo}}</li>
</ul>
  </header>
  <section class="create-container__section">
{{> yield}}
  </section>
  <footer class="footer">Copyright @BookMyTravel2</footer>
  </div>
</template>
```

We will add this layout to the `CreateTravelController` in `create_travel_controller.js`. Add the following piece of code as a property along with the subscriptions:

```
layoutTemplate: 'CreateTravelLayout',
```

Now, go to `http://localhost:3001/create_travel` and you will find that the layout reflects in the screen. Let's create the collection and then the form to create the documents in the collection. Run the following generator command to generate the `busservice` collection:

**iron g:collection busservice**

This command will create a `busservice.js` file in `app/lib/collections`. Go visit this file and you will find the `busservice` collection creation code as follows:

```
Busservice = new Mongo.Collection('busservice');
```

Followed by this line, there will be `allow` and `deny` methods that give us the ability to perform **Role-Based Access** (**RBA**) checks. Say, for example, if you want to deny a guest user from creating a bus service, you can do the check in the `deny` method's `insert` part and `return true` if positive. Doing so will deny the document insertion in the collection. You can use these methods to do extensive RBAC for your application. The skeleton is self-explanatory to explain the parameters we will get inside each operation. These `insert`, `update`, and `remove` operation pre-handlers will be executed before doing the respective operations via the collection instance (in our case `Busservice`). Different applications need different kinds of RBA conditions and so I leave this portion to you. Do not forget to make use of this in-built feature of MeteorJS as it is one of the important security features. We can perform authorization checks extensively using these handlers.

It is time to define the schema. If we cannot keep track of schema in a well-structured format, then it is going to be a big pain for maintenance. MeteorJS doesn't have built-in ways to create or define the schema. However, we have packages that can help us do this. We are going to use the `aldeed: collection2` package, which will help us to attach a schema and do validation if required. The Git repo `https://github. com/aldeed` has some important packages that are very useful for development. Add the package by running the following command:

**`iron add aldeed:collection2`**

The schema for the `busservice` collection is as follows:

```
// Validation keys and messages
SimpleSchema.messages({
  source_destination_same: "[label] cannot be same as Starting
  point",
  destination_source_same: "[label] cannot be same as Destination
  point",
  endDateTime_lessthan_startDateTime: "[label] cannot be past to
  start date and time",
  startDateTime_lessthan_endDateTime: "[label] cannot be past to
  start date and time"
});

//Schema for busservice collection
BusServiceSchema = new SimpleSchema({
    name:{
      type: String,
      label: "Name",
      max: 200
    },
    agency:{
      type: String,
      label: "Agency",
      max: 1024
    },
    seats: {
      type: Number,
      label: "Total Seats",
      min: 10,
      max: 50
    },
    source: {
      type: String,
      label: "Starting Point",
```

```
  max: 200,
  custom: function() {
    if((this.value || "").toLowerCase() ==
    (this.field("destination").value || "").toLowerCase()) {
      return "destination_source_same";
    }
  }
},
destination: {
  type: String,
  label: "Destination Point",
  max: 200,
  custom: function() {
    if((this.value || "").toLowerCase() ==
    (this.field("source").value || "").toLowerCase()) {
      return "source_destination_same";
    }
  }
},
startDateTime: {
  type: Date,
  label: "Departure Time",
  min: moment().add(1, "days").toDate(),
  max: moment().endOf("year").toDate(),
  custom: function() {
    if(this.value >= this.field("startDateTime").value) {
      return "startDateTime_lessthan_endDateTime";
    }
  }
},
endDateTime: {
  type: Date,
  label: "Arrival TIme",
  min: moment().add(1, "days").toDate(),
  max: moment().endOf("year").toDate(),
  custom: function() { //custom validation
    if(this.value <= this.field("startDateTime").value) {
//error message identifier added in SimpleSchema.messages api.
      return "endDateTime_lessthan_startDateTime";
    }
  }
},
fare: {
  type: Number,
```

```
      label: "Fare",
      min: 100
    },
    createdAt: {
      type: Date,
      label: "Created At",
      autoValue: function() {
        if (this.isInsert) {
          return new Date;
        }
      }
    },
    updatedAt: {
      type: Date,
      label: "Updated At",
      autoValue: function() {
        if (this.isUpdate) {
          return new Date();
        }
      },
      denyInsert: true,
      optional: true
    },
    available_seats: {
      type: Number,
      label: "Available Seats",
      autoValue: function(doc) {
        if (this.isInsert) {
          return doc.seats;
        }
      }
    },
    createdBy: {
      type: String,
      optional: true,
      autoValue: function() {
        return this.userId
      }
    }
  });


Busservice.attachSchema(BusServiceSchema);
```

Add the preceding code after the `busservice` collection instantiation. The code is self-explanatory. For more details on the `meteor-collection2` package, visit `https://github.com/aldeed/meteor-collection2`. You will find a lot of useful information and methods that can help you to do a better job in schema definition and validation. We can define many kinds of validations right in the schema definition itself, and the error messages can be defined along with the schema. If you go through the schema definition, you can figure out the minimum and maximum validations for some fields and custom validation for the arrival date and time and the departure date and time, which compares the dates. Similarly, one can define any custom validation and can access the document data from the form inside the custom validation methods.

The best part of defining a schema is the form creation. The `aldeed` repository has another package called `AutoForm`, which can read schema and create forms without much effort. Let's install the `AutoForm` package by running the following command:

```
iron add aldeed:autoform
```

Visit the `AutoForm` package documentation at `https://github.com/aldeed/meteor-autoform`. There is a lot of information to craft your forms carefully with plenty of options.

Inside the `create_travel.html` file in `client/templates/create_travel`, add the following template code:

```
{{> quickForm collection="Busservice" id="CreateBusServiceForm"
  type="insert" omitFields="createdBy, updatedAt, createdAt,
  available_seats" buttonContent="Create"}}
```

Visit the browser and you will find that the form is created with less effort. In the preceding code, we have just specified the collection name, which is the instance we created (not the mongo collection name). Along with that, we have specified `id`, `type`. There is an attribute `omitFields` that facilitates to omit fields that we don't want to show in the form. The `buttonContent` attribute value will appear in the submit button of the form. A single line has created a form for us. Pretty timesaving, isn't it?

The styles are missing. Add the `twbs:bootstrap` package to the application. This will add styles to some extent. The forms generated by `AutoForm` are bootstrap complaints with bootstrap-related classes in the markups. We will add the rest of the styles from the previous chapter. Copy the custom styles we had created in the previous chapter to the `main.css` file of this application. Now, the form is usable, but there are some more details we need to understand.

The schema that we had defined has the `Date` type for the fields for arrival time, departure time, `createdAt`, and `updatedAt`, which you might have noticed. They are `Date` and not `DateTime`. There is no `DateTime` type and so the `AutoForm` package will also generate forms with the appropriate fields having the `Date` type and not `Datetime`. In our case, the arrival time and departure time must be `Datetime` and not just `Date`. So, we cannot use the `quickform` component as it is. To solve this problem, when we need customization in the form, the `AutoForm` package has given us ways to define the customized form in compliance with the schema. Replace `{{quickform ...}}` with the following code:

```
{{#autoForm collection="Busservice" id="CreateBusServiceForm"
    type="method" class="container"
    meteormethod="createBusService"}}
    {{> afQuickField name='name'}}
    {{> afQuickField name='agency'}}
    {{> afQuickField name='seats' min="10" max="50"}}
    {{> afQuickField name='source'}}
    {{> afQuickField name='destination'}}
    {{> afQuickField name='startDateTime' type="datetime-local"}}
    {{> afQuickField name='endDateTime' type="datetime-local"}}
    {{> afQuickField name='fare'}}
    <button class="btn btn-lg btn-primary btn-block"
    type="submit">Create</button>
{{/autoForm}}
```

Instead of using `quickForm`, we are using the `autoForm` and `afQuickField` components to create our form. You can learn the syntax and other possible attribute options from the `AutoForm` package documentation. What we have done here is a minimal usage. However, a notable thing is `type` and `meteormethod`. We have specified the type of the form to be `method` and when we use method as type, it is mandatory to mention the method name in the `meteormethod` attribute. What this means is, on submit, the meteor method `createBusService` will be called by passing all the field values. The `afQuickField` component takes the names that must be the keys defined in the schema and other form field attributes. We needed a `datetime` input field and thus we are using the `type="datetime-local"` attribute in the appropriate fields.

Another important feature of the `AutoForm` package is validation and error display. Based on the schema definition, the form values are validated and an appropriate error message is displayed right below the fields. Try wrong inputs in the fields and submit the form. You will see the error messages below the fields. We don't need to wire the error handling and the error message display. They are taken care by the package itself. Also, `AutoForm` provides hooks to perform any pre or post-submission operations, if necessary.

All the client-related work is done and we have to define the server method to insert the form. In our case, we just want to validate and insert the document. Create the `createBusService` method in `app/server/methods.js` as follows:

```
Meteor.methods({
  createBusService: function(busService) {
      busService.createdAt = new Date();
    busService.available_seats = parseInt(busService.seats, 10);
    check(busService, BusServiceSchema); //validates the form data
    against the schema in the server side
    Busservice.insert(busService);
  }
});
```

If all goes well, you will be able to save the data to the collection. Check in the database to confirm.

# Listing and search

Next is the listing part. Visit `http://localhost:3000` and you will find the text **Find me in app/client/templates/home/home**. Our home page is going to be the listing and search page. Let's start adding the layout, then the list template, and wire the data. Things start from the route. Our route for home page points to `HomeController` and `action` method. Check the `HomeController` and you will find that the layout is `MasterLayout` and the `action` method calls the `render` of the `Home` template.

In `master_layout.html`, add the following code that builds the two columns layout for our home page:

```
<template name="MasterLayout">
  <div class="home-container">
    <header class="header">
      <h1>
        {{#linkTo route="home"}}
          BookMyTravel2
        {{/linkTo}}</h1>
      <ul class="nav nav-pills">
        <li>
          {{#linkTo route="createTravel"}}
            Create
          {{/linkTo}}
        </li>
      </ul>
    </header>
```

```
      <section class="home-container__section">
        <div class="home-container__section__left container-fluid">
            {{> yield region="search"}}
        </div>
        <div class="main">
            {{> yield}}
        </div>
      </section>
      <footer class="footer">Copyright @BookMyTravel2</footer>
   </div>
</template>
```

This will create the two column layout. For maintainability purpose, instead of using the Home template from home.html, we will create the BusServiceList template using the generator and call it in the Home template. Run the following command in the root directory of the application:

**iron g:template BusServiceList**

The command will create the bus_service_list directory inside app/client/ templates. We will use the bus_service_list.html file to define our listing template. Add the following code inside the template tag:

```
<div class="container bus-list">
  <div class="row bus-list__row bus-list__header">
    <div class="bus-list__row__col col-md-3">Bus</div>
    <div class="bus-list__row__col col-md-1">Available seats</div>
    <div class="bus-list__row__col col-md-1">Start point</div>
    <div class="bus-list__row__col col-md-1">End point</div>
    <div class="bus-list__row__col col-md-2">Start time</div>
    <div class="bus-list__row__col col-md-2">Reaching time</div>
    <div class="bus-list__row__col col-md-1">Fare</div>
    <div class="bus-list__row__col last col-md-1">Book</div>
  </div>
  <div class="row bus-list__body">
  {{#if hasItem}}
    {{#each list}}
      <div class="bus-list__row">
    <div class="bus-list__row__col col-md-3">{{name}}
    <br />{{agency}}</div>
    <div class="bus-list__row__col col-md-
    1">{{available_seats}}/{{seats}}</div>
    <div class="bus-list__row__col col-md-1">{{source}}</div>
    <div class="bus-list__row__col col-md-1">{{destination}}</div>
    <div class="bus-list__row__col col-md-2">{{humanReadableDate
    startDateTime}}</div>
```

```
    <div class="bus-list__row__col col-md-2">{{humanReadableDate
    endDateTime}}</div>
    <div class="bus-list__row__col col-md-1">{{fare}}</div>
    <div class="bus-list__row__col last col-md-1">
    <a href="/book/{{_id}}">Book</a></div>
    </div>
    <div class="clear"></div>
    {{/each}}
  {{else}}
  <div class="row bus-list__row bus-list__row-empty">
    <div class="bus-list__row__col last col-md-12 text-center">
    No buses found</div>
  </div>
  {{/if}}
   </div>
   </div>
```

We have used some helper methods inside the template. Let's define them inside the bus_service_list.js file. Replace the helper's skeleton with the following code:

```
Template.BusServiceList.helpers({
  list: function() {
    return this.get();
  },
  hasItem: function() {
    return this.get().count();
  },
  humanReadableDate: function(date) {
    var m = moment(date);
    return m.format("MMM,DD YYYY HH:mm");
  }
});
```

One last thing to do is call the BusServiceList template in the Home template. Go to Home in home.html and replace the content with the following code:

```
<template name="Home">
  {{> BusServiceList}}
</template>
```

An empty list will be visible in the browser by this time. We have to wire the data. The busservice collection has to be published first. Let us use the generator itself to create the publication. Run the following command in the terminal:

**iron g:publish busservice**

This will add the publication code to `publish.js` in `app/server`. However, we need a slight modification here. Our publication should publish data, by default, in sorted order. So, let us change the return statement of the publication as follows:

```
return Busservice.find({}, {sort: {createdAt: -1}});
```

We have a proper place to subscribe the data. In `HomeController`, we have the `subscriptions` method where we can subscribe the data by name. Add the following subscription line of code to the `subscriptions` method:

```
this.subscribe("busservice", {});
```

Along with this, we will do the search as well. To give a small recap, because we are going to filter the same collection in the list template using the values from search template, we have used reactive variables. Whenever there is a search value, we filter the collection and update the reactive variable, which will update the list as per the search.

We will generate search-related templates and helpers using the generator. Run the following command:

**`iron g:template search`**

This will generate the search related files in the `search` directory under `app/client/templates`. In the `Search` template inside `search.html`, replace the existing content with the following code:

```
<div class="col-xs-12 col-sm-12 col-md-12 text-center top-space
  well well-sm">Search</div>
<div class="col-xs-12 col-sm-12 col-md-12 well well-sm">
    <div class="form" id="search-form">
        {{#autoForm collection="Busservice"
        id="SearchBusServiceForm"}}
        {{> afQuickField name='source'}}
      {{> afQuickField name='destination'}}
      {{> afQuickField name='startDateTime' type="date"}}
      {{> afQuickField name='fare'}}
      {{/autoForm}}
    </div>
</div>
```

The template is ready. To render this search template to the `search` region in `MasterLayout`, add the following code to the `action` method in `HomeController`:

```
this.render('Search', {to: 'search'});
```

This will render the `Search` template to the search region. Visit the browser and you will see the search form.

We have to introduce the reactive variable. To use reactive variables, we have to install the `reactive-var` package. Run the following command to add the package:

**iron add reactive-var**

Add the following initialization code to the beginning of the `action` method:

```
this.ReactiveBusServices = new ReactiveVar([]);
```

The controller supports the `data` method such as the `subscriptions` method. In the `data` method, we can prepare and format data that will be passed to the template to render. Add the following data preparation code to the `HomeController`:

```
data: function() {
    this.ReactiveBusServices.set(Busservice.find({}));
    return this.ReactiveBusServices;
},
```

From the code, you can figure out that we set the `Busservice` collection to the reactive variable we had created in the preceding snippet and then return the reactive variable. Visit the browser and you will find the list of services you had created earlier.

It is time to put the search in place. In `search.js`, inside the `app/client/templates/search` directory, replace the events skeleton with the following code:

```
Template.Search.events({
  "keyup input": _.throttle(function(e) {
    var source = $("[name='source']").val().trim(),
    destination = $("[name='destination']").val().trim(),
    date = $("[name='startDateTime']").val().trim(),
    fare = $("[name='fare']").val().trim(),
    search = {};
    if(source) search.source = {$regex: new RegExp(source),
    $options: "i"};
    if(destination) search.destination = {$regex: new
    RegExp(destination), $options: "i"};
    if(date) {
      var userDate = new Date(date);
      search.startDateTime = {
        $gte: userDate,
              $lte: new Date(moment(userDate).add(1,
              "day").unix()*1000)
          }
```

```
      }
    if(fare) search.fare = {$lte: parseInt(fare, 10)};
    if(Template.instance()) {
    Template.instance().data.set( Busservice.find(search, {sort:
    {createdAt: -1}})));
    }
  }, 200),
  "submit": function(e) {
    e.preventDefault();
  }
});
```

In the input field's `keyup` handler, we collect the form data, prepare them to be a proper search query, and filter the collection. Then, we set the filtered collection into the reactive variable. If you notice, `Template.instance().data` is the reactive variable we had returned from the `data` method of the `HomeController`. Perform a search and you will find that things are working as expected. Finally, listing and search is done.

# Reservation

The last part of the application is blocking and reserving seats. We need the `reservations` and `blocked_seats` collections to store the information. Let's use the generator to generate the collection. Run the following commands to generate the collections:

**iron g:collection reservations**

**iron g:collection blockedSeats**

These commands will create two files, `reservations.js` and `blocked_seats.js` under `app/lib/collections`. Each file has its own instantiation to the collections, respectively. We will define the schema to each of these collections as we did for the `busservice` collection.

Add the following schema definition to `blocked_seats.js`:

```
BlockedSeats.attachSchema(
  new SimpleSchema({
    bus:{
      type: String,
      label: "Bus",
      max: 200
    },
    seat:{
      type: Number,
```

```
        label: "Blocked Seat"
      },
      createdAt: {
        type: Date,
        label: "Created At",
        autoValue: function() {
          if (this.isInsert) {
            return new Date;
          }
        }
      },
      updatedAt: {
        type: Date,
        label: "Updated At",
        autoValue: function() {
          if (this.isUpdate) {
            return new Date();
          }
        },
        denyInsert: true,
        optional: true
      },
      createdBy: {
        type: String,
        optional: true,
        autoValue: function() {
          return this.userId
        }
      }
    })
  );
```

Again, we can define validations if needed. I will leave that to you. Similarly, we will add the schema definition for `reservations` collection as well. Add the following code to `reservations.js`:

```
Reservations.attachSchema(
  new SimpleSchema({
    bus:{
      type: String,
      label: "Bus",
      max: 200
    },
    seats_booked:{
      type: [Object],
```

```
          label: "Seats Booked",
          minCount: 1,
          maxCount: 10
      },
      "seats_booked.$.seat": {
        type: Number,
        optional: false
      },
      createdAt: {
        type: Date,
        label: "Created At",
        autoValue: function() {
          if (this.isInsert) {
            return new Date;
          }
        }
      },
      updatedAt: {
        type: Date,
        label: "Updated At",
        autoValue: function() {
          if (this.isUpdate) {
            return new Date();
          }
        },
        denyInsert: true,
        optional: true
      },
      createdBy: {
        type: String,
        optional: true,
        autoValue: function() {
          return this.userId
        }
      }
    })
  );
```

Now, we have to define the route to reach the reservation part. We will use the generator to create the route. Run the following command to generate the route:

**iron g:route book**

The generator adds a route for us in the `routes.js` file and creates a `BookController`, templates, and helpers. The new route generated is not what we needed. Let's modify it to the way we want it to be. Change the route path from `book` to `book/:_id`. The route clearly says we need `_id`, which we will get from the listing. Also, we need the bus service information of the concerned bus, reservation information of the bus, and the blocked seats information of the bus. In the listing, we already have the link to the reservation page. Let's wire the proper data and create the templates to show the seating information and other required information.

We have to register the required publications first. As we did for the `busservice` collection, we will use generators to generate the publications for the `reservations` and `blocked_seats` collections. Run the following commands one after the other to generate them:

**iron g:publish reservations**

**iron g:publish blocked_seats**

To the generated publications, we need to do small modifications to pull the data of only the concerned bus service. Modify the code to look like the code as follows:

```
Meteor.publish('busservice', function (query) {
  query = query || {};
    return Busservice.find(query, {sort: {createdAt: -1}});
});
Meteor.publish('reservations', function (query) {
  return Reservations.find(query);
});
Meteor.publish('blocked_seats', function (query) {
  return BlockedSeats.find(query);
});
```

Note that we have also modified the `busservice` publication to accommodate the query parameter. Similarly, other publications will also get an object, based on which the data is published.

We will also use `CreateTravelLayout` for the reservation page. Add the `layoutTemplate` property to the controller and put the value as `CreateTravelLayout`.

The next step is subscribing to these data. In `BookController`, add the following piece of code to subscribe the data in the `subscriptions` method:

```
this.subscribe('busservice', {
        _id: this.params._id
    });
```

```
this.subscribe('reservations', {
    bus: this.params._id
});
this.subscribe('blocked_seats', {
    bus: this.params._id
});
```

Next, we have to pass the data to the template. Add the following code to the data method:

```
var templateData = {
    _id: this.params._id,
    bus: Busservice.findOne({
        _id: this.params._id
    }),
    reservations: Reservations.find({
        bus: this.params._id
    }).fetch(),
    blockedSeats: BlockedSeats.find({
        bus: this.params._id
    }).fetch()
};
return templateData;
```

This is the data that is available in the Book template.

Let us create the UI to show the seating arrangement. Replace the content of the Book template in book.html under app/client/templates/book, with the following piece of code:

```
<div class="container busView">
  <div class="row text-center busView__title">
    {{bus.name}}<br />{{bus.agency}}
  </div>
  <div class="row col-md-4 busView__seats">
    <div class="col-md-12 busView__left">
    {{#each seatArrangement}}
      <div class="col-md-12 row-fluid">
      {{#each this}}
      <div id="seat{{this.seat}}" class="busView__seat {{blocked}}
      {{reserved}}">
      {{this.seat}}
      </div>
      {{#if middleRow}}<div class="busView__divider col-md-offset-
      3"></div>{{/if}}
```

```
      {{/each}}
      </div>
    {{/each}}
    </div>
  </div>
  <div class="row text-center busView__book"><button id="book"
  class="btn btn-primary">Book My Seats</button></div>
</div>
```

We need some helpers and event handlers to handle the interactions. Replace the skeleton events and helper methods with the following set of code:

```
Template.Book.events({
  "click .busView__seat:not(.reserved):not(.blocked)": function
  (e) {
    e.target.classList.add("blocked");
    var seat = {
      bus: Template.currentData().bus._id,
      seat: parseInt(e.target.id.replace("seat", ""), 10)
    };
    Meteor.call("blockThisSeat", seat, function(err, result) {
      if(err) {
        e.target.classList.remove("blocked");
      } else {
        var blockedSeats = Session.get("blockedSeats") || [];
        blockedSeats.push(seat);
        Session.set("blockedSeats", blockedSeats);
      }
    });
  },
  "click #book": function() {
    var blockedSeats = Session.get("blockedSeats");
    if(blockedSeats && blockedSeats.length) {
      Meteor.call("bookMySeats", blockedSeats, function (error,
      result) {
        if(result) {
          Meteor.call("unblockTheseSeats", blockedSeats,
          function() {
            Session.set("blockedSeats", []);
          });
        } else {
          alert("Reservation failed");
          console.log(error);
        }
```

```
      });
    } else {
      alert("No seat selected");
    }
  }
});

Template.Book.helpers({
  seatArrangement: function() {
    var arrangement = [],
      totalSeats = (this.bus || {}).seats || 0,
      blockedSeats = _.map(this.blockedSeats || [], function(item)
      {return item.seat}),
      reservedSeats = _.flatten(_.map(this.reservations || [],
      function(item) {return _.map(item.seats_booked,
      function(seat){return seat.seat;});}))),
      tmpIndex = 0;
      Session.set("blockedSeats", this.blockedSeats);
    arrangement[tmpIndex] = [];
    for(var l = 1; l <= totalSeats; l++) {
      arrangement[tmpIndex].push({
        seat: l,
        blocked: blockedSeats.indexOf(l) >= 0 ? "blocked" : "",
        reserved: reservedSeats.indexOf(l) >= 0 ? "reserved" : "",
      });
      if(l % 4 === 0 && l != totalSeats) {
        tmpIndex++;
        arrangement[tmpIndex] = arrangement[tmpIndex] || [];
      }
    }
    return arrangement;
  },
  middleRow: function () {
    return (this.seat % 2) === 0;
  }
});
```

I don't have to explain this code as you will be familiar with it from the previous chapter. Now, you will be able to see the seating arrangement of the bus in the browser. The only leftover portion is server-side handling.

Add the server methods to `methods.js` in `app/server`. We need three server methods that are called from the template handlers. Add the following code to `Meteor.methods`:

```
bookMySeats: function(reservations) {
  var insertRes = reservations.map(function(res) {
    return {
      seat: res.seat
    }
  });
  return Reservations.insert({
    bus: reservations[0].bus,
    seats_booked: insertRes
  }, function (error, result) {
    if(result) {
      Busservice.update({_id: reservations[0].bus}, {
        $inc: {
          available_seats: -insertRes.length
        }
      }, function() {});
    }
  });
},
blockThisSeat: function(seat) {
  debugger;
  BlockedSeats.insert(seat, function(error, result) {
    console.log(error);
    if(error) {
      throw Meteor.Error("Block seat failed");
    } else {
      Meteor.setTimeout(function() {
        BlockedSeats.remove({_id: result});
      }, 600000);// 10 mins
    }
  });
},
unblockTheseSeats: function(seats) {
  seats.forEach(function (seat) {
    BlockedSeats.remove({_id: seat._id});
  });
}
```

Now, you can block and reserve seats from the application. This not much, but what we have learned so far is pretty interesting, isn't it?

We have recreated the same old application in a much more maintainable way like a pro-developer. We have used advanced scaffolding techniques to build the application and have increases maintainability and predictability. Also, we have learned to secure database operations with the help of `allow` and `deny` methods available in the collection instances.

Is that all for this chapter? No. To become a real pro-developer, two more things are essential. One is debugging and another one is testing. We are going to cover both of them in this chapter.

# Debugging

It is not necessary to emphasize on the importance of debugging for developers. It is an everyday thing that we do to understand code, identify bugs, and verify that code works fine. At novice level, we use `print` statements to debug applications. In JavaScript, we have used the `alert` statements, which are deadly if missed. Then, there were the `console.log` statements that silently log the parameters to the browser console. We can also use the `console.log` statements in Node.js code. It logs the output to the server console, which is the terminal. Non-novice developers have a different level of testing because all of these print statements are time-consuming. The iteration it takes to identify and fix a bug is higher. That is why we needed advanced tools that can stop the code execution at the interested line of code and give us the ability to inspect the variable at the given context. The best example would be Chrome developer tools.

We can put break points in the code and stop the execution at every break point. We can hop between lines, inspect values, or even change and run them and verify that the change works—we can do a lot in the developer tools. As long as things are in the browser, we have these tools. What about Node.js environment? What about MeteorJS server code? How do we debug them?

Fortunately, we have libraries to help us. Have you heard of Node Inspector? It is a Node.js module and you can install it from NPM using the following command:

```
npm install -g node-inspector
```

Node-inspector gives us the ability to inspect and debug the server side code from the browser itself. It works perfectly and notably with Chrome.

Once installed, go to the MeteorJS application that we had developed and start the application in the debug mode using the following command:

```
iron debug
```

If you are using a plain MeteorJS application, run the following command:

```
meteor debug
```

This will start the application in the debug mode and you will see the following message in the command line:

```
To debug the server process using a graphical debugging interface, visit
this URL in your web browser: http://localhost:8080/debug?port=5858.
```

Copy the URL and open it in a new tab in the Chrome browser. Here, you will see the chrome developer tool in a full viewport as follows:
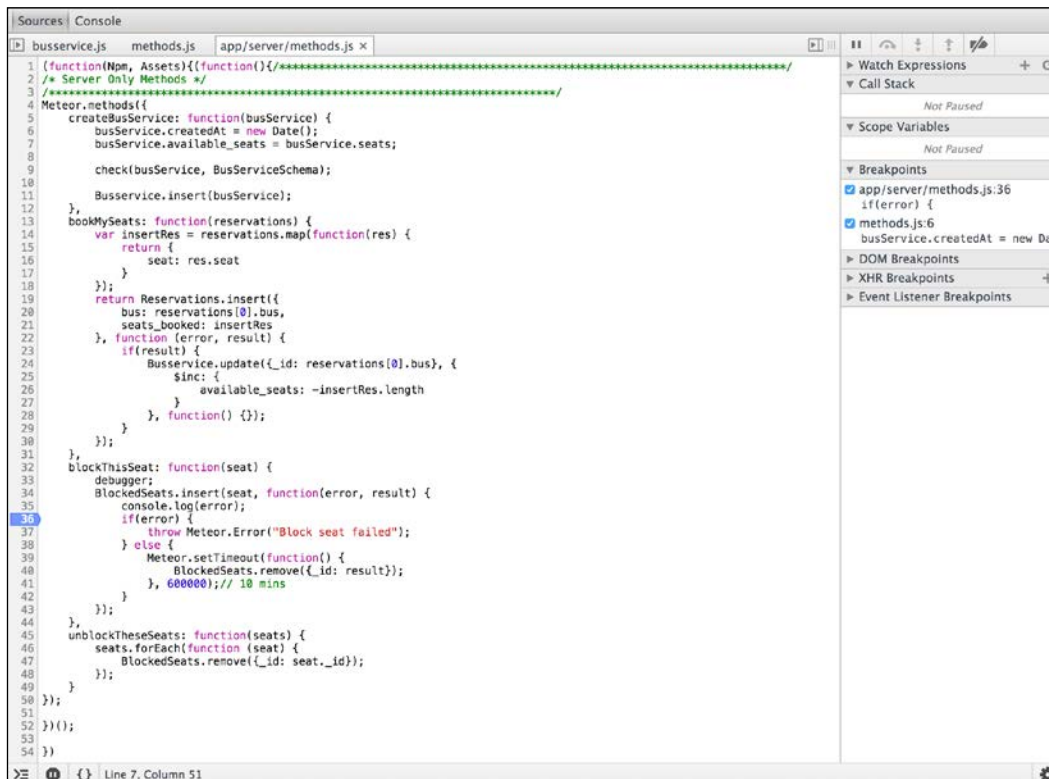
Refresh the application and then the node inspector tab twice or thrice; then, you will find all your server side files in the developer tools navigation area in the section on the left. Now, you can put a break point and do the appropriate action that will stop the execution at the break point. You can inspect your variables and know the values at the given context without any pain.

At times, we need the code to just stop at places where we can't put a breakpoint. In such cases, we can use the `debugger;` statement. The browser developer tool will stop at this point in the execution flow. From here on, you can continue the debugging, run through steps, or add new breakpoints. Give it a try in our application itself by adding a `debugger;` statement to the `blockThisSeat` server method and then try to block a seat from the interface we designed. This should stop at the `debugger` statement; from here, you can inspect parameters or add a breakpoint to the callback of the insert method and check what are the callback arguments and so on. Debugging this way will save a large amount of time than all the `print` statements.

# Meteor shell

This is another awesome feature of MeteorJS. Developers who are accustomed to Ruby on Rails will find this very similar. This is an interactive shell where one can access the application from the command line.

Run the application in non-debug mode. Open a new terminal and navigate to the app directory. Run the following command to get into the shell:

```
meteor shell
```

Now, you have the shell connected to the server, and we can call our server methods and do lot of other things right from the terminal. In our case, we can call the `blockThisSeat` method from the shell by calling `Meteor.call("blockThisSeat")`. This will throw an error as we need to pass proper parameters to the method. If we pass a proper argument, insertion will happen. Similarly, we can access all global variables from the shell. We have defined the `busservice` schema and the `Busservice` collection in global variables. We can access them from the shell. We can insert or update data using the `insert` or `update` methods on the `Busservice` instance. You can use it more or less like a browser console, except that we have access to the database. This again helps us to test methods, see data, run a manual operation, or debug things up to certain extent. Explore it and you will find it a lot more useful things.

# Testing MeteorJS application

Testing is as important as the development for a product. Testing is again a big cycle that ensures the quality of the product. As developers, we also play a major role in testing. In fact, we are the ones who start testing the product which then is passed to QA for extensive behavior, interaction, and all other possible testing. In this part of the chapter, we are going to learn how to write unit tests and integration tests.

MeteorJS doesn't provide any testing tools. However, it has announced Velocity (`http://velocity.meteor.com/`) as the official testing framework. You can read about Velocity on its website (`http://velocity.meteor.com/faq`).

# Velocity

Velocity is an ecosystem that allows us to use various existing testing tools to work together for our MeteorJS application. For example, Mocha, Jasmine, and Cucumber. They all will use the same reporter (the reporter is responsible for telling the user what tests have passed and failed). Visit the website at `http://velocity.meteor.com` and there is the list of supported testing tools. Choose one or more of your favorite tools and start writing test cases for the application.

All the theory doesn't help us do things. We need action, so let's write some real tests for our booking application using Velocity reporter and Jasmine.

# Testing BookMyTravel2

We need to install the Velocity reporter package and the Jasmine package for MeteorJS to get started. From the application root, using iron-cli, add the package as follows:

```
iron add velocity:html-reporter
```

If you are building apps without using iron-cli, run `meteor add velocity:html-reporter`. We have installed the reporter. Now, we have to install the testing framework. As I have mentioned earlier, we are going to use Jasmine.

Jasmine is a behavior-driven testing framework with simple syntax and concepts. Let us install the Jasmine package using the command `iron add sanjo:jasmine`. We are ready to write tests. Once installation is completed, visit the application in the browser. You will find a bulls eye kind of a dot in the right corner. Click on it and you will see the reporter that we have installed. Cool? Isn't it? It is reactive, which means, as you write code, it runs the tests and gives you the results.

Many suggest we write tests first and then build the application. However, many are against it. It is a highly debatable topic and, thus, I leave it you. However, if you are writing tests first and then building the application, this setup will be really helpful. In our case, we have built the application already, so we will write tests for the application, which again is not bad.

The Velocity interface gives us options to get started by providing some sample tests, which can help us know where and how to get started. You are free to click on those buttons and see the sample tests created. However, let me walk you through, which will give you a detailed idea. We have to follow some conventions to place our tests so that Velocity can find the tests and run them. The convention is very simple. We have to follow a particular directory structure to place our tests. We have to create the `tests` directory under the `app` directory followed by the sub-directory with the name of the testing framework. We are going to use `jasmine`. Inside the `jasmine` directory we have to create the `client` and `server` directories, where we can put client and server tests separately. Again, under each of these directories, we have to create two more directories with the name `unit` and `integration`. I hope you have guessed it. The `unit` directory is for unit tests and `integration` is for integration tests. So, for a MeteorJS application, we have got to write four sets of tests (unit and integration tests for the client and server).

We will write tests only for a small portion of the app, to the extent of knowing how to write tests for the MeteorJS application. You can write the rest of the tests by yourself as homework.

Firstly, we will write a server side unit test followed by server-side integration test. Inside the `app/tests/jasmine/server/unit` directory, create a `createBusService.js` file. Add the following test to the file we have created:

```
describe("CreateBusService", function() {
  'use strict';

  beforeEach(function() {
    Meteor.call("removeAllBusservice");
  });

  describe("Creating a service", function() {
    it("A new service must be created using server method
    createBusService", function() {
      spyOn(Busservice, "insert").and.returnValue(1);
      var service = {
        name: "2Test",
        agency: "Testing agency2",
        seats: 30,
```

```
        source: "Delhi",
        destination: "Bombay",
        startDateTime: moment().add(1, "days").toDate(),
        endDateTime: moment().add(2, "days").toDate(),
        fare: 1200
      };
      Meteor.methodMap.createBusService(service);
      expect(Busservice.insert).toHaveBeenCalledWith(service);
    });
  });
});
```

According to Jasmine's terminology, we describe a test suite and each test suite will have test specs. We can nest test suites, which is what we have done in the preceding code.

We have created a test suite with the name `CreateBusService`. Next is the child test suite, which is a nested or child test suite called `Creating a service`. The spec is the test that will be about the server method `createBusService`. That is why we named the specification as "`A new service must be created using the server method createBusService`". Inside the spec, we have asked Jasmine to spy on the `Busservice` collection's `insert` method. Then, we have defined a bus service object and passed it with the method call. One thing to notice is the `methodMap` property in the calling line:

```
Meteor.methodMap.createBusService
```

It is a map that is used for calling server methods. We have used it to call the `createBusService` method by passing the service object. The last part of the specification is the expect to be. Here, we do a comparison of the results with appropriate things. In our case, we just need to verify that `Busservice.insert` is called with the service object. So, we have asked Jasmine to spy on the `Busservice.insert` method and then compared the insertion parameter with the one we have passed. The code that does this operation is as follows:

```
expect(Busservice.insert).toHaveBeenCalledWith(service);
```

It is very important to go through the Jasmine documentation to learn more about writing efficient test cases. There are plenty of methods and their negations that support our test cases. Now, let us visit the browser and see the test case in action. Visit the application, click on the dot, and then you will see a green tick mark in the last section (*Jasmine-server-unit*). At the top, it says 1 test passed. It means the test is executed and has passed.

Similarly, we will create an integration test for the server. In `app/tests/jasmine/server/integration` directory, create a file `integration.js` (name doesn't matter) and add the following content:

```
describe("Serverside integration testing", function() {
  'use strict';
  describe("on collection definition", function() {
    it("Busservices collection instance must be defined",
    function() {
      expect(Busservice).not.toBeUndefined();
    });
  });
});
```

The test is self-explanatory. All it does is, tests whether the `Busservice` collection is defined or not. Visit the application in the browser and you will find a green tick mark in the jasmine-server-integration section of the reporter. Also, at the top, it will show two tests passed. You can add integration tests as much as possible here by creating new specs or new suites.

Now, let's create some tests for the client. Create a file `create_travel_service.js` under `app/tests/jasmine/client/unit`. We are going to write unit tests for the bus service creation. Inside the file, create a test suite as follows:

```
describe("CreateBusService", function() {
});
```

We're going to need a `beforeEach` function to clear the `busservice` collection before running every spec so that the list will not be cluttered. Add the following piece of code inside the test suite we created:

```
beforeEach(function() {
  Meteor.call("removeAllBusservice");
});
```

The `beforeEach` function calls a server method. If you are familiar with Jasmine, you know that `beforeEach` is called before executing every spec we have written. We are clearing the `busservice` collection by calling a server method. We need to create that server method before we run the test. Go to `methods.js` in the `app/server` directory and add the following piece of code:

```
removeAllBusservice: function() {Busservice.remove({});}
```

Calling this method will clear all documents in the `busservice` collection. You might ask, why do we write a method to remove all the documents from the collection, instead we can just call the remove method on the collection instance. In a server, it will work. MeteorJS doesn't allow you to remove all the documents of a collection from the browser console or the client code. This is the reason we have to define a method in the server, which does the job for us. You are free to play around these methods. For learning purposes, we are going to add three test specs in the client unit tests. Add the following to the test suite we have created after the `beforeEach` function:

```
it("should fail insertion if source and destination are same",
  function() {
  Meteor.call("createBusService", {
    name: "2Test",
    agency: "Testing agency2",
    seats: 30,
    source: "Bombay",
    destination: "Bombay",
    startDateTime: moment().add(1, "days").toDate(),
    endDateTime: moment().add(2, "days").toDate(),
    fare: 1200
  });
  expect(Busservice.find().count()).toBe(0);
});

it("should fail insertion if seats are more than 50", function() {
  Meteor.call("createBusService", {
    name: "2Test",
    agency: "Testing agency2",
    seats: 90,
    source: "Bombay",
    destination: "Chennai",
    startDateTime: moment().add(1, "days").toDate(),
    endDateTime: moment().add(2, "days").toDate(),
    fare: 1200
  });
  expect(Busservice.find().count()).toBe(0);
});

it("should create a record in the database", function() {
  Meteor.call("createBusService", {
    name: "Test",
    agency: "Testing agency",
```

```
    seats: 30,
    source: "Bombay",
    destination: "Kolkata",
    startDateTime: moment().add(1, "days").toDate(),
    endDateTime: moment().add(2, "days").toDate(),
    fare: 1200
  });
  expect(Busservice.find().count()).toBe(0);
});
```

There are two negative test cases and one positive test case. The test cases are pretty much straightforward. The first specification is to test that, if the source point and the destination point are the same, insertion should not happen. If you notice the parameter passed to the `createBusService` method call, you will find that both the source and destination are the same place. If insertion fails, the collection count will be `0`, and this is what we expect and assert in the first test.

The same is the case for second specification where seats count is invalid. We have a restriction from the schema that the max value for seats is 50. We have specified `90` in the second test, because of which insertion will fail. Again, the collection count will be `0` and the expected is also the same. The last one is a positive case where it should insert a document; therefore, the count of the collection will be `1`. The `expect` method verifies this at the end of the specification.

Finally, the client integration test remains. Create `integration.js` and add the following tests to it:

```
describe("Client template testing", function() {
  beforeEach(function() {
    Meteor.call("removeAllBusservice");
  });

  describe("Templates", function() {
    it("Home must be having bus-list division", function() {
      var div = document.createElement("DIV");
          var comp = UI.render(Template.Home, div);
          expect($(div).find(".bus-list").length).toBe(1);
    });
  });

  it("should show one row in the listing page", function(done) {
    Meteor.call("createBusService", {
      name: "Test",
```

```
        agency: "Testing agency",
        seats: 30,
        source: "Bombay",
        destination: "Kolkata",
        startDateTime: moment().add(1, "days").toDate(),
        endDateTime: moment().add(2, "days").toDate(),
        fare: 1200
    }, function() {
        expect($(".bus-list__body .bus-list__row").length).toBe(1);
        done();
    });
});

it("List template should have rows equal to docs in busservice
collection", function(done) {
    Meteor.autorun(function() {
        if(DDP._allSubscriptionsReady()) {
            done();
            var div = document.createElement("DIV");
                var comp = UI.render(Template.Home, div);
                expect($(div).find($(".bus-list__body .bus-
                list__row")).length).toBe(Busservice.find().count());
        }
    });
});
});
```

In the client integration testing, we are going to use the DOM to verify the states. The first test spec checks whether a division with the class `.bus-list` is present and also whether it is the only one present in the template. The second spec tries to insert a document and thereby confirms that it reflects in the listing page. We have used jQuery for selectors. You can also use plain JavaScript. Finally, the last test case verifies that the number of items in the collection is equal to the number of rows in the list.

Finally, if it is all good, in the browser, you will find that all the tests are passed without any failure, as shown in the following image:
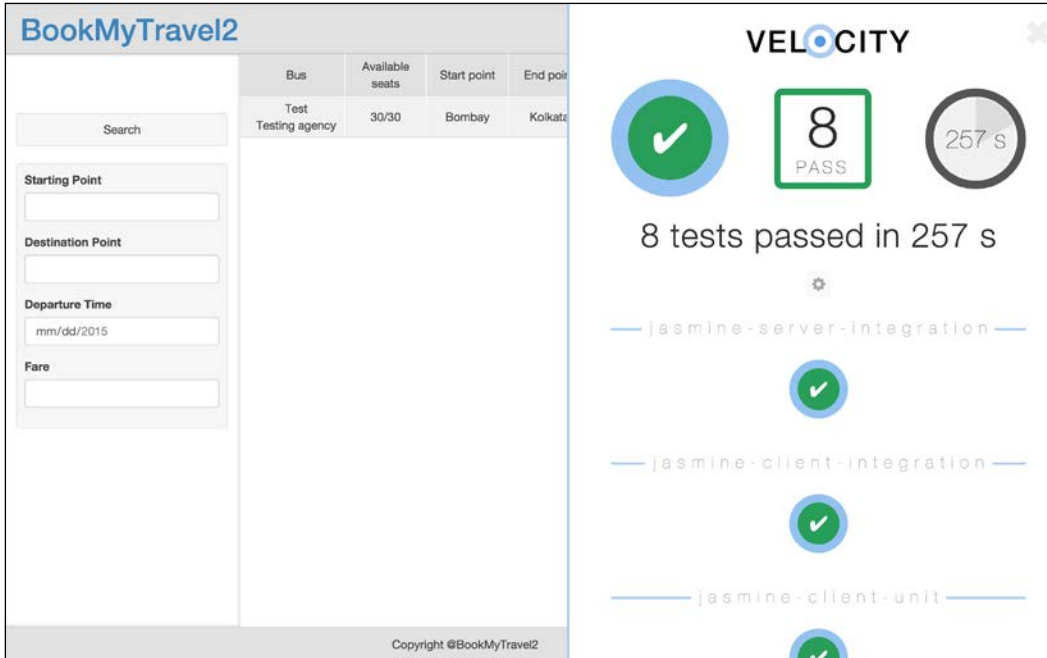


The specs and suites provided are the basic ones, which can be easily understood. However, we can write complex specifications, and Jasmine facilitates it pretty well. That is all about Jasmine.

Apart from Jasmine, we can use Mocha, Cucumber, Casper, and Robot frameworks to write and execute test cases. Other than Jasmine and Mocha, the rest of the frameworks are not yet mature enough to work with MeteorJS. There is not enough documentation or examples to explore. We would have to do trial and error, and a lot of exploration, from examples to the core packages. The scope of these frameworks is beyond this book, so I will stop this topic right here.

We have come a long way, but in the right way. In this chapter, we have gathered plenty of information and also learned about many resources. Coming to the end of the chapter, let us summarize what we have learned.

# Summary

This chapter covered most of the important aspects of developing a MeteorJS application. So far, what we have learned is quite enough to develop a maintainable application. The application has many other aspects to be improved in terms of features. You were familiarized with the way of developing quality and testable applications. Let us summarize what we have learned in this chapter.

We learned scaffolding a MeteorJS application, with which we can generate all the necessary components in one shot, or as an individual component. With scaffolding, we also learned that each collection object has the `allow` and `deny` methods, with which we can perform extensive role-based authorization checks on data-related operations. With `iron-router`, we learned to create a maintainable pattern to define a controller, layout, subscription, and data for templates. We learned to define schema, provide validation, display the validation errors in forms, and create custom validations.

Debugging is an important developer skill, and using tools to debug code will reduce our time. We learned all the means of debugging a MeteorJS application. We learned how to use Node-inspector, Meteor shell, and even the good old message printing. Finally, we learned to write tests for the MeteorJS application using velocity and Jasmine.

If you look at the summary, you can figure out that this chapter covered all the required aspects of a complete development. We have learned a lot in this chapter. In the next chapter, we will learn how to create packages. Cheers!

# 3
# Developing Reusable Packages

Packages are one of the key reasons for MeteorJS' growth. The framework is built in a way to create and use local packages, or use the packages created by other developers from the community. We can also use NPM packages in the application. This adds to the rate at which applications are developed using MeteorJS. As I have mentioned in the previous chapter, atmosphere is where you can find all the packages developed by the community. So, if we have created a package that can benefit others, we can put it in atmosphere.

A package is nothing but a functionality. `Accounts-ui` is a package that has the login and signup-related functionality. MomentJS is a famous date manipulation library. It was converted to suit MeteorJS' convention and was released as a package. We have some packages that are existing libraries, which are modified to be a MeteorJS package, and some packages that are written specifically for MeteorJS applications. As I had mentioned before, MeteorJS itself comprises a large number of packages. We can see the list of packages used by MeteorJS in GitHub (`https://github.com/meteor/meteor/tree/devel/packages`). Blaze is a package for UI rendering and manipulation. Some of the important functionalities of the framework are extended from packages. One such package is `tracker` which is responsible for the reactivity of the framework. Similarly, `mongo` is a package responsible for MongoDB operations. When most of the framework's functionality is provided by packages, why don't we explore it?

In this chapter, we are going to cover the following topics by developing an interesting package:

- Creating a package
- Using the package
- Testing the package
- Distributing the package

# Introduction to packages

In the previous chapters, we have used packages and I hope you remember them. We have used packages such as `iron-router`, `momentjs`, `reactive-var`, and so on. We have used the `meteor add <package name>` command to add packages. When we run this command, MeteorJS searches for the interested package in `https://packages.meteor.com` and adds it to the application. If we want to use a specific version of a package because of compatibility reasons, then we have to use `meteor add <package name><@version>`. There are MeteorJS applications that are built just with custom-made packages. There are developers who believe that it is good to build their modules and functionalities as custom packages that is evolving as a pattern to build the MeteorJS applications. Take a look at the Stack Overflow answer (`http://stackoverflow.com/a/26733023/407342`), which will give an explanation about it. This clearly gives us the picture of how important packages are in MeteorJS.

# An installed package

Have you ever wondered where MeteorJS keeps all the packages? You can find them in the `.meteor` directory. MeteorJS converts the files of a package into one or more files based on the environment of execution during its build process. Create a MeteorJS application and visit the `.meteor` directory and its subdirectories. Inside the `.meteor/local/build/programs/server/packages` directory, we will find the compiled version of the core packages used by MeteorJS at the server. Similarly, inside the `.meteor/local/build/programs/web.browser/packages` directory, we will find the compiled version of packages used by MeteorJS at the client end.

These are the directories where the actual code that is executed in the appropriate environment lives. We can even find that all the templates are converted into the `.js` files. Let's refocus on our mission. When we add a package, MeteorJS creates files from the package to a version that is suitable to run in the environments and adds it to the appropriate build directories.

# Creating a package

In this section, we will learn how to create a custom package and use it in our application.

> The packages creation example that we are going to see is with respect to MeteorJS version 1.1.x. Note that there could be changes in the API or the structure with the next release.

We have to know two things about creating packages. If we are going to create a package for public use or common purpose (such as using it across applications), then it is always good to keep the package out of the application. However, if we are going to create a package to wrap the modules of the application, which means it is going to be application-specific, then we will keep it inside the packages directory inside the application.

Why would someone write their application modules as packages? There are a lot of advantages. When we write a package, we gain control over the order in which the files are run. Also, we expose what needs to be exposed. We can write tests for that particular module inside the package itself; thus, it is all self-contained, which makes it easy to maintain.

We are going to build a small application in which we will create a custom package to implement a module of the application. The application is all about showcasing some products, and each product can be added to the cart. Let's start developing the application.

Create a MeteorJS application with the name `ProductKart` using the `meteor create ProductKart` command. Remove the `autopublish` and `insecure` packages. To keep things simple, we are not going to add server and client separations. We will write both the server and the client code inside `ProductsKart.js`. The templates will be inside `ProductsKart.html`. However, we will create two directories for the application; one is the packages directory where our package is going to live and the other is public, from where the images of the product will be served. Let us do some ground work to get the application ready.

We will insert some products during application startup from the code to keep things simple. Then, we will add templates to display them in the application. Replace the content of `ProductsKart.js` with the following code:

```
//products collection
ProductsCollection = new Mongo.Collection("products");

//Client side code
```

```
if (Meteor.isClient) {
  Meteor.subscribe("products");//subscribing to the products
  collection
  Template.ProductList.helpers({  //template helper for
  ProductList template in ProductsKart.html
    products: function() {
      return ProductsCollection.find({});
    }
  });
}

//Server side code
if (Meteor.isServer) {
  Meteor.startup(function() {
//On startup add the insert the following objects to mongo
  collection if the collection is empty
    var products = [{
      name: "Puppy",
      image: "puppy.jpg",
      prize: 12000
    }, {
      name: "Shoe",
      image: "shoe.jpg",
      prize: 1900
    }, {
      name: "Cup Cake",
      image: "cupcake.jpg",
      prize: 200
    }];
    if (ProductsCollection.find({}).count() <= 0) {
      products.forEach(function(item) {
        item.createdAt = new Date();
        item.updatedAt = null;
        ProductsCollection.insert(item);
      });
    }
  });
  //Publishing the products collection
  Meteor.publish("products", function() {
    return ProductsCollection.find({});
  });
}
```

Developer comments in the preceding code explain clearly what is done in which part of the code. All we have done is, we have inserted a few hard-coded products into the products collection when the application starts so that we will have something to add to the cart. We have created the products collection instance at the first line of the code and published it at the end. In the `Meteor.isClient` block, we have subscribed the collection and defined a helper for listing the product.

To display the products, we have to define the template. Replace the content of `ProductsKart.html` with the following code:

```html
<head>
  <title>ProductsKart</title>
</head>
<body>
  <header>
    <div class="logo">ProductsKart</div>
  </header>
  <div class="container">
    {{> ProductList}}
  </div>
  <footer>
    copyright@NoBody
  </footer>
</body>

<template name="ProductList">
  <ul class="list">
    {{#each products}}
    {{> Product}}
    {{/each}}
  </ul>
</template>

<template name="Product">
  <li class="list-item">
    <img src="/{{image}}" />
    <div class="name">{{name}}</div>
    <div class="prize">{{prize}} INR</div>
  </li>
</template>
```

Here, we have just defined two templates; one for listing and the other is for each product that is a child template for the listing template. Get some product images and add them to the public directory and rename them according to the database records that we inserted at the application startup. If we look at the product template, we can figure out that we are going to display the image of the product, the name, and then the prize.

> The `public` directory is where we have to put static content such as images, which is a convention in MeteorJS.

If you start the application, you will find in the browser that the products are displayed but not aligned. Let us add some styles to make them look pretty. Add the following styles to the `css` file:

```
body {margin: 0;padding: 0;}
header, footer {
  padding: 20px 30px;background-color: #aaaccc;}
footer {text-align: center;}
.logo {font-size: 20px;}
.bucketContainer {
  position: absolute;
  right: 30px;top: 20px;}
.container {
  height: calc(100vh - 121px);overflow: auto;}
.list {
  width: 100%;
  padding: 2%;margin: 0;
  box-sizing: border-box;}
.list-item {
  width: 29%;
  list-style: none;
    padding-bottom: 100%;
    display: inline-block;
     padding: 0 2%;
    text-align: center;}
.list-item img {width: 100%;}
.list-item > * {margin-bottom: 5px;}
button {
  padding: 6px 10px;
  background-color: #00aacc;
  color: #fff;
  border: 0;
  cursor: pointer;
  border-radius: 5px;}
```

Visit the browser and you will find that things are in place and they look pretty.

The required application is ready and we also have products to add to the cart. We are going to develop the cart as a package for our application, which will export the necessary templates and variables to display the cart and manipulate the cart data.

The packages directory is where we are going to create the package. MeteorJS provides us with the following command, with which we can create packages:

```
meteor create –package <developer>:<package name>
```

From this command, we can figure out that we need a developer account in `https://www.meteor.com`. Is this mandatory? No. However, if you want to distribute your package to the community, then it is mandatory. If you are going to use the package just for local development, then it is not required. In our case, we are going to keep it local.

Before we start, we need a name for our package. We will keep it as `bucket`. Let's create the `bucket` package by running the `meteor create –package bucket` command. Now, visit the packages directory and you will find a directory with the name `bucket`. Inside the `bucket` directory, there are some files in which we will write our package code. Before starting to write the package, we need to learn some conventions that we have to follow while writing a package.

Get acquainted with the `package.js` file, as it is one of the most important files of the package. It is more like the bootstrap file of an application. You can find out about this file in the MeteorJS official documentation (`http://docs.meteor.com/#/full/packagejs`). You will find three sections in the `packages.js` file.

# Package.describe

In this section, we have to describe the package. This section is very important when we distribute the package for public use. The properties inside the describe part are self-explanatory. If we are distributing, then the name of the package must be in the `<developer-name>:<package-name>` format and is mandatory. All these properties will help you to know about the package. In our case, we can ignore them. The following code is an example of the `describe` API:

```
Package.describe({
  name: 'bucket',
  version: '0.0.1',
    // Brief, one-line summary of the package.
  summary: '',
  // URL to the Git repository containing the source code for
  this package.
```

```
    git: '',
    // By default, Meteor will default to using README.md for
    documentation.
    // To avoid submitting documentation, set this field to null.
    documentation: 'README.md'
});
```

# Package.onUse

The `onUse` section is the most important section of the file. Here, we define what files to load in which environment and in what order. MeteorJS doesn't automatically load all the files of the package as it does for the application. We have to list even from the very basic package to the custom files that we need for our package using the `onUse api` method. Inside the function that we pass to the `Package.onUse` method, we get an `api` variable as an argument that is an object with certain methods, which helps us to register our files and other packages that our package is going to use:

```
Package.onUse(function(api) {
  api.versionsFrom('1.1.0.2');
  api.addFiles('bucket.js');
});
```

The two `api` methods are as follows:

- `api.versionsFrom`: This takes a string that represents the version of the Meteor core packages we want to use.
- `api.addFiles`: This takes an individual file or an array of files followed by the environment. If we want to use a file both in the server and the client, pass them both in the array.

There are a few more methods that can be useful, as follows:

- `api.use`: This helps us to define what other packages our package is going to use
- `api.imply`: This allows us to access the packages used by this package
- `api.export`: This helps us to expose variables to the appropriate environments

# Package.onTest

The `onTest` method helps us write tests for the package. We will use jasmine to write tests for our package and velocity to run the tests. We will take a detailed view of this section at the end of the chapter:

```
Package.onTest(function(api) {
  api.use('tinytest');
  api.use('bucket');
  api.addFiles('bucket-tests.js');
});
```

# The bucket package

We are free to do anything inside the package, in the same way we do in the application. Let us first describe the features of the package and then we'll start structuring the package code. The bucket is nothing but a cart. We can add products to the cart and remove them from the cart. We should provide a way to reach the cart, which can be a division anywhere in the application. We can display the number of items in a cart inside this division. When the user clicks on the division, it will show the entire cart with all the items added to the cart.

In technical terms, we need a collection to store the cart items, templates to show the division that we spoke about, another template to list the items in the cart, and some helpers for these templates. Let us start creating the required components.

# Collection

Create the collections directory inside our package directory (`packages/bucket`). Add a file, `bucket.js`, inside the collections directory. We will define a collection for our cart by adding the following line of code to this file:

```
BucketCollection = new Mongo.Collection("user_s_bucket");
```

# Templates

Next, we need templates. Create a directory called `templates` inside our package directory. Our package will provide three templates for the application. One is the division that we spoke about earlier, next is the button to add or remove items to and from a cart, and one is to display the whole cart. Create a file, `bucket.html`, inside the templates directory and add the following template to it:

```
<template name="Bucket">
  <a href="#" class="bucket {{classes}}">{{name}} <sup><span
  id="no-of-items">{{item}}</span> {{text}}</sup></a>
</template>
```

This template is the division that, when clicked, will display the entire cart. This template will also display the number of items in the cart. There are some configurable properties that can be passed from the application while using this template. If the other developer wants to add some classes, or change the display text or the items count follow-up text, he/she can do it right in the usage rather than changing them with in the package.

Time to see the work of our partially done template! As I have mentioned earlier, MeteorJS doesn't load all these files that we have created. So we have to do this explicitly in `package.js`. Change the content of the `Package.onUse` method to look like the following code:

```
api.versionsFrom('1.1.0.2');
api.use(["templating", "mongo"]);
api.addFiles('bucket.js');
api.addFiles('./collections/bucket.js', ['client', 'server']);

api.addFiles('./templates/bucket.html', ["client"]);
```

Watch the code snippet carefully. We have to explicitly mention what other packages our package will use. In the second line of the code snippet, we have mentioned `templating` and `mongo`. These are very basic packages, but still we have to mention them. In the fourth and fifth line, we add the files that we had created just before.

After this, one more step is pending; that is, to add the package to the application. So far, we have created the package, but haven't added it to the application. Let us do it now. Run the `meteor add bucket` command in the application root. This will add the package to the application. Now, if we check the packages list in the application using the `meteor list` command, the command line output will display our custom package with a plus sign next to the version that indicates that it is a local package:

```
bucket            0.0.1+
meteor-platform   1.2.2  Include a standard set of Meteor packages in
  your app


+ These packages are built locally from source.
```

We have successfully created a package and added it to the application. We have to use the package in our application now.

# Using the package

We can use the template from our package in the application. Append the following template call to the header section of `ProductsKart.html`:

```
<div class="bucketContainer">{{> Bucket classes="icon-bucket"
  name="Cart" text="items" bucketContainer="#bucketPlace"}}</div>
```

After adding this, the cart division will be displayed in the top-right corner of the application in the browser. Hope you got a fair idea of how the packages work. Let us complete the rest of the package features.

We have created the collection in our package, but haven't published or subscribed it. We will publish the collection in the server by adding the following code to the `bucket.js` file inside our package directory:

```
if(Meteor.isServer) {
  Meteor.publish("bucket", function() {
    return BucketCollection.find({});
  });
}
```

We will subscribe for the collection in the same file by adding the following code:

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

To display the total items in the bucket collection, we need a helper. Create the `helpers` directory in parallel with the `templates` directory and add the `bucketHelper.js` file. Add the following helper to this file:

```
Template.Bucket.helpers({
  item: function() {
    return BucketCollection.find({}).count();
  }
});
```

In the `package.js` file, we have to add this file so that it is available in the application. Add the following line to the `Package.onUse` method along with the existing content:

```
api.addFiles('./helpers/bucketHelper.js', ["client"]);
```

If you visit the browser now, you will find the bucket division at the top-right corner, displaying **0 items** that was previously just **items**.

The next step is to provide the "Add to Cart" buttons from the package. We will create the buttons by adding the following template to bucket.html:

```
<template name="AddOrRemoveButton">
  {{#if remove}}
  <button class="removeFromBucketBtn
  {{classes}}">{{removeButtonText}}</button>
  {{else}}
  <button class="addToBucketBtn {{classes}}"
  disabled="{{disabled}}">{{addButtonText}}</button>
  {{/if}}
</template>
```

We will use the same template for "Add to Cart" and also for the "Remove From Cart" functionality. We are going to use this button template in the application. Modify the Product template in ProductKart.html to accommodate the button template as shown in the following code:

```
<template name="Product">
  <li class="list-item">
    <img src="/{{image}}" />
    <div class="name">{{name}}</div>
    <div class="prize">{{prize}} INR</div>
    {{> AddOrRemoveButton addButtonText="Add to Cart"
    removeButtonText="Remove From Cart"}}
  </li>
</template>
```

This change will add a button to all the products in the list. We have to attach the database operation to add items to the user_s_bucket collection when the button is pressed. To be more precise, when the "Add to Cart" button is clicked, we have to collect the product data and add it to the user_s_bucket collection that we had created in the package. We will bind a click event to the button and call a server method to insert into the user_s_bucket collection. In bucketHelper.js, add the following event handler:

```
Template.AddOrRemoveButton.events({
  "click .addToBucketBtn": function(e) {
    Meteor.call("addToBucket", Template.parentData());
  },
  "click .removeFromBucketBtn": function(e) {
    Meteor.call("removeFromBucket", Template.parentData());
  }
});
```

When the "Add to Cart" button is clicked, it will call the `addToBucket` server method with the parent data. The parent data will have the complete product object. Similarly, when "Remove From Cart" (we will add this in the whole cart section) is clicked, it will call another server method, `removeFromBucket`, with the parent data.

Now, we have to define both these server methods. Add the following piece of code to `bucket.js` after the `BucketCollection` publication:

```
Meteor.methods({
    "addToBucket": function(product) {
      BucketCollection.insert({
        item: product,
        product_id: product._id,
        createdAt: new Date(),
        updatedAt: null
      });
    },
    "removeFromBucket": function(product) {
      BucketCollection.remove({
        product_id: product._id
      });
    }
  });
```

These server methods simply add to the collection and remove from the collection. This is pretty easy to understand at first glance. Click on the **Add to Cart** button and you will see the **Cart** division at the top-right corner, which reflects the change.

We don't want to add a duplicate product to the cart. So, we will disable the button once added to the cart. To disable it, add the following code to `bucketHelper.js`:

```
Template.AddOrRemoveButton.helpers({
  remove: function() {
    return
    Template.instance().hasParentTemplate
    ("BucketItemslistWrapper");
  },

  disabled: function() {
    var product = Template.parentData();
    return !!(BucketCollection.findOne({product_id:
    product._id}));
  }
});
```

This helper will check whether the product is present in the `user_s_bucket` collection and, if present, it will disable the button. Also, the other helper, `remove`, will help us decide when to show the add button and the remove button. The remove button will be shown only when we display the selected products in the whole cart. We will come back to that later. So far, we have implemented the cart division and the button to add items to the cart. The remaining part is displaying the whole cart.

Displaying the entire cart while clicking on the **Cart** division at the top-right is a little complicated in terms of code. Firstly, we will define the template for the whole cart section. Add the following template to `bucket.html`:

```html
<template name="BucketItemsList">
  <div class="bucketItemsListContainer {{displayStatus}}"></div>
</template>
```

Our cart section doesn't need to implement a new rendering logic or template to show the items in the cart. It can reuse the same template used by the application to list the product. However, the package must be informed about which template to reuse. So, while using the `BucketItemsList` template in the application, we will have to pass the list template's name as data to the `BucketItemsList` template as given in the following template:

```
{{> BucketItemsList parentTemplate="ProductList"}}
```

Here, note that we are passing the template, in order to reuse it as part of the data, to the `BucketItemsList` template. In the `onRendered` callback, we will manipulate the passed template name and reuse it by feeding the bucket collection data. Add the following relative code that does the magic for us:

```
Template.BucketItemsList.onRendered(function() {
  this.autorun(function() {
      if(this.subscriptionsReady()) {
    var bucketItems = BucketCollection.find({}).fetch(),
        products = {
      products: bucketItems.map(function(bucketItem) {
          return bucketItem.item;
        })
      },
        selector = Template.instance()
          .$(".bucketItemsListContainer"),
        visibility = selector.is(":visible");
        Template["BucketItemslistWrapper"] = new Template(
      "BucketItemslistWrapper",
      Template[this.data.parentTemplate].renderFunction
```

```
    );
      selector.children().remove();
      if(bucketItems.length) {
    Blaze.renderWithData(
      Template["BucketItemslistWrapper"], products,
      selector[0]);
    } else {
        selector[0].innerHTML = "<div class='no-items'>No items
        in the cart</div>";
    }
  }
  }.bind(this));
});

Template.BucketItemsList.helpers({
  displayStatus: function() {
    var status = Session.get("BucketItemsListStatus");
    if(status === undefined) {
      return "hidden";
    }
    return Session.get("BucketItemsListStatus");
  }
});
```

In the `onRendered` callback, we are registering an `autorun` method to render the cart template whenever the subscription is ready. Once the subscription is ready, we collect all the products from the `user_s_bucket` collection. Then, the template that we had passed along with the data to the `BucketItemsList` template is cloned using the following code snippet:

```
Template["BucketItemslistWrapper"] = new Template(
  "BucketItemslistWrapper",
  Template[this.data.parentTemplate].renderFunction
);
```

The selected products are fed into the cloned template using the following code snippet:

```
Blaze.renderWithData( Template["BucketItemslistWrapper"],
  products, selector[0]);
```

This will render the whole cart. However, inside the cart, we need to show the **Remove from Cart** button instead of the **Add to Cart** button. To achieve this, we have to define the condition that decides when to use the **Remove from Cart** button. In `bucketHelper.js`, we have already added the `Template.AddOrRemoveButton.helpers` helper and the `remove` method. The `remove` method verifies that the `AddOrRemoveButton` template is used inside the `BucketItemslistWrapper` template using the `hasParentTemplate` method that has to be defined as given in the following code snippet:

```
Blaze.TemplateInstance.prototype.hasParentTemplate = function
    (name) {
    var view = Blaze.currentView;
    while (view) {
        if (view.name === name) {
            return true;
        }
        view = view.parentView;
    }
    return false;
};
```

Add the preceding method definition to the top of the `bucketHelper.js` file. From the `remove` method, it will check and resolve when to use the **Add to Cart** button and the **Remove from Cart** button. Also, we have already attached the click event for the **Remove from Cart** button that will call the server method to remove items from the `user_s_bucket` collection.

The only thing pending is to add the click interaction to the **Cart** division at the top-right. Add the following code snippet to the `bucketHelper.js` file:

```
Template.Bucket.events({
  "click .bucket": function(e) {
    e.preventDefault();
    if(Session.get("BucketItemsListStatus") === "hidden") {
      Session.set("BucketItemsListStatus", "");
    } else {
      Session.set("BucketItemsListStatus", "hidden");
    }
    $(".bucketItemsListContainer").slideToggle();
  }
});
```

In the preceding code snippet, we toggle the cart display when clicking on the **Cart** division in the top-right. To achieve this, we have used the session to maintain the state. Click on the **Cart** division and you will find the slide animation of the whole cart.

We need some package-specific styles. We will add it by creating the `styles.css` file inside the `stylesheet` directory inside our package directory. Add the following styles:

```
.hidden {display: none;}
.bucket {  text-decoration: none;color: #222;}
.no-items {
  text-align: center;
  padding: 20px;
  font-size: 20px;}
button[disabled="true"] {background-color: #aaa;}
```

Add the stylesheet to the `package.js` file's `Package.onUse` method so that it will reflect in the browser.

Finally, just for learning, we will expose a variable from a package to the application. Prepend the following code to `bucket.js` at our package directory:

```
Bucket = {
  getTotalPrize: function(prizeField) {
    prizeField = prizeField || "prize";
    var total = 0, itemsInCart = BucketCollection.find().fetch();
    itemsInCart.map(function(cartItem) {
      total += cartItem.item[prizeField];
      return total;
    });
    return total;
  },
  getTotalNumberOfItemInBucket: function() {
    return BucketCollection.find().count();
  }
};
```

We are trying to expose some APIs for the application to use from our package. However, to make it available, we have to add the following line to the `Package.onUse` method of the `package.js` file:

```
api.export('Bucket', ["client", "server"]);
```

Now, `Bucket.getTotalPrize()` is accessible in the application. Without the preceding line, the `Bucket` variable will not be available in the application.

The implementation is complete and the package is fully functional. Play around with it and don't forget to give a proud smile for what we have achieved. I hope what we have learned so far is fun and useful.

# Testing the package

Packages in MeteorJS facilitate writing tests by default. We can use the `jasmine` package to test our package. Add it to the application using the `meteor add sanjo:jasmine` command. Also, we need a test runner. We will use velocity's `html-reporter`, that we have used in the previous chapter. Add the `velocity` package using the `meteor add velocity:html-reporter` command.

Add the `tests` directory to our package directory and remove `bucket-tests.js`. We will write the client and server tests separately, as we did for the application in the previous chapter. Create the `server` and `client` directories inside the `tests` directory and add `server.js` and `client.js` inside the `server` and `client` directories, respectively.

Let us start adding some tests. Firstly, we will test our server-side code. Inside `server.js`, add the following code, which does some basic tests:

```
describe("Server tests", function() {
  it("BucketCollection should be defined", function() {
    expect(BucketCollection).toBeDefined();
  });

  it("Can call getTotalPrize in Bucket variable at server",
  function() {
    expect(Bucket.getTotalPrize()).toBe(0);
  });
});
```

Similarly, we will add a test case for a client by adding the following code to `client.js`:

```
describe("Template test", function() {
  it("Bucket template should be defined in client", function() {
    expect(Template.Bucket).not.toBeUndefined();
  });
});
```

We have to add these files in `package.js` for the test environment to recognize the tests. Modify `Package.onTest` to make it look like the following code:

```
Package.onTest(function(api) {
    api.use('sanjo:jasmine@0.15.4');
    api.use('bucket');

    api.addFiles('./tests/server/server.js', ['server']);
    api.addFiles('./tests/client/client.js', ['client']);
});
```

At the first line of the method, we informed the test environment to use `jasmine`. Mind the version number, as it is very important to mention it. Then, we have added the package we had created followed by the test specification files.

Also, we need to add an export to `Package.onUse` as follows:

```
api.export('BucketCollection', ["client", "server"]);
```

We do this to expose `BucketCollection` to the testing environment.

To run the test and see the report, we should stop the application. Run the following command in the application root, which will run a server for our tests:

```
VELOCITY_TEST_PACKAGES=1 meteor test-packages --driver-package
   velocity:html-reporter bucket
```

When we run this command, a server runs at `localhost:3000`, which will show the test reports. Visit `localhost:3000` in the browser and see that our tests have passed. We have written basic test cases that you can extend to any depths as needed by the package.

So far, we have added tests to our package and have also ran it. What next? Distributing the package for general public use.

# Distributing a package

To distribute a package, it is mandatory to follow certain rules. Being developers, we know how well a public library or framework needs to be maintained. It has to be annotated, documented, organized, updated, and above all supported. Only then, an open source library or framework will gain momentum. The same is the case here when we think about distributing a package.

First of all, we have to create the package out of our application and link it to our application using soft links. We don't have to create the package inside the `packages` directory, instead we keep it in some other convenient location and start writing the package. Then, we can soft link (`ln -s <path to package>`) the package to the `packages` directory inside the application. If we run the `meteor add <package>` command, things will work the way we have done so far in the previous sections.

Once you have created and completed the package, write tests as we did earlier. Write tests to significant portions at least to make sure the package is in good condition. Create a `git` repository to maintain the package. Add the package to the repository and follow proper Git conventions to maintain the package. Create tags and development versions for appropriate use. Also, add extensive documentation and the `ReadMe.md` file explaining every aspect of the package.

Another important thing to do is to fill the `package.js` file's *Package.describe* section with real values, which we have ignored before. Follow the name convention by creating a developer account in `meteor.com` and use it to prefix your package name, such as `<dev-acc-name:package name>`. Add the Git URL and summary so that it will be easy for other developers to find the repository and file issues, if any. Versioning is again important; you follow it as you wish, but do not ignore it.

Finally, when you feel that your package is ready to be released for public use, just run `meteor publish -create` from the package root directory. This will add the package to `https://packages.meteor.com`. The `-create` flag is to indicate that we are publishing the package for the first time. On subsequent releases, you can use just `meteor publish`. After publishing the package, it will be available at `https://atmospherejs.com/`. However, before publishing, make sure you are doing the right thing and not polluting the ecosystem. Read the guidelines link (`https://atmospherejs.com/i/publishing`) to know when to publish a package.

This is all we need to know about package development. We have covered all that we had committed to learn in this chapter.

# Summary

Let's summarize what we have learned in this chapter. We have learned how to create packages for a MeteorJS application. We have learned to describe a package for both local and public use. Also, we have learned how to add files to a package and export variables. We have used `jasmine` to test our package with the help of `velocity`. Finally, we have learned to follow the conventions to publish or distribute a package to the MeteorJS ecosystem.

I hope you have enjoyed the chapter. Now, you can go ahead and create some useful packages and distribute it for the community to use. In the next chapter, we will learn to use some of the popular frontend frameworks with a MeteorJS application.

# 4
# Integrating Your Favorite Frameworks

The rise of the frontend frameworks such as Backbone.js, Angular.js, Ember.js, React. js, and so on, has changed the way we write JavaScript. In fact, after the rise of these frameworks, the popularity of JavaScript peeked high enough for people to trust and build full-fledged JavaScript-based single page applications. Each framework has its own advantages and disadvantages, and such an analysis is out of the scope of this book. However, what really matters is how far they have driven application development using JavaScript.

In today's application development, every frontend developer has his choice of framework. A few developers adhere to Backbone.js, while many have migrated to Angular.js and Ember.js. React.js is getting way popular, and is can even be used as a view layer for other frameworks such as Backbone.js. This shift has forced backend systems to behave like data providers or data sources, and has exposed data via mostly REST API and rarely other API formats. These APIs become the only point of contact between the data and the application. Thus, the entire control of the application is concealed within the frontend framework.

MeterorJS is not an exception. Though MeterorJS has every needed component in-built, many developers don't want to spend time in learning another templating language or view component; for example, in MeteorJS case, it is Blaze. Many also prefer to use their favorite frontend framework with MeterorJS. So we are going to learn how to use some of these frontend frameworks with MeterorJS. Based on the popularity (GitHub stars), Angular.js and React.js are chosen for this chapter.

In this chapter, we are going to cover the following topics:

- Using Angular.js with MeteorJS
- Using React.js with MeteorJS
- A simple data visualization with d3.js and MeteorJS
- A general idea of using any frontend framework with MeteorJS and Angular.js

Every framework has its own idea. Angular.js has the idea of smart HTML with an inbuilt two-way binding powered by the digest cycle. It is a very impressive idea for a framework, but has its own disadvantages in terms of performance. It is the reason Angular2 is developed in a way, to overcome the performance issue, and also beta is out for all Angular.js enthusiasts to try it out. As far as this chapter is concerned, we will use Angular1 because Angular2 is not yet ready for production.

As you may have guessed, we will learn to use Angular.js with MeterorJS by developing a small application. Before starting, we have to know a few things about using Angular.js with MeteorJS. Angular.js, by default, supports a two-way binding. With MeteorJS, we will get a three-way binding, which is also called the ultimate reactivity. Much more powerful. Like a typical single page application, we will use MeterorJS to provide data with reactivity and Angular.js to govern and render the data in the browser.

Can we use Angular.js as it is with MeteorJS? Yes, of course. However, what is the fun in that while we can do things even better! If we still want to use Angular.js separately, then all we need to do is build a REST-based backend system using MeterorJS and use Angular.js to build the application user interfaces.

In this chapter, we will use Angular.js with MeteorJS to get the most out of the combination. To use Angular.js with MeteorJS, there is an excellent package called `angular-meteor` (`https://github.com/Urigo/angular-meteor`) that bridges the gap between both the frameworks by providing certain conventions and APIs. This package has been added as an official package to support Angular.js. So we believe this package will be maintained by the Meteor Development Group.

The package provides a way to wire the data subscription with the `$scope` object. Whenever there is a change in data, this wiring updates the `$scope` object and you know what happens then. This kind of reactivity is called a three-way binding in Angular.js terms. To support routes, we will use the `angular-ui-router` package.

This is all we need to know. Let's start developing an application using Angular.js with MeteorJS. What's the idea of the application? Everyone wants delicious food at the right time. To serve delicious food to people, we will develop an application that will showcase the food menu items. To keep it simple, we will develop the food item create/edit page and a display page with the login support. If you wish, you can extend the application to support every desired feature of your taste.

There is nothing much to code in the MeterorJS server environment specific to Angular.js. We will follow the same approach we used in the previous chapters to the develop applications. Firstly, define database schema, then add or remove related packages, publish the required data, and add methods to perform database operations. The remaining is the Angular.js part where we will configure an Angular.js app and a module, as well as some routes to support create, edit, and display and some helper directives.

The name of the product is very important to reach as many users as possible. We will name it ngFoodMenu. Create a MeterorJS application with the name `ngFoodMenu`, and remove all the default files. Create the `client`, `server`, and `lib` directories and add the `index.html` file at the root of the application.

# The server-side setup – FoodMenu

Firstly, we will complete server-side development quickly and precisely. Then, we will concentrate on the Angular.js part.

# Collection

Inside the `lib` directory, add the `collections.js` file. Here, we will put our collection that will be shared across the application. It is better to use the `Collection2` package to define a schema and validation. However, to demonstrate, we will assume the schema and develop things. Add the following piece of code to `collections.js`:

```
/** name, chef_name, contents, time, image, quantity,
  prize, is_published, is_deleted, created_at, updated_at
*/
FoodMenu = new Mongo.Collection("foodMenu");
```

The fields in the collection are given in the preceding code comment. We have defined the collection and created an instance as well.

# Publish

We will publish the collection. However, before publishing, we need to remove the `autopublish` and `insecure` packages. After removing them, add `publish.js` to the `server` directory.

Add the following publication registration code to the `publish.js` file:

```
Meteor.publish("foodMenu", function (args) {
  args = args || {};
  return FoodMenu.find(_.extend({is_published: true, is_deleted:
  false}, args));
});
```

# Access rules

We will add access rules to the collection using the `allow` and `deny` methods of the collection instance. Create the `accessRules.js` file in the `server` directory and add the following code to the file:

```
FoodMenu.allow({
  insert: function() {
    if (Meteor.user().profile.role === "admin") {
      return true;
    }
  },
  update: function(userId, doc) {
    if (Meteor.user().profile.role === "admin") {
      doc.updated_at = new Date();
      return true;
    }
  },
  remove: function() {
    return true;
  }
});

FoodMenu.deny({
  insert: function() {
    return false;
  },
  update: function() {
```

```
      return false;
    },
    remove: function() {
      return false;
    }
  });
```

Looking at the code, you can figure out that we need to add the login and the user role to create or update documents in the `FoodMenu` collection.

# Methods

We need to expose methods from the server to perform database operations. Create the `methods.js` file in the `server` directory and add the following methods to the file:

```
Meteor.methods({
  "SaveItem": function(item) {
    item.created_at = new Date();
    FoodMenu.insert(item, function(err, res) {
      console.log(err, res);
    });
  },
  "UpdateItem": function(item) {
    item.updated_at = new Date();
    var id = item._id; delete item._id;
    FoodMenu.update(id, {$set: item}, function(err, res) {
      console.log(err, res);
    });
  }
});
```

For now, this is all we need to do in the server. The code must be very familiar and is very much self-explanatory. It does just the basic operations.

# The client-side setup – FoodMenu

The real crux of the chapter lies in the client. Let's add the `angular-meteor` package to the application to start with the client. Add the package using the `meteor add urigo:angular` command. We are ready to write Angular.js code.

# Client packages

In the client, we will have two sections. The first section is the header, where we will display the logo, navigation link, and accounts section. We are going to use the `accounts-password`, `twbs:bootstrap`, and `ian:accounts-ui-bootstrap-3` packages for the login and general styles. The other section is the container, where we will display the create/edit form or the listing based on the route. As we mentioned earlier, we also need `angular-ui-router` for routes. Add all these packages using the `meteor add` command. To add `angular-ui-router`, run the following command:

```
meteor add angularui:angular-ui-router
```

Create the `directories`, `lib`, `stylesheets`, `controllers`, `directives`, and `templates` in the `client` directory.

# Application styles

Inside the `stylesheets` directory, create `styles.css` and add the following styles to the file:

```
body {margin: 0;padding: 0;}
header {background-color: #1e4560;color: #fff;padding: 20px 10px;
  display: flex;}
.logo {font-size: 30px;flex: 1 1 auto;}
.logo  a {color: #fff;text-decoration: none;}
.logo a:hover {text-decoration: none;}
.login-signup {align-self: center;}
.login-signup ul {margin-bottom: 0;}
.login-signup ul li{list-style: none;display: inline-block;}
.login-signup ul li > a {color: #fff;}
.login-signup ul li .dropdown-menu {right: 0;left: auto;}
.food-list {list-style: none;text-align: center;padding: 0;}
.food-list__item {margin-top: 4em;box-shadow: 1px 1px 1px 1px
  #ccc;}
@media (min-width: 992px) {
  .food-list__item:nth-child(4n), .food-list__item:first-child {
     margin-left: 4%;}
}
.food-list__item > div {padding: 10px;border-bottom: 1px solid
  #ccc;}
.food-list__item > div:last-child {border-bottom: none;}
.food-list__item__name {font-size: 16px;font-weight: bold;}
.food-list__item__prize {padding: 10px;}
/** Commons **/
.ptr {cursor: pointer;}
```

# The Angular.js application

Once the ground work is done, get ready to write some real code. Add the `app.js` file inside the `client/lib` directory. In this file, we will define the Angular.js app and the required routes. Add the following app declaration code to the file and carefully go through them:

```
angular.module('FoodMenu', ['angular-meteor', 'ui.router']);
```

The name of the Angular.js app module is `FoodMenu`. We have defined the dependencies as well. The application object is created. Now, we can define our application HTML. Add the following code to `index.html` at the application root:

```
<head>
    <title>ngFoodMenu</title>
    <base href="/">
</head>

<body ng-App="FoodMenu">
    <header siteheader></header>
    <div ui-view></div>
</body>
```

# The header section

As we discussed earlier, you can see the two sections of the application inside the `ng-App` scope. From HTML, we can deduce that the header is a directive with the name `siteheader`. Let's create this directive. Create the `header.js` file inside the `client/directives` directory and add the following directive creation code:

```
angular.module('FoodMenu').directive('siteheader', function () {
    return {
        templateUrl: 'client/directives/header.ng.html'
    };
});
```

The preceding code snippet creates the directive and registers it to the module. From the code, it is very clear that the template must be in `header.ng.html` under `client/directives`. Note the `.ng.html` extension carefully. This is required to make sure MeterorJS doesn't process this HTML file like the other non-angular HTML files.

Add the following template code to the `header.ng.html` file:

```
<div class="logo"><a href="/">ngFoodMenu</a></div>
<div class="login-signup">
  <ul>
    <li ng-if="currentUser && currentUser.profile.role ===
    'admin'" class="btn"><a href="/add-item">Add Items</a></li>
    <meteor-include src="_loginButtons"></meteor-include>
  </ul>
</div>
```

This is a small piece of code, but with some new things. Where did the template get the `currentUser` object from? Angular.js's root scope (`$rootScope`) provides the `currentUser` object. How did it get attached to `$rootScope`? The `angular-meteor` package does the magic. The next important thing to note is the `meteor-include` tag. We can use MeterorJS templates inside the Angular.js templates. To do this, the `angular-meteor` package has created a special directive called `meteor-include`. We are including the `loginButtons` template from the `accounts-ui-bootstrap3` package. The _ prefix is nothing special but, because the `loginButtons` template is exposed with the name `_loginButtons` from the package, we use it that way. For more references on `meteor-include`, check out the `angular-meteor` (https://github.com/Urigo/angular-meteor#meteor-include) GitHub repository. Now, visit the application in the browser. Remember to start the application.

The next task is to create a user and provide him/her with the admin privileges. We don't want to spend time configuring the signup to accommodate a user role. Instead, we will do it in the database directly. Create a user from the signup form in the header, go to database of the application, and run the following command:

```
db.users.update({_id: "<user id>"}, {$set: {profile: {role: "admin"}}});
```

Find your user ID from the users' collection and use it in the update query. This will give the "admin" privileges to the user. If all goes well, you will see the "Add Items" link in the header. This is because we have configured it in the `siteheader` directive to appear for the admin user. Did you notice the reactivity of the Angular.js template here? We added the role of the user in the database that is reflected to `$rootScope`, which in turn reflects in a directive template. Do you see the power of the full-stack reactivity or three-way data binding? We just need to configure the reactive data source to the reactive entities (scopes) of the Angular.js framework and things will work out.

# The application container section

We are done with the header. The next part of the application to develop a form page to create a food menu item, and this page should be reached via a route.

## Angular.js routes

We will add all our routes to the `client/lib/app.js` file. Append the following route-related code to `app.js`:

```
angular.module('FoodMenu').config([
'$urlRouterProvider', '$stateProvider', '$locationProvider',
    function($urlRouterProvider, $stateProvider,
    $locationProvider) {
        $locationProvider.html5Mode(true);
        $stateProvider
            .state("addItem", {
                url: '/add-item',
                templateUrl: 'client/templates/additem.ng.html',
                controller: 'CreateItemController',
                resolve: {
                    "currentUser": ["$meteor", function($meteor) {
                        return $meteor.requireUser();
                    }]
                }
            });
        $urlRouterProvider.otherwise('/');
    }
]);
```

In the preceding code snippet, we are configuring the required route for the creation page. If you are familiar with Angular.js, you will know the dependency injection happening in the first line of the code. The important part to concentrate on is the `.state` method of `$stateProvider`. The first parameter is the name with which we can refer this route. The second parameter has the actual route, template file path, controller name, and resolve property, which helps to wait for resources such as subscriptions and the user object. The `$meteor.requireUser();` method call will throw the "AUTH_REQUIRED" error if a user object is not present. We can capture this error in the `$stateChangeError` event at `$rootScope` in order to redirect the user to the page they access to, or to do some other relevant operations. The `angular-meteor` package exposes the `Meteor` object as `$meteor` and injects when needed in a pure Angular.js style.

# The CreateItem controller

For the route to do something useful, we have to define the controller and the template as specified in the route. Add the createItemController.js file inside the client/controllers directory. Create a controller by adding the following piece of code:

```
angular.module("FoodMenu").controller("CreateItemController", [
  "$scope", "$meteor", "$state",
  function($scope, $meteor, $state) {
    var setVal = function() {
      $scope.item = {
        name: "",
        chef_name: "",
        contents: "",
        time: "Lunch",
        image: "",
        quantity: 1,
        prize: 20,
        is_published: true,
        is_deleted: false,
        created_at: "",
        updated_at: null
      };
    }
    setVal();
    $scope.selectImage = function(element) {
      $scope.item.image = element.files[0];
    }
    $scope.save = function() {
      Images.insert($scope.item.image,
        function(err, res) {
        if (!err) {
          $scope.item.image = res._id;
          $meteor.call("SaveItem", $scope.item);
          setVal();
          $state.go("list");
        } else {
          console.log(err);
        }
      });
    }
  }
]);
```

In the controller, we define the required properties and methods in the scope. We have defined an `item` property in `$scope` that has all the form fields as an Angular.js model.

# Uploading images

Our application is going to support image upload. How can we show a list of food items without pictures? Users will be disappointed, won't they? To add the image upload feature in the application, we are going to use the `CollectionFS` package. We will use a different mongo collection to store the uploaded images' details. It is highly recommend to visit the `Meteor-CollectionFS` package repository (`https://github.com/CollectionFS/Meteor-CollectionFS`) to see what other features the package offers. Add the package using the `meteor add cfs:standard-packages` command. This is the standard package.

## The CollectionFS collection

For the image collection to work, we have to follow certain steps so that the images that we try to upload will be uploaded to the appropriate storage and the record is maintained properly in the database. We need to create a mongo collection for images, which means we need a `Mongo.Collection` instance to get the handle of the mongo collection similar to the `FoodMenu` instance. However, instead of using `Mongo.Collection`, the `Meteor-CollectionFS` package has its own way of creating the instance. Append the following piece of code to `lib/collections.js`:

```
Images = new FS.Collection("images", {
  stores: [new FS.Store.GridFS("images")],
  filter: {
    allow: {
      //allow only images in this FS.Collection
      contentTypes: ['image/*']
    }
  }
});
```

Now, we know how to create the `FS` collection instance. The `Meteor-CollectionFS` package supports different types of storage. Based on the storage we choose, we have to install the additional storage package. In our case, it is the `gridfs` storage and so we will add the `gridfs` package by executing the following command:

**meteor add cfs:gridfs**

From the preceding code, it is clear that the `FS.Collection` constructor takes the collection name as the first parameter, while the second parameter is an object that specifies which storage we would like to use and the optional filters, if any. In our case, we have specified the filter to allow only the files of the type image.

## Access rules

To this collection instance, we need to specify the access rules; it is mandatory to specify them, at least the `allow` rules. We can add the `deny` method if needed. Append the following access rule code to `server/accessRules.js`:

```
//add your access rules here
Images.allow({
  insert: function() {
    return true;
  },
  update: function(userId, doc) {
    return true;
  },
  remove: function() {
    return true;
  },
  download: function() {
    return true;
  }
});
```

## Publish images

We have to publish this collection to the client, which can be done by appending the following piece of code to `server/publish.js`:

```
Meteor.publish("images", function (argument) {
    argument = argument || {};
  return Images.find(argument);
});
```

Now, if you visit the `$scope.images` declaration that is commented in `createItemController.js`, you will find the `$meteor` object has the `collectionFS` API with which we can refer to the `Images` collection. Keep in mind that the `Images` collection is a `CollectionFS` collection instance and not a normal `Mongo.Collection` instance. If it was a normal `Mongo.Collection` instance, we would use `$meteor.collection(<collection instance>)`. We will see it when we subscribe to the `FoodMenu` collection later. Just for the sake of knowing, it is put over there and commented.
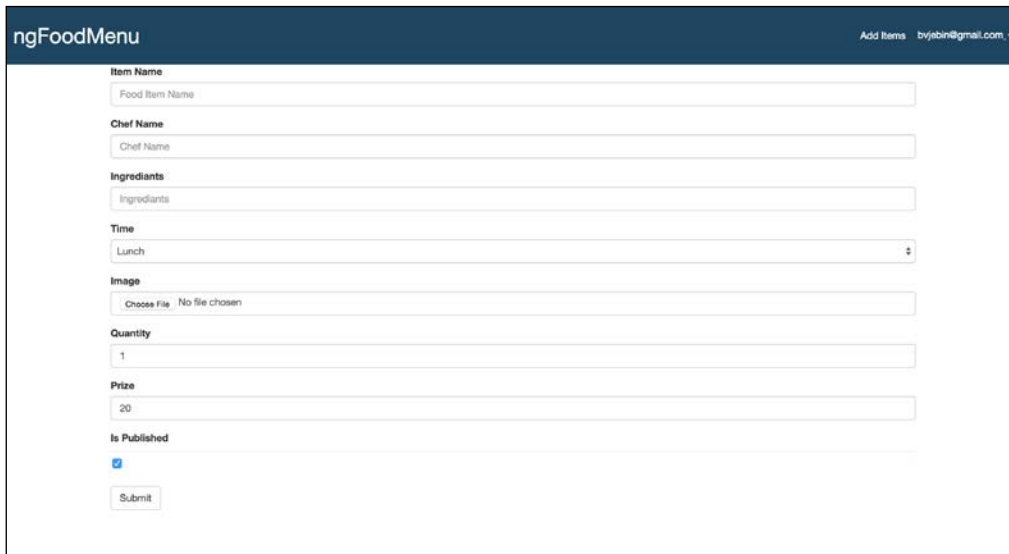
# The AddItem Angular.js template

The only remaining part is the template. Create a template file addItem.ng.html inside the client/templates directory and add the following template code to it:

```html
<div class="container">
  <form role="form">
      <div class="form-group">
          <label for="name">Item Name</label>
          <input required type="text" class="form-control"
          id="name" placeholder="Food Item Name"
          ng-model="item.name" />
      </div>
      <div class="form-group">
          <label for="chef_name">Chef Name</label>
          <input type="text" class="form-control" id="chef_name"
          placeholder="Chef Name" ng-model="item.chef_name"/>
      </div>
      <div class="form-group">
          <label for="contents">Ingredients</label>
          <input required type="text" class="form-control"
          id="contents" placeholder="Ingrediants"
          ng-model="item.contents" />
      </div>
      <div class="form-group">
          <label for="time">Time</label>
          <select class="form-control" id="time"
          ng-model="item.time">
            <option value="BreakFast">BreakFast</option>
            <option value="Lunch">Lunch</option>
            <option value="Dinner">Dinner</option>
          </select>
      </div>
      <div class="form-group">
          <label for="image">Image</label>
          <input type="file" class="form-control" id="image"
          placeholder="Image"
          onchange="angular.element(this).scope().
          selectImage(this)" ng-model="item.image" />
      </div>
      <div ng-if="edit===true" class="form-group row">
        <img ng-src="{{imageSrc}}" class="col-md-4" />
      </div>
      <div class="form-group">
          <label for="quantity">Quantity</label>
          <input required type="number" class="form-control"
          id="quantity" placeholder="Quantity"
          ng-model="item.quantity" />
      </div>
```

```
            <div class="form-group">
                <label for="prize">Prize</label>
                <input required type="number" min="20" max="2000"
                class="form-control" id="prize" placeholder="Prize"
                ng-model="item.prize" />
            </div>
            <div class="form-group">
                <label for="is_published">Is Published</label>
                <input type="checkbox" class="form-control"
                id="is_published" placeholder="Is Published"
                ng-model="item.is_published" />
            </div>
            <div ng-if="edit===true" class="form-group">
                <label for="is_deleted">Is Deleted</label>
                <input type="checkbox" class="form-control"
                id="is_deleted" placeholder="Is Deleted"
                ng-model="item.is_deleted" />
            </div>
            <button type="submit" class="btn btn-default"
            ng-click="save()">Submit</button>
        </form>
    </div>
```

Visit `http://localhost:3000/add-item` in the browser and you will find the form for creating food items, as shown in the following screenshot:

# Demystifying the logic

In the preceding template, the model is bound to the appropriate fields using the ng-model directive. The template is self-explanatory. When an image file is selected, the selectImage method is called, which sets the file object to the image property in $scope.item. On submission, the save method is called, which will save the image first. When successful, the $scope.item object is passed to the server by calling the SaveItem server method for persisting the food item data along, with the inserted image record ID.

Did you notice that we insert image into the Images collection right in the client, but we call the server method to insert food item data document into the FoodMenu collection? The reason is that we cannot transfer the file object to a server via a server method. It will transfer only the meta data but not the file object. So, we have no option but to insert from the client. The angular-meteor package provides the save method on collection instances to be called $scope.foodMenu.save(). However, it is always good to persist the data from the server.

Where are the inserted files, by the way? If you are aware of GridFS, you know the answer. For those of you who don't know what it is, GridFS does nothing but store the file as chunks in MongoDB. In MeteorJS Mongo console, look for collections that start with the cfs prefix using db.cfs followed by tab. You will see the list of collections created to store the file chunks and other metadata. If you are against storing files in a database, go for fileSystem or Amazon S3. The CollectionFS document provides us a few choices.

# Listing food items

So far, we have developed an interface to create food items. Next, we can develop a listing page that is going to be the landing page for the application in which we will display all the food items created. We need to specify a route, a controller, and a template to complete the listing of the items.

## Route

Add the following piece of state to client/lib/app.js right after the addItem route, as follows:

```
$stateProvider
  .state("addItem", {
    url: '/add-item',
    templateUrl: 'client/templates/additem.ng.html',
```

```
                    controller: 'CreateItemController',
                    resolve: {
                        "currentUser": ["$meteor", function($meteor) {
                            return $meteor.requireUser();
                        }]
                    }
                })
                .state('list', {
                    url: '/',
                    templateUrl: 'client/templates/foodList.ng.html',
                    controller: 'FoodListController',
            resolve: {
                        "subscribe": ["$meteor", function($meteor) {
                            return $meteor.subscribe('images');
                        }]
                    }
                })
```

## Controller

Create the `foodListController.js` controller file inside the `client/controllers` directory and add the following controller code:

```
angular.module("FoodMenu").controller("FoodListController", [
    "$scope", "$meteor",
    function($scope, $meteor) {
      $scope.foodMenu = $meteor.collection(FoodMenu)
            .subscribe('foodMenu');
      $scope.imageSrc = function(id) {
        return Images.findOne({_id: id}).url();
      }
      $scope.deleteItem = function(item) {
        delete item.$$hashKey;
        item.is_deleted = true;
        item.is_published = false;
        $meteor.call("UpdateItem", item);
      }
    }
  ]
);
```

In the preceding controller, we subscribe to the `FoodMenu` collection and then attach the `deleteItem` method to delete the food item from the list. We also get the image URL for the image using the `.url()` method in the `imageSrc` method provided by the `CollectionFS` document.

## Template

The next task is to add the template for listing the food items. Create `foodList.ng.html` inside `client/templates` and add the following template code to the file:

```
<div ng-controller="FoodListController" class="container">
  <ul class="food-list row">
    <li class="col-xs-12 col-md-3 col-md-offset-1 food-list__item"
    ng-repeat="item in foodMenu">
      <img ng-src="{{imageSrc(item.image)}}" width="100%" />
      <div class="food-list__item__name">{{item.name}}</div>
      <div class="food-list__item__prize">Prize:
      <strong>{{item.prize}}</strong></div>
      <div class="food-list__item__quantity">Available Quantity:
      <strong>{{item.quantity}}</strong></div>
      <div class="food-list__item__contents">Ingredients:
      <strong>{{item.contents}}</strong></div>
      <div class="food-list__item__chef">Chef:
      <strong>{{item.chef_name}}</strong></div>
      <div class="food-list__item__chef">Time:
      <strong>{{item.time}}</strong></div>
      <div ng-if="currentUser && currentUser.profile.role ===
      'admin'" class="food-list__item__edit">
        <a href="/edit-item/{{item._id}}">Edit</a> |
        <a href="javascript:void(0)"
        ng-click="deleteItem(item)">Delete</a>
      </div>
    </li>
  </ul>
</div>
```

In the template, we show the edit and delete links only to the admin user. While deleting, we call the `UpdateItem` server method to set the `is_deleted` flag as true. This will eliminate all the deleted items from our subscription because of the query in the publication. Prepare and add some more items for your users and make them happy!

# Editing food items

The next task is to allow the admin user to edit the food items.

# Route

We will have a different controller and route, but we will reuse the same template. Add the following route to the app configuration, as we did earlier:

```
.state("editItem", {
    url: '/edit-item/:itemId',
        templateUrl: 'client/templates/additem.ng.html',
        controller: 'EditItemController',
        resolve: {
          "currentUser": ["$meteor", function($meteor) {
                return $meteor.requireUser();
            }],
      "subscribe": ["$meteor", "$stateParams", function($meteor,
      $stateParams) {
                        return $meteor.subscribe('foodMenu', {_id:
                        $stateParams.itemId});
                }]
        }
    });
$urlRouterProvider.otherwise('/');//handling wrong routes
```

The code explains a lot. We need to pass the food item document ID with the route. We need to create a controller with the name `EditItemController`. In the resolve part, we can subscribe to the food item document that we want to edit.

Append the following piece of code to the same `app.js` file to handle redirection in case of unauthorized users, which we discussed earlier:

```
angular.module("FoodMenu").run(["$rootScope", "$state",
    function($rootScope, $state) {
    $rootScope.$on("$stateChangeError", function(event, toState,
        toParams, fromState, fromParams, error) {
        if (error === "AUTH_REQUIRED") {
            $state.go('list');
        }
    });
}]);
```

# The EditItem controller

Let's create the `EditItemController` by adding a controller file,
`editItemController.js`, in the `client/controllers` directory. Add the following
controller code to the file:

```
angular.module("FoodMenu").controller("EditItemController", [
  "$scope","$meteor",'$stateParams',"$state",
  function($scope, $meteor, $stateParams, $state) {
    $scope.edit = true;//flag to indicate edit mode
    $scope.item = $meteor.object(FoodMenu, $stateParams.itemId,
    false); //getting single document
    $scope.imageSrc = "";

    var imageId = $scope.item.image; //old image id
    //Subscribing to image associated with this document
    $meteor.subscribe('images', {_id:
    $scope.item.image}).then(function() {
      try {
        $scope.imageSrc = Images.findOne({_id:
        $scope.item.image}).url();
      } catch(e) {}
    });
    $scope.selectImage = function(element) {
      $scope.item.image = element.files[0];
      //Instant update using FileReader
      var fileReader = new FileReader();
      fileReader.readAsDataURL($scope.item.image);
      fileReader.onloadend = function() {
        $scope.$apply(function() {
              $scope.imageSrc = fileReader.result;
          });
      }
    }
    //Toggling the properties using $watch
  $scope.$watch("item.is_deleted", function(newValue, OldValue) {
      if(newValue === true) {
        $scope.item.is_published = false;
      } else {
        $scope.item.is_published = true;
```

```
      }
    });
    $scope.$watch("item.is_published", function(newValue,
    OldValue) {
      if(newValue === true) {
        $scope.item.is_deleted = false;
      } else {
        $scope.item.is_deleted = true;
      }
    });
    $scope.save = function() {
      //Image updated
      if($scope.item.image instanceof File) {
        //removing previous image from database
        Images.remove({_id: imageId});
        //Insert the new image
        Images.insert($scope.item.image, function(err, res) {
          if (!err) {
          //On success call the server method to update the
          document after update the image id
            $scope.item.image = res._id;
            $meteor.call("UpdateItem",
            $scope.item.getRawObject());
            $state.go("list");//redirect
          } else {
            console.log(err);
          }
        });
      } else {
        $meteor.call("UpdateItem", $scope.item.getRawObject());
        $state.go("list");
      }
    }

  }
]);
```

## Demystifying controller logic

In the edit controller, we define an edit flag in `$scope` to help us display a few elements in the template only in edit mode. Next, we get only the required collection that we are going to edit. The `$meteor.object` method does this if we pass the collection and the ID of the interested document. The third parameter to `$meteor.object` accepts boolean, which is a flag to indicate if we want to update the document immediately as the values change in `$scope`. The very next line is where we subscribe the `Images` collection so that we can display the image while editing. Once the subscription is ready, it updates the `imageSrc` property, which in turn will render the image. In the `selectImage` method, we show an immediate file preview using `FileReader`. There are two watch expressions defined to toggle the `is_deleted` and `is_published` properties. Finally, the `save` method is where we update the database document by calling the server method. If a new image is uploaded, the old image is removed from the database, the new image is saved to the database, and then `$scope.item` is updated in the database document. If an image is not changed, we just update the document by calling the server method.

The `FoodMenu` application is ready. We have developed an Angular-Meteor application. Now, treat yourself at your favorite restaurant for what we have achieved. As usual, there is a lot of scope to improve the application. Take it as homework and learn some new solutions. There is a great tutorial available at `http://angular-meteor.com/api/AngularMeteorObject` that can help us to do even more stuff with Angular.js in MeteorJS.

# React.js with MeteorJS

Developing an application with Angular.js is fun. Similarly, we are going to develop the same application in React.js. React.js is simpler than Angular.js. Though there are not many tutorials or applications out there, giving it a try is worth it.

The reactivity part of React.js revolves around `states` and `props`. The `States` and `props` are equivalent to `$scope` in Angular.js. The change in value will render the markups again. However, React.js provides more handle over when to update the markup with a set of APIs. This is a more react-specific topic to discuss. The only thing to remember is, too many rerenders will affect application performance. So, you should efficiently mind how you handle the state changes.

Like `angular-meteor`, we have MeterorJS specific React.js packages. There are two important packages: `react` and `react-template-helper`. With the React.js package, we get the `ReactMeteorData` mixin that provides a few methods to deal with MeteorJS' reactive data. The package also provides a jsx compiler and obviously the react runtime. The `react-template-helper` package helps to embed the React.js components into Blaze templates. There are a only few packages in development. We have to wait some more for the ecosystem to grow.

# ReactFoodMenu

Let's start developing our `FoodMenu` application using React.js. Create a MeterorJS application with the name `ReactFoodMenu` and remove all default files.

## Setup

Remove the `autopublish` and `insecure` packages. Create the `server`, `client`, and `lib` directories and `index.html` file inside the application root.

## Server

Copy all the files from the `server` and `lib` directories from the previous application and paste them into the appropriate directories in the current application. We are using the same server implementation that we used in the previous application, except for a small change in the server method. We need to return the inserted documents ID from the method. So, change the `SaveItem` method in `server/methods.js` to look like the following code:

```
"SaveItem": function(item) {
  item.created_at = new Date();
  return FoodMenu.insert(item, function(err, res) {
    console.log(err, res);
  });
},
```

The rest of the changes will only be in the client. It's just that, instead of Angular.js, we will use React.js components.

## Client

Inside the `client` directory, create the `add-edit-item`, `commons`, `header`, `list`, and `stylesheets` directories. Copy the same `styles.css` from the previous application into the `client/stylesheets` directory.

## Client packages

All the copy-paste stuff is done and it's time to write React.js code. We will install the necessary packages and start writing React.js code. Add react-specific packages, such as the `react` and `react-template-helper` packages, using the `meteor add` command. Then, add the `cfs:standard-packages` and `cfs:gridfs` packages for file upload handling. Add `twbs:bootstrap`, `accounts-password`, and `ian:accounts-ui-bootstrap-3` for styles and accounts, and `meteorhacks:flow-router` for routing.

# The first React.js component

To start with, let us add a loading component for our application. Create the `loading.jsx` file in the `client/commons` directory. Add the following loading component `jsx` code to the file:

```
Loading = React.createClass({
  render: function() {
    return (
      <div className="loading">Loading...</div>
    );
  }
});
```

This is all it takes to create a loading component. We can use this component to display while other components are waiting for a subscription. There are things to note here. The `Loading` variable must be global so that it can be accessed across files. If you have worked with React.js before, this syntax is pretty easy to understand. Every component must have a render method that returns markups. These markups look like HTML, but they are not really HTML. If you look closely at the markup inside the `render` method, you will find the `className` attribute instead of the `class` attribute. Similarly, there are other attributes and markups that are jsx-specific. Later, the `jsx` compiler will compile the markups to HTML and render them.

# The header section

Let us develop the header portion first and provide ways to create a login. Add the following piece of HTML to `index.html`:

```
<head>
    <title>ReactFoodMenu</title>
</head>
<body>
    <header id="siteheader">
```

```
        <div class="logo"><a href="/">ReactFoodMenu</a></div>
      <div class="login-signup">
        <ul>
          {{> loginButtons}}
        </ul>
      </div>
      </header>
      <div class="container" id="sitebody"></div>
  </body>
```

Start the application and visit it in the browser. You will find the header is ready. We could have designed the header as a react component and then rendered. However, we would have had to render it on every route change that was being added to maintenance. It is better to put it as a plain HTML markup to avoid overhead. Another reason is the `loginButtons` template, which cannot be used in the react component. As of now, there is no way to integrate MeterorJS templates inside the react component's `jsx` code. However, vice versa is possible with the help of the `react-template-helper` package. We will see how to use it.

# The React.js component in Blaze

In the header, we have to show the **Add Items** link for the admin user. We will create a Blaze template that will use the React.js component to show the link. We can do it just by using Blaze. However, to demonstrate how to use the React.js component inside Blaze, we will perform this step. Create the `addItemsButton.html` template file into the `client/header` directory. Add the following template code to the file:

```
<template name="addItemBtn">
  <!-- without which react component won't work -->
  <div>
  {{> React component=AddItemLink currentUser=currentUser}}
  </div>
</template>
```

This is how we have to use the React.js component inside the Blaze template. There are two things to notice in the preceding code. There is a `div` wrapper to the React. js component because we cannot use the component as a direct child to the Blaze template. Another important point is, React.js doesn't allow us to render components with siblings. So we cannot add anything as sibling to the react component. The `component` attribute is mandatory and this is where we pass the component we want react to render inside the Blaze template. The rest of the attributes are optional and are available as `props` inside the component.

However, for Blaze to recognize the React.js component, we have to create a
template helper that will pass the React.js component to the template. Create the
`addItemBtnHelper.js` helper file inside the `client/header` directory and add the
following helper code:

```
Template.addItemBtn.helpers({
  AddItemLink: function() {
    return AddItemLink;
  },
  currentUser: function() {
    return Meteor.user();
  }
});
```

Finally, create the React.js component by creating a `jsx` file, `addItemsLink.jsx`,
as follows:

```
AddItemLink = React.createClass({
  render: function() {
    if(this.props.currentUser &&
    this.props.currentUser.profile.role === "admin") {
      return <a className="btn btn-primary" href="/add-item">Add
      Items</a>
    } else {
      return null;
    }
  }
});
```

Add the template we created in `index.html` inside the header above the
`loginButtons` template, as follows:

```
<div class="login-signup">
  <ul>
    <li>{{> addItemBtn}}</li>
    {{> loginButtons}}
  </ul>
</div>
```

The **Add Items** link is ready. Create an account and go to the database and make
yourself the admin user using the following query:

**db.users.update({_id: <id>}, {$set: {profile: {role: "admin"}}})**

After the query execution, the link will show up in the header. The header part is
now done.

# The container section

We'll concentrate on the food item creation part. To get started, we need a route. Instead of `iron-router`, we are going to use `FlowRouter` in this application. `FlowRouter` is getting popular because of the design concerns of many developers. Take a look at FlowRouter's GitHub repository (`https://github.com/kadirahq/flow-router`). Regarding the routes in the React-Meteor combination, there are two ways to do things. One is using the `FlowRouter`, which we are going to look at in a while. Another is to use `ReactRouter`, which is a little complicated. If you want to explore `ReactRouter`, you can look at the `ReactRouter` package (`http://rackt.github.io/react-router/`) and get some idea.

# The application route

Coming back to our application, as we have decided to use `FlowRouter`, create a route file, `router.jsx`, inside the `client` directory. We have created it to be a `jsx` file because we will mount our react components inside the route.

## Route – AddItem

Let's add the `/add-item` route to the router file. Add the following route code to the `router.jsx` file:

```
FlowRouter.route("/add-item", {
  name: "addItem",
  action: function(params) {
     $(document).ready(function() {
      var siteBody = document.getElementById("sitebody");
      //clearing previous component to clear the state
      React.unmountComponentAtNode(siteBody);
      React.render(<AddEditItem />, siteBody);
     });
  }
});
```

`FlowRouter` allows us to specify a name to our route to refer it elsewhere and also an action that will be the implementation logic. It also allows us to add subscription registration, which we will see while adding other routes. Look at the package documentation if you want to know more. In the action method, we have written react-specific code:

```
React.render(<AddEditItem />, siteBody);
```

This line renders the component in a given DOM element. The `React.render` method takes a react component as the first parameter and a plain DOM element as the second parameter. Before render, we need to unmount or remove anything present inside the container that we use (`siteBody`). React will try to reuse the DOM elements that might cause issues while switching between the add and edit screens.

# The AddEditItem component

It is time to create the `AddEditItem` component. We will use the same component to handle both add and edit. Create the `add-edit-item.jsx` file inside the `client/ add-edit-item` directory and add the component code to it:

```
AddEditItem = React.createClass({
  mixins: [ReactMeteorData],
  getMeteorData: function() {
    return {};
  },
  handle: true,//will be used inside getMeteorData method
  imageFile: null,//will be used to manipulate image for saving
  render: function() {
    return null;
  }
});
```

When we deal with MeterorJS reactive data, we have to add the `ReactMeteorData` mixin to the component. This mixin provides the `getMeteorData` method and the `data` properties that help the component to react to the reactive data from MeteorJS. Whatever is returned from the `getMeteorData` method will be available in `this. data` inside the other methods of the component. Replace the `getMeteorData` method body with the following code to handle the rendering logic:

```
return {
  //Use handle to show loading state
  subscriptionLoading: !this.handle
};
```

The `this.data.subscriptionLoading` property will be useful in edit mode. In the component methods, we can use `this.data.subscriptionLoading` to check the ready status of the subscriptions, if any. In the `render` method, we can use `this.data.subscriptionLoading` like the following code to show the loading component:

```
if(this.data.subscriptionLoading) {
  return <Loading />
}
```

## The initial state

React.js components have the `getInitialState` method in which we will define our default values for our form and fields. The idea is to put all the fields in to the `state` and send it to server while saving the data. Add the following initial state declaration code to the component:

```
getInitialState: function() {
  return {
    name: "",
    chef_name: "",
    contents: "",
    time: "Lunch",
    quantity: 1,
    prize: 10,
    is_published: true,
    is_deleted: false
  };
}
```

There is a small change of plan here. We are not going to store the image ID in the `FoodMenu` collection's document. Instead, we will store the `FoodMenu` collection's document ID in the `Images` collection's document.

## Component handlers

We need three more helpers to work with the form. One to handle images, one to save the form data, and one to handle the toggling of the checkboxes. Add the following methods to the component:

```
selectImage: function(e) {
  var fileReader = new FileReader();
  this.imageFile = e.target.files[0];//will be used for saging
  this.refs.imgDisplay.getDOMNode().src = "";
  fileReader.readAsDataURL(file);
  fileReader.onloadend = function() {
    this.refs.imgDisplay.getDOMNode().src = fileReader.result;
  }.bind(this);
},
```

This method will be called `onChange` of the file input field. React provides the `refs` attribute to reference a DOM element by the name with which we are setting the image chosen for the image tag next to the file input for an instant preview.

Next, add the field change handler code to the component:

```
handleChange: function(e) {
  var obj = {};
  obj[e.target.id] = e.target.value;
  if(e.target.type === "checkbox") {
    if(e.target.id === "is_published") {
      obj["is_deleted"] = !e.target.checked;
    }
    if(e.target.id === "is_deleted") {
      obj["is_published"] = !e.target.checked;
    }
    obj[e.target.id] = e.target.checked;
  }
  this.setState(obj);
},
```

In this method, we will do two things. One is set the changes to the state and another is toggle the checkboxes. One small application-level convention is to make the ID of the fields the database collection field name, as in the `getInitialState` method, so that we don't have to write separate handlers for each field.

Next is the submit handler. Add the following code to the component:

```
save: function(e) {
  e.preventDefault();
  var method = "SaveItem";
  if(this.imageFile instanceof File) {
    var image = new FS.File(this.imageFile);
    Meteor.call(method, this.state, function(errSave,
    resSavedId) {
    if (!errSave) {
      image.metaData = {itemId: (resSavedId || this.props.id)};
      Images.insert(image, function(err, res) {
      if (!errSave) {
        FlowRouter.go("list");
      } else {
        console.log(err);
      }
      });
    } else {
      console.log(err);
    }
```

```
      }.bind(this));
    } else {
      Meteor.call(method, this.state, function(err, res) {
      if (!err) {
        FlowRouter.go("list");
      } else {
        console.log(err);
      }
      });
    }
  },
```

We save the data first and then the image into which we set `metaData.itemId`, which is the ID of the document inserted. On success, we redirect the user to the listing page.

## React.js markups

Finally, we will provide the markups for our component. Add the following code to be returned instead of `null` from the `render` method of the component:

```
(<div className="container">
  <form role="form" onSubmit={this.save}>
      <div className="form-group">
          <label htmlFor="name">Item Name</label>
          <input required type="text" className="form-control"
          id="name" onChange={this.handleChange} placeholder="Food
          Item Name" defaultValue="" value={this.state.name} />
      </div>
      <div className="form-group">
          <label htmlFor="chef_name">Chef Name</label>
          <input type="text" className="form-control"
          id="chef_name" onChange={this.handleChange}
          placeholder="Chef Name" value={this.state.chef_name}/>
      </div>
      <div className="form-group">
          <label htmlFor="contents">Ingrediants</label>
          <input required type="text" className="form-control"
          id="contents" onChange={this.handleChange}
          placeholder="Ingrediants" value={this.state.contents} />
      </div>
      <div className="form-group">
```

```
    <label htmlFor="time">Time</label>
    <select className="form-control" id="time"
    defaultValue={this.state.time}
    onChange={this.handleChange}>
       <option value="BreakFast">BreakFast</option>
       <option value="Lunch">Lunch</option>
       <option value="Dineer">Dinner</option>
    </select>
</div>
<div className="form-group">
    <label htmlFor="image">Image</label>
    <input required={true} type="file" ref="imageFile"
    className="form-control" id="image" placeholder="Image"
    onChange={this.selectImage} />
</div>
<div className="form-group"><img ref="imgDisplay"
width="300px" src={this.state.image} /></div>
<div className="form-group">
    <label htmlFor="quantity">Quantity</label>
    <input required type="number" className="form-control"
    id="quantity" onChange={this.handleChange}
    placeholder="Quantity" value={this.state.quantity} />
</div>
<div className="form-group">
    <label htmlFor="prize">Prize</label>
    <input required type="number" min="20" max="2000"
    className="form-control" onChange={this.handleChange}
    id="prize" placeholder="Prize" value={this.state.prize}
     />
</div>
<div className="form-group">
    <label htmlFor="is_published">Is Published</label>
    <input type="checkbox" ref="is_published"
    className="form-control" id="is_published"
    onChange={this.handleChange} placeholder="Is Published"
    checked={this.state.is_published} value="is_published"
    />
</div>
<div className="form-group">
    <label htmlFor="is_deleted">Is Deleted</label>
    <input type="checkbox" ref="is_deleted"
    className="form-control" id="is_deleted"
    onChange={this.handleChange} placeholder="Is Deleted"
    checked={this.state.is_deleted} value="is_deleted" />
```

```
        </div>
        <button type="submit"
        className="btn btn-default">Submit</button>
    </form>
</div>);
```

Now, we will be able to create food items. Visit `http://localhost:3000/add-item` in the browser and you will see the form to create food items. The redirection won't work because we haven't created the list page route yet.

We have to revisit the `AddEditItem` component to provide the edit functionality, which we will do after the listing page.

# The listing section

Add food item component is ready. We need the listing page to display the added items. Let's start developing the listing page.

## The listing route

Add the following list page route to the `router.jsx` file:

```
FlowRouter.route("/", {
  name: "list",
  subscriptions: function(params) {
      this.register('foodMenu', Meteor.subscribe('foodMenu'));
      this.register('images', Meteor.subscribe('images'));
    },
  action: function(params) {
    $(document).ready(function() {
        var siteBody = document.getElementById("sitebody");
        React.render(<List />, siteBody);
    });
  }
});
```

`FlowRouter` allows us to register subscriptions from the route by a key, but does not wait for the subscription to complete. If we want to wait for the subscription, we can use the `FlowRouter.subsReady` method inside the component. We have subscribed for both the collections in the route. Route is ready for use.

## The list component

However, the application will throw the "List is not defined" exception. We have to define our `List` component. Create the `list.jsx` file inside the `client/list` directory and add the following code to the file:

```
List = React.createClass({
  mixins: [ReactMeteorData],
  getMeteorData() {
    var foodMenuHandle = FlowRouter.subsReady("foodMenu")
      imagesHandle = FlowRouter.subsReady("images");

    return {
      subscriptionLoading: !foodMenuHandle || !imagesHandle,
      menu: FoodMenu.find({}).fetch(),
      currentUser: Meteor.user()
    };
  },

  deleteItem: function(id) {
    Meteor.call("UpdateItem", {
      _id: id,
      is_deleted: true,
      is_published: false
    });
  },

  render: function() {
    //showing loading component till subscriptions are ready
    if(this.data.subscriptionLoading) {
      return <Loading />;
    }
    if(!this.data.menu.length) {
      return <div className="no-data">No Items Available</div>
    }
    return (
      <ul className="food-list row">
        {
          this.data.menu.map(function (item) {
            var image = Images.findOne({"metaData.itemId":
            item._id}).url(),
```

```
            edit = "/edit-item/"+item._id;
        return (
          <li key={item._id} className="col-xs-12 col-md-3
          col-md-offset-1 food-list__item">
            <img ref="image" src={image} width="100%" />
            <div className="food-
            list__item__name">{item.name}</div>
            <div className="food-list__item__prize">Prize:
            <strong>{item.prize}</strong></div>
            <div className="food-
            list__item__quantity">Available Quantity:
            <strong>{item.quantity}</strong></div>
            <div className="food-
            list__item__contents">Ingredients:
            <strong>{item.contents}</strong></div>
            <div className="food-list__item__chef">Chef:
            <strong>{item.chef_name}</strong></div>
            {
            this.data.currentUser &&
            this.data.currentUser.profile.role === 'admin' ?
              (<div className="food-list__item__edit">
                <a href={edit}>Edit</a> | <a
                href="javascript:void(0)"
                onClick={this.deleteItem.bind(this,
                item._id)}>Delete</a>
              </div>) : null
            }
          </li>
        );
          }.bind(this))
      }
    </ul>
  );
  }
});
```

Here, we wait for the subscription using the `FlowRouter.subsReady` method that will return boolean values. Once the subscription is ready, the `getMeteorData` method will be called, which will set the new values in `this.data`. Then, we render the component using the `FoodMenu` documents that are available in `this.data.menu`. We iterate over the `menu` array and form the list items. On each `li`, we have to add a `key` attribute that is a react convention. We have added the edit and delete links for an admin user as well.

The `Images` collection is not returned from the `getMeteorData` method. It is not mandatory to return anything, unless we want them to be available in `this.data`. We can directly call the `Images` collection by the instance and use it as we did inside the map loop. The rest of the code is self-explanatory.

## The edit items route

The last part of the application is editing the food item we have created. Add the following edit route to the `router.jsx` file:

```
FlowRouter.route("/edit-item/:_id", {
  subscriptions: function(params, queryParams) {
        this.register('editItem', Meteor.subscribe('foodMenu',
        {_id: params._id}));
        this.register('editItemImage', Meteor.subscribe('images',
        {"metaData.itemId": params._id}));
    },
  action: function(params) {
    $(document).ready(function() {
      var siteBody = document.getElementById("sitebody");
      React.unmountComponentAtNode(siteBody);
      React.render(<AddEditItem edit={true} id={params._id} />,
      siteBody);
    });
  }
});
```

In the preceding route, we have added a subscription registration only for the interested item, by passing a parameter to the subscription registration. We are passing properties to the react component to indicate it in edit mode. We also pass the ID of the item we are going to edit.

## Edit patch

We will use these properties to make the Add Item form sufficient for edit as well. If we fetch the food item document and set it on `state`, it will get populated in the form as per our code in the `AddEditItem` component. In the `getInitialState` method, we will do this by prepending the following code:

```
if(this.props.edit === true) {
  FlowRouter.subsReady("editItem", function() {
    this.setState(FoodMenu.findOne({_id: this.props.id}));
  }.bind(this));
}
```

We check whether it is in edit and then wait for the subscription to be ready. Once it is ready, find the item from the collection and set it to the react component state. We have to wait for both the `FoodMenu` and `Images` collections to be ready before rendering. Prepend the following code to the `getMeteorData` method to make the `this.data.subscriptionLoading` flag `true`:

```
if(this.props.edit === true) {
  Tracker.autorun(function() {
  this.handle = FlowRouter.subsReady("editItem") &&
  FlowRouter.subsReady("editItemImage");
  }.bind(this));
}
```

This piece of code will check for the edit mode and register a tracker. In MeteorJS, a Tracker can run a method when any referred reactive data source changes. We use `Tracker.autorun` to set the `this.handle` flag to `true` when both the subscriptions are ready.

In the `save` method, we have purposefully defined a variable named `method` and have assigned the server method name to it. Based on the mode of operation, we can change the variable value that will call the appropriate methods of the server. Change the `method` variable declaration to match the following code:

```
var method = this.props.edit === true ? "UpdateItem" : "SaveItem";
```

In edit mode, we call the `UpdateItem` server method; in create mode, we call the `SaveItem` server method.

Finally, in the `render` method, we need to make two changes. Prepend the following code that will take care of displaying loading components until the subscriptions are ready, and set the image URL for displaying the image in edit mode:

```
if(this.data.subscriptionLoading) {
  return <Loading />
}
var image = this.props.edit === true ?
Images.findOne({"metaData.itemId": this.props.id}).url() : "";
```

We have to change the `required` attribute in the file input field so that it will not be mandatory while editing. Change the required attribute to `required={this.props. edit === true ? null: true}`, which will do the job. These are all the changes needed for edit to work. Go to listing and click on edit. You will find that things are working as expected.

For more information on react with MeteorJS, visit `http://react-in-meteor. readthedocs.org/en/latest/`.

# d3.js with MeteorJS

As a final part of the chapter, we are going to develop a simple data visualization using d3.js with MeteorJS

d3.js is a famous data visualization library. It has a number of methods to support various kinds of visualization. We are going to see a small example that can portray the power of d3.js when combined with MeteorJS' reactivity.

## DataViz

Create a MeterorJS application, such as `DataViz`. Add the d3.js package to the application using the `meteor add d3js:d3` command and make the following changes.

## HTML

Replace the body content of `dataViz.html` with the following HTML code:

```html
<header id="siteheader">
  <div class="logo"><a href="/">Data Viz</a></div>
</header>
<div class="container" id="sitebody"></div>
```

Remove the template below the body tag.

## Server

We will insert a few pieces of visualization-related data to a collection. Remove the content of `dataViz.js` and add the following set of codes:

```javascript
DataViz = new Mongo.Collection("dataViz");
if (Meteor.isServer) {
  Meteor.startup(function() {
    if (DataViz.find({}).count() <= 0) {
      var counter = 1;
      var id = setInterval(Meteor.bindEnvironment(function() {
        DataViz.insert({
          sno: counter++,
          temperature: Math.random() + 30
        });
        if (DataViz.find({}).count() === 20) {
```

```
            clearInterval(id);
        }
      }), 2000);
    }
  });
}
```

We have declared a collection and then inserted some data into the collection on application startup. We have inserted data in an interval of two seconds using the `setInterval` timer method. Did you notice `Meteor.bindEnvironment` inside the `setInterval` callback? MeteorJS usually runs the server code in Fiber. When we write a sync code with callback, out of MeteorJS, the callback must be run within the Fiber. To do this, we use `Meteor.bindEnvironment` that will not only run the callback within Fiber, but also binds the code with the variables from the environment.

Why did we insert data in two second interval? The visualization that we are going to develop should be dynamic. There are charts that instantly update themselves as the data arrives at various intervals. These kinds of charts will be very suitable to see MeterorJS reactivity in data visualization. We are going to develop a spline chart that will update itself as we insert data to the collection. To see the instant update, we are delaying each insertion by two seconds.

# Client – d3.js code

Next, let us add the d3.js code that draws the axes and the line as per the data. Append the following visualization code to the `dataViz.js` file:

```
if (Meteor.isClient) {
  $(document).ready(function() {
    var margin = {
        top: 20,
        right: 20,
        bottom: 30,
        left: 50
      },
      canvas = document.getElementById("sitebody"),
      width = 1000,
      height = 600,
      data = DataViz.find({}).fetch(),
      fields = ["sno", "temperature"];

    var x = d3.scale.linear().range([0, width]);
```

```
    var y = d3.scale.linear().range([height, 0]);

    var xAxis = d3.svg.axis().scale(x).orient("bottom");
    var yAxis = d3.svg.axis().scale(y).orient("left");
    var line = d3.svg.line()
      .interpolate("basis")
      .x(function(d) {
        return x(d[fields[0]]);
      })
      .y(function(d) {
        return y(parseFloat(d[fields[1]]));
      });

    var svg = d3.select("#sitebody").append("svg")
      .attr("width", width + margin.left + margin.right)
      .attr("height", height + margin.top + margin.bottom)
      .append("g")
      .attr("transform", "translate(" + margin.left + "," +
      margin.top + ")");
    svg.append("g")
      .attr("class", "x axis x-axis")
      .attr("transform", "translate(0," + height + ")")
      .call(xAxis);

    svg.append("g").attr("class", "y axis y-axis").call(yAxis);
    svg.append("path").datum(data).attr("class", "line").attr("d",
    line);
  });
}
```

In the preceding code, we have defined the margins and the container to draw the chart. We have fetched the collection and then defined the axes-related calculations. The `d3.svg.line()` method is the line function that draws each segment using the data we had inserted during application startup. We append a `svg` tag into the container and set the layout properties. Finally, we feed the data fetched from the collection to the line function we had defined earlier and drew the lines using a `path` element.

The preceding code will work for static data. How will it reflect the data that is inserted after the chart is rendered? We have to redraw the chart as and when the data is inserted. There are two ways to do it. Using `.observe` on cursor or by using `Tracker.autorun`.

We are going to use the cursor observer in our example. Append the following code to the ready function in the preceding code:

```
DataViz.find({}).observe({
  added: function(doc) {
    var datum = {};
    datum[fields[0]] = doc[fields[0]];
    datum[fields[1]] = doc[fields[1]];
    data.push(datum);
    if (data.length > 10) {
      data.splice(0, 1);
    }
    svg.select(".line")
      .transition()
      .duration(100)
      .attr("d", line)
    x.domain(d3.extent(data, function(d) {
      return d[fields[0]];
    }));
    y.domain(d3.extent(data, function(d) {
      return parseFloat(d[fields[1]]);
    }));
    svg.select('.x-axis').call(xAxis);
    svg.select('.y-axis').call(yAxis)
  }
});
```

The cursor's observers provide us the ability to run a set of codes when a document is added, changed, or removed from the collection. In the preceding code, on every insertion, the cursor's `added` callback that we have passed will be called. We update the chart by inserting that data into the chart and thereby redrawing the axes and the lines with the new data. With the `observe` method, we can add the `changed` and `removed` callback as well.

To see things in action, remove all the records from the mongo collection using the meteor mongo console and restart the application. Visit the browser and you will find the chart is instantly updating the lines every couple of seconds, as follows:
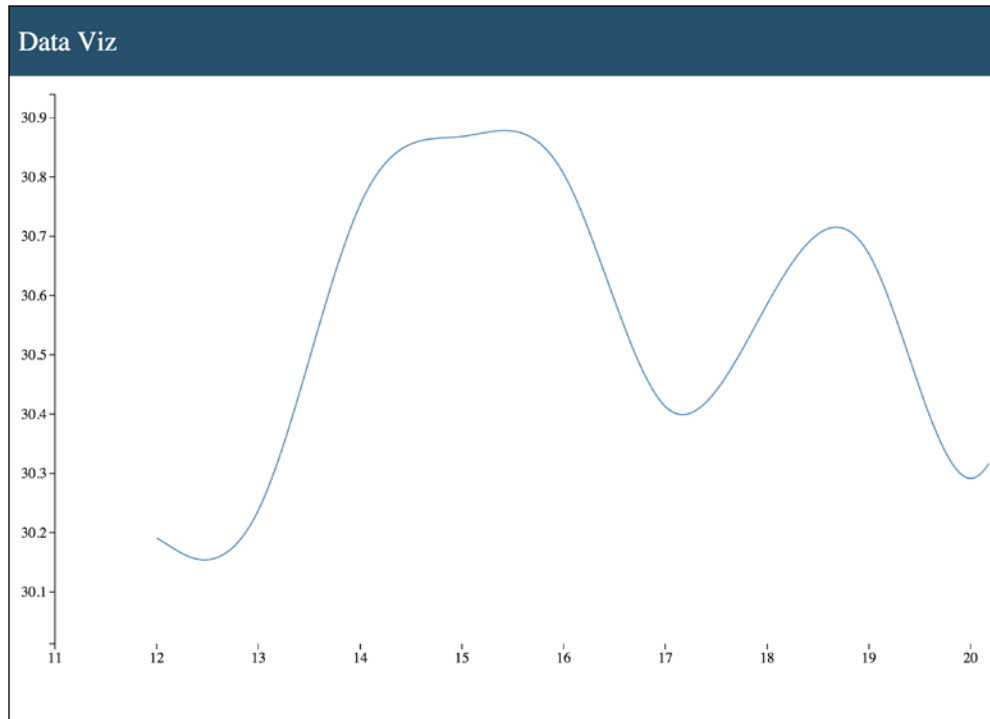


From the browser console, if we insert new data, it will also be reflected in the chart. This is a pretty simple example, but shows how to wire MeteorJS' reactivity with d3.js. This gives us the confidence that we can transfer MeteorJS' reactivity into any piece of code that we write. We have to just find the right place to wire the reactivity in order to redraw the data. However, be sure that it is not against performance.

If you want to build highly reactive data visualization for the analysis of your application, now you know what to do. You have an amazing combination in hand, rock your team with it.

# Integrating any frontend framework with MeteorJS

From all the above applications, one important thing we should note is how we wire MeteorJS' data source reactivity into our frontend setup. In the case of Angular.js and React, we have packages that do the job elegantly. However, it is not mandatory to have a package to achieve this. In the above d3.js example, all we did was found a way to wire the reactivity to the chart drawing code. We need to do the same with any other frontend framework. We have to figure out a way to induce this reactivity into the frontend framework's flow.

For example, suppose we want to use Backbone.js as our frontend framework, we just have to figure out how to insert the new document into the Backbone collection as and when data is added to the mongo collection. We can use the Tracker or cursor observers to do the job.

Blaze is awesome. It will be improved by the community. However, if you prefer your choice of framework, with MeteorJS, it is very much possible to integrate it. Before making the decision, think twice about whether the effort you are going to put is worth it.

# Summary

I hope you have enjoyed the chapter and the pace in which we have learned to develop a MeterorJS application using popular frontend frameworks. Let us summarize everything that we have learned in this chapter. We learned to use Angular.js using the `angular-meteor` package, and React.js using the `react` package. We also learned to use d3.js with MeteorJS' reactivity to support reactive visualizations. With an example, we have learned that we can observe the collection cursor that opens up a way to listen on MeterorJSCollection, which is key to integrate any frontend framework or library with MeteorJS. You can experiment more by using MeterorJS with the frameworks of your choice and share it with the community.

In the next chapter, we will learn how to animate interfaces in MeterorJS applications using various libraries.

# 5
# Captivating Your Users with Animation

Animations contribute to both the user experience and the aesthetics of an application. Previously, animations were performed on sites using Flash, which isn't native. After jQuery's simple APIs to perform animations and transitions, both animations and transitions became common on many websites. With HTML5 and CSS3, we got access to many native APIs and properties that helped to animate and transit elements using hardware acceleration. To help developers achieve 60 fps, which is the required frame rate for a perfect jank-free animation, libraries such as Velocity.js and Famo.us appeared. With SVG, we have the ability to create any shape, and this again made available another set of interesting animatable interfaces.

With the recent advancements in terms of animation in HTML, CSS, and JavaScript, there are several tools available to design animations. Chrome has an animation timing control tool in the developer tool. All this focus on animations has led to the design of very interesting and intuitive animations for applications, which improves user experience.

MeteorJS is highly reactive. Whenever there is a change of data, it reflects this in the interface right away. With this kind of instant reactivity, it can be difficult to perform animation, but it is not that hard. In MeteorJS roadmap, there is a place for animation. So we can expect hooks and APIs to perform animations. Does this mean we can't perform animation at present? No, absolutely not. We can still do animation up to a certain extent, and this is what we are going to learn in this chapter.

In this chapter, we are going to cover the following topics:

- Animation in Blaze templates
- Animation using MeteorJS packages with Velocity.js
- Animating SVG using `snap.svg`, d3.js
- Animations using Famo.us

# Animation in Blaze templates

Usually, with Blaze and MeteorJS reactive collections, we want to perform animation in lists while adding or removing items from the collection. Let's see how to animate a reactive list's manipulation.

Create a MeteorJS application, such as `AnimationPartyBlaze`, and a collection by prepending the following code to `AnimationPartyBlaze.js`:

```
MyList = new Mongo.Collection("my_list");
```

Create a template that shows the list of documents from the collection we have created and append it to `AnimationPartyBlaze.html`, as follows:

```
<template name="mylist">
  <div>
    <ul>
      {{#each list}}<li>{{this.name}}</li>{{/each}}
    </ul>
  </div>
</template>
```

We have to provide a helper that passes the collection to the template. Create a template helper in `AnimationPartyBlaze.js` for the preceding template and return the collection, as follows:

```
Template.mylist.helpers({
      list: function() {
          return MyList.find({});
      }
   });
```

To display it, add the template as `{{> mylist}}` next to the `hello` template inside the `body` tag. Now, if we insert a document to the `my_list` mongo collection, this template will show the data. However, we want this insertion to show up with some sort of animation.

Blaze doesn't have inbuilt APIs to perform animation. Instead, all it provides is a hook with which we can perform animations by other available means. We can use CSS classes that perform animation or we can write our own animation logic. For the preceding list's manipulation example, we will use `animate.css`. Add `animate.css` to the application using the `meteor add natestrauser:animate-css` command. Now, we are free to use the `animate.css` animation helper classes. To learn more about `animate.css`, visit the GitHub pages at `https://daneden.github.io/animate.css/`.

What is that hook and how are we going to use it? During DOM manipulation, if Blaze finds the `_uihooks` property in the DOM element, then it will try to execute the respective hooks. We will see that in the code right away. Create the `onRendered` callback for the `mylist` template and add the hook, as follows:

```
Template.mylist.onRendered(function() {
    this.find(".list-container__ul")._uihooks = {
        insertElement: function(node, next) {
            node.classList.add("slideInRight", "animated");
            $('ul')[0].insertBefore(node, next);
        }
    };
});
```

Carefully observe the `_uihooks` property. We can assign an object to this property, which can have the `insertElement`, `removeElement`, and `moveElement` methods. These methods will be executed when Blaze tries to insert, remove, or move the child elements of the element to which these hooks are attached. In our case, the hooks are attached to the `ul` element because we want to animate the `li` structure.

When we provide hooks explicitly, then it is our responsibility to handle the operation as well. In the `insertElement` hook, we are explicitly handling the insertion at the last line of the method by calling `insertBefore`. The hook methods are called with the node, which is to be manipulated (inserted in this case), and the next element for our convenience as arguments. We are adding the animation helper classes from `animate.css` right before the insertion using the `classList` property. See it in action by inserting a few documents to the collection from the browser console using the `MyList.insert({name: "Hello World!"});` code.

Two important things to remember about the hooks are as follows:

- Hooks will work only for elements that are under `nodeType 1`. This means, hooks will not be called on texts that come under `nodeType 3`. To understand more about the `nodeType` property, take a look at the MDN link (`https://developer.mozilla.org/en/docs/Web/API/Node/nodeType`).

- The other important thing is, _ before the `_uihooks` property. In developer notations, a property prefixed by _ means that it can undergo changes and it is not official. So, `_uihooks` is not an official way to animate in MeteorJS. We have to keep an eye on the releases about the changes related to the hook. Why are we using it then? The answer is, because this is the only way, as of now, available to do all the animations within Blaze.

Lists are the common examples to show animations with reactive data sources in MeteorJS. It demonstrates only cursors or collections. What about things in session? A simple thing would be the default `hello` template that comes as a boilerplate code when we create a MeteorJS app. When we click on the button, it increments the `counter` variable in session that reflects immediately on the screen. Can we animate this change? Yes, and let us see how we can do it.

Add the following `onRendered` callback to the `AnimationPartyBlaze.js` file inside the client block:

```
Template.hello.onRendered(function() {
    $('p')[0]._uihooks = {
        insertElement: function(node, next) {
            node.classList.add("zoomIn", "animated");
            $('p')[0].insertBefore(node, next);
        },
        removeElement: function(node) {
            node.classList.add("zoomOut", "animated");
            $('p')[0].removeChild(node);
        }
    };
});
```

From the code, we can clearly figure out that we are performing the `zoomIn` and `zoomOut` animations on insertion and removal respectively. Is it doing the job? Give it a try in the browser. It doesn't work for the reason that the node that we are trying to insert must be of `nodeType 1`.

What is really getting inserted is the the text node which is not of `noteType 1` and thus the hook will not work. How to make it work? If we wrap the `{{counter}}` entry in the `hello` template with a `span` tag, will it work? It won't and the reason is, Blaze splits the text and treats the `{{counter}}` entry as a text node (a separate node). So, the change to the text node doesn't work again.

Change the `hello` helper to look like the following code:

```
Template.hello.helpers({
    counter: function() {
        return "<span>"+Session.get('counter')+"</span>";
    }
});
```

Notice carefully that we are returning the counter from the session, wrapping it with a `span` tag. For Blaze to render it as an HTML tag, change `{{counter}}` to `{{{counter}}}` in the `hello` template. Now, click on the button and see the animation. Pretty cool, isn't it?

Why did this work, but wrapping the span in the template not work? It is because the node that changes (inserted/removed) must be of `nodeType 1`. In the previous case, Blaze detected the text to be the changing entity, not the `span` tag, and thus discarded the hooks. However, in this case, we are inserting/removing `span` every time the value changes. As the `Span` tag is being `nodeType 1`, Blaze executes the hooks.

I hope this gives you a clear idea about how to animate with Blaze and the `_uihooks` property, on any reactive data source.

# Animation using MeteorJS packages with Velocity.js

There are a few packages that help us perform animation. In this section, we will see how to perform page transition on every route change. Also, we will see how to animate elements using a MeteorJS package.

Animation packages that support page transitions are mostly router-specific. We will use the `ccorcos:transitioner` package that is specific to `iron-router`. Let's resume our animation party.

Create a MeteorJS application, such as `AnimationPartyTransition`. Add the `iron:router` and `ccorcos:transitioner` packages to the application. Remove the `hello` template-related code totally from the `.js` and HTML files. We need a layout that will be rendered by `iron-router`. Add the following layout code to `AnimationPartyTransition.html`:

```
<template name="homeLayout">
  {{> yield}}
</template>
```

To demonstrate page transitions, we need two routes. Add the following routes to the client block inside `AnimationPartyTransition.js`:

```
Router.map(function() {
    this.route('home', {
        path: '/',
        template: 'home',
        layoutTemplate: "homeLayout"});
    this.route('aboutus', {
        path: '/about-us',
        template: 'aboutUs',
        layoutTemplate: "homeLayout"});
  });
```

From the preceding routes code, we can see that there are the `home` and `about us` routes. We have to create respective templates for each route. Add the following templates to `AnimationPartyTransition.html`:

```
<template name="home">
    <div class="container home">
      <div>In Home</div>
      <a href="{{pathFor 'aboutus'}}">About us</a>
    </div>
</template>
<template name="aboutUs">
  <div class="container aboutus">
      <div>In About us</div>
      <a href="{{pathFor 'home'}}">Home</a>
    </div>
</template>
```

We will add CSS right away in order to style the pages. Add the following styles to
`AnimationPartyTransition.css`:

```css
body {height: 100vh;margin: 0;padding: 0;color: #fff;background-color:
#907C60;}
.container {text-align: center;font-size: 4em;}
.home {background-color: #2980b9;}
.aboutus {background-color: #c0392b;}
```

Routes are functional. Try visiting the routes after starting the application. Time to
add page transitions. The `ccorcos:transitioner` package gives us the ability to
define what type of transition we want to have on every route change. Also, we have
to follow certain conventions. Firstly, we have to wrap `{{> yield}}` in the layout
with `transitioner` such as `{{#transitioner}}{{> yield}}{{/transitioner}}`.
Then, we will define what type of transition we need on which routes.

Right next to the route's definition in the client block, add the following transition
definitions:

```
Transitioner.transition({
        fromRoute: 'home',
        toRoute: 'aboutus',
      velocityAnimation: {
    in : "transition.slideRightBigIn",
          out: "transition.slideLeftBigOut"
        }});
```

The package offers us the `transition` method with which we can define
our transition type for various routes. In this case, we have defined the
`slideRightBigIn` and `slideLeftBigOut` transitions when navigating from the
home page to the about us page. If we want to have the transition vice versa, then we
have to define it separately.

What if we want all the route changes to have one common transition? Isn't that the
use case most of the times? The `transitioner` package offers us this ability as well.
We can define default transition as follows:

```
Transitioner.default({
  in : "transition.slideRightBigIn",
        out: "transition.slideLeftBigOut"
  });
```

This default registration will transit all the routes with the provided transition. Now, the question that arises is, what is this `transition.slideRightBigIn`? The `transitioner` package uses Velocity.js for animation. The `in` and `out` properties take either the transition supported by Velocity.js, or we can define custom logics. To learn about transitions supported by Velocity.js, visit the **Effects: Pre-Registered** section at `http://julian.com/research/velocity/#uiPack`.

Let's explore the other option of the custom `in` and `out` transitions without using Velocity.js' preregistered transitions. The `in` and `out` properties take normal methods too. If we know how the plugin functions, we will find it really easy to write custom transitions. The package internally uses `_uihooks` to perform animation, as we did in the previous section. Visit the source code (`https://github.com/ccorcos/meteor-transitioner/blob/master/lib/transitioner.coffee`) and it will be very clear. The `in` property is treated in the `insertElement` hook, and the `out` property is treated in the `removeElement` hook. We are wrapping our templates with the `transitioner` template in the layout that takes control and registers these transitions to its hook in the `onRendered` callback. Pretty simple, isn't it?

So, we can assume that we will get the inserted node and the next node in our custom transition methods, and write the methods as we did in the previous section. Take a look at the following example:

```
Transitioner.default({
  in : function(node, next) {
    $node = $(node)
      $.Velocity.hook($node, "translateX", "100%");
      $node.insertBefore(next)
        .velocity({translateX: ['0%', '100%']}, duration: 500,
        easing: 'ease-in-out', queue: false);
  },
        out: function(node) {
    $node = $(node)
        $node.velocity({translateX: '-100%'},
          duration: 500, easing: 'ease-in-out', queue: false,
          complete: function() {
                  $node.remove();
      });
  }
  });
```

It is not necessary to use Velocity.js inside the transition methods. We can use other animation libraries also, if required. Velocity.js is a very efficient open source animation library that can perform loads of animations, that are jank-free. It is worth giving it a try. The following is a pseudo syntax of the Velocity.js animation API:

```
element.velocity({
  width: "500px",
  property2: value2
}, {
  /* Velocity's default options */
    duration: 400,
    easing: "swing",
    queue: "",
    begin: function(){},
    progress: function(){},
    complete: function(){},
    display: undefined,
    visibility: undefined,
    loop: false,
    delay: false,
    mobileHA: true
});
```

Velocity.js supports other simple syntaxes as well. Take a look at the Velocity.js documentation (`http://julian.com/research/velocity/`) to learn more.

What we have done is a simple page transition on the route change. The package does not suffice for all the needs and it is still under development. We have to keep that in mind while using this package for production.

If you are using `FlowRouter` for your application, page transitions can be performed using `mcissel:flow-transition`. Visit the package at `https://atmospherejs.com/mcissel/flow-transition`.

Next, we are going to see animation using the package. So far, what we have seen is page transition. What if we want to perform animation with the help of packages?

We will use the `percolate:momentum` package to do the same list animation we did in the previous section. Add the `percolate:momentum` package to this application. Create a collection, `MyList`, and add a template to show the list:

```
//prepend this to AnimationPartyTransition.js
MyList = new Mongo.Collection("my_list");

//Append this to AnimationPartyTransition.html
<template name="list">
  <div class="ul-wrapper">
    {{#momentum plugin="fade"}}
      {{#each list}}
      <div>{{this.name}}</div>
      {{/each}}
    {{/momentum}}
  </div>
</template>
```

We need a helper for the list template to serve the list. Add the following helper to the client block in `AnimationPartyTransition.js`:

```
Template.list.helpers({
    list: function() {
        return MyList.find();
    }
});
```

We have to place this list template in our page templates, as follows:

```
<template name="home">
    <div class="container home">
      <div>In Home</div>
      <a href="{{pathFor 'aboutus'}}">About us</a>
    {{> list}}
    </div>
</template>
```

Insert few documents into the collection from the console, and we will see that the insertion in the template happens with the `fadeIn` effect. The package takes care of the animation for us. The only convention we have to follow is to use the momentum template as a wrapper for each list element. Momentum takes the `plugin` attribute as a mandatory one, and it is highly recommended that you visit the momentum GitHub repository (`https://github.com/percolatestudio/meteor-momentum/`) to learn more about the plugin related attributes.

We have one caveat with momentum that generates a `div` element, as a wrapper for our content. For example, if we want to use the `ul` and `li` structure, we cannot use it because we cannot wrap `li` with `div` inside `ul`. So, we have to change the markups to be compatible with what the package offers.

Under the hood, the plugin again uses Velocity.js and `_uihooks` to perform the animation. It isn't surprising, is it?

Go to `www.atmosphere.com` and search for animation packages. You can find a lot of useful packages for the Web and mobile. Carefully choose the one that fits your need and use it in the application.

# Animation using Snap.svg

In this section, we will see how to use the `snap.svg` library with MeteorJS. We are going to develop an interesting application in which we will use `snap.svg` to manipulate SVG. It will be helpful if we have knowledge about the SVG `path` and `circle` elements before starting this section.

To those who don't know about `snap.svg`, it is a SVG manipulation library. We could even say it is like jQuery but for SVG. We will discuss and learn to use the animation APIs provided by `snap.svg`. It is a great tool if we are using SVG for interfaces in the application.

The application we are going to develop is about logging our life events, similar to the Facebook timeline. To keep things simple and to concentrate more on the animation part, we will insert data directly to the database rather than developing interfaces to insert data. Anyway, MeteorJS will take care of reactivity and thus whatever we insert will be available on the screen. We are going to create some SVG interfaces to show the timeline beautifully.

Create an application, for example, `AnimationPartyLifeEvents`. Remove all the hello template-related code. Firstly, we will create a collection to persist the life events. Prepend the following collection instance creation code to `AnimationPartyLifeEvents.js`:

```
/* index, title, content, event_time, is_deleted, created_at,
  updated_at */
LifeEvents = new Mongo.Collection("life_events");
```

We need templates to show each life event. Append the following templates to
`AnimationPartyLifeEvents.html`:

```
<template name="lifeEvents">
  <div class="container">
    {{#each lifeEvents}}
       {{> event}}
    {{/each}}
  </div>
</template>
<template name="event">
  <div class="svg-container {{direction}}">
    <div class="tooltip-content">{{this.content}}</div>
    <svg width="{{svgLayout.width}}"
    height="{{svgLayout.height}}"></svg>
  </div>
</template>
```

Include `{{> lifeEvents}}` inside the `body` tag to render the template.

Inside the client block of `AnimationPartyLifeEvents.js`, create the following
template helper to serve the collection:

```
Template.lifeEvents.helpers({
    lifeEvents: function() {
        return LifeEvents.find({is_deleted: false}, {sort:
        {event_time: -1}
        });
    }
});
```

If you watch the `event` template closely, you can figure out that we are supplying
the SVG layout dimensions from the helper, because we have to use these layout
dimensions later for the SVG manipulation. Add the following layout dimension
code and the template helper code to the client block of the `.js` file:

```
var svgLayout = {height: 200, width: 400};
Template.event.helpers({
    isEven: function() {
        return this.index % 2 === 0;
    },
    svgLayout: svgLayout,
    direction: function() {
```

```
            return this.index % 2 === 0 ? "tooltip-west" :
            "tooltip-east";
        }
    });
```

The template helpers will be helpful when we render the SVG elements.

We will create a method to support data insertion to the collection and keep it accessible to the client. In production, we should not do this. Add the following method after the client block (not within):

```
Meteor.methods({
    insertLifeEvent: function() {
        var currentId = (LifeEvents.findOne({},
        {sort:{index:-1}}) || {}).index || 0;
        event = {
            index: (currentId+1),
            title: currentId+" event",
            content: "Lorem Ipsum is simply dummy text of the
            printing and typesetting industry. Lorem Ipsum has
            been the industry's standard dummy text ever since
            the 1500s.",
            event_time: new Date(),
            is_deleted: false,
            created_at: new Date(),
            updated_at: null
        };
        event.content = event.content.substring(0, 120);
        return LifeEvents.insert(event);
    }
});
```

Just calling the method from the browser console will insert random data into the collection.

A bit of CSS will make the application ready. Add the following CSS to `AnimationPartyLifeEvents.css` and start the application:

```
@import url(http://fonts.googleapis.com/css?family=Gochi+Hand);
body {margin: 0;padding: 0;}
.container {min-height: 200px; overflow: auto; margin-top: 2%;}
.svg-container:first-child {margin-top: 70px;}
.svg-container:not(:first-child) {margin-top: -60px;}
```

```css
.svg-container {margin: auto;width: 400px;position: relative;}
/* Tooltip */
.tooltip-content {
  position: absolute; background: #e35583; z-index: 9999; width:
  200px; bottom: 95%; left: -70%; max-height: 150px;
  margin-bottom: -1em; padding: 20px; border-radius: 20px;
  font-size: 1.1em; text-align: center; color: #fff; opacity: 0;
  cursor: default; pointer-events: none; font-family: 'Gochi
  Hand', cursive; -webkit-font-smoothing: antialiased;
  -webkit-transition: opacity 0.3s, -webkit-transform 0.3s;
  transition: opacity 0.3s, transform 0.3s;
}
.tooltip-west .tooltip-content {
  left: 25em;-webkit-transform-origin: -2em 50%; transform-origin:
  -2em 50%; -webkit-transform: translate3d(0,50%,0)
  rotate3d(1,1,1,30deg); transform: translate3d(0,50%,0)
  rotate3d(1,1,1,30deg); }
.tooltip-east .tooltip-content {
  right: 4em; -webkit-transform-origin: calc(100% + 2em) 50%;
  transform-origin: calc(100% + 2em) 50%; -webkit-transform:
  translate3d(0,50%,0) rotate3d(1,1,1,-30deg); transform:
  translate3d(0,50%,0) rotate3d(1,1,1,-30deg); }
.tooltip-content.show {
  opacity: 1;pointer-events: auto;
  -webkit-transform: translate3d(0,50%,0) rotate3d(0,0,0,0);
  transform: translate3d(0,50%,0) rotate3d(0,0,0,0); }
/* Gap "bridge" and arrow */
.tooltip-content::before,
.tooltip-content::after {
  content: ''; position: absolute;}
.tooltip-content::before { height: 100%; width: 3em;}
.tooltip-content::after { width: 2em; height: 2em; top: 50%;
  margin: -1em 0 0; background: url(/tooltip2.svg) no-repeat
  center center; background-size: 100%; }
.tooltip-west .tooltip-content::before,
.tooltip-west .tooltip-content::after {
  right: 99%; /* because of FF, otherwise we have a gap */}
.tooltip-east .tooltip-content::before,
.tooltip-east .tooltip-content::after {
  left: 99%; /* because of FF, otherwise we have a gap */}
.tooltip-east .tooltip-content::after {
  -webkit-transform: scale3d(-1,1,1);
  transform: scale3d(-1,1,1);}
```

All the ground work is done. As we insert life events to the collection, MeteorJS will render the event template for each life event. The `event` template has a `svg` tag. Inside this `svg` tag, we are going to insert a `path` element and animate it to show the life event pretty nicely. Look at the following image to get a fair idea of what the path element will look like:
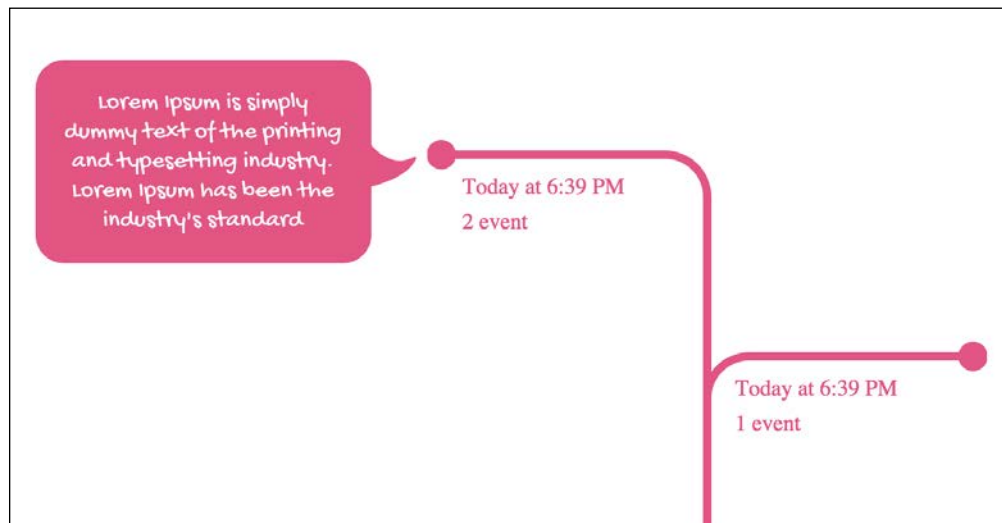


We are going to draw the `path` element inside the `onRendered` callback of the event template. Add the following callback code to the client block of the `.js` file:

```
Template.event.onRendered(function() {
  /** Here goes our code **/
});
```

We will initialize a few variables at the beginning of the callback for later usage. Add the initialization code to the callback:

```
var data = this.data,
      isEven = (this.data.index % 2 === 0),
      circleRadius = 10,
      arcRadius = 30,
      pathStrokeWidth = 6,
      pathData = {
          x: svgLayout.width / 2,
          y: svgLayout.height,
          lines: {
```

```
                verticalLineTo: -(svgLayout.height - arcRadius
                - 22),
                horizontalLineTo: (svgLayout.width / 2) -
                arcRadius - circleRadius
            },
            arc: arcRadius + ", " + arcRadius + " 0 0 1 " +
            arcRadius + ", -" + arcRadius
        };

    if(isEven === false) {
        pathData = {
            x: svgLayout.width / 2,
            y: svgLayout.height,
            lines: {
                verticalLineTo: -(svgLayout.height - arcRadius
                - 22),
                horizontalLineTo: -((svgLayout.width / 2) -
                arcRadius - circleRadius)
            },
            arc: arcRadius + ", " + arcRadius + " 0 0 0 -" +
            arcRadius + ", -" + arcRadius
        }
    }
```

Now, we need the `snap.svg` library to start drawing the required elements
into the `svg`. Add the package to the application; using the `meteor add
jeremy:snapsvg` command and also add `moment.js` packages to the application to
show the life event time.

Using `snap.svg` APIs, we are going to find the `svg` element inside our template and
then insert the `path` element with appropriate metrics to make the element look
like the one in the preceding image. The following code does the operation as we
mentioned earlier:

```
var eventSvg = Snap(this.find("svg")),
    eventPath = eventSvg
    .path(Snap.format("M{x},{y}v{lines.verticalLineTo}a{arc}h
{lines.horizontalLineTo}", pathData));
```

In the first line, we got the `snap.svg` instance of the `svg` element from the event
template. Then, we inserted a `path` element using `.path` API. The `.path` API takes
coordinates for the `d` attribute. The `snap.svg` library provides a cleaner way to
format the `d` attribute, and we can observe this in the preceding code.

The logic to draw the `path` line is pretty simple. Get to the center-bottom of the `svg`, draw a vertical line to the top, make an arc of the desired radius, and then based on the direction, we extend the `path` to the right or to the left end of the `svg`. We decide whether to go right or left, based on the index field from the collection document. We want to show the even ones to the right and the odd ones to the left. This will be a tree-like elegant interface to show the data.

Next, we are going to animate the `path` element. This is where we are going to explore the animation part of `snap.svg`. Append the following code to the `onRendered` callback:

```
var pathLength = eventPath.getTotalLength();
        eventPath
            .attr({
                id: "squiggle",
                fill: "none",
                strokeWidth: pathStrokeWidth,
                stroke: "#e35583",
                strokeMiterLimit: "10",
                strokeDasharray: "0 0",
                strokeDashOffset: pathLength
            });

        eventPath.attr({
            "stroke-dasharray": pathLength + " " + pathLength,
            "stroke-dashoffset": pathLength
        }).animate({
            "stroke-dashoffset": 0
        }, 2500, mina.bounce);
```

The path elements are usually animated using the `stroke-dashoffset` and `stroke-dasharray` properties. Every `path` element has length that can be calculated using the `.getTotalLength()` API. We first set `pathLength` to both those properties and then animate it to `0` using the `.animate` API provided by `snap.svg`. We can use the animation API as `Snap.animate` or `<snap instance of element>.animate`. The `.animate` API takes an object of properties with the result value (that is, the value at the end), duration, timing function, and callback. It is similar to jQuery's `.animate` API. Insert a life event using the method call and visit the application in the browser. You will be able to see the `path` element and its bounce animation.

While the preceding code animates the `path` element, we also want a circle to move along the same `path`. On the hover of the circle, we are going to show the content of the event in a silky tooltip. Now, add the circle-related code:

```
var circle = eventSvg.circle(svgLayout.width / 2,
svgLayout.height, circleRadius);
    circle.attr({
        fill: "#e35583",
        stroke: "#e35583",
        strokeWidth: 1,
        cursor: "pointer"
    })
    .mouseover(function() {
        this.find(".tooltip-content").classList.add("show");
    }.bind(this))
    .mouseout(function() {
        this.find(".tooltip-
        content").classList.remove("show");
    }.bind(this));
```

In the preceding code, we have created a circle using the `.circle` API and have provided the necessary attributes. Along with this, we have bound the `mouseover` and `mouseout` events to the circle. What the event handler does is, it adds or removes a class (show) to show the tooltip. The tooltip styles and animations are handled from CSS. The only remaining part is to animate the circle along the path and show the `event_time` and `title` under the `path` element with the `fadeIn` effect. Add the following circle animation code to do all that we mentioned earlier:

```
setTimeout(function() {
        Snap.animate(0, pathLength, function(value) {
            movePoint = eventPath.getPointAtLength(value);
            circle.attr({
                cx: movePoint.x,
                cy: movePoint.y
            });
        }, 2500, mina.bounce, function() {
            var x, y = 50, text;
            if(isEven === true) {
                x = (svgLayout.width/2 + 20);
            } else {
                x = 25;
```

```
        }
        text = eventSvg
            .text(x, y,
            [moment(data.event_time).calendar(),
            data.title])
            .attr({
                opacity: 0,
                fill: "#e35583"
            })
            .animate({opacity: 1}, 1000);
        text.select('tspan:nth-of-type(2)').attr({x: x,
        dy: 25, "font-style": "italics"});
    });
}, 0);
```

Here, we use `Snap.animate` to animate the circle along the path. However, how do we know the *x* and *y* coordinates for the circle to move? We have the `.getPointAtLength` API that will give *x* and *y* coordinates of the path at the given length. Given that we know the total length already, the `.animate` API will supply the length at every tick, and with that, we can calculate the *x* and *y* co-ordinates. The co-ordinates are then supplied to the circle's `cx` and `cy` properties.

One more thing to note is the callback. The `.animate` API takes a callback that will be called once the animation is completed. We've used this callback to show the text with the `fadeIn` animation, as we discussed earlier. Inside the callback, we create a `text` element using the `.text` API and supply the co-ordinates and the content. If we pass the content as an array of texts, each content item will be rendered inside an individual `tspan`. Finally, we apply some styles to position the `tspan`. Now, visit the browser and you will see the cool animation effects.

Now, create some more life events using the method call from the browser console and you will see the timeline growing. We have placed the tooltip content already in the template. The event handler that we had bound and the CSS will take care of the animation effect of the tooltip. Hover over the circle of each life event and you will see the content in a tooltip with a sleek animation. We can do the same animation with `snap.svg` if we create the tooltip in SVG. One can find all the `snap.svg` APIs at `http://snapsvg.io/docs/`.

Hope this beautiful animation helps you in your next SVG interface animation.

# Animation using d3.js

We looked at a d3.js example in the previous chapter. We now know how to handle MeteorJS reactivity. In this section, we are going to learn how to perform animations and transitions using d3.js.

What are we going to develop to demonstrate d3.js animations? We are going to develop a visualization that will show climate change in world locations. Cool, isn't it?

Our objective is to learn how to perform animations using d3.js. Most of the code samples will be pretty well known to you. To keep things simple, we are going to draw a world map with the help of the `DataMaps` package. The idea of the application is to show circles on the map at the specified latitude and longitude with simple animations.

Let us start by creating an application, for example, `AnimationPartyd3`. Remove all the hello template-related code, including the content of the `body` tag in `AnimationPartyd3.html`. Insert a `div` container where we will draw the, map as follows:

```
<div id="container"></div>
```

Add the following CSS to `AnimationPartyd3.css`:

```
body { margin: 0; padding: 0; background: rgba(0,0,0,0.9); }
svg { overflow: hidden; top: 50%; left: 50%; position: absolute;
  transform: translate(-50%, -50%); }
```

Let's draw the map. To start with, we need to install the `DataMaps` package that will install d3.js as a dependency. Install it with the following command:

**meteor add hyperborea:datamaps**

Now, we are all set to draw the map. `DataMaps` gives us simple APIs to draw a map. It also provides us APIs to draw arcs and bubbles on the map. However, we want to control the bubble animation and thus we will create bubbles with the custom logic. Add the following map creation code to the client block of `AnimationPartyd3.js`:

```
$(document).ready(function() {
  var map = new Datamap({
          scope: 'world',
          element: document.getElementById('container'),
          projection: 'mercator',
```

```
                height: 800,
                width: 1000,
                fills: {
                    defaultFill: '#1abc9c'
                }
            });
        });
```

Start the application from the terminal and visit it in the browser. You should see a world map. It is that simple!

We need a collection where we will insert the temperature and location details, which will indeed be displayed in the map. Prepend the following collection instantiation code to `AnimationPartyd3.js`:

```
/** lat, long, temperature in celcius **/
ClimateChange = new Mongo.Collection("climate_change");
```

We need a minimum and maximum range to determine the relative radius of the circles. Add the following code to the client block inside the ready function of `AnimationPartyd3.js`:

```
    var linearScale = d3.scale.linear().domain([-100,
    100]).range([5, 20]);
        function draw(data) {
            map.svg.selectAll(".temperature")
                .data(data)
                .enter()
                .append("circle")
                    .attr("id", function(d) {
                        return "circle"+d._id;
                    })
                    .attr("fill", "#e74c3c")
                    .attr("stroke", "#f1c40f")
                    .attr("stroke-width", 2)
                    .attr("cx", function(d) {
                        return map.projection([d.long, d.lat])[0];
                    })
                    .attr("cy", function(d) {
                        return map.projection([d.long, d.lat])[1];
                    })
                    .attr("r", function(d) {
                        return 2;
```

```
                })
                .transition()
                .duration(1000)
                .ease("bounce")
                    .attr("r", function(d) {
                        return linearScale(d.temperature);
                    });
        }
```

The `linearScale` variable is a scaling function that helps us calculate the relative size of the circle's radius with respect to the temperature. The `draw` function is where we draw the circles in the map. We need to translate the latitude and longitude to the map container dimensions. The map object has the projection API of the `d3.geo` layout that helps us to get the translated top and left dimensions in the map. We assign those values to `cx` and `cy` of the circle. Using the `linearScale` function, we convert the temperature to the circle radius and assign it in the last part of the draw function. We apply the radius with an animation.

We can perform animation or transition using d3.js with the help of the `.transition`, `.duration`, and `.ease` APIs. Before assigning values to any property, if we add the `.transition` API, it will be animated. An important point to note is to have an initial value to the property we are trying to animate. The `.duration` API is the time duration we want the animation to happen. The `.ease` API helps you to provide a timing function to the animation. For more easing effects, visit the repository at `https://github.com/d3/d3-ease`.

A few other important APIs that help us perform complex animations are `attrTween`, `arcTween`, and `interpolate`. One can also register callbacks to the start and end of the animation using the `.each` API. To learn more about the d3 animation APIs, visit `https://github.com/mbostock/d3/wiki/Transitions`.

Add the following code to observe the insertion, updates, and removal of documents in the collection via a cursor:

```
    ClimateChange.find().observe({
            added: function(doc) {
                draw([doc]);
            },
            changed: function(doc) {
                var circle = map.svg.select("#circle"+doc._id);
                circle
                    .transition()
                    .duration(1000)
                    .ease("bounce")
```

```
                       .attr("cx", function() {
                           return map.projection([doc.long,
                           doc.lat])[0];
                       })
                       .transition()
                       .duration(1000)
                       .ease("bounce")
                       .attr("cy", function() {
                           return map.projection([doc.long,
                           doc.lat])[1];
                       })
                       .transition()
                       .duration(1000)
                       .ease("bounce")
                       .attr("r", function() {
                           return linearScale(doc.temperature);
                       });

               },
               removed: function(doc) {
                   var circle = map.svg.select("#circle"+doc._id);
                   circle
                       .transition()
                       .duration(1000)
                       .ease("bounce")
                       .attr("r", function() {
                           return 0;
                       })
                       .call(function() {
                           this.remove();
                       });
               }
           });
```

On update, we select the appropriate circle and update the coordinates or radius with the animation as per the changed value in the document. The same is the case for removal. Insert data from the browser console using the following code:

```
ClimateChange.insert({lat: 11.867351, long: 77.871094,
   temperature: 30});
```

The circles animate on insertion. Similarly, on update of the latitude or longitude, the circles will transit to the new position. On removal, the radius reduces to 0 with transition.

Think of some sensors inserting the data from various locations to this application and imagine you are monitoring the temperature of various locations on a 60-inch screen. Pretty cool, right? Hope this example helps you to get started and play with d3.js animations.

# Animation using the Famo.us engine

Famo.us is a high-performance frontend library for creating interfaces and animations. At this point of time, the library is under development. Version 0.3.5 is the most supported version, but again deprecated. Version list is in alpha state and thus can't be put to production. We will experiment a little with version 0.3.

There are packages that help us use the Famo.us engine with MeteorJS. We will use the `mjn:famous` package that provides all the components of the Famo.us engine. There is a popular package called `famous-meteor-views` that provides famous components such as surfaces and modifiers in the form of templates to be used along with MeteorJS templates. However, because of the change in the Famo.us engine, `famous-meteor-views` has to undergo changes that are still under development.

Famo.us has different principles for creating and rendering interfaces. It doesn't give us control of DOM because DOM querying is costly. The Famo.us engine has the idea of creating a context and then adding surfaces to the context that can be modified spatially using modifiers. There are two ways of creating an application. One is to build the entire application using Famo.us; another is to create only the required interfaces with Famo.us. Famo.us is meant to create a full application and thus we will use MeteorJS' reactivity only to update Famo.us interfaces. We are going to look at one such simple example. Let's create a small application that synchronizes the spatial position of elements across the.

Create an application, such as `AnimationPartyFamous`. Remove all the `hello` template-related codes both in js and HTML files. Add the `mjn:famous` package and start the application. In the `.js` file, replace the existing code with the following piece of code:

```
Notes = new Mongo.Collection("notes");
if (Meteor.isClient) {
    $(document).ready(function() {
        var Context = famous.core.Engine.createContext(),
            Surface = famous.core.Surface,
            Draggable = famous.modifiers.Draggable,
            StateModifier = famous.modifiers.StateModifier,
```

```
        Transform = famous.core.Transform,
        notesModifiers = {}, notesSurfaces = {}, flag = false;

    Notes.find().observe({
        added: function(doc) {
            var surface = new Surface({
                size: [200, 200],
                content: '<div><div>'+doc.title+'</div>
                <div>'+doc.content+'</div></div>',
                id: doc.id,
                properties: {
                    backgroundColor: 'rgba(200, 200, 200,
                    0.5)',
                    textAlign: 'center',
                    cursor: 'drag'
                }
            });
            notesSurfaces[doc._id] = surface;

            var draggable = new Draggable();

            surface.pipe(draggable);

            var mod = new StateModifier({
                transform: Transform.translate(doc.x-10,
                doc.y-10, 0)
            });

            notesModifiers[doc._id] = mod;

            draggable.on("end", function() {
                var position = draggable.getPosition();
                flag = true;
                Notes.update({_id: doc._id}, {$set: {x:
                position[0], y: position[1]}});
            });

            Context.add(mod).add(draggable).add(surface);
            mod.setTransform(Transform.translate(doc.x, doc.y,
            0), {duration: 1000})
        },
        changed: function(doc, oldDoc) {
```

```
            if(!flag) {
                notesModifiers[doc._id].setTransform
                (Transform.translate((doc.x), (doc.y), 0),
                {duration: 1000});
            }
            flag = false;
        },
        removed: function(doc) {
            notesModifiers[doc._id].setTransform
            (Transform.scale(0, 0, 0), {duration: 1000});
            notesSurfaces[doc._id].render = function() {return
            null;}
            delete notesSurfaces[doc._id];
            delete notesModifiers[doc._id];
        }
    });
});
}
```

Go through the preceding code carefully. Most of it will be familiar. We have defined a collection called `notes`. We have inserted `title`, `content`, the `x` and `y` positions of each note. As we insert, the notes are displayed in the screen at the specified position with a smooth transition. The package we've installed provides us with the Famo.us components and we have to use it from the namespace `famous`. We created a context at the beginning and used it to attach all views.

In the `added` block of the `observe` method, we created a surface for each note and set its position using the modifier. Then, we made it draggable by adding a draggable instance to the surface, and registered an event to update the new position in the database on the drag end. Finally, we attached the modifier and the surface to the context.

On the `changed` block, we just updated the new position to the transition modifier. There is no way to get the modifier instance from the DOM; thus, we maintained a store object to hold all the modifiers and surfaces by the note `_id`. Based on the note `_id`, we get the appropriate modifier and set the new position.

On the `removed` block, we just removed the surface and deleted the instance from the store object.

Now, open the two browser instances and visit the application. Insert documents to the collection using `Notes.insert({title: "Note 1", content: "Note content 1", x: 100, y: 100});` from the browser console and start dragging the note. You will see the note moving to the new position in the other browser instance as well. Based on the new position, we update the `Transform.translate` modifier, and it syncs the position.

Famo.us is capable of many things. We just have to wait a while for it to be matured and adopted widely by the frontend community. I hope you enjoyed the simple but sleek sync application.

# Summary

Animations are adorable. They serve both the aesthetics and user experience. In this chapter, we learned how to use animation packages and libraries with MeteorJS. There won't be anything to stop us from doing cool animations hereafter in our MeteorJS applications.

Let us summarize all that we have learned in this chapter.

Animation in Blaze templates can be done using the unofficial `_uihooks` hook. We have learned to animate interfaces and also transit pages with the help of packages. There are plenty of packages that, again, rely on `_uihooks` to perform animation and transitions. Most of the packages use Velocity.js to animate properties.

We have learned to use `snap.svg` and its `.animate()` API to manipulate and animate SVG interfaces. Similarly, we have learned to use d3.js to perform animation using the `.transition`, `.duration`, and `.ease` APIs.

Finally, we have experimented with interface creation and animation using the Famo.us engine.

Using animations and transitions in the right places in the right quantities enhances the design and experience to a great extent. Get your hands dirty by experimenting more on all the techniques we have learned so far.

In the next chapter, we will learn about developing REST API in a MeteorJS application and we will also dive deep into the reactivity of MeteorJS.

<div align="right">

# 6

</div>

# Reactive Systems and REST-Based Systems

Reactive systems are the ones that can propagate changes based on the data flow. In this chapter, we are going to dive deep into MeteorJS' reactivity. It is necessary to understand Trackers in the first place, which is the reason for MeteorJS' reactivity. We will also learn about the Tracker library with simple examples.

Apart from reactivity, we can build REST-based systems using MeteorJS. It is common that data-based applications want to expose data to a third-party application via REST APIs. We will learn about what are the options available to implement RESTAPIs.

Modern applications can involve volumes of data. It is wise to use pagination to handle volumes of data. We will discuss the problems that we face and how to handle the problems while using pagination in MeteorJS applications.

In this chapter, we are going to cover the following topics:

- MeteorJS' reactivity
- REST-based systems using MeteorJS
- Handling volumes of data

## An overview of MeteorJS' reactivity

One of the important and prominent features of the MeteorJS framework is the reactivity. The framework provides reactivity out of the box. MeteorJS' reactivity has reduced development time considerably.

A notable thing is that developers need not wire the change to the views. The view layer is reactive enough to update the interfaces almost instantaneously. MeteorJS employs Blaze brilliantly to handle the instant updates to the views. In the previous chapter, we saw how Blaze breaks the templates to DOM element in order to update the elements that can undergo change when the data changes.

Apart from the view layer, MeteorJS provides us with the ability to auto-run methods when data changes. When there is a change in the cursor, we can run an arbitrary method to do some operation. This reactivity is provided to the extent that it can be used even with entities that are not reactive by default. For example, we can listen to arbitrary value changes and react to them by executing appropriate methods.

When used in a controlled state, MeteorJS' reactive data sources are great. However, make sure they are under control because using too much reactivity might not be performant.

# MeteorJS' reactivity

In any reactive systems, to achieve reactivity, we can follow different approaches. One can follow the poll and diff mechanism, but this is not cool because systems have to constantly poll and compute differences. This makes the client snappy at times. Another approach is eventing. Eventing is good, but can become messy very easily. Reactive programming is another approach, not much used, but a powerful solution for bigger applications.

MeteorJS follows a reactive programming approach. It is declarative. In the sense that, when a user is logging in it, changes the login button to display the user's name if the authentication succeeds. Why did MeteorJS choose reactive programming instead of other approaches? Ultimately, applications are going to display some sort of data via interfaces. Updating interfaces is going to be mandatory for most of the applications.

MeteorJS wants to offer an easy but powerful, less time-consuming way to write interfaces that can update themselves when data changes. Poll and diff could have been a suitable choice, but this is not great in terms of performance. Eventing will make the application maintenance costly. To keep things simple, a declarative way of updating an interface is efficient. Angular.js became very popular because of its declarative approach for updating interfaces. MeteorJS also chose the same paradigm that led it to choose reactive programming as a solution.

When we look at reactivity closely, there are two participants in a reactive environment. One is the entity that triggers the change. MeteorJS has a set of components that act as reactive datasources that can trigger change. They are database collection cursors, session variables, and the `Meteor.user`, `Meteor.userId`, `Meteor.loggingIn`, `Meteor.status`, and `ready` methods of a subscription handle.

Apart from these built-in reactive datasources, we can create custom reactive data sources with the help of the `reactive-dict` and `reactive-var` packages. We have used the `reactive-var` package in the first two chapters to create custom reactive data sources to propagate changes between two templates.

The other participant in a reactive environment is a reactive consumer, which could be any arbitrary method or interface that reacts to the change triggered by the reactive data sources. Templates are typical examples of this case. We can write custom methods also that can execute when a change is triggered. We have used `Tracker.autorun` in the past to make a few custom methods listen to reactive data sources.

All these reactivities raise the curiosity of "how does it happen?". MeteorJS documentation answers the curiosity pretty straightforwardly. The Tracker is responsible for MeteorJS' reactivity.

# Tracker

We know that MeteorJS is built using a set of packages. There is one specific package that enables all these reactivities and it is nothing but the **Tracker** library. Visit the package (`https://github.com/meteor/meteor/tree/devel/packages/tracker`) for more details.

Tracker, previously called Deps, is a reactive library that is less than 1 KB. MeteorJS uses it extensively to keep components reactive. Blaze is built to be Tracker-aware so that it can react to the changes. Similarly, MiniMongo, `reactive-var`, `reactive-dict`, and reactive data sources that we have seen are all Tracker-aware. Basically, MeteorJS uses Tracker wherever it makes sense.

We have seen the reactivity between data collection cursors and templates in the previous chapters. MeteorJS' boilerplate code has a session counter example that portrays reactivity between the session and templates that we saw in the last chapter while learning animation hooks. If you have played with login interfaces, you will be very much aware of the `Meteor.loggingIn` reactive property.

To demonstrate the power of Tracker, let us create a small example. Create a new MeteorJS application, such as `CustomReactivity`, and replace the whole of `CustomReactivity.js` with the following code:

```
var trackerDeps = new Tracker.Dependency;
function counterVal() {
    trackerDeps.depend();
    return Math.floor(Math.random() * (100 - 10 + 1)) + 10;
}
function changeCounterVal() {
    trackerDeps.changed();
}
function getRandomInt(min, max) {
    return ;
}
if(Meteor.isClient) {
    Template.hello.helpers({
        counter: function() {
            return counterVal();
        }
    });
    Template.hello.events({click button": changeCounterVal});
}
```

The function or class instantiation, `newTracker.Dependency`, gives a Tracker dependency instance. With that instance, we can make any method to be a reactive trigger and reactive data source. The `changeCounterVal` method acts as a reactive trigger. The `counterVal` method that is a reactive data source will be run whenever there is change triggered at the Tracker dependency instance. The `counterVal` method acts as a reactive data source and thus the template updates itself whenever `changeCounterVal` triggers a change. In the template events helper, we have registered a click event on the button that will trigger a change by calling `changeCounterVal`. When a change is triggered on the Tracker dependency instance, the `counterVal` method will run and the template that is the consumer of the `counterVal` method updates itself with the new random value.

Simple, but a very powerful package! Let's explore the behind-the-scene logic of Tracker for a better understanding. The reactivity of Tracker-aware libraries such as Blaze is due to the `autorun` method of the Tracker. For any method to be reactive to changes, we have to pass it to `Tracker.autorun`. Blaze internally does the same because of this, it reactively changes the DOM as the data changes.

Let's see the custom reactivity ourselves. Add the following piece of code to the `CustomReactivity.js` file:

```
Tracker.autorun(function() {
    console.log(counterVal());
});
```

Keep the browser console open and click on the button in the `hello` template. You will see the log of the `counterVal` method's return value.

When we call `autorun`, a computation object is created. Whenever an `autorun` is executing code, the global variable `Tracker.currentComputation` is set to the computation that goes with that `autorun`. Any Tracker dependent method will look if it is inside `autorun`; in our case, it is `counterVal`. If it is inside, then it will hold a reference to the computation object created and also arrange to call a method on it when the change is triggered. This tells the computation created by `autorun` to rerun.

Precisely, `Tracker.currentComputation` acts as a mediator point between the data and the function that should run when the data changes.

What if we want to stop an `autorun`? The function we pass into the `autorun` as an argument gets the computation object as a parameter. On any given condition, we can stop the `autorun` by calling the `stop` method on the computation object. When we call stop, the computation is cleaned up. Out of `autorun`, if we want to handle the computation, we can assign the return value of `Tracker.autorun` to a variable that is a computation object, and use it to call `stop`. It is good to know that if there is an exception at the initial run, the computation will be stopped.

Is there a way to detect the first computation? Yes. The computation object that we get as an argument has the `firstRun` property with which we can ensure whether it is a first run.

There is an important convention to follow inside the `autorun`. If we create an object inside the `autorun`, we have to destroy it inside the autorun. The reason being, autorun recreates the object every time it runs. So, there is a possibility of memory leak. We have to be careful while creating objects inside autorun and not forget to destroy it once it is not needed anymore.

Calling `stop` on the computation object inside the autorun may not stop the autorun sometimes. Take a look at the following example:

```
var run = Tracker.autorun(function() {
    if(Session.equals("counter", 2)) {
run.stop();
    }
});
```

The `run` variable is a computation object and we are trying to stop it inside the `autorun`. If `autorun` hasn't returned, and `stop` is called before for some reason, then `autorun` is not going to stop. To ensure that it has stopped, we can use the computation object argument passed to our function from `autorun`, as follows:

```
var run = Tracker.autorun(function(comp) {
    if(Session.equals("counter", 2)) {
comp.stop();
    }
});
```

We can nest autoruns. Nesting autoruns is interesting and allows us to see an example of how it works. To demonstrate nesting, we will use the `reactive-dict` package. We can use session or `reactive-var` as well. Because we haven't used `reactive-dict` anywhere yet, let's give it a try. The `reactive-dict` package is very similar to session, but doesn't persist over hot code pushes. Add the `reactive-dict` package to the `CustomReactivity` application.

Add the following code to `CustomReactivity.js`:

```
varmarket=newReactiveDict();
market.set("sale","high");
market.set("demand","less");

varsaleRun=Tracker.autorun(function(){
console.log(market.get("sale"));
vardemandRun=Tracker.autorun(function(){
console.log(market.get("demand"));
});
});
```

We have created a `ReactiveDict` instance. We are setting `sale` and `demand` to the dictionary. Then, we are nesting autoruns where the outer one prints the `sale` value and the inner one prints the `demand` value. Try changing the `demand` value and you will observe that only the inner `autorun` runs and not the outer one. However, if you change the `sale` value, both the autoruns will run. This is the behavior we have to understand while creating nested autoruns. Every time the outer `autorun` is run, a new computation object is created for the inner `autorun`. However, the old one is cleared automatically.

We can rerun the `autorun` function using the `invalidate()` method on the computation object. The object also has three properties, `stopped`, `invalidated`, and `firstRun`, which can be used in various situations. When the `changed` method of the Tracker dependency instance is called, it calls the `invalidate` method. You can learn even more from the source available at `https://github.com/meteor/meteor/blob/devel/packages/tracker/tracker.js`.

When we have a set of updates to reactive datasources, they are done in batches. A simple example is as follows:

```
var data = new ReactiveDict();
data.set("favoriteFood", "chicken");
Tracker.autorun(function () {
  console.log(data.get("favoriteFood"));
});
console.log("start update");
data.set("favoriteFood", "waffles");
data.set("favoriteFood", "pie");
console.log("finish update");
```

If you execute the preceding code, the `finish update` log statement will be printed before `data.set`. This is because of the batch update the Tracker follows. If you want it to happen in order, use `Tracker.flush()` before the last log statement. This has a different effect. Every `.set` will call `autorun` method. Incase of database updates, this might have adverse effects.

# Optimizations in autoruns

Retrieving data inside `autorun` is the same as subscribing to change notifications for that data. We should try to get only the needed data inside `autorun` to use the memory efficiently. Whenever possible, try not to use the `.get` method of the reactive datasource:

```
Tracker.autorun(function() {
    if(Session.get("counter") === 2) {
        //do something
    }
});
```

The preceding code is inefficient. It uses a `get` on reactive data source session. Instead, use the `.equals` method that will be efficient.

Inside `autorun`, we should use `fetch` on collections instead of `find`, which returns just a cursor to the documents. The `fetch` returns the documents and there is a dependency set to all the documents returned by fetch.

In terms of using queries (`find`) inside autorun, it is good to use the queries set with filter and field to reduce the data fetched on every run. However, while fetching a count, we should use `.count()` instead of `.fetch().length`. When we use `.fetch()`, it fetches all the documents and the dependency is set on all the documents unnecessarily. Instead, if we use `.count()`, we set the dependency only on the count and that is efficient.

This is all we need to know about Tracker. The information is much bigger than the library itself, isn't it? With great power comes great responsibility. Use Trackers well and watch out for performance. To read more, visit `http://manual.meteor.com/#deps-howdepsworks`.

# REST-based systems

**Representation State Transfer** (**REST**) is a software architecture style for the Web and Web-based applications. REST has become very popular and there are hundreds of applications powered by REST APIs. REST APIs enable us to build distributed and multitier systems efficiently. A typical single page, JavaScript-driven application are mostly powered by REST APIs. Whenever we have a backend with data storage, many clients want it to be REST-based so that the backend layer can be decoupled from the frontend layer. In similar terms, many developers want to try whether MeteorJS can also act as a REST-based system. Yes, it is possible.

MeteorJS is an apt solution for applications that need to handle data reactively in both the backend and frontend. If we want MeteorJS to be used just to serve data, this isn't a good choice. Instead, we can look out for other better options. Still, for the sake of learning, let's give it a try and see all the caveats it has.

The following are the three important reasons to implement REST APIs in MeteorJS applications:

- To serve data to external application
- To transfer data that DDP cannot handle (we used HTTP transfer in case of image upload in *Chapter 4*, *Integrating Your Favorite Frameworks*)
- For applications that don't want to create or maintain WebSockets

There are many ways to implement REST APIs in a MeteorJS application. Considering the router package that we are very much familiar with, `iron-router` supports REST APIs out of the box. Also, there are other solutions that enable us to write REST APIs in MeteorJS applications.

We are going to look into `iron-router` and `restivus` in this chapter to get familiar with implementing the REST-based system using MeteorJS.

# REST with iron-router

If you have watched the routes created using `iron-router`, you will know that we can specify the target environment for each route. The target environment can be the server or the client:

```
Router.route('/files/:filename', function () {
  //action
}, {where: 'server'});
```

The third argument is where we specify the environment using the `where` key. If we specify it to be `client`, the package expects those routes to be called by the MeteorJS application client. If we specify `server`, then the route can be accessed from any other application. If you are using `iron-router` and want to expose data to a third-party application in the REST style, then you have the solution right in hand. You just need to specify the server routes as you wish and serve the data.

Let us create an application to see how it works. Create a MeteorJS application, such as `APIHero`. Add `iron-router` to the application using the `meteor add iron:router` command. Now, what are we going to expose?

Let's have a collection to store books and authors' name. We have to create a collection and then create routes to add to the collection, read from the collection, update the documents in the collection, and delete items from the collection. Create the `author_details` collection by adding the following code to `APIHero.js`:

```
AuthorDetails = new Mongo.Collection("author_details");
```

Iron-router provides two syntaxes to specify server-side routes. The first syntax doesn't provide a separation about the type of request whether it is post, get, and so on. We have to deduce it from the request. The syntax is as follows:

```
Router.route('/book-by-author/:name', function () {
  if(this.request.method === "GET") {
  //Do something
  }
}, {where: 'server'});
```

The request object has all the required information. The other syntax gives methods to handle the request type, as follows:

```
Router.route('/author', {where: 'server'})
  .get(function () {
    this.response.end('get request\n');
  })
  .post(function () {
    this.response.end('post request\n');
  });
```

The package filters the request by its type and calls the appropriate method when we use this syntax.

Let's create a route to add documents to the `AuthorDetails` collection. Create a file, `router.js`, and add the following route:

```
Router.onBeforeAction(function() {
  if (this.request.method == 'OPTIONS') {
    this.response.setHeader("Access-Control-Allow-Origin", "*");
    this.response.setHeader( 'Access-Control-Allow-Headers',
'Content-Type' );
    }
    this.next();
});


Router.route("/author-details", {where: "server"})
  .post(function() {
    var data = this.request.body;
    AuthorDetails.insert(data, function(err, res) {
      if(!err) {
        this.response.statusCode = 201;
        this.response.setHeader("Access-Control-Allow-Origin",
"*");
        this.response.setHeader("Content-Type",
"application/json");
        this.response.setHeader('Access-Control-Allow-Headers',
'Origin, X-Requested-With, Content-Type, Accept');
          this.response.setHeader('Access-Control-Allow-Methods',
'POST, PUT, GET, DELETE');
        this.response.end(JSON.stringify({status: "success",
response: {id: res}}));
      }
    }.bind(this));
  });
```

`iron-router` has less abstraction in terms of request and response handling. So, we have to manually set the header to enable **Cross-Origin Resource Sharing (CORS)**. Also, it is very important to handle the request method OPTIONS in the `onBeforeAction` hook provided by the package as given in the preceding code. Do not forget to call `this.next()` in the `onBeforeAction` hook. How are we going to test our API?

Ideally, you can create a small application and run a web server from where you can fire a POST request with the appropriate data to the route that we have created. Let's take a shortcut. If you are good in cURL, write a cURL request and check whether the API works. If not, run the following code in the browser console of some other application tab:

```
(function request() {
  var xmlhttp  = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
              console.log(xmlhttp);
            };
            xmlhttp.open("POST", "http://localhost:3000/author-
details", true);
            xmlhttp.setRequestHeader("Content-Type",
"application/json")
            xmlhttp.send(JSON.stringify({name: "Paulo Cohelo",
book: " The Alchemist"}));
  })();
```

The snippet makes a POST request to our API with the author details data and the API inserts it to the database. Check it in the database by visiting the mongo console.

Let's create another API to get all the author details. In the same route, add a get handler to respond with the appropriate data, as follows:

```
.get(function() {
  this.response.statusCode = 200;
  this.response.setHeader("Access-Control-Allow-Origin", "*");
  this.response.setHeader("Content-Type", "application/json");
  this.response.setHeader('Access-Control-Allow-Headers',
'Origin, X-Requested-With, Content-Type, Accept');
    this.response.setHeader('Access-Control-Allow-Methods',
'POST, PUT, GET, DELETE');
  this.response.end(JSON.stringify({status: "success", response:
AuthorDetails.find({}).fetch()}));
  });
```

Visit the same API in the browser; you will see all the data inserted using the POST request.

# API guidelines

There are many things, as a developer, we should keep in mind while writing REST APIs. Let's see what they are:

- `StatusCode`: It is good practice to set `statusCode` to the response and it is important to set the appropriate ones. In the case of create, setting the `statusCode` to `201` makes more sense than `200`, which merely says successful. In case of failures, it is better to set the code that portrays the reason explicitly. If the request parameters are non-parsable, we can return `422`, which means unprocessable entity. The complete set of status codes and their meaning can be found at `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

- **JSON request and response**: It is good to use JSON for both request and response. It is a widely accepted format of data representation, and most of the platforms support JSON.

- **CORS**: In the preceding snippets, we set Access-Control-Allow-Origin to be `*`, which means anyone can access these APIs from any domain. Most of the time, this is not going to be the case. Unless we are writing APIs accessible to anyone, it is good to specify the names of the domains that we would like to give access to our APIs. Also, we can set what kind of headers and methods are allowed. This gives even more granular control on what to allow and what not to allow.

- **Handling request data**: Never trust user inputs. Every developer must keep this in mind to avoid security mishaps. We have to validate every parameter that we are going to use for any serious operation. In our author-details API example, we should validate the name and book for valid strings and duplicate cases. Blindly inserting will have undesirable consequences.

- **Handling response data**: This provides what is absolutely necessary in the response. If required, allow the request to specify what fields are required in the response. While querying the collection, fetch only the specified fields and set them in the response. This way, we don't have to expose everything. Also, it is necessary to validate the fields requested. For example, if we are exposing `users` collections, exposing every field is madness. So, we should validate the requested fields if present in the request and, if they are not present, provide only the fields that are essential.

- **Volumes of data**: Over a period of time, the data volume becomes high. APIs should not be sending all the data at one shot. It is always better to implement pagination via `sort`, `limit`, and `skip` properties, or by using some packages. Set a default number of results in the result set and respond to the request if no pagination limit is specified in the request. Also, set a max limit so that the requestor doesn't ask for too much data in one shot.

- **Error response**: If it is a user interface-based application, we could display all errors with helping cues. We are delivering data in terms of REST APIs and thus we should be very elaborate in explaining the error responses. It is good practice to keep the error response structure as consistent as possible so that the consumer applications can handle errors without much pain. The following is a good example of a consistent error response:

```
{errors: [{
  errorCode: 2003,
  errorMessage: "Requested book not found"
}]}
```

It is good to have errors as an array so that we can accommodate multiple errors and the response is consistent enough. The `errorCode` property is a custom numbering to standardize the error messages.

- **Handling Route error**: This is nothing but handling misspelled or wrongly formed routes. We should not leave the consumers blank if they are requesting an unknown route. Handle them by sending `404` responses to such kinds of routes.

- **Versioning**: It is quite common to make changes to the APIs. However, how are we going to maintain the change in a way that doesn't break the existing consumer applications? Versioning comes to the rescue. On every change in the API response or request format, make it mandatory to update the version and notify the user about it. This will save the existing consumers of your API.

- **Authentication and authorization**: If you are exposing data that is restricted to a user session, make sure you read the headers for the required token or user ID. Proper authentication must be provided while exposing APIs of such sort. Similarly, authorization rules must be specified to allow access to data.

- **Testing APIs**: It is very important to test the APIs before releasing. Automate them by writing simple shell scripts that can cURL your APIs with dummy data. There are other tools as well out there that can automate the testing. For developer testing, tools such as Postman do a wonderful job. Make sure the APIs are stable before releasing.

With `iron-router`, maintaining most of these guidelines is a cumbersome job as it provides low-level abstraction to create APIs. We have to take care of them manually.

# REST with restivus

`Restivus` provides high-level abstraction to handle REST-related data. The package is inspired by `RestStop2` and `Collection API` packages. Unlike `iron-router`, `restivus` handles CORS and other obvious header settings by itself. `Restivus` provides CRUD operation via APIs, out of the box. Specify the collection on which CRUD operations must be exposed, `restivus` will take care of the rest. `Restivus` offers in-built authentication, both login and logout, and also provides control over authorization via roles. Apart from these features, it also has versioning in-built with the package. `restivus` was initially built on the top of `iron-router`, but later moved to `simple:json-routes`.

Let us see how to create APIs using `restivus` with an example. Create a MeteorJS application, for example, `APISuperHero`. Add the `restivus` package to the application using the `meteoraddnimble:restivus` command. We will start by creating users and then exposing users followed by creating CRUD for author-details with authentication.

Create the `server` directory to keep the APIs inside the server. Create `api.js` and put it inside the `server` directory, and add the following code to instantiate the `restivus` object:

```
var API = new Restivus({
  useDefaultAuth: true,
  prettyJson: true,
  apiPath: '/'
});
```

We can specify many more options while instantiating `restivus`. An example with a complete set of options is, as follows:

```
new Restivus({
    apiPath: 'my-api/',
    auth: {
      token: 'auth.apiKey',
      user: function () {
        return {
          userId: this.request.headers['X-User-Id'],
```

```
            token: this.request.headers['X-Auth-Token']
          };
        }
      },
      defaultHeaders: {
  'Content-Type': 'application/json'
      },
      onLoggedIn: function () {
        console.log(this.user.username + ' (' + this.userId + ')
  logged in');
      },
      onLoggedOut: function () {
        console.log(this.user.username + ' (' + this.userId + ')
  logged out');
      },
      prettyJson: true,
      useDefaultAuth: true,
      version: 'v1'
    });
```

In the `apiPath` option, if specified, the REST APIs will be namespaced with the `apiPath` value. The default `apiPath` value is `api` and, if we want to get rid of the default and have no `apiPath` at all, we have to explicitly specify `apiPath` to be "/". If the `apiPath` property is `rest-api`, our URLs will be `http://<domain.name>/rest-api/<entity>`. Similarly, when specified, versioning can be used in the URLs. If we are going to use the authentication, then we have to specify the `auth` object, which is nothing but specifying how to identify the user object and what headers to look for in the request. We will look at it in detail shortly.

To add CRUD support to the collections, we just have to add the collection instance to the `restivus` instance using the `addColleciton` method as follows:

```
  API.addCollection(Meteor.users);
```

We can also provide an optional parameter, with which we can specify what all methods or endpoints to expose, which methods need authentication, which ones need authorization, and so on.

When APIs have access to the `Meteor.users` collection, we have to be very cautious. We should not allow `PUT` requests and `DELETE` without authentication, and `getAll` must not be provided because we don't want to expose all the users data. The `POST` requests can be allowed if, and only if, we want to expose APIs to create new documents in the collection. In our case, we want to allow users to signup. So, we are going to take all these into consideration and add our first API for accessing the `Meteor.users` collection:

```
Api.addCollection(Meteor.users, {
    excludedEndpoints: ['getAll'],
    routeOptions: {
      authRequired: true
    },
    endpoints: {
      post: {
        authRequired: false
      }
    }
});
```

We are excluding the `getAll` request for this collection because we don't want any user to see all the users in the system. The `routeOptions` option is where we specify if this CRUD operation requires authentication. We can add options to specify if authentication is required at an endpoint or method level. Here, we are specifying that `POST` requests for this collection don't require authentication.

> At each endpoint, we can specify custom actions, if we want a different set of operations to perform than the usual.

`Restivus` uses the `accounts` package for user creation and thus we have to add the `accounts-password` package for the user creation to work. Once added, we are all set to create a new user. Shall we create a `POST` request from a browser console and see if the user is created in the `APISuperHero` application? We cannot send an arbitrary request payload to create a user. MeteorJS' accounts package accepts certain values only. `Restivus` uses the `accounts-password` package for user creation and thus we also have to send the parameters expected by the `accounts-password` package. `Restivus` uses the `Accouts.createUser` methods to create a user using a username/e-mail and password. We have to send the following payload in the `POST` request:

```
{
"email": "jack@mail.com",
"password": "password",
```

```
   "profile": {
   "firstName": "Jack",
   "lastName": "Rose"
     }
   }
```

In the previous example, we have used `XMLHttpRequest` to create and send request. We will use the same this time as well. Run the following code in a browser console:

```
      var xhr = new XMLHttpRequest();
       xhr.onreadystatechange = function() {
           if(xhr.readyState === 4) {
             console.log(xhr.responseText);
           }
       };
       xhr.open("POST", "http://localhost:3000/users", true);
       xhr.setRequestHeader("Content-Type", "application/json");
       xhr.send(JSON.stringify({
   "email": "xxx@xyz.com",
   "password": "123456",
   "profile": {
   "firstName": "first name",
   "lastName": "last name"
             }
       }));
```

You will get a response with the profile details and the user ID, as follows:

```
   {
   "status": "success",
   "data": {
   "_id": "vmvKJgeridXsaZKvs",
   "profile": {
   "firstName": "Jebin",
   "lastName": "dev"
     }
   }
   }
```

Visit the application's mongo collection and check whether a user is created using the `db.users.find()` command. You will find the created user document. Let's try a get request to get the profile details of the same user we have created above, by firing a `GET` request. Visit `http://localhost:3000/users/vmvKJgeridXsaZKvs` in a new browser tab and you will get the following access denied error message:

```
{
"status": "error",
"message": "You must be logged in to do this."
}
```

We have specified in the configuration that any API on the `Meteor.users` collection needs the authentication. So we are not allowed to fetch the user object unless we authenticate ourselves. To authenticate, we have to login to the application, and `restivus` already provides us APIs to login and logout. On login, we will get user ID and token that we have to pass along with the headers on each request that requires authentication. Before that, we have to add the auth configuration to `restivus` instantiation. Change the `restivus` instantiation in `api.js` to look like the following code:

```
var API = new Restivus({
  auth: {
      token: "services.resume.loginTokens.hashedToken",
      user: function () {
        return {
          userId: this.request.headers['x-user-id'],
          token: Accounts._hashLoginToken(this.request.headers['x-
auth-token'])
        };
      }
    },
  useDefaultAuth: true,
  prettyJson: true,
  apiPath: "/"
});
```

The `auth` object takes two properties: `token` and `user`. The token property is the key to identify the hashed token in users' collection document and, in our case, it must be `services.resume.loginTokens.hashedToken`. Go to the database and find the user you are trying to login with and traverse in the order mentioned in the preceding hierarchy. You will find the hashed token.

The user property takes a function in which we have to return either the user object or the user ID and the hashed token. We have to return the hashed token of the login token we got as a result of the login request. You can see this in the code. If we are using custom authentication, this is the point where we have to customize to authenticate the user. We are not customizing the authentication and thus we will just send the `user-id` and `login-token` in the request header for authenticating our requests.

We will call login and get the `user-id` and `login-token` as follows:

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if(xhr.readyState === 4) {
        console.log(xhr.responseText);
    }
};
xhr.open("POST", "http://localhost:3000/login", true);
xhr.setRequestHeader("Content-Type", "application/json");
xhr.send(JSON.stringify({
"email": "xxx@xyz.com",
"password": "123456"
}));
```

The response received is as follows:

```
{
"status": "success",
"data": {
"authToken": "Fp9vKkPudMffAMEwWCj_8lav6zcahTnLliHbeH0cJaf",
    "userId": "pa2BNuYyXv8pAGrzN"
  }
}
```

We got the `login-token` and `user-id`, which allows us to set in the request that had failed before because of authentication error, as follows:

```
var xhr = new XMLHttpRequest();
 xhr.onreadystatechange = function() {
     if(xhr.readyState === 4) {
         console.log(xhr.responseText);
     }
 };
```

```
          xhr.open("GET",
"http://localhost:3000/users/pa2BNuYyXv8pAGrzN", true);
          xhr.setRequestHeader("Content-Type", "application/json");
          xhr.setRequestHeader("X-Auth-Token",
"Fp9vKkPudMffAMEwWCj_8lav6zcahTnLliHbeH0cJaf");
          xhr.setRequestHeader("X-User-Id", "pa2BNuYyXv8pAGrzN");
          xhr.send();
```

Fire the request and you will get the requested user. In the preceding snippet, we are setting the headers with specific keys. `Restivus` has been set to allow only a set of headers in the request. They are `Origin`, `X-Requested-With`, `Content-Type`, `Accept`, `X-User-Id`, and `X-Auth-Token`. So, we have to follow this convention while setting the header in our requests.

Provided that you are logged in, you now have access to `GET` any user object if you know the user ID. What if we want to restrict that and allow you to fetch only the logged in user's object and no one else's user object? To check this, we need to override the endpoint action. Let's do this by changing the API configuration to look as follows:

```
API.addCollection(Meteor.users, {
  excludedEndpoints: ['getAll'],
  routeOptions: {
    authRequired: true
  },
  endpoints: {
    get: {
      authRequired: true,
      action: function() {
        if (this.request.headers['x-user-id'] ===
this.urlParams.id) {
          var entity;
          entity = Meteor.users.findOne(this.urlParams.id, {
            fields: {
              profile: 1
            }
          });
          if (entity) {
            return {
              status: 'success',
              data: entity
            };
          } else {
```

```
                return {
                  statusCode: 404,
                  body: {
                    status: 'fail',
                    message: 'User not found'
                  }
                };
              }
          } else {
            return {
              statusCode: 401,
              body: {
                status: 'Unauthorized',
                message: 'You are not allowed to do this operation'
              }
            };
          }
        }
      },
    post: {
      authRequired: false
    }
  }
});
```

We have added the `get` endpoint to override the default behavior and we have also specified the action property that must be a function where we can do our operation of checking the user and returning the response appropriately. Create one more user and try to get the new user using the `get` request we fired before with the previous login token and user ID; it will throw `401` responses, as follows:

```
{
"status": "Unauthorized",
"message": "You are not allowed to do this operation"
}
```

We got a response as expected and as configured in the preceding code.

The package takes away lots of pain points by configuring API routes for each collection added. One important thing to keep in mind is, `restivus` treats the `Meteor.users` collection a little different from other collections. It restricts its scope to profile entry only, in the case of GET, PUT, and GETALL. However, for other collections, there is no such restriction and it makes sense too.

Finally, what if we want to define some custom routes? The package also provides the ability to do this by exposing the `addRoute` method on the instantiated object, as follows:

```
API.addRoute('app-users/:id', {authRequired: false}, {
  get: function() {
    var entity;
    entity = Meteor.users.findOne(this.urlParams.id, {
      fields: {
        profile: 1
      }
    });
    if (entity) {
      return {
        status: 'success',
        data: entity
      };
    } else {
      return {
        statusCode: 404,
        body: {
          status: 'fail',
          message: 'User not found'
        }
      };
    }
  }
});
```

In the preceding code, we added a custom API to get the user object. The `addRoute` method takes the path, auth flag object, and endpoints object. Whatever endpoints we define inside can be called on this route. Visit `http://localhost:3000/app-users/<user-id>` in the browser and you will find it returning the user object. Simple and pretty, isn't it?

There are other packages as well to help us set up REST APIs in MeteorJS applications. `CollectionAPI`, `Picker`, `HTTP.Publish`, and `http-methods` are some other choices we have. Explore them and choose according to your need.

In this next section, we are going to discuss handling data in volumes. This next section doesn't have any connection with REST implementation. Do not confuse yourself by relating things. This section is meant for the reader to understand how to handle volumes of data with the help of pagination.

# Handling volumes of data

One of the important principles of MeteorJS is, data is everywhere. The framework refers and stores data in the MongoDB at the server; at the client, there is MiniMongo in which the data is stored more or less like in the MongoDB and is referred (queried) with an unified approach as in the server. What if to the application serve's volumes of data? Let us discuss how we can efficiently handle volumes of data in this section.

With `autopublish`, the server sends all the available data to the client. Imagine the server has 1 GB of data and it sends all of it to the client. The Web client being a browser, may or may not handle based on the system's capability. If you are a Web developer and have used lower-end machines to debug using Chrome's debugger, you might have encountered application crashes. Similarly, if we opt to persist logs on navigation in chrome, we will encounter a crash sooner or later. When the browser can't allot enough memory for application because of memory shortage, there is a high possibility that the application will crash. In the case of a MeteorJS applications, if we allow more data to reside on the client, there are chances of your application to crash. What can we do to avoid this?

Send only what is absolutely necessary to the client from the server and request only what is needed from the client. When we have to build a data analytics application, we have to show an appropriate result to the user. This doesn't mean we have to show every possible value at one shot. We can design the interfaces in a way that we can split the data into chunks and request only the interested chunk from the server. Splitting the data and loading those chunks means nothing but paging the data. Upon pagination, we can apply one more round of optimization where we can specify only the required fields rather than subscribing to the entire document. In this section, we are going to build an application, inject random data, and then study how to apply pagination and other optimizations.

Create an application, such as `PageData`. Remove the `autopublish` and `insecure` packages from the application. We need some amount of data to apply pagination and see how things work. Let's write a small script that can import data from the CSV file. You can download the CSV file from the associated code folder that is available for downloading from Packt's website. To import, we can use two approaches. One is using `mongoimport` that must be downloaded and then used with the `mongoimport` command from the terminal. Another approach is using NPM modules in our application, and reading the file and inserting the records to the collection. We are going to follow the second approach because we can learn how to use NPM packages in MeteorJS applications.

To use NPM packages, we should require them in our application. If the package is a core package (`fs`, `path`), then just requiring is enough. However, if it is not a core package, then we must ask MeteorJS to install the package for us. Also, there is a package that does the job for us. Install the `meteorhacks:npm` package that enables us to use NPM packages in the MeteorJS application. It needs a `package.json` file in which we have to specify the packages we want to use and its version. It is mandatory to restart the application for the packages to be available. To use the installed package, we should use the `Meteor.npmRequire` method.

In our case, we need a CSV parser for which we will use the `csv-parse` NPM package. It also needs the `stream-transformer` package for reading the file as stream. Create a `package.json` file at the application root and add the following JSON code:

```
{
"csv-parse": "1.0.0",
"stream-transform": "0.1.0"
}
```

Start the application and in the terminal, you will see MeteorJS installing the mentioned packages for us. Now, we have to write the logic to create a collection and insert mock data. Remove the `hello` template-related code from all files and add the following piece of code inside `Meteor.startup` in the server block at `PageData.js`:

```
        EmployeeDetails = new Mongo.Collection("employee_details");
         var fs = Npm.require('fs');
         var path = Npm.require('path');
         var parse = Meteor.npmRequire('csv-parse');
         var transform = Meteor.npmRequire('stream-transform');

        function loadData() {
            var basepath = path.resolve('.').split('.meteor')[0];
            var parser = parse({
                delimiter: ','
            });
            var input = fs.createReadStream(basepath +
'Emp_details-MOCK_DATA.csv');
            var transformer =
transform(Meteor.bindEnvironment(function(row, index)
{
                EmployeeDetails.insert({
                    name: row[0],
                    email: row[1],
                    IBAN: row[2],
```

```
                        address: row[3],
                        city: row[4],
                        country: row[5],
                        dob: row[6],
                        joining_date: row[7],
                        gender: row[8],
                        age: row[9],
                        emp_id: row[10]
                    });
                }, function() {})
                , {
                    parallel: 10
                });
                input.pipe(parser).pipe(transformer);
            }
            loadData();
```

We have created a collection at the beginning of the code snippet and then we got
the reference of the NPM packages that we are going to use. We have to use `NPM.`
`require` for core packages and `Meteor.npmRequire` for noncore packages. Using
the `path` package, we will resolve the file path and then create a stream of the file.
A parser is created, which is piped to the file stream and then a transformer. A
transformer is called with each row of data. We insert each row into the collection
we have created. Go to the application's database and check whether it has inserted
1000 documents to the `employee_details` collection. Comment out the `loadData`
function call once the insertion is completed. It will keep inserting data on every
application startup.

Now, we have some data to play around with. Let's try displaying the data in pages.
Remove the data insertion script to keep things clean. Move the collection instance
creation code to the top of the file so that it is accessible both in the client and the
server. Let's publish the collection from the server block, as follows:

```
if (Meteor.isServer) {
    Meteor.publish("employee_details", function(query, projection)
{
        query = query || {};
        projection = projection || {};
        limit = projection.limit || 10;
        skip = projection.skip || 0;
        fields = projection.fields || {name: 1, emp_id: 1};
        return EmployeeDetails.find(query, {
            limit: limit,
```

```
        skip: skip,
        fields: fields,
    sort: {emp_id: 1}
      });
   });
}
```

Our publication takes two parameters, a query and a projection, which is nothing but an object having the `limit`, `skip`, and `fields` properties. For pagination, `limit` and `skip` are very important. The `skip` property is equivalent to the first argument we pass for limit in a MySQL query. Let's subscribe to the publication from the client and display the data.

Add the following template to `PageData.html` and include the template in the `body` tag as `{{>empDetails}}`:

```
<template name="empDetails">
<div>
    {{#each collection}}
<div>{{name}}</div>
<div>{{emp_id}}</div>
    {{/each}}
<button>Next</button>
</div>
</template>
```

In `PageData.js`, inside the client block, add the following subscription and template handler code:

```
    Meteor.subscribe("employee_details");
    Session.set("page", 0);
    Template.empDetails.helpers({
        collection: function() {
            return EmployeeDetails.find({}, {
                limit: 10,
                skip: Session.get("page") * 10,
    sort: {emp_id: 1}
            });
        }
    });
    Template.empDetails.events({
"click button": function() {
            Session.set("page", (Session.get("page") + 1));
        }
    });
```

We are using session to keep a track of the page number. In the template helper, we are limiting the query to 10 documents only. This is because we haven't specified in our subscription how many documents to fetch; it will fetch only 10 documents as per our publication setup. Start the application and visit it in the browser. You will see the names of the employees with the next button. Click on **next** and you will see no more data.

The application has 1000 records. However, the client has only 10 because of the subscription. To implement a proper pagination, we need the total count of the collection beforehand so that we can have the logic to decide as to when to show the previous and next page button. Another issue with MeteorJS is, the cursors are not aware of the subscription's specificity. When there are multiple subscriptions that update the same cursor that affects the result set with pagination.

Handling data with pagination in a MeteorJS application is a bit tricky. There is no native support, but it is there in the roadmap. At the moment, we can rely on packages that have solved all the previously mentioned problems in custom ways. What if we want to develop our own solution? Let's see what we can do about it.

A subscription is always reactive. Whenever there is an insertion, the publication will send the data to the subscriber if it makes sense. In the preceding example, we are sorting the collection by `emp_id` and fetching the first 10 documents. If we insert a new document with `emp_id` that can fall within this 10-document range, you will see that it update the template immediately. However, if we insert a new record that falls out of the 10-document range, the subscription will not receive the data. Also, if there is another subscription that plays with the same collection, we will end up with the cursor having irrelevant data. Now you can understand why pagination has a problem in MeteorJS applications.

The first issue, knowing the total count, can be solved by having a different publication and subscription that gives the total count of the documents present in the database based on the query. Adding the following publication and subscription in the server and client, respectively, can help us solve the first issue:

```
 //In server
Meteor.publish("employee_count", function(query, projection) {
    query = query || {};
    return EmployeeDetails.find(query).count();
 });
//In client
var empCountHandle = Meteor.subscribe("employee_count");
```

Use the subscription handle `empCountHandle` to handle the UI update.

The second issue is cursors are not aware of the subscription. This can be addressed in two ways. One is by using the `FindFromPublication` package that offers us methods to have a cursor that is publication/subscription-aware. Instead of publishing data using `Meteor.publish` in the server, use `FindFromPublication.publish`. In the client, subscribe as usual but access the collection using the following code:

```
EmployeeDetails.findFromPublication("employee_details", {},
options);
```

Now, other subscriptions' data will not affect the cursor returned by the `findFromPublication` method.

The other solution is to use template-level subscriptions. With the latest version of MeteorJS, it is possible to create a subscription inside templates. The advantage is that the subscription will be destroyed once the template is destroyed. While using template-level subscriptions, we have to wait for the subscription to complete and then find the documents with the updated limit. This way, we can make sure we get the data from the server before showing it. Check at the pseudo code, as shown in the following code snippet, to achieve the second approach:

```
Template.empDetails.onCreated(function() {
        var instance = this;
        // initialize the reactive variables
    //one for subscription and other for querying
        instance.loaded = new ReactiveVar(0);
        instance.limit = new ReactiveVar(5);

        //Autorun to detect changes to the limit set by click
event
    instance.autorun(function() {
            // get the limit
            var limit = instance.limit.get();

            // subscribe to the posts publication
            var subscription =
instance.subscribe('employee_details', limit);

            // if subscription is ready, set limit to newLimit
            if (subscription.ready()) {
                instance.loaded.set(limit);
            }
        });
        //Cursor – should be used in the helpers to return the
collection document.
        instance.empDetails = function() {
```

```
            return EmployeeDetails.find({}, {
                limit: instance.loaded.get()
            });
        }
    });
```

In the preceding code, we have limited and loaded properties attached to the template instances that are reactive variables. When the user clicks on the next button, we will set the limit to the new value. This will trigger the `autorun` method we have setup inside the `onCreated` method. With the new limit, we subscribe for the new set of documents and wait for it to be received with the help of the `.ready` method of the subscription handle. Once ready, then we set the new limit to the `loaded` property, which in turn provides the new set of documents to the template. It is necessary to use `Template.instance().empDetails()` inside the template helper to return the current document collection.

I hope this helps you to write a custom logic to handle pagination. What are the packages available to handle pagination? There are many and notable ones are `doctorpangloss:filter-collections` and `alethes:pages`.

The `filter-collections` package offers more than pagination. It supports filtering, sorting, searching, and pagination. You can find the demo of the package at `http://filtercollections.meteor.com/`. It works well with `Collection2` as well. The package addresses the pagination problems we discussed using the `FindFromPublication` package style. We have to publish using the `FilterCollections.publish` method and in the client, we can use `newFilterCollections` to set up the operations. We can specify templates and other sort and filter parameters while instantiating `FilterCollections` on a collection. With the instance, we can do almost all related operations programmatically. The package gives a lot more control over the data. Visit the documentation at `https://atmospherejs.com/doctorpangloss/filter-collections` if you want to give it a shot.

The `pages` package is an easy-to-use and bootstrap-integrated solution. It allows both sorting and filtering and supports infinite scroll out of the box. Also, the package offers templates to show the navigation button for pagination. Visit the package documentation at `https://github.com/alethes/meteor-pages` to learn other available options.

The preceding example should have helped us to understand problems and solutions to handle large volumes of data with pagination in a MeteorJS application.

# Summary

In this chapter, we have learned a few internals of MeteorJS. Let us summarize what we have learned in this chapter. We have learned end to end about MeteorJS' reactivity to its depths by looking at Tracker.js, which is MeteorJS' core reactivity secret. We have learned how to build REST APIs and REST-based systems using various packages and solutions with MeteorJS. Finally, we have spent time to learn how to handle large volumes of data using pagination. We have learned the problems with pagination because of MeteorJS' reactivity and other principles and also learned how to handle them.

I hope you have enjoyed the chapter.

In the next chapter, we are going to learn how to deploy and scale MeteorJS applications.

# 7
# Deploying and Scaling MeteorJS Applications

So far, we have learned various concepts, tools, and internal ideas of MeteorJS framework and how to use them in developing applications. From a developer's perspective, we have a fair experience on developing MeteorJS applications. By this time, you pretty well know that MeteorJS is, more or less, a collection of nontraditional and advanced concepts put together as a framework.

A traditional way of building web applications is to use HTTP and until today most of the applications are built over HTTP transaction. We know the problems with HTTP and there are plenty of reasons why HTTP transfer is slow. Most of the production infrastructure and stacks are meant for HTTP. In contrary, MeteorJS is new, not widely adopted yet (hopefully it will be), and primarily uses sockets and other concepts that are not traditional.

Finding or creating infrastructures and platforms to host a MeteorJS application is quite a cumbersome task. This doesn't mean we hardly find them, but the options are limited. However, it is notable that such a young framework has quite a few supporting hosting solutions. A few among them are Modulus.io, `www.digitalocean.com`, and Heroku. Can't we deploy a MeteorJS application out of these solutions? Can't we have a dedicated, self-owned infrastructure and platform to host a MeteorJS application?

Many developers and early adopters of MeteorJS ask one particular question before taking the decision to use MeteorJS for their product development. The question is "will it scale?".

In this chapter, we are going to learn about deploying and scaling MeteorJS applications. Precisely, we are going to cover the following topics:

- Understanding MeteorJS application deployment
- Using build tools for MeteorJS applications
- Using deployment tools for MeteorJS applications
- Scaling a MeteorJS application using database oplog tailing setup
- Using third-party MeteorJS hosting solutions

# Understanding MeteorJS application deployment

Have you ever wondered how MeteorJS builds and runs the application while developing applications locally? MeteorJS bundles the JavaScript code, packages, and templates every time we run the application. The `.meteor/local/build` directory is the place where all the bundled files are located. You will find a copy of your code inside the `.meteor/local/build` directory. When we debug our application using `node-inspector` via `meteor debug`, we always see that the execution stops at `.meteor/local/build/main.js`. This is the entry point for the application.

MeteorJS compiles the templates to JavaScript functions and splits the code based on target environment(the server or the client) before serving them. If you have used any compile to JavaScript language such as coffee, or CSS preprocessors such as SASS, all the appropriate files are converted to JavaScript and CSS files during the initial stage of the build process.

The important block of the application is database and the database resides inside the `.meteor/local/db` directory. Once the application starts, the connection to the local database (database name is meteor) is established. If we want to connect with an external database, we should specify the connection string using the environment variable `MONGO_URL` as follows:

```
MONGO_URL=mongodb://user:password@localhost:27017/<database-name>
```

Once the database connection is established, socket connections are initiated between the client and the server and things go on by transferring data based on publication and subscriptions. From deployment perspective, we could see that it is a bundled Node.js application that is running and connecting to a MongoDB.

MeteorJS bundles the application into a Node.js application with its own build tool. However, it doesn't build the application to be a traditional Node.js application, instead as a Node.js application that uses Fibers. To learn about Fibers, visit the Fibers GitHub repository at `https://github.com/laverdet/node-fibers`.

Roughly, all we need is a machine with Node.js and MongoDB. It is that simple. Then, why are people finding it difficult to deploy a MeteorJS application? There are issues to be addressed from different directions. If we build the application using the `meteor build` command in a Mac machine and deploy it to a Linux machine, there are chances that the application will not work as expected. There could be cross-platform issues. For example, the Fibers package that MeteorJS uses is platform-dependent. So it is important to remove and install the Fibers package in the target hosting environment after porting the built files to the target hosting environment. Not just Fibers, any other platform-dependent package used in the application might have the same issue. If you don't want to do any of these in the hosting platform, then build the application in a machine that is similar to the hosting platform in terms of operating system and processor architecture (32/64 bit).

# Build tools for MeteorJS applications

Isobuild is an in-built build tool for MeteorJS. Another popular build tool is demeteorizer, which again is built on the top of Isobuild.

# Isobuild

MeteorJS offers a simple build tool called Isobuild. It converts a single source into a set of runnable programs with respect to the target platforms (Android/iOS/Web). If you are developing a cross-platform application using MeteorJS, then this build process can create builds in accordance with the mobile platforms as well, if added in the application. When you create a MeteorJS application, Isobuild is added to the application. Isobuild will build the source when the following commands are run:

```
meteor run

meteor deploy

meteor build
```

Isobuild can include the NPM package and the PhoneGap package based on the target platform. It has its own package format called Isopack, which we can see inside the `.meteor/local` directory.

During a build process, Isobuild does transpiling, minifying, generating source maps, resolving package references, bundling assets, and so on. If we need any custom functionality, we can extend Isobuild with plugins. What do we get when we run the `meteor build` command? Let us check it out by running the command in one of the projects we had created in the previous chapters. Firstly, try the `meteor help build` command in the application root and you will see the set of options we can pass to the build process.

It is mandatory to provide an output location path to place the created bundle. So we have to run `meteor build <path/to/some/directory>`. Once run, visit that location and extract the bundle file (tarball) to see what is inside. You will find a `README` file that has some instructions about setting environment variables related to the database, root URL, port, and so on. There is the `main.js` file that is the entry point for the application. It is like `index.html` for a web server. To start our application, we have to run the `main.js` file using Node.js. Based on the target platforms added (Android/iOS/Web), we will see the respective directories. In our case, we haven't added any mobile platform. So it will be only the `web.browser` directory and the `server` directory, and they are present inside the `programs` directory.

This bundle provides the server and client source codes. If we have a database setup ready, then we can run this bundle from anywhere just by setting those environment variables as described in the `README` file. I hope this gives a fair idea about what is happening under the hood and how to get the source for deployment from your local MeteorJS application.

# Demeteorizer

Demeteorizer uses the `meteor build` command to create builds. This means, demeteorizer bundles the application using the `meteor build` command and, additionally, adds the `package.json` file to the bundle. The bundle is created in the `.demeteorizer` directory at the root of the application. You can specify an alternate output directory, if necessary, while running the `demeteorizer` command. Demeteorizer can build your application in debug mode using the `demeteorizer –debug` command, which doesn't minify the code so as to enable easy debugging. Once we have bundled our application using demeteorizer, we can then run the same bundle as a Node.js application anywhere just by a few steps as follows:

1. Navigate to `bundle/programs/server` directory into and run the `npm install` command

2. Set up the required environment variables (`MONGO_URL`, `ROOT_URL`, and `PORT`)

3. Start the application using the `npm start` command

When we run `npm install` in the bundle directory, it reads the `package.json` created earlier and installs the necessary dependency packages for the application. Why is it necessary to have `package.json`? What is the purpose of it?

Usually, when we want to run an existing Node.js application, we will install the dependencies by running the `npm install` command. This command reads the `package.json` and installs all the dependencies mentioned in the `package.json`. In our case, demeteroizer adds Fibers, semver, and other dependent NPM packages to the `package.json` file. It also removes Fibers from the `node_modules` directory in the bundle demeteroizer creates. In the target hosting environment, when we run `npm install`, it will install Fibers packages afresh to avoid platform issues.

After installing all the dependencies, we can either run the `main.js` file directly or run the `npm start` command, which will again run the `main.js` file only. Make sure you are ready with the database and environment variable setup.

For more details,visit the demeteorizer package at `https://github.com/onmodulus/demeteorizer`.

# Deploying a MeteorJS application

Meteor Up and meteor-deployment-manager are a few noticeable deployment tools available for MeteorJS deployment. A common thing about both these tools is that all of them are NPM packages. We have to install them using NPM and run the appropriate commands in the terminal. Let's see them in detail.

# Meteor Up

Meteor Up is a popular MeteorJS deployment tool. It only supports deployment for Debian/Ubuntu Linux flavors and Open Solaris target machines at the moment. Mup is a complete solution for MeteorJS application deployment. It builds the application more like what demeteorizer did using the `meteor build` command and also installs the platform-dependent packages such as Fibers before running the application in the target host.

You can install mup using the `npm install -g mup` command. There are two steps to complete the deployment using mup. One is to set up mup, and other is to deploy the application. To set up mup, use the `mup setup` command. It will create the `mup.json` and `settings.json` files in the directory where the preceding command is run. The `mup.json` file is completely documented with comments, and this file is where we have to specify the hosting credentials. It is better to keep mup files out of the application so that you don't expose the credentials of the host.

In the `mup.json` file, we provide the hosting server's IP address, username, and password for SSH login. Mup will try to log in to the server and then copy the built application bundle to the host when we run the `mup deploy` command. In the `mup.json` file, we can specify whether we want to install a new mongo database, and which version of Node.js to use. It is also important to specify the local application path for mup to find. Mup takes care of setting up the environment variables, but we have to provide them in the `mup.json` file.

In the hosting environment, mup keeps the application at `opt/<appName>/app`. If the `appName` is not specified in the `mup.json` file, the default `appName` will be meteor. Mup uses `upstart` to monitor the application in terms of restarting (upstart is an event-based replacement for the `/sbin/init` daemon that handles the starting of tasks and services during boot, stopping them during shutdown and supervising them while the system is running). The configuration file for upstart will be at `/etc/init/<appName>.conf`. From the hosting server, we can start and stop the application using the `start <appName>` and `stop <appName>` commands. Logs for the application can be reached at `/var/log/upstart/<appName>.log`.

If we have our mongo database created by mup, it cannot be accessed from outside. We have to SSH log in to the hosting server and access the database. The name of the database will also be the `appName` itself. Once logged in, we can run `mongo <appName>` to access the application database.

Mup provides other commands to control the application from the local machine. The `mup reconfigure` command will reconfigure the application at the hosted server if we want to change environment variables or `settings.json`. We can start, stop, or restart the application using `mup start`, `mup stop`, and `mup restart`, respectively.

Mup has many features including deploying to multiple servers, SSL support, and using SSH keys for connecting to the host and custom meteor binaries. Visit the GitHub repository of mup at `https://github.com/arunoda/meteor-up`.

# Meteor deployment manager

**Meteor-deployement-manager** (**MDM**) is very similar to mup. It does exactly what mup does, except that it requires the application to be a Git repository. Similar to `mup.json`, here we need `deploy.json` where we can specify the server details, which can be generated using the `mdm generate` command.

For mdm to work, we have to do a certain basic setup in the hosting server. We have to install Git, Nginx, and MongoDB. It is good to have a nonroot user to run the Node.js commands. So create a user with the `adduser` command. Give permission to the user to execute commands without a password, by adding the user to the sudoers list. At the same time, we don't want to give the new user all sorts of permissions, instead just the permissions required. We want the new user to start and stop the application. So add the following line to `/etc/sudoers.d/<appName>`:

```
meteor ALL = (root) NOPASSWD: /sbin/start <appName>, /sbin/stop
  <appName>, /sbin/restart <appName>
```

Export the MongoDB connection string to the environment variable, as we did before using the `export MONGO_URL="mongodb://localhost:27017"` command. Next is to set up Nginx as a proxy to allow the access to port 80. For security reasons, we will have Nginx to act as a proxy and send the requests to our application.

Take the backup of Nginx configuration file and replace it with the following content:

```
events {
    worker_connections  1024;
}

http {
  include        mime.types;
  gzip  on;

  server {
    listen 80;
    server_name your-app.com *.your-app.com;
    location / {
      proxy_pass http://127.0.0.1:2000/;
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
      proxy_set_header Host $host;
    }
  }
}
```

We are going to run our application at port 2000 and so, in the configuration, we are specifying the `proxy_pass` directive to be `http://127.0.0.1:2000/`. Now, start Nginx using the `service nginx start` command.

MDM also uses upstart to manage the application. We will add an upstart script that will monitor and manage the application. Create a configuration file with the application name in the /etc/init directory, and add the following code and modify it as per your application:

```
#!upstart
description "Application Upstart"

env APP_NAME='<appName>'
env PORT='2000'
env ROOT_URL='http://www.your-app.com'
env NODE_BIN='/home/meteor/.nvm/v0.10.40/bin/node'

env SCRIPT_FILE="<path/to/>bundle/main.js" # Entry point for the
  nodejs app
env RUN_AS="meteor"#use we added before

start on (local-filesystems and net-device-up IFACE=eth0)
stop on shutdown

script
export LOG_FILE="/home/meteor/$APP_NAME/log/upstart.log"
touch $LOG_FILE
chown $RUN_AS:$RUN_AS $LOG_FILE
chdir /home/meteor/$APP_NAME/builds/current
exec sudo -u $RUN_AS sh -c " PORT=$PORT ROOT_URL='$ROOT_URL'
  $NODE_BIN $SCRIPT_FILE >> $LOG_FILE 2>&1"
end script

respawn
respawn limit 5 60
```

Modify the preceding script carefully as per your application.

Once all these configurations are done, we have to install a specific version of Node.js, and we should install as the new user we had created in order to avoid any file permission issues. Log in as the new user and install the version we had mentioned earlier in the upstart configuration file using the nvm or n packages. Also, install MeteorJS so that MDM can build the application.

Create a directory with the application's name to clone your application. MDM expects to have the builds, log, working, and source directories inside the application directory. Create them all and initialize Git inside the source directory. After init, add the remote origin using the git remote add origin <repo url> command and then pull the latest code. MDM expects an origin remote to be present.

All the server setup is done. In a local development machine, we have to install MDM via NPM, and then inside the application, run `mdm generate` to create a `deploy.json` file. The file has a set of options to fill in so that MDM can deploy the latest code and start the application. The file content is as follow:

```
{
  "options": {
    "meteorite": false,
    "insecure": false,
    "meteorRelease": "1.2.0.1"
  },
  "environments": {
    "staging": {
      "hostname": "staging.your-app.com", //host ip or domain
      "port": 22, //port to login with
      "username": "meteor", //new user we created
      "password": "secure-password", //login password
      "deploymentDirectory": "<path/to>", //
      "gitBranch": "master",
      "taskName": "leaderboard"
    }
  }
}
```

The `taskName` property must match the upstart configuration file name we had created in the server. The `deploymentDirectory` must be the path to the application directory we had created in the server. The `gitBranch` property is the one to specify which branch to deploy. Once all these configurations are done, push the local code to the branch and run `mdm deploy` followed by the `mdm start` command, which will deploy and then start the application at the specified server.

Quite a bit of work, but this does the job. You can check the mdm package at `https://www.npmjs.com/package/meteor-deployment-manager`.

# Scaling a MeteorJS application

Whatever we have discussed is good for a single instance application. If our application grows with a bigger user base and if the traffic is increasing, we obviously have to scale the system. Most of the time, scaling can be done horizontally, which means we will add more instances and then route the traffic to instances having less load at the moment. With some scaling solutions such as meteor cluster, we can scale vertically by adding more cores to the server.

Let's see in detail how Nginx and meteor cluster help us in scaling our application.

# Scaling with Nginx

We have seen how to use Nginx while doing MDM setup. Now, let us see how to use it to scale MeteorJS applications. It is good to install the latest Nginx that supports WebSockets. We are going to use Nginx to be a load balancer, which will redirect the requests to the application instances based on the load in each instance. All we need to do is change the Nginx configuration file to accommodate the other instances and to enable a sticky session.

We have to do the basic setup first, which will proxy the requests to the actual application. Let's us see a sample configuration used by a Meteor platform itself. The following is the basic configuration used by a Meteor platform after bundling:

```
# we're in the http context here
map $http_upgrade $connection_upgrade {
  default upgrade;
  ''      close;
}

# the Meteor / Node.js app server
server {
  server_name yourdomain.com;

  access_log /etc/Nginx/logs/yourapp.access;
  error_log /etc/Nginx/logs/yourapp.error error;

  location / {
    proxy_pass http://localhost:3000;
  # http://wiki.Nginx.org/HttpProxyModule
    proxy_set_header X-Real-IP $remote_addr;
  # pass the host header -
  http://wiki.Nginx.org/HttpProxyModule#proxy_pass
    proxy_set_header Host $host;
    # recommended with keepalive connections -
    http://Nginx.org/en/docs/http/ngx_http_proxy_module.html#
    proxy_http_version
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
  }
}
```

This is a basic configuration where we have specified our domain, the proxy entry and the HTTP context. The load balancing configuration is as follows

```
upstream myAppName {
  ip_hash;                 # for sticky sessions, more below
  server 10.0.0.1:3000;  # server 1, core 1
  server 10.0.0.1:3001;  # server 1, core 2
  server 10.0.0.2:3000;  # server 2, core 1
  server 10.0.0.2:3001;  # server 2, core 2
  # or whatever other appropriate combination
}

server {
  listen 80;
  server_name www.myapp.com
  # and all other "server" directives from previous section

  location / {
    # the "hostname" below must be same myAppName from upstream
    directive above
    proxy_pass http://myAppName/;
    proxy_set_header X-Real-IP $remote_addr;
   proxy_set_header Host $host;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
  }
}
```

The upstream directive is where we specify that we need a sticky session with the `ip_hash` directive. This will take care of the request from a unique IP address to be sent to the same instance next time. According to the configuration, we will have four instances running at various ports and the requests will be load balanced among them. It is very important to use the same upstream directive name in the `proxy_pass` directive.

Nginx also serves another purpose. If we want to enable SSL, but don't want the MeteorJS application to support it, we can configure Nginx to support SSL and pass through the request to the application as a non-SSL request. This is called SSL termination, and Nginx is pretty good for solving this issue as well.

Now that we have a load balancer, we can run as many instances as we need to scale our application. However, it is important to note that when we add a new instance and reconfigure the load balancer, it will reset the existing DDP connections.

# Scaling with Meteor cluster

Cluster is a MeteorJS package that has to be installed with the application using the `meteor add meteorhacks:cluster` command. Why Cluster? What problem does it solve? There are a number of issues Cluster solves; lets see all of them.

Cluster offers an in-house solution for the scaling problem. Unlike the previous scaling method using Nginx, we can configure our application to scale, without having to install and configure any other external server or proxy.

As mentioned earlier, when we add a new instance to a Nginx load balancer, it will reconfigure the server, which will lead to the resetting of DDP connections. With Cluster, this is not going to be the case. Also, if the load balancer goes down, then there is no more access to the application. We need to set up multiple load balancers to handle such a case. Cluster offers an elegant solution to this problem as well.

Let's understand what Cluster does; this will give us a fair idea of how it solves all the above mentioned problems. When we install the Cluster package to our application, we make the application running in the primary instance (one pointed via the DNS) to act as the load balancer as well. You don't need a separate server or tool for load balancing. The Cluster package is capable of identifying new instances and directing traffic to them. This prevents connection reset.

How does Cluster do all of this? It does something called a service discovery using MongoDB. If you are familiar with micro services, you would know what a service discovery is (you can learn about micro services from the MeteorJS documentation website). Cluster maintains the instance details in MongoDB, and uses this to identify instances and to route the traffic. This MongoDB need not be necessarily a new database. We can use the existing database as well. During deployment, we just have to export a few environment variables for Cluster to identify things and then everything will be in place. The following are the environment variables required for Cluster to work:

```
export CLUSTER_DISCOVERY_URL=mongodb://host:port/db,
# this is the direct URL to your server (it could be a private
  URL)
export CLUSTER_ENDPOINT_URL=http://ipaddress
# mark your server as a web service (you can set any name for
  this)
export CLUSTER_SERVICE=web
```

Now that we have all the environment variables in place, we can add/remove as many instances as we want and Cluster will take care of load balancing.

How does Cluster handle a sticky session? When a request comes from a unique IP address, a cookie is set in the client, which will help to redirect the request to the same instance every time the request comes from the same IP address.

# Balancers

The problem of a single load balancer being bottleneck is not solved yet. If the default balancer, which is nothing but the server pointed via DNS, is down, then the entire setup will be inaccessible. Cluster has a solution for this too. Convert a few other instances to act as balancers by exporting another environment variable as `export CLUSTER_BALANCER_URL=<another instance url>`. This will ensure that, even if the primary instance is down, requests are balanced via the other configured instance.

# The multicore support

Cluster also supports multicore servers. To leverage multicore support, just export another environment variable as `export CLUSTER_WORKERS_COUNT=auto`. How does Cluster support multicore processing? It creates the clone of the application, which is called worker, and then uses a different way to distribute the traffic. Static requests will be served by the primary application, SockJS requests will be proxied to the workers via the primary application, and WebSocket requests will be handled directly by the workers. This is how we can scale MeteorJS applications vertically.

# The SSL support

It is important to support SSL for security. Being a Node.js application, how can we support SSL? Previously, we had Nginx as a proxy, which can do SSL termination for us. With Cluster in place, we have no component to terminate SSL.

We can handle them in many ways. If you are using mup, then it takes care of it by installing `Stud` in the front of the application. Stud does the SSL termination part. If we are not using mup, then we have to use a proxy. A proxy need not be Nginx, instead we can have the Node.js package `http-proxy` (`https://www.npmjs.com/package/http-proxy`) or `Meteor-SSL-proxy` (`https://github.com/Tarang/Meteor-SSL-proxy`) that will do the job.

# Mup and Cluster

Mup and Cluster goes hand in hand. We know that mup supports multiple environment deployment. So we can specify all the environment variable that we had created in the `mup.json` file and then mup will take care of the Cluster setup. The sample `mup.json` of `https://bulletproofmeteor.com` is as follows:

```
{
  // Server authentication info
  "servers": [
    {
      "host": "ip-1",
      "username": "root",
      "pem": "./bulletproofdo",
      "env": {
        "CLUSTER_BALANCER_URL":
        "https://one.bulletproofmeteor.com"
      }
    },
    {
      "host": "ip-2",
      "username": "root",
      "pem": "./bulletproofdo",
      "env": {
        "CLUSTER_BALANCER_URL":
        "https://two.bulletproofmeteor.com"
      }
    },
    {
      "host": "ip-3",
      "username": "root",
      "pem": "./bulletproofdo",
      "env": {
        "CLUSTER_BALANCER_URL":
        "https://three.bulletproofmeteor.com"
      }
    },
    {
      "host": "ip-4",
      "username": "root",
      "pem": "./bulletproofdo"
    }
  ],

  // Install MongoDB in the server, does not destroy local MongoDB
```

```
      on future setup
      "setupMongo": false,

      // WARNING: Node.js is required! Only skip if you already have
      Node.js installed on server.
      "setupNode": true,

      // WARNING: If nodeVersion omitted will setup 0.10.33 by
      default. Do not use v, only version number.
      "nodeVersion": "0.10.33",

      // Install PhantomJS in the server
      "setupPhantom": true,

      // Application name (No spaces)
      "appName": "meteor",

      // Location of app (local directory)
      "app": "../",

      // Configure environment
      "env": {
        "ROOT_URL": "https://bulletproofmeteor.com",
        "MONGO_URL": "mongodb://app:password@mongodb.com/dbname",
        "MONGO_OPLOG_URL": "mongodb://app:password@mongodb.com/local",
        "DISABLE_WebSockets": "1",

        "CLUSTER_DISCOVERY_URL":
        "mongodb://app:password@mongodb.com/dbname",
        "CLUSTER_SERVICE": "web"
      },

      // Meteor Up checks if the app comes online just after the
      deployment
      // before mup checks that, it will wait for no. of seconds
      configured below
      "deployCheckWaitTime": 15
    }
```

If you watch carefully, you can find that all the necessary environment variables are placed in the appropriate blocks.

Cluster offers a solution for most of the common problems. No more complaining about MeteorJS' scalability.

For more information about the Cluster package, check out the Git repository at
`https://github.com/meteorhacks/cluster`.

# The oplog tailing setup

The oplog tailing is again another aspect of making MeteorJS applications faster.
Firstly, let's understand what is oplog tailing and then we'll discuss how to use it
with MeteorJS applications.

To discuss oplog tailing, we have to know the MongoDB replica set. When running
applications in production, it is common to have more than one databases process
as a backup. One of them will be a primary process and the rest will be secondary
processes. Write happens to the primary process from the application and the data is
then copied to the secondary processes. How do secondary processes copy the data?
It doesn't simply copy the entire primary database on every interval. This is where
operation logs, called oplogs, come into the picture.

Whenever there is a change in the data, the operation log will keep track of it. The
secondary processes copy these operations and update themselves. This oplog can
serve as an important optimization factor for a MeteorJS application and thus we
need to have a replica set for MongoDB in production.

# Creating a replica set

Let's see how to create a replica set for MongoDB. The following command will
create different MongoDB processes:

```
mongod --replSet <some-id> --port 27017 --dbpath <database/path>
  --fork --logpath <database/log/path>


mongod --replSet <some-id> --port 27016 --dbpath <database/path>
  --fork --logpath <database/log/path>
```

The `<some-id>` must be a unique string, which must be same for both the
commands. We also need this ID to create the replica set. Once these processes
are ready, we can create the replica set. Log in to the mongo shell of the one of the
above processes (whichever you want to keep as primary) and run the following
commands:

```
config = { _id: '<some-id>', members: [{_id: 0, host:
  'localhost:27017'}, {_id: 1, host: 'localhost:27016}]};
rs.initiate(config);
```

If everything is fine, the replica set will be initiated. It might take some time. To check the status, we can use the `rs.status()` command in the shell. We can change the primary and secondary configurations by the following code:

```
config.members[0].priority = 1
config.members[1].priority = 0.5
rs.reconfig(config)
```

The member with highest priority will be the primary database.

We need a separate user to read the oplogs. Get into the admin database of the MongoDB process and add a user as follows:

```
db.addUser({user:'oplogger',pwd:'YOUR_PASSWORD',roles:[],
 otherDBRoles:{local:["read"]}})
```

MongoDB maintains the `local.oplog.rs` collection to track all the changes. We have created a user `oplogger` and given access to the user to read the oplog collection. Now, our replica set is ready.

# Accessing the oplog from an application

To expose oplogs to a MeteorJS application, we have to export an environment variable as follows:

```
export MONGO_OPLOG_URL='mongodb://oplogger:
 YOUR_PASSWORD@localhost:27017,localhost:27016/
 local?authSource=admin'
```

Now, the MeteorJS application will listen to the oplogs and optimize the data diff operation. If you are using upstart in production, add the necessary scripts to start the replica set as well.

# Third-party MeteorJS hosting solutions

So far, we have seen various solutions and options to set up our own infrastructure for hosting a MeteorJS application. Now, we have a broad picture about what all components are required to host a MeteorJS application. It is not always possible to have a self-set infrastructure. Most of the developers try to get a hosting space from some providers and then host the application. In the case of MeteorJS, we have to be very specific and picky about the third-party hosting solutions for various reasons.

It is important to check whether the sticky session support is available. Many will claim they do have, but later there won't be. Similarly, it is better to have WebSocket support. Even if WebSocket support is not there, MeteorJS will fall back to SockJS.

However, while choosing, choose the one with the support. It is good to host MongoDB in one of the MongoDB hosting providers, if you are not an expert in it. The MongoDB solutions out there have tested against the possible security issues, and this will be safer for a production application. Unless you are a MongoDB expert or you don't know everything about securing your database, prefer a provider.

Let's see a few popular hosting solutions available for hosting a MeteorJS application.

# Meteor Galaxy

Meteor Galaxy is MeteorJS' official hosting platform. Galaxy hosts our application in the Docker containers running on AWS EC2. If any container fails, Galaxy will monitor and replace it with a functional container. There are many more advantages of using Galaxy. One can get the real-time metrics of his/her application. There is a dashboard that provides access to track the connected clients and monitor the application. Galaxy coordinates with version updates and makes the transition between versions smooth enough.

Another advantage is that we can just use the `meteor deploy` command itself to host our application. To learn more about pricing and other support, visit `https://www.meteor.com/galaxy`.

# Modulus.io

Modulus.io is a hosting solution targeting mostly Node.js applications. They are the ones that created demeteorizer build tool to make MeteorJS application work on their platform. Modulus.io runs on AWS and thus it is pretty stable for production apps. Modulus.io offers servos (VPS) with specific configurations that can be scaled very easily. You can run your own MongoDB in the environment or opt out for external MongoDB solutions as well. There is no explicit oplog support in Modulus.io. We have to contact their team to get the support. Modulus.io supports the sticky session, WebSockets, SSL endpoints, and obviously Node.js support; there are many MeteorJS applications running in Modulus.io. There is also a free trial for the new comers to give it a try. As a whole, it is a solution with some of the required stuffs that are in place.

# Digital Ocean

**Digital Ocean** (**DO**) is very popular among MeteorJS developers. Many like the hosting solution because of the traditional approach. DO offers droplets (VPS) that is a bare VPS where we can configure our application. DO has published many articles about how to host MeteorJS applications in their servers. One such tutorial is available at `https://www.digitalocean.com/community/tutorials/how-to-deploy-a-meteor-js-application-on-ubuntu-14-04-with-nginx`, which is a comprehensive article for deploying MeteorJS applications in Ubuntu 14.04. Mup works with other platforms as well. However, it is very much compatible with DO. Prizing is also fair. If you want to use Nginx for proxying and load balancing, then DO is a good choice.

# Database solutions

Compose is one of the widely used MongoDB solutions for MeteorJS applications. They provide oplog support out of the box. We just have to create the database, add a user, and copy both MongoDB URL and oplog URL to the application hosting server. This will give access to the MongoDB. They maintain backups as well. Compose.io offers a trial to try their solution.

MongoLab is another MongoDB solution available. There is a sandbox version with 0.5GB storage. You can use it to try the solution. They offer both dedicated and shared hosting for MongoDB.

There are many other hosting solutions for MeteorJS. Heroku and Nodejitsu are mentionable ones. Similarly, we can host MeteorJS applications in AWS and GCE as well.

I hope the chapter provides a complete idea of deploying and scaling MeteorJS applications.

# Summary

Will MeteorJS application scale? You know the answer. With the right tools and techniques, obviously it will. Let us summarize all that we have learned so far in this chapter.

To deploy a MeteorJS application, we have to build it using `meteor bundle` or demeteorizer. An app must be bundled against a proper Node.js version and platform.

MongoDB must be installed and the required environment variables must be set to connect the MongoDB to a MeteorJS application.

Mup and MDM are noticeable deployment tools available to deploy a MeteorJS application.

We can use Nginx and meteor cluster to load balance and scale MeteorJS applications.

It is good to opt out external MongoDB solutions if we are skeptical about MongoDB setup and security.

As a whole, definitely, a MeteorJS application can hit production without any blocks.

I hope you enjoyed the chapter. In the next chapter, we will learn to develop applications for a mobile platform using MeteorJS.

# 8
# Mobile Application Development

**Responsive Web Development** (**RWD**) is a popular term in Web development. Every newly created website out there now has an implicit requirement to be responsive; responsive for the sake of being mobile-friendly. The traffic from mobile platforms has increased drastically in the past five years and it is expected to grow even more. There is a big shift in how the content is being delivered to the user. There is a huge personalization need that has given birth to the contents being delivered via mobile apps.

Every mobile development platform has plenty of apps in their app store. The categories of apps range from entertainment through business to personal care and what not. It has become very common to search for an interesting app in the app store. The need to make life smarter has its face at every direction, which has resulted in a rapid increase in mobile app development.

While native app development is consistently embraced, still there is a huge community of developers who are going towards multiple mobile platforms supported hybrid applications. Hybrid applications are the ones that are developed in combination with native components using Web technologies and can be used in most of the popular mobile platforms. Cordova, Sencha, Kendo UI, Ionic, React Native, and NativeScript are some examples that either compile down to native code or render inside a WebView in an application.

Hybrid applications those ones that have access to native controls, but run on a WebView. Many prefer hybrid applications because they are one codebase that serves multiple platforms. This means less to maintain, less to code, less resources, and less time to develop.

JavaScript being so popular is playing a key role in hybrid application development. Now, we can proudly say that this misunderstood, underestimated language is truly universal. MeteorJS had the idea of one codebase for multiple platforms and so it supports mobile platforms as well. Currently, it supports only iOS and Android and support for Windows is expected in the near future. Underneath, MeteorJS uses Cordova/PhoneGap to build the application. In this chapter, we are going to learn how to develop applications for mobile platforms using MeteorJS, as follows:

- Getting started
- Developing a simple mobile application
- Builds and deployment
- More about mobile app development using MeteorJS

# Getting started

We have seen the fabulous work of MeteorJS in the browser front. Let's see how it supports mobile platforms. The MeteorJS team has put a huge effort to support mobile platforms with a single codebase. As we mentioned earlier, it uses Cordova to create builds for mobile platforms. This enables us to use PhoneGap utilities in our application. It is good to know about Cordova before getting into the crux of this section.

Cordova, actually Apache Cordova, is a mobile development framework. With Cordova, we can build mobile applications using HTML, CSS, and JavaScript. The bigger advantage is, we don't have to write code targeting each mobile platform. What does Cordova do to run the Web-related code in mobile devices? It provides a container to run the HTML, CSS, and JavaScript code. The container also helps you to access some of the native device functionalities such as GPS, accelerometer, camera, and so on. Those who have worked with Web views in a mobile platform will know exactly what Cordova does. Cordova hides the platform differences and exposes the device functionality in a unified manner to the JavaScript environment.

MeteorJS uses Cordova to create builds, and then we have ways to test it in an emulator or in devices. Let's see how to create an application and launch it in a mobile device using MeteorJS.

Create a MeteorJS application, say `mobileMeteor`. Add the targeting platforms to the application. For example, we want both Android and iOS so we run the following commands one after the other:

```
meteor add-platform android
meteor add-platform ios
```

We know that there is already a code for the counter example using session in the application. Let's use the same to get started with the mobile app. To get started, we have to do certain preparations. We can run the application both in a real device and an emulator. In the case of iOS, we need Xcode to be installed. Agree to the terms and conditions and install Xcode from the App store. If you really want to run your application with Xcode on a real device, you need to have an Apple developer account. In the case of Android, we need JDK and the Android SDK to be installed. It is important to install the Android SDK that is supported by Cordova. It is good to set the Android platform path in the environment path variable for Cordova to find out if you are placing the Android SDK in a custom path.

Once the setup is ready, we can run the app on an emulator just by running the `meteor run ios` command in the case of iOS, and `meteor run android` in the case of Android. This will start the local server and then the application is opened in the emulator. To open the application in the device, connect the device to the local machine. Make sure both the device and machine are in the same local network. Run the `meteor run android-device` command that will start the application in the connected Android device. Similarly, running `meteor run ios-device` will start the application in the connected iOS device. You will be able to see the same boilerplate counter button example opening up in the mobile screen.

For Android, you have to enable the developer options in settings and then reconnect to the local machine if you face the **device not found** kind of errors.

How simple it is to create a mobile application! Very impressive, isn't it? If we take care of the responsiveness of the UI elements, with a single code, we can deploy an application to Web and mobile platforms. Also, we know that MeteorJS provides hot code push. This is applicable for mobile applications as well. Try to make some HTML changes in the templates and you will find the change happening in the application. We can push the updates constantly to our mobile application, like we do for the Web applications.

In the application, if we need to run any set of code specific to Cordova, we can run it using the available flag as follows:

```
if(Meteor.isCordova) {
  //Your stuff here
}
```

We don't have the `Cordova` directory like the `client` or `server` directories that are available only on respective environments. That's all about the basics. Now, let's create a meaningful application to learn more.

# Developing a simple mobile application

Let's create a simple mobile messenger application. The idea of the application is to check the logged-in user's contacts list and then if anyone from those contacts is using the application, it will show them in a list from where the user can start messaging them in the application. The application will allow you to send one-to-one messages to one of the user's contacts and, on the other end, the user can view it and reply. This would be a good start as we will learn many important things to build applications.

Let's break down the problem statement, which will be easy for us to build the application:

1. We need to provide a login interface and while logging in, we must save the necessary details, such as the phone number, to the server database.

2. After logging in, the user should see the contacts using this application, which means we need access to the user's contacts. Once contacts are fetched, we will check whether any of them are using the application by checking against the database. We are not going to store the contacts in the database.

3. If there are matching contacts, the application should show those contacts with their name and phone number.

4. On tapping one of the contacts in the list, we should take the user to the message screen where the user can start typing messages and send them to the other contact.

5. On the other end, the user should receive the message.

We know our goal and before we start, we have to name our application; let's keep it `TellMe`. Create a MeteorJS application with the name `TellMe` and add Android and iOS platforms to the application as we did in the previous section.

# The login interface

We will allow the user to sign up with an e-mail. To keep things light and simple, we will start with a normal sign up provided by the accounts package. You can add the sign up with a mobile number later as your homework. Add the accounts-password and accounts-ui packages to the application. We need some serious restructuring to separate the server code and client code. Create the lib, server, and client directories in the application root directory. To clean up the existing boilerplate code, let's remove both the .css and .js files and keep only the .html file. Remove the hello template-related code from the HTML file including the content inside the body tag.

We need routes to maintain the application state and navigation. We will tie our layouts to routes so that we don't have to micro manage layout rendering. We are going to use FlowRouter for routing and Blaze-Layout for layout management. Add FlowRouter using the meteor add kadira:flow-router command and add blaze-layout using the meteor add kadira:blaze-layout command.

In the lib directory, we will add router.js where we will keep all our routes. Let's write our first route that will render the login layout and the login buttons inside the layout. Add the following home route to router.js as follows:

```
FlowRouter.route('/', {
    action: function() {
        if(!Meteor.userId()) {
        BlazeLayout.render("loginLayout");
      Accounts._loginButtonsSession
        .set('dropdownVisible', true);
        } else {
        FlowRouter.go("/profile");
        }
    }
});
```

FlowRouter allows us to decouple routes from an application logic and keeps it to be merely a router rather than being the application manager such as iron-router. Here, in the route, we have specified that, when http://<domain.name>/ is hit, the loginLayout template should be rendered. We have to create the loginLayout template and this belongs to the client.

Inside the `client` directory, create a directory with the name `layouts` to keep all our layout-related files. In the `layouts` directory, create the `loginLayout.html` file and add the following template code that includes the `loginButtons` template helper as well:

```
<template name="loginLayout">
  <div class="loginContainer">
      {{> loginButtons}}
  </div>
</template>
```

When you use the `{{> loginButtons}}` helper to show the login form, you will see the **Sign In** link with an arrow. We want to show the entire login or signup form without the toggle. To show the form by default, we set the `dropdownVisible` property in the router. If the user has logged in already, he or she will be redirected to the profile route.

Now, visit the browser and you will see that the form shows up without the link. We need to add some CSS to bring the form to the center. Create the `stylesheets` directory inside `client` and add the `login.css` file into it. Add the following CSS to bring the form to the center of the page:

```
/** reset **/
body {
  padding: 0;
  margin: 0;
  background: #f6f6f6;
  height: 100vh;
}
/** reset end **/
/** Login Section **/
.loginContainer {
  height: 100vh;
  display: flex;
  flex: 1;
  align-items: center;
  justify-content: center;
  flex-direction: column;
}
.loginContainer header {
  font-size: 2em;
  font-style: italic;
}
.loginContainer .login-close-text, .loginContainer .login-link-
  text {
  display: none;
}
```

```
.loginContainer .accounts-dialog {
  display: block;
  margin: 0;
}
.loginContainer #login-dropdown-list {
  position: static;
}
.loginContainer .additional-link-container {
  display: inline-block;
  width: 49%;
  text-align: right;
}
.loginContainer .additional-link-container a {
  float: none !important;
}
.loginContainer #signup-link {
  text-align: left !important;
}
/** Login Section Ends **/
```

Visit the browser and you will see that the form is in the center of the viewport. We are ready with the login interface.

We know that it will look pretty on the Web. How about in mobile devices? Stop the application and run it on a mobile or emulator and see how it looks. The screenshot of the application running in an iOS simulator is as follows:
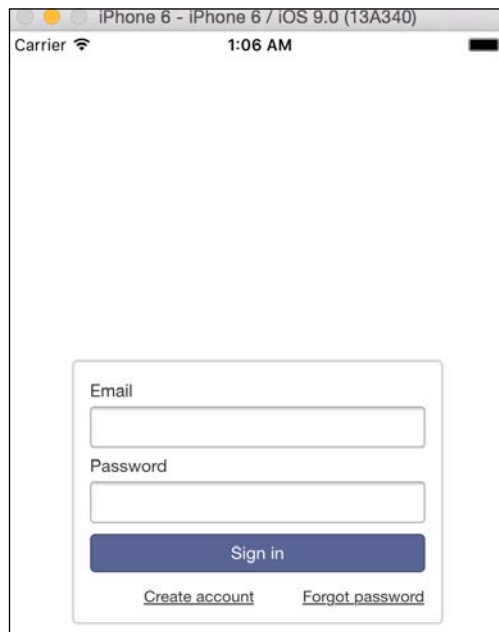
If you have observed, during the start of the application, there was a screen with a black background and white stars shining over with a MeteorJS logo at the bottom. This screen is the splash screen that usually is shown in mobile applications during startup. Where did it come from? MeteorJS, when packages our application for a mobile platform, places these images by default. Even in the emulator or mobile phone, the application has got the default MeteorJS logo icon that is packed during the build process.

What if we want to change them and keep a custom one? Who actually takes care of placing the splash screen and the app icons? Cordova takes care of all these. If you have done some examples with Cordova, you might have come across the `config.xml` file where we can specify these details of the splash screen and app icons. MeteorJS allows us to do the same via a JavaScript file. This file should be `mobile-config.js` and must be placed at the root directory of the application. A sample configuration is as follows:

```
App.info({
  id: 'com.meteor.mobile.myMessenger',
  name: 'myMessenger',
  description: 'Send message to your contacts',
  author: 'Meteor User',
  email: 'noreply@mymessenger.com',
  website: 'http://mymessenger.com'
});


App.icons({
  // iOS
  'iphone': 'resources/icons/icon-60x60.png',
  'iphone_2x': 'resources/icons/icon-60x60@2x.png',
  'iphone_3x': 'resources/icons/icon-60x60@3x.png',
  'ipad': 'resources/icons/icon-76x76.png',
  'ipad_2x': 'resources/icons/icon-76x76@2x.png',

  // Android
  'android_ldpi': 'resources/icons/icon-36x36.png',
  'android_mdpi': 'resources/icons/icon-48x48.png',
  'android_hdpi': 'resources/icons/icon-72x72.png',
  'android_xhdpi': 'resources/icons/icon-96x96.png'
});
```

```
App.launchScreens({
  // iOS
  'iphone': 'resources/splash/splash-320x480.png',
  'iphone_2x': 'resources/splash/splash-320x480@2x.png',
  'iphone5': 'resources/splash/splash-320x568@2x.png',
  'iphone6': 'resources/splash/splash-375x667@2x.png',
  'iphone6p_portrait': 'resources/splash/splash-414x736@3x.png',
  'iphone6p_landscape': 'resources/splash/splash-736x414@3x.png',

  'ipad_portrait': 'resources/splash/splash-768x1024.png',
  'ipad_portrait_2x': 'resources/splash/splash-768x1024@2x.png',
  'ipad_landscape': 'resources/splash/splash-1024x768.png',
  'ipad_landscape_2x': 'resources/splash/splash-1024x768@2x.png',

  // Android
  'android_ldpi_portrait': 'resources/splash/splash-200x320.png',
  'android_ldpi_landscape': 'resources/splash/splash-320x200.png',
  'android_mdpi_portrait': 'resources/splash/splash-320x480.png',
  'android_mdpi_landscape': 'resources/splash/splash-480x320.png',
  'android_hdpi_portrait': 'resources/splash/splash-480x800.png',
  'android_hdpi_landscape': 'resources/splash/splash-800x480.png',
  'android_xhdpi_portrait': 'resources/splash/splash-
  720x1280.png',
  'android_xhdpi_landscape': 'resources/splash/splash-
  1280x720.png'
});

// Set PhoneGap/Cordova preferences
App.setPreference('BackgroundColor', '0xff0000ff');
App.setPreference('HideKeyboardFormAccessoryBar', true);
App.setPreference('Orientation', 'default');
App.setPreference('Orientation', 'all', 'ios');
```

This covers almost all possible sizes, resolutions, and orientations of mobile devices. We know how to play with the splash screen and the app icon now. We can also add Cordova/PhoneGap-specific preferences in this file.

In mobile devices, it is easy to gather a lot of information with or without users' intervention. The e-mail, mobile number, contacts, call history, Wi-Fi information, gallery photos, camera, or other media access are very common. Cordova facilitates to get access to all this information.

To interact with the native platform, we have to ask Cordova to provide access to the required information. To talk to Cordova from MeteorJS, we have to write the required feature access in the platform-specific language and expose it in JavaScript. This is how, many native features are accessed from the JavaScript environment. Most of the common use cases are written as plugins and are released to be consumed from JavaScript. Visit `http://cordova.apache.org/plugins` to search for your requirement. In the case of MeteorJS, we have to add the plugin as a Cordova package to the application. We will see them in detail when we fetch the contacts of the user.

# The profile interface

Coming back to the application, we need to get the phone number of the user so that we can rely on phone numbers to check against our user base. To get the phone number, there are many ways to get it using Cordova plugins. However, they are not reliable. If you are a native developer and know how to do that, write a plugin and contribute it to the community. As far as this book is concerned, building a plugin is out of its scope. So we will provide a simple interface where the user can fill in his name and phone number by himself, which the application will store as the profile information of the user.

Immediately after logging in, the user should be redirected to the profile screen. Let's add a profile route and layout to render the profile completion section. Add the following profile route to `router.js`:

```
FlowRouter.route("/profile", {
  action: function() {
    Accounts._loginButtonsSession.set('dropdownVisible', false);
    if(Meteor.userId()) {
      BlazeLayout.render("layout", {main: "profileForm"});
    } else {
      FlowRouter.go("/");
    }
  }
});
```

After logging in, we need some piece of code that can redirect to appropriate routes as we cannot add that code in a router. Based on the value returned by `Meteor.user()`, we will redirect the user to the root route or profile route. However, we cannot access `Meteor.user()` in the route as per the `FlowRouter` design.

So, we are going to set a tracker on `Meteor.user()` and redirect to appropriate routes as follows:

```
Tracker.autorun(function() {
  if(!Meteor.user()) {
    FlowRouter.go("/");
  } else {
    var user = Meteor.user();
    if (user && user.profile && user.profile.phone) {
      FlowRouter.go("/contacts");
      return false;
    }
    FlowRouter.go("/profile");
  }
});
```

After logging in, the user will be redirected to the profile route. If the profile is completed, the user will be redirected to the contacts page.

In the profile route, we are going to render a layout called `layout` and there is a dynamic template section called `main`, in which we are going to pass the `profileForm` template. Create `layout.html` under the `layouts` directory and add the following layout template code to it:

```
<template name="layout">
    <div class="profileContainer">
        <div class="header">
            <div class="logo">TellMe</div>
            {{> loginButtons}}
        </div>
        <div class="container well well-lg">
            {{> Template.dynamic template=main}}
        </div>
    </div>
</template>
```
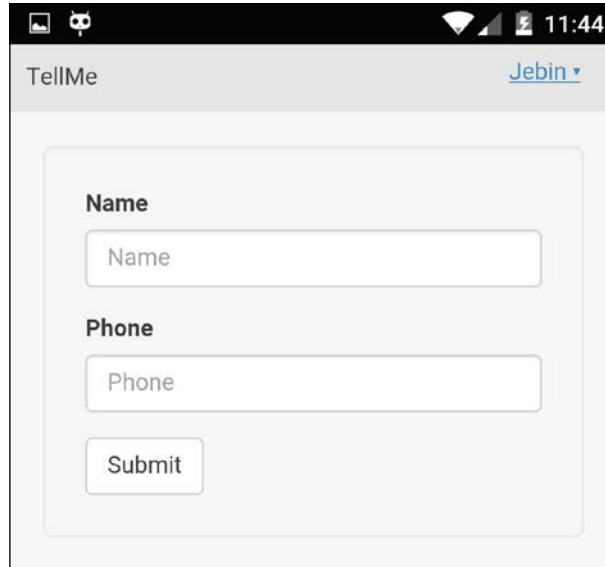
The `loginButtons` helper is also placed in this layout to show the sign out button. As discussed earlier, we have a dynamic template block as well.

The layout is ready and redirections are in place. Now, we need to display the interface for profile completion. Let us organize all profile completion-related code in the `profile` directory under the `client` directory. Create the `profile` directory and add the `profileForm.html` and `profileForm.js` files into the directory. Add the following template and helper code to HTML and `.js` files, respectively:

```html
<template name="profileForm">
    <form class="action">
      <div class="form-group">
            <label for="name">Name</label>
            <input type="text" class="form-control" id="name"
            placeholder="Name">
        </div>
        <div class="form-group">
            <label for="phone">Phone</label>
            <input type="text" class="form-control" id="phone"
            placeholder="Phone">
        </div>
        <button type="submit" class="btn btn-
        default">Submit</button>
    </form>
</template>
```

```js
Template.profileForm.events({
  "submit form": function(e) {
    e.preventDefault();
    var name = e.target.querySelector("#name").value;
    var phone = e.target.querySelector("#phone").value;
    Meteor.call("updateProfile", name, phone, function(error,
    result) {
      if(!error) {FlowRouter.go("/contacts");}
    });
  }
});
```

The profile form will look like the following image:



The template has a form and the helper collects the name and phone number from the form on submission and sends it to the server in order to save it with the user object. There is a server method call that we have to add. Create `methods.js` in the `server` directory and add the following method to save the profile details:

```
Meteor.methods({
  updateProfile: function(name, phone) {

    Meteor.users.update({_id: Meteor.userId()}, {$set: {profile:
    {name: name, phone: phone}}});
  }
});
```

For the sake of simplicity, there is no validation added in the preceding code snippet. However, in a real application, it is very much important to add a proper validation before adding any value to the database. On success, the user will be redirected to the contacts page and here is where the real work starts.

# The contacts interface

We need a route, a template, and template helpers to show the appropriate contacts who are already using the application. Add the following contacts route to `router.js`:

```
FlowRouter.route('/contacts', {
    action: function() {
        if(!Meteor.userId()) {
            FlowRouter.go("/");
        } else {
            Accounts._loginButtonsSession.set('dropdownVisible',
            false);
            BlazeLayout.render("layout", {main: "contacts"});
        }
    }
});
```

To keep contacts-related templates and helpers, let's create the `contacts` directory under the `client` directory. Create the `contacts.html` and `contactsHelper.js` files inside the `contacts` directory and add the following contacts template code to the HTML file:

```
<template name="contacts">
  <div class="contacts">
    {{#if isLoading}}
    <div class="loading">Loading...</div>
    {{else}}
      {{#if isContactsAvailable}}
      <ul class="list">
        {{#each contacts}}
        <li>
          <div>{{this.name}}</div>
          <div>{{this.phone}}</div>
        </li>
        {{/each}}
        </ul>
      {{else}}
        <div class="text-center">No contact available</div>
      {{/if}}
    {{/if}}
  </div>
</template>
```

Before showing this template, we have to read the device contacts, search them against the application database, fetch all matching users, and finally show them in a list. This is where Cordova plugins come into play. We are going to use the Cordova contact fetching the `cordova-plugin-contactsPhoneNumer` plugin. Check the plugin details at `https://github.com/dbaq/cordova-plugin-contacts-phone-numbers.git`. How do we install this plugin into a MeteorJS application? It is pretty much like adding a package. To install this plugin, we have to run the following command:

```
meteor add cordova:cordova-plugin-contactsPhoneNumer@ https://github.com/
dbaq/cordova-plugin-contacts-phone-numbers.git#6fa2e27afa3c54c18e88eb8ba0
649dd4e4200ce2
```

We have to prefix the plugin name with `cordova:`, followed by the name of the plugin which is then followed by `@` and either the version number (such as `1.1.0`) or Git URL of the plugin. If we are using Git URL, it is mandatory to mention the hash of the commit we want to use. If we run the `meteor list` command, we should be able to see the added Cordova plugin. To remove the plugin, run the application with the `meteor remove cordova:cordova-plugin-contactsPhoneNumer` command.

This way we can add any Cordova plugin to the MeteorJS application. Before adding, check for the platforms the plugin supports. A few plugins are written targeting only Android and not iOS or vice versa. So test the plugin before using it against all the platforms in which you want your application to run. Assuming that you have added the plugin to the application, we are going to write code to read the contacts as exposed by the plugin. We are now going to do this in the `contacts` template's `onCreated` callback. Add the following callback to the `contactsHelper.js` file:

```
Template.contacts.onCreated(function() {
  var _this = this;
  this.contacts = [];
  this.loadingContacts = new ReactiveVar(true);
  if (Meteor.isCordova) {
    navigator.contactsPhoneNumbers.list(function(contacts) {
      var contacts = _.filter(contacts, function(eachContact) {
        if (eachContact.phoneNumbers.length &&
        eachContact.phoneNumbers[0].type === "MOBILE") {
          eachContact.phone = eachContact.phoneNumbers[0].number;
          return true;
        }
      });
      Meteor.call("checkContacts", contacts, function(error,
      result) {
```

```
        if(!error) {
          _this.contacts = result;
        }
        _this.loadingContacts.set(false);
      });
    }, function(error) {
      _this.loadingContacts.set(false);
      console.error(error);
    });
  } else {
    _this.loadingContacts.set(false);
  }
});
```

Let's take a look at the preceding code snippet in detail.

We have two variables, `contacts` and `loadingContacts`, at the template instance level. We are going to use the `contacts` array to store the result contacts that we get after matching the contacts that are read from the device against the database. The `loadingContacts` flag is a reactive variable, which means we have to add the `reactive-var` package to the application. Add it using the `meteor add reactive-var` command. This flag is going to help us in the template to show the loading cue while the app is doing the matching operation.

We are going to fetch the contacts only in devices; thus, we need to check whether it is the Cordova environment to run the plugin exposed methods. That is why we check `Meteor.isCordova`. In the `else` part, we set the `loadingContacts` flag to `false` so that, if the user checks the contacts page in the desktop browsers, it will not show the loading cue, instead it will show an empty list.

The plugin exposes the `navigator.contactsPhoneNumbers.list` method that takes a success callback and a failure callback. Most of the plugins follow this pattern of exposing methods that takes success and failure callbacks.

Every plugin author has their own way of exposing the interfaces to access the native features. Some may attach the methods to the `window` object directly, some may namespace it under `window.plugins`, and a few may add it to the `navigator` object. It is important to watch the plugin documentation to learn how to access the feature using the exposed interfaces.

Coming back to the application, we will get the device contacts as the parameter of the success callback. The `contacts` parameter will hold contacts as an array of objects. The sample is as follows:

```
[{
        "id": "1",
        "firstName": "Kate",
        "lastName": "Bell",
        "displayName": "Kate Bell",
        "phoneNumbers": [{
            "number": "(555) 564-8583",
            "normalizedNumber": "(555) 564-8583",
            "type": "MOBILE"
        }, {
            "number": "(415) 555-3695",
            "normalizedNumber": "(415) 555-3695",
            "type": "OTHER"
        }]
}]
```

In the success callback, we just filter the contacts that have `phoneNumbers` and we get only the number that is of the type `mobile`. We send those processed phone numbers to the server for the matching operation. We have created a server method `checkContacts` to perform the operation. The `checkContacts` method will look like the following code:

```
checkContacts: function(contacts) {
  contacts = contacts || [];
  if(contacts.length) {
    var numbers = _.map(contacts, function(eachContact) {
      return eachContact.phone;
    });
    return Meteor.users.find({"profile.phone": {$in:
    numbers}}).fetch().map(function(user) {
      return _.extend(user.profile, {_id: user._id});
    });
  } else {
    return [];
  }
}
```

Add this server method snippet to the `methods.js` file. In this method, we frame a mongo query with the `$in` operator to check the phone numbers against the `profile.phone` entry in the users collection. The matched documents are returned with their `_id`, which we will use for navigation later.

In the template `onCreated` callback, while calling this server method with the contacts fetched, it will return the matched contacts that we will assign to the `this.contacts` array and this will be displayed in the template. Also, it is important to change the `loadingContacts` flag to `false` in both the callbacks to remove the loading cue. Let's add the template, template helpers, and events-related code.

The following code is for the contacts template that has to go in `contacts.html`:

```
<template name="contacts">
  <div class="contacts">
    {{#if isLoading}}
    <div class="loading">Loading...</div>
    {{else}}
      {{#if isContactsAvailable}}
        <ul class="list">
          {{#each contacts}}
          <li>
            <div>{{this.name}}</div>
            <div>{{this.phone}}</div>
          </li>
          {{/each}}
        </ul>
      {{else}}
        <div class="text-center">No contact available</div>
      {{/if}}
    {{/if}}
  </div>
</template>
```

The contact list will look like the following image:
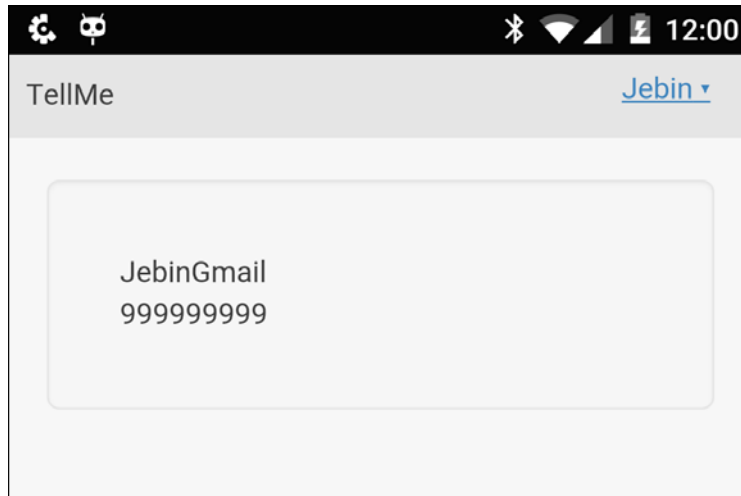


The following code is for the template helpers and events that has to be added in
`contactsHelper.js`:

```
Template.contacts.helpers({
  isLoading: function() {
    return Template.instance().loadingContacts.get();
  },
  isContactsAvailable: function() {
    return !!Template.instance().contacts.length;
  },
  contacts: function() {
    return Template.instance().contacts;
  }
});

Template.contacts.events({
  "click .list li": function(e) {
    FlowRouter.go("/messenger/"+this._id);
  }
});
```

Again, go through all these individual pieces together, and it will make a lot more sense.

To see the matching contacts, you need to run the app on a mobile device and you should have created at least two users with the phone numbers from your contacts so that the matching operation returns those two contacts. The above events snippet clearly tells you that, by clicking on each of the contacts, we are navigating to a new route.

When a contact is shown, it means that the contact is using this application. So we can send a message via our application. To send a message, we need a separate interface. Let us build that interface and add a route to reach that interface.

# The messages interface

The first thing is to add a new route. We need a different layout for messages. We will provide a screen that will show the name of the contact to whom the user is going to send the message, a list of messages sent and received, and a text area where the user can type the message. Users can reach the messages interface with the route `/messenger/:id`.

Add the following route to `router.js` to handle the messages interface navigation:

```
FlowRouter.route("/messenger/:id", {
  action: function() {
    BlazeLayout.render("messengerLayout", {main: "messenger"});
  }
});
```

We need a different layout, so let's create `messengerLayout.html` in the `layouts` directory. Add the following layout template to the HTML file:

```
<template name="messengerLayout">
    <div class="messengerContainer">
        {{> Template.dynamic template=main}}
    </div>
</template>
```

We will keep all message-related code inside a new directory called `messenger`. Create the `messenger` directory under the `client` directory and add the `messenger.html` and `messengerHelper.js` files. Add the following HTML to the `messenger.html` file:

```
<template name="messenger">
  <div class="messenger-wrapper">
```

```
      <div class="name text-center">{{name}}</div>
      <ul>
        {{#each messages}}
          <li class="{{mineClass}}">
            {{> message}}
          </li>
        {{/each}}
      </ul>
    </div>
    <div class="type-area text-center">
      <textarea name="message" class="editor"
      id="message"></textarea>
      <button type="button" id="send" class="btn btn-
      primary">Send</button>
    </div>
</template>

<template name="message">
  <div class="senderName">{{senderName}}</div>
  <div class="message">
    <div class="message">{{content}}</div>
    <div>{{readableDate(created_at)}}</div>
  </div>
</template>
```

Add the following helpers to the `messengerHelper.js` file that reads the messages
and supplies the required variable to the template:

```
Template.messenger.helpers({
  messages: function() {
    var senderId = Meteor.userId(),
      rxrId = FlowRouter.getParam("id");
    return Messages.find({
      $or: [{
        $and: [{
          sender_id: Meteor.userId()
        }, {
          receiver_id: rxrId
        }]
      }, {
        $and: [{
          sender_id: rxrId
        }, {
```

```
          receiver_id: Meteor.userId()
        }]
      }]
    });
  },
  mineClass: function() {
    return this.sender_id === Meteor.userId() ? 'mine' :
    'not-mine';
  },
  name: function() {
    return Meteor.users.findOne({_id:
    FlowRouter.getParam("id")}).profile.name;
  }
});

Template.messenger.events({
  "click #send": function(e) {
    e.target.setAttribute("disabled", true);
    var textarea = document.getElementById("message");
    Messages.insert({
      content: textarea.value,
      sender_id: Meteor.userId(),
      receiver_id: FlowRouter.getParam("id"),
      created_at: Date.now(),
      senderName: Meteor.user().profile.name
    }, function(error, result) {
      if(!error) {
        textarea.value = "";
        e.target.removeAttribite("disabled");
      }
    });
  }
});
var months = ["Jan", "Feb", "March", "April", "May", "June",
  "July", "August", "September", "October", "November",
  "December"]
Template.message.helpers({
  readableDate: function() {
    var date = new Date(this.created_at);
    return date.getDate()+", "+months[date.getMonth()].substr(0,
    3);
  }
});
```

Here is where we need a collection to store all the messages. So let's create the
Messages collection. Add the collections.js file to the lib directory and add the
following collection instantiation code:

```
Messages = new Mongo.Collection("messages");
```

Whenever the user types a message in the text area and presses the send button, the message will be inserted in the Messages collection and, thus, via DDP, it will be sent to the other client. The beauty is we don't need to use any real-time message delivery vendors to implement an instant messaging application. MeteorJS explicitly provides the required tools right in place.

If you now run the application in a device, you will see a broken page. Add the stylesheet of styles.css from the project folder in the Packt library to the application and then the messages screen will look like the following screenshot:

Now, you can improve the application in many ways. You can prompt the user to take a selfie for the display picture. You can add push notification support to the application. You can have an option to create group messages. It is all left to your imagination and practice.

# Builds and deploying

We need to create a build to deploy our application. From the previous chapter, we have learned that MeteorJS has a build creation tool that can build a single source into builds for both Web and mobile platforms. We have to use the following command to create a build:

```
meteor build <path>
```

After creating the build, we can find the Android build under the `<path>/android` directory and, similarly, the iOS build under the `<path>/ios` directory. While running the build process for a mobile, we need to specify the server details to which the mobile application is going to connect. For example, if we want the `TellMe` application to work on a mobile, while running the build process, we have to run the following command:

```
meteor build <path> --server https://tellme.com:8080
```

Once we get the builds, we can distribute it to Play Store or App Store.

# Hot code push

We know that MeteorJS does hot code push to the clients. This is a big advantage for mobile apps. For native mobile applications, it takes time to get the build to the user. Some users will update and some will not. However, in the case of MeteorJS, the applications can be updated without releasing builds. Whenever there is a change in the code, the changed build is downloaded at the mobile device end and once the download completes, the application is updated immediately. This update can be handled from the mobile end by showing some UI cues to reload the application.

# More about the mobile app development

There are many things other than writing an application logic, which we have to know while developing mobile apps with MeteorJS. Let's see what are they.

# Accessing plugin methods

To access the plugin methods, we should wait until the application starts. It is good to write the plugin accessing code inside the `Meteor.startup` callback. However, this is not mandatory. If you know for sure that you are going to access the plugin methods only after the startup, then you need not write them under the startup callback. In the preceding application that we developed, we called the `navigator.contacts.list` method in the template's `onCreated` callback because we knew that the application would have been started by the time, the execution reaches the `onCreated` callback.

# Debugging

Knowing how to debug an application is very important for developers. While developing hybrid applications for a mobile, it is a little hard to debug. This doesn't mean we can't debug. Moreover, we are not going to debug the native bridge of Cordova, but only the HTML, CSS, and JavaScript part.

## Debugging Android

For Android, Chrome DevTools can help us debug the mobile application. Connect the mobile device to a system that has Chrome and then visit `chrome://inspect` in the browser. This will show the list of devices connected to the machine as shown in the following screenshot:
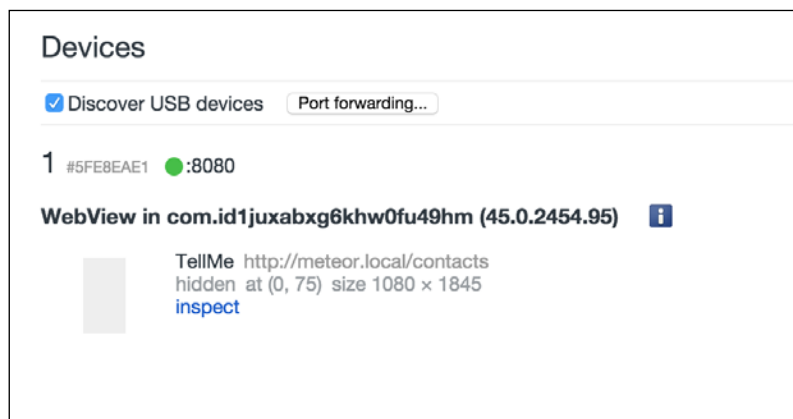
It is important to enable the debugging option on the mobile device. You can learn about enabling the debugging option from the developer tool's documentation link (`https://developer.chrome.com/devtools/docs/remote-debugging`). Once enabled, the device will be visible in the list of devices connected and also it will show the Web views under the device. There will be an inspect link below each Web view as seen in the preceding screenshot. Click on the **web view** that you want to inspect and the Chrome DevTools will show the HTML, CSS, and JavaScript content of the **web view**. We can debug as we usually do in Chrome DevTools. This is the best way to debug the JavaScript content of hybrid applications.

# Debugging iOS

For iOS, we can use the Safari browser instead of Chrome. Once connected with the device, open the Safari browser and find the Develop menu item in the browser menu. Open this and you will find the **Simulator** sub-menu in the case of a simulator or **iPhone**, if you are running the application on a real device. Click on that and another set of menus will open that will show your application server, which in the case of local will be `meteor.local`. Click on that and you will see the Web inspector of Safari opening the **web view** with all the HTML, CSS, and JavaScript content. Then, we can add breakpoints and debug the code. The following screenshot will show you the menu that will bring the Web inspector tool:
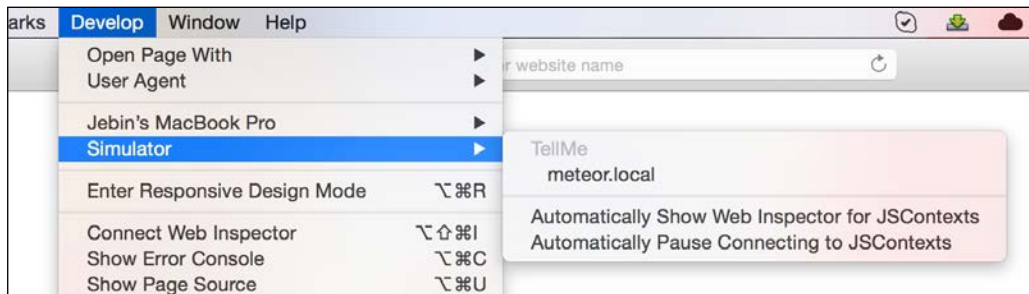


# Testing

The testing packages added to the application against each device platform that we are going to support are very important. We have learned about tiny tests that MeteorJS uses to test the packages. We can use the same test runner to test the mobile packages as well. The following is the command to test packages:

```
meteor test-packages [--ios] [--android] [--ios-device] [--android-device]
```

We can specify which platform to run the tests on; either on emulators or in real devices by passing arguments to the command.

# Packages

If you visit `https://atmospherejs.com/` and search for mobile packages, you will find many useful packages that can provide a range of support for the applications. Community contributors have written a number of packages to make development smooth and easy. Some remarkable ones are mobile-experience, `mdg:camera`, `fastclick`, and `mdg:geolocation`. Because it is HTML and CSS, we can use Twitter Bootstrap for styles. We have already seen how to use Cordova or PhoneGap plugins with MeteorJS.

One of the problems faced by hybrid mobile applications is connectivity. If there is no connectivity, the user cannot use the application. To avoid this, use the device database to store the data and access it as offline storage to support the application while offline. This will enhance the user experience. There are good plugins that allow us to access the database from the Web view.

## The package development

We have learned how to develop packages for MeteorJS. If you want to develop packages that can also support mobile devices, it is are easy to develop. Whatever we learned about package development in *Chapter 3*, *Developing Reusable Packages* still applies here. We should know only two more things.

If our package is going to depend on other Cordova plugins, we have to mention it. For example, if we want to create a MeteorJS package for the `contactsPhoneNumber` Cordova plugin that we used earlier, then we have to specify that in `package.js` as follows:

```
Cordova.depends({
    "cordova-plugin-contactsPhoneNumer":
    "https://github.com/dbaq/cordova-plugin-contacts-phone-
    numbers.git#6fa2e27afa3c54c18e88eb8ba0649dd4e4200ce2"
});
```

While creating a package, we saw that we have to specify the files that our package is going to use by using the `api.addFiles` method in `package.js`. Also, we can specify which file to use in which environment:

```
api.addFiles('contactsPhoneNumber.js', 'web');
```

The preceding code adds the `contactsPhoneNumber.js` file to all the clients. What if we want to add it only to mobile builds? We can do this by passing `web.cordova` as the second parameter instead of just `web`. This will tell the build tool to add this file only for mobile client builds.

This is all you need to know to build a package in order to support mobile applications in MeteorJS.

We have learned most of what we are required to know about mobile applications development using MeteorJS. I hope that the chapter has helped you to get started on mobile application development.

# Summary

Let us summarize the things that we have learned in this chapter. MeteorJS provides support to develop applications for the Web, Android, and iOS. A single source can be built to serve both the Web and mobile platforms. MeteorJS uses Cordova to create mobile builds(`.apk` or `.ipa`). MeteorJS allows us to use Cordova plugins like the usual MeteorJS package. We can test run an application both in the emulator or real devices, and MeteorJS provides enough support to run an application with ease. We can debug applications using browsers. Finally, we have plenty of packages in the MeteorJS ecosystem to support mobile application development.

I hope you find the chapter informative and helpful. In the upcoming chapter, we are going to learn whatever is left to know about MeteorJS application development.

# 9
# Best Practices, Patterns, and SEO

After all the learning so far, we know that MeteorJS is an excellent framework for building Web and Web-based applications. With the instant data sharing and reactive UI mechanisms, it is a best choice for applications to involve in intensive real-time data sharing. We developed a range of such dynamic applications, in the previous chapters, that need real-time data sharing across clients.

We have learned to develop applications using MeteorJS for both Web and mobile. We have also learned many internal concepts of MeteorJS, which every developer must know to develop applications. We have seen how MeteorJS builds applications for different platforms and how to deploy them for production. This is pretty much everything we need to know to decide and adopt MeteorJS for development.

Is that all? MeteorJS is young. There is still a long way to go. Being a Node.js-based framework, some patterns used in Node.js applications can be used in MeteorJS also. However, MeteorJS holds some of the advanced building blocks that need some specialized thinking. How to manage subscriptions effectively and optimize them? There are quite a few things that are specific to MeteorJS and need deep thinking to yield manageable results.

This chapter is meant to learn those techniques, methods, patterns, and safety measure that are not covered in the previous chapters. What is the best way to manage subscription? What MeteorJS offers to support SEO? What patterns should we follow for application development? What are some of the best and important practices we have to follow while development? Many such questions will be answered.

A few of the best practices might be repeated, still, it is good to summarize them to emphasize their importance. With MeteorJS version 1.2, there are big new things added to MeteorJS. Finally, it is important to know where to look for help, who is contributing, and whom to follow, in order to learn more stuff about MeteorJS.

More precisely, the following are the items we are going to focus on in this chapter:

- Summarizing the concepts
- Best practices and application patterns
- SEO
- Meeting the community

# Summarizing the concepts

This section is just a recap of the important concepts and building blocks of MeteorJS. You are free to skip this section if you are familiar with all these items that are given in the following sections.

To start with, maybe you should look again at the principles of MeteorJS at `http://docs.meteor.com/#/full/sevenprinciples`. Based on these principles, MeteorJS employs certain libraries, protocols, concepts, algorithms, and tools to bring to you the awesome framework. Let's look at them.

# Publishing/subscribing

The idea of publishing and subscribing is a significant concept to understand how to develop a typical MeteorJS application. It deceives to be as simple as it reads, but it is one of the toughest concepts of MeteorJS that many don't get right at the first glance.

We have seen and have extensively used publishing/subscribing in our applications in the previous chapters. We can publish a query to the clients from the server. The publication can be subscribed from the client. What is so complex in that? Complexity arises when we break down the data to be published in chucks, also known as pagination. It is an interesting problem that can help us understand how exactly publishing and subscription works. Publishing/subscribing involves **Distributed Data Protocol** (**DDP**), MiniMongo, MergeBox, and Trackers.

At the server, we publish a specific set of data. What if that specific set changes because of an insertion or updation from some other client? The publication sends the new data to the necessary clients. This new data is figured out by a process called MergeBox. This new data is sent to the client via DDP. The client receives it and puts it into MiniMongo, and then tracker invalidates all the templates and tracker registered functions. This will make these templates and functions to rerun.

If we subscribe data in chunks, then at a particular instance, the client holds only limited data. When we perform sorting or aggregation, we don't do it on the whole database, but only on the documents available in the client. Always keep this in mind.

Subscriptions are costly. A server maintains a copy of the subscribed data in its primary memory to serve the changes to the client instantly. So use it wisely and don't forget to kill the ones that are not in use, and don't kill those that we need across the application.

# DDP

DDP is a cool socket-based data transferring protocol. MeteorJS chose this protocol as its default way of communication over HTTP because of the latency. DDP can be implemented using WebSockets or long polling techniques. MeteorJS uses Socket.io for implementing DDP.

# MergeBox

It is an intelligent algorithm that can find the difference between the existing and the new data and decide what to send to the client when new data arrives at the database. However, it has a limitation. It works only on top-level fields. For example, you have a document that has a nested structure as follows:

```
{
  name: {
    first: "Paulo",
    last: "Cohelo"
  }
}
```

When there is change in the `first` property that is at the second level, MergeBox will send the complete `name` property to the client. Note this limitation in mind that while designing the database document structure.

# MiniMongo

This is a kind of NoSQL Web SQL. It is an implementation of almost the MongoDB itself in the client. Note that it is almost the same but not the same. MeteorJS has it this way so that querying will be uniform both in the server and the client. When a subscription starts, MeteorJS will create a similar collection with the same document structure in MiniMongo. However, only the subscribed data will be there in MiniMongo.

# Data retrieval from Mongo

Whenever there is a change in the data in the database, the server runs the MergeBox process and finds the diff and sends that over the wire to the clients. As the database grows, the complexity of the diffing process increases. To optimize this, MeteorJS tries to read oplog of Mongo, if provided with the access. If you are in production, it is a must to enable oplog tailing and to provide access to MeteorJS to read it.

# Session

Session is a reactive data source that can hold data even after application refresh. Session is an extension of the reactive-dict and local storage persistence. You can set any EJSON-able/undefined values to session.

# Sticky session

Don't get confused with the session mentioned in the preceding section; sticky session is something to be worried about while deploying your application to the server. It is nothing but, whenever requests come from a particular client, all the requests should be directed to one and the same server. So the case arises when you have a load balancer or a proxy in front of your servers. It is a must to enable sticky session when you have multiple servers involved. Make sure your hosting vendors provide sticky session support before you make the hosting decision.

# Fibers

Fibers is the main underlying differentiator in terms of MeteorJS usage of Node.js. MeteorJS extensively uses fibers for its internal work. Why fibers? Fibers allows you to intercept your operation anytime and allows other operations to use the CPU. Greater control of execution may be the reason of choice. Fibers is very interesting topic that can keep you busy for a while. It will fascinate you. You can get to know more about it at `https://github.com/laverdet/node-fibers`.

# Trackers

Trackers is a simple but very powerful library to rerun some code when there is a change in the related data. MeteorJS uses it in all of its reactive data sources and templates.

# Blaze

The view layer of MeteorJS is Blaze, which is available in the client-side. Blaze is bound with trackers and it provides us the ability to create ultra simple, highly reactive interfaces. Blaze is equivalent to Angular.js, React.js, and so on. With Meteor 1.2, we can use Angular.js and React.js in Meteor.js applications. However, Blaze will still be there in the application even if you don't use it. There is a plan to externalize Blaze so that the user can choose which UI framework to keep in the application.

# Packages

Packages is pretty much everything in MeteorJS. MeteorJS, as a framework, is built using a set of packages. Meteor, Trackers, Mongo, WebApp, underscore, reactive-var, MiniMongo, DDP, Tinytest, and accounts are some of the packages that MeteorJS relies on very much. MeteorJS also provides us the ability to create and use custom packages. Apart from this, we can include NPM packages to in application and Cordova plugins in the case of mobile development.

# Build tools

MeteorJS has state-of-the-art build tools. The build tool can create build for both mobile and Web platforms. It compiles the HTML code to JavaScript code, transpiles ES2015 code using Babel (of Meteor 1.2), combines packages to individual files, creates HTML documents for Web apps, bundles mobile-related code with the build in the case of a mobile, and so on.

These basic blocks together make the mighty MeteorJS framework. MeteorJS as a command-line tool is different; it allows us to work on a terminal to support developing MeteorJS applications.

Keep all these basic blocks in mind so that you can visualize the complete flow of information in your application, which will help you develop and debug applications faster.

# Best practices

We will discuss or summarize the important practices that we have to follow while developing a serious MeteorJS application. The following are some of the best practices and important to-dos for every MeteorJS application:

- Securing database operations
- Database indexing
- oplog support
- Error handling
- Testing
- Managing subscriptions
- Subscribing only the needed data
- Application directory structure
- Serving static assets
- Application namespacing
- Transformation classes
- Identifying scalability issues using Kadira

# Securing database operations

It is needless to say how important it is to verify every piece of data before inserting or updating it in the database. From the beginning, it has been said, don't believe the inputs from the users. We have heard about SQL injection, XSS injections, and many more such kinds of attacks that succeed due to not verifying the data against the necessary validation logic.

Whenever there is access to a database-related operation without any abstraction layer, the developers must be very cautious about the data. It is better to write the database operation code, as if it is defensive to malicious inputs. In the case of SQL, rather than writing raw queries and executing them, it is good to use prepared statements, which will save us, at least from the basic SQL injections. MongoDB doesn't have prepared statements. We always tend to frame a query and pass it to the operation methods (find, update, and remove).

Attackers can try to attack for various purposes. One would be to pull some valuable private information; another would be to slow down the application by keeping the server busy in junk operations; another would be to break down the application itself. NoSQL databases are more vulnerable to perform all these kinds of attacks.

Before performing any database operation, it is important to check whether the person is authorized to do the operation. User A should not be allowed to update user B's profile. As developers, we must think twice before exposing the Users collection to the real world. The Users collection has the e-mail, password, and profile details, so the update or find operation on these data should be wrapped with role-based access.

Let's look at an example here about how to handle the Users collection and what could possibly go wrong. Usually, when we want to update a user document from the application, we expose server methods that will be called from clients with a payload. Then in the server, inside the method, we interpret the payload and frame a query and perform the database operation. Well, this is all good.

For example, our application allows access to view other users' profile, which is common today. Anyone who knows how to watch the data in the network console of a browser can easily make out the user ID of other users. From the client source code, if an attacker finds the server method that updates the Users collection document, it will be easy for him to make a server call with a custom payload targeting update operation on other users' database documents:

```
Meteor.methods({
  updateUser: function(id, profile) {
    Meteor.users.update({_id: id}, {$set: {profile: profile});
  }
});
```

The preceding code snippet explains how one can easily update any other user's profile if only the `_id` value is known. We can make sure that the user is updating only his own profile by mentioning the query in the update method to be `{_id: Meteor.userId()}`. This will work well.

However, if you really want to get the user ID as a parameter, an even more elegant way would be to check whether the ID is equal to the intended ID before doing the operation; something like the code snippet as follows:

```
updateUser: function(id, profile) {
    if(id !== Meteor.userId()) {
      throw new Meteor.Error('Unauthorized');
    }
    Meteor.users.update({_id: id}, {$set: {profile: profile});
  }
```

This case applies for any other entity of database collection that involves `_id`.

Do you find anything else suspicious in this code? Updating the profile object as given in this code is risky, isn't it? It is not good to give the users the control of updating the entire object. What might go wrong here? What if an attacker calls the method with a proper user ID, but with a profile object that has volumes of junk data. The attacker can send any payload with the profile object. What if he tries to send 25 MB of junk string with the profile object? It won't harm the other users, but will eat the memory of the system for no good reason. To avoid this, create a new object and copy only those properties that must be stored before insertion.

The next thing to take care of is, special characters. Having certain special characters in MongoDB queries can break the operation. Characters such as curly braces, back slashes, and semi-colons can create a mess in our queries. So, before storing the inputs, sanitize them using URL encoding. It would be good to check for types, if required. MeteorJS provides the required validation methods to perform various checks over data using the `check` function and the `Match` patterns:

```
check(value, pattern)
check(data, String)
check(office, {building: String, room: Number})
```

The `check` function offered by the framework enables us to do a broad validation over the data. Also, we can define match patterns and test them using the `Match.test` method. You can learn more about these validation functions and methods in the following MeteorJS docs links:

* `http://docs.meteor.com/#/full/check`
* `http://docs.meteor.com/#/full/matchpatterns`

In addition to these, if we are using packages such as `collection2`, which helps us to define the input type and validation logics, the database operations will be more secured.

Another notable query, where many make a mistake, is the `$where` clause. Never ever pass in the argument as the `$where` clause without checking whether it is the intended one. The following is a server method that returns the product's documents based on the query passed in the parameter:

```
Meteor.methods({
  findProducts: function(query, fields) {
    return Products.find({$where: query}, fields);
  }
});
```

Here, we can expect two common attacks. One is, an attacker can pull all the documents from the database even if they don't want to, by just passing the query as `1 == 1`. An other attack is to get all the fields of the document. If the fields parameter has a value to be an empty object, without filtering, an attacker can get all the fields of the document. What if the attacker gets such a method on user collections? Think about the consequence. It can give him access to the hash tokens, password strings, e-mail IDs, and profile details. It is very much necessary to verify the query before using it in conjunction with the `$where` clause.

Mongo queries allow the functions to be executed against documents. So beware not to create any such database operation methods that take a function as an argument to be passed in the query. Another usual attack that is followed in MongoDB is injecting script that keeps the CPU load at 100 percent, thereby making the server process slow. The following query runs a function that takes `userInput` as one of its operands:

```
db.myCollection.find( { active: true, $where: function() { return
  obj.credits - obj.debits < userInput; } } );
```

If anyone can supply `userInput` as the `0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000` value, then the query becomes the statement as follows:

```
db.myCollection.find( { active: true, $where: function() { return
  obj.credits - obj.debits < 0;var date=new Date(); do{curDate =
  new Date();}while(curDate-date<10000); } } );
```

This query will keep the CPU load at 100 percent for 10 seconds. What if the time given is higher? The consequences can be worse.

These are a few common ways to exploit the database. It is better to prevent than to suffer. So think of every possible ways to secure the application database from malicious inputs. Precisely, use defensive coding techniques.

# Database indexing

Indexing database is not something specific to MeteorJS. It is a general thing we do when we create production applications. Indexes help search to be efficient and fast. MeteorJS doesn't explicitly support to create indexes. However, we can create indexes directly in the database. To learn more about indexes in MongoDB, visit the MongoDB documentation at `https://docs.mongodb.org/manual/core/indexes-introduction/#index-introduction`.

If you are familiar with MongoDB indexes, then you might know that MongoDB provides various index types. Choose the one that suits your needs. MeteorJS provides unofficial support to create an index with the `_ensureIndex` method that can be called on collection instances. Watch the underscore prefix in the method, which means that we have to use it at our own risk. Also, `_ensureIndex` can bring downtime to your application because it is a database blocking operation. It is better to do the indexing from MongoDB console.

Let's look at how to check whether the database documents are indexed. MongoDB allows us to see them using the `.explain` method. We can use them as follows:

```
db.users.find({}, {"profile.name": 1}).explain();
```

The preceding query will result back with a BasicCursor. If we have indexed the collections, the explain method will result in BtreeCursor.

MeteorJS concentrates highly on the application logic development. So, as developers, it is our role to take care of indexing the database for better performance.

# oplog tailing

In the previous chapter, we learned how MeteorJS serves data to the client from the server. It makes a difference of the necessary data and then sends only the difference to the client. However, this will be inefficient in the case of large databases. To overcome this problem, if given the access to oplog of the database, MeteorJS can perform the difference calculation very efficiently.

As developers, when moving an application to production, it is important to have oplog tailing support as an important item in the check list. Also, it is required to choose a database vendor who can give oplogs access. If you have an in-house MongoDB setup, then make sure you create the master-slave backup system to ensure oplog is maintained and can be accessed by MeteorJS. We learned about setting up oplog tailing in *Chapter 7*, *Deploying and Scaling MeteorJS Applications*.

# Error handling

In programming, error handling is as equally important as writing application logic. When developing MeteorJS applications, we have to extensively handle the errors. To start with, we have to handle 404 pages. Wrong routes, invalid inputs, network connectivity issues, authorization violations, authentication failures, and so on, are major problems we encounter on a day-to-day basis. As a responsible developer, we have to spend time in properly handling all the earlier mentioned criteria.

MeteorJS provides an error class to define and throw exceptions using the `Meteor.Error` class. The syntax and an example is as follows:

```
new Meteor.Error(error, [reason], [details])
```

An example of throwing excepting when the user is trying to post a comment without logging in into the application, is as follows:

```
throw new Meteor.Error("logged-out",
"The user must be logged in to post a comment.");
```

According to MeteorJS documentation, the only way of communicating errors from the server to the client is via `Meteor.Error`. If we throw any other error, the client will receive it as `500` error. From a server method, if we intend to throw an exception, then we must use the `Meteor.Error` class so that the exception can be transferred to the client on the wire properly. Read more about exception handling at `http://docs.meteor.com/#/full/meteor_error`.

# Testing

Testing our application ensures stability. There are varieties of testing possible, but as a developer, unit tests are the ones we usually care about. We have already learned in *Chapter 2*, *Developing and Testing an Advanced Application* about writing tests using Tinytest and Jasmine. Apart from unit tests, we can write behavior tests or automations using Robot Framework or Cucumber. Velocity is the official test runner for MeteorJS, which can run multiple tests. Make use of velocity and write stable applications.

# Managing subscriptions

We learned many things about subscriptions in the previous chapters. Subscriptions are asynchronous. Every time we call `Meteor.subscribe`, the server creates a cache to keep track of subscribed data. This keeps the server RAM quite busy. For this very reason, we have to stop the subscription once we are done with it. The subscription handle has the `.stop` method with which we can stop the subscription.

Being costly in terms of memory, it is wise to maintain subscriptions as efficiently as possible. Immediately terminate those subscriptions that are not needed anymore and keep those that are required across the page without resubscribing them on every navigation.

Let's look at a common case where developers usually get things wrong. When we are implementing a search, we allow the user to type the query string and then subscribe with the new query every time:

```
if(query) {
  Meteor.subscribe("searchQuery", query);
}
```

We tend to invoke this subscription whenever there is a change in the query. For example, the user has searched for `subscriptions` in the first query. This would have created a new subscription. The next time when he searches `publishing`, again the same preceding code will be invoked and a new subscription will be called. However, what about the previous one? Don't you think it stays there forever? Won't that add additional cache to the server RAM? As the number of searches increases, the application will become slow.

Every time we do such recurrent subscription, it is required to stop the previous subscription using the `stop` method. Inside a Tracker computation, subscriptions are managed this way. It is better to make the query a reactive variable and put it on the tracker.

In the same line, there is a popular subscription pattern called template-level subscriptions. Subscriptions are created within template callbacks and are destroyed when the template is destroyed. There is no need to destroy subscriptions explicitly. On the template instance, we can call the `subscribe` method and also wait until all the subscriptions that are attached to a template are ready. MeteorJS provides the `Template.subscriptionsReady` flag. If a template and subscription has a direct relation, better write it at the template level.

# Publish/subscribe only the necessary data

We have discussed about publishing and subscribing only the necessary data, quite a few times in some of the previous chapters. To reminds ourselves, let's see how to subscribe only what is necessary in a subscription. Whenever we define a publication, it is the best practice to send what has to be sent to the client. Instead of sending the complete document of the collection, send only the necessary fields of the document.

We have Web sockets for data transfer. So why is it so important to filter the necessary fields in publication? Remember that memory issue in the previous section? Every subscription creates a cache in a server's primary memory. If the document size is huge, it is going occupy a certain amount space and it may end up with a serious bottleneck.

Another reason is, why to load the client memory with data that is not necessary for the client? Browsers can hold only a certain amount of data in their memory. Once the limit is reached, the browser will crash. So we have to wisely handle the volume of data that MiniMongo will hold.

Whenever we create a cursor, add a list of fields to filter the query, as much as possible, such as the following query:

```
Meteor.publish('projects', function(type) {
    return MyCollection.find(type, {fields: {
        type: 1,
          content: 1
      }});
});
```

# Application directory structure

There is no hard, defined rule for a directory structure in MeteorJS applications. It is the developers' choice to create the structure to suffice his/her need. It is good to keep in mind how MeteorJS loads the files. We already know that, if there is a server directory, the files inside this directory will be loaded only in the server environment. Similarly, the files inside the client directory will be loaded only in the client environment. A public directory is used for static content and a private directory is used for serving assets to the server. Other than these directories, everything else will be loaded both in the client and server environment.

In the client environment, view the page source of a sample application and you will come to know that the packages are the ones loaded first. Then, the templates are loaded followed by the files, if any, in the `lib` directory. Finally, all the other files are loaded. If the order of loading matters the most, then it is better to write the application as a set of packages. This is because we can define the order of loading files, only in packages.

In general, directory separation increases developers' ability to understand code. Writing all collection instantiation inside the collection directory will make it easy for anyone to check for collection instantiation code there. Code sharing between the client and the server is easy in MeteorJS applications. If we want some collections to be specific to a client only, a few to be specific to a server only, and a few others to be available on both environments, then create multiple collection directories under the client, server, and lib directories and write the code. Routes can also be handled in the same way.

The public directory in MeteorJS is used for static file serving. We will learn about the public directory in the following section. A directory with the name `private`, will be available only in the server environment. The private directory is also a kind of public directory, but only for a server. We can use it to serve assets to the server such as a configuration JSON file or a country code list JSON file, and so on. They can be accessed using the `Assets` API.

> To learn more about the file structure in detail, visit MeteorJS documentation at `https://docs.meteor.com/#/basic/filestructure`.

# Serving static assets

Many developers use the public directory to serve static content, which is the sole purpose of the public directory in MeteorJS applications. While serving static content, we have to think about the underlying truth. A MeteorJS application is ultimately a Node.js application. It has been proven a number of times that Node.js is not good at serving static content. Why is Node.js not performing well at serving static content? The answer to this question is out of the scope of this book. However, Googling a little bit will help you get the answer.

To solve the problem, we have to use a proxy or CDNs to serve the static content. We have already seen in *Chapter 7*, *Deploying and Scaling MeteorJS Applications* how to configure Nginx as proxy. Nginx is good at serving static files too. So, if we add the following configuration to Nginx, static files will be served by Nginx and not MeteorJS:

```
## serve static files by nginx instead of Meteor (the public/
  folder)
location ~ \.(jpg|jpeg|png|gif|mp3|ico|pdf) {
    root /opt/meteor/app/programs/web.browser/app; # this should
    point at the content from the public folder
```

```
        access_log off;
        expires 30d;
        add_header Pragma public;
        add_header Cache-Control "public";
    }
```

It is not necessary to use Nginx only. There are many other alternatives and you are free to use any proxy that you are familiar with.

# Application namespacing

From the beginning, it has been said that global variables are unhealthy. In MeteorJS, we have seen that global variables are created very often. Many have serious problems about using global variables in their MeteorJS applications.

The solution is to namespace. Namespacing is nothing but creating one global variable and then adding every logical module of code as the children of that global variable, thereby avoiding the creation of a large number of global variables. Namespacing has proven to be the best solution for many big applications.

However, is it possible to use namespacing in MeteorJS application? Yes it is, but the usage is very limited. This is because we don't know the order at which the files will be loaded. It is unpredictable whether the required namespaced module will load before the other module that requires it.

A certain portion of the application can be namespaced for sure. Collections and route definitions can be namespaced under the application namespace. However, if we have some logical blocks under the namespace, they are not guaranteed to be available in a proper order.

For those who have problems with global variables and want to avoid them as much as possible, developing the application as a set of packages will solve the problem. In packages, we have the control to export the variable into the environment. Also, there won't be a loading order issue in the case of packages. We will look at how to work with packages in order to avoid global variables in the *Application patterns* section.

# Transformation classes

If we want to manipulate the documents of the collection while fetching, we should use the transform property to define the transformation while creating the collection instance as follows:

```
new Meteor.Collection("products", {
  transform: function(doc) {
    return doc.has_discount = (doc.actual_prize -
    doc.selling_prize) > 0;
  }
});
```

This will ensure that we don't write any transformation logic in the template helpers. We can use the transformation to create models out of each document with helper methods on them.

# Latency compensation

This is one of the important principles of MeteorJS. We discussed in the earlier chapters how MeteorJS saves data and updates the UI. It kind of simulates the save operation and updates the UI at the action triggered side. However, once the data reaches the server and gets saved to the database, a confirmation comes to the action triggered side. If saving fails, the UI will revert back to the old state.

Many see this as a flickering problem. If you consider this kind of latency compensation as a serious problem, because it gives false impression to the user, you can use a different approach to persist data. We have used this approach in many of our examples. We can define server methods that can persist the data instead of calling insert or update directly from the client. We just have to call the server method with the appropriate data. This will avoid the problem of flickering.

Some developers might consider this against the principle of MeteorJS. However, at the end of the day, it is the user's experience that matters the most. So the decision is yours to make.

# Identifying performance and scalability issues

Most of the time, we need a profiling tool to profile and know the performance of the production code. The applications that are used by real users will have a different set of issues than the ones in a testing environment. To monitor the server for performance and scalability issues, we obviously need a tool. MeteorJS doesn't have one by default.

However, one of the MeteorJS community members has created a fantastic tool called Kadira that can help us know the ins and outs of the production application. Kadira helps us to sort out how much time subscriptions take, method call response time, what is CPU usage, and many more in real-time. This provides an in-depth insight into the activities in a production environment. Many developers have identified bottlenecks in their production using Kadira.

Using Kadira in MeteorJS applications is damn easy. Just sign up, create a new app, add the `meteorhacks:kadira` package to your application, and write the connection code `Kadira.connect('<appId>', '<appSecret>')` to a file in the server directory. This is all we need to do. Visit your app in Kadira and you will find the metrics for your application. You can get the `appId` and `appSecret` identifiers from the setting panel of your app that you created in Kadira.

To learn more about Kadira, visit `https://kadira.io/academy/meteor-performance-101/content/getting-started-with-kadira`.

Following all these best practices can help you write a better stable MeteorJS application.

# Application patterns

As MeteorJS is young, there is no definite pattern that we can follow for our application. However, people are testing against their needs and are coming up with various patterns to follow. Here, in this section, we will see two popular patterns that are talked about by the community.

# The package pattern

Let's break the mystery and learn what package pattern is. We know that, by definition, we can build a MeteorJS application using packages. There are applications out there built by developers using this pattern. Why is it that some developers embrace creating an application only by writing their business logic inside packages? To answer this question, we must understand the following:

- What might probably go wrong with the usual way of writing code in a typical MeteorJS application?
- How the packages offer a greater control over our code

# Problems with the usual way of writing MeteorJS code

We discussed some of the problems earlier. Let's look at them again, little more elaborately.

The first and foremost problem is polluting the global space. We know that the usual way of writing MeteorJS applications encouraged writing variables in the global space. We know why it is not good. Anytime it will wreck the ship.

The second problem is code organization. If we are not good at logically separating code with DRY and KISS principles, within a week we will end up in spaghetti code, which will be hard to maintain.

The next problem is testing. When we don't properly organize the logical blocks of our application, then obviously testing the spaghetti code will also be a mess. Code coverage cannot be measured thoroughly.

These three problems are the major one that need to be addressed one way or another way. We have chosen to use the package pattern to escape the aforementioned problems.

# What we must know about packages

If you are not familiar with MeteorJS packages, it is highly recommended that you learn how to create a package and see how it works with MeteorJS from *Chapter 3*, *Developing Reusable Packages* of this very book. Precisely, with packages, we can solve all the problems mentioned in the preceding sections.

Packages provide us the ability to declare all the dependencies inside the package. Packages are reusable and thus we can easily follow the DRY principle. We have control over the loading order of our dependencies. We can write tests for each individual package that is very self-contained. We have the control to export only what we want to export to the client. Did we address all the problems mentioned earlier? Yes, of course, we did. Let's see in action how to create an application with packages, because saying is not doing, isn't it?

To demonstrate the package pattern, we are going to create a small application called DigiNotes. The idea of the application is to provide an interface to write notes and them, legibly. However, the goal is to create the application with custom packages rather than creating it as an application that we created in the past.

# DigiNotes

Create a MeteorJS application with the name `DigiNotes`. Remove all the `hello` template-related code from both the `js` and `html` files, and also remove the `autopublish` and `insecure` packages. Now, we'll have to create packages to start building our application.

With packages, we can write an absolutely modular application, provided we understand how packages work. Packages are added to the DOM in the order we add them to the application. After compiling, MeteorJS creates one file per package, which will include the appropriate content of all the files involved in the package in the order we have specified in the `package.js` file.

Namespacing can help us in writing a maintainable application. In packages, we can create namespaces for each package. If you are good at application design, you will love this approach of creating applications. Application designing is very much opinionated. For example, if we want to create a customer portal, we can modularize it at a granular level of having the add-customer module, edit module, list module, and so on. Some developers might not want that much modularization, instead they will keep the customer portal as one module. It is very much opinionated, and the approach of creating packages is flexible enough to follow any of your ideas.

We are going to create two packages. One package for namespace declaration and the other one for the notes module. Then, you can extend the application as you wish. The following command will create a package with the boilerplate code in our application under the packages directory:

```
meteor create --package <package-name>
```

Make sure you already have the packages directory created in the application root directory. In our case, we are going to create a package for namespacing with the name `diginotes`. We can use this package as the parent for all other packages. The application-wide features such as layouting, application-wide styles, libraries that are used application-wide, and so on, can be added in this package. For now, we are just going to add a namespace for the application.

Run the `meteor create –package diginotes` command. This will create a `diginotes` directory under the `packages` directory. We have to create the namespace for the application. Go to `diginotes.js` and add the following code to it:

```
DigiNotes = {};
DigiNotes.version = "0.1";
```

Visit the `package.js` file and append the following line of code to the `Package.onUse` method:

```
api.export("DigiNotes");
```

We have exported our namespace for the application and other packages to use. In the same file, change the `documentation` property of the `Package.describe` method to `null`, as it is just a local package.

Our parent namespacing package is ready. We have to just add it to the application. Run the `meteor add diginotes` command and once added, run `meteor list` to see the custom package in the list. It will be denoted with the + sign as it is a local package. How do we test whether the namespace is available in the application? Start the application, go to the browser console, and type `DigiNotes.version`. It will print `"0.1"`.

Next, build a package under this namespace, which is nothing but the notes module. Create a package with the name `diginotes:notes` by running the following command:

**meteor create –package diginotes:notes**

Naming the packages with appropriate names is again highly opinionated. Choose names carefully so that any new member to the team can identify the package taxonomy. We have the package skeleton ready once the command is run.

What are the things we need to complete in this package? We need module-specific namespace (`DigiNotes.notes`), collection, pub/sub code, templates, helpers, and stylesheet. The server and client directory separation won't work with packages. Instead, we have to mention the environment for each file in the `package.js` file. For code organization purposes, you can create any folder of your choice. However, our package is pretty small; thus, no folders but only files.

We will keep the namespace code in the `notes.js` file as follows:

```
DigiNotes.notes = {};
```

Though we are not going to use it anywhere now, it is put in the file for demonstration. You can make use of it to add anything, which makes sense for you, into this namespace. We can use it in other packages. For example, if we want to use the collection that we declare in this module in the application or in any other package, then we can attach the instance to this namespace and refer it from anywhere. As of now, it is going to do nothing.

Next, we have to declare the collection. Create the `collection.js` file inside this package, and add the following collection declaration code and access permission code:

```
NotesCollection = new Mongo.Collection("notes");
NotesCollection.allow({
  insert: function() {
  return true;
  }
});
```

We need to publish it, so let's create the `publish.js` file and add the following publication code:

```
Meteor.publish("notes", function() {
    return NotesCollection.find({});
});
```

Do not forget the best practice that we learned earlier this chapter. Apply them to this publication query.

Let's define our templates. Create `template.html` and add the following templates to the file:

```
<template name="notes">
  <form id="note-form">
    <textarea id="note" cols="30" rows="10" placeholder="Note"></
textarea>
    <input type="submit" value="Add" />
  </form>
  <div class="notes">
    {{#if isLoading}}
      Loading...
    {{else}}
      {{#if hasNotes}}
        {{#each notes}}
          {{>note}}
        {{/each}}
      {{else}}
        No notes created.
      {{/if}}
    {{/if}}
  </div>
</template>
```

```
<template name="note">
  <div class="note-container">
    <textarea class="note"
    readonly="true">{{description}}</textarea>
    <div class="controls">
      <button class="edit"> </button>
      <button class="delete"> </button>
    </div>
  </div>
</template>
```

The preceding code displays the notes and provides a form to post notes. There are edit and delete buttons, but it is not functional and is left for you to write some code along.

We need helpers for this template. Create the helpers.js file and add the following helper code to the file:

```
Template.notes.onCreated(function() {
  Template.instance().subscribe("notes");
  this.isLoading = false;
  if(Template.subscriptionsReady) {
    this.isLoading = false;
  }
});
Template.notes.helpers({
  notes: function() {
    return NotesCollection.find({});
  },
  hasNotes: function() {
    return NotesCollection.find({}).count();
  }
});

Template.notes.events({
  "submit #note-form": function(event) {
    event.preventDefault();
    var elem = document.getElementById("note"),
      description = elem.value;
    if(description = description.trim()) {
      NotesCollection.insert({
        description: description,
        createdAt: Date.now(),
```

```
        updatedAt: null
      }, function(e, result) {
        if(!e) {
          elem.value = "";
        }
      });
    } else {
      noty({text: "Please enter some text!", type: "error",
      timeout: 2000});
    }
  }
});
```

In the preceding helpers, we subscribed to the publication inside the template's `onCreated` callback. We also created some helper variables to check whether the subscription is loaded or not. Then, we had helpers to get the count and documents to the template. Now, we bind a submit event in which the note created is inserted to the database. Once inserted, the form is cleared. We are using an external plugin, `noty`, to show the error message.

Finally, add styles that can make the UI look pretty. Create `styles.css` and add the following styles to the file:

```
body {
  display: flex;
  height: 100vh;
  margin: 0;
  padding: 0;
  font-size: 14px;
}

#note-form {
  display: flex;
  flex-direction: column;
  width: 250px;
  max-width: 250px;
  padding: 20px;
  border-right: 1px solid #ccc;
}
#note-form input {
  margin-top: 20px;
  cursor: pointer;
  background: #0ac;
}
```

```css
textarea {
  border-color: #e2e2e2;
  width: 100%;
  padding: 10px;
  border-radius: 4px;
  box-sizing: border-box;
  font-size: inherit;
}
button, input[type="submit"] {
  padding: 10px 20px;
  color: #fff;
  border: 0;
  border-radius: 4px;
  font-size: inherit;
  outline: none;
}

.notes {
  padding: 10px;
}
.note-container {
  width: 200px;
    background: #e3e3e3;
    padding: 10px;
    box-shadow: 1px 1px 1px 1px #ccc;
    border-radius: 5px;
    display: inline-block;
    margin: 10px;
}

.note-container .controls {
  display: flex;
    justify-content: space-around;
}
.note-container .controls > button {
  width: 49%;
  cursor: pointer;
}
.note-container .controls .edit {
  background-color: green;
}
```

```
.note-container .controls .delete {
  color: #fff;
  background: red;
}
.note-container textarea {
  outline: none;
  background: inherit;
  resize: none;
}
```

All the required code is in place. Now it's time to get them to work together. We have to mention each of these files in `package.js`, without which nothing is going to work. Go to `package.js` and add the files we have created.

Add the following piece of code to the `Package.onUse` method:

```
api.use(['ecmascript', 'templating', 'mongo', "hedcet:noty",
  "diginotes@0.0.1"]);
```

Here, we have mentioned the basic packages this package relies on. Templating and Mongo are basic packages, but still, while writing packages, we have to mention them as well. Along with them, we have included the `noty` package and our parent namespace package. This will make sure that before adding the `diginotes:notes` package, the `diginotes` packages are added. Many developers end up in a circular dependency while adding their custom packages. The only way to escape this, is by thinking through your design properly.

Add the rest of the files right next to the preceding line:

```
api.addFiles(['collection.js'], ["client", "server"]);
api.addFiles (['templates.html', 'helpers.js', 'styles.css'],
  ["client"]);
api.addFiles (['publish.js'], ["server"]);
```

You can clearly see that `collection.js` is available both in the server and the client, `templates.html` and `helpers.js` are available only in the client, and `publish.js` is available only in the server. This is how we have to declare which files are to be used on which environments. Now, we are all set to go. Add the package to the application by running the `meteor add diginotes:notes` command and include the required template `{{> notes}}` in the `body` tag of the `DigiNotes.html` file and then we have the application ready.

Start the application and add some notes that will look like the following image:



If you don't want to engage your templates with the application as we did earlier, include layouts. Then, directly from packages, you can add the templates. Another way is to manipulate the DOM. You can add the edit and delete functionality by yourself.

You can extend the application by creating another module, such as reminders, which will allow you to create time-based reminders. It is up to your imagination. This simple example helps you to get started.

While creating an application only with a package, make sure you follow logical terms for naming each and every part of your code. You should be able to draw a flow diagram of your modules at every stage of the application so that you can avoid design-related hassle. Give it a shot if you like this way of developing modular applications. It helps you to increase your design-level thinking too.

# MVC

MVC is a popular design pattern used by most of the application programming languages. If you are using a JavaScript framework, then you must have encountered either MVC directly, or a slight variation of it. MVC is a design concern to organize various pieces of code into models, controllers, and views based on the purpose. Here, in MeteorJS, how can we achieve MVC?

There is no hard rule to define MVC in your application. Some developers want to keep the code clean and organized under different directories in accordance with MVC by creating models, views, and controllers directories, and then organizing code under each of these directories to portray the separation. Some may not need that kind of structure, instead it is good enough if the components are self-explanatory about their purposes.

MeteorJS is of the second category, wherein MVC is defined just by the role or by the purpose of the components we use in the application. Collections are categorized as models, templates and template helpers can be considered as views, and the server methods or the operational logic at the client end can be considered as controllers. Again, this is opinionated. If you are using iron-router, the router component plays the controller role of executing business logic and thereby employing appropriate templates and also updating the models.

If you really want MVC defined by the separation of directories, then MeteorJS doesn't restrict us from doing so. We are free to create any directory, but by adhering to the rules (the server and client directory environment separation) of MeteorJS.

# SEO

SEO is an important topic that we haven't discussed in any part of this book. Let's look at it comprehensively in this section. MeteorJS is an ideal choice for interactive applications. However, it still doesn't restrict us to use it for applications that need to be indexed for search engines. One such example would be online tutorial publishing applications. These kinds of applications need SEO the most. SEO can be a deciding factor while choosing which framework to go with.

MeteorJS applications are more or less like single-page applications in terms of loading the HTML. If you look at the page source in the browser, you will see only an empty body tag. The reason being that the HTML is loaded dynamically. How then can we provide content, appropriate content, to the search engines that try to index our application pages? MeteorJS provides a solution to these problems.

However, before looking at the solution, let us know how exactly indexing is carried out by search engines. Search engines find your Web pages by means of references and then send their bots, also called crawlers or spiders, to get the content of our Web page. Spiders simply hit the page and collect whatever is loaded as a result. The collected content is then read with various ranking rules and is ranked by the search engine.

What is more important for a Web developer than to get his/her pages indexed properly by the search engine? When the spiders come crawling to our Web pages, we have to show the pages to them nicely. Apart from the content, there are a few other things we want to set in the page such as title, meta description, and so on, for SEO purposes. To learn more about indexing dynamically loaded sites, read the Google developers documentation at `https://developers.google.com/webmasters/ajax-crawling/docs/getting-started?hl=en`.

There are two ways of allowing the crawlers to know that our site content has dynamically loaded the content. One is by prepending an exclamation mark (!) before the hash part such as `https://example.com/a/site#!key=value`. If we don't use the hash part in our routes, then we can include a meta tag `<meta name="fragment" content="!">` in the page, which will tell the crawler to look for an HTML snapshot of the page at the specified URL: `https://example.com/a/site?_escaped_fragment_=key=value`. We have to make sure that, when the crawler visits the pages with this URL, we provide the content nicely.

# Spiderable

Considering that MeteorJS applications are dynamic applications that load the templates on demand, how are we going to nicely show our pages to the crawlers or spiders? Here, MeteorJS employs a small trick. It gives us the ability to load the HTML-like traditional applications, but only for the spiders. How is that possible? There is a package called spiderable that will get the appropriate content from the server when the spiders come crawling to our pages. We have to add the spiderable package to our application. Along with it, we have to install PhantomJS. When the crawlers hit the page with the query `string _escaped_fragment_`, a spiderable package will take the request and render that page in PhantomJS, which is a headless browser and will present that content to the crawler. This solves the problem. You can test it by visiting your site in the browser with the `_escaped_fragment_` query string. You will see the HTML content inside the body that was empty earlier.

What about setting the meta tags based on the page? If you are using iron-router, the job is pretty simple. Iron-router provides `onBeforeAction` hooks in which we can set the required meta information to the page. There is a popular package that comes in handy to set meta data to the page that is ms-seo. To learn more, visit the GitHub repository of the package at `https://github.com/DerMambo/ms-seo`.

Few people have problems with PhantomJS. The problem could be reasonable. When the crawler visits multiple pages of your application at once, MeteorJS will spawn a separate instance of PhantomJS for each request that can keep the CPU terribly busy. Other problems are, hard to debug if something goes wrong in PhantomJS with respect to your application; PhantomJS may not be available in certain cloud hostings, and PhantomJS runs a bit older version of JavaScript. We can see that the problems are reasonable and thus we need an alternative. Is there one?

Yes, but highly custom and slightly time-consuming; maybe even violating the DRY principle. The solution is to have server-side routes using some packages of your interest and then having the templates in the server-end, compiling and responding as per the request. We cannot use the client-side templates in the server. For this reason, we have to use a package such as `meteorhacks:ssr` for compiling the templates. It is better to keep the templates in the private directory and get them using `Assets.getText` (`"home.html"`). There are many routers out there. Pick one that supports the server-side route and then you are ready to go. This approach is actually called server-side rendering.

There is a popular package called fast-render that helps in server-side rendering, but it doesn't add to SEO. It helps to reduce the load time of your application. It works with both Iron-router and `FlowRouter`, seamlessly. To learn more about fast-render, visit the GitHub repository at `https://github.com/kadirahq/fast-render`.

Try whichever SEO support mechanism you like from the above two methods and see how it works. If anyone claims MeteorJS cannot support SEO, tell them how it can be done.

# ES2015 and MeteorJS

ES2015 is on the move. If you are a JavaScript developer, by this time, you might have tasted some of the drops of the new features coming to JavaScript. There are transpilers, such as Babel that allows us to write ES2015 code right away. MeteorJS also has extended the support for ES2015 features. No special setup is needed and the support is added from Meteor 1.2. This is a major milestone for MeteorJS. Having access to an environment where you can write ES2015 code, you can learn it very easily without having to do any setup explicitly. There is no more reason to avoid experimenting with the ES2015 features within the application that you are building. Make use of it and learn them faster.

# Meet the community

It is to be noted that a framework, that is so young, has a very big community. The MeteorJS community is so big that you will find answers to most of your questions quickly. Being an awesome framework, no wonder MeteorJS has got so many followers. The number of packages in the `https://atmospherejs.com` clearly says that people are very much engaged to get this framework to its heights. In this section, we are going to see what sites to follow, where to seek help, and whom to follow in order to learn more about MeteorJS.

First and foremost, anyone having any programming-related problems will visit Stack Overflow. MeteorJS also has a Meteor tag in Stack Overflow where numbers of questions are posted and are answered too. Some of these answers will help you tune your MeteorJS knowledge. Visit Meteor-specific questions in Stack Overflow at `http://stackoverflow.com/questions/tagged/meteor`.

MeteorJS has its own discussion forum too, and you can visit it at `https://forums.meteor.com/`. Similar to Stack Overflow, one can ask questions in this forum. One can also discuss the core features of MeteorJS, what MeteorJS is lacking, what is coming up in the releases, if someone has faced any strange issues, and how they had solved it. People also talk a lot about packages, integration with other frontend frameworks such as React.js, Angular.js, testing applications, hosting solutions, and many more. Get involved and see what you can learn from the community members.

Atmosphere, which is the MeteorJS package repository, is another important site that every MeteorJS developer must visit. It is not necessary to explain; we have visited it many times to search for packages at `https://atmospherejs.com/`. To consume or to publish a MeteorJS package, we must visit `https://atmospherejs.com`.

Meteorhacks (`https://meteorhacks.com/`) is a fantastic site to learn many things about MeteorJS. The articles posted in this site are specific to MeteorJS and are quite simple and well-explained. The site has very interesting articles that have good explainations of the subscriptions, MergeBox, hosting solutions, and so on.

Another excellent learning material for MeteorJS is available at `https://bulletproofmeteor.com`. This is an interactive tutorial site where a part of the content is free and the rest is premium; it is definitely worth the money.

Meteorpedia is a site where we can look for solutions for various MeteorJS-related problems. The site has a number of articles where many aspects of the framework are discussed. To view this, visit `http://www.meteorpedia.com/read/Main_Page`.

Many like to watch than to read. For them, there is a YouTube channel that has a lot of video content to learn about MeteorJS. Many enthusiastic developers who have done wonders with MeteorJS have shared their experiences. Visit the YouTube channel at `https://www.youtube.com/user/MeteorVideos`.

If you are one of those enthusiasts who likes to visit people from the community, we have meetups. The MeteorJS team has organized a number of meetups in the past and is still doing a few here and there. Get to know about the happenings at `http://meteor.meetup.com/`.

Twitter is the best source to keep ourselves updated. MeteorJS' official tweet handle is `@meteorjs`. You can follow the Twitter account at `https://twitter.com/meteorjs`. Every official news about MeteorJS can be found here. You can even raise your concerns with the MeteorJS development team via this Twitter page.

Contributors are those people who have gained significant experience and are giving back the learnings to the community. MeteorJS has contributors who have spent time to give their knowledge to the community. A few other contributors have contributed by creating packages for fellow community developers. There are so many contributors and we'll see a few. One of the notable community members to follow is `@arunoda`, the one behind Kadira, who experimented with MeteorJS to its extreme and contributed to the community in terms of packages and learning materials. Josh Owens is another member who has wide experience in building MeteorJS applications. Follow him at `@joshowens`. He has his blog at `http://joshowens.me/` where he writes about his experiences with MeteorJS. There are a number of articles and videos published by Josh Owens to teach community members. There are many such significant members; maybe you can become one such notable contributor too someday.

Some of the notable applications built using MeteorJS are as follows:

- Atmosphere (`https://atmospherejs.com`)
- Kadira
- Telescope (`www.telescopeapp.org`)
- BulletProofMeteor (`www.bulletproofmeteor.com`)
- Meteorpedia

> There are many applications already in the market that have been built using MeteorJS. The official documentation of MeteorJS (`http://docs.meteor.com/#/full/`) is a good source to learn more about MeteorJS.

Finally, the Meteor GitHub page is the source code. What lies there? The truth itself. Whenever you get time and if you are interested, visit meteor GitHub page at `https://github.com/meteor/meteor/` and learn the internals by yourself. The packages used by MeteorJS can be reached at `https://github.com/meteor/meteor/tree/devel/packages`. You will learn to write MeteorJS code in the MeteorJS way, if you start understanding the source. Nothing can tell us the truth more than the source code itself.

# Summary

Finally, we are at the end of this chapter and also at the end of this book. Let's summarize what we have learned in this chapter.

As MeteorJS is a young framework, there are not many specific patterns or best practices to follow, just the fundamental software principles.

There are some tried and tested best practices that might work for a few and might not for some others. They are no hard rules, but the best practices can be bent if they were to yield what exactly we are looking for.

There is a not-so-popular application pattern for MeteorJS applications, which is the package pattern. MVC is again available, but it is in an arguable state. So make use of MVC as per your understanding and need.

MeteorJS provides ES2015 support out of the box now from its 1.2 release.

The MeteorJS community is vast and there are many exclusive sites where we can look for help and learning materials.

There are a number of promising applications being built using MeteorJS. It is time to use it for your next application too.

I hope you have found the chapter useful and informative.

We are at the finishing line of the book. Now, you must be confident enough to create products using MeteorJS because you know enough to develop, deploy, and scale a MeteorJS application. Practice more and share your experience with the community. There is a lot to learn if you are a beginner, and it is absolutely worth it. MeteorJS has the ability to change how we develop applications.

MeteorJS is driving the change in how we build Web and Web-based applications. Be a part of the change and do wonders that can make the world a better place to live.

# Index

**Thank you for buying**
# Mastering MeteorJS Application Development

## About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

## About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

## AngularJS Web Application Development Cookbook

ISBN: 978-1-78328-335-4          Paperback: 346 pages

Over 90 hands-on recipes to architect performant applications and implement best practices in AngularJS

1.  Understand how to design and organize your AngularJS application to make it efficient, performant, and scaleable.

2.  Discover patterns and strategies that will give your insights into the best ways to construct production AngularJS applications.

3.  Get the most out of AngularJS by gaining exposure to real-world examples.
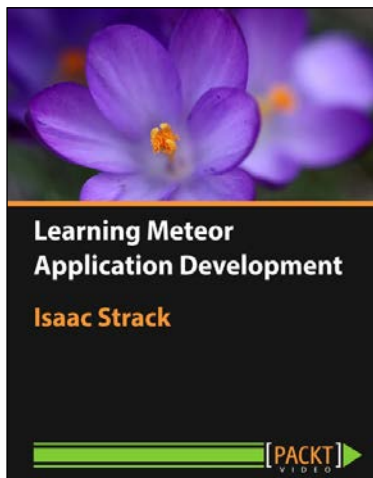
## Learning Meteor Application Development [Video]

ISBN: 978-1-78439-358-8          Duration: 01:52 hours

An informative walkthrough for creating a complete, multi-tier Meteor application from the ground up

1.  Master the fundamentals for delivering clean, concise Meteor applications with this friendly, informative guide.

2.  Implement repeatable, effective setup and configuration processes and maximize your development efficiency on every project.

3.  Utilize cutting-edge techniques and templates to reduce the complexity of your applications and create concise, reusable components.

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