Mastering MeteorJS Application Development

Mastering MeteorJS Application Development shows you how to do even more with MeteorJS — if you’re ready to try a comprehensive course through one of the most exciting frameworks in web development today, this is the book you need.

You’ll not only learn how MeteorJS makes Web development easier, but also how you can make using MeteorJS easier by automating and simplifying tasks so you can be confident you have full control of everything in your workflow — especially everything that could go wrong. With further insights on developing for mobile — and how MeteorJS can help you tackle the challenges of the trend — and details on incorporating SEO strategies into your application, this book isn’t just a code tutorial — it’s about creating a product that users love.

Who this book is written for

If you’ve already had some experience with MeteorJS but want to learn how to build even better modern web applications, this book has been created for you. It provides you with a comprehensive look at one of the most important frameworks being used on the web today.

What you will learn from this book

- Get to grips with the basics and learn how to build a complete real-time application with MeteorJS
- Find out how MeteorJS makes full-stack development simple — become a better developer, fast
- Find out how to write custom packages for applications — so you can build your project exactly how you want
- Integrate React.js and Angular.js into your project
- Design and develop high quality animations that will give your UI the edge
- Build MeteorJS to serve as a REST-based application and reactive system
- Learn how to host a MeteorJS application and then scale it for data
- Learn how to make sure you implement an effective SEO strategy in your MeteorJS application

Jebin B V

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.


$ 44.99 US
£ 28.99 UK

Prices do not include local sales tax or VAT

Community Experience Distilled
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Building a Meteor.JS Web Application'
- A synopsis of the book’s content
- More information on Mastering Meteor.JS Application Development
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.
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.
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:

```javascript
/* 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:

```javascript
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:

```html
<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-single-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` 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` 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 as follows:

```javascript
Meteor.startup(function () {
  smtp = {
    username: '', // eg: bvjebin@meteorr.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:

```javascript
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';
  }
  Accounts.emailTemplates.verifyEmail.text = function(user, url) {
    return 'click on the following link to verify your email address: ' + url;
  }
});
```
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](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`, 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`:

```javascript
("/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:

```javascript
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">
        {{#linkTo route="home"}}List{{/linkTo}}
      </ul>
    </header>
    <section class="create-container__section">
      {{> yield}}
    </section>
  </div>
  <footer class="footer">Copyright ©Packt</footer>
</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:

```html
<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>
```
Chapter 1

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;}
.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;}
```
Building a MeteorJS Web Application

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:

![Image of the form with styles applied](image)

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:

```javascript
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:

```javascript
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:

```javascript
Router.route("/", {
  name: "home",
  layoutTemplate: "homeLayout",
  template: "home",
  yieldRegions: {
    travelSearch: {to: "search"}
  }
});
```
Building a MeteorJS Web Application

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:

```html
<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):

```html
<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>
  </div>
</template>
```
<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>
</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">
<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="">
</form>
</div>
</template>
Building a MeteorJS Web Application

```html
<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:

```javascript
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:

```javascript
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:

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

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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()  
    };```
Building a MeteorJS Web Application

```javascript
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:

```javascript
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:

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

[24]
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:

```javascript
Template.bookTravel.helpers({
  seatArrangement: function() {
    var arrangement = [],
        totalSeats = (this.bus || {}).seats || 0,
        blockedSeats = _.map(this.blockedSeats || [], function(item) {
          return item.seat},
        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 & 1 != 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:

```javascript
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:

```javascript
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.
Where to buy this book

You can buy Mastering MeteorJS Application Development from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.