Community Experience Distilled

Mastering Backbone.js

Abiee Echamea

Backbone.js is a popular library for building single page applications and is used by many start-ups around the world because of its flexibility, robustness, and simplicity. It allows you to bring your own tools and libraries so you can make amazing web apps with your own rules.

With this book you will start with organizing your Backbone.js application so you know where to put each module and how to connect them. From organizing your code in a logical and physical way, you will go on to delimiting view responsibilities and working with complex layouts. The following chapters will help you to manage module dependencies in your projects, explore strategies to upload files to a RESTful API, and store information directly in the browser for use with Backbone.js. After testing your application, you will be ready to deploy it to your production environment. The final chapter will cover different flavors of authorization.

Who this book is written for

This book is for those developers who know the basic concepts of Backbone.js and want to build scalable applications with it. If you are looking for the best practices of using Backbone.js, applied to real-world applications, this book is for you. You will be able to apply architectural principles to create amazing web applications easily.

What you will learn from this book

- Build web applications that scale with Backbone.js
- Design a powerful architecture that eliminates maintenance nightmares
- Use common patterns and best practices in Backbone.js web application development
- Pack your applications so they can be deployed to production environments
- Clean up your code organization with a simple and maintainable architecture
- Test your components and gain confidence in your code
- Deal with common scenarios such as file uploading and login issues

Who this book is written for

This book is for those developers who know the basic concepts of Backbone.js and want to build scalable applications with it. If you are looking for the best practices of using Backbone.js, applied to real-world applications, this book is for you. You will be able to apply architectural principles to create amazing web applications easily.

Prices do not include local sales tax or VAT

$ 49.99 US
£ 31.99 UK

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Architecture of a Backbone application'
- A synopsis of the book’s content
- More information on Mastering Backbone.js
About the Author

**Abiee Echamea** is a passionate technology software engineer; he fell in love with computers at the age of 8 and wrote his first program at 12. He has written applications in many technologies, from Visual Basic to full stack web applications. Now, Abiee is a software architect involved in the full development cycle, creating many Backbone applications successfully and designing the architecture and development process. He has founded a company to work with cutting-edge technology as a CTO. He has developed the skills to master the JavaScript and Backbone libraries to build maintainable projects for his company. Abiee loves to share his knowledge with the community and is looking for ways to improve his engineering skills.
Preface

Backbone is an amazing library to build web applications; it's small, simple, and yet powerful. It provides a set of small and focused objects to be used as bricks when building frontend applications.

The beauty of Backbone is that it gives you the freedom to build your applications with your rules. However, with great power comes great responsibility; Backbone does not tell you anything about how to structure your applications. Keep in mind that Backbone is not a framework but a library.

After years of working with Backbone projects, making code experiments, and exploring code from other developers, I have identified patterns and best practices when building frontend web apps with Backbone.

This book explains how to give structure to your applications. It gives you the tools and strategies to create robust and maintainable web apps. It will help you define and assign the right responsibilities to the Backbone objects and define a new set of glue objects.

In the book, you will build a functional application applying the concepts that are exposed here. The application is simple enough to put in to practice the core concepts when building scalable frontend applications with Backbone. At any time, you can see the project code in the book repository at https://github.com/abiee/mastering-backbone.

What this book covers

Chapter 1, Architecture of a Backbone application, deals with the project organization at two levels: logical and physical. On the logical side, you will learn how to connect the Backbone objects, while on the physical side, you will see where to put your scripts.
Preface

Chapter 2, *Managing views*, helps you extract the common patterns of views and create a new set of general purpose views that can be used on any Backbone application. These views will remove a lot of boilerplate code when managing views.

Chapter 3, *Model bindings*, explains how to deal with complex REST resources and helps you handle embedded resources and keep it in sync with views.

Chapter 4, *Modular code*, covers dependency management and script bundling with Browserify. Modern applications are becoming more JavaScript-intensive, so it's a good idea to handle dependencies in a smarter way.

Chapter 5, *Dealing with files*, it covers the common requirement for web applications to upload files to a server, this chapter tells you how to do it in Backbone with a REST server.

Chapter 6, *Store data in the browser*, shows you how to store data in the browser and how to do it from a Backbone perspective. The chapter shows how to build two drivers to transparently store Backbone models in localStorage and indexedDB instead of a remote server. This can be useful if you want to create offline applications.

Chapter 7, *Build like a pro*, tells you how you can automatize common and repetitive tasks in a script. It will dramatically improve your productivity. It describes how you can build a development workflow that automatically refreshes your project every time you make a small change.

Chapter 8, *Testing Backbone applications*, shows you the strategies and best practices when testing frontend code.

Chapter 9, *Deploy to production*, shows you how to deploy the project to a production server. While high-demand applications need a sophisticated platform, this chapter gives you the starting point for small apps.

Chapter 10, *Security*, teaches you how to authenticate against the REST servers and how to manage sessions from the Backbone side.
One of the best things about Backbone is the freedom to build applications with the libraries of your choice, no batteries included. Note that Backbone is not a framework but a library; due to this, building applications with Backbone can be challenging as no structure is provided. You, as a developer, are responsible for code organization and how to wire the pieces of the code across the application; it's a big responsibility. Bad decisions can lead to buggy and unmaintainable applications that nobody wants to work with.

Code organization on small Backbone applications is not a big deal. Create a directory for models, collections, and views; put a router for all possible routes; and write the business logic directly in the views. However, this way of developing Backbone applications is not suitable for bigger projects. There should be a better way to separate responsibilities and file organization in order to create maintainable applications.

This chapter can be difficult to understand if you don't know Backbone at all; to understand the principles that are exposed here better, you will need to understand at least the basics of Backbone. Therefore, if you are a beginner in Backbone, I would encourage you to first understand what Backbone is and how it works.

The goal of this chapter is to explore the best practices of project organization on two main levels: logic organization and file structure. In this chapter, you will learn the following:

- Delegating the right responsibilities to the objects provided by Backbone
- Defining plain JavaScript objects in order to deal with logic out of scope of Backbone objects
- Splitting the application into small and maintainable scripts
- Creating a clean file structure for your projects
Subapplications based architecture

We can compose a Backbone application with many independent subapplications. The subapplications should work independently. You can think about each one as a small Backbone application, with its own dependencies and responsibilities; it should not depend on other subapplications directly.

Subapplications should be focused on a specific domain area. For example, you can have a subapplication for invoices, another for the mailbox, and one more for payments; with these subapplications in place, you can build an application in order to manage payments through email.

To decouple subapplications from each other, we can build an infrastructure application responsible for managing the subapplications, bootstrapping the whole application, and providing the subapplications with common functions and services:

![Diagram of Backbone application with subapplications]

Figure 1.1. Composition of a Backbone application with subapplications

You can use the infrastructure application to provide your subapplications with services such as confirmation and dialog messages, notification pop-ups, modal boxes, and so on. The infrastructure application does nothing by itself; it behaves as a framework for the subapplications.

When a subapplication wants to communicate with another subapplication, the infrastructure application can be used as a communication channel, it can take advantage of the Backbone.Event object in order to send and receive messages.

In the following figure, you can see a scenario where the subapplications communicate through the infrastructure application. When the user clicks on Compose message in the Mailbox subapplication, the infrastructure application creates and renders the Compose mail subapplication and allows the user to write an e-mail.
When the user is done, they have to click on the **Send** button in the **Compose** subapplication; then the e-mail is sent through a RESTful API or using plain SMTP, don't care, the important thing is that, when it finishes, it triggers an event in the **email:sent** infrastructure application.

The infrastructure application forwards the event to the Mailbox subapplication, so that the list of emails that are sent can be updated. Another interesting thing is that the infrastructure application can use the **email:sent** event to show a successful pop-up message to the user to tell them that the email was successfully sent:

![Figure 1.2. Communication between subapplications](image)

**Subapplication anatomy**

As mentioned earlier, a subapplication is like a small Backbone application; they should be independent of other subapplications and work as a standalone. You should be able to put the Compose mail subapplication on a blank page without any other subapplication and still be able to send emails.

To achieve this, the subapplications should contain all the necessary objects that are to be auto-contained. You can see that the entry point of the subapplication is **Backbone.Router**. When the browser changes the URL and a route is matched for a given subapplication, the router creates a subapplication controller and delegates it the route handling.

The subapplication controller coordinates the models/collections and how they are shown. The controller can instruct the **Application** infrastructure to show a loading message while the data is fetched and when it's done, the controller can build the necessary views with the models and collections that are recently fetched in order to show them in the DOM.
In short, a subapplication behaves exactly like a small Backbone application, with the main difference being that it uses the Application infrastructure to delegate common tasks and a communication channel between the subapplications.

In the next sections, we will examine how these parts are connected and I will show you the code for a working Contacts application. The following figure shows an anatomy view of a subapplication:

![Diagram of subapplication architecture]

**Figure 1.3. Anatomy of a subapplication**

**Responsibilities of Backbone objects**

One of the biggest issues with the Backbone documentation is not to have a clue about how to use its objects. You, as developers, should figure out the responsibilities for each object across the application; if you have some experience working with Backbone, then you would know how difficult it would be to build a Backbone application.
In this section, I will describe the best uses of the Backbone objects. Starting at this point, you will have a clearer idea about the scope of responsibilities in Backbone and this will lead the design of our application architecture. Keep in mind that Backbone is a library with only the foundation objects; therefore, you will need to bring your own objects and structure to make scalable, testable, and robust Backbone applications.

**Views**

The only responsibilities of views are to handle the Document Object Model (DOM) and listen for low-level events (jQuery/DOM events), and transform them into domain ones. The Backbone Views works closely with template engines in order to create markups that represent the information that is contained in models and collections.

Views abstract the user interactions, transforming their actions into business value data structures for the application. For example, when a click event is triggered from a Save button in the DOM, the view should transform the event into something similar to a `save:contact` event using Backbone Events with the data written in the form. Then, a domain-specific object can apply some business logic to the data and show a result.

It is a rule that business logic on views should be avoided; however, basic form validations such as accept only numbers are allowed. Complex validations should still be done on the model or the controller.

**Models**

Backbone Models are like database gateways in the server side, their main use is to fetch and save data to and from a RESTful server and then provide an API to the rest of the application in order to handle the information. They can run general-purpose business logic, such as validation and data transformation, handle other server connections, and upload an image for a model.

The models do not know anything about views; however, they can implement functionality that is useful for views. For example, you can have a view that shows the total of an invoice and the invoice model can implement a method that does the calculation, leaving the view without knowledge of the computation.
Collections
You can think of Backbone Collections as a container of a set of Backbone Models, for example, a Collection of Contacts models. With a model, you can only fetch a single document at time; however, Collections allow us to fetch lists of Models.

A big difference from Models is that Collections should be used as read-only, they fetch the data but they should not write in the server; also it is not usual to see business logic here.

Another use for Collection is to abstract RESTful APIs responses as each server has different ways to deal with a list of resources. For instance, while some servers accept a skip parameter for pagination, others have a page parameter for the same purpose. Another case is on responses, a server can respond with a plain array, while others prefer to send an object with a data, list, or other key, where the array of objects is placed. There is no standard way. Collections can deal with these issues, making server requests transparent for the rest of the application.

Routers
Routers have a simple responsibility: listening for URL changes in the browser and transforming them into a call to a handler. A router knows which handler to call for a given URL. Also, they have to decode URL parameters and pass them to the handlers. The infrastructure application bootstraps the application; however, routers decide which subapplication will be executed. In this way, routers are a kind of entry point.

Objects not provided by Backbone
It is possible to develop Backbone applications only using the Backbone objects that are described in the previous section; however, for a medium-to-large application, it's not sufficient. We need to introduce a new kind of object with delimited responsibilities that use and coordinate Backbone foundation objects.

Subapplication façade
This object is the public interface of the subapplications. Any interaction with the subapplications should be done through its methods. The calls made directly to internal objects of the subapplication are discouraged. Typically, methods on this controller are called from the router; however, they can be called from anywhere.
The main responsibility of this object is to simplify subapplication internals. Its main work is to fetch data from the server through models or collections and, if an error occurs during the process, it is responsible to show an error message to the user. Once the data is loaded in a model or collection, it creates a subapplication controller that knows the views which should be rendered and have the handlers deal with its events.

**Subapplication controller**

A controller acts like an air traffic controller for views, models, and collections. When given a Backbone data object, it will instantiate and render the appropriate views and then coordinate them. On complex layouts, it is not an easy task to coordinate the views with the models and collections.

The Business logic for the use cases should be implemented here. The subapplication controller implements a **mediator pattern**, allowing other basic objects such as views and models keep it simple and loose coupling.

Due to loose coupling reasons, a view should not directly call to methods or events of other views Instead of this, a view triggers events and the controller handles the event and orchestrates the views behavior if necessary. Note how views are isolated, handling just its owned portion of DOM and triggering events when required to communicate something.

**Contacts application**

In this book, we will develop a simple contacts application in order to demonstrate how to develop Backbone applications following the principles explained throughout this book. The application should be able to list all the available contacts in RESTful API and provide the mechanisms to show and edit them.
The application starts when the Application infrastructure is loaded in the browser and the `start()` method on it is called. It will bootstrap all the common components and then instantiate all the available routers in the subapplications:

```
// app.js
var App = {
    Models: {},
    Collections: {},
    Routers: {},
    start() {
        // Initialize all available routes
        _.each(_.values(this.Routers), function(Router) {
            new Router();
        });

        // Create a global router to enable sub-applications to
        // redirect to other urls
        App.router = new DefaultRouter();
        Backbone.history.start();
    }
}
```
The entry point of subapplication is given by its routes, which ideally share the same namespace. For instance, in the contacts subapplication, all the routes start with the `contacts/` prefix:

- **Contacts**: This lists all available contacts
- **contacts/new**: This shows a form to create a new contact
- **contacts/view/:id**: This shows an invoice given its ID
- **contacts/edit/:id**: This shows a form to edit a contact

Subapplications should register its routers in the `App.Routers` global object in order to be initialized. For the Contacts subapplication, the `ContactsRouter` does the job:

```javascript
// apps/contacts/router.js
'use strict';

App.Routers = App.Routers || {};  

class ContactsRouter extends Backbone.Router {
    constructor(options) {
        super(options);
        this.routes = {
            'contacts': 'showContactList',
            'contacts/page/:page': 'showContactList',
            'contacts/new': 'createContact',
            'contacts/view/:id': 'showContact',
            'contacts/edit/:id': 'editContact'
        };
        this._bindRoutes();
    }

    showContactList(page) {
        // Page should be a positive number greater than 0
        page = page || 1;
        page = page > 0 ? page : 1;
        var app = this.startApp();
        app.showContactList(page);
    }

    createContact() {
        var app = this.startApp();
    }
```
When the user points its browser to one of these routes, a route handler is triggered. The handler function parses the URL and delegates the request to the subapplication façade:

![Diagram of route delegation to Subapplication Façade](image)

**Figure 1.5. Route delegation to Subapplication Façade**
The `startSubApplication()` method in the `App` object starts a new subapplication and closes any other subapplication that is running at a given time, this is useful to free resources in the user's browser:

```javascript
var App = {
    // ...
    // Only a subapplication can be running at once, destroy any
    // current running subapplication and start the asked one
    startSubApplication(SubApplication) {
        // Do not run the same subapplication twice
        if (this.currentSubapp &&
            this.currentSubapp instanceof SubApplication) {
            return this.currentSubapp;
        }
        // Destroy any previous subapplication if we can
        if (this.currentSubapp && this.currentSubapp.destroy) {
            this.currentSubapp.destroy();
        }
        // Run subapplication
        this.currentSubapp = new SubApplication({
            region: App.mainRegion
        });
        return this.currentSubapp;
    },

    // 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.

    // The `App.mainRegion` attribute is an instance of a `Region` object that points to a DOM
    // element in the page; regions are useful to render views in a contained region of the
    // DOM. We will learn more about this object in Chapter 2, Managing views.
```
When the subapplication is started, a façade method is called on it to handle the user request. The responsibility of the façade is to fetch the necessary data from the RESTful API and pass the data to a controller:

![Diagram of a Backbone application architecture](image)

```javascript
// apps/contacts/app.js
'use strict';

class ContactsApp {
  constructor(options) {
    this.region = options.region;
  }

  showContactList() {
    App.trigger('loading:start');
    App.trigger('app:contacts:started');

    new ContactCollection().fetch({
      success: (collection) => {
        // Show the contact list subapplication if
        // the list can be fetched
        this.showList(collection);
        App.trigger('loading:stop');
      },
      fail: (collection, response) => {
        // Show error message if something goes wrong
        App.trigger('loading:stop');
      }
    });
  }
}
```
showNewContactForm() {
    App.trigger('app:contacts:new:started');
    this.showEditor(new Contact());
}

showContactEditorById(contactId) {
    App.trigger('loading:start');
    App.trigger('app:contacts:started');

    new Contact({id: contactId}).fetch({
        success: (model) => {
            this.showEditor(model);
            App.trigger('loading:stop');
        },
        fail: (collection, response) => {
            App.trigger('loading:stop');
            App.trigger('server:error', response);
        }
    });
}

showContactById(contactId) {
    App.trigger('loading:start');
    App.trigger('app:contacts:started');

    new Contact({id: contactId}).fetch({
        success: (model) => {
            this.showViewer(model);
            App.trigger('loading:stop');
        },
        fail: (collection, response) => {
            App.trigger('loading:stop');
            App.trigger('server:error', response);
        }
    });
}

showList(contacts) {
    var contactList = this.startController(ContactList);
}
Architecture of a Backbone application

contactList.showList(contacts);
}

showEditor(contact) {
    var contactEditor = this.startController(ContactEditor);
    contactEditor.showEditor(contact);
}

showViewer(contact) {
    var contactViewer = this.startController(ContactViewer);
    contactViewer.showContact(contact);
}

startController(Controller) {
    if (this.currentController &&
        this.currentController instanceof Controller) {
        return this.currentController;
    }

    if (this.currentController &&
        this.currentController.destroy) {
        this.currentController.destroy();
    }

    this.currentController = new Controller({
        region: this.region
    });
    return this.currentController;
}

The façade object receives a region object as argument in order to indicate to the
subapplication where it should be rendered. The Region objects will be explained in
detail in Chapter 2, Managing views.

When the façade is fetching data from the RESTful server, a loading:start event is
emitted on the App object in order to allow us to show the loading in progress view
for the user. When the loading finishes, it creates and uses a controller that knows
how to deal with the model or fetched collection.
The business logic starts when the controller is invoked, it will render all the necessary views for the request and show them to the user, then it will listen for user interactions in the views:

![Diagram](image)

Figure 1.7. Controller creates the necessary views

For the `ContactList` controller, here is a very simple code:

```javascript
// apps/contacts/contactLst.js
class ContactList {
    constructor(options) {
        // Region where the application will be placed
    }
}
```
**Architecture of a Backbone application**

```javascript
this.region = options.region;

// Allow subapplication to listen and trigger events,
// useful for subapplication wide events
_.extend(this, Backbone.Events);
}

showList(contacts) {
  // Create the views
  var layout = new ContactListLayout();
  var actionBar = new ContactListActionBar();
  var contactList = new ContactListView({collection: contacts});

  // Show the views
  this.region.show(layout);
  layout.getRegion('actions').show(actionBar);
  layout.getRegion('list').show(contactList);

  this.listenTo(contactList, 'item:contact:delete',
                this.deleteContact);
}

createContact() {
  App.router.navigate('contacts/new', true);
}

deleteContact(view, contact) {
  let message = 'The contact will be deleted';
  App.askConfirmation(message, (isConfirm) => {
    if (isConfirm) {
      contact.destroy({
        success() {
          App.notifySuccess('Contact was deleted');
        },
        error() {
          App.notifyError('Ooops... Something went wrong');
        }
      });
    }
  });
}

// Close any active view and remove event listeners
// to prevent zombie functions
```
destroy() {
    this.region.remove();
    this.stopListening();
}

The function that handles the request is very simple and follows the same pattern for all other controllers, as follows:

- It creates all the necessary views with the model or collection that is passed
- It renders the views in a region of the DOM
- It listens for events in the views

If you don't entirely understand what region and layout means, don't worry, I will cover the implementation of these objects in detail in Chapter 2, Managing views. Here, the important thing is the algorithm described earlier:

As you can see in the above figure, the contact list shows a set of cards for each contact in the list. The user is allowed to delete a contact by clicking on the delete button. When this happens, a contact:delete event is triggered, the controller is listening for the event and uses the deleteContact() method to handle the event:

deleteContact(view, contact) {
    let message = 'The contact will be deleted';
    App.askConfirmation(message, (isConfirm) => {
        if (isConfirm) {

Figure 1.8. ContactList controller result

[17]
Architecture of a Backbone application

```javascript
contact.destroy({
  success() {
    App.notifySuccess('Contact was deleted');
  },
  error() {
    App.notifyError('Ooops... Something went wrong');
  }
});
```

The handler is pretty easy to understand, it uses the `askConfirmation()` method in the infrastructure app to ask for the user confirmation. If the user confirms the deletion, the contact is destroyed. The infrastructure App provides two methods to show notifications to the user: `notifySuccess()` and `notifyError()`.

The cool thing about these App methods is that the controllers do not need to know the details about the confirmation and notification mechanisms. From the controller point of view, it just works.

The method that asks for the confirmation can be a simple `confirm()` call, as follows:

```javascript
// app.js
var App = {
  // ...  
  askConfirmation(message, callback) {
    var isConfirm = confirm(message);
    callback(isConfirm);
  }
};
```

However, in the modern web applications, using the plain `confirm()` function is not the best way to ask for confirmation. Instead, we can show a Bootstrap dialog box or use an available library for that. For simplicity, we will use the nice JavaScript SweetAlert library; however, you can use whatever you want:

```javascript
// app.js
var App = {
  // ... 

  askConfirmation(message, callback) {
    var options = {
      title: 'Are you sure?',
      type: 'warning',
    }
```
text: message,
showCancelButton: true,
confirmButtonText: 'Yes, do it!'
confirmButtonColor: '#5cb85c'
cancelButtonText: 'No'
);

// Show the message
swal(options, function(isConfirm) {
callback(isConfirm);
});

Figure 1.9. Using SweetAlert for confirmation messages

We can implement the notification methods in a similar way. We will use the JavaScript noty library; however, you can use whatever you want:

// app.js
var App = {
    // ...

    notifySuccess(message) {
        new noty({
            text: message,
            layout: 'topRight',
            theme: 'relax',
            type: 'success',
            timeout: 3000 // close automatically
        });
    }
Architecture of a Backbone application

},

notifyError(message) {
    new noty({
        text: message,
        layout: 'topRight',
        theme: 'relax',
        type: 'error',
        timeout: 3000 // close automatically
    });
}
};

Figure 1.10. Using noty to show notification messages

This is how you can implement a robust and maintainable Backbone application; please go to the GitHub repo for this book in order to see the complete code for the application. The views are not covered in the chapter as we will see them in detail in Chapter 2, Managing views.

File organization

When you work with MVC frameworks, file organization is trivial. However, Backbone is not an MVC framework, therefore, bringing your own file structure is the rule. You can organize the code on these paths:

- **apps/**: This directory is where modules or subapplications live. All subapplications should be on this path
- **Components/**: These are the common components that multiple subapplications require or use on the common layout as a breadcrumbs component
- **core/**: Under this path, we can put all the core functions such as utilities, helpers, adapters, and so on
- **vendor/**: On vendor, you can put all third-party libraries; here you can put Backbone and its dependencies.
- **app.js**: This is the entry point of the application that is loaded from index.html
• Subapplications can have a file structure as they are a small Backbone Application.
• models/: This defines the models and collections
• app.js: This is the application façade that is called from the router
• router.js: This is the router of the application that is instantiated by the root application at bootstrap process
• contactList.js, contactEditor.js, contactViewer.js: These are the controllers for the application

For a contacts application, the code organization can be as shown in the following:

![File structure diagram]

Figure 1.11. File structure
Architecture of a Backbone application

Summary
We started by describing, in a general way, how a Backbone application works. It describes two main parts: a root application and subapplications. The root application provides a common infrastructure to other smaller and focused applications that we call subapplications.

Subapplications should be loose coupled with other subapplications and should own its resources such as views, controllers, routers, and so on. A subapplication manages a small part of the system with a well-focused business value and the communication between subapplications and infrastructure application is made through an events-driven bus with Backbone.Events.

The user interacts with the application using views that a subapplication renders. A subapplication controller orchestrates interaction between views, models, and collections and owns the business logic for the use case.

Finally, a file system organization explains the right sites to put your files and keep your project clean and organized. This organization does not follow an MVC pattern; however, it is powerful and simple. It encapsulates all the necessary code for a module in a single path (subapplication paths) instead of putting all the code across multiple paths.

In this way the structure of Backbone applications has been proven to be robust, a proof for this is that several open source applications such as TodoMVC follow (more or less) the principles exposed here. It facilitates the testability of the code due to separation of responsibilities so that each object can be tested separately.

Large Backbone applications are often built on top of Backbone Marionette as it reduces the boilerplate code; however, Marionette uses its own conventions to work. If you are fine with it using its own conventions, you will be happy to use Marionette on top of Backbone.

However, if you love the freedom of doing things your way, you may prefer plain Backbone and create your own utilities and classes.

In the next chapter, I will show you how to manage and organize views and simplify the complex layouts, identifying the common uses of the views. You will build general purpose views that will be useful for all your projects and forget about the implementation of the render() method.
Where to buy this book
You can buy Mastering Backbone.js 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.