

Structuring



BACKBONE.JS

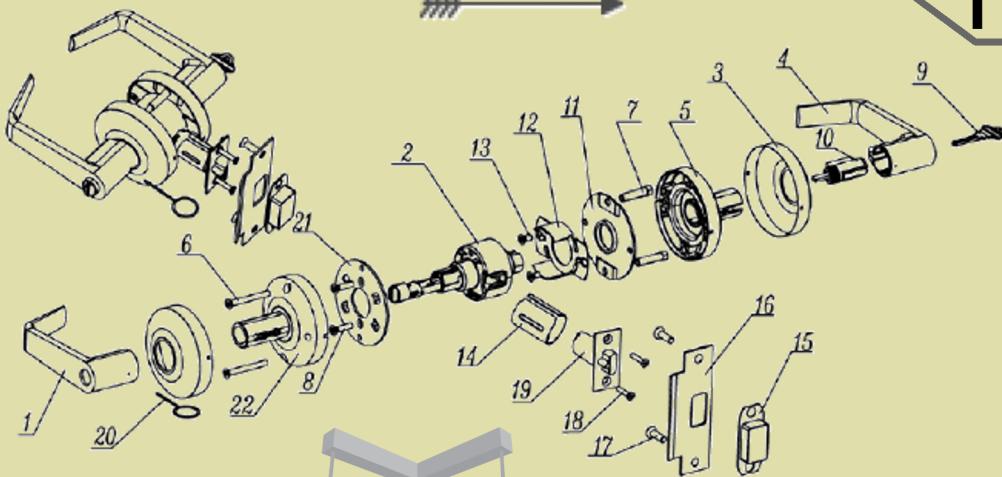
Code

with

REQUIRE.JS



RequireJS



and

**M**arionette<sup>js</sup>

Modules

A Gentle Introduction

by David Sulc

# **Structuring Backbone Code with RequireJS and Marionette Modules**

Learn to use javascript AMD in your apps the easy way

David Sulc

This book is for sale at <http://leanpub.com/structuring-backbone-with-requirejs-and-marionette>

This version was published on 2013-12-14



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 David Sulc

## **Tweet This Book!**

Please help David Sulc by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I just bought @davidsulc's book on @marionettejs and RequireJS. Interested? Check it out at <https://leanpub.com/structuring-backbone-with-requirejs-and-marionette>

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=%23>

## **Also By David Sulc**

[Backbone.Marionette.js: A Gentle Introduction](#)

# Contents

<b>Who This Book is For . . . . .</b>	<b>i</b>
<b>Following Along with Git . . . . .</b>	<b>ii</b>
Jumping in for Advanced Readers . . . . .	iii
<b>Why Use AMD? . . . . .</b>	<b>1</b>
<b>Getting Started . . . . .</b>	<b>2</b>
Adding RequireJS to the Mix . . . . .	3
Using Shims . . . . .	6
<b>Loading the Main App . . . . .</b>	<b>10</b>
Discovering Modules . . . . .	10
Turning ContactManager into a RequireJS Module . . . . .	12
Including the Dialog Region as a Dependency . . . . .	14
<b>Listing Contacts . . . . .</b>	<b>18</b>
Modularizing the Contact Entity . . . . .	18
Using the Contact Entity from the Main App . . . . .	21
Listing Contacts from the List Sub-Apppplication . . . . .	23
Requiring Views and Templates . . . . .	27
<b>Editing Contacts with a Modal . . . . .</b>	<b>41</b>
Expliciting Dependencies . . . . .	49
<b>Creating Contacts with a Modal . . . . .</b>	<b>51</b>
Exercise Solution . . . . .	51
Further Decoupling of the Contact Module . . . . .	53
<b>Filtering Contacts . . . . .</b>	<b>55</b>
Exercise Solution . . . . .	55
<b>Adding the Loading View . . . . .</b>	<b>60</b>
Exercise Solution . . . . .	60
<b>Showing Contacts . . . . .</b>	<b>64</b>

## CONTENTS

Exercise Solution . . . . .	64
<b>Editing Contacts . . . . .</b>	<b>69</b>
Exercise Solution . . . . .	69
<b>Reimplementing the About Sub-Application . . . . .</b>	<b>73</b>
Exercise Solution . . . . .	73
<b>Reimplementing the Header Sub-Application . . . . .</b>	<b>77</b>
Exercise Solution . . . . .	77
<b>Marionette Modules VS Plain RequireJS Modules . . . . .</b>	<b>86</b>
Converting the Contact Entity to Plain RequireJS Modules . . . . .	86
Using a Separate Module as the Event Aggregator . . . . .	90
Converting the Collection . . . . .	90
Rewriting the Show View . . . . .	94
Rewriting the Show Action . . . . .	94
<b>Start/Stop Modules According to Routes . . . . .</b>	<b>96</b>
Implementation Strategy . . . . .	96
Modifying the AboutApp . . . . .	101
<b>Mixing and Matching Modules . . . . .</b>	<b>103</b>
Converting the About Sub-Application to Plain RequireJS Modules . . . . .	103
Managing Start/Stop with Mixed Module Types . . . . .	105
<b>Optimizing the Application with r.js . . . . .</b>	<b>107</b>
Loading with Almond.js . . . . .	109
<b>Closing Thoughts . . . . .</b>	<b>112</b>
Keeping in Touch . . . . .	112
<b>Other Books I've Written . . . . .</b>	<b>113</b>

# Who This Book is For

This book is first and foremost for the readers of “[Backbone.Marionette.js: A Gentle Introduction](#)<sup>1</sup>” who want to take their knowledge to the next level, by using RequireJS to structure their code and load dependencies.

To follow along, you’ll need to have at least passing familiarity with the Contact Manager application developed in my previous book. Put another way, the focus of this book is how to use RequireJS with Marionette: it does NOT explain how to leverage the various components Marionette provides (that’s covered in the book linked above).

---

<sup>1</sup><https://leanpub.com/marionette-gentle-introduction>

# Following Along with Git

This book is a step by step guide to rebuilding the Contact Manager application to use RequireJS. As such, it's accompanied by source code in a Git repository hosted at <https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette><sup>2</sup>.

Throughout the book, as we code our app, we'll refer to commit references within the git repository like this:



Git commit with our basic index.html:

[d54f31fc9c2463c435abf1a32a361406500bff65<sup>3</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/d54f31fc9c2463c435abf1a32a361406500bff65)

This will allow you to follow along and see exactly how the code base has changed: you can either look at that particular commit in your local copy of the git repository, or click on the link to see an online display of the code differences.



Any change in the code will affect all the following commit references, so the links in your version of the book might become desynchronized. If that's the case, make sure you update your copy of the book to get the new links. At any time, you can also see the full list of commits [here](#)<sup>4</sup>, which should enable you to locate the commit you're looking for (the commit names match their descriptions in the book).

Even if you haven't used Git yet, you should be able to get up and running quite easily using online resources such as the [Git Book](#)<sup>5</sup>. This chapter is by no means a comprehensive introduction to Git, but the following should get you started:

- Set up Git with Github's [instructions](#)<sup>6</sup>
- To get a copy of the source code repository on your computer, open a command line and run

```
git clone git://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette.git
```

- From the command line move into the structuring-backbone-with-requirejs-and-marionette folder that Git created in the step above, and execute

<sup>2</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette>

<sup>3</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/d54f31fc9c2463c435abf1a32a361406500bff65>

<sup>4</sup><https://github.com/davidsulc/marionette-gentle-introduction/commits/master>

<sup>5</sup><http://git-scm.com/book>

<sup>6</sup><https://help.github.com/articles/set-up-git>

```
git show d54f31fc9c2463c435abf1a32a361406500bff65
```

to show the code differences implemented by that commit:

- '-' lines were removed
- '+' lines were added

You can also use Git to view the code at different stages as it evolves within the book:

- To extract the code as it was during a given commit, execute

```
git checkout d54f31fc9c2463c435abf1a32a361406500bff65
```

- Look around in the files, they'll be in the exact state they were in at that point in time within the book
- Once you're done looking around and wish to go back to the current state of the code base, run

```
git checkout master
```



## What if I don't want to use Git, and only want the latest version of the code?

You can download a [zipped copy of the repository](#)<sup>7</sup>. This will contain the full Git commit history, in case you change your mind about following along.

## Jumping in for Advanced Readers

My goal with this book is to get you comfortable enough to use RequireJS in your own Marionette projects, and it rebuild an existing application from start to finish with RequireJS. Therefore, after presenting the main concepts you need to know when using RequireJS, the book guides you in rewriting the ContactManager application by pointing out various aspects to bear in mind, and then providing a step by step guide to the actual implementation.

Although you'll learn the most by following along with the code, you can simply skim the content and checkout the Git commit corresponding to the point in the book where you wish to join in.

---

<sup>7</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/archive/master.zip>

# Why Use AMD?

Asynchronous Module Definition (or AMD for short) is one answer to the problems that plague modern web development:

- web applications require lots of javascript code;
- to manage application complexity, javascript code needs to be broken down into smaller files;
- these smaller javascript files have dependencies on one another that aren't clear from their inclusion via `<script>` tags: the tag sequence only gives a rough sense of order, not a dependency tree;
- loading a long list of javascript files with includes slows down your web page: each file requires a separate HTTP request, and is blocking.

By using RequireJS in a web app, you get the following benefits:

- your javascript code can be distributed among many smaller files;
- you explicitly declare the dependencies each javascript module requires (both libraries and other modules);
- you can build a single, optimized, and minified javascript file for production deployment.

# Getting Started

First, let's grab a copy of the source code for our basic Contact Manager application (as it was developed in my “[Backbone.Marionette.js: A Gentle Introduction](#)<sup>8</sup>” book) from [here](#)<sup>9</sup>. We’re doing this for convenience, since we’ll now have all libraries and assets (images, icons, etc.) available and won’t have to deal with fetching them in the next chapters. Downloading the entire app also means we have all of our modules ready to go, simply waiting to be integrated in our RequireJS app.

Now, let’s start with a basic `index.html` file, by removing all javascript includes, and templates:

## Our initial index.html

---

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Marionette Contact Manager</title>
6     <link href=". /assets/css/bootstrap.css" rel="stylesheet">
7     <link href=". /assets/css/application.css" rel="stylesheet">
8     <link href=". /assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
9   </head>
10
11  <body>
12    <div id="header-region"></div>
13
14    <div id="main-region" class="container">
15      <p>Here is static content in the web page. You'll notice that it
16        gets replaced by our app as soon as we start it.</p>
17    </div>
18
19    <div id="dialog-region"></div>
20  </body>
21 </html>
```

---

You may have noticed that we’ve got some CSS includes that aren’t needed yet. We’ll just leave them there, so we can avoid dealing with them later: they’ll already be included when we do need them.

---

<sup>8</sup><https://leanpub.com/marionette-gentle-introduction>

<sup>9</sup><https://github.com/davidsulc/marionette-gentle-introduction/archive/master.zip>



Git commit with our basic index.html:

32fce9a507ac80b9b27e364536de9e7adeeb203f<sup>10</sup>

## Adding RequireJS to the Mix

With our basic `index.html` in place, let's move on to our next goal: using RequireJS to load our libraries. So let's get RequireJS from [here<sup>11</sup>](#) and save it in `assets/js/vendor/require.js`. Now, we can add RequireJS to our `index.html` (line 22):

### Adding RequireJS (`index.html`)

---

```

1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <title>Marionette Contact Manager</title>
6          <link href=".//assets/css/bootstrap.css" rel="stylesheet">
7          <link href=".//assets/css/application.css" rel="stylesheet">
8          <link href=".//assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
9      </head>
10
11     <body>
12
13         <div id="header-region"></div>
14
15         <div id="main-region" class="container">
16             <p>Here is static content in the web page. You'll notice that it
17                 gets replaced by our app as soon as we start it.</p>
18         </div>
19
20         <div id="dialog-region"></div>
21
22         <script src=".//assets/js/vendor/require.js"></script>
23     </body>
24 </html>

```

---

So how do we actually load our libraries with RequireJS? We're going to add a `data-main` attribute to the `script` tag, which will make RequireJS automatically load the provided javascript file. This file,

<sup>10</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/32fce9a507ac80b9b27e364536de9e7adeeb203f>

<sup>11</sup><http://requirejs.org/docs/release/2.1.8/comments/require.js>

in turn, will configure RequireJS per our environment, load the top-level dependencies and libraries, and then start our Contact Manager application. So let's change our script tag on line 22 to

```
<script data-main=". /assets/js/require_main.js"
       src=". /assets/js/vendor/require.js"></script>
```

That was easy enough! Of course, we now need to actually write *require\_main.js* to provide configuration directives. Let's start by simply using RequireJS to load jQuery and display jQuery's version number:

assets/js/require\_main.js

---

```
1 require(["vendor/jquery"], function(){
2   console.log("jQuery version: ", $.fn.jquery);
3 });
```

---

If you refresh `index.html`, you'll now see jQuery's version number printed in the console. How is this happening? First, RequireJS determines the `baseUr1`, which is the path from which all other paths are computed. By default, it is set to the path provided to the `data-main` attribute above (minus the file portion, of course). So in our case, the `baseUr1` is `assets/js`.

Then, within *require\_main.js*, we call RequireJS's `require` function, which takes 2 arguments:

1. an array of dependencies that need to be loaded before executing the callback function;
2. a callback function that gets executed once all the required dependencies have been loaded.



Note that RequireJS also defines a `requirejs` function, which is interchangeable with `require`.

In the dependency array, we provide the path to the file we want to load *relative to the baseUr1 path*. As we indicated jQuery was located at `vendor/jquery`, and our `baseUr1` is `assets/js` (because that's where our `require_main.js` file is located), RequireJS will look for the library file in `assets/js/vendor/jquery.js`. So our code essentially instructs "once jQuery is loaded from `assets/js/vendor/jquery.js`, print its version number."



Note that we specify file without extensions. For example, to require jQuery we had "vendor/jquery" as the dependency, and RequireJS automatically added the ".js" extension.

But we can still improve our code by adding some global configuration:

**assets/js/require\_main.js**


---

```

1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     jquery: "vendor/jquery"
5   }
6 });
7
8 require(["jquery"], function(jq){
9   console.log("jQuery version: ", jq.fn.jquery);
10  console.log("jQuery version: ", $.fn.jquery);
11 });

```

---



Due to the equivalence of `require` and `requirejs`, the `config` method could also be called via `require.config`. Why do I use `requirejs` on line 1 and then `require` on line 8? It's just a matter of personal taste: I find that `requirejs.config` explicitly indicates we are configuring RequireJS, and using the `require` function call reads better (i.e. “require these files before proceeding”). But be aware of the `requirejs` and `require` synonyms when you look at RequireJS code.

So we've added some configuration, but what does it do? First, we provide a `baseUrl` path. This isn't really necessary in our case (since we're using the same path as the default), but I find it helps with clarity: since all the other paths indicated in the file will be defined relative to the `baseUrl`, it's practical to have it explicitly defined on the same page.

Next, we've added a `paths`<sup>12</sup> object to specify paths that aren't directly under the `baseUrl` path. This allows us to create a “`jquery`” key to use when requiring the library, and RequireJS will know it should be loaded from the *vendor/jquery* location.



From the [documentation](#)<sup>13</sup>:

The path that is used for a module name should **not** include an extension, since the path mapping could be for a directory. The path mapping code will automatically add the `.js` extension when mapping the module name to a path.



By using `path fallbacks`<sup>14</sup>, you can easily load libraries from CDNs and default to using a local version should the CDN be unavailable.

---

<sup>12</sup><http://requirejs.org/docs/api.html#config-paths>

<sup>13</sup><http://requirejs.org/docs/api.html#config-paths>

<sup>14</sup><http://requirejs.org/docs/api.html#pathsfallbacks>

With this configuration in place, we can now simply require “jquery” and RequireJS will know where to look. In addition, note that RequireJS will provide the array of dependencies as arguments to the callback function: in the code above, we have the “jquery” dependency assigned to the jq variable, which allows us to then call jq.fn.jquery to get jQuery’s version number. As you can tell from the console output, using the provided callback argument is equivalent to using the global \$ variable.



JQuery’s case is a bit special, so let’s talk about it a bit. Ever since version 1.7, jQuery is AMD-compatible meaning that it can be loaded with a module loader such as RequireJS. In that case, the module loader will receive a reference to jQuery which can be used in the callback function. All of this happens *in addition* to jQuery’s registering the global \$ variable.

However, with most other modules and libraries, no global variables will be registered. That’s one of the objectives when using RequireJS: don’t register objects in the global namespace, and require the ones you need instead, passing them into the callback function.

## Using Shims

We’ve seen how to load an AMD-compatible library (namely, jQuery), so let’s move on to loading one that is *not* compatible with AMD: Underscore. By using the new shim object in RequireJS, we can still load AMD-incompatible scripts with RequireJS: all we need to do is provide a little more information. Here we go:

assets/js/require\_main.js

```
1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     jquery: "vendor/jquery",
5     underscore: "vendor/underscore"
6   },
7
8   shim: {
9     underscore: {
10       exports: "_"
11     }
12   }
13 });
14
15 require(["underscore", "jquery"], function(un){
16   console.log("jQuery version: ", $.fn.jquery);
```

```
17 console.log("underscore identity call: ", _.identity(5));
18 console.log("underscore identity call: ", un.identity(5));
19});
```

---

First, we indicate where Underscore is located (line 5). With that done, we add the “underscore” key to the `shim` object and indicate that it should return the value of the `_` global to the callback function. Notice that, thanks to the shim, we have access to both the global `_` value (line 17) and the `un` callback value (line 18). If we hadn’t used a shim, we wouldn’t be able to use the `un` callback value, only the `_` global.



How come we require 2 files, but have only one callback argument? Because we only explicitly state the callback arguments we’re going to be using (and for which we need a variable to reference). So why bother having the “jquery” as a dependency? Because even though we’re not using an explicit callback argument value, we know that within the callback function jQuery has been loaded and can be used via the global \$ variable. This is due to jQuery’s particularity of registering a global variable we can reference: there’s no need to use a callback argument.

Let’s move on to loading something with dependencies: Backbone. Here we go:

assets/js/require\_main.js

```
1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     backbone: "vendor/backbone",
5     jquery: "vendor/jquery",
6     json2: "vendor/json2",
7     underscore: "vendor/underscore"
8   },
9
10  shim: {
11    underscore: {
12      exports: "_"
13    },
14    backbone: {
15      deps: ["jquery", "underscore", "json2"],
16      exports: "Backbone"
17    }
18  }
19});
```

```
20
21 require(["backbone"], function(bb){
22   console.log("jQuery version: ", $.fn.jquery);
23   console.log("underscore identity call: ", _.identity(5));
24   console.log("Backbone.history: ", bb.history);
25 });



---


```

Let's start studying this code in the shim (lines 14-17):

1. we declare a "backbone" key we can then use to require the library;
2. we indicate that Backbone depends on jQuery, Underscore, and JSON2. This implies that before loading Backbone, RequireJS has to load these dependencies;
3. we indicate that the global Backbone variable should be returned to the callback function.

Naturally, we need to tell RequireJS where to find the various files so we've had to add paths for Backbone (line 4) and JSON2 (line 6).

With that configuration set up, we can require Backbone on line 21 and use it within the callback function. But how come we can still use jQuery and Underscore without requiring them (lines 22-23)? Because of the dependency tree: Backbone depends on those libraries (see above), so once Backbone and its dependencies have been loaded we have access to jQuery and Underscore via their global variables (since they're both Backbone dependencies). RequireJS magic in action!



Try adding Marionette to *require\_main.js* on your own, then check below. Things to remember:

- You'll need to specify a path for Marionette;
- Marionette depends on Backbone;
- Marionette registers the global `Marionette` variable.

You can see the solution on the next page.

## assets/js/require\_main.js

---

```

1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     backbone: "vendor/backbone",
5     jquery: "vendor/jquery",
6     json2: "vendor/json2",
7     marionette: "vendor/backbone.marionette",
8     underscore: "vendor/underscore"
9   },
10
11   shim: {
12     underscore: {
13       exports: "_"
14     },
15     backbone: {
16       deps: ["jquery", "underscore", "json2"],
17       exports: "Backbone"
18     },
19     marionette: {
20       deps: ["backbone"],
21       exports: "Marionette"
22     }
23   }
24 });
25
26 require(["marionette"], function(bbm){
27   console.log("jQuery version: ", $.fn.jquery);
28   console.log("underscore identity call: ", _.identity(5));
29   console.log("Marionette: ", bbm);
30 });

```

---

Adding Marionette is pretty straightforward, as it's no different from adding Backbone. However, do note that since Marionette depends on Backbone, we don't need to specify that it also depends (e.g.) on Underscore: RequireJS will load Underscore, then Backbone, then Marionette, due to the dependency relationships we've indicated in the config method.



Git commit loading basic libraries:

0b8d7506bf18dd930ccf824897a7d839040ccc9c<sup>15</sup>

---

<sup>15</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/0b8d7506bf18dd930ccf824897a7d839040ccc9c>

# Loading the Main App

Now that we understand the basics about how RequireJS works, let's use it to require our Contact Manager application and start it. We'll start with a very basic app file, to give us a first taste of defining and working with RequireJS modules. You should already have a file in *assets/js/app.js*, so save it elsewhere, or just overwrite it. We don't need it yet, but will be using it soon enough. Here's our basic application:

assets/js/app.js

---

```
1 var ContactManager = new Marionette.Application();
2
3 ContactManager.on("initialize:after", function(){
4   console.log("Contact Manager has started");
5 });
```

---



This code should be obvious to you. If not I suggest you read my [book introducing Marionette](#)<sup>16</sup>.

All our app does for now is print a message to the console once it has started, which is enough to start working with RequireJS. So now what? To be able to use our Contact Manager application as a dependency, we'll need to define it as a RequireJS module. In simple terms, a RequireJS module indicates the required dependencies, and returns a value to be used in the callback function provided to a `require` call.

## Discovering Modules

If you look at our simple code above, you can see that we're going to need access to Marionette. In addition, we're going to need access to `ContactManager` in many places in our application, e.g. to define modules. Here's what our Contact Manager looks like as a RequireJS module:

---

<sup>16</sup><https://leanpub.com/marionette-gentle-introduction>

assets/js/app.js

---

```
1 define(["marionette"], function(Marionette){  
2     var ContactManager = new Marionette.Application();  
3  
4     ContactManager.on("initialize:after", function(){  
5         console.log("Contact Manager has started");  
6     });  
7  
8     return ContactManager;  
9 });
```

---

The only modifications we've added are the call to `define` on line 1, and returning the reference to our application on line 8. Now, let's return to our `require_main.js` file, to require and start our app:

assets/js/require\_main.js

---

```
1 requirejs.config({  
2     // edited for brevity  
3 });  
4  
5 require(["app"], function(ContactManager){  
6     ContactManager.start();  
7 });
```

---

As explained earlier, we're requiring the main application's file (which evaluates to `assets/js/app.js` due to the `baseUrl` value), which we receive in the provided callback. We can then start the application as usual. There is a major difference with RequireJS, however: `ContactManager` is no longer a global variable, and can be accessed only via requiring the main application.



What about debugging? If we type `ContactManager` in the console, we won't have a usable reference to our application. So how can we debug our code? We simply need to require it, like so:

```
var ContactManager = require("app");
```

After that, the `ContactManager` variable will point to our application instance.

## Turning ContactManager into a RequireJS Module

Enough with the baby code, let's move on to the real deal. This is what our (original) *app.js* looks like:

assets/js/app.js

---

```
1 var ContactManager = new Marionette.Application();
2
3 ContactManager.addRegions({
4     headerRegion: "#header-region",
5     mainRegion: "#main-region",
6     dialogRegion: Marionette.Region.Dialog.extend({
7         el: "#dialog-region"
8     })
9 });
10
11 ContactManager.navigate = function(route, options){
12     options || (options = {});
13     Backbone.history.navigate(route, options);
14 };
15
16 ContactManager.getCurrentRoute = function(){
17     return Backbone.history.fragment
18 };
19
20 ContactManager.on("initialize:after", function(){
21     if(Backbone.history){
22         Backbone.history.start();
23
24         if(this.getCurrentRoute() === ""){
25             ContactManager.trigger("contacts:list");
26         }
27     }
28 });
```

---

Let's start by turning it into a module, just as we've done above:

assets/js/app.js

```
1 define(["marionette"], function(Marionette){  
2     var ContactManager = new Marionette.Application();  
3  
4     ContactManager.addRegions({  
5         headerRegion: "#header-region",  
6         mainRegion: "#main-region",  
7         dialogRegion: Marionette.Region.Dialog.extend({  
8             el: "#dialog-region"  
9         })  
10    });  
11  
12    ContactManager.navigate = function(route, options){  
13        options || (options = {});  
14        Backbone.history.navigate(route, options);  
15    };  
16  
17    ContactManager.getCurrentRoute = function(){  
18        return Backbone.history.fragment  
19    };  
20  
21    ContactManager.on("initialize:after", function(){  
22        console.log("Contact Manager has started");  
23        if(Backbone.history){  
24            Backbone.history.start();  
25  
26            if(this.getCurrentRoute() === ""){  
27                ContactManager.trigger("contacts:list");  
28            }  
29        }  
30    });  
31  
32    return ContactManager;  
33});
```

We've simply added the code at the beginning and end to turn it into a module, and we've also added a console message on line 22. What happens when we refresh the page? We get a nice error saying that "Marionette.Region.Dialog is undefined", because we're trying to use it on lines 7-9, but we've never bothered to include it. If we comment those 3 lines and refresh, we can see that everything else is ok...

## Including the Dialog Region as a Dependency

Now that we've confirmed the only issue with our code is the missing dialog region, let's include it. Actually, no. *YOU* try and figure out how to include it and make it work. As much as possible, this book will be structured as a series of exercises where you rewrite the application based on your current knowledge of RequireJS. At each step, a list of things to remember will be provided, and once you've written the code (or at least attempted to), the book will continue by explaining the implementation in more detail. This method will allow you to make some mistakes and start *really* understanding how to rewrite an app to use RequireJS, instead of reading code and think "well, this is pretty obvious...". Ready? Let's go!



Specify our dialog region (located in `assets/js/apps/config/marionette/regions/dialog.js`) as a dependency to our main application file. Here are a few things you'll need to do:

- define `dialog.js` as a RequireJS module, and don't forget:
  - the dialog region will have a dependency on jQuery UI in addition to Marionette
  - modules need to return the value that is to be used by dependents
- add an entry for jQuery UI in the RequireJS configuration within `require_main.js`
- include the path to the newly defined dialog module to the dependency array in the `define` call within the main application (i.e. the main application's RequireJS module depends on the dialog region)

Here are a few additional pointers:

- as recommended by the RequireJS [documentation<sup>17</sup>](#), you should rename the jQuery UI file to `jquery-ui.js` (i.e. without version information in the filename)
- don't forget that javascript objects need their property names to be valid javascript identifiers, or strings (for more info, see [here<sup>18</sup>](#)). Therefore, I suggest you use the "jquery-ui" string as the identifier for the jQuery UI library within `require_main.js`
- within the `shim` object in `require_main.js`, you can specify that jQuery UI depends on jQuery by providing an array of dependencies. We haven't covered this in the book yet, but try and figure it out from the [documentation<sup>19</sup>](#) (read the section on jQuery and Backbone plugins)
- when indicating the location of the dialog region module (for the main app's dependency array), don't forget paths are to be indicated relative to the `assets/js/baseUrl`

You can see the solution on the next page.

<sup>17</sup><http://requirejs.org/docs/api.html#jsfiles>

<sup>18</sup>[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working\\_with\\_Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects)

<sup>19</sup><http://requirejs.org/docs/api.html#config-shim>

Let's start by adding jQuery UI to our setup:

assets/js/require\_main.js

---

```
1 requirejs.config({
2     baseUrl: "assets/js",
3     paths: {
4         backbone: "vendor/backbone",
5         jquery: "vendor/jquery",
6         "jquery-ui": "vendor/jquery-ui",
7         json2: "vendor/json2",
8         marionette: "vendor/backbone.marionette",
9         underscore: "vendor/underscore"
10    },
11
12    shim: {
13        underscore: {
14            exports: "_"
15        },
16        backbone: {
17            deps: ["jquery", "underscore", "json2"],
18            exports: "Backbone"
19        },
20        marionette: {
21            deps: ["backbone"],
22            exports: "Marionette"
23        },
24        "jquery-ui": ["jquery"]
25    }
26 });
27
28 require(["app"], function(ContactManager){
29     ContactManager.start();
30 });
```

---

Now, we can convert our dialog region to be a RequireJS module:

assets/js/apps/config/marionette/regions/dialog.js

---

```

1 define(["marionette", "jquery-ui"], function(Marionette){
2     Marionette.Region.Dialog = Marionette.Region.extend({
3         onShow: function(view){
4             this.listenTo(view, "dialog:close", this.closeDialog);
5
6             var self = this;
7             this.$el.dialog({
8                 modal: true,
9                 title: view.title,
10                width: "auto",
11                close: function(e, ui){
12                    self.closeDialog();
13                }
14            });
15        },
16
17        closeDialog: function(){
18            this.stopListening();
19            this.close();
20            this.$el.dialog("destroy");
21        }
22    });
23
24    return Marionette.Region.Dialog;
25 });

```

---

Once again, the only modifications to turn the existing code into a RequireJS module, were to specify the dependencies and put the existing code within the callback function (line 1), and return the proper reference value (line 24). On the first line, note that we're requiring jQuery UI, but aren't using the value within the callback function's arguments. This is because we need jQuery UI to be present (for the call to dialog on line 7), but there's no explicit call to jQuery UI within the code. In contrast, we need a reference to Marionette, because we explicitly use it on line 2.<sup>20</sup>

Then, we simply need to add our dialog region to the required dependencies in our main app's module definition:

---

<sup>20</sup>Note that this isn't technically true: we're using the standard Marionette build, which registers a global `Marionette` reference.

---

assets/js/app.js

---

```
1 define(["marionette", "apps/config/marionette/regions/dialog"],  
2                     function(Marionette){  
3     var ContactManager = new Marionette.Application();  
4  
5     ContactManager.addRegions({  
6         headerRegion: "#header-region",  
7         mainRegion: "#main-region",  
8         dialogRegion: Marionette.Region.Dialog.extend({  
9             el: "#dialog-region"  
10        })  
11    });  
12  
13    // unmodified code edited for brevity  
14  
15    return ContactManager;  
16});
```

---

Once again, we require the dialog region as a dependency, but don't add it to the callback arguments. That's because the dialog region needs to be loaded by the time we use it on line 8, but there's no direct reference to it (since we access it via the Marionette object).

Now, when we refresh the page, we can see our message in the console, with no errors.



Git commit loading the dialog region dependency:

0d19d5798baa9cd35decc27a6798d7101ee18dbc<sup>21</sup>

---

<sup>21</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/0d19d5798baa9cd35decc27a6798d7101ee18dbc>

# Listing Contacts

So far, we've got our main app starting properly, requiring its dependencies, and displaying a message in the console. In this chapter, we'll work on getting our list of contacts displayed. The first thing we need to do is turn our contacts entity into a module.

When rewriting an existing application to use RequireJS, it's important to structure your approach based on the dependency tree, and to work in as small increments as possible. This allows you to check your reimplementation with less new code, and allows you to track down issues much more efficiently. Naturally, this is an established best practice but it takes on particular importance when working with RequireJS: trying to determine why certain libraries won't load properly after a huge diff is not enjoyable.

With that in mind, here's how we will approach this iteration:

1. Define modules for our contact entity, and make sure it's working properly (from the console);
2. Require the module from the main app file, and check we can work with it properly when loading it as a dependency;
3. Ensure we can obtain the contacts from the "list" controller;
4. Display the contact within views;
5. Using RequireJS to load the view templates.

## Modularizing the Contact Entity



Let's get started: first, you need to make a module out of our configuration for localstorage use (at `assets/js/apps/config/storage/localstorage.js`). Don't forget that this module will depend on the Backbone.localstorage plugin, and that in turn the plugin depends on Backbone.

You can see the solution on the next page.

Let's start by adding the Backbone.localstorage plugin to our *require\_main.js*:

---

**assets/js/require\_main.js**


---

```

1 requirejs.config({
2     baseUrl: "assets/js",
3     paths: {
4         // edited for brevity
5         localStorage: "vendor/backbone.localStorage",
6         // edited for brevity
7     },
8
9     shim: {
10        // edited for brevity
11        "jquery-ui": ["jquery"],
12        localStorage: ["backbone"]
13    }
14 });
15
16 require(["app"], function(ContactManager){
17     ContactManager.start();
18 });

```

---



Don't forget commas when adding attributes (e.g. after line 11)

With the plugin now accessible via the “localStorage” key, let's convert our storage configuration code into a RequireJS module:

---

**assets/js/apps/config/storage/localstorage.js**


---

```

1 define(["app", "localStorage"], function(ContactManager){
2     ContactManager.module("Entities", function(Entities, ContactManager,
3                                         Backbone, Marionette, $, _){
4         // edited for brevity
5     });
6
7     return ContactManager.Entities.configureStorage;
8 });

```

---

And now, we can also turn our contact entity into a module:

assets/js/entities/contact.js

```
1 define(["app", "apps/config/storage/localstorage"], function(ContactManager){  
2   ContactManager.module("Entities", function(Entities, ContactManager,  
3                                         Backbone, Marionette, $, _){  
4     // edited for brevity  
5   });  
6  
7   return ;  
8});
```



Note that the code above doesn't exactly conform the RequireJS best practice of "one module per file" (see [here<sup>22</sup>](#)). The cleanest way to proceed would be to define one module for the Contact model, and another for the ContactCollection collection (which would have the "contact" module as a dependency). These modules would return the defined model or collection, respectively.

Why aren't we doing that? Well, we're simplifying our dependency tree somewhat since the Contacts collection doesn't need to depend on the Contact model: they're both in the same Marionette module. In addition, don't forget we need our "contact:entity" and "contact:entities" handlers to be registered before we can use them in our application. This means that RequireJS needs to load them, and it's easier to just have everything in one place (model, collection, event handlers). And, last but not least, we are going to interact with our contact entities using the request-response mechanism, so there's no need for us to have an explicit reference to the model or collection.

In our case, we're using a single module to define 2 separate things (a model and a collection), so there's really no clear return value to provide as the module return value. We still include a `return` statement for clarity (indicating we haven't forgotten the return value, we're returning "nothing"). How will this code be functional? Because we simply need it to be loaded: we're not going to access the contact entities by explicit reference, we're going to request their values via Marionette's request-response mechanism so we have no need for a "usable" return value.

So far, so good, let's verify our contact entity module works, by executing the following code in the console:

<sup>22</sup><http://requirejs.org/docs/api.html#modulenotes>

### Testing the contact entity module from the console

---

```

1 require(["app", "entities/contact"], function(ContactManager){
2     var fetchingContacts = ContactManager.request("contact:entities");
3     $.when(fetchingContacts).done(function(contacts){
4         console.log(contacts);
5     });
6 });

```

---

After executing that code, we can see our contacts listed in the console. Success!

## Using the Contact Entity from the Main App

Now that we know our contact entity module is functional, let's use it from within our main application to display the contacts in the console once the app is initialized. Here we go:

assets/js/app.js

---

```

1 define(["marionette", "entities/contact",
2         "apps/config/marionette/regions/dialog"], function(Marionette){
3     var ContactManager = new Marionette.Application();
4
5     // edited for brevity
6
7     ContactManager.on("initialize:after", function(){
8         var fetchingContacts = ContactManager.request("contact:entities");
9         $.when(fetchingContacts).done(function(contacts){
10             console.log(contacts);
11         });
12
13         if(Backbone.history){
14             Backbone.history.start();
15
16             if(this.getCurrentRoute() === ""){
17                 ContactManager.trigger("contacts:list");
18             }
19         }
20     });
21

```

---

```
22     return ContactManager;
23 } );
```

---

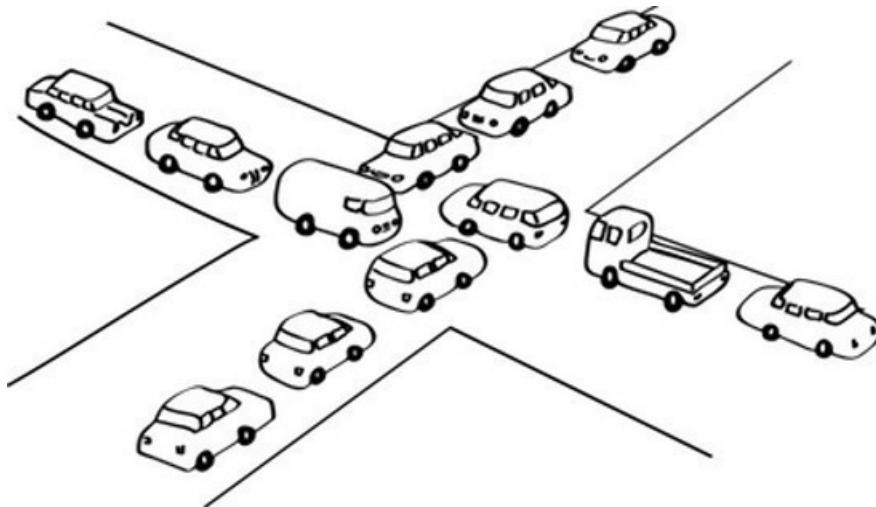
We refresh the page, and boom! An error: “`TypeError: ContactManager is undefined`” in file `localStorage.js`. Why is this happening? Let’s think about how RequireJS is traversing the dependency tree:

1. In `require_main.js`, we require the `assets/js/app.js` module;
2. In `app.js`, we require `assets/js/entities/contact.js` (among others);
3. In `contact.js`, we require `assets/js/apps/config/storage/localstorage.js` (among others);
4. In `localStorage.js`, we require `assets/js/app.js` (among others).

So if we summarize the situation (and skip extra steps), we have the following situation:

- `app.js` needs `localStorage.js` (via the dependency tree: steps 2 and 3 above)
- `localStorage.js` needs `app.js` (as a direct dependency: step 4 above)

Which is a [circular dependency](#)<sup>23</sup>: 2 modules depend on one another, so they don’t load properly. The error message basically indicates that the `ContactManager` module can’t be loaded in the `localStorage.js` file (due to the circular dependency) so an `undefined` return value gets assigned to the callback argument. Circular dependencies can be thought of visually like this<sup>24</sup>:



Each vehicle wants to go forward, so nobody gets to move forward...

How do we fix this situation? Push the dependency requirements as far down as possible (i.e. as close as possible to where they are needed). In fact, the only dependencies that should be listed in the module definition are those without which the module can’t function, even momentarily. In most cases, the only such dependency will be the main app (so we can execute the module definition function). With that in mind, here’s our rewritten main app:

<sup>23</sup><http://requirejs.org/docs/api.html#circular>

<sup>24</sup>Illustration from <http://csunplugged.org/routing-and-deadlock>

---

assets/js/app.js

```

1 define(["marionette", "apps/config/marionette/regions/dialog"],
2         function(Marionette){
3     var ContactManager = new Marionette.Application();
4
5     // edited for brevity
6
7     ContactManager.on("initialize:after", function(){
8         require(["entities/contact"], function(){
9             var fetchingContacts = ContactManager.request("contact:entities");
10            $.when(fetchingContacts).done(function(contacts){
11                console.log(contacts);
12            });
13        });
14
15        if(Backbone.history){
16            Backbone.history.start();
17
18            if(this.getCurrentRoute() === ""){
19                ContactManager.trigger("contacts:list");
20            }
21        }
22    });
23
24    return ContactManager;
25 });

```

---

We've removed the dependency from the module definition, and we simply require our contacts where we need them (namely, on line 8). Now, when we refresh we can see our contacts in the console output.



Note that since we're progressing in short iterations, it was relatively straightforward to pinpoint the issue plaguing our code. Debugging this type of problem with lots of new code would have been more time intensive, as the error message regarding the circular dependency isn't particularly explicit. So keep your code diffs small!

## Listing Contacts from the List Sub-Apppplication

First, we obviously need to remove the contact-fetching code from the main app, so our event listener looks like this:

assets/js/app.js

---

```
1 ContactManager.on("initialize:after", function(){
2     if(Backbone.history){
3         Backbone.history.start();
4
5         if(this.getCurrentRoute() === ""){
6             ContactManager.trigger("contacts:list");
7         }
8     }
9 });

```

---

Next, we'll start with a basic controller for our List sub-application:

assets/js/apps/contacts/list/list\_controller.js

---

```
1 ContactManager.module("ContactsApp.List", function(List, ContactManager,
2                                         Backbone, Marionette, $, _){
3     List.Controller = {
4         listContacts: function(){
5             var fetchingContacts = ContactManager.request("contact:entities");
6
7             $.when(fetchingContacts).done(function(contacts){
8                 console.log(contacts);
9             });
10        }
11    }
12 });

```

---



Turn the code above into a module. Don't forget to add the dependency on the contact entity module in the proper place.

You can see the solution on the next page.

Here's our List controller as a module:

assets/js/apps/contacts/list/list\_controller.js

---

```

1 define(["app"], function(ContactManager){
2   ContactManager.module("ContactsApp.List", function(List, ContactManager,
3                                         Backbone, Marionette, $, _){
4     List.Controller = {
5       listContacts: function(){
6         require(["entities/contact"], function(){
7           var fetchingContacts = ContactManager.request("contact:entities");
8
9           $.when(fetchingContacts).done(function(contacts){
10             console.log(contacts);
11           });
12         });
13       }
14     });
15   });
16
17   return ContactManager.ContactsApp.List.Controller;
18 });

```

---

Notice that we're sticking to our guidelines: only *app* is listed as a module dependency (since otherwise the module wouldn't work at all), and we've required *entities/contact* only where it's needed (lines 6-12).

Finally, let's write a basic main file for our Contacts sub-app:

assets/js/apps/contacts/contacts\_app.js

---

```

1 define(["app"], function(ContactManager){
2   ContactManager.module("ContactsApp", function(ContactsApp, ContactManager,
3                                         Backbone, Marionette, $, _){
4     ContactsApp.Router = Marionette.AppRouter.extend({
5       appRoutes: {
6         "contacts": "listContacts"
7       }
8     });
9
10    var API = {

```

---

```
11     listContacts: function(criterion){
12         require(["apps/contacts/list/list_controller"], function(ListController){
13             ListController.listContacts(criterion);
14         });
15     }
16 };
17
18 ContactManager.on("contacts:list", function(){
19     ContactManager.navigate("contacts");
20     API.listContacts();
21 });
22
23 ContactManager.addInitializer(function(){
24     new ContactsApp.Router({
25         controller: API
26     });
27 });
28 });
29
30 return ContactManager.ContactsApp;
31});
```

---

And again, only *app* is listed as a module dependency, while we require *apps/contacts/list/list\_controller* only for lines 12-14 where it's actually needed.



Note that on line 11, we're passing the `criterion` argument that was in the original code, even though we won't be using it yet. At this point the criterion isn't yet parsed from any route, so its value will simply be `undefined` for now.

With all these pieces in place, we can take the final step to wire everything up: requiring the main `Contacts` file. Each sub-app needs to be required by the main application's file, before Backbone's `History` object is started. This is because the routing controllers for the sub-applications need to be active before Backbone routing is activated. That way, by the time Backbone starts processing routing events (e.g. parsing the URL fragment in the current URL), the sub-applications are ready to react and route to the proper action. So let's require the `Contacts` app before starting the `History`:

## app.js

---

```

1 ContactManager.on("initialize:after", function(){
2     if(Backbone.history){
3         require(["apps/contacts/contacts_app"], function () {
4             Backbone.history.start();
5
6             if(ContactManager.getCurrentRoute() === ""){
7                 ContactManager.trigger("contacts:list");
8             }
9         });
10    }
11 });

```

---

Note that due to the change in scope, we've change this on line 6 to ContactManager. If we refresh our page, we can see the contacts are properly dumped to the console. Great!



Note that our "contacts" module provides a return value, but we're not referencing it because we don't need direct access to it: we simply need it to be loaded.

## Requiring Views and Templates

So far, so good. But now, let's display our contacts within a composite view. What are we going to need for that to happen? First, we'll add the templates to our page:

## index.html

---

```

1 <!DOCTYPE html>
2 <html lang="en">
3     <head>
4         <meta charset="utf-8">
5         <title>Marionette Contact Manager</title>
6         <link href=".assets/css/bootstrap.css" rel="stylesheet">
7         <link href=".assets/css/application.css" rel="stylesheet">
8         <link href=".assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
9     </head>
10
11     <body>

```

---

```
12   <div id="header-region"></div>
13
14   <div id="main-region" class="container">
15     <p>Here is static content in the web page. You'll notice that it gets
16       replaced by our app as soon as we start it.</p>
17   </div>
18
19   <div id="dialog-region"></div>
20
21   <script type="text/template" id="contact-list">
22     <thead>
23       <tr>
24         <th>First Name</th>
25         <th>Last Name</th>
26         <th></th>
27       </tr>
28     </thead>
29     <tbody>
30     </tbody>
31   </script>
32
33   <script type="text/template" id="contact-list-none">
34     <td colspan="3">No contacts to display.</td>
35   </script>
36
37   <script type="text/template" id="contact-list-item">
38     <td><%= firstName %></td>
39     <td><%= lastName %></td>
40     <td>
41       <a href="#contacts/<%= id %>" class="btn btn-small js-show">
42         <i class="icon-eye-open"></i>
43         Show
44       </a>
45       <a href="#contacts/<%= id %>/edit" class="btn btn-small js-edit">
46         <i class="icon-pencil"></i>
47         Edit
48       </a>
49       <button class="btn btn-small js-delete">
50         <i class="icon-remove"></i>
51         Delete
52       </button>
53     </td>
```

```

54    </script>
55
56    <script type="text/template" id="contact-list-layout">
57      <div id="panel-region"></div>
58      <div id="contacts-region"></div>
59    </script>
60
61    <script type="text/template" id="contact-list-panel">
62      <button class="btn btn-primary js-new">New contact</button>
63      <form class="form-search form-inline pull-right">
64        <div class="input-append">
65          <input type="text" class="span2 search-query js-filter-criterion">
66          <button type="submit" class="btn js-filter">Filter contacts</button>
67        </div>
68      </form>
69    </script>
70
71    <script data-main="./assets/js/require_main.js"
72           src="./assets/js/vendor/require.js"></script>
73  </body>
74 </html>

```

---

As you can see on lines 21-69, we've simply included the relevant templates from the original application. Then, we need to turn our view code into a module. To do so, we're going to change the original code somewhat: we'll have a separate Marionette module to hold our views. This will enable us to have an explicit return value for the module, and will conform to the "one module per file" maxim:

`assets/js/apps/contacts/list/list_view.js`

```

1 define(["app"], function(ContactManager){
2   ContactManager.module("ContactsApp.List.View", function(View, ContactManager,
3                                         Backbone, Marionette, $, _){
4     View.Layout = Marionette.Layout.extend({
5       template: "#contact-list-layout",
6
7       regions: {
8         panelRegion: "#panel-region",
9         contactsRegion: "#contacts-region"
10      }
11    });

```

```
12
13     View.Panel = Marionette.ItemView.extend({
14         template: "#contact-list-panel",
15
16         triggers: {
17             "click button.js-new": "contact:new"
18         },
19
20         events: {
21             "click button.js-filter": "filterClicked"
22         },
23
24         ui: {
25             criterion: "input.js-filter-criterion"
26         },
27
28         filterClicked: function(){
29             var criterion = this.$(".js-filter-criterion").val();
30             this.trigger("contacts:filter", criterion);
31         },
32
33         onSetFilterCriterion: function(criterion){
34             $(this.ui.criterion).val(criterion);
35         }
36     });
37
38     View.Contact = Backbone.Marionette.ItemView.extend({
39         tagName: "tr",
40         template: "#contact-list-item",
41
42         events: {
43             "click": "highlightName",
44             "click td a.js-show": "showClicked",
45             "click td a.js-edit": "editClicked",
46             "click button.js-delete": "deleteClicked"
47         },
48
49         flash: function(cssClass){
50             var $view = this.$el;
51             $view.hide().toggleClass(cssClass).fadeIn(800, function(){
52                 setTimeout(function(){
53                     $view.toggleClass(cssClass)
```

```
54      }, 500);
55  });
56 },
57
58 highlightName: function(e){
59   this.$el.toggleClass("warning");
60 },
61
62 showClicked: function(e){
63   e.preventDefault();
64   e.stopPropagation();
65   this.trigger("contact:show", this.model);
66 },
67
68 editClicked: function(e){
69   e.preventDefault();
70   e.stopPropagation();
71   this.trigger("contact:edit", this.model);
72 },
73
74 deleteClicked: function(e){
75   e.stopPropagation();
76   this.trigger("contact:delete", this.model);
77 },
78
79 remove: function(){
80   this.$el.fadeOut(function(){
81     $(this).remove();
82   });
83 }
84 });
85
86 var NoContactsView = Backbone.Marionette.ItemView.extend({
87   template: "#contact-list-none",
88   tagName: "tr",
89   className: "alert"
90 });
91
92 View.Contacts = Backbone.Marionette.CompositeView.extend({
93   tagName: "table",
94   className: "table table-hover",
95   template: "#contact-list",
```

```
96     emptyView: NoContactsView,
97     itemView: View.Contact,
98     itemViewContainer: "tbody",
99
100    initialize: function(){
101        this.listenTo(this.collection, "reset", function(){
102            this.appendHtml = function(collectionView, itemView, index){
103                collectionView.$el.append(itemView.el);
104            }
105        });
106    },
107
108    onCompositeCollectionRendered: function(){
109        this.appendHtml = function(collectionView, itemView, index){
110            collectionView.$el.prepend(itemView.el);
111        }
112    }
113 });
114 });
115
116 return ContactManager.ContactsApp.List.View;
117});
```

---

On line 2, we're declaring the `ContactsApp.List.View` sub-module instead of `ContactsApp.List`. In keeping with the logic, we're using `View` as the callback argument, and all the views hang off of that object.

With this new RequireJS module implemented, our controller module in `assets/js/apps/contacts/list/list_controller.js` will be able to:

- require the RequireJS module containing the views it needs from the `assets/js/apps/contacts/list/list_view.js` file;
- return the `ContactManager.ContactsApp.List.Controller` value.

Let's now use our views from within the controller:

assets/js/apps/contacts/list/list\_controller.js

```
1 define(["app", "apps/contacts/list/list_view"], function(ContactManager, View){
2   ContactManager.module("ContactsApp.List", function(List, ContactManager,
3                                         Backbone, Marionette, $, _){
4     List.Controller = {
5       listContacts: function(){
6         require(["entities/contact"], function(){
7           var fetchingContacts = ContactManager.request("contact:entities");
8
9           var contactsListLayout = new View.Layout();
10          var contactsListPanel = new View.Panel();
11
12          $.when(fetchingContacts).done(function(contacts){
13            var contactsListView = new View.Contacts({
14              collection: contacts
15            });
16
17            contactsListLayout.on("show", function(){
18              contactsListLayout.panelRegion.show(contactsListPanel);
19              contactsListLayout.contactsRegion.show(contactsListView);
20            });
21
22            contactsListView.on("itemview:contact:delete",
23                                function(childView, model){
24                  model.destroy();
25                });
26
27              ContactManager.mainRegion.show(contactsListLayout);
28            });
29          });
30        }
31      });
32    });
33
34    return ContactManager.ContactsApp.List.Controller;
35  });
});
```

As you can see, we're declaring our module depends on the views, and we're returning the module at the end. And now, if we refresh the page, we finally have something to display!

First Name	Last Name			
Alice	Arten	Show	Edit	Delete
Bob	Brigham	Show	Edit	Delete
Charlie	Campbell	Show	Edit	Delete

Displaying our contacts

Before we consider ourselves done with displaying the contacts collection, let's clean up our code a bit more. We're using RequireJS to load dependencies, and templates are a dependency for the view, right? Let's use RequireJS to load the view templates, so we can remove them from the `index.html`.

First, we need a template loader, which we can get [here<sup>25</sup>](#) and save in `assets/js/vendor/tpl.js`. Then, we need to tell RequireJS about it (line 10):

### require\_main.js

```

1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     backbone: "vendor/backbone",
5     jquery: "vendor/jquery",
6     "jquery-ui": "vendor/jquery-ui",
7     json2: "vendor/json2",
8     localstorage: "vendor/backbone.localStorage",
9     marionette: "vendor/backbone.marionette",
10    tpl: "vendor/tpl",
11    underscore: "vendor/underscore"
12  },
13
14  shim: {
15    // edited for brevity
16  }
17 });
18

```

<sup>25</sup><https://raw.github.com/ZeeAgency/requirejs-tpl/master/tpl.js>

```
19 require(["app"], function(ContactManager){  
20   ContactManager.start();  
21 });
```

---

Now, we can save our templates to independent files, and require them with the view file. In addition to allowing us to save each template in its own file, we no longer need HTML identifiers, because we are going to directly provide compiled templates to our views.

So let's start by moving all of our templates to separate files within a *templates* sub-folder.

assets/js/apps/contacts/list/templates/layout.tpl

```
1 <div id="panel-region"></div>  
2 <div id="contacts-region"></div>
```

---

assets/js/apps/contacts/list/templates/panel.tpl

```
1 <button class="btn btn-primary js-new">New contact</button>  
2 <form class="form-search form-inline pull-right">  
3   <div class="input-append">  
4     <input type="text" class="span2 search-query js-filter-criterion">  
5     <button type="submit" class="btn js-filter">Filter contacts</button>  
6   </div>  
7 </form>
```

---

assets/js/apps/contacts/list/templates/list.tpl

```
1 <thead>  
2   <tr>  
3     <th>First Name</th>  
4     <th>Last Name</th>  
5     <th></th>  
6   </tr>  
7 </thead>  
8 <tbody>  
9 </tbody>
```

---

assets/js/apps/contacts/list/templates/list\_item.tpl

---

```

1 <td><%= firstName %></td>
2 <td><%= lastName %></td>
3 <td>
4   <a href="#contacts/<%= id %>" class="btn btn-small js-show">
5     <i class="icon-eye-open"></i>
6     Show
7   </a>
8   <a href="#contacts/<%= id %>/edit" class="btn btn-small js-edit">
9     <i class="icon-pencil"></i>
10    Edit
11   </a>
12   <button class="btn btn-small js-delete">
13     <i class="icon-remove"></i>
14   Delete
15   </button>
16 </td>
```

---

assets/js/apps/contacts/list/templates/none.tpl

---

```

1 <td colspan="3">No contacts to display.</td>
```

---

With the templates ready, let's modify our *list\_view.js* file to use compiled templates as dependencies:

assets/js/apps/contacts/list/list\_view.js

---

```

1 define(["app",
2   "tpl!apps/contacts/list/templates/layout.tpl",
3   "tpl!apps/contacts/list/templates/panel.tpl",
4   "tpl!apps/contacts/list/templates/none.tpl",
5   "tpl!apps/contacts/list/templates/list.tpl",
6   "tpl!apps/contacts/list/templates/list_item.tpl"],
7   function(ContactManager, layoutTpl, panelTpl,
8         noneTpl, listTpl, listItemTpl){
9     ContactManager.module("ContactsApp.List.View", function(View, ContactManager,
10                           Backbone, Marionette, $, _){
11       View.Layout = Marionette.Layout.extend({
```

```
12     template: layoutTpl,
13
14     regions: {
15         panelRegion: "#panel-region",
16         contactsRegion: "#contacts-region"
17     }
18 });
19
20 View.Panel = Marionette.ItemView.extend({
21     template: panelTpl,
22
23     triggers: {
24         "click button.js-new": "contact:new"
25     },
26
27     events: {
28         "click button.js-filter": "filterClicked"
29     },
30
31     ui: {
32         criterion: "input.js-filter-criterion"
33     },
34
35     filterClicked: function(){
36         var criterion = this.$(".js-filter-criterion").val();
37         this.trigger("contacts:filter", criterion);
38     },
39
40     onSetFilterCriterion: function(criterion){
41         $(this.ui.criterion).val(criterion);
42     }
43 });
44
45 View.Contact = Backbone.Marionette.ItemView.extend({
46     tagName: "tr",
47     template: listItemTpl,
48
49     events: {
50         "click": "highlightName",
51         "click td a.js-show": "showClicked",
52         "click td a.js-edit": "editClicked",
53         "click button.js-delete": "deleteClicked"
```

```
54     },
55
56     flash: function(cssClass){
57       var $view = this.$el;
58       $view.hide().toggleClass(cssClass).fadeIn(800, function(){
59         setTimeout(function(){
60           $view.toggleClass(cssClass)
61         }, 500);
62       });
63     },
64
65     highlightName: function(e){
66       this.$el.toggleClass("warning");
67     },
68
69     showClicked: function(e){
70       e.preventDefault();
71       e.stopPropagation();
72       this.trigger("contact:show", this.model);
73     },
74
75     editClicked: function(e){
76       e.preventDefault();
77       e.stopPropagation();
78       this.trigger("contact:edit", this.model);
79     },
80
81     deleteClicked: function(e){
82       e.stopPropagation();
83       this.trigger("contact:delete", this.model);
84     },
85
86     remove: function(){
87       this.$el.fadeOut(function(){
88         $(this).remove();
89       });
90     }
91   });
92
93   var NoContactsView = Backbone.Marionette.ItemView.extend({
94     template: noneTpl,
95     tagName: "tr",
```

```

96     className: "alert"
97 });
98
99 View.Contacts = Backbone.Marionette.CompositeView.extend({
100     tagName: "table",
101     className: "table table-hover",
102     template: listTpl,
103     emptyView: NoContactsView,
104     itemView: View.Contact,
105     itemViewContainer: "tbody",
106
107     initialize: function(){
108         this.listenTo(this.collection, "reset", function(){
109             this.appendHtml = function(collectionView, itemView, index){
110                 collectionView.$el.append(itemView.el);
111             }
112         });
113     },
114
115     onCompositeCollectionRendered: function(){
116         this.appendHtml = function(collectionView, itemView, index){
117             collectionView.$el.prepend(itemView.el);
118         }
119     }
120 });
121 });
122
123     return ContactManager.ContactsApp.List.View;
124 });

```

---



This approach can also be implemented with other template types, provided you have the right RequireJS template loader plugin. For example, a template loader plugin for handlebars can be found [here<sup>26</sup>](#).

And of course, we remove our templates from the `index.html` to return it back to normal:

---

<sup>26</sup><https://github.com/SlexAxton/require-handlebars-plugin>

**index.html**

---

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Marionette Contact Manager</title>
6     <link href=".assets/css/bootstrap.css" rel="stylesheet">
7     <link href=".assets/css/application.css" rel="stylesheet">
8     <link href=".assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
9   </head>
10
11  <body>
12    <div id="header-region"></div>
13
14    <div id="main-region" class="container">
15      <p>Here is static content in the web page. You'll notice that it gets
16        replaced by our app as soon as we start it.</p>
17    </div>
18
19    <div id="dialog-region"></div>
20
21    <script data-main=".assets/js/require_main.js"
22          src=".assets/js/vendor/require.js"></script>
23  </body>
24 </html>
```

---

When we refresh the page, our contacts are still displayed properly, but our code is much more modular. Success!



Git commit displaying the contacts in a composite view:

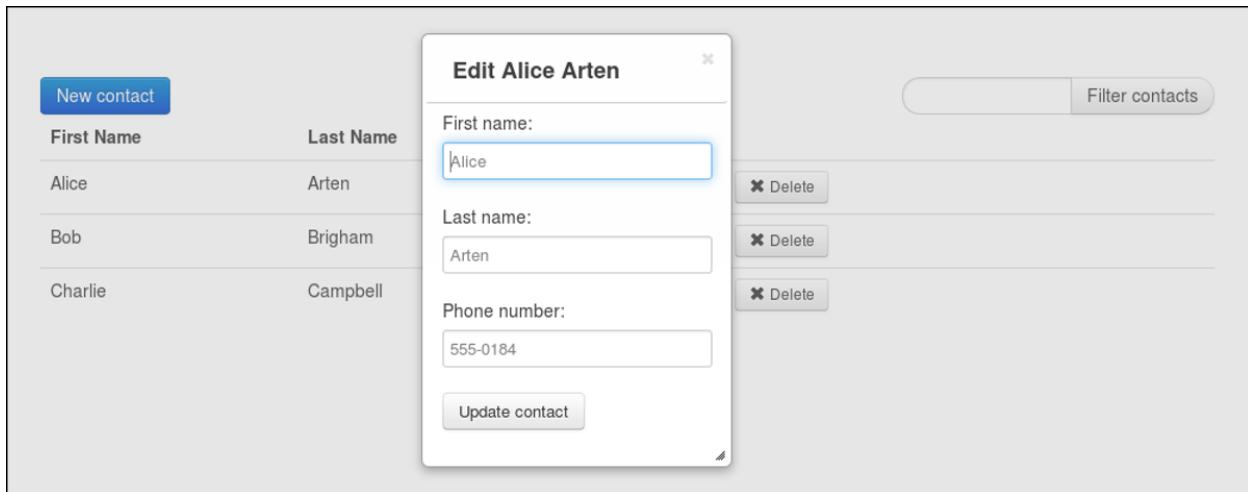
210d0621a67ed0470aa870612011fd4b609dfce<sup>27</sup>

---

<sup>27</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/210d0621a67ed0470aa870612011fd4b609dfce>

# Editing Contacts with a Modal

Let's continue adding functionality to our "list" view, by enabling users to edit contacts from a modal window.



Editing contact "Alice"

The first thing we want, is our application to react when the "edit" button is clicked (lines 8-10):

assets/js/apps/contacts/list/list\_controller.js

```
1 //edited for brevity
2
3 contactsListLayout.on("show", function(){
4     contactsListLayout.panelRegion.show(contactsListPanel);
5     contactsListLayout.contactsRegion.show(contactsListView);
6 });
7
8 contactsListView.on("itemview:contact:edit", function(childView, model){
9     console.log("edit button clicked");
10 });
11
12 contactsListView.on("itemview:contact:delete", function(childView, model){
13     model.destroy();
14 });
15
16 //edited for brevity
```

When we refresh the page and click an “edit” button, we can see our message in the console. Great, now let’s display the modal window, using the original code:

assets/js/apps/contacts/list/list\_controller.js

```
1 contactsListView.on("itemview:contact:edit", function(childView, model){  
2     var view = new ContactManager.ContactsApp.Edit.Contact({  
3         model: model  
4     });  
5  
6     view.on("form:submit", function(data){  
7         if(model.save(data)){  
8             childView.render();  
9             view.trigger("dialog:close");  
10            childView.flash("success");  
11        }  
12    }  
13    else{  
14        view.triggerMethod("form:data:invalid", model.validationError);  
15    }  
16});  
17 ContactManager.dialogRegion.show(view);  
18});
```

---

If we click an “edit” button after refreshing, we’ll have an error: “TypeError: ContactManager.ContactsApp.Edit is undefined”, due to line 2 in the code above. Why? Because we haven’t required our view before trying to instantiate it!

Naturally, before we can include the “edit” view, we need to turn it into a module. Here we go:

assets/js/apps/contacts/edit/edit\_view.js

```
1 define(["app"], function(ContactManager){  
2     ContactManager.module("ContactsApp.Edit.View", function(View, ContactManager,  
3                                         Backbone, Marionette, $, _){  
4         View.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({  
5             initialize: function(){  
6                 this.title = "Edit " + this.model.get("firstName");  
7                 this.title += " " + this.model.get("lastName");  
8             }  
9         });  
10    });  
11});
```

```

8     },
9
10    onRender: function(){
11      if(this.options.generateTitle){
12        var $title = $("<h1>", { text: this.title });
13        this.$el.prepend($title);
14      }
15
16      this.$(".js-submit").text("Update contact");
17    }
18  });
19});
20
21  return ContactManager.ContactsApp.Edit.View;
22});
```

---



Don't forget to adapt the original code: we're now using the `View` sub-module (line 2) instead of the `Edit` sub-module.

Now that we've got our edit view in a module, let's require it in our list controller:

#### assets/js/apps/contacts/list/list\_controller.js

```

1 contactsListView.on("itemview:contact:edit", function(childView, model){
2   require(["apps/contacts/edit/edit_view"], function(EditView){
3     var view = new EditView.Contact({
4       model: model
5     });
6
7     view.on("form:submit", function(data){
8       if(model.save(data)){
9         childView.render();
10        view.trigger("dialog:close");
11        childView.flash("success");
12      }
13      else{
14        view.triggerMethod("form:data:invalid", model.validationError);
15      }
16    });
17  });
```

```

18     ContactManager.dialogRegion.show(view);
19   });
20 });

```

---

So now let's refresh our page and try the "edit" button again. Boom, another error: "TypeError: ContactManager.ContactsApp.Common is undefined" in the *edit\_view.js* file. If we look at this file, we can see the edit modal view extends from the common form view (line 4 in code listing above for *edit\_view.js*). So clearly, this common form view should be a dependency in our module and we should fix it:

#### Adding the common form as a dependency in assets/js/apps/contacts/edit/edit\_view.js

---

```

1 define(["app", "apps/contacts/common/views"], function(ContactManager){
2   ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
3                                         Backbone, Marionette, $, _){
4     Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
5       initialize: function(){
6         this.title = "Edit " + this.model.get("firstName");
7         this.title += " " + this.model.get("lastName");
8       },
9
10      onRender: function(){
11        if(this.options.generateTitle){
12          var $title = $("<h1>", { text: this.title });
13          this.$el.prepend($title);
14        }
15
16        this.$(".js-submit").text("Update contact");
17      }
18    });
19  });
20 });

```

---

So let's try our "edit" button again. This time, we get 2 errors, both in *edit\_view.js* on line 3 (corresponding to lin4 above):

- TypeError: ContactManager.module is not a function
- TypeError: ContactManager.ContactsApp.Common is undefined

These errors are thrown due to line 3, but our *edit\_view.js* file is actually fine. The problem we have is that we're requiring *apps/contacts/common/views.js* without converting it to a RequireJS module. In fact our *views.js* currently looks like this:

apps/contacts/common/views.js

```
1 ContactManager.module("ContactsApp.Common.Views", function(Views, ContactManager,
2                                         Backbone, Marionette, $, _){
3     Views.Form = Marionette.ItemView.extend({
4         // edited for brevity
5     });
6 });
```

So when RequireJS loads this file, it tries to call the `ContactManager.module` method. But since we're using RequireJS, we don't have a globally defined `ContactManager` Marionette application, so that fails and causes our first error. And since we weren't able to declare a new "`ContactManager.ContactsApp.Common.Views`" module (which would create the "Common" parent module if necessary), we get our second error in `edit_view.js` when trying to run this line of code:

```
Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({ . . .});
```

This, in turn, is because there is no "`ContactManager.ContactsApp.Common`" module defined, as explained above.



Note that the error were raised in `assets/js/apps/contacts/edit/edit_view.js`, but were actually due to our code in `apps/contacts/common/views.js`. It's important you don't just take the errors at face value, and take the time to trace the code to see what is really happening. And once again, all of this debugging is made *much* easier when working in shorter iterations!

Now that we have a better grasp of the errors raised by RequireJS, let's fix them.



Turn the common form view "`apps/contacts/common/views.js`" into a RequireJS module. Don't forget you'll need to extract (and require) a template.

You can see the solution on the next page.

First, let's make ourselves a template:

assets/js/apps/contacts/common/templates/form.tpl

---

```

1 <form>
2   <div class="control-group">
3     <label for="contact-firstName" class="control-label">First name:</label>
4     <input id="contact-firstName" name="firstName"
5        type="text" value="<%>= firstName %>"/>
6   </div>
7   <div class="control-group">
8     <label for="contact-lastName" class="control-label">Last name:</label>
9     <input id="contact-lastName" name="lastName"
10       type="text" value="<%>= lastName %>"/>
11   </div>
12   <div class="control-group">
13     <label for="contact-phoneNumber" class="control-label">Phone number:</label>
14     <input id="contact-phoneNumber" name="phoneNumber"
15       type="text" value="<%>= phoneNumber %>"/>
16   </div>
17   <button class="btn js-submit">Save</button>
18 </form>

```

---

And now, we'll turn our common form view into a module:

apps/contacts/common/views.js

---

```

1 define(["app", "tpl!apps/contacts/common/templates/form.tpl"],
2       function(ContactManager, formTpl){
3         ContactManager.module("ContactsApp.Common.Views", function(Views,
4                           ContactManager, Backbone, Marionette, $, _){
5           Views.Form = Marionette.ItemView.extend({
6             template: formTpl,
7
8             events: {
9               "click button.js-submit": "submitClicked"
10            },
11
12            submitClicked: function(e){
13              e.preventDefault();
14              var data = Backbone.Syphon.serialize(this);

```

```
15      this.trigger("form:submit", data);
16  },
17
18  onFormDataInvalid: function(errors){
19    var $view = this.$el;
20
21    var clearFormErrors = function(){
22      var $form = $view.find("form");
23      $form.find(".help-inline.error").each(function(){
24        $(this).remove();
25      });
26      $form.find(".control-group.error").each(function(){
27        $(this).removeClass("error");
28      });
29    }
30
31    var markErrors = function(value, key){
32      var $controlGroup = $view.find("#contact-" + key).parent();
33      var $errorEl = $("<span>", { class: "help-inline error", text: value });
34      $controlGroup.append($errorEl).addClass("error");
35    }
36
37    clearFormErrors();
38    _.each(errors, markErrors);
39  }
40 });
41 });
42
43 return ContactManager.ContactsApp.Common.Views;
44});
```

---

Now, if we refresh and click “edit”, we FINALLY see our modal window. But if we change a contact’s name and click “Update contact”, we get an error: “TypeError: Backbone.Syphon is undefined”...



Fix our app so that we can properly update contacts.

You can see the solution on the next page.

Naturally, the solution is to add Backbone.Syphon to our *require\_main.js* file, and declare it as a dependency for our common contact form.

---

#### Adding Backbone.Syphon to assets/js/require\_main.js

---

```

1 requirejs.config({
2     baseUrl: "assets/js",
3     paths: {
4         backbone: "vendor/backbone",
5         "backbone.syphon": "vendor/backbone.syphon",
6         // edited for brevity
7     },
8
9     shim: {
10        // edited for brevity
11        backbone: {
12            deps: ["jquery", "underscore", "json2"],
13            exports: "Backbone"
14        },
15        "backbone.syphon": ["backbone"],
16        // edited for brevity
17    }
18 });
19
20 require(["app"], function(ContactManager){
21     ContactManager.start();
22 });

```

---



---

#### Declaring Backbone.Syphon as a dependency (apps/contacts/common/views.js)

---

```

1 define(["app", "tpl!apps/contacts/common/templates/form.tpl", "backbone.syphon"],
2       function(ContactManager, formTpl){
3         ContactManager.module("ContactsApp.Common.Views", function(Views,
4                           ContactManager, Backbone, Marionette, $, _){
5             Views.Form = Marionette.ItemView.extend({
6                 // edited for brevity
7             });
8         });
9
10        return ContactManager.ContactsApp.Common.Views;
11    });

```

---

And now our contact editing functionality works as expected!

## Expliciting Dependencies

Note that the module we've defined in *apps/contacts/common/views.js* returns the module definition. Therefore, we can adapt our code in *assets/js/apps/contacts/edit/edit\_view.js* to take advantage of that fact.

Original assets/js/apps/contacts/edit/edit\_view.js

---

```

1 define(["app", "apps/contacts/common/views"], function(ContactManager){
2   ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
3                                         Backbone, Marionette, $, _){
4     Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
5       // edited for brevity
6     });
7   });
8 });

```

---

Improved assets/js/apps/contacts/edit/edit\_view.js

---

```

1 define(["app", "apps/contacts/common/views"],
2       function(ContactManager, CommonViews){
3     ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
4                                         Backbone, Marionette, $, _){
5       Edit.Contact = CommonViews.Form.extend({
6         // edited for brevity
7       });
8     });
9 });

```

---

Here, we've made use of the module definition's return value on line 2, and used it on line 5. Our app worked fine before, so why are we changing this? It makes dependencies obvious: in the second version, you can clearly see that the `Form` we're using is defined in the `CommonViews` module from file *apps/contacts/common/views.js*. But both code versions are functionally equivalent, and neither introduces any global variables (thanks to our using Marionette modules to encapsulate everything). Ultimately, the version you chose will come down to personal taste.



Git commit implementing contact editing from a modal window:

[6fa8341707ccced8e9aaa227ff5594a41435e5f<sup>28</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/6fa8341707ccced8e9aaa227ff5594a41435e5f)

---

<sup>28</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/6fa8341707ccced8e9aaa227ff5594a41435e5f>

# Creating Contacts with a Modal



Implement the required functionality so users can create new contacts. You'll find the solution below.

## Exercise Solution

First, we need to refactor our “new” view into a RequireJS module:

assets/js/apps/contacts/new/new\_view.js

---

```
1 define(["app", "apps/contacts/common/views"],
2         function(ContactManager, CommonViews){
3     ContactManager.module("ContactsApp.New.View", function(View, ContactManager,
4                                         Backbone, Marionette, $, _){
5         View.Contact = CommonViews.Form.extend({
6             title: "New Contact",
7
8             onRender: function(){
9                 this.$(".js-submit").text("Create contact");
10            }
11        });
12    });
13
14    return ContactManager.ContactsApp.New.View;
15});
```

---

Then, we can restore the original behavior in the list controller, without forgetting to require the “new” view (line 7):

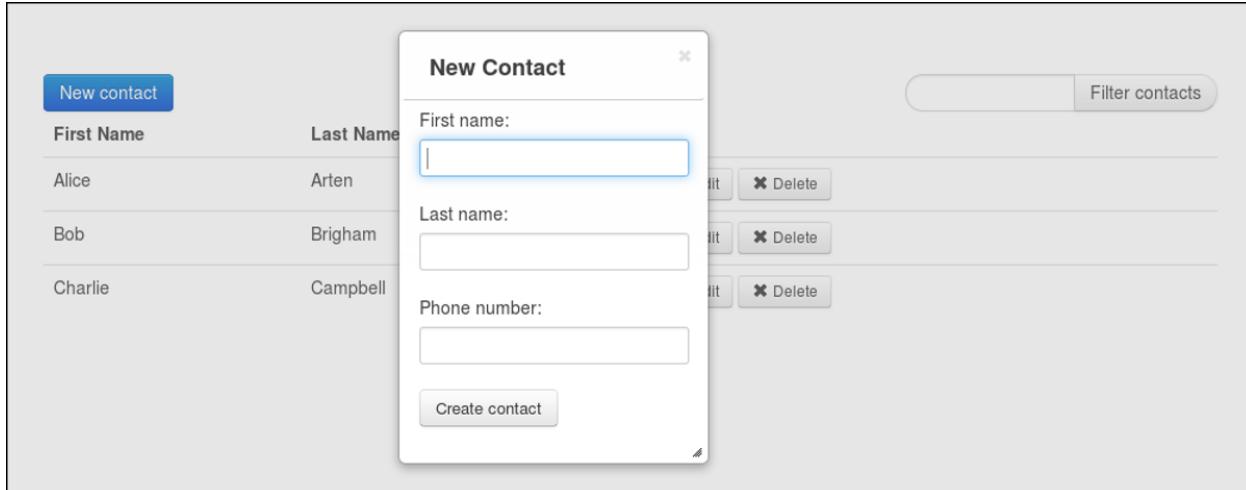
assets/js/apps/contacts/list/list\_controller.js

```
1 contactsListLayout.on("show", function(){
2     contactsListLayout.panelRegion.show(contactsListPanel);
3     contactsListLayout.contactsRegion.show(contactsListView);
4 });
5
6 contactsListPanel.on("contact:new", function(){
7     require(["apps/contacts/new/new_view"], function(NewView){
8         var newContact = new ContactManager.Entities.Contact();
9
10        var view = new NewView.Contact({
11            model: newContact
12        });
13
14        view.on("form:submit", function(data){
15            if(contacts.length > 0){
16                var highestId = contacts.max(function(c){ return c.id; }).get("id");
17                data.id = highestId + 1;
18            }
19            else{
20                data.id = 1;
21            }
22            if(newContact.save(data)){
23                contacts.add(newContact);
24                view.trigger("dialog:close");
25                var newContactView = contactsListView.children.findByModel(newContact);
26                if(newContactView){
27                    newContactView.flash("success");
28                }
29            }
30            else{
31                view.triggerMethod("form:data:invalid", newContact.validationError);
32            }
33        });
34
35        ContactManager.dialogRegion.show(view);
36    });
37 });
38
39 contactsListView.on("itemview:contact:edit", function(childView, model){
```

```
40 // edited for brevity
41});
```

---

Now when we click on “New contact” after refreshing, we’ve got our modal window.



Creating a new contact

## Further Decoupling of the Contact Module

If you take a look at the code above, we’re explicitly creating a new contact on line 8. This rubs me the wrong way, because we said [earlier](#) that we’d only interact with the contact entities by using the request-response mechanism. So let’s fix that (line 3):

assets/js/apps/contacts/list/list\_controller.js

```
1 contactsListPanel.on("contact:new", function(){
2   require(["apps/contacts/new/new_view"], function(NewView){
3     var newContact = ContactManager.request("contact:entity:new");
4
5     var view = new NewView.Contact({
6       model: newContact
7     });
8
9     view.on("form:submit", function(data){
10       if(contacts.length > 0){
11         var highestId = contacts.max(function(c){ return c.id; }).get("id");
12         data.id = highestId + 1;
13       }
14     });
15   });
16 });
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
219
220
221
222
223
224
225
226
227
228
229
229
230
231
232
233
234
235
236
237
238
239
239
240
241
242
243
244
245
246
247
247
248
249
249
250
251
252
253
254
255
256
256
257
258
259
259
260
261
262
263
264
265
266
267
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
311
312
313
313
314
315
315
316
317
317
318
319
319
320
321
321
322
323
323
324
325
325
326
327
327
328
329
329
330
331
331
332
333
333
334
335
335
336
337
337
338
339
339
340
341
341
342
343
343
344
345
345
346
347
347
348
349
349
350
351
351
352
353
353
354
355
355
356
357
357
358
359
359
360
361
361
362
363
363
364
365
365
366
367
367
368
369
369
370
371
371
372
373
373
374
375
375
376
377
377
378
379
379
380
381
381
382
383
383
384
385
385
386
387
387
388
389
389
390
391
391
392
393
393
394
395
395
396
397
397
398
399
399
400
401
401
402
403
403
404
405
405
406
407
407
408
409
409
410
411
411
412
413
413
414
415
415
416
417
417
418
419
419
420
421
421
422
423
423
424
425
425
426
427
427
428
429
429
430
431
431
432
433
433
434
435
435
436
437
437
438
439
439
440
441
441
442
443
443
444
445
445
446
447
447
448
449
449
450
451
451
452
453
453
454
455
455
456
457
457
458
459
459
460
461
461
462
463
463
464
465
465
466
467
467
468
469
469
470
471
471
472
473
473
474
475
475
476
477
477
478
479
479
480
481
481
482
483
483
484
485
485
486
487
487
488
489
489
490
491
491
492
493
493
494
495
495
496
497
497
498
499
499
500
501
501
502
503
503
504
505
505
506
507
507
508
509
509
510
511
511
512
513
513
514
515
515
516
517
517
518
519
519
520
521
521
522
523
523
524
525
525
526
527
527
528
529
529
530
531
531
532
533
533
534
535
535
536
537
537
538
539
539
540
541
541
542
543
543
544
545
545
546
547
547
548
549
549
550
551
551
552
553
553
554
555
555
556
557
557
558
559
559
560
561
561
562
563
563
564
565
565
566
567
567
568
569
569
570
571
571
572
573
573
574
575
575
576
577
577
578
579
579
580
581
581
582
583
583
584
585
585
586
587
587
588
589
589
590
591
591
592
593
593
594
595
595
596
597
597
598
599
599
600
601
601
602
603
603
604
605
605
606
607
607
608
609
609
610
611
611
612
613
613
614
615
615
616
617
617
618
619
619
620
621
621
622
623
623
624
625
625
626
627
627
628
629
629
630
631
631
632
633
633
634
635
635
636
637
637
638
639
639
640
641
641
642
643
643
644
645
645
646
647
647
648
649
649
650
651
651
652
653
653
654
655
655
656
657
657
658
659
659
660
661
661
662
663
663
664
665
665
666
667
667
668
669
669
670
671
671
672
673
673
674
675
675
676
677
677
678
679
679
680
681
681
682
683
683
684
685
685
686
687
687
688
689
689
690
691
691
692
693
693
694
695
695
696
697
697
698
699
699
700
701
701
702
703
703
704
705
705
706
707
707
708
709
709
710
711
711
712
713
713
714
715
715
716
717
717
718
719
719
720
721
721
722
723
723
724
725
725
726
727
727
728
729
729
730
731
731
732
733
733
734
735
735
736
737
737
738
739
739
740
741
741
742
743
743
744
745
745
746
747
747
748
749
749
750
751
751
752
753
753
754
755
755
756
757
757
758
759
759
760
761
761
762
763
763
764
765
765
766
767
767
768
769
769
770
771
771
772
773
773
774
775
775
776
777
777
778
779
779
780
781
781
782
783
783
784
785
785
786
787
787
788
789
789
790
791
791
792
793
793
794
795
795
796
797
797
798
799
799
800
801
801
802
803
803
804
805
805
806
807
807
808
809
809
810
811
811
812
813
813
814
815
815
816
817
817
818
819
819
820
821
821
822
823
823
824
825
825
826
827
827
828
829
829
830
831
831
832
833
833
834
835
835
836
837
837
838
839
839
840
841
841
842
843
843
844
845
845
846
847
847
848
849
849
850
851
851
852
853
853
854
855
855
856
857
857
858
859
859
860
861
861
862
863
863
864
865
865
866
867
867
868
869
869
870
871
871
872
873
873
874
875
875
876
877
877
878
879
879
880
881
881
882
883
883
884
885
885
886
887
887
888
889
889
890
891
891
892
893
893
894
895
895
896
897
897
898
899
899
900
901
901
902
903
903
904
905
905
906
907
907
908
909
909
910
911
911
912
913
913
914
915
915
916
917
917
918
919
919
920
921
921
922
923
923
924
925
925
926
927
927
928
929
929
930
931
931
932
933
933
934
935
935
936
937
937
938
939
939
940
941
941
942
943
943
944
945
945
946
947
947
948
949
949
950
951
951
952
953
953
954
955
955
956
957
957
958
959
959
960
961
961
962
963
963
964
965
965
966
967
967
968
969
969
970
971
971
972
973
973
974
975
975
976
977
977
978
979
979
980
981
981
982
983
983
984
985
985
986
987
987
988
989
989
990
991
991
992
993
993
994
995
995
996
997
997
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
15
```

```

13     }
14     else{
15         data.id = 1;
16     }
17     if(newContact.save(data)){
18         contacts.add(newContact);
19         view.trigger("dialog:close");
20         var newContactView = contactsListView.children.findByModel(newContact);
21         if(newContactView){
22             newContactView.flash("success");
23         }
24     }
25     else{
26         view.triggerMethod("form:data:invalid", newContact.validationError);
27     }
28 });
29
30     ContactManager.dialogRegion.show(view);
31 });
32 });

```

---

Of course, this means we need to register a handler to return a new contact instance (lines 5-7):

---

#### assets/js/entities/contact.js

---

```

1 ContactManager.reqres.setHandler("contact:entity", function(id){
2     return API.getContactEntity(id);
3 });
4
5 ContactManager.reqres.setHandler("contact:entity:new", function(id){
6     return new Entities.Contact();
7 });

```

---

And now, we've got our code cleaned up: we only interact with our entities via the request-response mechanism. Note that even though we don't need a direct reference to the contact entity in the list controller, we still need to require it in our code so the response handler is registered by the time we need it.



Git commit implementing contact creation from a modal window:

[7313005bb3de50092aea2ec0736a8e0afe1c3c53<sup>29</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/7313005bb3de50092aea2ec0736a8e0afe1c3c53)

---

<sup>29</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/7313005bb3de50092aea2ec0736a8e0afe1c3c53>

# Filtering Contacts



Once again, you're on your own: implement the filtering functionality in our app, so users can filter the contacts displayed on the "list" page. Don't forget to wire up the routing! I suggest you implement the functionality in 3 stages:

1. Get the filtering working;
2. Update the URL fragment when a user filters the list;
3. Set the application to the proper initial state when a route with a filter criterion is triggered.

And as usual, the solution follows.

## Exercise Solution

We're going to need our "filtered collection" entity, so let's turn it into a RequireJS module:

assets/js/entities/common.js

---

```
1 define(["app"], function(ContactManager){  
2   ContactManager.module("Entities", function(Entities, ContactManager,  
3                                         Backbone, Marionette, $, _){  
4     Entities.FilteredCollection = function(options){  
5       // edited for brevity  
6     };  
7   });  
8  
9   return ContactManager.Entities.FilteredCollection;  
10});
```

---

Now, we can use it in our controller (line 12):

Using the filtered collection (assets/js/apps/contacts/list/list\_controller.js)

```
1 define(["app", "apps/contacts/list/list_view"], function(ContactManager, View){
2     ContactManager.module("ContactsApp.List", function(List, ContactManager,
3                                         Backbone, Marionette, $, _){
4         List.Controller = {
5             listContacts: function(criterion){
6                 require(["entities/contact"], function(){
7                     var fetchingContacts = ContactManager.request("contact:entities");
8
9                     var contactsListLayout = new View.Layout();
10                    var contactsListPanel = new View.Panel();
11
12                    require(["entities/common"], function(FilteredCollection){
13                        $.when(fetchingContacts).done(function(contacts){
14                            var filteredContacts = ContactManager.Entities.FilteredCollection({
15                                collection: contacts,
16                                filterFunction: function(filterCriterion){
17                                    var criterion = filterCriterion.toLowerCase();
18                                    return function(contact){
19                                        if(contact.get("firstName").toLowerCase().
20                                            indexOf(criterion) !== -1
21                                            || contact.get("lastName").toLowerCase().
22                                                indexOf(criterion) !== -1
23                                            || contact.get("phoneNumber").toLowerCase().
24                                                indexOf(criterion) !== -1){
25                                            return contact;
26                                        }
27                                    };
28                                }
29                            });
30
31                            var contactsListView = new View.Contacts({
32                                collection: filteredContacts
33                            });
34
35                            contactsListPanel.on("contacts:filter", function(filterCriterion){
36                                filteredContacts.filter(filterCriterion);
37                            });
38
39                            contactsListLayout.on("show", function(){
40                                contactsListLayout.panelRegion.show(contactsListPanel);
```

```
41         contactsListLayout.contactsRegion.show(contactsListView);
42     });
43
44     // edited for brevity
```

---

And now we've got our filtering working. Let's move on to the next piece of the puzzle: updating the URL fragment to indicate the current filter criterion. For that bit of magic, we'll simply restore the original sections of code: we need to trigger an event indicating the user has filtered the list, and we need our router to respond appropriately (namely updating the URL fragment).

#### Triggering the filtering event on line 7 (assets/js/apps/contacts/list/list\_controller.js)

---

```
1 var contactsListView = new View.Contacts({
2   collection: filteredContacts
3 });
4
5 contactsListPanel.on("contacts:filter", function(filterCriterion){
6   filteredContacts.filter(filterCriterion);
7   ContactManager.trigger("contacts:filter", filterCriterion);
8 });
```

---

#### Responding to the filtering event on lines 6-13 (assets/js/apps/contacts/contacts\_app.js)

---

```
1 ContactManager.on("contacts:list", function(){
2   ContactManager.navigate("contacts");
3   API.listContacts();
4 });
5
6 ContactManager.on("contacts:filter", function(criterion){
7   if(criterion){
8     ContactManager.navigate("contacts/filter/criterion:" + criterion);
9   }
10 } else{
11   ContactManager.navigate("contacts");
12 }
13});
```

---

So now we have our URL updated when a user filters the list of contacts. The last step is to display a filtered list when the user loads a URL containing a `filter` query string. In other words, we need to restore the proper application state by using routing. First, we'll add the routing code we need:

**Adding a route to process filtering criteria (assets/js/apps/contacts/contacts\_app.js)**

---

```
1 ContactsApp.Router = Marionette.AppRouter.extend({
2     appRoutes: {
3         "contacts(/filter/criterion::criterion)": "listContacts",
4     }
5 });
6
7 var API = {
8     listContacts: function(criterion){
9         require(["apps/contacts/list/list_controller"], function(ListController){
10            ListController.listContacts(criterion);
11        });
12    }
13};
```

---

Since we're already passing on the `criterion` value (line 8) from our earlier code, that's all we need to add to our routing code. Now, we need to get our view to filter the list:

**assets/js/apps/contacts/list/list\_controller.js**

---

```
1 listContacts: function(criterion){
2     require(["entities/contact"], function(){
3         var fetchingContacts = ContactManager.request("contact:entities");
4
5         var contactsListLayout = new View.Layout();
6         var contactsListPanel = new View.Panel();
7
8         require(["entities/common"], function(FilteredCollection){
9             $.when(fetchingContacts).done(function(contacts){
10                var filteredContacts = ContactManager.Entities.FilteredCollection({
11                    // edited for brevity
12                });
13
14                if(criterion){
15                    filteredContacts.filter(criterion);
16                    contactsListPanel.once("show", function(){
17                        contactsListPanel.triggerMethod("set:filter:criterion", criterion);
18                    });
19                }
20            });
21        });
22    });
23}
```

---



Don't forget to add the `criterion` argument to the function definition on line 1!

Since the `onSetFilterCriterion` method is already defined in our panel, there's nothing left for us to do. We've got filtering working properly!

First Name	Last Name			
Alicen	Arten	Show	Edit	Delete
Charlie	Campbell	Show	Edit	Delete

Filtering contacts containing “li”



Git commit implementing contact filtering

a28419c9e8923fb6149d3f245aa1f965b0eea3b6<sup>30</sup>

---

<sup>30</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/a28419c9e8923fb6149d3f245aa1f965b0eea3b6>

# Adding the Loading View



Add the loading view to the “list” controller, but this time you won’t get any hints and will have to determine dependencies on your own... You’ll find the solution below.

## Exercise Solution

Let’s get started by creating the template file we will need:

assets/js/common/templates/loading.tpl

---

```
1 <h1><%= title %></h1>
2 <p><%= message %></p>
3 <div id="spinner"></div>
```

---

We can now use the template when converting the view file into a RequireJS module:

assets/js/common/views.js

---

```
1 define(["app", "tpl!common/templates/loading.tpl"],
2         function(ContactManager, loadingTpl){
3     ContactManager.module("Common.Views", function(Views, ContactManager,
4                                         Backbone, Marionette, $, _){
5         Views.Loading = Marionette.ItemView.extend({
6             template: loadingTpl,
7
8             initialize: function(options){
9                 var options = options || {};
10                this.title = options.title || "Loading Data";
11                this.message = options.message || "Please wait, data is loading.";
12            },
13
14            serializeData: function(){
15                return {
```

```
16         title: this.title,
17         message: this.message
18     }
19 },
20
21     onShow: function(){
22         var opts = {
23             lines: 13, // The number of lines to draw
24             length: 20, // The length of each line
25             width: 10, // The line thickness
26             radius: 30, // The radius of the inner circle
27             corners: 1, // Corner roundness (0..1)
28             rotate: 0, // The rotation offset
29             direction: 1, // 1: clockwise, -1: counterclockwise
30             color: "#000", // #rgb or #rrggbb
31             speed: 1, // Rounds per second
32             trail: 60, // Afterglow percentage
33             shadow: false, // Whether to render a shadow
34             hwaccel: false, // Whether to use hardware acceleration
35             className: "spinner", // The CSS class to assign to the spinner
36             zIndex: 2e9, // The z-index (defaults to 2000000000)
37             top: "30px", // Top position relative to parent in px
38             left: "auto" // Left position relative to parent in px
39         };
40         $("#spinner").spin(opts);
41     }
42 });
43 });
44
45     return ContactManager.Common.Views;
46 });
```

---

Sadly, this code is going to cause us problems: when we execute line 40, we won't have any library providing the `spin` method. In other words, this common loading view needs to depend on the `Spin.js` library. Let's add it to the config:

---

assets/js/require\_main.js

```

1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     // edited for brevity
5     marionette: "vendor/backbone.marionette",
6     spin: "vendor/spin",
7     "spin.jquery": "vendor/spin.jquery",
8     tpl: "vendor/tpl",
9     // edited for brevity
10 },
11
12   shim: {
13     // edited for brevity
14     localStorage: ["backbone"],
15     "spin.jquery": ["spin", "jquery"]
16   }
17 });
18
19 require(["app"], function(ContactManager){
20   ContactManager.start();
21 });

```

---

And now, we can require the Spin.js library as a dependency:

---

assets/js/common/views.js

```

1 define(["app", "tpl!common/templates/loading.tpl", "spin.jquery"],
2           function(ContactManager, loadingTpl){
3   ContactManager.module("Common.Views", function(Views, ContactManager,
4                                         Backbone, Marionette, $, _){
5     // edited for brevity
6   });
7
8   return ContactManager.Common.Views;
9 });

```

---

With our view file functional, let's use it in our list controller:

assets/js/apps/contacts/list/list\_controller.js

---

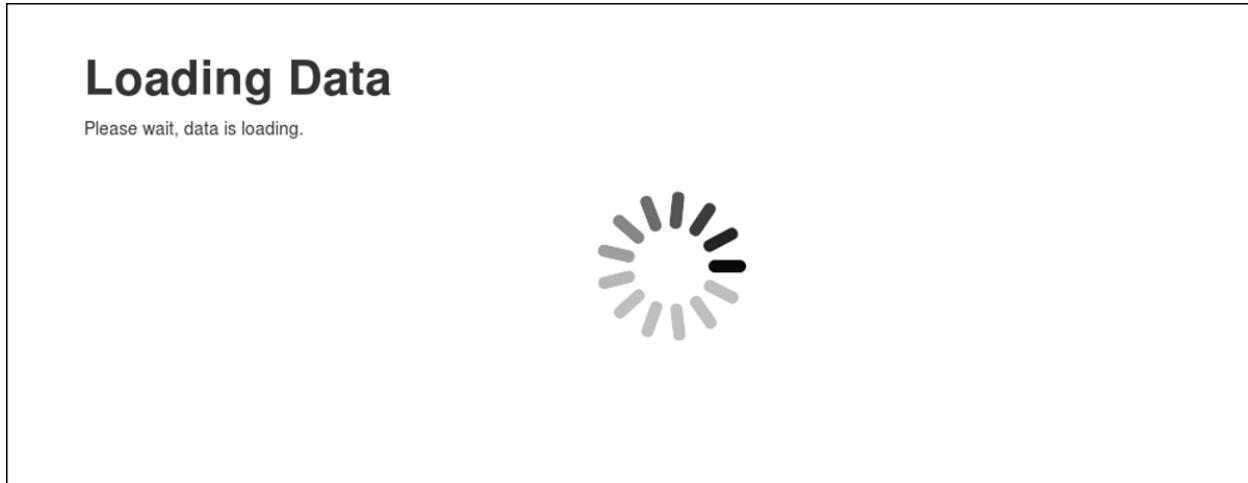
```

1 define(["app", "apps/contacts/list/list_view"], function(ContactManager, View){
2   ContactManager.module("ContactsApp.List", function(List, ContactManager,
3                                         Backbone, Marionette, $, _){
4     List.Controller = {
5       listContacts: function(criterion){
6         require(["common/views", "entities/contact"], function(CommonViews){
7           var loadingView = new CommonViews.Loading();
8           ContactManager.mainRegion.show(loadingView);
9
10          var fetchingContacts = ContactManager.request("contact:entities");
11
12          // edited for brevity
13
14        });
15      }
16    }
17  });
18 });

```

---

And now the loading view displays before our contacts are loaded and rendered:



The loading view



Git commit adding a loading view to the contact listing action:

a18839075804dc90d45ac8491df54a2ce6a5100f<sup>31</sup>

---

<sup>31</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/a18839075804dc90d45ac8491df54a2ce6a5100f>

# Showing Contacts



Implement the functionality required to display contacts when a user clicks on a “show” link. You’ll find the solution below.

## Exercise Solution

First, let’s get our list controller to react when the “show” link is clicked:

Reacting to the show link click (assets/js/apps/contacts/list/list\_controller.js)

```
1 contactsListView.on("itemview:contact:show", function(childView, model){  
2   console.log("show link clicked");  
3 };  
4  
5 contactsListView.on("itemview:contact:edit", function(childView, model){  
6   // edited for brevity  
7 });
```

With this in place, we’re able to process the click event on the “show” link. Now, we need to turn our “show” code into a RequireJS module. Let’s start by adding templates:

Template to display a contact (assets/js/apps/contacts/show/templates/view.tpl)

```
1 <h1><%= firstName %> <%= lastName %></h1>  
2 <a href="#contacts/<%= id %>/edit" class="btn btn-small js-edit">  
3   <i class="icon-pencil"></i>  
4   Edit this contact  
5 </a>  
6 <p><strong>Phone number:</strong> <%= phoneNumber %></p>
```

Template to display a missing contact (assets/js/apps/contacts/show/templates/missing.tpl)

---

```
1 <div class="alert alert-error">This contact doesn't exist !</div>
```

---

With these templates in place, we can require them for use in our view file:

Using the templates and adapting the view module for RequireJS (assets/js/apps/contacts/show/show\_view.js)

---

```
1 define(["app",
2         "tpl!apps/contacts/show/templates/missing.tpl",
3         "tpl!apps/contacts/show/templates/view.tpl"],
4         function(ContactManager, missingTpl, viewTpl){
5             ContactManager.module("ContactsApp.Show.View", function(View,
6                               ContactManager, Backbone, Marionette, $, _){
7                 View.MissingContact = Marionette.ItemView.extend({
8                     template: missingTpl
9                 });
10
11                View.Contact = Marionette.ItemView.extend({
12                    template: viewTpl,
13
14                    events: {
15                        "click a.js-edit": "editClicked"
16                    },
17
18                    editClicked: function(e){
19                        e.preventDefault();
20                        this.trigger("contact:edit", this.model);
21                    }
22                });
23            });
24
25            return ContactManager.ContactsApp.Show.View;
26        });
    
```

---

Once again, don't forget to have the views in their View sub-module (declared on line 5, used on lines 8 and 12). Let's move on to the controller:

**Adapting the show controller for RequireJS (assets/js/apps/contacts/show/show\_controller.js)**

---

```
1 define(["app", "apps/contacts/show/show_view"], function(ContactManager, View){
2     ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
3                                         Backbone, Marionette, $, _){
4         Show.Controller = {
5             showContact: function(id){
6                 require(["common/views", "entities/contact"], function(CommonViews){
7                     var loadingView = new CommonViews.Loading({
8                         title: "Artificial Loading Delay",
9                         message: "Data loading is delayed to demonstrate
10                            using a loading view."
11                     });
12                     ContactManager.mainRegion.show(loadingView);
13
14                     var fetchingContact = ContactManager.request("contact:entity", id);
15                     $.when(fetchingContact).done(function(contact){
16                         var contactView;
17                         if(contact !== undefined){
18                             contactView = new View.Contact({
19                                 model: contact
20                             });
21
22                             contactView.on("contact:edit", function(contact){
23                                 ContactManager.trigger("contact:edit", contact.get("id"));
24                             });
25                         }
26                         else{
27                             contactView = new View.MissingContact();
28                             model: contact
29                         }
30
31                         ContactManager.mainRegion.show(contactView);
32                     });
33                 });
34             }
35         });
36     });
37
38     return ContactManager.ContactsApp.Show.Controller;
39 });
```

---

Note that on line 6 we've required the module containing our loading view and the contact entity. With our "show" action ready, let's wire up the routing code:

assets/js/apps/contacts/contacts\_app.js

---

```
1 ContactsApp.Router = Marionette.AppRouter.extend({
2     appRoutes: {
3         "contacts": "listContacts",
4         "contacts(?filter=:criterion)": "listContacts",
5         "contacts/:id": "showContact"
6     }
7 });
8
9 var API = {
10     listContacts: function(criterion){
11         // edited for brevity
12     },
13
14     showContact: function(id){
15         require(["apps/contacts/show/show_controller"], function(ShowController){
16             ShowController.showContact(id);
17         });
18     }
19 };
20
21 // edited for brevity
22
23 ContactManager.on("contacts:filter", function(criterion){
24     // edited for brevity
25 });
26
27 ContactManager.on("contact:show", function(id){
28     ContactManager.navigate("contacts/" + id);
29     API.showContact(id);
30 });
```

---

All that's is triggering the proper event from our list controller:

assets/js/apps/contacts/list/list\_controller.js

```
1 contactsListView.on("itemview:contact:show", function(childView, model){  
2   ContactManager.trigger("contact:show", model.get("id"));  
3 });
```

And with that, our app displays contacts properly!

## Alicen Arten

 Edit this contact

Phone number: 555-0184

Displaying a contact

## Artificial Loading Delay

Data loading is delayed to demonstrate using a loading view.



The loading view



Git commit displaying contacts:

[8e13cf5ecd8c10d750959c5607a21f698e408407<sup>32</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/8e13cf5ecd8c10d750959c5607a21f698e408407)

---

<sup>32</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/8e13cf5ecd8c10d750959c5607a21f698e408407>

# Editing Contacts

We've now got a functional view to display our contacts, except it has a non-functional "edit this contact" button. You know what's coming, don't you?



Implement the functionality required to edit contacts when a user clicks on the "edit this contact" link in the show view. You'll find the solution below.

## Exercise Solution



This solution is quite terse and doesn't include much hand-holding beyond showing you the corrected code. This is because the implementation is nearly identical to the "show" action we've covered [earlier](#).

The button already triggers an event when it is clicked, and the "edit" view has already been turned into a RequireJS module (so we could use it in a modal window to edit contacts in the "list" view, [remember](#)?). Let's move on to turning our "edit" controller into a module:

assets/js/apps/contacts/edit/edit\_controller.js

```
1 define(["app", "apps/contacts/edit/edit_view"], function(ContactManager, View){
2   ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
3                                         Backbone, Marionette, $, _){
4     Edit.Controller = {
5       editContact: function(id){
6         require(["common/views", "entities/contact"], function(CommonViews){
7           var loadingView = new CommonViews.Loading({
8             title: "Artificial Loading Delay",
9             message: "Data loading is delayed to demonstrate
10               using a loading view."
11           });
12           ContactManager.mainRegion.show(loadingView);
13
14           var fetchingContact = ContactManager.request("contact:entity", id);
15           $.when(fetchingContact).done(function(contact){
```

```
16     var view;
17     if(contact !== undefined){
18         view = new View.Contact({
19             model: contact,
20             generateTitle: true
21         });
22
23         view.on("form:submit", function(data){
24             if(contact.save(data)){
25                 ContactManager.trigger("contact:show", contact.get("id"));
26             }
27             else{
28                 view.triggerMethod("form:data:invalid",
29                                     contact.validationError);
30             }
31         });
32     }
33     else{
34         view = new ContactManager.ContactsApp.Show.MissingContact();
35     }
36
37     ContactManager.mainRegion.show(view);
38 });
39 });
40 }
41 };
42 );
43
44 return ContactManager.ContactsApp.Edit.Controller;
45});
```

---

Then, of course, we need to add the code to our routing controller to get our app to react to the “contact:edit” event (triggered by the “edit this contact” button in the “show” view):

assets/js/apps/contacts/contacts\_app.js

```
1 appRoutes: {
2     "contacts": "listContacts",
3     "contacts(?filter=:criterion)": "listContacts",
4     "contacts/:id": "showContact",
5     "contacts/:id/edit": "editContact"
6 }
7
8 var API = {
9     // edited for brevity
10
11     showContact: function(id){
12         // edited for brevity
13     },
14
15     editContact: function(id){
16         require(["apps/contacts/edit/edit_controller"], function(EditController){
17             EditController.editContact(id);
18         });
19     }
20 };
21
22 // edited for brevity
23
24 ContactManager.on("contact:show", function(id){
25     // edited for brevity
26 });
27
28 ContactManager.on("contact:edit", function(id){
29     ContactManager.navigate("contacts/" + id + "/edit");
30     API.editContact(id);
31 });
```

And with that, we've turned our last contact-related functionality into a functioning RequireJS module.

## Edit Alicen Arten

First name:

Last name:

Phone number:

Editing a contact

## Artificial Loading Delay

Data loading is delayed to demonstrate using a loading view.



The loading view



Git commit implementing contact editing:

[e1d6dfcda8b64af1c3e0b65a8e7a54b0b472efd6<sup>33</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/e1d6dfcda8b64af1c3e0b65a8e7a54b0b472efd6)

---

<sup>33</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/e1d6dfcda8b64af1c3e0b65a8e7a54b0b472efd6>

# Reimplementing the About Sub-Application

Now that we're done with the "contacts" sub-application, it's time to tackle the "about" sub-application.



Re-implement the entire "about" sub-application as a RequireJS module. Don't forget about navigation: if the user enters the "about" URL fragment directly into the address bar, he should be taken to the "about" sub-application. The solution follows.

## Exercise Solution

We'll start by creating a template for our only view:

---

assets/js/apps/about/show/templates/message.tpl

```
1 <h1>About this application</h1>
2 <p>This application was designed to accompany you during your learning.</p>
3 <p>Hopefully, it has served you well !</p>
```

---

We can then move on to view:

---

assets/js/apps/about/show/show\_view.js

```
1 define(["app", "tpl!apps/about/show/templates/message.tpl"],
2         function(ContactManager, messageTpl){
3     ContactManager.module("AboutApp.Show.View", function(View, ContactManager,
4                                         Backbone, Marionette, $, _){
5         View.Message = Marionette.ItemView.extend({
6             template: messageTpl
7         });
8     });
9
10    return ContactManager.AboutApp.Show.View;
11});
```

---

And continue with the controller:

assets/js/apps/about/show/show\_controller.js

---

```

1 define(["app", "apps/about/show/show_view"], function(ContactManager, View){
2     ContactManager.module("AboutApp.Show", function(Show, ContactManager,
3                                         Backbone, Marionette, $, _){
4         Show.Controller = {
5             showAbout: function(){
6                 var view = new View.Message();
7                 ContactManager.mainRegion.show(view);
8             }
9         };
10    });
11
12    return ContactManager.AboutApp.Show.Controller;
13 });

```

---

We've now worked our way up to the "about" sub-application's level, so let's adapt that file:

assets/js/apps/about/about\_app.js

---

```

1 define(["app"], function(ContactManager){
2     ContactManager.module("AboutApp", function(AboutApp, ContactManager,
3                                         Backbone, Marionette, $, _){
4         AboutApp.Router = Marionette.AppRouter.extend({
5             appRoutes: {
6                 "about" : "showAbout"
7             }
8         });
8
9
10        var API = {
11            showAbout: function(){
12                require(["apps/about/show/show_controller"], function(ShowController){
13                    ShowController.showAbout();
14                });
15            }
16        };
17
18        ContactManager.on("about:show", function(){
19            ContactManager.navigate("about");
20            API.showAbout();

```

```
21    });
22
23    ContactManager.addInitializer(function(){
24      new AboutApp.Router({
25        controller: API
26      });
27    });
28  });
29
30  return ContactManager.AboutApp;
31});
```

---

And finally, we need the main application to require the “about” sub-app before starting, so our sub-application is listening for route changes by the time we execute Backbone’s `history.start`. If you fail to do this, users won’t be able to route directly to the “about” sub-application by entering the URL fragment into the address bar, because no routing controller will be listening for the navigation event when it is triggered.

#### assets/js/app.js

```
1 ContactManager.on("initialize:after", function(){
2   if(Backbone.history){
3     require(["apps/contacts/contacts_app", "apps/about/about_app"], function () {
4       Backbone.history.start();
5
6       if(ContactManager.getCurrentRoute() === ""){
7         ContactManager.trigger("contacts:list");
8       }
9     });
10  }
11});
```

---

Great! Our “about” app is now working properly!

## About this application

This application was designed to accompany you during your learning.

Hopefully, it has served you well !

The “about” message



Git commit reimplementing the “about” sub-application:

[343f847f2af2b817af7bf86e1dc716613f18b3a4<sup>34</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/343f847f2af2b817af7bf86e1dc716613f18b3a4)

---

<sup>34</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/343f847f2af2b817af7bf86e1dc716613f18b3a4>

# Reimplementing the Header Sub-Application

The last step in converting our application to using RequireJS is to reimplement the header sub-application, so it manages the navigation menu for us. The header sub-app is slightly different in how it needs to be reimplemented, but I'll let you debug that on your own.



Reimplement the header sub-app. As mentioned, you'll need to use some critical thinking because it's not entirely comparable to a "standard" sub-app such as the "contacts" sub-app. Things you should bear in mind:

1. This sub-app will depend on an external library;
2. Event listeners need to be registered before that event is triggered, or nothing will happen (otherwise they wouldn't be listening soon enough);
3. Pay attention to the contents of *assets/js/apps/header/header\_app.js*.

As usual, the solution follows.

## Exercise Solution

We're going to need access to our header models before we can render them, so let's convert them into a RequireJS module. But the headers will depend on Backbone.Picky to mark the active header, so let's add it to *assets/js/require\_main.js* first:

`assets/js/require_main.js`

---

```
1 requirejs.config({
2   baseUrl: "assets/js",
3   paths: {
4     backbone: "vendor/backbone",
5     "backbone.picky": "vendor/backbone.picky",
6     "backbone.syphon": "vendor/backbone.syphon",
7     // edited for brevity
8   },
9 }
```

```

10  shim: {
11    underscore: {
12      exports: "_"
13    },
14    backbone: {
15      deps: ["jquery", "underscore", "json2"],
16      exports: "Backbone"
17    },
18    "backbone.picky": ["backbone"],
19    "backbone.syphon": ["backbone"],
20    // edited for brevity
21  }
22 });

```

---

Now we can convert our header to a RequireJS module, including its dependency:

`assets/js/entities/header.js`

```

1 define(["app", "backbone.picky"], function(ContactManager){
2   ContactManager.module("Entities", function(Entities, ContactManager,
3                                         Backbone, Marionette, $, _){
4     // edited for brevity
5   });
6
7   return ;
8 });

```

---

Once again, we have an empty return value, because we're going to access the headers via the request-response mechanism. In the same vein, we're adding the headers to the shared `Entities` module.

Of course, we're going to need to display these headers, so let's create templates for them:

Template for one header item (`assets/js/apps/header/list/templates/list_item.tpl`)

```

1 <a href="#<%= url %>"><%= name %></a>

```

---

Template to contain all headers (assets/js/apps/header/list/templates/list.tpl)

```
1 <div class="navbar-inner">
2   <div class="container">
3     <a class="brand" href="#contacts">Contact manager</a>
4     <div class="nav-collapse collapse">
5       <ul class="nav"></ul>
6     </div>
7   </div>
8 </div>
```

With the templates written, we can implement the view module:

assets/js/apps/header/list/list\_view.js

```
1 define(["app",
2         "tpl!apps/header/list/templates/list.tpl",
3         "tpl!apps/header/list/templates/list_item.tpl"],
4         function(ContactManager, listTpl, listItemTpl){
5   ContactManager.module("HeaderApp.List.View", function(View, ContactManager,
6                                         Backbone, Marionette, $, _){
7     View.Header = Marionette.ItemView.extend({
8       template: listItemTpl,
9       tagName: "li",
10      // edited for brevity
11    });
13
14   View.Headers = Marionette.CompositeView.extend({
15     template: listTpl,
16     className: "navbar navbar-inverse navbar-fixed-top",
17     itemView: View.Header,
18     itemViewContainer: "ul",
19
20     // edited for brevity
21   });
22 });
23
24   return ContactManager.HeaderApp.List.View;
25 });
```

Moving on to the controller, we can turn it into a RequireJS module:

assets/js/apps/header/list\_controller.js

---

```
1 define(["app", "apps/header/list/list_view"], function(ContactManager, View){
2   ContactManager.module("HeaderApp.List", function(List, ContactManager,
3                                         Backbone, Marionette, $, _){
4     List.Controller = {
5       listHeader: function(){
6         require(["entities/header"], function(){
7           var links = ContactManager.request("header:entities");
8           var headers = new View.Headers({collection: links});
9
10          // edited for brevity
11
12          ContactManager.headerRegion.show(headers);
13        });
14      },
15
16      setActiveHeader: function(headerUrl){
17        // edited for brevity
18      }
19    };
20  });
21
22  return ContactManager.HeaderApp.List.Controller;
23});
```

---



Don't forget to require the "header entity" module on line 6.

With our controller reimplemented, let's continue with the main "header" file:

assets/js/apps/header/header\_app.js

```
1 define(["app"], function(ContactManager){
2   ContactManager.module("HeaderApp", function(Header, ContactManager,
3                                             Backbone, Marionette, $, _){
4     var API = {
5       listHeader: function(){
6         require(["apps/header/list/list_controller"], function(ListController){
7           ListController.listHeader();
8         });
9       }
10    };
11
12   ContactManager.commands.setHandler("set:active:header", function(name){
13     ContactManager.HeaderApp.List.Controller.setActiveHeader(name);
14   });
15
16   Header.on("start", function(){
17     API.listHeader();
18   });
19 });
20
21   return ContactManager.HeaderApp;
22 });
```

Now that our “header” app is fully reimplemented, let’s tackle the tricky part. As you can see on lines 16-18 above, the sub-app registers a “start” listener with code to execute when this sub-module is started. Since we have *not* set this module’s `startWithParent` value to `false`, it will be started with the parent module. As it happens, the parent module is the main application: in other words, the “header” sub-app will be started as soon as the `ContactManager` main application is started.

If we want the code in our “start” listener to be executed, we’re going to have to make sure the listener gets registered before `ContactManager` is started, or by the time the event listener is configured and waiting, the event will already have fired. End result: nothing happens, and developers go crazy looking for bugs... How can we make sure the listener is active before our main application is started? Simply by requiring it before calling our app’s `start` method:

assets/js/apps/require\_main.js

---

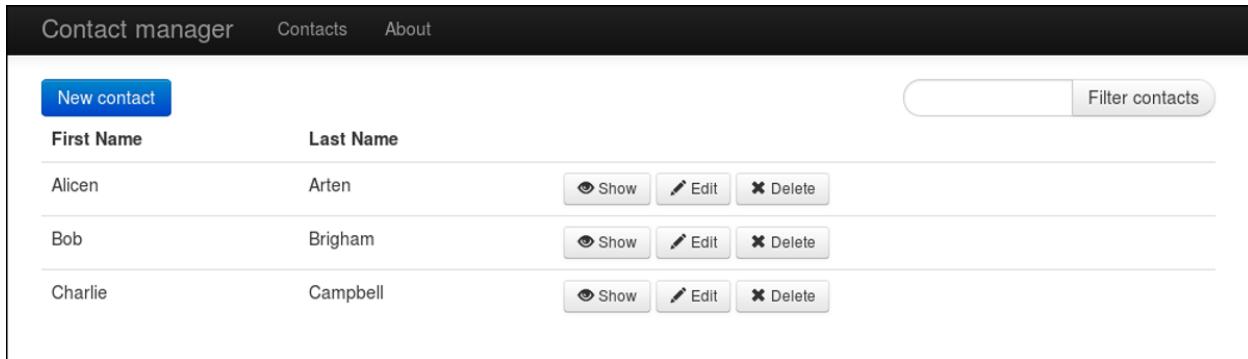
```

1 requirejs.config({
2   baseUrl: "assets/js",
3
4   // edited for brevity
5 });
6
7 require(["app", "apps/header/header_app"], function(ContactManager){
8   ContactManager.start();
9 });

```

---

As you can see on line 7, we're requiring the “header” sub-app before starting our main application. This way the “start” listener gets registered, and *then* we start the app, triggering the “start” event to which the “header” sub-app can respond. After all our hard work, our header menu displays in our application.



The navigation header

Unfortunately, our sub-application doesn't highlight the active header when a user enters a URL fragment directly into the address bar, although the active header is properly highlighted when the user clicks on a header entry. To fix this slight problem, we need to add code to each sub-app so it “orders” the header app to highlight their entry:

---

assets/js/apps/contacts/contacts\_app.js

```

1  var API = {
2      listContacts: function(criterion){
3          require(["apps/contacts/list/list_controller"], function(ListController){
4              ListController.listContacts(criterion);
5              ContactManager.execute("set:active:header", "contacts");
6          });
7      },
8
9      showContact: function(id){
10         require(["apps/contacts/show/show_controller"], function(ShowController){
11             ShowController.showContact(id);
12             ContactManager.execute("set:active:header", "contacts");
13         });
14     },
15
16     editContact: function(id){
17         require(["apps/contacts/edit/edit_controller"], function(EditController){
18             EditController.editContact(id);
19             ContactManager.execute("set:active:header", "contacts");
20         });
21     }
22 };

```

---

assets/js/apps/about/about\_app.js

```

1  var API = {
2      showAbout: function(){
3          require(["apps/about/show/show_controller"], function(ShowController){
4              ShowController.showAbout();
5              ContactManager.execute("set:active:header", "about");
6          });
7      }
8  };

```

---

If we give that a shot, it fails, because we don't have the "set:active:header" handler ready. Here is our current code:

---

assets/js/apps/header/header\_app.js

```

1 define(["app"], function(ContactManager){
2   ContactManager.module("HeaderApp", function(Header, ContactManager,
3                                             Backbone, Marionette, $, _){
4     var API = {
5       listHeader: function(){
6         require(["apps/header/list/list_controller"], function(ListController){
7           ListController.listHeader();
8         });
9       }
10    };
11
12   ContactManager.commands.setHandler("set:active:header", function(name){
13     ContactManager.HeaderApp.List.Controller.setActiveHeader(name);
14   });
15
16   Header.on("start", function(){
17     API.listHeader();
18   });
19 });
20
21   return ContactManager.HeaderApp;
22 });

```

---

As you can see, line 13 isn't within a require callback, so we can't be sure the List controller is loaded by the time we need it. Let's refactor this code to require the List controller as a module dependency:

assets/js/apps/header/header\_app.js

---

```

1 define(["app", "apps/header/list/list_controller"],
2        function(ContactManager, ListController){
3   ContactManager.module("HeaderApp", function(Header, ContactManager,
4                                             Backbone, Marionette, $, _){
5     var API = {
6       listHeader: function(){
7         ListController.listHeader();
8       }
9     };

```

```

10
11     ContactManager.commands.setHandler("set:active:header", function(name){
12         ListController.setActiveHeader(name);
13     });
14
15     Header.on("start", function(){
16         API.listHeader();
17     });
18 });
19
20     return ContactManager.HeaderApp;
21 });

```

---

We've required the `List` controller at the top level on lines 1 and 2, and we can access its reference on lines 7 and 12. With this modification in place, our header now properly highlights the active menu entry:

First Name	Last Name			
Alicen	Arten	Show	Edit	Delete
Bob	Brigham	Show	Edit	Delete
Charlie	Campbell	Show	Edit	Delete

The highlighted “Contacts” entry



Git commit reimplementing the “header” sub-application:

[58433228d70846a3af9153ca6796fa97c3d0e14<sup>35</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/58433228d70846a3af9153ca6796fa97c3d0e14<sup>35</sup>)

---

<sup>35</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/58433228d70846a3af9153ca6796fa97c3d0e14>

# Marionette Modules VS Plain RequireJS Modules

Marionette provides a module mechanism, as does RequireJS, and we've been using both simultaneously. Why? Because Marionette modules provide us with some functionality that RequireJS doesn't: the ability to start/stop a module (and its sub-modules), which we'll get around to implementing in the next chapter.

Since we haven't yet used the start/stop ability provided by Marionette modules (which we'll do in the [next chapter](#)), their use is debatable. In this chapter, we'll see what some modules would look like when using "plain" RequireJS modules, which is all you need in many projects. But once again, whether you use Marionette modules in addition to RequireJS modules depends on your particular circumstances (and, to some degree, personal taste).

## Converting the Contact Entity to Plain RequireJS Modules

We'll start by converting our contact entity, since we've currently go both the model *and* the collection defined in the same RequireJS module, which goes against recommended practice. But we need to start at the beginning: with our "local storage" adapter. Here's our current file:

assets/js/apps/config/storage.js

---

```
1 define(["app", "localStorage"], function(ContactManager){
2     ContactManager.module("Entities", function(Entities,
3                                         ContactManager, Backbone, Marionette, $, _){
4         var findStorageKey = function(entity){
5             // edited for brevity
6         };
7
8         var StorageMixin = function(entityPrototype){
9             // edited for brevity
10        };
11
12        Entities.configureStorage = function(entity){
13            _.extend(entity.prototype, new StorageMixin(entity.prototype));
14        };
15    });
16});
```

```

14      };
15  });
16
17  return ContactManager.Entities.configureStorage;
18 });

```

---

And here's what our file looks like as a plain RequireJS module:

assets/js/apps/config/storage.js

```

1 define(["localStorage"], function(){
2   var findStorageKey = function(entity){
3     // edited for brevity
4   };
5
6   var StorageMixin = function(entityPrototype){
7     // edited for brevity
8   };
9
10  return configureStorage = function(entity){
11    _.extend(entity.prototype, new StorageMixin(entity.prototype));
12  };
13 });

```

---

As you can see, we no longer need a reference to the `ContactManager` application since we don't reference it within the module. And once we've defined the desired behavior, we simply return (lines 10-12) the function we'll be using in other modules.



The return value of RequireJS objects doesn't necessarily have to be an object: any valid return value from a function is allowed.

With our local storage rewritten as a plain RequireJS module, we can move on to our contact entity. This is what we currently have:

assets/js/entities/contact.js

---

```
1 define(["app", "apps/config/storage/localstorage"], function(ContactManager){  
2     ContactManager.module("Entities", function(Entities,  
3                                         ContactManager, Backbone, Marionette, $, _){  
4         Entities.Contact = Backbone.Model.extend({  
5             // edited for brevity  
6         });  
7  
8         Entities.configureStorage(Entities.Contact);  
9  
10        Entities.ContactCollection = Backbone.Collection.extend({  
11            // edited for brevity  
12        });  
13  
14        Entities.configureStorage(Entities.ContactCollection);  
15  
16        var initializeContacts = function(){  
17            // edited for brevity  
18        };  
19  
20        var API = {  
21            getContactEntities: function(){  
22                // edited for brevity  
23            },  
24  
25            getContactEntity: function(contactId){  
26                // edited for brevity  
27            }  
28        };  
29  
30        ContactManager.reqres.setHandler("contact:entities", function(){  
31            return API.getContactEntities();  
32        });  
33  
34        ContactManager.reqres.setHandler("contact:entity", function(id){  
35            return API.getContactEntity(id);  
36        });  
37  
38        ContactManager.reqres.setHandler("contact:entity:new", function(id){  
39            return new Entities.Contact();  
40        });
```

```

41     });
42
43     return ;
44 });

```

---

As mentioned above, we need to break this into 2 separate files: one for the model, another for the collection. Here's our model file (note the different file path):

`assets/js/entities/contact/model.js`

```

1 define(["jquery", "backbone", "app", "apps/config/storage/localstorage"],
2         function($, Backbone, ContactManager, configureStorage){
3     var Contact = Backbone.Model.extend({
4         // edited for brevity
5     });
6
7     configureStorage(Contact);
8
9     var API = {
10         getContactEntity: function(contactId){
11             var contact = new Contact({id: contactId});
12             var defer = $.Deferred();
13             // edited for brevity
14             return defer.promise();
15         }
16     };
17
18     ContactManager.reqres.setHandler("contact:entity", function(id){
19         return API.getContactEntity(id);
20     });
21
22     ContactManager.reqres.setHandler("contact:entity:new", function(id){
23         return new Contact();
24     });
25
26     return Contact;
27 });

```

---

In this file, we define only the Contact model and its related handlers. As you can see on lines 1 and 2, we need to define all the dependencies we'll be referencing. This wasn't the case earlier, because Marionette modules provide references to many dependencies (jQuery, Backbone, etc.) within the

callback definition. You can also see we’re returning the Contact object, although we’re not going to use a direct reference to it in any other module: we’ll access the Contact model instances via the request-response mechanism. But since we’ve got a single definite object to use as a return value, we might as well use it.

## Using a Separate Module as the Event Aggregator

In the code above, we’re adding ContactManager as a dependency so that we can then use its reqres attribute. We can afford to include the entire main application as a dependency because it is quite small. If you have a larger main application file (with many dependencies), you’ll probably want to define a dedicated module to provide the request-response event aggregator (as indicated [here<sup>36</sup>](#)).

RequireJS module providing an event aggregator (assets/js/vent.js)

---

```
1 define(["Backbone", "Marionette"], function(Backbone) {
2   return new Backbone.Wreqr.EventAggregator();
3 });
```

---

If you compare this definition with the linked page in Marionette’s wiki, you’ll notice the syntax is slightly different: the wiki is using the [advanced technique<sup>37</sup>](#) for including all Marionette components as individual AMD modules. We’re simply using the “everything included” build, so we’re declaring Backbone as a dependency in order to reference it on line 2, and we have Marionette as a dependency so that it gets loaded and registers the Wreqr attribute we need to use.

Then, we can use our event aggregator module like so:

Using the event aggregator in another module’s definition

---

```
1 define(["assets/js/vent"], function(vent) {
2   vent.on("eventName", function() {});
3   vent.trigger("eventName");
4 });
```

---

## Converting the Collection

With our model rewritten, let’s move on to rewriting our Contact collection. Here’s our current code:

<sup>36</sup><https://github.com/marionettejs/backbone.marionette/wiki/Using-marionette-with-requirejs#example-with-central-vent>

<sup>37</sup><https://github.com/marionettejs/backbone.marionette/wiki/Using-marionette-with-requirejs#advanced-usage>

```
assets/js/entities/contact.js
define(["app", "apps/config/storage/localstorage"], function(ContactManager){
  ContactManager.module("Entities", function(Entities,
    ContactManager, Backbone, Marionette, $, _){
    // edited for brevity

    Entities.ContactCollection = Backbone.Collection.extend({
      url: "contacts",
      model: Entities.Contact,
      comparator: "firstName"
    });

    Entities.configureStorage(Entities.ContactCollection);

    var initializeContacts = function(){
      var contacts = new Entities.ContactCollection([
        { id: 1, firstName: "Alice", lastName: "Arten",
          phoneNumber: "555-0184" },
        { id: 2, firstName: "Bob", lastName: "Brigham",
          phoneNumber: "555-0163" },
        { id: 3, firstName: "Charlie", lastName: "Campbell",
          phoneNumber: "555-0129" }
      ]);
      contacts.forEach(function(contact){
        contact.save();
      });
      return contacts.models;
    };

    var API = {
      getContactEntities: function(){
        var contacts = new Entities.ContactCollection();
        var defer = $.Deferred();
        contacts.fetch({
          success: function(data){
            defer.resolve(data);
          }
        });
        var promise = defer.promise();
        $.when(promise).done(function(contacts){
          if(contacts.length === 0){
```

```

41         // if we don't have any contacts yet, create some for convenience
42         var models = initializeContacts();
43         contacts.reset(models);
44     }
45 });
46 return promise;
47 },
48
49 getContactEntity: function(contactId){
50     // edited for brevity
51 }
52 };
53
54 ContactManager.reqres.setHandler("contact:entities", function(){
55     return API.getContactEntities();
56 });
57
58 // edited for brevity
59 });
60
61 return ;
62 });

```

---

Once again, we'll need to store this in a different file. Here's our new module:

#### assets/js/entities/contact/collection.js

```

1 define(["jquery", "backbone", "app",
2         "entities/contact/model", "apps/config/storage/localstorage"],
3         function($, Backbone, ContactManager, ContactModel, configureStorage){
4     var ContactCollection = Backbone.Collection.extend({
5         url: "contacts",
6         model: ContactModel,
7         comparator: "firstName"
8     });
9
10    configureStorage(ContactCollection);
11
12    var initializeContacts = function(){
13        var contacts = new ContactCollection([
14            { id: 1, firstName: "Alice", lastName: "Arten",

```

```

15      phoneNumber: "555-0184" },
16      { id: 2, firstName: "Bob", lastName: "Brigham",
17        phoneNumber: "555-0163" },
18      { id: 3, firstName: "Charlie", lastName: "Campbell",
19        phoneNumber: "555-0129" }
20    ]);
21    contacts.forEach(function(contact){
22      contact.save();
23    });
24    return contacts.models;
25  };
26
27  var API = {
28    getContactEntities: function(){
29      var contacts = new ContactCollection();
30      var defer = $.Deferred();
31      contacts.fetch({
32        success: function(data){
33          defer.resolve(data);
34        }
35      });
36      var promise = defer.promise();
37      $.when(promise).done(function(contacts){
38        if(contacts.length === 0){
39          // if we don't have any contacts yet, create some for convenience
40          var models = initializeContacts();
41          contacts.reset(models);
42        }
43      });
44      return promise;
45    }
46  };
47
48  ContactManager.reqres.setHandler("contact:entities", function(){
49    return API.getContactEntities();
50  });
51
52  return ContactCollection;
53 });

```

---



Since we're using plain RequireJS modules, we need to declare our Contact model as a dependency, so that we can reference it on line 6.

## Rewriting the Show View

We also need to rewrite the view module, so that it looks like this:

assets/js/apps/contacts/show/show\_view.js

---

```

1 define(["marionette", "tpl!apps/about/show/templates/message.tpl"], function(Mari\
2 onette, messageTpl){
3     return {
4         Message: Marionette.ItemView.extend({
5             template: messageTpl
6         })
7     };
8 });

```

---

## Rewriting the Show Action

Now that we've got shiny new modules for our contact model/collection, let's rewrite our "show" action (as an example) to use the plain RequireJS module defining the Contact model:

assets/js/apps/contacts/show/show\_controller.js

---

```

1 define(["app", "apps/contacts/show/show_view"], function(ContactManager, View){
2     return {
3         showContact: function(id){
4             require(["common/views", "entities/contact/model"], function(CommonViews){
5                 // edited for brevity
6
7                 var fetchingContact = ContactManager.request("contact:entity", id);
8                 $.when(fetchingContact).done(function(contact){
9                     // edited for brevity
10
11                     ContactManager.mainRegion.show(contactView);
12                 });
13             });
14         }
15     });

```

---

As you can see, on line 4 we're requiring the contact *model* so we can request a contact instance on line 7. Once again, we're not using a direct reference to the Contact model as we request it via the request-response mechanism. By requiring the contact entity, we're guaranteeing the "contact:entity" handler will have been registered and will return the data we need. Also of importance is the fact that we're no longer defining a named Controller object: we're directly returning the object as the RequireJS module's return value.

# Start/Stop Modules According to Routes

In this chapter, we'll get our modules to start and stop according to routes. If you've been paying attention in the previous chapter, you'll remember that the main reason we're using *Marionette* modules instead of plain RequireJS modules, is to be able to start/stop them (and their sub-modules) easily.

Before we get into the thick of things, let's take a moment to discuss *why* we'd want to start/stop modules according to routes. This is not an approach you should apply to all projects indiscriminately, as in many cases it will introduce extra complexity overhead with little benefit. So when would your app benefit from start/stop functionality? There are 2 main types of cases:

- when your sub-application requires a library that isn't used anywhere else (e.g. a graphing library);
- when your app needs to load a lot of data on initialization, and needs to free the memory when another sub-application loads (e.g. a dashboard with drill-down functionality).

In these cases, we'll be able to load the libraries and/or data when the sub-app starts, and free the memory as the user navigates away from the sub-application (e.g. he's no longer viewing the dashboard requiring advanced visualization and lots of data).

## Implementation Strategy

In our current implementation, our modules start automatically with their parent module, and this happens all the way to the main application. In other words, all of the modules we've defined start as soon as the main application is running. We need this to be modified slightly to add module start/stop: obviously, we don't want the sub-apps to start automatically, but only when we tell them to. But implementing this behavior isn't as simple as telling the modules not to start automatically: we need the *routers* for each sub-app to start automatically with the main app, so that they can react to routing events (and start the appropriate module, if necessary).

What we need to solve this challenge is to have a common "router" module that will contain the routing code for all of our sub-apps (and will start automatically); and then we can set `startWithParent` to `false` in the sub-app's main module (e.g. `ContactsApp`). Here's what that looks like:

assets/js/apps/contact/contacts\_app.js

```
1 define(["app"], function(ContactManager){  
2     ContactManager.module("ContactsApp", function(ContactManager,  
3                                     Backbone, Marionette, $, _){  
4         ContactsApp.startWithParent = false;  
5  
6         ContactsApp.onStart = function(){  
7             console.log("starting ContactsApp");  
8         };  
9  
10        ContactsApp.onStop = function(){  
11            console.log("stopping ContactsApp");  
12        };  
13    });  
14  
15    ContactManager.module("Routers.ContactsApp", function(ContactsAppRouter,  
16                                         ContactManager, Backbone, Marionette, $, _){  
17        ContactsAppRouter.Router = Marionette.AppRouter.extend({  
18            appRoutes: {  
19                // edited for brevity  
20            }  
21        });  
22  
23        var API = {  
24            // edited for brevity  
25        };  
26  
27        // navigation event listeners (edited for brevity)  
28  
29        ContactManager.addInitializer(function(){  
30            new ContactsAppRouter.Router({  
31                controller: API  
32            });  
33        });  
34    });  
35  
36    return ContactManager.ContactsAppRouter;  
37});
```

Let's take a look at our changes:

- on lines 2-13, we define a new Marionette module called “ContactsApp”. Within it, we
  - declare (on line 4) that it shouldn’t start with its parent app (i.e. the main application);
  - add an `onStart` function (lines 6-8) which will be executed each time the application starts. This is where you would load extra libraries or data required by the sub-app;
  - add an `onStop` function (lines 10-12) to free memory by dereferencing variables (e.g. using `delete38` or by assigning them to `null`) that are no longer of use when the user navigates away from the sub-app.
- on line 15, we’ve renamed our module from `ContactsApp` to `Routers.ContactsApp`. This is because we’re going to use the `Routers` Marionette module to hold all of our routing controllers, and it will start (automatically) with the main application. The rest of the code hasn’t been modified (except for renaming `ContactsApp` to `ContactsAppRouter`, such as on line 30).



What about the sub-modules such as `List`? We’re leaving those sub-modules as they are: we need their `startWithParent` property to be `true` (which is the default value). This way, we prevent our top-level `ContactsApp` from starting automatically when `ContactManager` is started, but when we manually start `ContactsApp` its sub-modules (`List`, etc.) will also start (because they are configured to `startWithParent`).

With this code change, we’ve defined the top-level `ContactsApp` sub-application module within `assets/js/apps/contact/contacts_app.js`, which is the first file to get required for our contacts sub-app. Therefore, `ContactsApp`’s `startWithParent` is set to `false` immediately and won’t start unless we explicitly start it ourselves. We can then define the Marionette module with the routing controller code, and use it as the return value for our `RequireJS` module.

But we’re still missing a major piece of the puzzle: we need code to start the sub-app we need, and stop all other sub-apps. Let’s imagine we have 2 very large Marionette sub-applications that expose dashboards. One of them deals with customer data (where they live, how often they shop, their profile, etc.), and the other displays aggregate financial information (e.g. sales over the last 2 weeks by customer segment, number of visits, stock turnover, etc.). Both of these sub-applications provide advanced analytics and multiple ways to display and regroup data.

To enhance performance, the most frequently analyzed data is loaded when the sub-app is initialized. This way, when a user wants to change the analysis he’s looking at (e.g. from “overall sales” to “single male sales”), the sub-app already has the data available and can display the new analysis rapidly. Unfortunately, all this data takes up memory, and when the user moves on to a different sub-app, this data isn’t necessary anymore: the memory it’s using should be freed. This can be accomplished by stopping the current module, and dereferencing the loaded data within the Marionette module’s `onStop` method.

With all this module starting and stopping happening, it becomes necessary to have a function we can call to ensure the sub-app we want to use is active, and stop all the others. Here’s what it looks like:

---

<sup>38</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete>

assets/js/app.js

```
1 ContactManager.startSubApp = function(appName, args){
2     var currentApp = ContactManager.module(appName);
3     if (ContactManager.currentApp === currentApp){ return; }
4
5     if (ContactManager.currentApp){
6         ContactManager.currentApp.stop();
7     }
8
9     ContactManager.currentApp = currentApp;
10    currentApp.start(args);
11 };
12
13 ContactManager.on("initialize:after", function(){
14     // edited for brevity
15});
```

And now, of course, we need to execute this method before any controller action:

assets/js/apps/contact/contacts\_app.js

```
1 var API = {
2     listContacts: function(criterion){
3         require(["apps/contacts/list/list_controller"], function(ListController){
4             ContactManager.startSubApp("ContactsApp");
5             ListController.listContacts(criterion);
6             ContactManager.execute("set:active:header", "contacts");
7         });
8     },
9
10    showContact: function(id){
11        require(["apps/contacts/show/show_controller"], function(ShowController){
12            ContactManager.startSubApp("ContactsApp");
13            ShowController.showContact(id);
14            ContactManager.execute("set:active:header", "contacts");
15        });
16    },
17
18    editContact: function(id){
```

```

19   require(["apps/contacts/edit/edit_controller"], function(EditController){
20     ContactManager.startSubApp("ContactsApp");
21     EditController.editContact(id);
22     ContactManager.execute("set:active:header", "contacts");
23   });
24 }
25 };

```

---

As you can see above, there's a fair bit of duplicated code. Let's clean that up with a new "private" function:

`assets/js/apps/contact/contacts_app.js`

```

1 var executeAction = function(action, arg){
2   ContactManager.startSubApp("ContactsApp");
3   action(arg);
4   ContactManager.execute("set:active:header", "contacts");
5 };
6
7 var API = {
8   listContacts: function(criterion){
9     require(["apps/contacts/list/list_controller"], function(ListController){
10       executeAction(ListController.listContacts, criterion);
11     });
12   },
13
14   showContact: function(id){
15     require(["apps/contacts/show/show_controller"], function(ShowController){
16       executeAction(ShowController.showContact, id);
17     });
18   },
19
20   editContact: function(id){
21     require(["apps/contacts/edit/edit_controller"], function(EditController){
22       executeAction(EditController.editContact, id);
23     });
24   }
25 };

```

---



Modify the "AboutApp" to start its own module, but only when explicitly ordered to. You'll find the solution below.

## Modifying the AboutApp

Here's our modified file for the "about" sub-app:

assets/js/apps/about/about\_app.js

```
1 define(["app"], function(ContactManager){  
2     ContactManager.module("AboutApp", function(AboutApp, ContactManager,  
3                                         Backbone, Marionette, $, _){  
4         AboutApp.startWithParent = false;  
5  
6         AboutApp.onStart = function(){  
7             console.log("starting AboutApp");  
8         };  
9  
10        AboutApp.onStop = function(){  
11            console.log("stopping AboutApp");  
12        };  
13    });  
14  
15    ContactManager.module("Routers.AboutApp", function(AboutAppRouter,  
16                                         ContactManager, Backbone, Marionette, $, _){  
17        AboutAppRouter.Router = Marionette.AppRouter.extend({  
18            appRoutes: {  
19                "about" : "showAbout"  
20            }  
21        });  
22  
23        var API = {  
24            showAbout: function(){  
25                require(["apps/about/show/show_controller"], function(ShowController){  
26                    ContactManager.startSubApp("AboutApp");  
27                    ShowController.showAbout();  
28                    ContactManager.execute("set:active:header", "about");  
29                });  
30            }  
31        };  
32  
33        ContactManager.on("about:show", function(){  
34            ContactManager.navigate("about");  
35            API.showAbout();  
36        });  
37    });  
38});
```

```
37
38     ContactManager.addInitializer(function(){
39         new AboutAppRouter.Router({
40             controller: API
41         });
42     });
43 );
44
45     return ContactManager.AboutAppRouter;
46 });
```

---

With these changes in place, you can see the various messages being printed to the console as you navigate around (e.g. by clicking on the links in the header menu). In addition, if you manually enter URLs in the address bar, the modules will also be started/stopped properly (and display the console messages). Success!



Git commit implementing module start/stop according to routes:

[3bf9ff22f4bb7645c074bc636783dc7aae1bd23a<sup>39</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/3bf9ff22f4bb7645c074bc636783dc7aae1bd23a)

---

<sup>39</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/3bf9ff22f4bb7645c074bc636783dc7aae1bd23a>

# Mixing and Matching Modules

We've talked about why you might want to use Marionette modules in a RequireJS project [earlier](#), and we also saw how you could write a [plain RequireJS module](#) in your project. Naturally, in some cases you might want to use both types of modules: plain RequireJS modules for sub-apps that don't need to (un)load libraries and resources, and Marionette modules for those that do. To cover this use case, we'll convert our "about" sup-app to use only plain RequireJS modules.

## Converting the About Sub-Application to Plain RequireJS Modules

assets/js/apps/about/about\_view.js

```
1 define(["marionette", "tpl!apps/about/show/templates/message.tpl"],  
2                     function(Marionette, messageTpl){  
3     return {  
4         Message: Marionette.ItemView.extend({  
5             template: messageTpl  
6         })  
7     };  
8 });
```

assets/js/apps/about/show\_controller.js

```
1 define(["app", "apps/about/show/show_view"], function(ContactManager, View){  
2     return {  
3         showAbout: function(){  
4             var view = new View.Message();  
5             ContactManager.mainRegion.show(view);  
6         }  
7     };  
8 });
```

assets/js/apps/about/about\_app.js

```
1 define(["app", "marionette"], function(ContactManager, Marionette){
2     var Router = Marionette.AppRouter.extend({
3         appRoutes: {
4             "about" : "showAbout"
5         }
6     });
7
8     var API = {
9         showAbout: function(){
10            require(["apps/about/show/show_controller"], function(ShowController){
11                ContactManager.startSubApp("AboutApp");
12                ShowController.showAbout();
13                ContactManager.execute("set:active:header", "about");
14            });
15        }
16    };
17
18    ContactManager.on("about:show", function(){
19        ContactManager.navigate("about");
20        API.showAbout();
21    });
22
23    ContactManager.addInitializer(function(){
24        new Router({
25            controller: API
26        });
27    });
28
29    return Router;
30});
```

If you use the application now, you'll see it works. However, our start/stop code has a small issue we'll need to fix. Here's our current code:

---

assets/js/app.js

```

1 ContactManager.startSubApp = function(appName, args){
2     var currentApp = ContactManager.module(appName);
3     if (ContactManager.currentApp === currentApp){ return; }
4
5     if (ContactManager.currentApp){
6         ContactManager.currentApp.stop();
7     }
8
9     ContactManager.currentApp = currentApp;
10    currentApp.start(args);
11 };

```

---

Line 2 is causing trouble: the call to `module` returns the Marionette module *if it exists*, otherwise it returns a *new* Marionette module. So in our case, since the `AboutApp` Marionette module doesn't exist, it's going to be instantiated. This isn't the behavior we want, so let's fix it.

## Managing Start/Stop with Mixed Module Types

The first thing we'll do is pass `null` to indicate the current sub-app isn't a Marionette module, and therefore doesn't need to be started or stopped.

assets/js/apps/about/about\_app.js

```

1 var API = {
2     showAbout: function(){
3         require(["apps/about/show/show_controller"], function(ShowController){
4             ContactManager.startSubApp(null);
5             ShowController.showAbout();
6             ContactManager.execute("set:active:header", "about");
7         });
8     }
9 };

```

---

As you can see, we've modified line 4 to pass the `null` value, since our "about" app is no longer a Marionette module. Next, we need to modify our module starting code to properly manage both module types.

assets/js/app.js

---

```
1 ContactManager.startSubApp = function(appName, args){
2   var currentApp = appName ? ContactManager.module(appName) : null;
3   if (ContactManager.currentApp === currentApp){ return; }
4
5   if (ContactManager.currentApp){
6     ContactManager.currentApp.stop();
7   }
8
9   ContactManager.currentApp = currentApp;
10  if(currentApp){
11    currentApp.start(args);
12  }
13 };
```

---

Our change is pretty straightforward:

- we set `currentApp` to the module (if it exists), or we use `null` (line 2);
- if the current module is a Marionette module, we stop it (lines 5-7);
- if the new module is a Marionette module, we start it (lines 10-12).

With this modification in place, our app happily runs both module types!



Git commit mixing module types:

[ef42ce031913810542566f928d84e26e2f7850c5<sup>40</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/ef42ce031913810542566f928d84e26e2f7850c5)

---

<sup>40</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/ef42ce031913810542566f928d84e26e2f7850c5>

# Optimizing the Application with r.js

So far, we've rewritten our app to use a mix of "plain" RequireJS and Marionette modules, all handled by RequireJS. As you'll recall, RequireJS automatically handles the dependencies and loads the javascript files in the proper order. However, this order doesn't change that often (certainly not on each page request), so why are we having this work done over and over? Wouldn't it be smarter to just compute the dependencies once, store the resulting code in a single file and use that in our application? Of course it is, and that's one of the goals of `r.js`, the optimizer provided by RequireJS.

Before we can run the optimizer on our application, we'll need to install node.js (download [here<sup>41</sup>](#)). Then, download the `r.js` file from [here<sup>42</sup>](#) and store it in your application's root path.<sup>43</sup>

With that out of the way, we're almost ready to run the optimizer. We'll need a `build file44`:

`assets/js/build.js`

---

```
1 ({  
2     baseUrl: ".",
3     name: "require_main",
4     mainConfigFile: "require_main.js",
5     out: "require_main.built.js"
6 })
```

---

Here's what each attribute does (find out more [here<sup>45</sup>](#)):

- `baseUrl` defines the url from which all other paths are determined, and it defaults to the build file's path, in our case `assets/js` (since that's where we've stored our build file). This would mean that our libraries would be searched for within `assets/js/assets/js`, because the build file would have a default base path of `assets/js`, and our main configuration file defines *its own* `baseUrl` attribute as `assets/js`. Put both of those together, and we have the optimizer looking for libraries in `assets/js/assets/js`, which isn't what we want... Instead, we need to provide the actual path we want to use, relative to the application's base directory. In our case we want it to be that same base directory, hence the ":" path;

---

<sup>41</sup><http://nodejs.org/download/>

<sup>42</sup><http://requirejs.org/docs/download.html#rjs>

<sup>43</sup>Note that, per the [instructions](#), you can also use the optimizer after installing it with npm.

<sup>44</sup><http://requirejs.org/docs/optimization.html#wholeproject>

<sup>45</sup><https://github.com/jrburke/r.js/blob/master/build/example.build.js>

- since we're only optimizing one module (the main module, which will recursively optimize all its dependencies), we can provide the module's name inline with the `name` attribute. Since the `assets/js/require_main.js` file defines our main module, we simply provide its name here, and RequireJS figures out where it is located (via the `baseUrl` properties);
- `mainConfigFile` allows us to tell the optimizer where we have our configuration options, so we don't need to repeat them on the command line. Once again, the file's location is determined via the `baseUrl` properties;
- `out` is the name of the built file we want the optimizer to generate. This way, we'll have a single, optimized, and uglified file for inclusion in the deployed application. Here too, it's location is determined by the `baseUrl` properties.

We're now ready to run the optimizer. From our application's root directory, use a command console to run

```
node r.js -o assets/js/build.js
```

You'll notice that we've indicated the location of our build file with the `-o` option. Once the optimizer is done running, you'll have a brand new file: `assets/js/require_main.built.js` containing our entire app in a single minified file.

Let's try our new file by using it in our `index.html` file:

### index.html

---

```
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <title>Marionette Contact Manager</title>
6          <link href=".//assets/css/bootstrap.css" rel="stylesheet">
7          <link href=".//assets/css/application.css" rel="stylesheet">
8          <link href=".//assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
9      </head>
10
11     <body>
12         <div id="header-region"></div>
13
14         <div id="main-region" class="container">
15             <p>Here is static content in the web page. You'll notice that
16                 it gets replaced by our app as soon as we start it.</p>
17         </div>
18
```

```

19   <div id="dialog-region"></div>
20
21   <script src=".//assets/js/require_main.built.js"></script>
22 </body>
23 </html>

```

---

As you can see, we've replaced our original RequireJS include on line 21 with our newly optimized file. But when we refresh our index page, our app doesn't load properly. Instead we get an error message indicating that "define is not defined". What does that mean? Well, even though we don't need RequireJS to process the dependencies, we still need it to load our module. So we'll need to include the RequireJS library before we load our app file:

### index.html

```

1 <script src=".//assets/js/vendor/require.js"></script>
2 <script src=".//assets/js/require_main.built.js"></script>

```

---



Git commit using the single optimized file:

a736595a0fec466aee049ae91429c086b2b6049<sup>46</sup>

Our app now loads properly, but it seems a little wasteful doesn't it? We're loading RequireJS with all its functionality (and file size!) only to load our unique module. There must be a better way!

## Loading with Almond.js

Almond.js is a minimal AMD module loader also written by James Burke (the man behind RequireJS). It's meant to be lightweight in exchange of providing a very limited set of features, which is exactly what we're looking for! Let's download it from [here<sup>47</sup>](#) and save it as *assets/js/vendor/almond.js*.



Using Almond does come with some [restrictions<sup>48</sup>](#), among which the inability to use dynamic code loading (e.g. attempting to use a CDN-hosted library and falling back to a local version).

Then, we need to change our build file to include Almond in the optimized file:

<sup>46</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/a736595a0fec466aee049ae91429c086b2b6049>

<sup>47</sup><https://raw.github.com/jrburke/almond/latest/almond.js>

<sup>48</sup><https://github.com/jrburke/almond#restrictions>

## assets/js/build.js

---

```

1 ({ 
2   baseUrl: ".",
3   name: "vendor/almond",
4   include: "require_main",
5   mainConfigFile: "require_main.js",
6   out: "require_main.built.js"
7 })

```

---

Notice the changes on lines 3 and 4: we're telling the optimizer to build Almond and *include* our main application. After recompiling our app, we adapt our index page:

## index.html

---

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Marionette Contact Manager</title>
6     <link href=".//assets/css/bootstrap.css" rel="stylesheet">
7     <link href=".//assets/css/application.css" rel="stylesheet">
8     <link href=".//assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
9   </head>
10
11  <body>
12    <div id="header-region"></div>
13
14    <div id="main-region" class="container">
15      <p>Here is static content in the web page. You'll notice that
16          it gets replaced by our app as soon as we start it.</p>
17    </div>
18
19    <div id="dialog-region"></div>
20
21    <script src=".//assets/js/require_main.built.js"></script>
22  </body>
23 </html>

```

---



Notice that on line 21, we include only the compiled file: RequireJS and Almond do *not* get included.

We can now reload our app, aaaaaaaaaaaaand it doesn't work! Among others, we get the "Error: undefined missing entities/header" error. How is this possible, when it clearly worked fine with RequireJS? It's because we have nested require calls (i.e. declaring dependencies *within* the module, instead of only in its definition). Luckily the fix is to simply add a `findNestedDependencies` attribute to our build file:

#### assets/js/build.js

```
1  ({  
2    baseUrl: ".",
3    name: "vendor/almond",
4    include: "require_main",
5    mainConfigFile: "require_main.js",
6    out: "require_main.built.js",
7    findNestedDependencies: true
8  })
```

We compile the app again with the same command

```
node r.js -o assets/js/build.js
```

to generate a new version of our compiled application. This time when we reload our application, it works fine!



Git commit using almond to load the optimized application:

[85d3d3dd40d0cebd858224c3903a12d6998d668d<sup>49</sup>](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/85d3d3dd40d0cebd858224c3903a12d6998d668d)

---

<sup>49</sup><https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/85d3d3dd40d0cebd858224c3903a12d6998d668d>

# Closing Thoughts

We've now rewritten the application with RequireJS and introduced a few new ideas along the way. You should now have a good grasp of how RequireJS works, and you should be able to apply what you've learned to your own projects.

If you've enjoyed the book, it would be immensely helpful if you could take a few minutes to write your opinion on the book's [review page<sup>50</sup>](#). Help others determine if the book is right for them!

Would you like me to cover another subject in an upcoming book? Let me know by email at [davidsulc@gmail.com](mailto:davidsulc@gmail.com) or on Twitter (@davidsulc). In the meantime, see the next chapter for [my current list of books](#).

Thanks for reading this book, it's been great having you along!

## Keeping in Touch

I plan to release more books in the future, where each will teach a new skill (like in this book) or help you improve an existing skill and take it to the next level.

If you'd like to be notified when I release a new book, receive discounts, etc., sign up to my mailing list at [You can also follow me on Twitter: @davidsulc](http://davidsulc.com/mailing_list<sup>51</sup></a>. No spam, I promise!</p></div><div data-bbox=)

---

<sup>50</sup><https://leanpub.com/structuring-backbone-with-requirejs-and-marionette/feedback>

<sup>51</sup>[http://davidsulc.com/mailing\\_list](http://davidsulc.com/mailing_list)

# Other Books I've Written

Presumably, you're already comfortable with how Marionette works. If that isn't quite the case, take a look the book where the Contact Manager application is originally developed: [Backbone.Marionette.js: A Gentle Introduction<sup>52</sup>](#).

---

<sup>52</sup><https://leanpub.com/marionette-gentle-introduction>