

Structuring



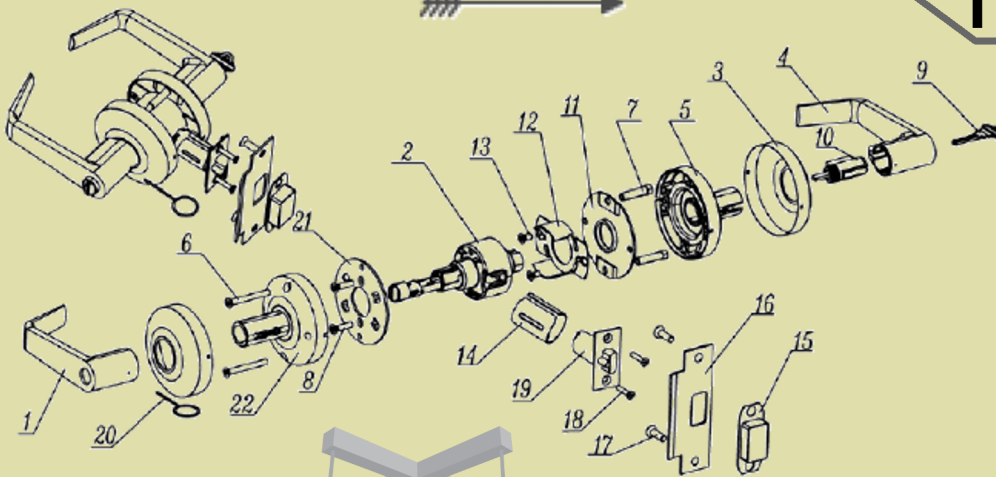
BACKBONE.JS

Code

with



RequireJS



and

Marionette^{js}

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 2014-05-18



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 - 2014 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=#>

Also By **David Sulc**

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

[Backbone.Marionette.js: A Serious Progression](#)

Contents

Who This Book is For	i
Following Along with Git	ii
Jumping in for Advanced Readers	iii
Why Use AMD?	1
Getting Started	2
Adding RequireJS to the Mix	3
Using Shims	6
Loading the Main App	10
Discovering Modules	10
Turning ContactManager into a RequireJS Module	11
Including the Dialog Region as a Dependency	13
Listing Contacts	18
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
Editing Contacts with a Modal	40
Expliciting Dependencies	48
Creating Contacts with a Modal	49
Exercise Solution	49
Further Decoupling of the Contact Module	51
Filtering Contacts	53
Exercise Solution	53
Adding the Loading View	58
Exercise Solution	58
Showing Contacts	62

CONTENTS

Exercise Solution	62
Editing Contacts	67
Exercise Solution	67
Reimplementing the About Sub-Application	71
Exercise Solution	71
Reimplementing the Header Sub-Application	74
Exercise Solution	74
Marionette Modules VS Plain RequireJS Modules	82
Converting the Contact Entity to Plain RequireJS Modules	82
Using a Separate Module as the Event Aggregator	85
Converting the Collection	86
Rewriting the Show View	89
Rewriting the Show Action	90
Start/Stop Modules According to Routes	92
Implementation Strategy	92
Modifying the AboutApp	96
Mixing and Matching Modules	99
Converting the About Sub-Application to Plain RequireJS Modules	99
Managing Start/Stop with Mixed Module Types	101
Optimizing the Application with r.js	103
Loading with Almond.js	105
Closing Thoughts	108
Keeping in Touch	108
Other Books I've Written	109

Who This Book is For

This book is first and foremost for the readers of “[Backbone.Marionette.js: A Gentle Introduction](https://leanpub.com/marionette-gentle-introduction)¹” 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).

¹<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>².

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:

[0531c10e9ac3c0a2ffeb2a3eaf354f621d261e9](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/0531c10e9ac3c0a2ffeb2a3eaf354f621d261e9)³

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](#)⁴, 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](#)⁵. 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](#)⁶
- 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

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

³<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/0531c10e9ac3c0a2ffeb2a3eaf354f621d261e9>

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

⁵<http://git-scm.com/book>

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


```
git show 0531c10e9ac3c0a2ffeb2a3eaaf354f621d261e9
```

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 0531c10e9ac3c0a2ffeb2a3eaaf354f621d261e9
```

- 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](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/archive/master.zip)⁷. 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.

⁷<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](https://leanpub.com/marionette-gentle-introduction)"⁸ book) from [here](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/archive/04a62e80d62df430fcc84d12ed46b3842baaf6ff.zip)⁹. 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.

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

⁹<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/archive/04a62e80d62df430fcc84d12ed46b3842baaf6ff.zip>



Git commit with our basic index.html:

[0531c10e9ac3c0a2ffeb2a3eaaf354f621d261e9](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/0531c10e9ac3c0a2ffeb2a3eaaf354f621d261e9)¹⁰

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](http://requirejs.org/docs/release/2.1.8/comments/require.js)¹¹ 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, 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

¹⁰<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/0531c10e9ac3c0a2ffeb2a3eaaf354f621d261e9>

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

```
<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 *baseUrl*, 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 *baseUrl* 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 baseUrl path*. As we indicated jQuery was located at *vendor/jquery*, and our *baseUrl* 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`¹² 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](#)¹³:

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](#)¹⁴, you can easily load libraries from CDNs and default to using a local version should the CDN be unavailable.

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

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

¹⁴<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:

[bed1aad25ed8f2d19921804c4f68bd40219aa4f9](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/bed1aad25ed8f2d19921804c4f68bd40219aa4f9)¹⁵

¹⁵<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/bed1aad25ed8f2d19921804c4f68bd40219aa4f9>

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](#)¹⁶.

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:

¹⁶<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](http://requirejs.org/docs/api.html#jsfiles)¹⁷, 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects)¹⁸). 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](http://requirejs.org/docs/api.html#config-shim)¹⁹ (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.

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

¹⁸https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

¹⁹<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 2.²⁰

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

²⁰Note 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:

[bdf30d6ed9d53f7b917d302c0eac0a04ea842830](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/bdf30d6ed9d53f7b917d302c0eac0a04ea842830)²¹

²¹<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/bdf30d6ed9d53f7b917d302c0eac0a04ea842830>

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](http://requirejs.org/docs/api.html#modulenotes)²²). 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 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:

²²<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 });
```

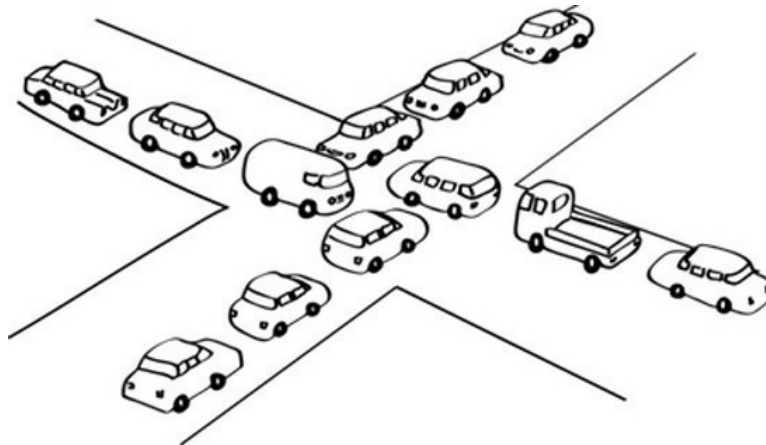
We refresh the page, and boom! An error: “TypeError: ContactManager is undefined” in file *localStorage.js*. Why is this happening? Let’s think about how RequireJS is traversing the dependency tree:

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**²³: 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 this²⁴:



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:

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

²⁴Illustration 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-Appppplication

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     triggers: {
43         "click td a.js-show": "contact:show",
44         "click td a.js-edit": "contact:edit",
45         "click button.js-delete": "contact:delete"
46     },
47
48     events: {
49         "click": "highlightName"
50     },
51
52     flash: function(cssClass){
53         var $view = this.$el;
54         $view.hide().toggleClass(cssClass).fadeIn(800, function(){
55             setTimeout(function(){
56                 $view.toggleClass(cssClass)
57             }, 500);
58         });
59     }
60 });
```

```
58     });
59   },
60
61   highlightName: function(e){
62     this.$el.toggleClass("warning");
63   },
64
65   remove: function(){
66     this.$el.fadeOut(function(){
67       $(this).remove();
68     });
69   }
70 });
71
72 var NoContactsView = Backbone.Marionette.ItemView.extend({
73   template: "#contact-list-none",
74   tagName: "tr",
75   className: "alert"
76 });
77
78 View.Contacts = Backbone.Marionette.CompositeView.extend({
79   tagName: "table",
80   className: "table table-hover",
81   template: "#contact-list",
82   emptyView: NoContactsView,
83   itemView: View.Contact,
84   itemViewContainer: "tbody",
85
86   initialize: function(){
87     this.listenTo(this.collection, "reset", function(){
88       this.appendHtml = function(collectionView, itemView, index){
89         collectionView.$el.append(itemView.el);
90       }
91     });
92   },
93
94   onCompositeCollectionRendered: function(){
95     this.appendHtml = function(collectionView, itemView, index){
96       collectionView.$el.prepend(itemView.el);
97     }
98   }
99 });
```



```
100     });
101
102     return ContactManager.ContactsApp.List.View;
103 });
```

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/contact-s/list/list_controller.js` will be able to:

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

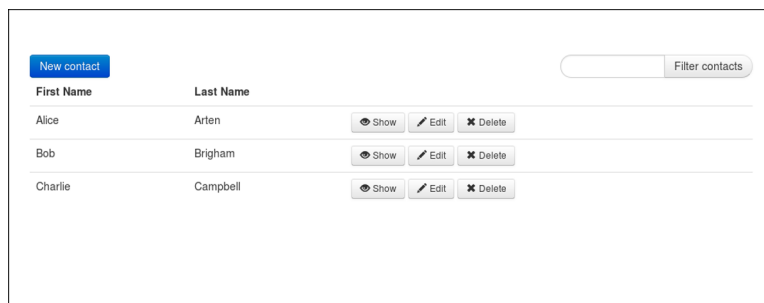
Let's now use our views from within the controller:

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

[illegible]

```
24         args.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](https://raw.githubusercontent.com/diciccale/requirejs-underscore-tpl/master/underscore-tpl.js)²⁵ and save in `assets/js/vendor/underscore-tpl.js`. As the template loader depends on RequireJS's text loader, get it from [here](https://raw.githubusercontent.com/requirejs/text/master/text.js)²⁶ and save it in `assets/js/vendor/text.js`. Then, we need to tell RequireJS about these additions (lines 10-11 and line 18):

²⁵<https://raw.githubusercontent.com/diciccale/requirejs-underscore-tpl/master/underscore-tpl.js>

²⁶<https://raw.githubusercontent.com/requirejs/text/master/text.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     localStorage: "vendor/backbone.localStorage",
9     marionette: "vendor/backbone.marionette",
10    text: "vendor/text",
11    tpl: "vendor/underscore-tpl",
12    underscore: "vendor/underscore"
13  },
14
15  shim: {
16    // edited for brevity
17    localStorage: ["backbone"],
18    tpl: ["text"]
19  }
20 });
21
22 require(["app"], function(ContactManager){
23   ContactManager.start();
24 });
```

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     triggers: {
50         "click td a.js-show": "contact:show",
51         "click td a.js-edit": "contact:edit",
52         "click button.js-delete": "contact:delete"
53     },
54
55     events: {
56         "click": "highlightName"
57     },
58
59     flash: function(cssClass){
60         var $view = this.$el;
61         $view.hide().toggleClass(cssClass).fadeIn(800, function(){
62             setTimeout(function(){
63                 $view.toggleClass(cssClass)
64             }, 500);
65         });
66     },
67
68     highlightName: function(e){
69         this.$el.toggleClass("warning");
70     },
71
72     remove: function(){
73         this.$el.fadeOut(function(){
74             $(this).remove();
75         });
76     }
77 });
```

```

78
79     var NoContactsView = Backbone.Marionette.ItemView.extend({
80         template: noneTpl,
81         tagName: "tr",
82         className: "alert"
83     });
84
85     View.Contacts = Backbone.Marionette.CompositeView.extend({
86         tagName: "table",
87         className: "table table-hover",
88         template: listTpl,
89         emptyView: NoContactsView,
90         itemView: View.Contact,
91         itemViewContainer: "tbody",
92
93         initialize: function(){
94             this.listenTo(this.collection, "reset", function(){
95                 this.appendHtml = function(collectionView, itemView, index){
96                     collectionView.$el.append(itemView.el);
97                 }
98             });
99         },
100
101         onCompositeCollectionRendered: function(){
102             this.appendHtml = function(collectionView, itemView, index){
103                 collectionView.$el.prepend(itemView.el);
104             }
105         }
106     });
107 });
108
109     return ContactManager.ContactsApp.List.View;
110 });

```



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](https://github.com/SlexAxton/require-handlebars-plugin)²⁷.

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

²⁷<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!



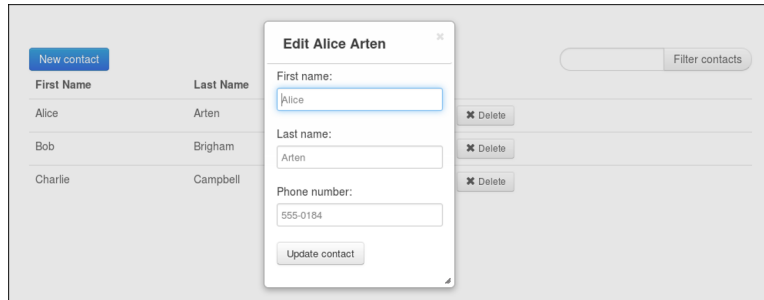
Git commit displaying the contacts in a composite view:

[efaf557ca95b5dcafd449dee5e183cf9641ff66c](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/efaf557ca95b5dcafd449dee5e183cf9641ff66c)²⁸

²⁸<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/efaf557ca95b5dcafd449dee5e183cf9641ff66c>

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, args){
9    console.log("edit button clicked");
10 });
11
12 contactsListView.on("itemview:contact:delete", function(childView, args){
13   args.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, args){
2   var model = args.model;
3   var view = new ContactManager.ContactsApp.Edit.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 });
```

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      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, args){
2     require(["apps/contacts/edit/edit_view"], function(EditView){
3         var model = args.model;
4         var view = new EditView.Contact({
5             model: model
6         });
7
8         view.on("form:submit", function(data){
9             if(model.save(data)){
10                 childView.render();
11                 view.trigger("dialog:close");
12                 childView.flash("success");
13             }
14             else{
15                 view.triggerMethod("form:data:invalid", model.validationError);
16             }
17         });
18
19         ContactManager.dialogRegion.show(view);
20     });
21 });

```

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 line 4 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 = $("", { 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:

[b9abf892593a6c008fed9bc81d5580a40a4e57e](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/b9abf892593a6c008fed9bc81d5580a40a4e57e)²⁹

²⁹<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/b9abf892593a6c008fed9bc81d5580a40a4e57e>

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

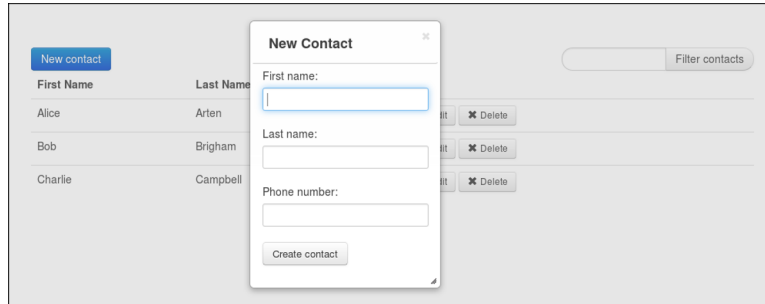
```
1 define(["app", "apps/contacts/common/views"],
2         function(ContactManager, CommonViews){
3     ContactManager.module("ContactsApp.New.View", function(View, ContactManager,
4                                                                Backbone, Marionette, $, _){
5         View.Contact = CommonViews.Form.extend({
6             title: "New Contact",
7
8             onRender: function(){
9                 this.$(".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, args){
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         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:

[5e76e5be0e840186f8fa0d16adde76009df68349](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/5e76e5be0e840186f8fa0d16adde76009df68349)³⁰

³⁰<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/5e76e5be0e840186f8fa0d16adde76009df68349>

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        }

```



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

[69ac4611cbcd69d5a955171d60c901d26daed2f4](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/69ac4611cbcd69d5a955171d60c901d26daed2f4)³¹

³¹<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/69ac4611cbcd69d5a955171d60c901d26daed2f4>

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

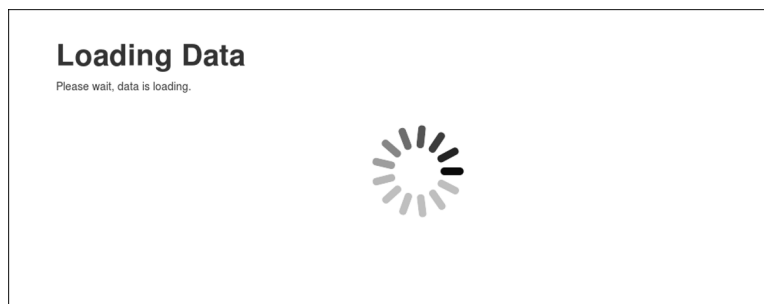
```
1 define(["app", "tpl!common/templates/loading.tpl", "spin.jquery"],
2       function(ContactManager, loadingTpl){
3   ContactManager.module("Common.Views", function(Views, ContactManager,
4       Backbone, Marionette, $, _){
5     // edited for brevity
6   });
7
8   return ContactManager.Common.Views;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

[53414168d8e1c308eac1b7fcd079164a212d82ac](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/53414168d8e1c308eac1b7fcd079164a212d82ac)³²

³²<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/53414168d8e1c308eac1b7fcd079164a212d82ac>

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, args){
2   console.log("show link clicked");
3 });
4
5 contactsListView.on("itemview:contact:edit", function(childView, args){
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`

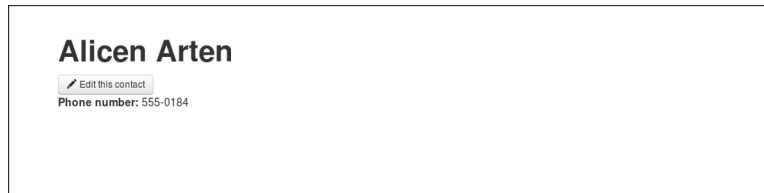
```
1 ContactsApp.Router = Marionette.AppRouter.extend({
2   appRoutes: {
3     "contacts": "listContacts",
4     "contacts(?filter=:criterion)": "listContacts",
5     "contacts/:id": "showContact"
6   }
7 });
8
9 var API = {
10   listContacts: function(criterion){
11     // edited for brevity
12   },
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:

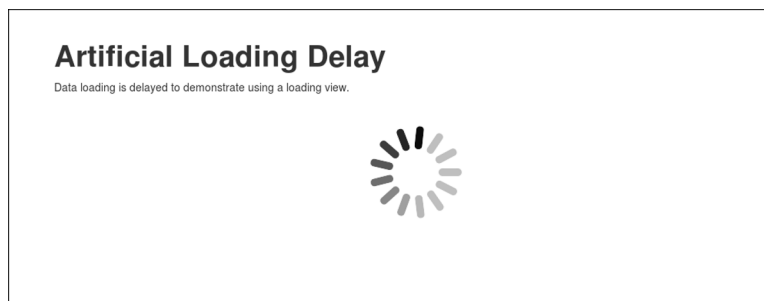
assets/js/apps/contacts/list/list_controller.js

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

And with that, our app displays contacts properly!



Displaying a contact



The loading view



Git commit displaying contacts:

[e7bd3faf092203fa9e04429d11d01707f19994](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/e7bd3faf092203fa9e04429d11d01707f19994)³³

³³<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/e7bd3faf092203fa9e04429d11d01707f19994>

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 = {

```

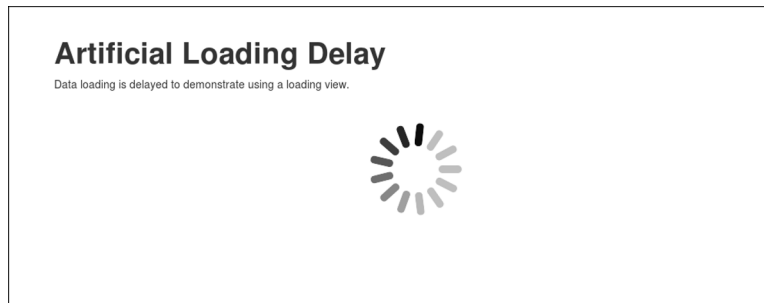
```
9  // edited for brevity
10
11  showContact: function(id){
12    // edited for brevity
13  },
14
15  editContact: function(id){
16    require(["apps/contacts/edit/edit_controller"], function(EditController){
17      EditController.editContact(id);
18    });
19  }
20 };
21
22 // edited for brevity
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.



The screenshot shows a web form titled "Edit Alicen Arten". It contains three input fields: "First name:" with the value "Alicen", "Last name:" with the value "Arten", and "Phone number:" with the value "555-0184". Below these fields is a button labeled "Update contact".

Editing a contact



The loading view



Git commit implementing contact editing:

[57afa9b845899fba124279adcdef9f5af936bf0b](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/57afa9b845899fba124279adcdef9f5af936bf0b)³⁴

³⁴<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/57afa9b845899fba124279adcdef9f5af936bf0b>

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         });
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.

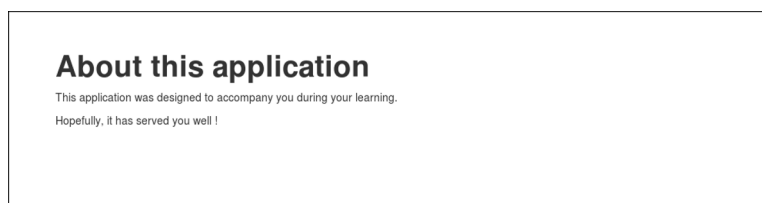
assets/js/app.js

```

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!



The “about” message



Git commit reimplementing the “about” sub-application:

[48141812e6dc3a7563aef35fa9f24d001c5040a7](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/48141812e6dc3a7563aef35fa9f24d001c5040a7)³⁵

³⁵<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/48141812e6dc3a7563aef35fa9f24d001c5040a7>

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
11       // edited for brevity
12     });
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 }

```

```
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 });
```

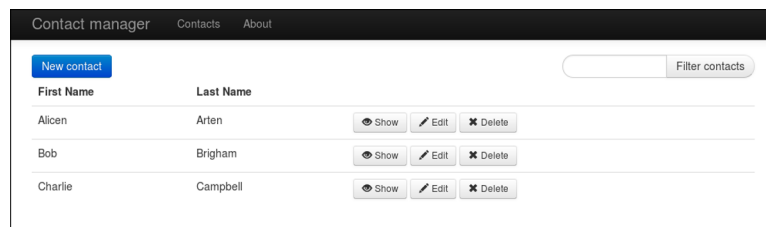
Now that our “header” app is fully reimplemented, let’s tackle the tricky part. As you can see on lines 16-18 above, the sub-app registers a “start” listener with code to execute when this sub-module is started. Since we have *not* set this module’s `startWithParent` value to `false`, it will be started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

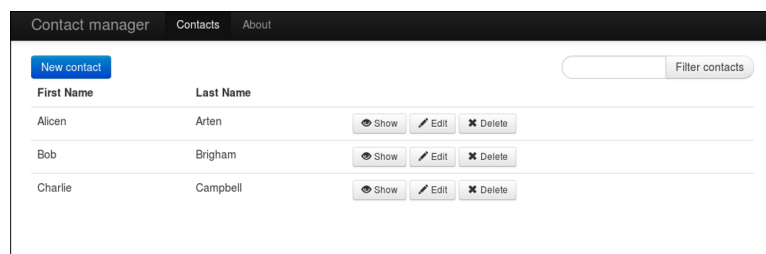
assets/js/apps/header/header_app.js

```

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:



The highlighted “Contacts” entry



Git commit reimplementing the “header” sub-application:

[e088787a5a9a9affd18d54ea418f10f909715c37](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/e088787a5a9a9affd18d54ea418f10f909715c37)³⁶

³⁶<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/e088787a5a9a9affd18d54ea418f10f909715c37>

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 got 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
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](#)³⁷).

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](#)³⁸ 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:

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         // edited for brevity
5
6         Entities.ContactCollection = Backbone.Collection.extend({
7             url: "contacts",
8             model: Entities.Contact,
9             comparator: "firstName"
10        });
```

³⁷<https://github.com/marionettejs/backbone.marionette/wiki/Using-marionette-with-requirejs#example-with-central-vent>

³⁸<https://github.com/marionettejs/backbone.marionette/wiki/Using-marionette-with-requirejs#advanced-usage>

```
11
12     Entities.configureStorage(Entities.ContactCollection);
13
14     var initializeContacts = function(){
15         var contacts = new Entities.ContactCollection([
16             { id: 1, firstName: "Alice", lastName: "Arten",
17               phoneNumber: "555-0184" },
18             { id: 2, firstName: "Bob", lastName: "Brigham",
19               phoneNumber: "555-0163" },
20             { id: 3, firstName: "Charlie", lastName: "Campbell",
21               phoneNumber: "555-0129" }
22         ]);
23         contacts.forEach(function(contact){
24             contact.save();
25         });
26         return contacts.models;
27     };
28
29     var API = {
30         getContactEntities: function(){
31             var contacts = new Entities.ContactCollection();
32             var defer = $.Deferred();
33             contacts.fetch({
34                 success: function(data){
35                     defer.resolve(data);
36                 }
37             });
38             var promise = defer.promise();
39             $.when(promise).done(function(contacts){
40                 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(["app",
2      "tpl!apps/contacts/show/templates/missing.tpl",
3      "tpl!apps/contacts/show/templates/view.tpl"],
4      function(ContactManager, missingTpl, viewTpl){
5  return {
6      MissingContact: Marionette.ItemView.extend({
7          template: missingTpl
8      }),
9
10     Contact: Marionette.ItemView.extend({
11         template: viewTpl,
12
13         events: {
14             "click a.js-edit": "editClicked"
15         },
16
17         editClicked: function(e){
18             e.preventDefault();
19             this.trigger("contact:edit", this.model);
20         }
21     })
22 };
23 });

```

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 }
16 });
```

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(ContactsApp, 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 `delete`³⁹ 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:

³⁹<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        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:

[c434ac6cc6e8bf50a43a432b10035ff1c338ea65](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/c434ac6cc6e8bf50a43a432b10035ff1c338ea65)⁴⁰

⁴⁰<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/c434ac6cc6e8bf50a43a432b10035ff1c338ea65>

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:

[d06665e7bd81e47e37363622918e32033c1cadac](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/d06665e7bd81e47e37363622918e32033c1cadac)⁴¹

⁴¹<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/d06665e7bd81e47e37363622918e32033c1cadac>

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](#)⁴²). Then, download the `r.js` file from [here](#)⁴³ and store it in your application's root path.⁴⁴

With that out of the way, we're almost ready to run the optimizer. We'll need a [build file](#)⁴⁵:

`assets/js/build.js`

```
1 ({
2     baseUrl: ".",
3     name: "require_main",
4     mainConfigFile: "require_main.js",
5     out: "require_main.built.js",
6     wrapShim: true
7 })
```

Here's what each attribute does (find out more [here](#)⁴⁶):

- `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;

⁴²<http://nodejs.org/download/>

⁴³<http://requirejs.org/docs/download.html#rjs>

⁴⁴Note that, per the [instructions](#), you can also use the optimizer after installing it with npm.

⁴⁵<http://requirejs.org/docs/optimization.html#wholeproject>

⁴⁶<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;
- `wrapShim` allows us to use AMD modules (e.g. Backbone) as dependencies in our shim configuration. Without this, AMD modules might be evaluated too late in our optimized file, resulting in an error. For more about this, refer to the [documentation](#)⁴⁷ (section starting with "Important notes for shim config", search for `wrapShim`).

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
```

⁴⁷<http://requirejs.org/docs/api.html#config-shim>

```

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:

`c9a85f609dcaadc5f08ca4fb024e3fab3bfd0a4d`⁴⁸

Our app now loads properly, but it seems a little wasteful doesn't it? We're loading RequireJS will 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](https://raw.githubusercontent.com/jrburke/almond/latest/almond.js)⁴⁹ and save it as `assets/js/vendor/almond.js`.



Using Almond does come with some [restrictions](https://github.com/jrburke/almond#restrictions)⁵⁰, 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).

⁴⁸<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/c9a85f609dcaadc5f08ca4fb024e3fab3bfd0a4d>

⁴⁹<https://raw.githubusercontent.com/jrburke/almond/latest/almond.js>

⁵⁰<https://github.com/jrburke/almond#restrictions>

Then, we need to change our build file to include Almond in the optimized 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     wrapShim: true
8 })
```

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, aaaaaaaaaaand 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   wrapShim: true,
8   findNestedDependencies: true
9 })
```

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:

[7a51af4e6ea3cb400813e4dd87fab3e072c311c0](https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/7a51af4e6ea3cb400813e4dd87fab3e072c311c0)⁵¹

⁵¹<https://github.com/davidsulc/structuring-backbone-with-requirejs-and-marionette/commit/7a51af4e6ea3cb400813e4dd87fab3e072c311c0>

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](#)⁵². Help others determine if the book is right for them!

And once you're ready to progress in your Marionette knowledge, take a look at my [Marionette: A Serious Progression](#)⁵³ book!

Would you like me to cover another subject in an upcoming book? Let me know by email at 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 davidsulc.com/mailling_list⁵⁴. No spam, I promise!

You can also follow me on Twitter: [@davidsulc](#)

⁵²<https://leanpub.com/structuring-backbone-with-requirejs-and-marionette/feedback>

⁵³<https://leanpub.com/structuring-backbone-with-requirejs-and-marionette>

⁵⁴http://davidsulc.com/mailling_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](https://leanpub.com/marionette-gentle-introduction)⁵⁵.

And if you're looking to learn more advanced Marionette concepts, make sure you read [Marionette: A Serious Progression](https://leanpub.com/marionette-serious-progression)⁵⁶! It introduces advanced techniques for you to use on your apps:

- dealing with legacy APIs by rewriting `Backbone.sync`;
- wrapping third-party APIs to be able to use them as Backbone models and collections;
- server interaction: how to handle server responses, updating client-side data when the server data changed, etc.
- handling authentication and authorization;
- using model relationships;
- using i18n client-side;
- developing reusable views (e.g. a paginated view);
- improving memory management: start/stop models, using the prototype, using `listenTo`;

⁵⁵<https://leanpub.com/marionette-gentle-introduction>

⁵⁶<https://leanpub.com/marionette-serious-progression>