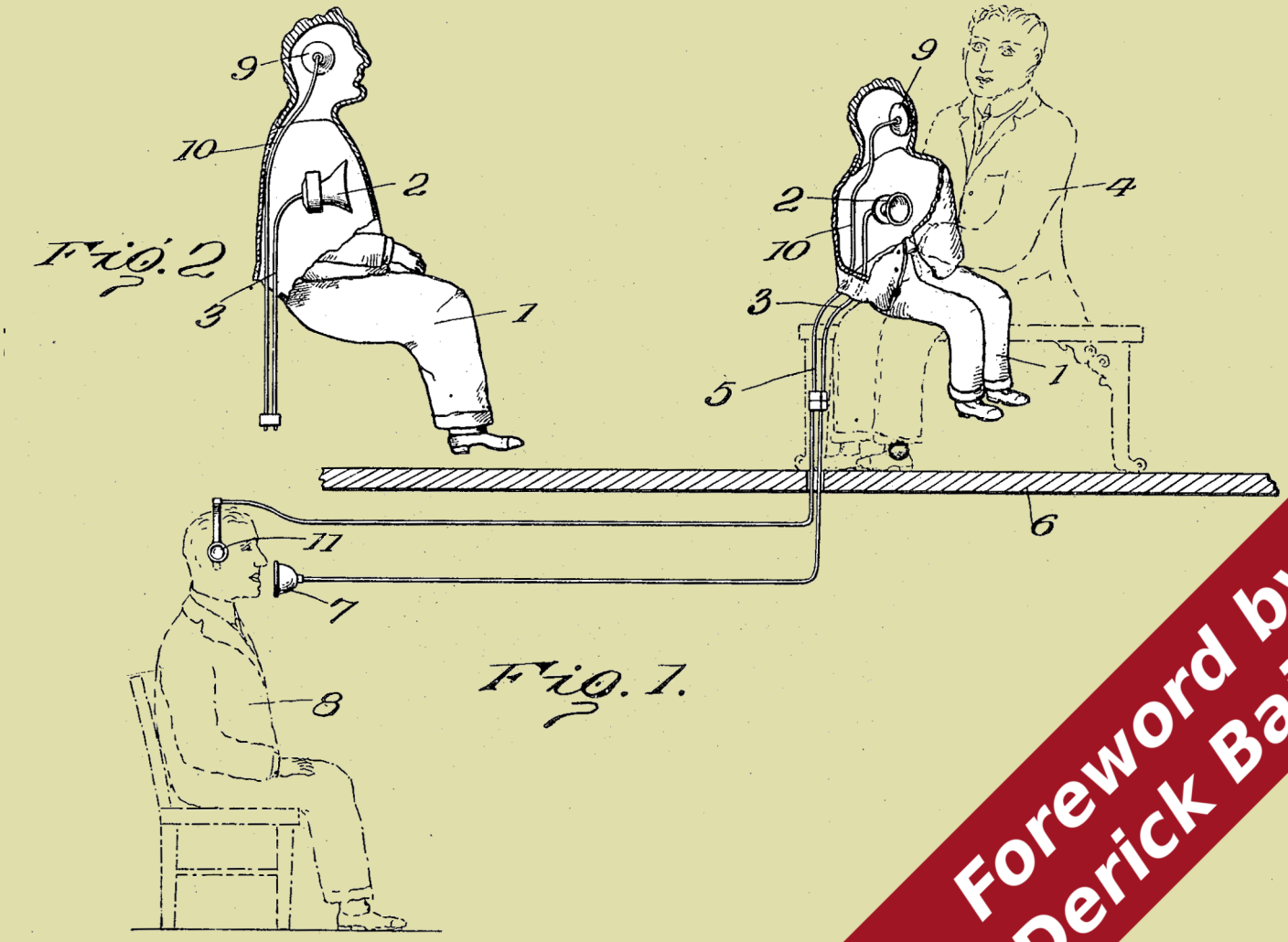
Build a Marionette.js app, one step at a time

Backbone Marionette.js



Foreword by Derick Bailey

A Gentle Introduction

by David Sulc

Backbone.Marionette.js: A Gentle Introduction

Build a Marionette.js app, one step at a time

David Sulc

Also By David Sulc

Structuring Backbone Code with RequireJS and Marionette Modules

Contents

Foreword from Derick Bailey

The open and flexible nature of Marionette allows it to be used in more ways than can be imagined. I've seen applications that I would never have dreamed of, built with it: games, financial reporting tools, search engines, mobile applications, ticket sales and e-commerce, database management systems, and more. The down side of this flexibility, though, is documentation. Creating a comprehensive suite of documents that show all of the different ways that the parts can be combined is an overwhelming task.

Plenty of introductory articles, blog posts and videos exist out there on the web. But, very little of this information moved beyond the simple patterns of "replace this Backbone code with this Marionette code". Putting the pieces together requires a new level of abstraction and thinking, and a new set of patterns to work with. And this documentation simply did not exist, even if some application developers other than myself were using these higher level patterns.

It wasn't until Brian Mann started producing his BackboneRails.com[1] screencasts, and David Sulc started writing this book, that the Marionette community began to see all of the patterns of implementation that I was advocating, in one place. And I'm so very happy to see David writing this book and Brian producing those screencasts. The community needs this information. The documentation gap is finally being closed.

This is the book that I wanted to write, but never had time to write. It is a complete and thorough introduction to building scalable applications with Marionette.js. Better still, it advocates and demonstrates the same patterns and principles that I use in my own applications. You owe it to yourself to work through all of the exercises in this book, even if you are a seasoned Backbone and Marionette developer. David has done a wonderful job of breaking down the architecture of large Marionette applications, lighting the path for each step of the journey.

– Derick Bailey, creator of Marionette.js[2]

[1] http://BackboneRails.com

[2] http://marionettejs.com

Cover Credits

The cover image depicts a “theatrical appliance” designed to help ventriloquists by having a partner voice their puppet. The image is from patent application 1,197,543 filed in 1914, which you can view here[3].

[3] http://patentimages.storage.googleapis.com/pdfs/US1197543.pdf

Who This Book is For

This book is for web developers who want to build highly interactive javascript applications. This book will cover using Backbone.Marionette.js to achieve that goal, and will empower you to build your own applications by understanding how Marionette apps are built.

All you'll need to follow along is a basic understanding of javascript and the DOM (Document Object Model), such as being able to manipulate elements on the page using a jQuery selector. In other words, if you've used a few jQuery libraries here and there, you should be able to follow along just fine.

Following Along with Git

This book is a step by step guide to building a complete Marionette.js application. As such, it's accompanied by source code in a Git repository hosted at https://github.com/davidsulc/marionette-gentle-introduction[4].

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

Git commit with our scaffold code:

e0eac08aa3287522fcab3301cf03ff81a60f1ddf[5]

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[6], 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[7]. 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[8]
- To get a copy of the source code repository on your computer, open a command line and run

```
git clone git://github.com/davidsulc/marionette-gentle-introduction.git
```

- From the command line move into the `marionette-gentle-introduction` folder that Git created in the step above, and execute

[4] https://github.com/davidsulc/marionette-gentle-introduction
[5] https://github.com/davidsulc/marionette-gentle-introduction/commit/e0eac08aa3287522fcab3301cf03ff81a60f1ddf
[6] https://github.com/davidsulc/marionette-gentle-introduction/commits/master
[7] http://git-scm.com/book
[8] https://help.github.com/articles/set-up-git

```
git show e0eac08aa3287522fcab3301cf03ff81a60f1ddf
```

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 e0eac08aa3287522fcab3301cf03ff81a60f1ddf
```

- 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[9]. 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 tackle your own Marionette projects, so it assumes very little knowledge. 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.

[9] https://github.com/davidsulc/marionette-gentle-introduction/archive/master.zip

Setting Up

In this book, we're going to build an application step by step. The finished application can be seen at http://davidsulc.github.io/marionette-gentle-introduction[10].

The first order of business before we can start programming our application, is setting up our "scaffold". We'll be using pretty basic stuff:

- Bootstrap CSS and their starter template[11]
- Marionette.js and dependencies

Easy, right?

Asset Organization

Before we get in the thick of things, let's quickly consider how we'll organize the various files (CSS, JS, etc.) that we'll be using in this project. In order to maintain our sanity as the files increase in number, we'll need some sort of system to keep the files tidy so we don't spend our time looking for things:

- project folder
  - index.html
  - assets
    - css
    - img
    - js
      - vendor

Within the *js* folder, we'll use a *vendor* subfolder to contain the javascript files that are provided ready-to-use (e.g. Marionette.js, jQuery, etc.). The javascript code that we will produce as we build our application will go within the *js* folder.

[10] http://davidsulc.github.io/marionette-gentle-introduction

[11] http://twitter.github.io/bootstrap/examples/starter-template.html

Getting Our Assets

The URLs provided below link to the most recent library versions, which might not be backward compatible. To make your learning smoother and avoid unnecessary difficulties, you might want to instead download the library versions used in the book from the repository[12].

Let's start by getting the various javascript libraries we'll need, saving them in *assets/js/vendor*:

- jquery[13]
- json2[14]
- underscore[15]
- backbone[16]
- backbone.marionette[17]

You'll notice we'll be using the development (uncompressed) versions, mainly for the convenience of having error messages that make sense. Besides, most modern web frameworks provide means to minify/obfuscate javascript when going into production.

Next, let's get the Bootstrap CSS: download and extract the zip file[18], then move *css/bootstrap.css* to your project folder in *assets/css/bootstrap.css*. In addition, move the images Bootstrap uses from *img* to your project folder in *assets/img*.

So now that we've got the javascript libraries and CSS we'll be needing, let's go ahead and create our HTML, based on the Bootstrap starter template[19]. We'll modify it slightly, so we don't have non-functional things in our page (e.g. menu items that don't work), and we'll also need to include the various javascript files we've just obtained. Here's what we'll start with:

[12] https://github.com/davidsulc/marionette-gentle-introduction
[13] http://code.jquery.com/jquery-1.10.2.js
[14] https://raw.github.com/douglascrockford/JSON-js/master/json2.js
[15] http://underscorejs.org/underscore.js
[16] backbonejs.org/backbone.js
[17] http://marionettejs.com/downloads/backbone.marionette.js
[18] http://twitter.github.io/bootstrap/assets/bootstrap.zip
[19] http://twitter.github.io/bootstrap/examples/starter-template.html

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    </head>
8
9    <body>
10
11     <div class="navbar navbar-inverse navbar-fixed-top">
12       <div class="navbar-inner">
13         <div class="container">
14           <span class="brand">Contact manager</span>
15         </div>
16       </div>
17     </div>
18
19     <div class="container">
20       <p>Here is static content in the web page. You'll notice that it gets
21           replaced by our app as soon as we start it.</p>
22     </div>
23
24     <script src="./assets/js/vendor/jquery.js"></script>
25     <script src="./assets/js/vendor/json2.js"></script>
26     <script src="./assets/js/vendor/underscore.js"></script>
27     <script src="./assets/js/vendor/backbone.js"></script>
28     <script src="./assets/js/vendor/backbone.marionette.js"></script>
29
30   </body>
31 </html>
```

Pay attention to the order we're including the javascript files (lines 24-28): dependencies must be respected. For example, Backbone depends on jQuery and Underscore, so it gets included after those two libraries.

If you open `index.html` now, you'll see we're not quite done: you can't see the placeholder text because it's hidden underneath the navigation bar on top. So let's quickly create a small

`application.css` we'll put in *assets/css* and include in our `index.html` file right after the Bootstrap CSS (line 6). Here's our `application.css`:

application.css

```
1 body {
2   /* 60px to move the container down and
3    * make room for the navigation bar */
4   padding-top: 60px;
5 }
```

Git commit with our scaffold code:

e0eac08aa3287522fcab3301cf03ff81a60f1ddf[20]

We can now get started with our app! We'll develop a "contact manager" application, which will store contact information on people (like a phone book). We're going to develop it step by step, explaining at each stage how the different Marionette components work together, and why we're refactoring code.

[20] https://github.com/davidsulc/marionette-gentle-introduction/commit/e0eac08aa3287522fcab3301cf03ff81a60f1ddf

Displaying a Static View

Now that we have the basics set up, let's use Marionette to display content in our `index.html`. We'll start by putting everything within the HTML file. But as you can guess, this approach isn't advisable for anything beyond a trivial application: you'd lose your mind. So we'll quickly get around to refactoring our simple application into something more robust.

Let's start by adding some javascript code at the bottom of our `index.html`:

```
<script type="text/javascript">
  var ContactManager = new Marionette.Application();

  ContactManager.start();
</script>
```

What did we do? Nothing really exciting: we simply declared a new Marionette application, then started it. If you refresh the `index.html` page in your browser, you'll see absolutely nothing has changed... This isn't surprising: our application doesn't do anything yet.

Let's now make our app display a message to the console once it has started:

```
<script type="text/javascript">
  var ContactManager = new Marionette.Application();

  ContactManager.on("initialize:after", function(){
    console.log("ContactManager has started!");
  });

  ContactManager.start();
</script>
```

Note we've defined the `initialize:after` handler code **before** we start the application.

If you refresh the page with (e.g.) Firebug's console open, you'll see the message we've just added. How about we make the app do somehting a little more useful (and visual) by displaying some static content?

Before we can have our app do that, we need to fulfill a few preconditions:

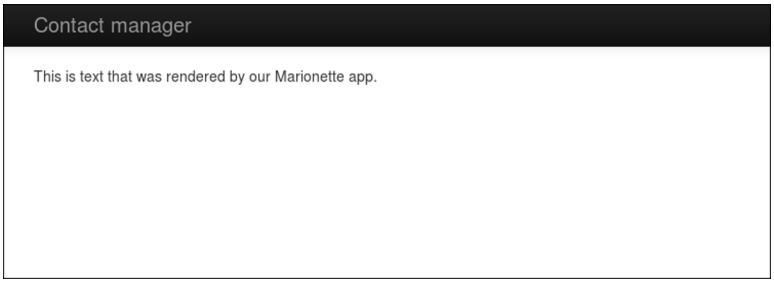
- We need to have a view to display
- We need a template to provide to our view, so it knows what to display and how to display it
- We need to have an area in our page where Marionette can insert the view, in order to display it

Let's have a quick look at what our `index.html` looks like after these additions:

index.html

```
<div id="main-region" class="container">
  <p>Here is static content in the web page. You'll notice that it gets
  replaced by our app as soon as we start it.</p>
</div>

<script type="text/template" id="static-template">
  <p>This is text that was rendered by our Marionette app.</p>
</script>

<!-- The javascript libraries get included here (edited for brevity) -->

<script type="text/javascript">
  var ContactManager = new Marionette.Application();

  ContactManager.addRegions({
    mainRegion: "#main-region"
  });

  ContactManager.StaticView = Marionette.ItemView.extend({
    template: "#static-template"
  });

  ContactManager.on("initialize:after", function(){
    var staticView = new ContactManager.StaticView();
    ContactManager.mainRegion.show(staticView);
  });

  ContactManager.start();
</script>
```

For brevity, I've included only the HTML below the navigation bar (and the lines to include the javascript libraries were also edited). Here's what our app looks like:



Displaying a static view

Instead of constantly declaring global variables to store things like our view definition (and thereby polluting the global namespace), we're attaching them to our app (as attributes) with (e.g.) `ContactManager.StaticView`.

What did we do with this change? We've simply added the features that are needed to display our view, as discussed above. Here's what it boils down to:

- We define a template on lines 6-8 and tell our view to use it on line 20
- We define a region for our app to display views (lines 15-17)
- We define a view to display on lines 19-21

Let's take a moment to explain a few aspects in more detail. In Marionette (and Backbone in general), views need to be provided with a template to display. This is because they have different responsibilities:

- templates
  - are basically HTML
  - govern "how things should be displayed" (what HTML should be in the view, CSS styles, where data should be displayed, etc.)
- views
  - are javascript objects
  - take care of "reacting to things that happen" (clicks, keeping track of a model, etc.)

This can be somewhat confusing if you're used to working with an MVC web framework such as Rails. In these, the template and view are typically mixed in the "view" part of the MVC: they get data from the model instances provided by the controller, then generate some HTML that is sent to the browser. What you must keep in mind is that once the HTML is rendered by these frameworks, it never gets modified: a new view may get created by the same controller (e.g. on refresh), but **this particular instance** will never be *modified*.

In Marionette apps, however, a view gets instantiated and the user will usually interact with it (click things, modify data somewhere else, etc.). Since we're not refreshing the page each time the user

clicks, we need to manage user interactions (clicks, e.g.) within the view. But in addition, if the user changes some data, the views displaying that data must update immediately (and remember: there won't be any server interaction, or page refresh). In other words, if a user modifies a contact's phone number within a popup window, all the views displaying that contact must be refreshed when the data is saved. But how can we refresh the data without the server, and how can we know the contact's information has changed? This is the view's responsibility in Marionette: it monitors the models it's displaying, and if those models change, the view renders itself again (using the same template). And to follow the "separation of concerns" pattern[21], the "views" functionality has been separated into templates (how to display information) and views (how to react to changes in the environment).

As you see more Marionette code, you'll notice that models, views, etc. get instantiated by providing a javascript object containing key-value properties. In javascript, `var myModel = { myAttribute: "myValue" }` declares a valid object, and `myModel.myAttribute` will return "myValue". To learn more about javascript objects, see Working with Objects[22] by the Mozilla Developer Network.

Our template is defined within a `script` tag, with `type` attribute of `text/template`. This is simply to trick the browser:

- it's not HTML, so the browser won't try to display it
- it's not javascript, so the browser won't try to execute it

However, we can conveniently set an `id` attribute to the `script` tag, allowing us to select it with jQuery. And that's exactly what's happening: on line 20, we're indicating which template to use by giving our view a jQuery selector and Marionette does the rest for us.

What about the region on lines 15-17? Well, as we mentioned above, Marionette will need somewhere within our page to display our view. To create a region to contain our view, we've simply provided a jQuery selector to the DOM element that will contain our view (notice we've added an `id` attribute to the tag on line 1). Of course, we'll see that having many regions come in handy with a more complex interface, but that's for later…

Now, instead of displaying a simple message in the console, we instantiate a new view when our application has started and use our pre-defined region to display it. You'll now understand how region definitions work (line 16): the key on the left is what we call our region within our Marionette application, while the value on the right is a jQuery selector present in our page. In other words, by declaring a region with `mainRegion: "#main-region"`, we're saying that calling

[21] http://en.wikipedia.org/wiki/Separation_of_concerns

[22] https://developer.mozilla.org/en/docs/JavaScript/Guide/Working_with_objects

```
ContactManager.mainRegion.show(staticView);
```

means "put the contents of `staticView` inside the element corresponding to the jQuery selector `#main-region`".

With our latest modifications, our `index.html` now looks like this:

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Marionette Contact Manager</title>
    <link href="./assets/css/bootstrap.css" rel="stylesheet">
    <link href="./assets/css/application.css" rel="stylesheet">
  </head>

  <body>

    <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <span class="brand">Contact manager</span>
        </div>
      </div>
    </div>

    <div id="main-region" class="container">
      <p>Here is static content in the web page. You'll notice that it gets
      replaced by our app as soon as we start it.</p>
    </div>

    <script type="text/template" id="static-template">
      <p>This is text that was rendered by our Marionette app.</p>
    </script>

    <script src="./assets/js/vendor/jquery.js"></script>
    <script src="./assets/js/vendor/json2.js"></script>
    <script src="./assets/js/vendor/underscore.js"></script>
    <script src="./assets/js/vendor/backbone.js"></script>
    <script src="./assets/js/vendor/backbone.marionette.js"></script>
```

```
34
35      <script type="text/javascript">
36        ContactManager = new Marionette.Application();
37
38        ContactManager.addRegions({
39          mainRegion: "#main-region"
40        });
41
42        ContactManager.StaticView = Marionette.ItemView.extend({
43          template: "#static-template"
44        });
45
46        ContactManager.on("initialize:after", function(){
47          var staticView = new ContactManager.StaticView();
48          ContactManager.mainRegion.show(staticView);
49        });
50
51        ContactManager.start();
52      </script>
53    </body>
54  </html>
```

We'll see more of Marionette's `ItemView` later on, but if you're in a hurry you can refer to the documentation[23].

Git commit to display our static view:

cc1dfe888a2cfaad206b83cd1148cc6e4dc64a0c[24]

Dynamically Specifying a View Template

In the code above, we've specified the template as a permanent attribute on our view because we're always going to want to use the same template in this case. But it's also possible to dynamically provide templates to views, so let's see how that's done. We already have our app working to display a static view that is "hard-coded" within our view definition. So let's override it at runtime with a different template.

First, we need to define a new template to use, which we'll include right below our existing template:

[23] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.itemview.md

[24] https://github.com/davidsulc/marionette-gentle-introduction/commit/cc1dfe888a2cfaad206b83cd1148cc6e4dc64a0c

```
<script type="text/template" id="different-static-template">
  <p>Text from a different template...</p>
</script>
```

Nothing special going on here, we've simply got different text to demonstrate the different template being used. Next, we need to provide the template to the view when we instantiate it, like so:

```
  var staticView = new ContactManager.StaticView({
    template: "#different-static-template"
  });
```

And there we have it! When this view is displayed in our main region, the new text will be displayed. Here's our `index.html` with a dynamically provided template:

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Marionette Contact Manager</title>
    <link href="./assets/css/bootstrap.css" rel="stylesheet">
    <link href="./assets/css/application.css" rel="stylesheet">
  </head>

  <body>

    <div class="navbar navbar-inverse navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <span class="brand">Contact manager</span>
        </div>
      </div>
    </div>

    <div id="main-region" class="container">
      <p>Here is static content in the web page. You'll notice that it gets
      replaced by our app as soon as we start it.</p>
    </div>

    <script type="text/template" id="static-template">
```

```
26        <p>This is text that was rendered by our Marionette app.</p>
27      </script>
28
29      <script type="text/template" id="different-static-template">
30        <p>Text from a different template...</p>
31      </script>
32
33      <script src="./assets/js/vendor/jquery.js"></script>
34      <script src="./assets/js/vendor/json2.js"></script>
35      <script src="./assets/js/vendor/underscore.js"></script>
36      <script src="./assets/js/vendor/backbone.js"></script>
37      <script src="./assets/js/vendor/backbone.marionette.js"></script>
38
39      <script type="text/javascript">
40        var ContactManager = new Marionette.Application();
41
42        ContactManager.addRegions({
43          mainRegion: "#main-region"
44        });
45
46        ContactManager.StaticView = Marionette.ItemView.extend({
47          template: "#static-template"
48        });
49
50        ContactManager.on("initialize:after", function(){
51          var staticView = new ContactManager.StaticView({
52            template: "#different-static-template"
53          });
54          ContactManager.mainRegion.show(staticView);
55        });
56
57        ContactManager.start();
58      </script>
59    </body>
60  </html>
```

Specifying Other View Attributes

Let's see how you can provide options that are passed to the view, enabling you to specify the HTML tag that is used to render your view, add an `id` and `class`, etc.

If you take a look at the source code (with Firebug or a comparable developer tool[25]) after Marionette has rendered our view, you'll see it is contained within a `div` tag:

```
<div>
  <p>This is text that was rendered by our Marionette app.</p>
</div>
```

This is because Marionette needs an element to contain the view being inserted within the DOM, and by default, it's a `div`. However, you can specify various attributes in your view, for example:

```
ContactManager.StaticView = Marionette.ItemView.extend({
  id: "static-view",
  tagName: "span",
  className: "instruction",
  template: "#static-template"
});
```

Such a view definition would generate the following HTML when the view gets rendered:

```
<span id="static-view" class="instruction">
  <p>This is text that was rendered by our Marionette app.</p>
</span>
```

To learn more about view options, take a look at the Backbone documentation for the View constructor[26]. It's worth noting that just like the template property, these options can be provided when the view is being instantiated:

```
var staticView = new ContactManager.StaticView({
  id: "static-view",
  tagName: "span",
  className: "instruction"
});
```

Exercise

Displaying a Single-Item List

Display a single list item within a `<ul>` element, using this template:

[25] To inspect the source code, you'll need a developer tool such as Firebug, or similar for your browser (some browsers have tools built-in). This is because if you use the browser's "View source code" menu entry, it will display the HTML it received originally. But since we've modified it heavily with javascript and we're interested in viewing the current state, we need to display the source using developer tools.

[26] http://backbonejs.org/#View-constructor

```
<script type="text/template" id="list-item-template">
  <li>One item</li>
</script>
```

In addition, leave the `ContactManager.StaticView` template definition as it is, and specify the template to render during the view's instantiation. You can, however, add other attributes to the view definition. You can see the exercise solution at the end of the book.

Displaying a Model

Now that we've covered displaying static content, let's move on to displaying content containing data from a model. As you may know, one of Backbone's selling points is the possibility to structure javascript applications with a Model-View-Controller[27] (MVC) pattern. In this pattern, we use so-called *models* to interact with our data, passing them onto views for rendering the information they contain. You can learn more about models in Backbone's documentation[28].

So let's declare a model within our javascript block, above our view declaration:

```
ContactManager.Contact = Backbone.Model.extend({});
```

That wasn't very hard. What did we do? We simply declared a model named `Contact` and attached it to our `ContactManager` app. As you can see, this model extends Backbone's model definition and inherits various methods from it. When we extend Backbone's base model like this, we provide a javascript object (which is empty in our case) that can contain additional information pertaining to our model (we'll get back to that later).

Same as before, we'll need a template and a view definition before we can display anything in the browser. Let's replace our previous template and `StaticView` with the following:

```
1 <script type="text/template" id="contact-template">
2   <p><%= firstName %> <%= lastName %></p>
3 </script>
4
5 ContactManager.ContactView = Marionette.ItemView.extend({
6   template: "#contact-template"
7 });
```

The template will be included within the HTML `body`, but **outside** of the script block containing our application code. Refer to the full `index.html` included below if you're unsure where this code gets inserted.

You'll notice that we've got some special `<%= %>` tags in there. These serve the same purpose as in many templating languages (ERB in Rails, PHP, JSP, etc.): they allow the templating engine to interpret them and include the resulting output within the rendered result. By default, Marionette

[27] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[28] http://backbonejs.org/#Model

uses Underscore's templating engine[29] where `<%= %>` means output will be displayed, and `<% %>` tags which allow arbitrary javascript to be executed (such as an `if` condition). Since the model is serialized and passed on to the view template, writing `<%= firstName %>` means the model's `firstName` attribute will be displayed.

So how do we display our view with model information? With our definitions written, we still need to create instances of a model and view, then display the view. All of this will happen within the `initialize:after` handler:

```
1  ContactManager.on("initialize:after", function(){
2    var alice = new ContactManager.Contact({
3      firstName: "Alice",
4      lastName: "Arten",
5      phoneNumber: "555-0184"
6    });
7
8    var aliceView = new ContactManager.ContactView({
9      model: alice
10   });
11
12   ContactManager.mainRegion.show(aliceView);
13 });
```

First, we create a model instance with data on lines 2-6: you'll notice we specify various model attributes and their respective values within a javascript object. Then, we create a new view instance and provide the model instance as an attribute on lines 8-10.

Remember how we discussed passing options to the view when it gets instantiated? That's exactly what we're doing here: when we use the contact view, we'll always be using the same template (and have indicated it in the view definition for convenience), but the model we'll want to display will change. Therefore, we leave the model attribute out of the view's definition, and we specify which model to use each time we instantiate a new view instance.

And all that's left to do after that is to display the view within the region (line 12), same as before.

Here's what our `index.html` looks like at this stage:

[29] http://underscorejs.org/#template

```
<div id="main-region" class="container">
  <p>Here is static content in the web page. You'll notice that it gets
  replaced by our app as soon as we start it.</p>
</div>

<script type="text/template" id="contact-template">
  <p><%= firstName %> <%= lastName %></p>
</script>

<!-- The javascript includes are here -->

<script type="text/javascript">
  var ContactManager = new Marionette.Application();

  ContactManager.addRegions({
    mainRegion: "#main-region"
  });

  ContactManager.Contact = Backbone.Model.extend({});

  ContactManager.ContactView = Marionette.ItemView.extend({
    template: "#contact-template"
  });

  ContactManager.on("initialize:after", function(){
    var alice = new ContactManager.Contact({
      firstName: "Alice",
      lastName: "Arten",
      phoneNumber: "555-0184"
    });

    var aliceView = new ContactManager.ContactView({
      model: alice
    });

    ContactManager.mainRegion.show(aliceView);
  });

  ContactManager.start();
</script>
```

And the visual result:

Displaying a model

Git commit to display our basic model view:

7e7e6f5a5c37ceea1b9419396464894e08bf7d23[30]

Using Model Defaults

What if our contact didn't have a first name? We don't want our app to break if the `firstName` attribute is missing: the template would be trying to retrieve an attribute that doesn't exist on the model. How can we manage this case? The functionality we're looking for is default values for model attributes.

To declare default attribute values, simply add a `defaults` object to the main object provided to our model definition:

```
ContactManager.Contact = Backbone.Model.extend({
  defaults: {
    firstName: ""
  }
});
```

If we now declare the following model instance

```
var contact = new ContactManager.Contact({
  lastName: "Arten",
  phoneNumber: "555-0184"
});
```

and we try to display the missing `firstName` attribute, the empty string we defined as the default value will be shown instead.

[30] https://github.com/davidsulc/marionette-gentle-introduction/commit/7e7e6f5a5c37ceea1b9419396464894e08bf7d23

Note that this code is included only to demonstrate default model attributes. It will not be part of our application's code: later on, we will add model validations to manage missing attribute values.

Introducing Events

Let's enrich our view slightly: we've got a phone number for Alice, so let's display it in an alert when her name is clicked.

Marionette views inherit all of Backbone's functionality, among which the ability to define events and their associated handlers. Here's what they look like:

```
events: {
  "click p": "alertPhoneNumber"
}
```

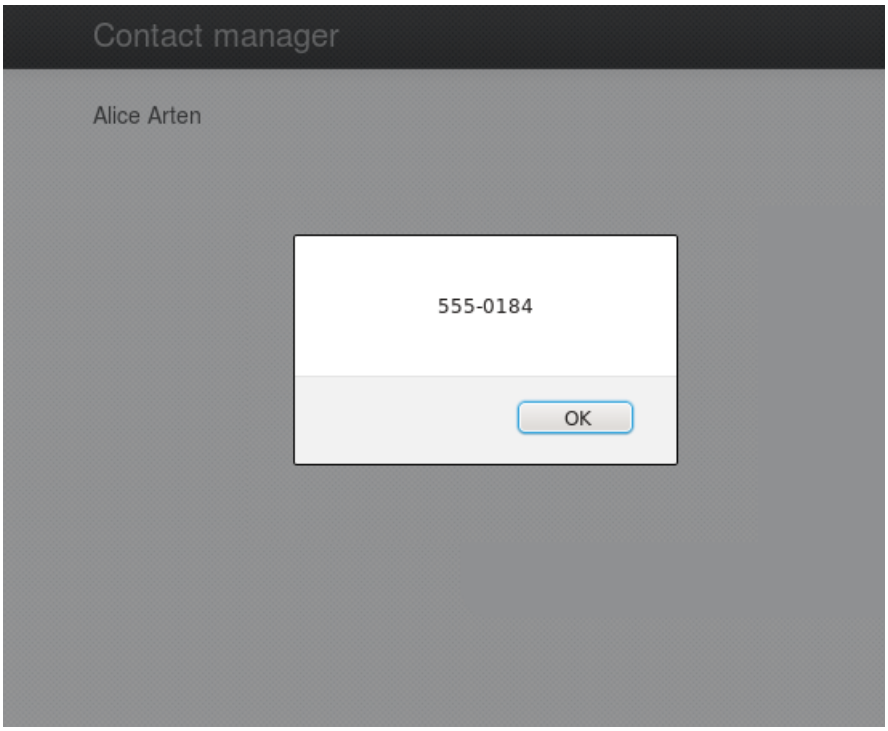
This event translates as "when the user `clicks` the `p` tag that can be found in this view, call the `alertPhoneNumber` function". If you've used jQuery, you'll recognize it's essentialy an event name followed by a selector (which could contain class names, etc.). Let's use this feature in our view to display Alice's phone number, by modifying our view declaration:

```
ContactManager.ContactView = Marionette.ItemView.extend({
    template: "#contact-template",

    events: {
      "click p": "alertPhoneNumber"
    },

    alertPhoneNumber: function(){
      alert(this.model.escape("phoneNumber"));
    }
  });
```

Backbone models' `escape`[31] works the same way as `get`[32]: they both return the value of the attribute provided as an argument, but `escape` will escape HTML content, protecting you from XSS attacks if you're displaying user-provided data within the HTML.

[31] http://backbonejs.org/#Model-escape
[32] http://backbonejs.org/#Model-get

If you now refresh the page and click on Alice's name, you'll see her phone number displayed. Pretty straightforward, right? You'll notice that since we're in the view definition when we're writing our `alertPhoneNumber` function, we have access to the view's model instance via `this.model`, even though *which* model instance will be used isn't known yet (it's provided when we instantiate the view, remember?).



Displaying an alert when clicking a contact

This code is not going to be included in our app, so you won't see it going forward.

Exercise

Displaying a Contact With No Phone Number

Add a default phone number of "No phone number!". Then create a new model without a phone number, and click on it. Make sure that "No phone number!" is displayed in the alert. You can see the exercise solution at the end of the book.

Displaying a Collection of Models

More often than not, we'll be dealing with several instances of a given model (e.g. a list of contacts). Backbone has built-in functionality for this purpose, named *collections* (you can learn more about them in Backbone's documentation[33]). These collections have many interesting features we'll look into later, but for now we'll focus on the functionality Marionette provides to display them.

Collections are very straightforward to define, for example:

```
1  var MyModel = Backbone.Model.extend({});
2
3  var MyCollection = Backbone.Collection.extend({
4    model: MyModel
5  });
```

As you can see, collections define which type of models they contain. Don't worry, we'll see a practical example in a few moments with our ContactManager app.

Introducing the CollectionView

Let's take a minute to think about what is required to display a list of multiple model instances. We'd need:

1. a collection to hold all the models
2. a mechanism to render the same view type for each model instance
3. somewhere to display all of these views

Fortunately, Marionette does all of this for us with a `CollectionView` that looks like this (from the documentation[34]):

[33] http://backbonejs.org/#Collection

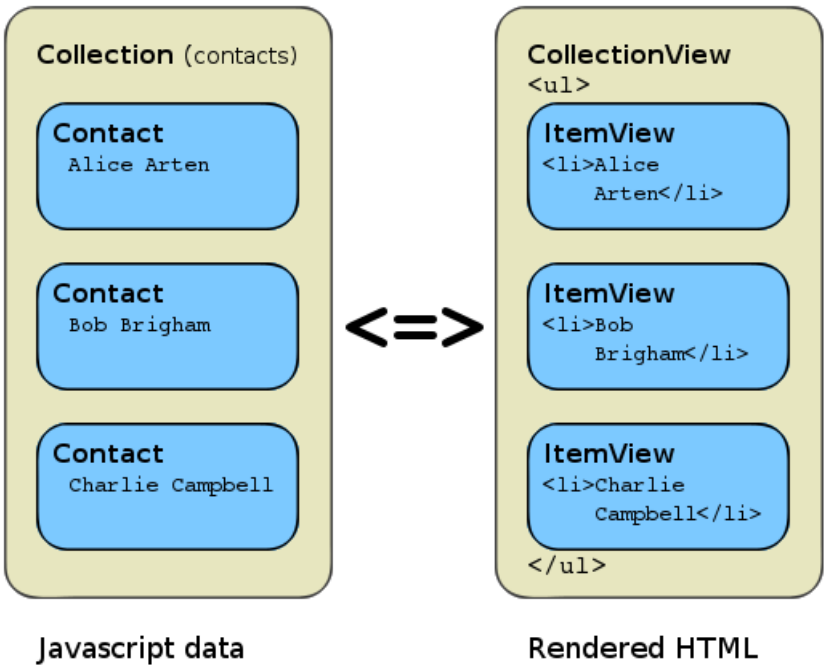
[34] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.collectionview.md

```
var MyItemView = Marionette.ItemView.extend({});

Marionette.CollectionView.extend({
  itemView: MyItemView
});
```

This code covers points 2 and 3 above: our `CollectionView` will render an instance of `MyItemView` for each model in the collection, and we can then show our `CollectionView` instance within our region to display all of these views at once. But where's our collection? As you can guess, it isn't defined anywhere as we'll most likely provide different collection configurations to our views: it could be a completely different list of models (e.g. a filtered list), or the same list sorted differently. Therefore, we'll simply pass the collection as an option when the view is instantiated, as we saw in the previous chapter.

Listing our Contacts with a CollectionView

So how do we implement this in our app? Let's display a collection of contacts as an unordered list (i.e. within a `ul` element). This is how our javascript data will be transformed into HTML, via a `CollectionView`:



Correspondence between javascript data and rendered HTML, using a CollectionView

First, we'll need a template and view to display each model:

```
<script type="text/template" id="contact-list-item">
  <li><%= firstName %> <%= lastName %></li>
</script>

ContactManager.ContactItemView = Marionette.ItemView.extend({
  template: "#contact-list-item"
});
```

Don't forget: templates go in the HTML section, while our views (being javascript) need to go within our application's `script` tag.

Now let's add a `CollectionView`:

```
ContactManager.ContactsView = Marionette.CollectionView.extend({
    tagName: "ul",
    itemView: ContactManager.ContactItemView
});
```

Why are we using the `tagName` attribute? It will make our view get wrapped within a `ul` element instead of the default `div`. Then, once we get our `li` elements rendered within the collection view (which is now a `ul` element), we'll have the list we want to be displayed.

We already have a contact model from last chapter, so let's create a collection:

```
ContactManager.ContactCollection = Backbone.Collection.extend({
  model: ContactManager.Contact
});
```

Now, all we need is to start everything up within our `initialize:after` handler:

```
1  ContactManager.on("initialize:after", function(){
2    var contacts = new ContactManager.ContactCollection([
3      {
4        firstName: "Bob",
5        lastName: "Brigham",
6        phoneNumber: "555-0163"
7      },
8      {
9        firstName: "Alice",
10       lastName: "Arten",
11       phoneNumber: "555-0184"
12     },
13     {
14       firstName: "Charlie",
15       lastName: "Campbell",
16       phoneNumber: "555-0129"
17     }
18   ]);
19
20   var contactsListView = new ContactManager.ContactsView({
21     collection: contacts
22   });
23
24   ContactManager.mainRegion.show(contactsListView);
25 });
```

It can be hard to see, but to create our collection instance we're providing an *array* of objects: note the `[]` characters on lines 2 and 18. The collection initializer will then create model instances for each element in the array.

Just to make sure, our code should now look like this:

index.html

```
1  <div id="main-region" class="container">
2    <p>Here is static content in the web page. You'll notice that it gets
3    replaced by our app as soon as we start it.</p>
4  </div>
5
6  <script type="text/template" id="contact-list-item">
7    <li><%= firstName %> <%= lastName %></li>
```

```
</script>

<!-- The javascript includes are here -->

<script type="text/javascript">
  var ContactManager = new Marionette.Application();

  ContactManager.addRegions({
    mainRegion: "#main-region"
  });

  ContactManager.Contact = Backbone.Model.extend({});

  ContactManager.ContactCollection = Backbone.Collection.extend({
    model: ContactManager.Contact
  });

  ContactManager.ContactItemView = Marionette.ItemView.extend({
    template: "#contact-list-item"
  });

  ContactManager.ContactsView = Marionette.CollectionView.extend({
    tagName: "ul",
    itemView: ContactManager.ContactItemView
  });

  ContactManager.on("initialize:after", function(){
    var contacts = new ContactManager.ContactCollection([
      {
        firstName: "Bob",
        lastName: "Brigham",
        phoneNumber: "555-0163"
      },
      {
        firstName: "Alice",
        lastName: "Arten",
        phoneNumber: "555-0184"
      },
      {
        firstName: "Charlie",
        lastName: "Campbell",
        phoneNumber: "555-0129"
```

```
50        }
51      ]);
52
53      var contactsListView = new ContactManager.ContactsView({
54        collection: contacts
55      });
56
57      ContactManager.mainRegion.show(contactsListView);
58    });
59
60    ContactManager.start();
61  </script>
```

Now, if we take a look at the result, it'll probably look ok. But that's only because modern browsers are really good at interpreting invalid/broken HTML markup. Let's inspect the source code to see what's been rendered:

```
<ul>
  <div>
    <li>Bob Brigham</li>
  </div>
  <div>
    <li>Alice Arten</li>
  </div>
  <div>
    <li>Charlie Campbell</li>
  </div>
</ul>
```

What's going on there? Well, if you recall, we've mentioned that Backbone will use a `div` to wrap views by default. Since we didn't specify a `tagName` for our item view, it was rendered within a `div`. But we don't want that extra tag, so what can we do? It's easy if you think about it: we want our contact's item view to be rendered within an `li` tag without any wrapping `div` tags, so we'll need to specify a `tagName` of `li`. But now that our `ItemView` will be using an `li` tag, there's no need for it in the template:

```
<script type="text/template" id="contact-list-item">
  <%= firstName %> <%= lastName %>
</script>

ContactManager.ContactItemView = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-list-item"
});
```

And now, when we refresh the page, we'll have our list properly rendered:

```
<ul>
  <li>Bob Brigham</li>
  <li>Alice Arten</li>
  <li>Charlie Campbell</li>
</ul>
```

Here's what our HTML now looks like:

Displaying a collection in an unordered list

Our corrected code now looks like this:

index.html

```
<div id="main-region" class="container">
  <p>Here is static content in the web page. You'll notice that it gets
  replaced by our app as soon as we start it.</p>
</div>

<script type="text/template" id="contact-list-item">
  <%= firstName %> <%= lastName %>
</script>

<!-- The javascript includes are here -->
```

```
11
12  <script type="text/javascript">
13    var ContactManager = new Marionette.Application();
14
15    ContactManager.addRegions({
16      mainRegion: "#main-region"
17    });
18
19    ContactManager.Contact = Backbone.Model.extend({});
20
21    ContactManager.ContactCollection = Backbone.Collection.extend({
22      model: ContactManager.Contact
23    });
24
25    ContactManager.ContactItemView = Marionette.ItemView.extend({
26      tagName: "li",
27      template: "#contact-list-item"
28    });
29
30    ContactManager.ContactsView = Marionette.CollectionView.extend({
31      tagName: "ul",
32      itemView: ContactManager.ContactItemView
33    });
34
35    ContactManager.on("initialize:after", function(){
36      var contacts = new ContactManager.ContactCollection([
37        {
38          firstName: "Bob",
39          lastName: "Brigham",
40          phoneNumber: "555-0163"
41        },
42        {
43          firstName: "Alice",
44          lastName: "Arten",
45          phoneNumber: "555-0184"
46        },
47        {
48          firstName: "Charlie",
49          lastName: "Campbell",
50          phoneNumber: "555-0129"
51        }
52      ]);
```

```
53
54      var contactsListView = new ContactManager.ContactsView({
55        collection: contacts
56      });
57
58      ContactManager.mainRegion.show(contactsListView);
59    });
60
61    ContactManager.start();
62  </script>
```

Git commit to display contacts within an unordered list:

f1c325d479d9b76f5f01fe0dcc64ab25f3fc8ff5[35]

Sorting a Collection

You're probably slightly annoyed our contacts aren't displayed in alphabetical order... If you're obsessed with sorting, you might have fixed that already by changing the order in which the models are created in the collection. But I'm sure you'll agree that's hardly a robust solution. Instead, let's have the collection do the hard work for us.

Backbone collections have an attribute called a comparator[36] which, when defined, will keep our collection in order. So let's tell our `contacts` collection to get itself sorted by `firstName`:

```
ContactManager.ContactCollection = Backbone.Collection.extend({
  model: ContactManager.Contact,

  comparator: "firstName"
});
```

Displaying our collection after implementing a comparator

[35] https://github.com/davidsulc/marionette-gentle-introduction/commit/f1c325d479d9b76f5f01fe0dcc64ab25f3fc8ff5

[36] http://backbonejs.org/#Collection-comparator

Git commit adding a comparator to our contacts collection:

34de82939366981a7db09c40995676f671c50f8b[37]

For more complex sorting needs, you can define functions to determine sorting order (see Backbone's documentation[38] and the solution to the exercise that follows).

Exercise

Sorting a Collection with a Function

Let's say we have the following collection:

```
1  var contacts = new ContactManager.ContactCollection([
2    {
3      firstName: "Alice",
4      lastName: "Tampen"
5    },
6    {
7      firstName: "Bob",
8      lastName: "Brigham"
9    },
10   {
11     firstName: "Alice",
12     lastName: "Artsy"
13   },
14   {
15     firstName: "Alice",
16     lastName: "Arten"
17   },
18   {
19     firstName: "Charlie",
20     lastName: "Campbell"
21   },
22   {
```

[37] https://github.com/davidsulc/marionette-gentle-introduction/commit/34de82939366981a7db09c40995676f671c50f8b

[38] http://backbonejs.org/#Collection-comparator

```
23      firstName: "Alice",
24      lastName: "Smith"
25    },
26  ]);
```

If you refresh the page, you'll see the contacts in the following order:

- Alice Tampen
- Alice Artsy
- Alice Arten
- Alice Smith
- Bob Brigham
- Charlie Campbell

What we'd like in this case, is to have them sorted by first name, then by last name in case of equality. Look at the documentation[39] and see if you can figure out how to define a comparator function that will display our collection in the following order:

- Alice Arten
- Alice Artsy
- Alice Smith
- Alice Tampen
- Bob Brigham
- Charlie Campbell

You can see the exercise's solutions at the end of the book.

[39] http://backbonejs.org/#Collection-comparator

Structuring Code with Modules

While building our app, we've simply been putting the code into `index.html` so far. While that's great for a small app, it won't scale as our code base grows. So let's refactor our code using Marionette modules[40]. The basic strategy we'll follow is this:

- one file for our general ContactManager app code (defining regions, the `initialize:after` handler, etc.): `assets/js/app.js`
- one module to manage our *entities* (i.e. models and collections), broken down into one file per type, e.g.: `assets/js/entities/contact.js`
- one module for each sub-application (e.g. the sub-application that will manage our contacts, or the one that will manage our header menu), with sub-modules for each "functional end result" (e.g. listing all contacts, or editing a single contact). For these sub-modules, we'll separate functionality into a `controller` and a `view`, giving us 2 files. Here's where we'd find the files to *list contacts*:
  - `assets/js/apps/contacts/list/list_controller.js`
  - `assets/js/apps/contacts/list/list_view.js`

This might be a bit much to take in at once, so I suggest you come back to this later if it doesn't quite make sense yet. It will be much easier to understand the concept in practice, so let's get started. If at any time you get overwhelmed by the architecture, you can refer to the diagram and explanation included at the end of the book.

Credit where credit is due: this structure is the one used by Brian Mann in his excellent screen casts at BackboneRails.com[41]

Extracting our App Definition

Let's start by defining the basics of our app in a separate file. So let's move this code

[40] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.application.module.md
[41] http://www.backbonerails.com/

Code to move from index.html to assets/js/app.js

```
var ContactManager = new Marionette.Application();

ContactManager.addRegions({
  mainRegion: "#main-region"
});
```

from `index.html` into `assets/js/app.js` (you'll need to create that file, since it doesn't exist yet). Naturally, we now need to include that javascript file in our `index.html` by adding it at the end of the other javascript includes:

```
<!-- The javascript includes are here -->

<script src="./assets/js/app.js"></script>
```

We'll refactor more as we go along, but this is a good start.

Moving Contacts to the Entities Module

Defining a Module

Now, we'll move our contacts to a separate module. That way, we can centralize their management, and when our app needs the contacts data, it can simply request them. Let's start by creating a new Marionette module[42]:

```
1 ContactManager.module("Entities", function(Entities, ContactManager,
2 Backbone, Marionette, $, _){
3 });
```

Lines 1 and 2 are actually a single line in our application, but are displayed across 2 lines for better readability. You will often come across this situation within the book, and you can keep the code on one line if you're following along.

Save this file as `assets/js/entities/contact.js` and include it within `index.html` by adding it at the end of the other javascript includes:

[42] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.application.module.md

```
<!-- The javascript includes are here -->

<script src="./assets/js/app.js"></script>
<script src="./assets/js/entities/contact.js"></script>
```

With the above module definition, we've defined a module on our `ContactManager` application. If you refer to the documentation[43], you'll see that module definitions can be given a callback that takes 6 arguments:

1. the module itself (i.e. what name we're going to use within the callback to refer to the module we're defining)
2. the application object that `module` was called from
3. Backbone
4. Backbone.Marionette
5. jQuery
6. Underscore

Accessibility within Modules

Still quoting the documentation[44]:

> You can add functions and data directly to your module to make them publicly accessible. You can also add private functions and data by using locally scoped variables.

Consider the following code:

```
ContactManager.module("Entities", function(Entities, ContactManager,
Backbone, Marionette, $, _){
  var alertPrivate = function(message){
    alert("Private alert: " + message);
  };

  Entities.alertPublic = function(message){
    alert("I will now call alertPrivate");
    alertPrivate(message);
  };
});
```

If you include that code (after the `app.js` file created above, of course), you could do the following:

[43] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.application.module.md#module-definitions
[44] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.application.module.md#module-definitions

```
ContactManager.Entities.alertPublic("Hello");
```

And the following alerts would be displayed sequentially:

1. I will now call alertPrivate
2. Private alert: Hello

In other words, you have access to the `alertPrivate` function from within the module callback. However, as soon as you leave that scope, `alertPrivate` can no longer be called: it is effectively a private method within the callback.

Moving the Contact Entities

Now that we have a better understanding of modules, let's move our contacts from `index.html` into their module:

assets/js/entities/contact.js

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    Entities.Contact = Backbone.Model.extend({});
4
5    Entities.ContactCollection = Backbone.Collection.extend({
6      model: Entities.Contact,
7      comparator: "firstName"
8    });
9  });
```

Notice that we've attached our model and collection as attributes to the *module*, and not the *application* as was the case before. In other words, we've gone from (e.g.) `ContactManager.Contact` to `Entities.Contact`. And, as explained above, since we're attaching them to the module, they will be publicly accessible (meaning we can access them from elsewhere in the application, if necessary).

Since we've changed where our contact definitions are, we need to adapt our `index.html`:

```
var contacts = new ContactManager.Entities.ContactCollection([
  // our contact data goes here
]);
```

Note we've simply changed `ContactManager.ContactCollection` to

`ContactManager.Entities.ContactCollection`

since the `ContactCollection` is now attached to the `Entities` module, and no longer directly to the `ContactManager` app.

If you refresh our page, everything will still be working properly: even though we're now using a module to house our contacts, the application behaves as before.

Getting our Contacts by Request

Let's now improve our `contact` entity file, by using Marionette.RequestResponse[45]. According to the documentation:

> This allows components in an application to request some information or work be done by another part of the app, but without having to be explicitly coupled to the component that is performing the work.

This will allow our app to request the contacts, and they will be provided by our module. But first, we need some code to initialize our contacts collection if it doesn't exist:

assets/js/entities/contact.js

```
ContactManager.module("Entities", function(Entities, ContactManager,
Backbone, Marionette, $, _){

  // model and collection definitions are here

  var contacts;

  var initializeContacts = function(){
    contacts = new Entities.ContactCollection([
      { id: 1, firstName: "Alice", lastName: "Arten",
        phoneNumber: "555-0184" },
      { id: 2, firstName: "Bob", lastName: "Brigham",
```

[45] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.requestresponse.md

```
13          phoneNumber: "555-0163" },
14        { id: 3, firstName: "Charlie", lastName: "Campbell",
15          phoneNumber: "555-0129" }
16      ]);
17    };
18
19    var API = {
20      getContactEntities: function(){
21        if(contacts === undefined){
22          initializeContacts();
23        }
24        return contacts;
25      }
26    };
27  });
```

At this time, we're not dealing with any persistent storage (server or client side), which will come later. Ignoring persistence for now will let you better understand how to manage data in memory, and will demonstrate certain concepts introduced above, such as data accessibility.

Note that we have added `id` attributes to our models, because we're going to need an attribute to uniquely identify our models (within our application) whether or not persistence is implemented.

Remember how we discussed public versus private data within the module definition callback? You can see it at work here: the `contacts` collection is stored within the variable declared on line 6. This variable will remain private, as it wasn't attached to the `Entities` module.

On lines 8-17, we define an `initializeContacts` function that will allow us to create some contacts if necessary. Once again, this function hasn't been attached to the model, so it will remain private and it will not be possible to call it from elsewhere in the application. As a side note, in a typical client-server configuration, you wouldn't create this type of data on the client (since you'd fetch it from the server). Nonetheless, this type of data initialization can be handy when dealing with data that is relevant to the user but doesn't need to be stored (or retrieved) server-side.

As a rule, try to expose as little as possible via public functions and attributes: this will lead to cleaner code and avoid coupling (by forcing data interactions to transit by defined interfaces).

On lines 19-26, we define an `API` object to contain the functions we will allow the rest of the application to use. In particular, notice that the `getContactEntities` function declared on line 17 can

return the `contacts` variable, because it's declared within the module definition's callback function. Outside of this module definition, the `contacts` variable won't be readable, since it will be out of scope. In addition, note that the `getContactEntities` function isn't technically *public* because it's not attached to the `Entities` module. This is by design, since we want to force the rest of the application to get the contacts by using a request.

How do requests work? Anywhere in our application, we can call

```
ContactManager.request("contact:entities");
```

and expect to get our `contacts` collection as the return value.

How does our application know what to do? We need to register a *request handler* to call when the `contact:entities` request is received. Let's add it to the bottom of our contact module:

assets/js/entities/contact.js

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    Entities.Contact = Backbone.Model.extend({});
4
5    Entities.ContactCollection = Backbone.Collection.extend({
6      model: Entities.Contact,
7      comparator: "firstName"
8    });
9
10   var contacts;
11
12   var initializeContacts = function(){
13     contacts = new Entities.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   };
22
23   var API = {
24     getContactEntities: function(){
25       if(contacts === undefined){
26         initializeContacts();
```

```
27        }
28        return contacts;
29      }
30    };
31
32    ContactManager.reqres.setHandler("contact:entities", function(){
33      return API.getContactEntities();
34    });
35  });
```

What is this `reqres` attribute on line 32? It's the "request-response" system, and it's automatically defined at the application level by Marionette.

As our new module is now functional, let's change our `initialize:after` handler in `index.html` to request the contacts:

```
ContactManager.on("initialize:after", function(){
  var contacts = ContactManager.request("contact:entities");

  // rest of the handler code
});
```

This is what our `index.html` looks like now:

index.html

```
<!-- The javascript library includes are here -->

<script src="./assets/js/app.js"></script>
<script src="./assets/js/entities/contact.js"></script>

<script type="text/javascript">
  ContactManager.ContactItemView = Marionette.ItemView.extend({
    tagName: "li",
    template: "#contact-list-item"
  });

  ContactManager.ContactsView = Marionette.CollectionView.extend({
    tagName: "ul",
    itemView: ContactManager.ContactItemView
```

```
15    });
16
17    ContactManager.on("initialize:after", function(){
18      var contacts = ContactManager.request("contact:entities");
19
20      var contactsListView = new ContactManager.ContactsView({
21        collection: contacts
22      });
23
24      ContactManager.mainRegion.show(contactsListView);
25    });
26
27    ContactManager.start();
28  </script>
```

It's starting to look a lot cleaner, isn't it? But we're not done yet...

Creating a Module for the ContactsApp Sub-Application

So far, we've created a separate module for our entities, and we've moved our contact model and collection there. If you think about it, the entities are general concepts that will be used throughout the ContactManager app. But our main ContactManager app could contain several sub-applications, such as:

- an app to manage contact information (create new contacts, edit their information, etc.);
- an app to manage contact groups (friends, coworkers, etc.) and which contacts belong to them;
- an app giving information about the ContactManager app (copyright info, compatibility, etc.);
- and more...

We're going to create a module to contain our ContactsApp, which will in turn contain a `ContactsApp.List` sub-module to list our contacts. This `List` sub-module will contain our controller to manage listing the contacts, and it will also contain the required views. Our structure will then look like so:

- ContactManager
  - ContactsApp
    - List
      - Controller
      - *various views*

Moving our Views

Let's declare our `ContactsApp.List` sub-module so we can move our views from `index.html` into this new file:

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

```
1  ContactManager.module("ContactsApp.List", function(List, ContactManager,
2  Backbone, Marionette, $, _){
3      List.Contact = Marionette.ItemView.extend({
4        tagName: "li",
5        template: "#contact-list-item"
6      });
7
8      List.Contacts = Marionette.CollectionView.extend({
9        tagName: "ul",
10       itemView: List.Contact
11     });
12 });
```

We've renamed our `ContactsView` to `Contacts`, and `ContactItemView` to `Contact`. These names will enable us to maintain the same naming conventions across modules, but weren't available before: `ContactManager.Contact` was our model definition. Now that view and entities are defined in separate modules, we no longer have naming conflicts and can pick the appropriate names.

With these new names in place, we've had to adapt our code on lines 3, 8, and 10 to use them.

Once again, the views are no longer attached to the ContactManager application, so we need to adapt our code on lines 3, 8, and 10, by replacing `ContactManager` with `List` since that's what we're calling our sub-module within the module definition callback (on line 1).

We've declared a `ContactsApp.List` sub-module, but we've never declared the `ContactsApp` module. Then why does this code work? As explained in the documentation[46], "When defining sub-modules using the dot-notation, the parent modules do not need to exist. They will be created for you if they don't exist."

Now, we need to adapt our `index.html`:

[46] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.application.module.md#defining-sub-modules-with--notation

index.html

```
1  <script src="./assets/js/app.js"></script>
2  <script src="./assets/js/entities/contact.js"></script>
3  <script src="./assets/js/apps/contacts/list/list_view.js"></script>
4
5  <script type="text/javascript">
6    ContactManager.on("initialize:after", function(){
7      var contacts = ContactManager.request("contact:entities");
8
9      var contactsListView = new ContactManager.ContactsApp.List.Contacts({
10       collection: contacts
11     });
12
13     ContactManager.mainRegion.show(contactsListView);
14   });
15
16   ContactManager.start();
17 </script>
```

Once again, don't forget to include this new file on line 3. Then, we need to adapt our view instantiation code on line 9, as the view is no longer directly attached to ContactManager.

So far, so good: our app is still working, and our code keeps looking better!

Creating a Controller

What is our `List` controller going to do? It's basically the application equivalent of an orchestra conductor: it coordinates the various pieces (typically models/collections and views), and gets them to produce a coherent result (i.e. a displayed page). Later on, when we add routing to our application, typing in URLs will fire various controller actions.

Here's our controller, again in a new file:

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 contacts = ContactManager.request("contact:entities");
6
7          var contactsListView = new List.Contacts({
8            collection: contacts
9          });
10
11         ContactManager.mainRegion.show(contactsListView);
12       }
13     }
14 });
```

The first thing to notice, is that we're defining this code within a `ContactsApp.List` sub-module. Therefore, on line 7 we can refer to our views (which are also defined within a `ContactsApp.List` sub-module) with `List.Contacts`.

We've already declared a sub-module named `ContactsApp.List` within our `list_view.js` file, so how come we can declare it again? And not only that, but they must be the same module, since we can refer to a view with `List.Contacts`. That's because Marionette is helping us out: "If [the sub-module exists], [it] will be used instead of creating a new one" (see the documentation[47]). This conveniently allows us to have our controller and views defined in separate files, while keeping them within the same Marionette sub-module.

We've created a `Controller` object attached to our sub-module on line 3, where we'll put all the functions we intend to be publicly available (such as the `listContacts` function on line 4). These public methods will typically be the ones that are triggered by entering URLs into the address bar.

Let's see how these changes impact our index page:

[47] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.application.module.md#defining-sub-modules-with--notation

index.html

```
<script src="./assets/js/app.js"></script>
<script src="./assets/js/entities/contact.js"></script>
<script src="./assets/js/apps/contacts/list/list_view.js"></script>
<script src="./assets/js/apps/contacts/list/list_controller.js"></script>

<script type="text/javascript">
  ContactManager.on("initialize:after", function(){
    ContactManager.ContactsApp.List.Controller.listContacts();
  });

  ContactManager.start();
</script>
```

As you've guessed, we've included the new file on line 4, then we simply call our controller's action on line 8.

Moving the App Initialization Handler

There's not much left in our index page, but let's finish cleaning up by moving our `intialize:after` handler code to our application file:

Code to move from index.html to assets/js/app.js

```
ContactManager.on("initialize:after", function(){
  ContactManager.ContactsApp.List.Controller.listContacts();
});
```

Our `app.js` file now contains the following code:

assets/js/app.js

```
var ContactManager = new Marionette.Application();

ContactManager.addRegions({
  mainRegion: "#main-region"
});

ContactManager.on("initialize:after", function(){
  ContactManager.ContactsApp.List.Controller.listContacts();
});
```

Now, we've successfully refactored our app into nice little components, and the only things left in our `index.html` are the file includes, the templates, and the code to call the `start` method on our application:

```
<script type="text/template" id="contact-list-item">
  <%= firstName %> <%= lastName %>
</script>

<!-- The javascript library includes are here -->

<!-- Our javascript application includes are here -->

<script type="text/javascript">
  ContactManager.start();
</script>
```

Note that since we're starting our app after all javascript files have been loaded, and we're not modifying the DOM in any way, we don't need to wrap our starter code in a `$(document).ready()` call (which would slow down startup). If you application requires waiting until the DOM is ready before starting, don't forget to wrap the call to `start` in a `$(document).ready()`.

Git commit refactoring the application into modules:

4ec383cbe63a62406e6fa0a86bfa8f15093a9674[48]

[48] https://github.com/davidsulc/marionette-gentle-introduction/commit/4ec383cbe63a62406e6fa0a86bfa8f15093a9674

Exercise

Declaring a Template Sub-Module

We want to access a `listItemView` template from within a dedicated sub-module with

```
ContactManager.ContactsApp.List.Templates.listItemView
```

What should our code look like? You can see the exercise solution at the end of the book.

Dealing with Templates

Although we've refactored our application, our templates haven't moved. How come? Well, mainly because managing templates can easily be accomplished on the server, from the quick and dirty technique of including a file containing the templates, to more elegant solutions involving compiled templates (such as using RequireJS[49] or sprockets[50] with Rails' asset pipeline). If you'd like to learn more about loading compiled templates from separate files using RequireJS, take a look at my book[51] on using RequireJS with Marionette.

For the sake of education, we'll cover overriding Marionette's template loader to demonstrate how we could organize our templates differently. However, this won't be part of the main text (or app development): refer to the "Overriding Marionette's Template Loader" chapter at the end of the book.

[49] http://requirejs.org/

[50] https://github.com/sstephenson/sprockets#javascript-templating-with-ejs-and-eco

[51] https://leanpub.com/structuring-backbone-with-requirejs-and-marionette

Using a CompositeView

Now that we've done some house cleaning in our code base, let's display our contacts within a table. Easy, right? We already have our contacts in a collection, so we simply need to render a `tr` DOM element for each one of them.

Let's modify the contact template to make it generate the contents for a table row:

Modifying the contact template in index.html

```
1 <script type="text/template" id="contact-list-item">
2   <td><%= firstName %></td><td><%= lastName %></td>
3 </script>
```

We now need our table row to be wrapped within a `tr` tag, so let's modify line `2` in the `Contact` view:

Modifying the `Contact` view in assets/js/apps/contacts/list/list_view.js

```
1 List.Contact = Marionette.ItemView.extend({
2   tagName: "tr",
3   template: "#contact-list-item",
4
5   // event-handling code is here
6 });
```

And last, but not least, we need our `Contacts` view to be contained within a table, instead of the current `ul` tag. Let's edit line `2` to change the tag:

Modifying the `Contacts` view in assets/js/apps/contacts/list/list_view.js

```
1 List.Contacts = Marionette.CollectionView.extend({
2   tagName: "table",
3   className: "table table-hover",
4   itemView: List.Contact
5 });
```

In the code above, we've also taken the opportunity to add some styling classes to our table on line 3, so that Bootstrap will style it for us (see documentation[52]).

Refresh the page, and our contacts are now displayed in our `table`. Let's now add "First Name" and "Last Name" column headers. How are we going to get our `CollectionView` to do this? Simple answer: it can't, because it wasn't designed for this. Loosly quoting the documentation[53], the `CollectionView` loops over a collection of models, renders each one with the provided `itemView` attribute, and adds all of those rendered views to the DOM element used by the `CollectionView`. Put another way, there's no means to specify a template organizing content.

Happily, Marionette won't leave us high and dry: the `CompositeView`[54] can "be used for scenarios where a collection needs to be rendered within a wrapper template". So let's create the template we'll use for our new `CompositeView`:

Table template to add to index.html

```
1 <script type="text/template" id="contact-list">
2   <thead>
3     <tr><th>First Name</th><th>Last Name</th></tr>
4   </thead>
5   <tbody>
6   </tbody>
7 </script>
```

We've already altered our `Contact` item view to be a table row, so now we can move on to modifying our `Contacts` view to become a `CompositeView`:

[52] http://twitter.github.io/bootstrap/base-css.html#tables

[53] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.collectionview.md

[54] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.compositeview.md

Modifying the `Contacts` view in assets/js/apps/contacts/list/list_view.js

```
1 List.Contacts = Marionette.CompositeView.extend({
2   tagName: "table",
3   className: "table table-hover",
4   template: "#contact-list",
5   itemView: List.Contact,
6   itemViewContainer: "tbody"
7 });
```

There are 3 main things you need to notice here:

- on line 1, we've changed our definition to inherit from `Marionette.CompositeView` instead of `CollectionView`;
- on line 4, we've specified the template our `CompositeView` should use;
- on line 6, we've told the `CompositeView` to render the child views within the `tbody` element.

Let's expand on that last point: a `CompositeView` can be seen as a more powerful `CollectionView` with a `template` attribute.[55] Since it extends from a `CollectionView`, the `CompositeView` will by default simply append the rendered child views to its own DOM element. But we want the rendered child views to go inside the `tbody` tag, which we indicate via the `itemViewContainer` attribute on line 6.

For more complex scenarios where you need more control on where/how to insert the rendered child views, you can override the `appendHtml`[56] function in your `CompositeView`.

Our `Contacts` view, being a `CompositeView`, naturally requires a collection to display. Luckily, since it used to be a `CollectionView`, we're already providing it with a collection on instantiation:

The code instantiating our `Contacts` view in assets/js/apps/contacts/list/list_controller.js

```
1 var contactsListView = new List.Contacts({
2   collection: contacts
3 });
```

[55] It is much more than that, but this is a good description of what we're going to use it for in this case.

[56] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.compositeview.md#compositeviews-appendhtml

There we go, our app now displays our contacts within a table, complete with headers!

Displaying our contacts within a table

Git commit displaying our contacts with a CompositeView:

86c2f5b497afe0236373666b863a7c73275350b5[57]

Exercise

Building your own CompositeView

Now that we've seen how to use a `CompositeView`, try to build one on your own to generate the following end result:

```
<div>
  <p>Here is the list of all the contacts we have information for:</p>
  <ul>
    <li>Alice Arten</li>
    <li>Bob Brigham</li>
    <li>Charlie Campbell</li>
  </ul>
</div>
```

You can see the exercise solution at the end of the book.

Using Events

As we saw earlier, we can define events and their associated handlers within views. Let's use this ability to toggle highlighting on the rows that get clicked, by toggling Bootsrap's "warning" class to the appropriate `tr` element:

[57] https://github.com/davidsulc/marionette-gentle-introduction/commit/86c2f5b497afe0236373666b863a7c73275350b5

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

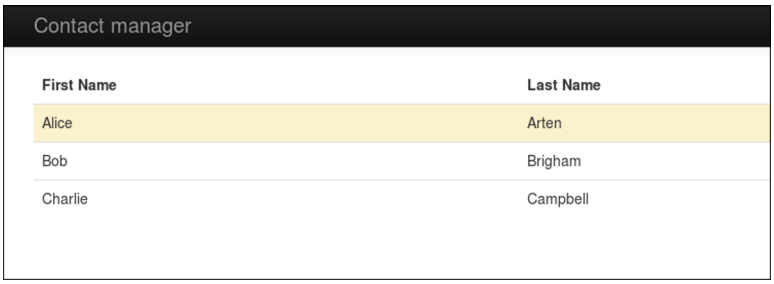
```
1  List.Contact = Marionette.ItemView.extend({
2    tagName: "tr",
3    template: "#contact-list-item",
4
5    events: {
6      "click": "highlightName"
7    },
8
9    highlightName: function(){
10     this.$el.toggleClass("warning");
11   }
12 });
```

The events object on lines 5-7 associates jQuery event listeners to handler functions. So in the above code on line 6, we're saying "when this view's DOM element is clicked, call function `highlightName`".

Each view instance has an `el` attribute[58] referencing the rendered DOM element. So within a view definition, using `this.el` will return the DOM element containing the view. And a Backbone view *also* has an `$el` attribute[59] which is different (and explained below).

Let's look at line 10: each view has an `$el` attribute returning a jQuery object wrapping the view's DOM element. In other words, `this.$el` is equivalent to `$(this.el)` but it's already conveniently ready for you to use. So in order to toggle the "warning" class on our `tr` containing the clicked item view, we simply call jQuery's `toggleClass`[60] method on the view's jQuery object.

Here's our table with a highlighted row:



Highlighting the row that was clicked

[58] http://backbonejs.org/#View-el
[59] http://backbonejs.org/#View-\protect\char"0024\relaxel
[60] http://api.jquery.com/toggleClass/

Git commit to toggle highlighting on the clicked row:

34ff4cb8375aa3a9ceb1f38be7c587cb06c6eaab[61]

Accessing the Event Object

Sometimes, we need to access the jQuery event object[62] that triggered the call to the handler function. This is typically the case with links: we want to have links pointing to certain URLs within our application, but we want to prevent the links from causing a page refresh (which is the default behavior). As you probably know, we can achieve this by calling `preventDefault`[63] on the `click` event object:

Demonstrating access to the Event Object

```
List.Contact = Marionette.ItemView.extend({
  tagName: "tr",
  template: "#contact-list-item",

  events: {
    "click": "highlightName"
  },

  highlightName: function(e){
    e.preventDefault();
    this.$el.toggleClass("warning");
  }
});
```

As you can see, the handler function will be provided the event object as an argument. So as long as you put it in the function signature, you'll have full access to it within the function's body.

Note that in our case, preventing the default action doesn't change any behavior. In addition, we won't be including this line in our application's code.

[61] https://github.com/davidsulc/marionette-gentle-introduction/commit/34ff4cb8375aa3a9ceb1f38be7c587cb06c6eaab

[62] http://api.jquery.com/category/event-object/

[63] http://api.jquery.com/event.preventDefault/

Exercise

Displaying the Contents of a Clicked Table Cell

Write an event handler that is triggered when the user clicks a `td` element. Make this handler display an alert containing the text within the `td` element.

Here are a few tips to get you going in the right direction:

- We discussed how to specify event selectors here
- You can retrieve which DOM element initiated an event with `e.target`[64]
- You can make DOM elements into jQuery objects by passing them to the `$()` function
- You can retrieve a jQuery object's text content with `.text()`[65]

You can see the exercise solution at the end of the book.

[64] http://api.jquery.com/event.target/
[65] http://api.jquery.com/text/

Events, Bubbling, and TriggerMethod

Now that we've got our contacts listed as we want them, let's add a button to delete a contact:

Modifying the contact template in index.html

```
1  <script type="text/template" id="contact-list-item">
2    <td><%= firstName %></td>
3    <td><%= lastName %></td>
4    <td>
5      <button class="btn btn-small">
6        <i class="icon-remove"></i>
7        Delete
8      </button>
9    </td>
10 </script>
```

Lines 5-8 will generate a "delete" button styled[66] with Bootstrap. We've also added an icon[67] to our button on line 6, courtesy of Bootstrap.

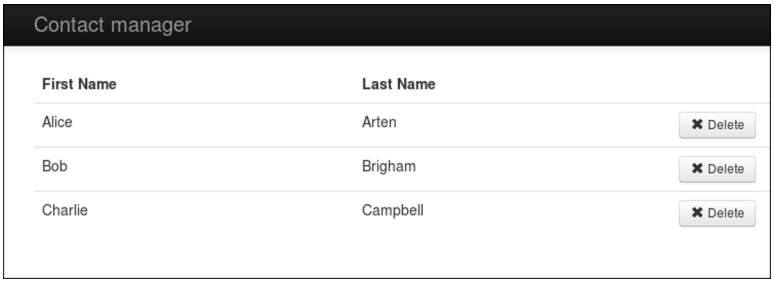
Since we've added a column to our item view, let's keep things in sync by adding a column to the table template (line 6):

Modifying the table template in index.html

```
1  <script type="text/template" id="contact-list">
2    <thead>
3      <tr>
4        <th>First Name</th>
5        <th>Last Name</th>
6        <th></th>
7      </tr>
8    </thead>
9    <tbody>
10   </tbody>
11 </script>
```

[66] http://twitter.github.io/bootstrap/base-css.html#buttons

[67] http://twitter.github.io/bootstrap/base-css.html#icons



Displaying a delete button

Take a look at our page, and you'll see our "delete" buttons. But if you click one of them, nothing happens (except the row getting highlighted). So let's add an event listener to our `Contact` view on line 7:

Modifying the `Contact` view in assets/js/apps/contacts/list/list_view.js

```
1  List.Contact = Marionette.ItemView.extend({
2    tagName: "tr",
3    template: "#contact-list-item",
4
5    events: {
6      "click": "highlightName",
7      "click button": function(){ alert("delete button was clicked"); }
8    },
9
10   highlightName: function(e){
11     this.$el.toggleClass("warning");
12   }
13 });
```

Our alert is properly displayed when we click a delete button, but its row was highlighted as an unwanted side-effect. The reason this happens is that the "delete" button element we clicked is located within the DOM's `tr` element: when we click *the button*, we therefore also click *the row*. And since we defined a listener for `click` events on the `tr` element (line 6, above), the corresponding handler is called.

To prevent our row from being highlighted when we click the "delete" button, we'll simply "hide" the click event from parent DOM elements with jQuery's stopPropagation[68] on line 15:

[68] http://api.jquery.com/event.stopPropagation/

Modifying the Contact view in assets/js/apps/contacts/list/list_view.js

```
List.Contact = Marionette.ItemView.extend({
  tagName: "tr",
  template: "#contact-list-item",

  events: {
    "click": "highlightName",
    "click button": "deleteClicked"
  },

  highlightName: function(e){
    this.$el.toggleClass("warning");
  },

  deleteClicked: function(e){
    e.stopPropagation();
    alert("delete button was clicked");
  }
});
```

When we now click our button, jQuery stops the event from propagating: the parent DOM elements never get notified about the `click` event. Mission accomplished!

What if I only want to *limit* propagation, not *stop* it?

The basic answer is: you won't. Instead, you can use a pub/sub mechanism so the parts of your app that need to respond will be notified when appropriate. But let's not get ahead of ourselves, we'll cover that later.

Although our event handling works when clicking the "delete" button, we can do much better. As it is, the event will trigger for *any* `button` element within our view: not really the behavior we want. So let's add a CSS class to make our selector more specific:

Adding a class to the contact template in index.html

```
<script type="text/template" id="contact-list-item">
  <td><%= firstName %></td>
  <td><%= lastName %></td>
  <td>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

Using the CSS selector

```
List.Contact = Marionette.ItemView.extend({
  // tagName and template attributes

  events: {
    "click": "highlightName",
    "click button.js-delete": "deleteClicked"
  },

  // event handler functions
});
```

You'll notice I used a CSS class prefixed by "`js-`": it allows me to differentiate between the CSS classes used for styling, and those used as selectors by the javascript functionality. This is particularly useful if you have designers and coders working on the same app, as it prevents app breakage: designers removing a class they no longer need won't break javascript functionlity, and coders doing away with unnecessary classes won't break the page display.

Now that we're pleased with our event selection, we can delete our contact when the button gets clicked:

Deleting the model from the `Contact` view (assets/js/apps/contacts/list/list_view.js)

```
1  List.Contact = Marionette.ItemView.extend({
2    // tagName and template attributes
3
4    events: {
5      "click": "highlightName",
6      "click button.js-delete": "deleteClicked"
7    },
8
9    // highlightName handler
10
11   deleteClicked: function(e){
12     e.stopPropagation();
13     this.model.collection.remove(this.model);
14   }
15 });
```

Since we haven't yet gotten around to implementing data persistence, our collection (and the models it contains) exists only in memory. So "deleting" a model is simply a matter of removing it from the collection on line 13.

Remember the scope we're in: an `ItemView` instance. So within the view, we only have access to the model that was provided to the view at instantiation. Although we don't have direct access to the `contacts` collection, each model keeps a reference to its parent collection within the `collection` attribute. Therefore, we can access our `contacts` collection from within our `Contact` view instance with `this.model.collection`.

Once we've got a reference to the collection, all that's left to do is call the `remove`[69] function, providing the view's model reference, and Marionette does the rest for us (closing the item view, unbinding event listeners to avoid zombies[70], etc.)[71]. Done!

Well, yes and no: it's functional, but it's not quite clean. We're impacting the application's data by deleting a contact, and processing data is the controller's job: views just display things. So let's delete our model the right way, by delegating that responsibility to our `List.Controller` object.

What about the highlightName handler?

We can leave that function in the view: it simply changes the display without affecting data, which is precisely the view's role.

[69] http://backbonejs.org/#Collection-remove

[70] http://lostechies.com/derickbailey/2011/09/15/zombies-run-managing-page-transitions-in-backbone-apps/

[71] This type of built-in management is exactly what makes Marionette so nice to work with. In plain Backbone, you'd need to manage these actions on your own.

Communicating via Events

Let's trigger an event indicating a contact should be deleted instead of deleting it directly in the view:

Triggering an event from the `Contact` view (assets/js/apps/contacts/list/list_view.js)

```
1  List.Contact = Marionette.ItemView.extend({
2    // tagName and template attributes
3
4    events: {
5      "click": "highlightName",
6      "click button.js-delete": "deleteClicked"
7    },
8
9    // highlightName handler
10
11   deleteClicked: function(e){
12     e.stopPropagation();
13     this.trigger("contact:delete", this.model);
14   }
15 });
```

The view's `trigger` method on line 13 is given an event name to trigger, and an argument to pass on. Next, we need to process this event within our controller:

Deleting a contact in the controller (assets/js/apps/contacts/list/list_controller.js)

```
1  List.Controller = {
2    listContacts: function(){
3      var contacts = ContactManager.request("contact:entities");
4
5      var contactsListView = new List.Contacts({
6        collection: contacts
7      });
8
9      contactsListView.on("itemview:contact:delete", function(childView,
10                                                              model){
11       contacts.remove(model);
12     });
```

```
13
14      ContactManager.mainRegion.show(contactsListView);
15    }
16  }
```

Once again, it's important to remember scopes when dealing with events:

- we triggered the "contact:delete" event from our `Contact` view instance, which is an `ItemView`
- we added the listener on `contactsListView`, which is a `CompositeView` containing multiple item views

Documentation[72] time:

> When an item view within a collection view triggers an event, that event will bubble up through the parent collection view with "`itemview:`" prepended to the event name.

Convention dictates that events are sectioned using ":" as a seperator. Also, it's worth noting that events aren't limited to 2 sections and that we could trigger an event named "my:super:long:event:name". Respecting this convention will be quite useful later on when we want to leverage triggerMethod as explained below.

Since `CompositeView` extends `CollectionView`, the above holds true. So when we listen for child view events at the composite view level, we need to prefix the triggered event name with "`itemview:`". Let's take a closer look at our event listener on lines 9-12: we register a listener for the event triggered by the child view (with the necessary "`itemview:`" prefix), and provide a callback function.

The callback function receives a reference to the child view that triggered the event, followed by the arguments that were provided when the event was triggered. In our case, we sent the view's model along so we need to add an argument in order to access the model within the callback. You'll notice that now, removing the model from the collection is a lot more readable: we already have a reference to the collection, so on line 11 we can simply call its `remove` method as above.

Animating the Removed ItemView

When a model is removed from the collection, its item view is automatically closed and removed from the DOM for us. Instead of having it disappear suddenly, let's have it gently fade out by animating it with jQuery. To achieve this, we simply need to leverage Marionette's lifecycle events. Here's how we do it:

[72] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.collectionview.md#itemview-event-bubbling-from-child-views

Animating removed child item views (assets/js/apps/contacts/list/list_view.js)

```
List.Contact = Marionette.ItemView.extend({
  // same code as before

  remove: function(){
    this.$el.fadeOut();
  }
});
```

As you can tell, this is pretty easy: Marionette calls an item view's `remove` method (if it's defined) when the corresponding model is removed from the collection referenced by the collection/composite view. If we refresh the page and delete a contact, we'll see it fade out of view. But when we inspect the HTML source, we can see the DOM element is still there, only *hidden*. To fix this, we need to remove the DOM element once it's done fading out, by calling Marionette's `remove` function for item views. Let's add that in a callback provided to `fadeOut`:

Animating removed child item views (assets/js/apps/contacts/list/list_view.js)

```
List.Contact = Marionette.ItemView.extend({
  // same code as before

  remove: function(){
    var self = this;
    this.$el.fadeOut(function(){
      Marionette.ItemView.prototype.remove.call(self);
    });
  }
});
```

On line 7, we're calling the `remove` function defined on Marionette's `ItemView` "class". Here's what that line breaks down to:

1. Get the `ItemView` "class" definition with `Marionette.ItemView.prototype` (more on that here[73]);
2. Refer to the `remove` function defined on the "class";
3. Call[74] the `remove` method, telling it to use the value in `self` when refering to `this` within the `remove` function's definition.

Put another way, line 7 basically calls the original `remove` function as if we hadn't redefined it. In effect, we're calling the original implementation by accessing the `ItemView`'s prototype, so the prototype chain will be rewound until a `remove` function definition is found in `Backbone.View`. Why go through all this trouble? Because this way the original `remove` code will clean up as necessary (e.g. stop listening to events, so the view instance can get garbage collected).

Git commit implementing removing a contact from the collection:

0e6f64e295b9f475baa1ee3991fa712c678df96a[75]

Exercise

Event Bubbling from Child Views

Modify the `highlightName` handler in the `List.Contact` view so that when a row is clicked, the *controller* prints the model data to the console with

```
console.log("Highlighting toggled on model: ", myModelInstance);
```

You can see the exercise solution at the end of the book.

[73] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain

[74] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call

[75] https://github.com/davidsulc/marionette-gentle-introduction/commit/0e6f64e295b9f475baa1ee3991fa712c678df96a

Introducing TriggerMethod

Let's see some more Marionette magic in action. When a user clicks the "delete" link, we already trigger a "contact:delete" event within the `ItemView` so the controller can delete the model. For argument's sake, let's say that we wanted to make our entire table (i.e. the parent `CompositeView`) fade out and fade back in when a contact gets deleted.[76]

When an event is triggered within a view, Marionette calls a corresponding method if it's defined (see documentation[77]). So in our `CompositeView`, we can define this magical corresponding method to make our table flash:

Leveraging `triggerMethod` (assets/js/apps/contacts/list/list_view.js)

```
List.Contacts = Marionette.CompositeView.extend({
  // attributes (tagName, etc.) are here

  onItemviewContactDelete: function(){
    this.$el.fadeOut(1000, function(){
      $(this).fadeIn(1000);
    });
  }
});
```

Note that this code only serves as an example and does *not* get added to our application, so you won't see it going forward.

Remember that the "contact:delete" event is triggered within a child `ItemView` and that the event that Marionette triggers on the parent view is prefixed with "`itemview:`". Since the event that gets bubbled up to the `CompositeView` is "itemview:contact:delete", the corresponding method that will be called is `onItemviewContactDelete` (see the documentation[78] for more information on the correspondence rules).

After refreshing the page, when we delete a contact, we can see our table successfully fades out and back in. All we had to do was define a method matching our event, the rest is wired up for us by Marionette! Don't worry, we'll use `triggerMethod` to implement intelligent functionality later.

[76] Yes, this is bad UX design, and it's a pretty dumb feature. But it illustrates the point...

[77] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.functions.md#marionettetriggermethod

[78] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.functions.md#marionettetriggermethod

Displaying Contacts in Dedicated Views

So far, we've implemented the functionality to display a list of our contacts. Let's now display contacts in a dedicated pages.

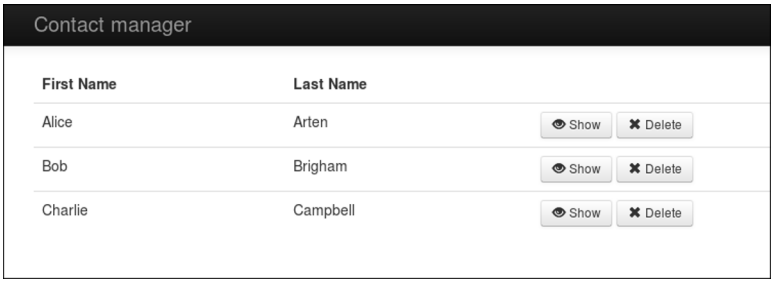
Wiring up the Show Event

First, let's add a "Show" button in our template:

Adding a 'show' button to the contact template in index.html

```
<script type="text/template" id="contact-list-item">
  <td><%= firstName %></td>
  <td><%= lastName %></td>
  <td>
    <a href="#" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

You'll notice we've already included the `js-show` class we'll use in our event selector. Here's our contacts list with the new "show" links:



The contacts list with "show" links styled as buttons

Now, we can have our view trigger an event when the link is clicked:

Event listener for 'show' link click

```
1  List.Contact = Marionette.ItemView.extend({
2    // tagName and template attributes
3
4    events: {
5      "click": "highlightName",
6      "click td a.js-show": "showClicked",
7      "click button.js-delete": "deleteClicked"
8    },
9
10   // highlightName event handler
11
12   showClicked: function(e){
13     e.preventDefault();
14     e.stopPropagation();
15     this.trigger("contact:show", this.model);
16   },
17
18   // deleteClicked event handler
19 });
```

We're using an event listener instead of a `trigger` object (see this exercise solution), because it allows us to prevent the event from bubbling up in the DOM and getting our row highlighted, as you may remember. In addition, we need to prevent the browser from navigating to the link's location: when we click the link, the browser will by default navigate to that location. The problem is, we don't want the browser navigating our app: when the user clicks the "show" link, the model gets displayed in a dedicated view, and that's it. So we prevent the link from leading our browser anywhere by calling `preventDefault`[79] on line 13.

[79] http://api.jquery.com/event.preventDefault/

If we're going to prevent the browser from navigating to the URL and we're not using the `href` attribute in any way, why bother having a link with an `href`? Because it enables our application to behave as expected: users can open links in new tabs, save link locations, etc. This behavior wouldn't be possible without using links with normal markup: using links with "#" as the `href` attribute breaks a lot of web functionality (which is why we'll fix them soon, by implementing routing).

We still need to process the triggered event (line 15) in our controller:

Listening to the 'contact:show' event (assets/js/apps/contacts/list/list_controller.js)

```
1  List.Controller = {
2    listContacts: function(){
3      var contacts = ContactManager.request("contact:entities");
4
5      var contactsListView = new List.Contacts({
6        collection: contacts
7      });
8
9      // "itemview:contact:delete" event listener
10
11     contactsListView.on("itemview:contact:show",
12                                         function(childView, model){
13       console.log("Received itemview:contact:show event on model ", model)
14     });
15
16     ContactManager.mainRegion.show(contactsListView);
17   }
18 }
```

When we refresh the page and click on a "show" link, we can see we've got the proper output in the console, so our events are wired up properly. We now need to get the controller to execute the proper actions when this event is triggered.

The ContactsApp.Show Sub-Module

As we saw in the chapter on structuring code, we'll be creating sub-modules for each "functional end result". Since "displaying a given contact" is a different functionality than "listing all the contacts", we'll need a new sub-module: `ContactsApp.Show`. Let's start by creating the controller:

The `ContactsApp.Show` controller in assets/js/apps/contacts/show/show_controller.js

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
Backbone, Marionette, $, _){
  Show.Controller = {
    showContact: function(model){
      console.log("showContact called for model ", model)
    }
  }
});
```

Don't forget to include this new file in `index.html`

We now call our new `Show` controller from our `List` controller:

Listening to the contact:show event (assets/js/apps/contacts/list/list_controller.js)

```
List.Controller = {
  listContacts: function(){
    // same code as above

    contactsListView.on("itemview:contact:show", function(childView,
                                                          model){
      ContactManager.ContactsApp.Show.Controller.showContact(model);
    });

    ContactManager.mainRegion.show(contactsListView);
  }
}
```

Our `Show` controller is being called when the "show" link gets clicked, but we still don't have anything to display. Let's add a template:

Adding a template in index.html to view a contact

```
<script type="text/template" id="contact-view">
  <h1><%= firstName %> <%= lastName %></h1>
  <p><strong>Phone number:</strong> <%= phoneNumber %></p>
</script>
```

And of course, we need to add a view to actually display the template, which we put in a new file:

View to display a contact (assets/js/apps/contacts/show/show_view.js)

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
Backbone, Marionette, $, _){
  Show.Contact = Marionette.ItemView.extend({
    template: "#contact-view"
  });
});
```

With all the supporting bits in place, we can now instantiate our view in the controller, and display it within the main region:

Displaying the rendered contact view (assets/js/apps/contacts/show/show_controller.js)

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
Backbone, Marionette, $, _){
  Show.Controller = {
    showContact: function(model){
      var contactView = new Show.Contact({
        model: model
      });

      ContactManager.mainRegion.show(contactView);
    }
  }
});
```

Let's quickly make sure you've included all the new files in `index.html`. The includes section should now look like this:

Including all our app files in index.html

```
<script src="./assets/js/app.js"></script>
<script src="./assets/js/entities/contact.js"></script>
<script src="./assets/js/apps/contacts/list/list_view.js"></script>
<script src="./assets/js/apps/contacts/list/list_controller.js"></script>
<script src="./assets/js/apps/contacts/show/show_view.js"></script>
<script src="./assets/js/apps/contacts/show/show_controller.js"></script>
```

When we refresh the page, our contact is now properly displayed in its own view when we click the "show" link:

A contact's dedicated display view

If you've used plain Backbone before, you'll appreciate not needing to handle closing the currently displayed views: Marionette does it for you.

Git commit displaying a contact in a dedicated view:

91e237d231a20f8d2e2ec565e3e12b0da98a5070[80]

[80] https://github.com/davidsulc/marionette-gentle-introduction/commit/91e237d231a20f8d2e2ec565e3e12b0da98a5070

Implementing Routing

Our `ContactManager` app now lets users navigate from the contacts index to a page displaying a contact. But once the user gets to the contact's page, he's stuck: the browser's "back" button doesn't work. In addition, users can't bookmark a contact's display page: the URL saved by the browser would be the index page. Later, when the user loads the bookmark again, the user will end up seeing the contact *list view* instead of the contact's display page he expected. To address these issues, we'll implement routing in our application.

How to Think About Routing

It's important that we define the router's role, in order to design our app properly. All a router does is

- execute controller actions corresponding to the URL with which the user first "entered" our Marionette app. It's important to note that the route-handling code should get fired *only* when a user *enters* the application by a URL, not each time the URL changes. Put another way, once a user is within our Marionette app, the route-handling shouldn't be executed again, even when the user navigates around;
- update the URL in the address bar as the user navigates within the app (i.e. keep the displayed URL in sync with the application state). That way, a user could potentially use the same URL (by bookmarking it, emailing it to a friend, etc.) to "restore" the app's current confguration (i.e. which views are displayed, etc.). Keeping the URL up to date also enables the browser's "back" and "forward" buttons to function properly.

It's very important to differentiate *triggering routing events* from *updating the URL*. In traditional web frameworks, actions are triggered by hitting their corresponding URLs. This isn't true for javascript web applications: our ContactManager has been working just fine (even "changing pages") without ever caring about the current URL.

And now that we have a basic app functioning as we want it to, we'll add in a router to manage the URL-related functionality. Our router will only get triggered by the first URL it recognizes, resulting in our app getting "initialized" to the correct state (i.e. showing the proper data in the proper views). After that initialization step has fired *once*, the router *only* keeps the URL up to date as the user navigates our app: changing the displayed content will be handled by our controllers, as it has been up to now.

Adding a Router to ContactsApp

Now that we have a better idea of how routing should be used, let's add a router to our `ContactsApp` by creating a new file:

Adding a router to our `ContactsApp` (assets/js/apps/contacts/contacts_app.js)

```
1  ContactManager.module("ContactsApp", function(ContactsApp, ContactManager,
2  Backbone, Marionette, $, _){
3    ContactsApp.Router = Marionette.AppRouter.extend({
4      appRoutes: {
5        "contacts": "listContacts"
6      }
7    });
8
9    var API = {
10     listContacts: function(){
11       console.log("route to list contacts was triggered");
12     }
13   };
14
15   ContactManager.addInitializer(function(){
16     new ContactsApp.Router({
17       controller: API
18     });
19   });
20 });
```

As you can tell from the module callback on line 1, we're defining the router within the `ContactsApp` module because it will handle the routes for all the sub-modules attached to `ContactsApp` (such as `List`, `Show`, etc.). On line 3, we attach a `Router` instance containing an `appRoutes`[81] object associating the URL fragments on the left with callback methods on the right.

Next, we define public methods within an `API` object on lines 9-13, which is provided to the router during instantiation on line 17. Note that the callback function (e.g. `listContacts`) specified in the `appRoutes` object above *must* exist in the router's controller. In other words, all the callbacks used in the `appRoutes` object must be located in our `API` object.

[81] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.approuter.md#configure-routes

Let's briefly talk about initializers: as you can see on line 15 above, we're adding an initializer by calling the aptly named `addInitializer` method. So why are we listening for the "initialize:after" event in other circumstances, instead of using `addInitializer`? Execution order. We can add initializers with calls to `addInitializer`, and the provided functions will be executed when the application is running. Then, once *all* initializers have been run, the "initialize:after" event is triggered. We'll discuss further the implications of this difference below.

Don't forget to add the sub-application file to our includes in `index.html`:

index.html

```
1  <script src="./assets/js/app.js"></script>
2  <script src="./assets/js/entities/contact.js"></script>
3
4  <script src="./assets/js/apps/contacts/contacts_app.js"></script>
5  <script src="./assets/js/apps/contacts/list/list_view.js"></script>
6  <script src="./assets/js/apps/contacts/list/list_controller.js"></script>
7  <script src="./assets/js/apps/contacts/show/show_view.js"></script>
8  <script src="./assets/js/apps/contacts/show/show_controller.js"></script>
```

When we enter "index.html#contacts" in our browser's address bar and hit enter, we expect to see "route to list contacts was triggered" in the console but nothing happens. That is because the URL management is delegated to Backbone's `history`[82], which we haven't started. So let's add the code for starting Backbone's `history` in our app's initializer:

Starting Backbone's `history` in assets/js/app.js

```
1  ContactManager.on("initialize:after", function(){
2    if(Backbone.history){
3      Backbone.history.start();
4    }
5  });
```

[82] http://backbonejs.org/#History

The difference between listening for the "initialize:after" event and calling the `addInitializer` method (as discussed above) has important implications for our application: we can only start Backbone's routing (via the `history` attribute) once *all* initializers have been run, to ensure the routing controllers are ready to respond to routing events. Otherwise (if we simply used `addInitializer`), Backbone's routing would be started, triggering routing events according to the URL fragments, but these routing events wouldn't be acted on by the application because the routing controllers haven't been defined yet!

Another important difference between "initialize:after" and `addInitializer` is if and when the provided function argument is executed:

- the "initialize:after" event listener can only respond to events triggered *after* it has been defined. This means that if you define your listener after the "initialize:after" event has been triggered, nothing will happen;
- the `addInitializer` method will execute the provided function when the app is running. This means that if the app isn't yet running, it will wait until the app has started before running the code; but if the app is already running by the time you call `addInitializer`, the function will be executed immediately.

If we now hit the "index.html#contacts" URL as an entry point, we'll see the expected output in our console. We've got history working!

But you'll also see that our app no longer lists our contacts: we've removed the line that called our `listContacts` action in the app initializer code, namely:

```
ContactManager.ContactsApp.List.Controller.listContacts();
```

We need to trigger this controller action from our `ContactsApp` routing controller:

Adding a router to our `ContactsApp` (assets/js/apps/contacts/contacts_app.js)

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

```
10      listContacts: function(){
11        ContactsApp.List.Controller.listContacts();
12      }
13    };
14
15    ContactManager.addInitializer(function(){
16      new ContactsApp.Router({
17        controller: API
18      });
19    });
20  });
```

We simply needed to change line 11 to execute the proper controller action, and we're in business: entering "index.html#contacts" in the browser's address bar displays our contacts, as expected. But if we go to "index.html", nothing happens. Why is that?

It's pretty simple, really: we've started managing our app's initial state with routes, but have no route registered for the root URL.

What about pushState?

Backbone allows you to leverage HTML5's pushState[83] functionality by changing your `history` starting code to `Backbone.history.start({pushState: true});` (see documentation[84]).

When using pushState, URL fragments look like the usual "/contacts/3" instead of "#contacts/3". This allows you to serve an enhanced, javascript-heavy version of the page to users with javascript-enabled browsers, while serving the basic HTML experience to clients without javascript (e.g. search engine crawlers). Be aware, however, that **to use pushState in your application your server has to respond to that URL**. This is a frequent error when trying out pushState.

You're free to have your server systematically respond with your `index.html` page regardless of the requested URL, but *something* needs to be sent to the client when the URL is requested (e.g. when loading a bookmark). When sending `index.html` to all client requests, you're bascially delegating the URL resolution to your Marionette app: when the browser will load `index.html`, the app will start along with the route-handling code, which will load the correct application state (since the route corresponding to the URL requested by the client will get triggered).

Another strategy is to progressively enhance your application, as Derick Bailey introduced in a blog post[85].

A great resource to read up on HTML5's History API is Dive Into HTML5[86], and the links provided in its "Further Reading" paragraph at the end.

Routing Helpers

Here's what we want to do: if the user comes to our app at the root URL, let's redirect him to "#contacts". The basic way of accomplishing this would be:

[83] http://www.whatwg.org/specs/web-apps/current-work/multipage/history.html#history
[84] http://backbonejs.org/#History-start
[85] http://lostechies.com/derickbailey/2011/09/26/seo-and-accessibility-with-html5-pushstate-part-1-introducing-pushstate/
[86] http://diveintohtml5.info/history.html

Redirecting to the root URL (assets/js/app.js)

```
1  ContactManager.on("initialize:after", function(){
2    if(Backbone.history){
3      Backbone.history.start();
4
5      if(Backbone.history.fragment === ""){
6        Backbone.history.navigate("contacts");
7        ContactManager.ContactsApp.List.Controller.listContacts();
8      }
9    }
10 });
```

On line 5, we check the URL fragment (i.e. the string that comes after "index.html" in the URL, ignoring the `#` character): if it's empty, we need to redirect the user. Except that in javascript web apps, "redirecting" is a bit of a misnomer: we're not redirecting anything (as we would be with a server), we are just

- updating the URL with the proper fragment (line 6)
- executing the proper controller action (line 7), which will display the desired views

You can achieve the same result by putting `Backbone.history.navigate("contacts", {trigger: true});` on line 6, and removing line 7. You will sometimes see this done in various places on the web, but it encourages bad app design and **it is strongly recommended you don't pass** `trigger:true` **to** `Backbone.history.navigate`. Derick Bailey (Marionette's creator) even wrote a blog post[87] on the subject.

Triggering routes to execute desired behavior is a natural reflex when you're coming from typical stateless web development, because that's how it works: the user hits a URL endpoint, and the corresponding actions are performed. And although triggering the route looks better at first glance (less code), it will expose you to design problems: if you're unable to get your app to behave as expected using controller methods, you've got issues that should be addressed. Keeping the `{trigger: false}` default when navigating will encourage the proper separation of app behavior and URL management, as discussed above.

Note that `navigate` doesn't just change the URL fragment, it also adds the new URL to the browser's history. This, in turn, makes the browser's "back" and "forward" buttons behave as expected.

Let's get back to our code and refactor: checking the current URL fragment and keeping it up to date are things we'll be doing quite frequently as we develop our app. Let's extract them into functions attached to our app:

[87] http://lostechies.com/derickbailey/2011/08/28/dont-execute-a-backbone-js-route-handler-from-your-code/

Redirecting to the root URL (assets/js/app.js)

```
1  var ContactManager = new Marionette.Application();
2
3  ContactManager.addRegions({
4    mainRegion: "#main-region"
5  });
6
7  ContactManager.navigate = function(route, options){
8    options || (options = {});
9    Backbone.history.navigate(route, options);
10 };
11
12 ContactManager.getCurrentRoute = function(){
13   return Backbone.history.fragment
14 };
15
16 ContactManager.on("initialize:after", function(){
17   if(Backbone.history){
18     Backbone.history.start();
19
20     if(this.getCurrentRoute() === ""){
21       this.navigate("contacts");
22       ContactManager.ContactsApp.List.Controller.listContacts();
23     }
24   }
25 });
```

We've simply declared helper functions on lines 7 and 12, and we then use them on lines 20-21. Note that line 8 essentially sets `options` to `{}` if none are provided (i.e. it sets a default value).

If you think about it, these helper functions aren't really specific to our application: they're closer to extensions of the Marionette framework. For simplicity's sake, we've kept the code above in the main app, but refer to the Extending Marionette chapter to see how this can be accomplished to clean up our code further.

DRYing up Routing with Events

Right now, our app is manually changing the URL and calling a controller action if the URL contains no fragement. But that isn't very DRY[88]: we'll end up setting route fragements and calling controller methods everywhere, and it will be a nightmare to maintain.

Instead, let's leverage events (line 6):

Triggering an event in 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  });
```

Then, we update the URL fragement and call the appropriate action within our controller by listening for that same event (lines 15-18):

Responding to the navigation event in assets/js/apps/contacts/contacts_app.js

```
1  ContactManager.module("ContactsApp", function(ContactsApp, ContactManager,
2  Backbone, Marionette, $, _){
3    ContactsApp.Router = Marionette.AppRouter.extend({
4      appRoutes: {
5        "contacts": "listContacts"
6      }
7    });
8
9    var API = {
10     listContacts: function(){
11       ContactsApp.List.Controller.listContacts();
12     }
13   };
14
```

[88] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

```
  ContactManager.on("contacts:list", function(){
    ContactManager.navigate("contacts");
    API.listContacts();
  });

  ContactManager.addInitializer(function(){
    new ContactsApp.Router({
      controller: API
    });
  });
});
```

Much better! We now have proper URL handling without needing to trigger routes. This will be very useful as the app grows: we can simply trigger the appropriate events for our sub-applications to respond and update the displayed information.

Git commit implementing our first route:

712ee23a322202c61ef9536062931b8da6daa494[89]

Adding a Show Route

Let's now add a route to show a given contact. In other words, we'll add a route handler for URL fragments that look like "contacts/3", where 3 would be the contact's `id`.

Let's start by taking care of displaying the contact when a URL of this type is hit, by adding to our `ContactsApp` file:

Responding to the 'contacts/ID' URLs (assets/js/apps/contacts/contacts_app.js)

```
ContactManager.module("ContactsApp", function(ContactsApp,
ContactManager, Backbone, Marionette, $, _){
  ContactsApp.Router = Marionette.AppRouter.extend({
    appRoutes: {
      "contacts": "listContacts",
      "contacts/:id": "showContact"
    }
  });

```

[89] https://github.com/davidsulc/marionette-gentle-introduction/commit/712ee23a322202c61ef9536062931b8da6daa494

```
10    var API = {
11      listContacts: function(){
12        ContactsApp.List.Controller.listContacts();
13      },
14
15      showContact: function(id){
16        ContactsApp.Show.Controller.showContact(id);
17      }
18    };
19
20    ContactManager.on("contacts:list", function(){
21      ContactManager.navigate("contacts");
22      API.listContacts();
23    });
24
25    ContactManager.addInitializer(function(){
26      new ContactsApp.Router({
27        controller: API
28      });
29    });
30  });
```

On line 6, we declare the route. You'll notice that we can provide parameters such as "`:id`" to match a single URL component between slashes, and provide it to the controller's handling function. You can learn more about route strings in the documentation[90].

Now we extract the contact's `id` from the URL, we send it on to our trusty `Show` controller (on line 16) to display the data. Except we're prodiving an `id` integer, when the controller's expecting a model instance. What to do? We could of course get the model in the routing controller before passing it on, but that's not the router's job. Besides, the `Show` controller should fetch the model on it's own to ensure it's displaying the latest data. So let's fix it:

Modifying the controller to receive an id as the argument (assets/js/apps/contacts/show/show_controller.js)

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3      Show.Controller = {
4        showContact: function(id){
5          var contacts = ContactManager.request("contact:entities");
6          var model = contacts.get(id);
```

[90] http://backbonejs.org/#Router-routes

```
7           var contactView = new Show.Contact({
8             model: model
9           });
10
11          ContactManager.mainRegion.show(contactView);
12        }
13      }
14    });
```

The code up here is slightly odd, because at this point we're dealing only with data in memory: it isn't saved anywhere, so we can't (e.g.) load a model instance directly from storage. Instead we retrieve the model instance from the collection with `get`[91] on line 6, which means we need to get a reference to our `contacts` collection first (line 5). Don't worry, this code will be refactored when we implement data storage. As our old friend Ned Stark would say, "persistence is coming"...

Let's refresh our page to load our app's new version, and manually type in the "#contacts/2" URL fragement in the address bar. When we hit enter, the appropriate contact is displayed. If we then enter "#contacts/3", another contact is displayed, as expected. In addition, pressing the browser's "back" button works properly. Fabulous!

But what if we enter "#contacts" in the address bar, then click the "show" button for the second contact? The proper contact is displayed, but the address bar still indicates "#contacts" instead of the correct "#contacts/2": we haven't updated the URL as we've navigated in our application. We could just update the fragment in our `List` controller by adding line :

Updating the URL fragment on display change (assets/js/apps/contacts/list/list_controller.js)

```
1 contactsListView.on("itemview:contact:show", function(childView, model){
2   ContactManager.navigate("contacts/" + model.get("id"));
3   ContactManager.ContactsApp.Show.Controller.showContact(model);
4 });
```

But that would mean duplicating the call to the controller action, and the `navigate` function from every place in our app where we'd want to display a contact. Not very DRY...

Instead, let's centralize that functionality within the routing controller, and simply trigger an event that the routing controller will react to:

[91] http://backbonejs.org/#Collection-get

Triggering an event to display a contact (assets/js/apps/contacts/list/list_controller.js)

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

Now, let's adapt our routing controller code:

Responding to the 'contacts/ID' URLs (assets/js/apps/contacts/contacts_app.js)

```
ContactManager.module("ContactsApp", function(ContactsApp, ContactManager, Backbo\
ne, Marionette, $, _){
  ContactsApp.Router = Marionette.AppRouter.extend({
    appRoutes: {
      "contacts": "listContacts",
      "contacts/:id": "showContact"
    }
  });

  var API = {
    listContacts: function(){
      ContactsApp.List.Controller.listContacts();
    },

    showContact: function(id){
      ContactsApp.Show.Controller.showContact(id);
    }
  };

  ContactManager.on("contacts:list", function(){
    ContactManager.navigate("contacts");
    API.listContacts();
  });

  ContactManager.on("contact:show", function(id){
    ContactManager.navigate("contacts/" + id);
    API.showContact(id);
  });

  ContactManager.addInitializer(function(){
```

```
    new ContactsApp.Router({
      controller: API
    });
  });
});
```

This implementation is much cleaner: from within our application, we simply indicate where the user should be led, and the routing controller takes care of the rest, namely

- updating the URL fragment
- executing the appropriate controller action

It's also possible to scope events to a sub-application by using (e.g.) `ContactManager.ContactsApp.trigger(...)`

One last thing, before we consider ourselves done: let's add the proper link to the show button, so users can do things like opening the link in a new tab, copying the link location, etc. We need to modify our template in `index.html`:

Adding the proper link value in our contact-list-item template (index.html)

```
<script type="text/template" id="contact-list-item">
  <td><%= firstName %></td>
  <td><%= lastName %></td>
  <td>
    <a href="#contacts/<%= id %>" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

Git commit implementing the show route:

b9160fcd134b4045dd341c4650170e6349d3bbd4[92]

[92] https://github.com/davidsulc/marionette-gentle-introduction/commit/b9160fcd134b4045dd341c4650170e6349d3bbd4

Why don't we make a route for delete?

The main reason: we have direct access to the model within our app, so there's no need for a route. This is a stark contrast to traditional server-side MVC, where you'd need a deletion route to determine the model to destroy.

Additionally, calling `navigate` with a deletion route would add it to the browser's history. This, in turn, will make you enter a world of pain as the user presses the "back" button and hits a URL meant to delete a model that no longer exists.

All in all, your experience with routing will be a much happier one if you bear in mind 2 guiding concepts:

- avoid *at all costs* passing "`trigger: true`" to `navigate`;
- if the user shouldn't be able to "save" (i.e. bookmark) and application's state, it shouldn't have a URL. In other words, nowhere in you app should you have defined routes for this action, nor should you call `navigate` for it.

To illustrate the last point, think about (e.g.) creating a new contact:

- the user should be able to bookmark the page with the creation form, so it gets a route, and gets navigated to;
- the user should *not* be able to bookmark the place in the application where a new model is instantiated and saved, so there's no associated route and we don't navigate to it: the action simply gets executed within the application.

Derick Bailey has a great blog post[93] on the issue.

Exercise

Getting Back to the Contacts List

When the user is on a contact's display page (e.g. "#contacts/2"), there's no way for him to return to the page listing all contacts (i.e. "#contacts"). Add a simple link to the contact page template (`#contact-view`) that will take the user to the page listing all contacts.

You can see the exercise solution at the end of the book.

[93] http://lostechies.com/derickbailey/2011/08/03/stop-using-backbone-as-if-it-were-a-stateless-web-server/

Note we're not including this "feature" within the main code, because we're going to implement a full-featured menu later. With the navigation header in place, there will be no need for this link on a contact's page.

Implementing a View for Nonexistent Contacts

Now that we have routing implemented, we need to manage routing issues. Notably, displaying an error message when a user loads the URL for a nonexistent contact (e.g. "#contacts/999", or "#contacts/foo"). Right now, our app will display a blank page, which isn't very helpful.

We'll need a new template to display our error message; let's add it to `index.html`:

The template for nonexistent contacts (index.html)

```
1  <script type="text/template" id="missing-contact-view">
2    <div class="alert alert-error">This contact doesn't exist !</div>
3  </script>
```

We'll also need an `ItemView` to display it (lines 3-5):

The item view for nonexistent contacts (assets/js/apps/contacts/show/show_view.js)

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3    Show.MissingContact = Marionette.ItemView.extend({
4      template: "#missing-contact-view"
5    });
6
7    Show.Contact = Marionette.ItemView.extend({
8      template: "#contact-view"
9    });
10 });
```

And, last but not least, we need to adapt our controller's `showContact` function to display the appropriate view:

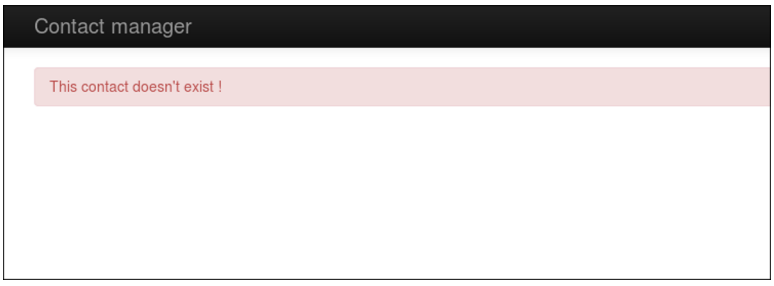
Making the controller display the right view (assets/js/apps/contacts/show/show_controller.js)

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3    Show.Controller = {
4      showContact: function(id){
5        var contacts = ContactManager.request("contact:entities");
6        var model = contacts.get(id);
7        var contactView;
8        if(model !== undefined){
9          contactView = new Show.Contact({
10           model: model
11         });
12       }
13       else{
14         contactView = new Show.MissingContact();
15       }
16
17       ContactManager.mainRegion.show(contactView);
18     }
19   }
20 });
```

First, we check if the model exists on line 8:

- if it does, instantiate a new contact view on lines 9-11
- otherwise, instantiate the view for nonexistent contacts on line 14

Finally, regardless of the view we've instantiated, display it within the `mainRegion` (same as before) on line 17. Here's the error message that gets displayed for the "#contacts/foo" URL:



Error message for nonexistent contacts

Git commit to display error message for nonexistnent contacts:

4b932caf88f7026e8414d6801ffa6877881a012e[94]

[94] https://github.com/davidsulc/marionette-gentle-introduction/commit/4b932caf88f7026e8414d6801ffa6877881a012e

Dealing with Persisted Data

Our goal in this chapter is to modify our app to deal with *persisted* data, as opposed to the in-memory management we've used up to this point. We'll be using web storage[95] to persist our data, so make sure you have a recent browser that can use this technology. You can find a browser compatibility chart at Can I use...[96], and you can learn more about web storage in Dive into HTML5[97].

Since they are more advanced, the implementation details of the persistence layer are covered in chapter Using Web Storage for Persistence at the end of the book. You won't need to understand the implementation details to follow along as we continue developing our app: using the web storage persistence will work the same as saving data to a remote server (except for one magical line I'll point out).

Implementing a RESTful API is beyond the scope of this book. However, in addition to the various free tutorials that undoubtedly cover your favorite stack, you can refer to Addy Osmani's book on Backbone[98] for more information:

- RESTful Persistence[99]
- Creating a RESTful API[100]
- Interacting with a server[101]

You can get the code to our app (now using persistence) from this zip file[102], or by using Git to checkout the proper commit in your cloned repository:

```
git fetch origin
git cherry-pick f4e0c7c5034fc367696ed31a3fe0bbe9a84ad39e
```

Alternatively, if you just want to view the code modifications, see commit

f4e0c7c5034fc367696ed31a3fe0bbe9a84ad39e[103]

[95] http://www.html5rocks.com/en/features/storage
[96] http://caniuse.com/#search=namevalue-storage
[97] http://diveintohtml5.info/storage.html
[98] http://addyosmani.github.io/backbone-fundamentals/
[99] http://addyosmani.github.io/backbone-fundamentals/#restful-persistence
[100] http://addyosmani.github.io/backbone-fundamentals/#creating-the-back-end
[101] http://addyosmani.github.io/backbone-fundamentals/#talking-to-the-server
[102] https://github.com/davidsulc/marionette-gentle-introduction/archive/f4e0c7c5034fc367696ed31a3fe0bbe9a84ad39e.zip
[103] https://github.com/davidsulc/marionette-gentle-introduction/commit/f4e0c7c5034fc367696ed31a3fe0bbe9a84ad39e

Adding a Location to our Entities

Backbone models and collections need to indicate where they are located on a server. And since we said our persistence strategy would work just like using a remote server, our `ContactCollection` will need a `url`[104] property defined like this:

Adding a `url` property to the `ContactCollection` (assets/js/entities/contact.js)

```
1 Entities.ContactCollection = Backbone.Collection.extend({
2   url: "contacts",
3   model: Entities.Contact,
4   comparator: "firstName"
5 });
```

On line 2, we indicate that our collection is located at the "contacts" URL fragment.

The `url` can also be defined as a function (see documentation[105]).

By providing a `url` to our collection definition, we let Backbone step in with its magic and manage all the persistence for us:

- to create a new contact, Backbone will POST the json data to "/contacts"
- to read the contact with id 3, Backbone will GET the json data from "/contacts/3"
- to update the contact with id 3, Backbone will PUT the json data to "/contacts/3"
- to delete the contact with id 3, Backbone will DELETE the json data to "/contacts/3"

As you can tell, Backbone expects to be interacting with a RESTful API implementation on the server (see documentation[106]). Naturally, in our case, all these calls get hijacked to use web storage instead because we're not using a server.

Since we also need to deal with models that don't belong to a collection (e.g. when loading a *single* contact to display), we need a means to indicate a specific model's location. For this purpose, we need to set a `urlRoot` property on the model:

[104] http://backbonejs.org/#Collection-url
[105] http://backbonejs.org/#Collection-url
[106] http://backbonejs.org/#Sync

Adding a `url` property to the `Contact` definition (assets/js/entities/contact.js)

```
1 Entities.Contact = Backbone.Model.extend({
2   urlRoot: "contacts"
3 });
```

The `urlRoot` can also be defined as a function (see documentation[107]). In addition, you may specify a `url` value (as a string or function) on the model if you don't want Backbone to generate the URL based on the `urlRoot` property or using the `url` defined on the model's collection.

Dealing with a non-RESTful API

If you're working with an existing API that doesn't match Backbone's RESTful API expectations, you'll have to do some work to get Backbone to manage persistence properly. These are worth looking into:

- overriding the model/collection's `parse`[108] method if your server is returning JSON data that doesn't match Backbone's expectations. In other words, you'll need to tell Backbone how to parse the received JSON data to turn it into a model instance;
- creating `url`[109] as a function if the server doesn't use the RESTful URLs expected by Backbone. This will make Backbone use the proper endpoint when calling `url()` on a model/collection to determine where data is persisted. Note that you'll most likely need to test for `this.id` when building the URL in the model's case, to determine whether you're dealing with a new model instance that has to be created/persisted, or if it's an existing model that needs to be modified/fetched;
- setting `Backbone.emulateHTTP`[110] to `true` if your server can't handle HTTP verbs;
- setting `Backbone.emulateJSON`[111] to `true` if your server can't handle data encoded as JSON;
- overriding the default `sync` method to suit your configuration. You can do this either on a per-entity basis (by putting the `sync`[112] function within the model/collection definition), or app-wide by overriding `Backbone.sync`[113].

[107] http://backbonejs.org/#Model-urlRoot
[108] http://backbonejs.org/#Model-parse
[109] http://backbonejs.org/#Model-url
[110] http://backbonejs.org/#Sync-emulateHTTP
[111] http://backbonejs.org/#Sync-emulateJSON
[112] http://backbonejs.org/#Model-sync
[113] http://backbonejs.org/#Sync

Configuring our Entities to use Web Storage

Now that we have `url` and `urlRoot` properties defined, we simply need to use the function we created in the appendix to get our contact entities to store their data locally:

Configuring our entities to use web storage (assets/js/entities/contact.js)

```
1  Entities.Contact = Backbone.Model.extend({
2    urlRoot: "contacts"
3  });
4
5  Entities.configureStorage(Entities.Contact);
6
7  Entities.ContactCollection = Backbone.Collection.extend({
8    url: "contacts",
9    model: Entities.Contact,
10   comparator: "firstName"
11 });
12
13 Entities.configureStorage(Entities.ContactCollection);
```

The only difference between our using web storage instead of a remote server are lines 5 and 13.

Loading our Contacts Collection

Since our contacts are now persisted, let's adapt how we load them:

Fetching persisted contacts in assets/js/entities/contact.js

```
1  getContactEntities: function(){
2    var contacts = new Entities.ContactCollection();
3    contacts.fetch();
4    return contacts;
5  }
```

Backbone knows where the contacts are stored (via the `url` property), so all we have to do is to `fetch`[114] them. Of course, we don't have any persisted contacts right now, and we don't have a way to create them either. So let's cheat by initializing a few contacts if our collection is empty when we load it:

Initializing the `contacts` collection (assets/js/entities/contact.js)

```
1  var initializeContacts = function(){
2    var contacts = new Entities.ContactCollection([
3      { id: 1, firstName: "Alice", lastName: "Arten",
4        phoneNumber: "555-0184" },
5      { id: 2, firstName: "Bob", lastName: "Brigham",
6        phoneNumber: "555-0163" },
7      { id: 3, firstName: "Charlie", lastName: "Campbell",
8        phoneNumber: "555-0129" }
9    ]);
10   contacts.forEach(function(contact){
11     contact.save();
12   });
13   return contacts;
14 };
15
16 var API = {
17   getContactEntities: function(){
18     var contacts = new Entities.ContactCollection();
19     contacts.fetch();
20     if(contacts.length === 0){
21       // if we don't have any contacts yet, create some for convenience
22       return initializeContacts();
23     }
24     return contacts;
25   }
26 };
```

You'll notice that on lines 10-12 we save each contact we've created. That way, they get persisted and we'll have some contacts to load next time around.

[114] http://backbonejs.org/#Collection-fetch

This code doesn't deal with latency! If you're using a remote server for persistence, it's quite possible the collection length will be checked before the data has returned from the server. We will see later how to deal with latency using jQuery deferreds[115].

In addition, this code won't properly deal with nonexistent contacts (i.e. the missing contact view won't be displayed), because that also requires using deferreds. But everything will be restored to proper working order in the next chapter!

Happily, since we're already requesting the contacts from the `Entities` module in our `List` submodule's controller, there's nothing to modify: listing contacts will properly load the contacts from storage.

Loading a Single Contact

However, we need to change our `Show` controller to load the single contact we wish to display. This is what our code looks like now:

Our current `Show` controller (assets/js/apps/contacts/show/show_controller.js)

```
1  Show.Controller = {
2    showContact: function(id){
3      var contacts = ContactManager.request("contact:entities");
4      var model = contacts.get(id);
5
6      // instantiate and display appropriate view
7    }
8  }
```

Let's change our `Show` controller to:

Changing the `Show` controller to fetch a single contact (assets/js/apps/contacts/show/show_controller.js)

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3    Show.Controller = {
4      showContact: function(id){
5        var contact = ContactManager.request("contact:entity", id);
6        var contactView;
7        if(contact !== undefined){
```

[115] http://api.jquery.com/category/deferred-object/

```
8           contactView = new Show.Contact({
9             model: contact
10          });
11        }
12        else{
13          contactView = new Show.MissingContact();
14        }
15
16        ContactManager.mainRegion.show(contactView);
17      }
18    }
19  });
```

On line 5, we request a single contact by triggering a request event, followed by an `id` argument. Of course, if we're requesting a single contact from our app, we need to implement the handler to process that request. Let's add it to our `Entities` module:

Adding a handler to return a single contact (assets/js/entities/contact.js)

```
1  var API = {
2    // getContactEntities function
3
4    getContactEntity: function(contactId){
5      var contact = new Entities.Contact({id: contactId});
6      contact.fetch();
7      return contact;
8    }
9  };
10
11 // handler for "contact:entities"
12
13 ContactManager.reqres.setHandler("contact:entity", function(id){
14   return API.getContactEntity(id);
15 });
```

Since we receive an `id` argument as part of the request, we simply declare it in our handler signature (line 13), and pass it on to the `getContactEntity` in our controller API. Then, on line 5, we create a new `Contact` instance with the proper `id`. Specifying the `id` attribute is important, because it lets Backbone know our contact instance already exists and has been persisted. That way, when we call `fetch` on line 6, the correct contact instance is retrieved, namely the one whose `id` matches.

Deleting a Contact

We also need to adapt the code deleting a contact, so the model gets erased from storage. Previously, we simply removed it from the collection, but we now need to actually *delete* it. This is our current code in the `List` controller:

Current code in the `List` controller for 'deleting' a contact (assets/js/apps/contacts/list/list_controller.js)

```
1 contactsListView.on("itemview:contact:delete", function(childView, model){
2   contacts.remove(model);
3 });
```

We simply need to delete the model itself with the appropriate `destroy`[116] method:

Properly deleting a contact in the `List` controller (assets/js/apps/contacts/list/list_controller.js)

```
1 contactsListView.on("itemview:contact:delete", function(childView, model){
2   model.destroy();
3 });
```

With this modification in place, when you click the "delete" button, the associated contact will actually be removed from storage.

Don't forget we put code in place to create 3 models if there are none in storage. So if you delete all the contacts and refresh the page, you'll once again see 3 contacts.

We now use persisted data, so if you delete a contact and return to the app at a later time, the deleted contact will still be gone. If you want to "reload" some contacts, simply delete all of them and our contact initialization code will run again to create new contacts.

The persisted data is tied to a given browser. If you switch browsers (even on the same computer), the stored data you will be served is likely to differ.

Git commit adapting our app to deal with persisted data:

a58fb0eb547368f583154ad2bf44201d860fe4e4[117]

[116] http://backbonejs.org/#Model-destroy

[117] https://github.com/davidsulc/marionette-gentle-introduction/commit/a58fb0eb547368f583154ad2bf44201d860fe4e4

Handling Data Latency

So far our app works as expected, but only because our data is loaded locally from web storage. Let's add some (artificial) latency to the code fetching our contact instance and see what happens when we attempt to display a contact in our `Show` sub-module.

Delaying our Contact Fetch

Let's add a 2 second delay before fetching a contact:

Adding a 2 second delay before fetching a contact (assets/js/entities/contact.js)

```
1  getContactEntity: function(contactId){
2    var contact = new Entities.Contact({id: contactId});
3    setTimeout(function(){
4      contact.fetch();
5    }, 2000);
6    return contact;
7  }
```

Now, if we navigate to "#contacts/1", we get an error indicating the `firstName` is not defined. That's due to our code in the `Show` sub-module's controller:

Our `Show.Controller` object (assets/js/apps/contacts/show/show_controller.js)

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3    Show.Controller = {
4      showContact: function(id){
5        var contact = ContactManager.request("contact:entity", id);
6        var contactView;
7        if(contact !== undefined){
8          contactView = new Show.Contact({
9            model: contact
10         });
11       }
```

```
12        else{
13          contactView = new Show.MissingContact();
14        }
15
16        ContactManager.mainRegion.show(contactView);
17      }
18    }
19  });
```

On line 5, we load our `contact` instance using `fetch`. But by the time we instantiate the view on line 8, the `contact` data hasn't yet been returned by the `fetch` method, so our model basically only has an `id` attribute defined. This `contact` instance gets passed on to create a `Show.Contact` instance, whose template is:

Template used by the `Show.Contact` view (index.html)

```
1  <script type="text/template" id="contact-view">
2    <h1><%= firstName %> <%= lastName %></h1>
3    <p><strong>Phone number:</strong> <%= phoneNumber %></p>
4  </script>
```

But as explained, the model instance that is used to fill in the template only has an `id` attribute defined, since the rest of the data hasn't yet been returned by the `fetch` call. And that is why our error is raised.

Using jQuery Deferreds

What we need is some way to wait until the `contact` data has been returned before instantiating our view. One way of achieving this is using a callback function, but that doesn't scale well: what if we need to wait for multiple data sources to be returned before displaying them? Recursively providing callbacks to callbacks starts getting very painful, very fast.

Instead, we'll use jQuery Deferreds[118], which will allow us to use a much cleaner mechanism to wait for the required data before instantiating a view. Let's use a deferred object to return a promise from our "contact:entity" handler:

[118] http://api.jquery.com/category/deferred-object/

Returning a promise (assets/js/entities/contact.js)

```
1  getContactEntity: function(contactId){
2    var contact = new Entities.Contact({id: contactId});
3    var defer = $.Deferred();
4    setTimeout(function(){
5      contact.fetch({
6        success: function(data){
7          defer.resolve(data);
8        }
9      });
10   }, 2000);
11   return defer.promise();
12 }
```

What's going on here? On line 3 we declare a `Deferred` object instance, and on line 11 we return a `promise` on that object. A deferred object is essentially "something that will happen later": it is used to synchronize code by having it react as the `promise` is updated (typically with succeess/failure).

By returning the deferred object's `promise`, we're basically saying "I promise I'll do something, and I'll update you as things progress". This contract allows code elsewhere to simply monitor the `promise` and react appropriately to any changes (e.g. fresh data coming in).

What's the difference between a deferred object and a promise?

Let's see jQuery's definitions:

- The `Deferred` object can register multiple callbacks into callback queues, invoke callback queues, and relay the success or failure state of any synchronous or asynchronous function (source[119])
- The `deferred.promise()` method allows an asynchronous function to prevent other code from interfering with the progress or status of its internal request. The `Promise` exposes only the `Deferred` methods needed to attach additional handlers or determine the state, but not ones that change the state (source[120])

You use a `Deferred` object instance where you need to be able to update the internal request's status (e.g. indicate when it's done), and you provide a `promise` to dependent code. That way, dependent code can monitor the deferred's progress, but cannot modify it. For all practical purposes, you can think of a `promise` as a *read-only* version of a `Deferred` object instance.

[119] http://api.jquery.com/category/deferred-object/

[120] http://api.jquery.com/deferred.promise/

How do we send that fresh data to the code monitoring the promise? By our call to `resolve` on line 7: when the `fetch` call succeeds, we `resolve` the deferred object, and we forward the received data.

Here's what it looks like on the other end, in our `Show` controller:

Using a promise to synchronize view display (assets/js/apps/contacts/show/show_controller.js)

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3    Show.Controller = {
4      showContact: function(id){
5        var fetchingContact = ContactManager.request("contact:entity", id);
6        $.when(fetchingContact).done(function(contact){
7          var contactView;
8          if(contact !== undefined){
9            contactView = new Show.Contact({
10             model: contact
11           });
12         }
13         else{
14           contactView = new Show.MissingContact();
15         }
16
17         ContactManager.mainRegion.show(contactView);
18       });
19     }
20   }
21 });
```

On line 5, we get the promise returned by our handler. We then wait on line 6 until the data has been fetched (`$when...done`) before displaying our view with the data.

What if you need to wait for multiple data sources? You simply add promises within the `when(...)` call. You can find an example in a post[121] on my blog.

Before we're done, we also need to manage the case where the contact doesn't exist. If it's not possible to load the requested contact, we'll simply return `undefined` in our handler:

[121] http://davidsulc.com/blog/2013/04/02/rendering-a-view-after-multiple-async-functions-return-using-promises/

Updating our promise code (assets/js/entities/contact.js)

```
getContactEntity: function(contactId){
  var contact = new Entities.Contact({id: contactId});
  var defer = $.Deferred();
  setTimeout(function(){
    contact.fetch({
      success: function(data){
        defer.resolve(data);
      },
      error: function(data){
        defer.resolve(undefined);
      }
    });
  }, 2000);
  return defer.promise();
}
```

Of course, we also need to adapt loading our entire collection to deal with latency. First, let's implement deferreds in our `getContactEntities` function:

Adding deferreds to `getContactEntities` (assets/js/entities/contact.js)

```
getContactEntities: function(){
  var contacts = new Entities.ContactCollection();
  var defer = $.Deferred();
  contacts.fetch({
    success: function(data){
      defer.resolve(data);
    }
  });
  var promise = defer.promise();
  $.when(promise).done(function(contacts){
    if(contacts.length === 0){
      // if we don't have any contacts yet, create some for convenience
      var models = initializeContacts();
      contacts.reset(models);
    }
  });
  return promise;
}
```

Using deferreds will usually look similar to this. The code in our example fetching a single contact is slightly more complicated, due to our artificial delay.

Here, we're doing the same as above: returning a promise (line 17), and resolving the deferred object with the data as soon as it's received (lines 5-7). But we're also initializing our contacts slightly differently: we use `initializeContacts` to provide us with models, and then we `reset`[122] the collection with these new models (i.e. all models in the collection are removed, and replaced by the models provided).

Any time you cause a collection to change (filtering the models to display, changing the sorting order, etc.), you can force a collection/composite view to rerender the entire collection by calling `myCollection.trigger("reset")`. This will trigger the `reset` event on `myCollection`, which the collection/composite view listens for. Marionette will then automatically get all the children views to rerender properly.

You can find the catalog of events triggered by Backbone in the documentation[123], but note that these events are triggered on Backbone entities (models and collections), and have nothing to do with the events triggered on views by Marionette (show, render, etc.). You can listen to the entity events with the same syntax as Marionette view events, e.g.: `contacts.on("add", function(...){...})`, and can naturally trigger any one of them manually by calling `trigger` as above.

Doing so allows us to demonstrate deferreds and Marionette functionality. Here's what happens when we don't have any contacts in our collection:

1. when the promise is resolved with data, our view renders the composite view with an empty collection;
2. our `initializeContacts` creates some contacts for us, and returns them;
3. we `reset` our collection with the new models, so our collection now contains the contacts we've just created;
4. our composite view automatically listens to the collection's `reset` event and rerenders all the child views, displaying the new contacts.

For this to work, we need to slightly change our `initializeContacts` to return a list of models, instead of a collection:

[122] http://backbonejs.org/#Collection-reset
[123] http://backbonejs.org/#Events-catalog

Returning a list of models from `initializeContacts` (assets/js/entities/contact.js)

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

And of course, we need to update the code in our `ContactsApp.List` controller to deal with a deferred object:

Processing a deferred object to list contacts (assets/js/apps/contacts/list/list_controller.js)

```
ContactManager.module("ContactsApp.List", function(List, ContactManager,
Backbone, Marionette, $, _){
  List.Controller = {
    listContacts: function(){
      var fetchingContacts = ContactManager.request("contact:entities");

      $.when(fetchingContacts).done(function(contacts){
        var contactsListView = new List.Contacts({
          collection: contacts
        });

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

        contactsListView.on("itemview:contact:delete",
        function(childView, model){
```

```
19            model.destroy();
20          });
21
22          ContactManager.mainRegion.show(contactsListView);
23        });
24      }
25    }
26 });
```

Our app is now properly dealing with data latency. Even though it's artificial in our case, most of your apps will probably deal with remote servers where latency must be taken into account.

Git commit dealing with latency:

048e2ee1d51e4e4644c589c1ec57ab67f80d9e23[124]

Displaying a Loading View

We currently have a 2 second delay when displaying a contact in our `Show` sub-module. Although our app behaves properly and waits for the data to be returned before displaying the view, it's not very communicative: the user has absolutely no feedback as to whether the app is actually doing anything useful while he's waiting. To address this issue, let's create a loading view to display while data is being fetched.

We'll use the spin.js[125] library, which will enable us to display a spinner animation without having to deal with images:

- get spin.js[126] and put the file in *assets/js/vendor/spin.js*
- get jquery.spin.js[127] and put the file in *assets/js/vendor/spin.jquery.js*

Notice I've renamed `jquery.spin.js` to `spin.jquery.js`. This is for organization purposes: I find it easier if the "spin.js" files stay grouped in folders by having similar names.

Let's now add these new files to `index.html`:

[124] https://github.com/davidsulc/marionette-gentle-introduction/commit/048e2ee1d51e4e4644c589c1ec57ab67f80d9e23
[125] http://fgnass.github.io/spin.js/
[126] http://fgnass.github.io/spin.js/dist/spin.js
[127] http://fgnass.github.io/spin.js/jquery.spin.js

Adding the spin.js files to index.html

```
<script src="./assets/js/vendor/jquery.js"></script>
<script src="./assets/js/vendor/json2.js"></script>
<script src="./assets/js/vendor/underscore.js"></script>
<script src="./assets/js/vendor/backbone.js"></script>
<script src="./assets/js/vendor/backbone.localstorage.js"></script>
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>
```

We'll need a template for our loading view, so let's add one to `index.html`:

Adding the loading view template (index.html)

```
<script type="text/template" id="loading-view">
  <h1>Artificial Loading Delay</h1>
  <p>Data loading is delayed to demonstrate using a loading view.</p>
  <div id="spinner"></div>
</script>
```

We need to add an element to contain our spinner, which is why we've included an empty `div`.

We'll also need a view to render our loading template. Where should we define it? Since a loading view is a common view that will be shared across the entire `ContactManager` application, let's define it within a new `Common` module (in a new file):

Defining our loading view within a `Common` module (assets/js/common/views.js)

```
ContactManager.module("Common.Views", function(Views, ContactManager,
Backbone, Marionette, $, _){
  Views.Loading = Marionette.ItemView.extend({
    template: "#loading-view",

    onShow: function(){
      var opts = {
        lines: 13, // The number of lines to draw
```

```
9           length: 20, // The length of each line
10          width: 10, // The line thickness
11          radius: 30, // The radius of the inner circle
12          corners: 1, // Corner roundness (0..1)
13          rotate: 0, // The rotation offset
14          direction: 1, // 1: clockwise, -1: counterclockwise
15          color: "#000", // #rgb or #rrggbb
16          speed: 1, // Rounds per second
17          trail: 60, // Afterglow percentage
18          shadow: false, // Whether to render a shadow
19          hwaccel: false, // Whether to use hardware acceleration
20          className: "spinner", // The CSS class to assign to the spinner
21          zIndex: 2e9, // The z-index (defaults to 2000000000)
22          top: "30px", // Top position relative to parent in px
23          left: "auto" // Left position relative to parent in px
24        };
25        $("#spinner").spin(opts);
26      }
27    });
28  });
```

Notice we've created a `Views` sub-module to contain our common views (line 1). Then, we define an `onShow` function on line 6: when a view is displayed, Marionette will trigger a "show" event and will execute the view's `onShow` function, if defined. We take advantage of this behavior to fire up our spinner automatically when the view is displayed. (To learn more about the spinner options, refer to the documentation[128].)

Let's not forget to include this new file in our `index.html`:

Including our common loading view in index.html

```
1 <script src="./assets/js/app.js"></script>
2 <script src="./assets/js/apps/config/storage/localstorage.js"></script>
3 <script src="./assets/js/entities/contact.js"></script>
4 <script src="./assets/js/common/views.js"></script>
```

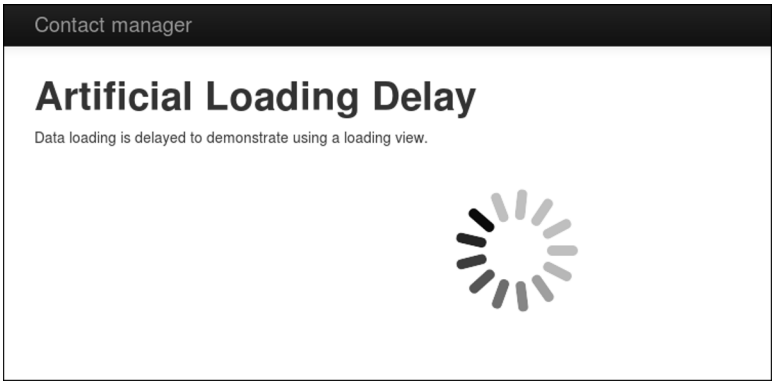
All that's left to do of course, is to display our loading view. Our app can already tell when a contact has been loaded, and will respond by displaying the contact view. So we can simply display the loading view within the main region, and Marionette will replace it with our contact view as soon as the data has been loaded. Here's what that looks like:

[128] http://fgnass.github.io/spin.js

assets/js/apps/contacts/show/show_controller.js

```
1  ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
2  Backbone, Marionette, $, _){
3    Show.Controller = {
4      showContact: function(id){
5        var loadingView = new ContactManager.Common.Views.Loading();
6        ContactManager.mainRegion.show(loadingView);
7
8        var fetchingContact = ContactManager.request("contact:entity", id);
9        $.when(fetchingContact).done(function(contact){
10         var contactView;
11         if(contact !== undefined){
12           contactView = new Show.Contact({
13             model: contact
14           });
15         }
16         else{
17           contactView = new Show.MissingContact();
18         }
19
20         ContactManager.mainRegion.show(contactView);
21       });
22     }
23   }
24 });
```

Lines 5-6 will create a new loading view and display it. When the `contact` has been fetched, the actual view we want to display is instantiated on lines 12-14, and is displayed on line 20. When Marionette displays this new view in the same region, it will automatically manage closing the view displayed in that region.



Our new loading view

The displayed view, once the contact data has loaded (after the 2 second delay)

Git commit displaying a loading view for contact display:

c1c44e1985b9e1301cf0987abe1858074eb7ae44[129]

Exercise

Implementing a Loading View for Listing the Contacts

As an exercise, try implementing a loading view for the `List` sub-module (i.e. the "#contacts" URL).

Since there's no artificial delay before we load the contacts collection, there's a good chance you won't actually see the loading view get displayed before the list of contacts is displayed.

[129] https://github.com/davidsulc/marionette-gentle-introduction/commit/c1c44e1985b9e1301cf0987abe1858074eb7ae44

Solution

(The solution is presented within the main text, as it will become part of our application's code.)

We already have a common loading view in place, and we're already using deferreds to know when the data is available. So we simply need to update our controller code to:

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 loadingView = new ContactManager.Common.Views.Loading();
6        ContactManager.mainRegion.show(loadingView);
7
8        var fetchingContacts = ContactManager.request("contact:entities");
9
10       $.when(fetchingContacts).done(function(contacts){
11         var contactsListView = new List.Contacts({
12           collection: contacts
13         });
14
15         contactsListView.on("itemview:contact:show",
16         function(childView, model){
17           ContactManager.trigger("contact:show", model.get("id"));
18         });
19
20         contactsListView.on("itemview:contact:delete",
21         function(childView, model){
22           model.destroy();
23         });
24
25         ContactManager.mainRegion.show(contactsListView);
26       });
27     }
28   }
29 });
```

All we do is display the loading view (lines 5-6), and let Marionette deal with replacing it once the data is loaded. Easy, right?

Git commit adding a loading view before listing contacts:
b5812c6f0a02083ac3832ed55f2fd1d39361370e[130]

Passing Parameters to Views and SerializeData

Let's take a step back and think about our app for a second. Our `List` controller displays a loading view saying that data is being loaded *with an artificial delay*. But that's not true! We need some way to display a different loading message for our `List` sub-module.

We could simply add a new template with a different loading message, and provide the proper template when instantiating the view (as we saw earlier). It would work, but it would be such a dirty hack you'd probably need to shower after writing it...

Given our loading view is intended to be shared across the application, a better course of action would be to provide the title and message to display when we instantiate the view. In other words, we need to be able to provide parameters to our loading view.

We'll need to store the `title` and `message` attributes in the view (if any are given) by using an `initialize` function. But how do we display them in the template? Well, Marionette calls the `serializeData` function to provide data to the template; we can write our own version to provide these new attributes:

Passing parameters to the template with `serializeData` (assets/js/common/views.js)

```
1  ContactManager.module("Common.Views", function(Views, ContactManager,
2  Backbone, Marionette, $, _){
3    Views.Loading = Marionette.ItemView.extend({
4      template: "#loading-view",
5
6      initialize: function(options){
7        var options = options || {};
8        this.title = options.title || "Loading Data";
9        this.message = options.message || "Please wait, data is loading.";
10     },
11
12     serializeData: function(){
13       return {
14         title: this.title,
15         message: this.message
16       }
```

[130] https://github.com/davidsulc/marionette-gentle-introduction/commit/b5812c6f0a02083ac3832ed55f2fd1d39361370e

```
17      },
18
19      onShow: function(){
20        var opts = {
21          // options for spin.js
22        };
23        $("#spinner").spin(opts);
24      }
25    });
26  });
```

Our initializer simply sets the title and message values, using defaults if none are given.

Prior to Backbone 1.1.0, additional attributes were automatically passed on within an `options` object, making this initilizer code necessary only from Backbone 1.1.0 and above.

All we're doing within `serializeData` on lines 14-15 is returning a JSON object with the attribute keys we want to access within the template. And here's the new version of our template:

Displaying view parameters in our template (index.html)

```
1  <script type="text/template" id="loading-view">
2    <h1><%= title %></h1>
3    <p><%= message %></p>
4    <div id="spinner"></div>
5  </script>
```

The loading view displayed by our `List` controller is now nice and generic: since we don't pass any `title` or `message` parameters, it falls back onto the generic message. And we can still display a message indicating data is being purposely delayed within our `Show` controller, by passing the desired `title` and `message` parameters:

Passing view parameters on instantiation (assets/js/apps/contacts/show/show_controller.js)

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
Backbone, Marionette, $, _){
  Show.Controller = {
    showContact: function(id){
      var loadingView = new ContactManager.Common.Views.Loading({
        title: "Artificial Loading Delay",
        message: "Data loading is delayed to
                  demonstrate using a loading view."
      });
      ContactManager.mainRegion.show(loadingView);

      // rest of showContact is unchanged

    }
  }
});
```

Git commit passing parameters to the loading view:
bd7ba386b6e6a97b9e90975c98bf30bdf4529eac[131]

[131] https://github.com/davidsulc/marionette-gentle-introduction/commit/bd7ba386b6e6a97b9e90975c98bf30bdf4529eac

Managing Forms: Editing a Contact

Let's create a dedicated view to edit a contact. The entire process is very similar to what we saw when displaying a contact, so we'll go over the steps quite quickly. We'll start by creating the new template (styled with Bootstrap):

Template for the contact form (index.html)

```
<script type="text/template" id="contact-form">
  <h1>Edit <%= firstName %> <%= lastName %></h1>
  <form>
    <div class="control-group">
      <label for="contact-firstName" class="control-label">
          First name:</label>
      <input id="contact-firstName" name="firstName"
          type="text" value="<%= firstName %>"/>
    </div>
    <div class="control-group">
      <label for="contact-lastName" class="control-label">
          Last name:</label>
      <input id="contact-lastName" name="lastName"
          type="text" value="<%= lastName %>"/>
    </div>
    <div class="control-group">
      <label for="contact-phoneNumber" class="control-label">
          Phone number:</label>
      <input id="contact-phoneNumber" name="phoneNumber"
          type="text" value="<%= phoneNumber %>"/>
    </div>
    <button class="btn js-submit">Save</button>
  </form>
</script>
```

We've added `id` attributes to the form fields using the convention of `contact-` (i.e. the model type) followed by the attribute name. Using such a convention will be useful later, as we'll need to determine the ids programmatically in order to display their corresponding error messages.

Then, we'll need a view to render this template. We create this view within a new `ContactsApp.Edit` sub-module:

The edit view, inside the `ContactsApp.Edit` sub-module (assets/js/apps/contacts/edit/edit_view.js)

```
ContactManager.module("ContactsApp.Edit", function(Edit,
ContactManager, Backbone, Marionette, $, _){
  Edit.Contact = Marionette.ItemView.extend({
    template: "#contact-form",

    events: {
      "click button.js-submit": "submitClicked"
    },

    submitClicked: function(e){
      e.preventDefault();
      console.log("edit contact");
    }
  });
});
```

We'll also need a controller to get the data and display the view:

The `Edit.Controller` (assets/js/apps/contacts/edit/edit_controller.js)

```
ContactManager.module("ContactsApp.Edit", function(Edit,
ContactManager, Backbone, Marionette, $, _){
  Edit.Controller = {
    editContact: function(id){
      var loadingView = new ContactManager.Common.Views.Loading({
        title: "Artificial Loading Delay",
        message: "Data loading is delayed to demonstrate
                  using a loading view."
      });
      ContactManager.mainRegion.show(loadingView);

      var fetchingContact = ContactManager.request("contact:entity", id);
      $.when(fetchingContact).done(function(contact){
        var view;
        if(contact !== undefined){
          view = new Edit.Contact({
```

```
            model: contact
          });
        }
        else{
          view = new ContactManager.ContactsApp.Show.MissingContact();
        }

        ContactManager.mainRegion.show(view);
      });
    }
  };
});
```

Of course, we need to include these 2 new files in `index.html`:

Including the Edit sub-module files (index.html)

```
<script src="./assets/js/apps/contacts/contacts_app.js"></script>
<script src="./assets/js/apps/contacts/list/list_view.js"></script>
<script src="./assets/js/apps/contacts/list/list_controller.js"></script>
<script src="./assets/js/apps/contacts/show/show_view.js"></script>
<script src="./assets/js/apps/contacts/show/show_controller.js"></script>
<script src="./assets/js/apps/contacts/edit/edit_view.js"></script>
<script src="./assets/js/apps/contacts/edit/edit_controller.js"></script>
```

And finally, we need our routing controller to manage this new route:

Updating the `ContactsApp` routing controller to manage the edit route (assets/js/apps/contacts/contacts_app.js)

```
ContactManager.module("ContactsApp", function(ContactsApp, ContactManager,
Backbone, Marionette, $, _){
  ContactsApp.Router = Marionette.AppRouter.extend({
    appRoutes: {
      "contacts": "listContacts",
      "contacts/:id": "showContact",
      "contacts/:id/edit": "editContact"
    }
  });

  var API = {
```

```
    // listContacts and showContact functions

    editContact: function(id){
      ContactsApp.Edit.Controller.editContact(id);
    }
  };

  // handlers for "contacts:list" and "contact:show"

  ContactManager.on("contact:edit", function(id){
    ContactManager.navigate("contacts/" + id + "/edit");
    API.editContact(id);
  });

  ContactManager.addInitializer(function(){
    new ContactsApp.Router({
      controller: API
    });
  });
});
```

When we refresh the page and enter "#contacts/1/edit", we'll see the form to edit the contact with `id` 1. And if we click the "Save" button, we'll see our "edit contact" message in the console output. So far, so good!

Let's now add an "edit" button to the template used to display a contact:

Adding an 'edit' link to our contact display template (index.html)

```
<script type="text/template" id="contact-view">
  <h1><%= firstName %> <%= lastName %></h1>
  <a href="#contacts/<%= id %>/edit" class="btn btn-small js-edit">
    <i class="icon-pencil"></i>
    Edit this contact
  </a>
  <p><strong>Phone number:</strong> <%= phoneNumber %></p>
</script>
```

We'll need to adapt our view to pass the event on to the controller:

Handling the 'edit' link's click event (assets/js/apps/contacts/show/show_view.js)

```
1  Show.Contact = Marionette.ItemView.extend({
2    template: "#contact-view",
3
4    events: {
5      "click a.js-edit": "editClicked"
6    },
7
8    editClicked: function(e){
9      e.preventDefault();
10     this.trigger("contact:edit", this.model);
11   }
12 });
```

And in our controller, we'll trigger the event our routing controller listens for (lines 13-15):

Triggering the routing event (assets/js/apps/contacts/show/show_controller.js)

```
1  Show.Controller = {
2    showContact: function(id){
3      // display loading view
4
5      var fetchingContact = ContactManager.request("contact:entity", id);
6      $.when(fetchingContact).done(function(contact){
7        var contactView;
8        if(contact !== undefined){
9          contactView = new Show.Contact({
10           model: contact
11         });
12
13         contactView.on("contact:edit", function(contact){
14           ContactManager.trigger("contact:edit", contact.get("id"));
15         });
16       }
17       else{
18         contactView = new Show.MissingContact();
19       }
20
21       ContactManager.mainRegion.show(contactView);
```

```
22      });
23    }
24  }
```

We've now got our navigation set up to get to the dedicated page to edit a contact.

Git commit adding the edit view, along with navigation:
494c86c37428a3405036202df4f075ac51da5e8b[132]

Saving the Modified Contact

Before we can save our contact's updated attributes, we need to extract them from the form. Luckily, Derick Bailey built another Backbone plugin to save us time: Backbone.Syphon[133]. Grab it from here[134] and save it in *assets/js/vendor/backbone.syphon.js*. Don't forget to add it to `index.html`:

Including Backbone.Syphon **in index.html**

```
<script src="./assets/js/vendor/jquery.js"></script>
<script src="./assets/js/vendor/json2.js"></script>
<script src="./assets/js/vendor/underscore.js"></script>
<script src="./assets/js/vendor/backbone.js"></script>
<script src="./assets/js/vendor/backbone.syphon.js"></script>
<script src="./assets/js/vendor/backbone.localstorage.js"></script>
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>
```

Let's change our view to trigger an event with the form's data:

[132] https://github.com/davidsulc/marionette-gentle-introduction/commit/494c86c37428a3405036202df4f075ac51da5e8b
[133] https://github.com/derickbailey/backbone.syphon
[134] https://raw.github.com/derickbailey/backbone.syphon/master/lib/backbone.syphon.js

Triggering a 'form:submit event' with the form's data (assets/js/apps/contacts/edit/edit_view.js)

```
1 submitClicked: function(e){
2   e.preventDefault();
3   var data = Backbone.Syphon.serialize(this);
4   this.trigger("form:submit", data);
5 }
```

As you can see on line 3, `Backbone.Syphon` does a whole bunch of work for us: all we need to do is provide a reference to the view instance containing the form. Then, `Syphon` will serialize the data per the documentation[135]:

> The default behavior for serializing fields is to use the field's "name" attribute as the key in the serialized object.

All that remains to be done is for our controller to listen to the event triggered on line 4 (above):

Handling the 'form:submit' event in the `Edit` controller (assets/js/apps/contacts/edit/edit_controller.js)

```
1  // within the `done` callback
2  if(contact !== undefined){
3    view = new Edit.Contact({
4      model: contact
5    });
6
7    view.on("form:submit", function(data){
8      contact.save(data);
9      ContactManager.trigger("contact:show", contact.get("id"));
10   });
11 }
```

We simply add a listener and save the contact (on line 8) with the attributes syphoned from the form (and provided by the triggered event). After updating the contact, we display the `show` view by triggering a routing event on line 9. If we enter "#contacts" in the address bar, we can see the contact has been properly updated and persisted.

[135] https://github.com/derickbailey/backbone.syphon/

Validating Data

As it stands, we can edit a contact and completely erase their first and last names. Our app will happily process the change, but that's not the behavior we want to have. Instead, we want to make sure *all* contacts have first and last names, and that last names have at least 2 letters in them. In other words, we need to validate our contact data.

To validate our contacts, we need to define a `validate`[136] function on our model. This function will always be called before the model is saved, to ensure the data is valid. If the data is invalid, `validate` will return an array of errors (inspired by Rails):

Adding a `validate` method to our `Contact` model (assets/js/entities/contact.js)

```
1  Entities.Contact = Backbone.Model.extend({
2    urlRoot: "contacts",
3
4    validate: function(attrs, options) {
5      var errors = {}
6      if (! attrs.firstName) {
7        errors.firstName = "can't be blank";
8      }
9      if (! attrs.lastName) {
10       errors.lastName = "can't be blank";
11     }
12     else{
13       if (attrs.lastName.length < 2) {
14         errors.lastName = "is too short";
15       }
16     }
17     if( ! _.isEmpty(errors)){
18       return errors;
19     }
20   }
21 });
```

Never trust client-side data! All client-side data can be tampered with and the server should **always** validate incoming data. Client-side validation should only be implemented *in addition* to server-side validation, to provide quicker error feedback to the user.

[136] http://backbonejs.org/#Model-validate

For more advanced validation needs, look into plugins such as backbone.validation[137]

In our controller, let's do the following:

- if the data is valid, we redirect to the "show" view
- if the data is not valid, we display an alert

Adapting behavior to data validity (assets/js/apps/contacts/edit/edit_controller.js)

```
1 view.on("form:submit", function(data){
2   if(contact.save(data)){
3     ContactManager.trigger("contact:show", contact.get("id"));
4   }
5   else{
6     alert("Unable to save data!");
7   }
8 });
```

If we try saving a contact with an empty first name, the alert is displayed. Great!

Displaying Errors in the Form

Now, let's change the controller to trigger a method on our view, which we'll use later to display the validation errors within the form:

Triggering a method on our view (assets/js/apps/contacts/edit/edit_controller.js)

```
1 view.on("form:submit", function(data){
2   if(contact.save(data)){
3     ContactManager.trigger("contact:show", contact.get("id"));
4   }
5   else{
6     view.triggerMethod("form:data:invalid", contact.validationError);
7   }
8 });
```

Of course, we need to define the associated method in our view:

[137] https://github.com/thedersen/backbone.validation

Defining the triggered method in our view (assets/js/apps/contacts/edit/edit_view.js)

```
1 Edit.Contact = Marionette.ItemView.extend({
2   // template and events attributes
3
4   // submitClicked function
5
6   onFormDataInvalid: function(errors){
7     console.log("invalid form data: ", errors);
8   }
9 });
```

Remember that `triggerMethod`[138] will automatically execute a function whose name corresponds to the event. There's nothing else for us to do: provided we name the function correctly, Marionette will execute it.

Try saving invalid data again (after refreshing, of course), and you'll see the message in the console. We now need to update our form with the various error messages. Since we receive an array containing all the error messages, we'll simply iterate over the array, updating the form field associated with each error. We'll use Underscore's `each`[139] method, so we'll need to provide an iterator function. Here it goes:

Displaying error messages (assets/js/apps/contacts/edit/edit_view.js)

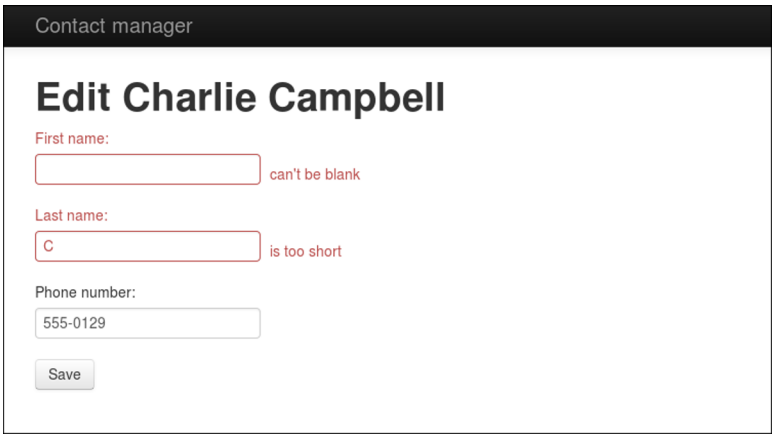
```
1 onFormDataInvalid: function(errors){
2   var self = this;
3   var markErrors = function(value, key){
4     var $controlGroup = self.$el.find("#contact-" + key).parent();
5     var $errorEl = $("<span>", {class: "help-inline error", text: value});
6     $controlGroup.append($errorEl).addClass("error");
7   }
8   _.each(errors, markErrors);
9 }
```

Our iterator function is defined on lines 3-7, and is provided each error in turn by the call on line 8. Here's what happens within the `markErrors` iterator:

[138] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.functions.md#marionettetriggermethod
[139] http://underscorejs.org/#each

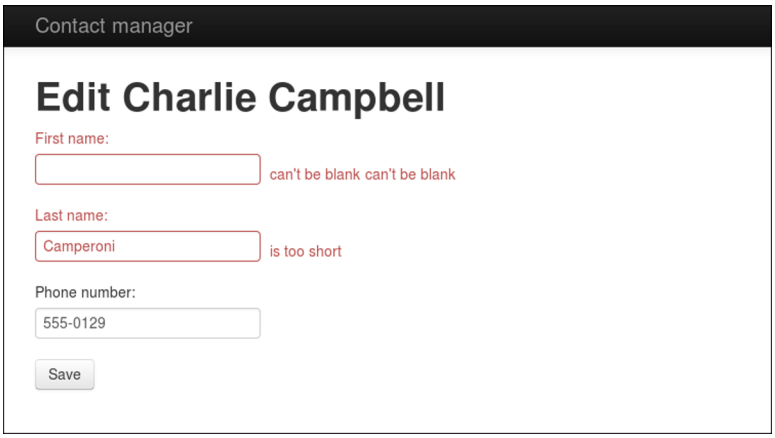
1. get the "control group" Bootstrap uses for the label/input group (line 4);
2. create a DOM element for the error message and style it with Bootstrap (line 5);
3. add the error DOM element to the form's control group (line 6).

Here's what our form looks like with errors:



A form with errors

So far, so good! Let's correct that second error by entering a longer last name: Camperoni. Take a look a our form now:



Incorrect error message display

As you can see, none of the errors display correctly: the first error is displayed twice, while the second one shouldn't be displayed at all. Where's the problem coming from? It's simply due to the fact we're not dealing with the typical stateless web development environment: since we're not refreshing the page each time, any error message we don't remove "manually" will remain on display.

Therefore, the solution to our problem is quite simple: remove all error messages from the form before displaying the new validation errors.

Clearing the visible errors before displaying the new ones (assets/js/apps/contacts/edit/edit_view.js)

```
1  onFormDataInvalid: function(errors){
2    var $view = this.$el;
3
4    var clearFormErrors = function(){
5      var $form = $view.find("form");
6      $form.find(".help-inline.error").each(function(){
7        $(this).remove();
8      });
9      $form.find(".control-group.error").each(function(){
10       $(this).removeClass("error");
11     });
12   }
13
14   var markErrors = function(value, key){
15     var $controlGroup = $view.find("#contact-" + key).parent();
16     var $errorEl = $("<span>", {class: "help-inline error", text: value});
17     $controlGroup.append($errorEl).addClass("error");
18   }
19
20   clearFormErrors();
21   _.each(errors, markErrors);
22 }
```

In our `clearFormErrors` function, we first locate the form within the view, on line 5. Lines 6-8 remove the error messages we've added to the DOM, and lines 9-11 remove the "error" class from the input control groups, which made Bootstrap give red outlines to the inputs. Now our error display works properly!

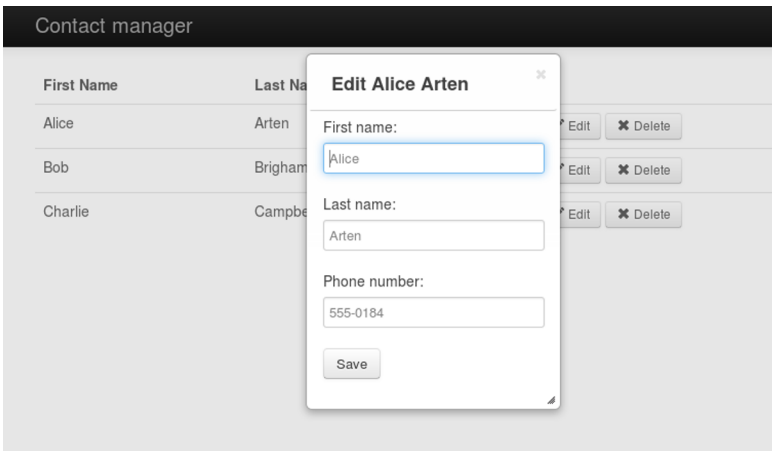
Git commit for data validation and error display:

d5f1112c869aaa2a12f2ee2ab2b03c028e0e05ba[140]

[140] https://github.com/davidsulc/marionette-gentle-introduction/commit/d5f1112c869aaa2a12f2ee2ab2b03c028e0e05ba

Displaying a Modal Window

We've already got a dedicated view to edit our contacts, but now we'll implement a modal view that will enable users to edit contacts directly from the page listing all contacts:



A modal window to edit a contact

This will enable us to learn not only about managing modal windows, but also how to refactor our code to centralize shared functionality (the forms, error display, etc.).

Using jQuery UI

We'll be using jQuery UI[141] to display the modal window, along with the jQuery UI Bootstrap theme[142], so let's get the files we need:

- get the jQuery UI files (without theme) from here[143]. Within the downloaded zip file, copy *js/jquery-ui-1.10.3.custom.js* into *assets/js/vendor/jquery-ui-1.10.3.js* (note we've remove the "custom" portion of the filename)
- download the Bootstrap theme for jQuery UI from here[144]. Within the downloaded zip file, copy *css/custom-theme/jquery-ui-1.10.0.custom.css* to *assets/css/jquery-ui-1.10.0.custom.css*. Copy the *css/custom-theme/images* folder to *assets/css* (i.e. you'll end up with an *images* folder within the *css* folder)

Of course, include these new files in your index.html:

[141] http://jqueryui.com/
[142] http://addyosmani.github.io/jquery-ui-bootstrap/
[143] http://jqueryui.com/download/#!themeParams=none
[144] http://addyosmani.github.io/jquery-ui-bootstrap/

Including jQuery UI and related CSS in index.html

```
<head>
  <meta charset="utf-8">
  <title>Marionette Contact Manager</title>
  <link href="./assets/css/bootstrap.css" rel="stylesheet">
  <link href="./assets/css/application.css" rel="stylesheet">
  <link href="./assets/css/jquery-ui-1.10.0.custom.css" rel="stylesheet">
</head>

<body>

  <!-- regions, templates, etc. -->

  <script src="./assets/js/vendor/jquery.js"></script>
  <script src="./assets/js/vendor/jquery-ui-1.10.3.js"></script>
  <script src="./assets/js/vendor/json2.js"></script>
  <!-- rest of javascript includes -->

  <!-- code starting our app -->
</body>
```

Adding the Edit Link

On our index page, we need an "edit" link to trigger the modal window with the edit form. So let's add the link to the template:

Adding an 'edit' link to our template (index.html)

```
<script type="text/template" id="contact-list-item">
  <td><%= firstName %></td>
  <td><%= lastName %></td>
  <td>
    <a href="#contacts/<%= id %>" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <a href="#contacts/<%= id %>/edit" class="btn btn-small js-edit">
      <i class="icon-pencil"></i>
```

```
      Edit
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

Although we're going to use the "edit" link to display a modal window, we've set an `href` value to point to the edit page. That way, if the user decides to open the "edit" link in a new browser tab, he will be able to edit the contact as expected.

We also need our view to process the `click` event:

Processing a click on the 'edit' link (assets/js/apps/contacts/list/list_view.js)

```
events: {
  "click": "highlightName",
  "click td a.js-show": "showClicked",
  "click td a.js-edit": "editClicked",
  "click button.js-delete": "deleteClicked"
},

// highlightName and showClicked functions

editClicked: function(e){
  e.preventDefault();
  e.stopPropagation();
  this.trigger("contact:edit", this.model);
},
```

Finally, our `List` controller must react when the "edit" link is clicked:

Reacting when the 'edit' link is clicked (assets/js/apps/contacts/list/list_controller.js)

```
// "itemview:contact:show" handler

contactsListView.on("itemview:contact:edit", function(childView, model){
  console.log("edit link clicked");
});

// "itemview:contact:delete" handler
```

So far, we're properly handling the user's click on the "edit" link. Now, we need to implement displaying the modal view containing the edit form.

Implementing Modal Functionality

If we're to display modal windows, we'll need a region to show them. Let's add a DOM element for our modal dialogs (line 6):

Adding a DOM element to contain our modals dialogs (index.html)

```
<div id="main-region" class="container">
  <p>Here is static content in the web page. You'll notice that it gets
     replaced by our app as soon as we start it.</p>
</div>

<div id="dialog-region"></div>
```

We also need Marionette to configure this new DOM element as a region within our application:

Declaring the dialog region (assets/js/app.js)

```
ContactManager.addRegions({
  mainRegion: "#main-region",
  dialogRegion: "#dialog-region"
});
```

Let's now display our edit form in a modal window when the user clicks the "edit" link in our `List` view:

Displaying the edit form in a modal window (assets/js/apps/contacts/list/list_controller.js)

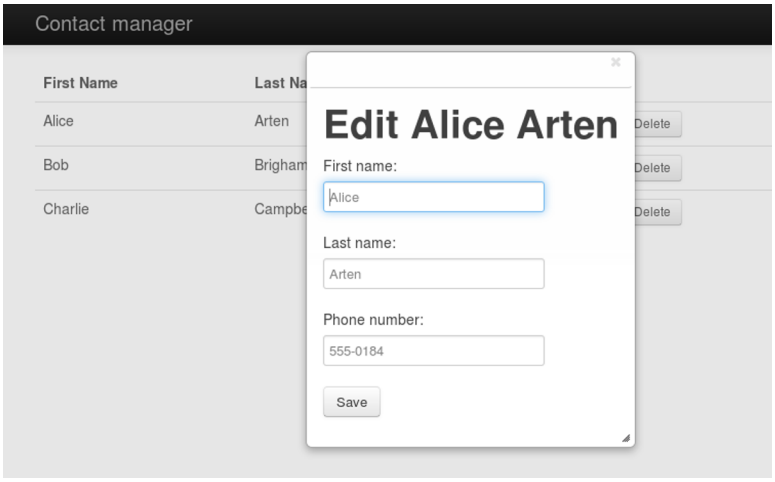
```
1  // "itemview:contact:show" handler
2
3  contactsListView.on("itemview:contact:edit", function(childView, model){
4    var view = new ContactManager.ContactsApp.Edit.Contact({
5      model: model
6    });
7
8    view.on("show", function(){
9      this.$el.dialog({
10       modal: true,
11       width: "auto"
12     });
13   });
14
15   ContactManager.dialogRegion.show(view);
16 });
17
18 // "itemview:contact:delete" handler
```

What's happening? We're reusing our `Edit.Contact` view from the `Edit` sub-module and displaying it within a modal. This is achieved by calling jQuery UI's `dialog` method on the view's element (see more about the dialog options in the documentation[145]). Take a look at how this behavior is organized:

1. create a view instance (lines 4-6);
2. add a callback to turn the view into a modal window when it is shown (lines 8-13);
3. show the view within the `dialogRegion`, which will trigger the code making it a modal window (line 15).

Notice we haven't declared or triggered a "show" event anywhere: Marionette automatically triggers that event when a view is displayed. So now we've got a functional modal view, but it looks a bit weird:

[145] http://api.jqueryui.com/dialog/



Our modal window

We're displaying a title for the modal view, but it really looks out of place. However, we *do* need the title to be displayed when we're on the *dedicated* edit page (e.g. at URL "#contacts/2"). We've seen previously that it's possible to provide parameters to views. So let's use a parameter to indicate when the edit view is being displayed as a modal view. This way, we can remove the title from the template, and insert it in the view if we're *not* displaying it as a modal view.

Let's start by removing the title from the template:

Contact form template, with title removed (index.html)

```
<script type="text/template" id="contact-form">
  <form>
    <div class="control-group">
      <label for="contact-firstName" class="control-label">
          First name:</label>
      <input id="contact-firstName" name="firstName"
          type="text" value="<%= firstName %>"/>
    </div>
    <div class="control-group">
      <label for="contact-lastName" class="control-label">
          Last name:</label>
      <input id="contact-lastName" name="lastName"
          type="text" value="<%= lastName %>"/>
    </div>
    <div class="control-group">
      <label for="contact-phoneNumber" class="control-label">
          Phone number:</label>
      <input id="contact-phoneNumber" name="phoneNumber"
```

```
19            type="text" value="<%= phoneNumber %>"/>
20      </div>
21      <button class="btn js-submit">Save</button>
22    </form>
23  </script>
```

When we instantiate the view, let's specify it's being displayed as a modal with the attribute provided on line 6:

Indicating the 'edit' form is modal (assets/js/apps/contacts/list/list_controller.js)

```
1  // "itemview:contact:show" handler
2
3  contactsListView.on("itemview:contact:edit", function(childView, model){
4    var view = new ContactManager.ContactsApp.Edit.Contact({
5      model: model,
6      asModal: true
7    });
```

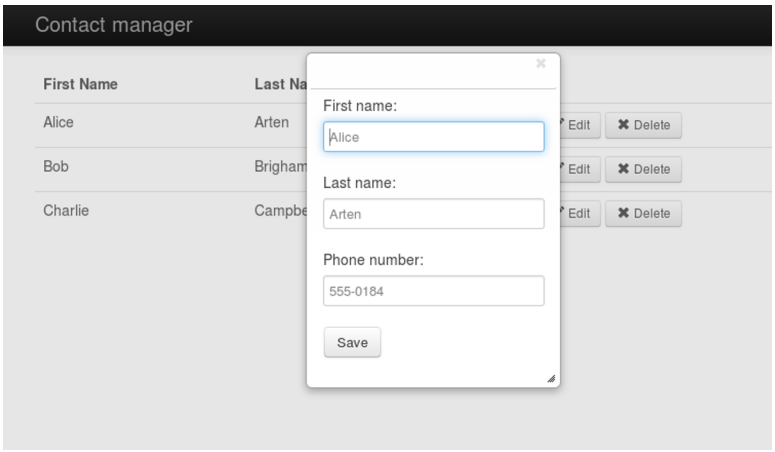
And now, within the edit view, if the `asModal` attribute is not provided or is false, we want to insert the title. How should we do this? We'll simply create a `title` attribute when the view is rendered, and insert it within the view by defining an `onRender` function. However, since the title should only be inserted when the view is *not* displayed as a modal, we need to make sure `asModal` is absent or `false`. Here's what our `onRender` looks like:

Defining `onRender` (assets/js/apps/contacts/edit/edit_view.js)

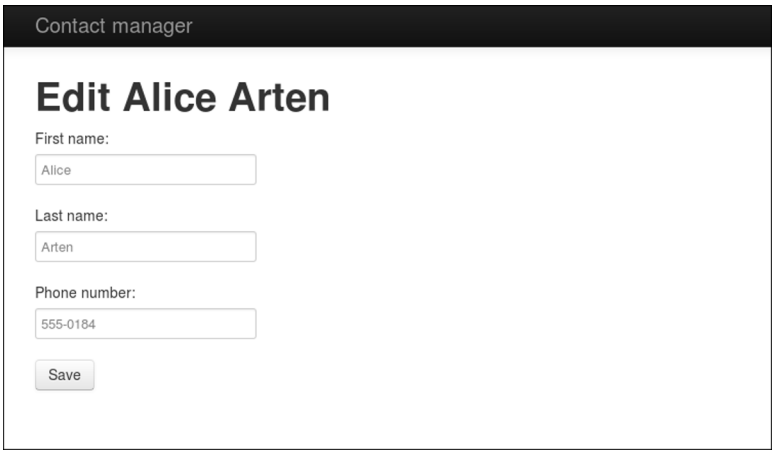
```
1  Edit.Contact = Marionette.ItemView.extend({
2    template: "#contact-form",
3
4    initialize: function(){
5      this.title = "Edit " + this.model.get("firstName");
6      this.title += " " + this.model.get("lastName");
7    },
8
9    // events and handlers
10
11   onRender: function(){
12     if( ! this.options.asModal){
13       var $title = $("<h1>", { text: this.title });
```

```
14        this.$el.prepend($title);
15      }
16    },
17
18    // onFormDataInvalid
19  });
```

Thanks to this bit of code, our modal window doesn't display a title any more, but the dedicated "edit" view still does:



Modal edit form: no title displayed



Dedicated edit form: title properly displayed

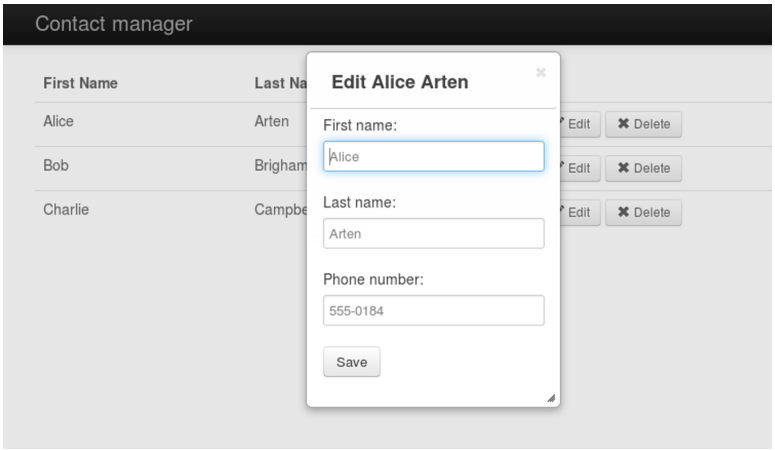
Let's add a title to our modal window, to give our users some context on what they're seeing. To do so, we'll pass the view's `title` attribute (defined above) to the call to `dialog` (per the documentation[146]):

[146] http://api.jqueryui.com/dialog/

Adding a title to the modal window (assets/js/apps/contacts/list/list_controller.js)

```
1  // "itemview:contact:show" handler
2
3  contactsListView.on("itemview:contact:edit", function(childView, model){
4    var view = new ContactManager.ContactsApp.Edit.Contact({
5      model: model
6    });
7
8    view.on("show", function(){
9      this.$el.dialog({
10       modal: true,
11       title: view.title,
12       width: "auto"
13     });
14   });
15
16   ContactManager.dialogRegion.show(view);
17 });
18
19 // "itemview:contact:delete" handler
```

Our modal now looks much better:



Displaying a title in the modal form

But wait, there's more! We can move the code creating the modal (lines 8-14) to the view itself: it belongs there, since it's really view-processing code. As you may remember, when Marionette triggers an event, it also executes the related function if it's defined on the object. So the code we have running on the "show" event, can be moved within an `onShow` function. But we only want this code to be run when we want to create a modal window, so once again we check the `asModal` option:

Defining onShow **(assets/js/apps/contacts/edit/edit_view.js)**

```
Edit.Contact = Marionette.ItemView.extend({
  template: "#contact-form",

  initialize: function(){
    this.title = "Edit " + this.model.get("firstName");
    this.title = this.title + " " + this.model.get("lastName");
  },

  // events and handlers

onRender: function(){
  if( ! this.options.asModal){
    var $title = $("<h1>", { text: this.title });
    this.$el.prepend($title);
  }
},

onShow: function(){
  if(this.options.asModal){
    this.$el.dialog({
      modal: true,
      title: this.title,
      width: "auto"
    });
  }
},

  // onFormDataInvalid
});
```

By defining the onShow function within the view, we can clean up our controller code:

Cleaned up controller code (assets/js/apps/contacts/list/list_controller.js)

```
// "itemview:contact:show" handler

contactsListView.on("itemview:contact:edit", function(childView, model){
  var view = new ContactManager.ContactsApp.Edit.Contact({
    model: model,
    asModal: true
  });

  ContactManager.dialogRegion.show(view);
});

// "itemview:contact:delete" handler
```

Shouldn't we clean this up more?

After all, displaying modals is a pretty widespread need within an application. So why would we want to duplicate the modal-rendering view across the various views we want to display as modals? Short answer: we don't. That's why we'll later cover creating a dedicated region to display modals that will abstract away all the modal-rendering code. Hang in there!

Handling the Modal Form Data

We still need to process the form and update the model, or display errors within the form. Displaying the errors on the form is pretty straightforward, since we just trigger the view's `onFormDataInvalid` method:

Displaying form errors (assets/js/apps/contacts/list/list_controller.js)

```
contactsListView.on("itemview:contact:edit", function(childView, model){
  var view = new ContactManager.ContactsApp.Edit.Contact({
    model: model,
    asModal: true
  });

  view.on("form:submit", function(data){
```

```
8       if(model.save(data)){
9         // code to be determined...
10      }
11      else{
12        view.triggerMethod("form:data:invalid", model.validationError);
13      }
14    });
15
16    ContactManager.dialogRegion.show(view);
17  });
```

What should happen when the contact is updated with no errors (line 9)? We need to update that contact's row within the composite view, and we'll also need to close the modal form:

Rendering the updated contact (assets/js/apps/contacts/list/list_controller.js)

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

On line 9, we render the child item view again. Since the view gets rerendered, the model's data is serialized and provided to the template again. In other words, we'll see the new data values displayed. Then, on line 10, we close the `dialogRegion` and Marionette takes care of closing the view it contains.

Since we're on a roll, let's add a final touch: highlighting the updated contact. Let's add a simple method to our item view, which will animate a low-tech version of a flash by toggling a Bootstrap CSS class on the table row:

Adding a `flash` method on the contact item view (assets/js/apps/contacts/list/list_view.js)

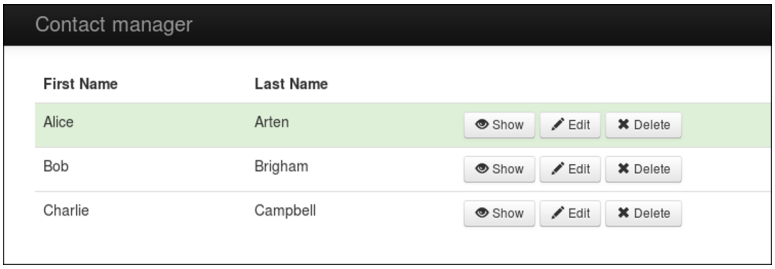
```
List.Contact = Marionette.ItemView.extend({
  // tagName, events, etc.

  flash: function(cssClass){
    var $view = this.$el;
    $view.hide().toggleClass(cssClass).fadeIn(800, function(){
      setTimeout(function(){
        $view.toggleClass(cssClass)
      }, 500);
    });
  },

  // event handlers
});
```

Now, we can simply call that method from our controller code:

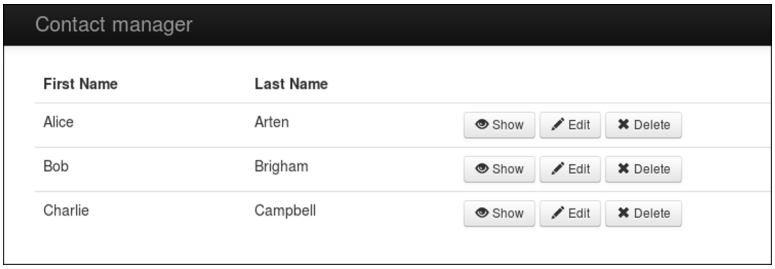
Triggering the `flash` on the updated contact (assets/js/apps/contacts/list/list_controller.js)

```
view.on("form:submit", function(data){
  if(model.save(data)){
    childView.render();
    ContactManager.dialogRegion.close();
    childView.flash("success");
  }
  else{
    view.triggerMethod("form:data:invalid", model.validationError);
  }
});
```

Here's what that looks like:



Right after updating the contact



Once the animation is over

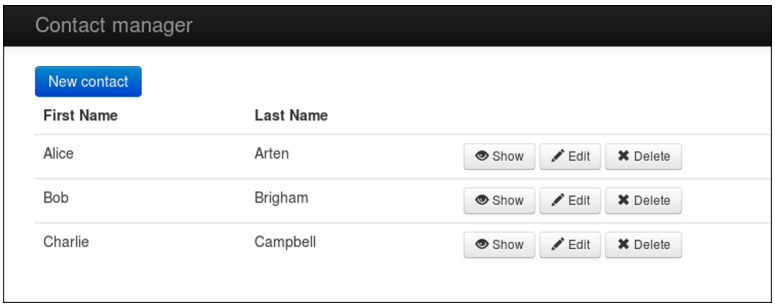
Git commit for editing a contact from a modal window:

d785c7b5e3078faa8000f269955e25c1f1d6dfc2[147]

[147] https://github.com/davidsulc/marionette-gentle-introduction/commit/d785c7b5e3078faa8000f269955e25c1f1d6dfc2

Subdividing Complex Views with Layouts

Now that we've got all the other CRUD functionality working, let's enable our app's users to create new contacts. We'll start with the visual modifications: we'll need a new area in our `List` view, so we can display a button to create new contacts. Something like this:



To accomplish this, we'll use a Marionnete `layout`[148]. According to the documentation, layouts are used

> for rendering an application layout with multiple sub-regions to be managed by specified region managers.
>
> A layout manager can also be used as a composite-view to aggregate multiple views and sub-application areas of the screen where multiple region managers need to be attached to dynamically rendered HTML.

This might seem like overkill at first: we just want a simple button, which could simply be added to the existing composite view. That's entirely correct, but what we really want to achieve is having a "control panel" above the contacts list which will get more functionality later. And in my experience, taking shortcuts by thinking "it's just a little addition" is precisely how apps tend to grow into huge, unmanageable tangles…

So let's get started with our new button! We'll need 2 new templates in `index.html`: one for the layout, and one for the view with our button. Let's add them:

[148] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.layout.md

Adding the layout and 'button view' templates (index.html)

```
1  <script type="text/template" id="contact-list-layout">
2    <div id="panel-region"></div>
3    <div id="contacts-region"></div>
4  </script>
5
6  <script type="text/template" id="contact-list-panel">
7    <button class="btn btn-primary js-new">New contact</button>
8  </script>
```

As you can see, our layout template on lines 1-4 has some `divs` with `id` attributes, which we'll use later to contain our layout's regions.

As usual, we need views to render our templates; and since they're part of the `List` display, we'll add them to that module:

Adding views to the `List` module (assets/js/apps/contacts/list/list_view.js)

```
1  ContactManager.module("ContactsApp.List", function(List, ContactManager,
2  Backbone, Marionette, $, _){
3    List.Layout = Marionette.Layout.extend({
4      template: "#contact-list-layout",
5
6      regions: {
7        panelRegion: "#panel-region",
8        contactsRegion: "#contacts-region"
9      }
10   });
11
12   List.Panel = Marionette.ItemView.extend({
13     template: "#contact-list-panel"
14   });
15
16   // Contact and Contacts views
17 });
```

We can now see how we define our regions within the layout (lines 6-9): we provide DOM ids that are present within our template. You'll note this is similar to the way we've defined our app's regions earlier.

Now that we have views and their associated templates, we still need to instantiate our layout and its views, before displaying the whole lot. Let's update our controller to use these new views:

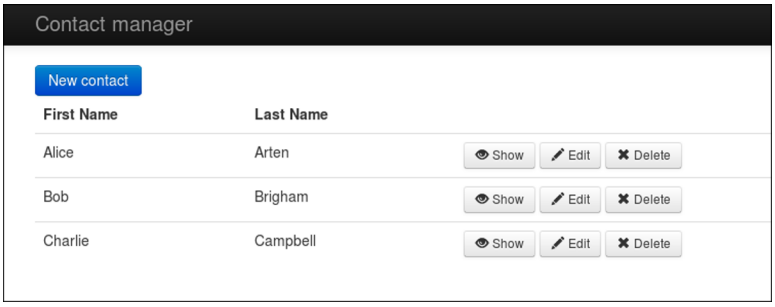
Updating the controller to use a layout (assets/js/apps/contacts/list/list_controller.js)

```
1  ContactManager.module("ContactsApp.List", function(List, ContactManager,
2  Backbone, Marionette, $, _){
3    List.Controller = {
4      listContacts: function(){
5        var loadingView = new ContactManager.Common.Views.Loading();
6        ContactManager.mainRegion.show(loadingView);
7
8        var fetchingContacts = ContactManager.request("contact:entities");
9
10       var contactsListLayout = new List.Layout();
11       var contactsListPanel = new List.Panel();
12
13       $.when(fetchingContacts).done(function(contacts){
14         var contactsListView = new List.Contacts({
15           collection: contacts
16         });
17
18         contactsListLayout.on("show", function(){
19           contactsListLayout.panelRegion.show(contactsListPanel);
20           contactsListLayout.contactsRegion.show(contactsListView);
21         });
22
23         // event handlers for the contactsListView are here
24
25         ContactManager.mainRegion.show(contactsListLayout);
26       });
27     }
28   }
29 });
```

What did we change? We instantiate our new views on lines 10-11. If you're paying attention, you'll have noticed that they're outside the `$.when` block. That's because there's no need to wait for the `contacts` data to be available before instantiating these views: they don't depend on the `contacts` data in any way.

On lines 18-21, we instruct our layout to display the proper views within the previously declared regions, as soon as the layout itself is displayed. Finally, we display the layout on line 25 (which in turn makes our other views be displayed).

And now our app looks just as we wanted it to:



Git commit using a layout:

d84e9308294420e0470a4f6f1e4e08255bce1b55[149]

Regions vs Layouts

Regions and layouts are similar, but different: both have "areas" where you can display things, but they serve different purposes.

Regions are areas in your application that will remain displayed (semi-)permanently as the user navigates. For example, you would use a region to display the navigation menu.

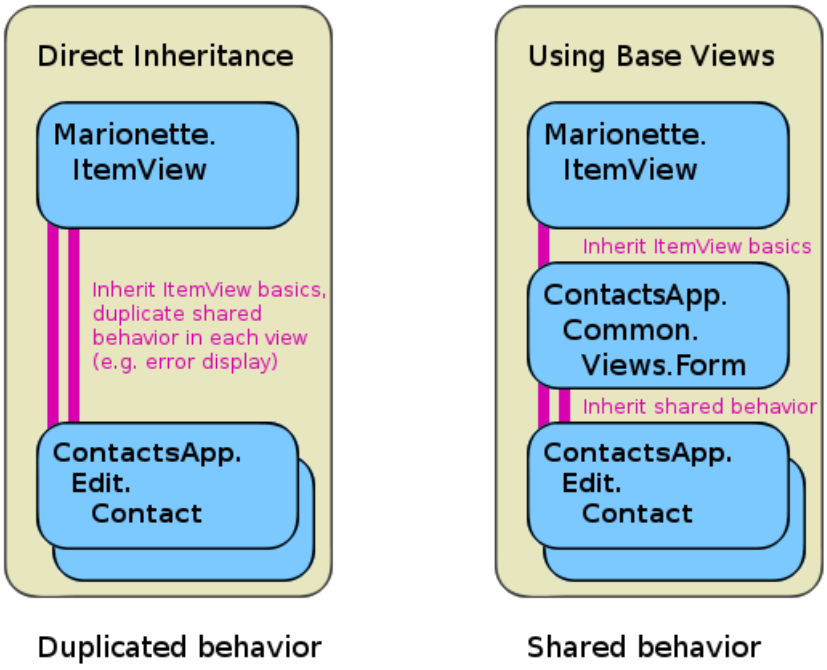
Layouts, on the other hand, are more like "super views": they behave like views, but *in addition* have areas where you can display sub-views (or sub-layouts!). In contrast to regions, layouts are meant to disappear when a user navigates somewhere else and that view is no longer necessary. In our case, when we (e.g.) display a contact in a dedicated view, the "list" layout and all the views it contains will be closed, removed, and replaced with the view displaying that particular contact. Put another way, the layout contained in the "main" region will be removed (along with the view it contains), but the "main" region itself will still exist: it will simply be displaying a different view (the dedicated contact display view).

[149] https://github.com/davidsulc/marionette-gentle-introduction/commit/d84e9308294420e0470a4f6f1e4e08255bce1b55

Extending from Base Views

With our "control panel" in place, we want to be able to click on that "New contact" button and display a modal form to create a new contact. We've already got a modal form to manage contact data: our "edit" view contains a form and manages error message display. If we think about it, that's most of the behavior we want in our form for new contacts. In fact, the main difference will probably just be the title: we'll want to display "New Contact" instead of "Edit John Doe".

Since we're sharing a *lot* of functionality (such as error display management), but need some features to behave differently (e.g. the displayed title), the best approach would be to have some sort of "form" view that would contain the common logic, and our "edit" and "new" views could extend from that and add their specific behavior. Here's the idea:



Duplicated behavior Shared behavior

In the current situation, we'd have to duplicate common functionality (such as error display management) within the "edit" and "new" views. But if we instead extend from a common view as on the right, we can define the shared behavior there, and extend that view to create our "new" and "edit" views. All that we'll have left to do in our new/edit views is to specify the functionality that is *specific* to that view. Sounds pretty good, doesn't it?

Can't we just reuse the edit form? Yes we could, technically: render the edit form with a new model instance (i.e. empty attributes) and then process the form data to create a new contact. But that would be going against the *single responsibility principle*[150]: the edit view shouldn't be used to create new contacts, it's meant only to *edit existing contacts* (as its name implies). In most cases, going against established software patterns and principles leads to unhappy times down the road...

Where should we store our common view? Since it will deal with contact forms, it should belong to the `ContactsApp` module, and not the main `ContactManager` application (where we have stored our common loading view). Let's create it:

Adding the common form view (assets/js/apps/contacts/common/views.js)

```
1  ContactManager.module("ContactsApp.Common.Views", function(Views,
2  ContactManager, Backbone, Marionette, $, _){
3    Views.Form = Marionette.ItemView.extend({
4      template: "#contact-form",
5
6      events: {
7        "click button.js-submit": "submitClicked"
8      },
9
10     submitClicked: function(e){
11       e.preventDefault();
12       var data = Backbone.Syphon.serialize(this);
13       this.trigger("form:submit", data);
14     },
15
16     onRender: function(){
17       if( ! this.options.asModal){
18         var $title = $("<h1>", { text: this.title });
19         this.$el.prepend($title);
20       }
21     },
22
23     onShow: function(){
24       if(this.options.asModal){
25         this.$el.dialog({
26           modal: true,
27           title: this.title,
```

[150] http://en.wikipedia.org/wiki/Single_responsibility_principle

```
28            width: "auto"
29          });
30        }
31      },
32
33      onFormDataInvalid: function(errors){
34        var $view = this.$el;
35
36        var clearFormErrors = function(){
37          var $form = $view.find("form");
38          $form.find(".help-inline.error").each(function(){
39            $(this).remove();
40          });
41          $form.find(".control-group.error").each(function(){
42            $(this).removeClass("error");
43          });
44        }
45
46        var markErrors = function(value, key){
47          var $controlGroup = $view.find("#contact-" + key).parent();
48          var $errorEl = $("<span>",
49                           { class: "help-inline error", text: value });
50          $controlGroup.append($errorEl).addClass("error");
51        }
52
53        clearFormErrors();
54        _.each(errors, markErrors);
55      }
56    });
57  });
```

Pretty straightforward: we simply extracted the common functionality from our edit view. Let's quickly add our new module to the `index.html`, right after we include the file containing our `ContactsApp`:

Including the common views (index.html)

```
1  <script src="./assets/js/apps/contacts/contacts_app.js"></script>
2  <script src="./assets/js/apps/contacts/common/views.js"></script>
```

Since we're centralizing the common functionality, we obviously need to remove it from our "edit"

view. At the same time, we need to extend from our common form view (line 3). Here's what we're left with:

Removing common functionality from the edit view (assets/js/apps/contacts/edit/edit_view.js)

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

With our common view defined and containing all of the shared functionality, let's define our "new" view in a new file:

Defining our 'new' view (assets/js/apps/contacts/new/new_view.js)

```
1 ContactManager.module("ContactsApp.New", function(New, ContactManager,
2 Backbone, Marionette, $, _){
3   New.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
4     title: "New Contact"
5   });
6 });
```

Short and sweet! We still need to include it:

Including the 'new' view (index.html)

```
1 <script src="./assets/js/apps/contacts/edit/edit_view.js"></script>
2 <script src="./assets/js/apps/contacts/edit/edit_controller.js"></script>
3 <script src="./assets/js/apps/contacts/new/new_view.js"></script>
4
5 <script type="text/javascript">
6   ContactManager.start();
7 </script>
```

And of course, we now need to instantiate and display it in our `List` controller. We start by triggering an event when the "create contact" button is clicked in our panel:

Triggering an event from the panel (assets/js/apps/contacts/list/list_view.js)

```
List.Panel = Marionette.ItemView.extend({
  template: "#contact-list-panel",

  triggers: {
    "click button.js-new": "contact:new"
  }
});
```

Notice we're using a `triggers` hash to make Marionette match an event selector with an event to trigger (see documentation[151]). We introduced the `triggers` hash here.

Then, we listen for and process that event in our `List` controller:

Processing the 'contact:new' event (assets/js/apps/contacts/list/list_controller.js)

```
ContactManager.module("ContactsApp.List", function(List, ContactManager,
Backbone, Marionette, $, _){
  List.Controller = {
    listContacts: function(){
      var loadingView = new ContactManager.Common.Views.Loading();
      ContactManager.mainRegion.show(loadingView);

      var fetchingContacts = ContactManager.request("contact:entities");

      var contactsListLayout = new List.Layout();
      var contactsListPanel = new List.Panel();

      $.when(fetchingContacts).done(function(contacts){
        var contactsListView = new List.Contacts({
          collection: contacts
        });

        contactsListLayout.on("show", function(){
          contactsListLayout.panelRegion.show(contactsListPanel);
          contactsListLayout.contactsRegion.show(contactsListView);
```

[151] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.view.md#viewtriggers

```
21          });
22
23          contactsListPanel.on("contact:new", function(){
24            var newContact = new ContactManager.Entities.Contact();
25
26            var view = new ContactManager.ContactsApp.New.Contact({
27              model: newContact,
28              asModal: true
29            });
30
31            view.on("form:submit", function(data){
32              var highestId = contacts.max(function(c){ return c.id; });
33              highestId = highestId.get("id");
34              data.id = highestId + 1;
35              if(newContact.save(data)){
36                contacts.add(newContact);
37                ContactManager.dialogRegion.close();
38                contactsListView.children.findByModel(newContact).
39                                                    flash("success");
40              }
41              else{
42                view.triggerMethod("form:data:invalid",
43                                   newContact.validationError);
44              }
45            });
46
47            ContactManager.dialogRegion.show(view);
48          });
49
50          // other event handlers
51        });
52      }
53    }
54  });
```

We've added a lot of new code (lines 23-48), so let's break it down:

- our form needs a model to display, because it puts the model's current attributes in the input fields. So we create a new contact on line 24;
- we instantiate a new instance of our "new contact" view on lines 26-29;
- when the form data is submitted, we need to process it. But first, we need to manually create an `id` for our new model. This would normally be done by the server, but we don't have one.

So we determine the highest `id` currenlty in use, then add that to the data used to create a contact instance (lines 32-34).
- if the data is valid, we proceed much like in our "edit" case by adding the new model to the collection and closing the dialog region on lines 36-37 (we'll go over this in more detail below). If the data isn't valid, we trigger the form's method so it will display the error messages (lines 42-43);
- finally, we display our "new contact" form on line 47.

When interacting with a server (via a RESTful API), the back end would create the new model and provide us with the new model instance by returning all model attributes, *including the* `id` *attribute*. That's the main point: the server would be setting the `id` attribute. Since we're not using a server, we need to set the `id` attribute manually on lines 32-34 by incrementing the highest `id` attribute we can find.

On line 36, we add the newly created model the `contacts` collection, which makes Marionette render a new item view to display the created model instance. We then close the dialog (line 37) and determine which item view was created for the new model, before animating it (lines 38-39).

If you try it out, you'll see our code fails miserably: our view can't display the `firstName` attribute in our template! That's because the contact we create on line 24 has no attributes: it's essentially an empty object that behaves as a contact model instance. How can we solve this issue and get our form to display properly? We have several options:

- use a different form template that doesn't require attributes
- checking for the attributes within the template
- set the required attributes on our new contact before instantiating the view
- define default values on our model

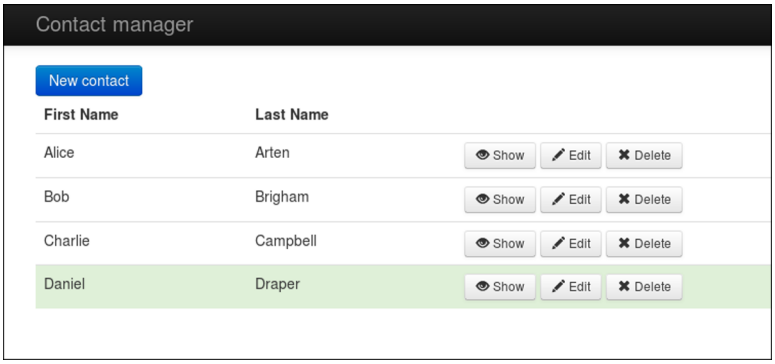
Let's go with the last option, since it allows us to reuse the existing form template, and won't require us to define empty attributes every place we'll need a new contact instance. Here's our `Contact` entity with default values:

Defining default attribute values for the `Contact` entity (assets/js/entities/contact.js)

```
1  Entities.Contact = Backbone.Model.extend({
2    urlRoot: "contacts",
3
4    defaults: {
5      firstName: "",
6      lastName: "",
7      phoneNumber: ""
8    },
```

```
9
10    // validate method
11  });
```

We can now create new contacts using our new feature. Here's what we see right after saving a new contact:



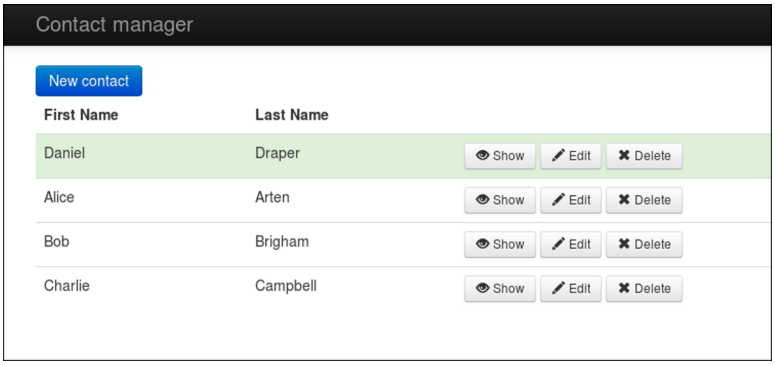
Animating the newly added contact

Let's add one last touch: we'd like to have the new item view added *at the top* of the list, instead of being appended at the bottom. Collection views define an `appendHtml` function that is used to determine how child views should be added to their container. By default they are appended, and we'd like to have this new view *prepended*. But we can't just override `appendHtml` in the view definition, because then our entire list of contacts would be displayed in reverse order. Instead, we want to define the `appendHtml` only *after* the entire collection has been displayed. A good way to achieve this is to override the function once the view's collection has been rendered, because at that point all the child views will have been inserted in the proper order. Here we go:

Overriding `appendHtml` after the collection has been rendered (assets/js/apps/contacts/list/list_view.js)

```
1  List.Contacts = Marionette.CompositeView.extend({
2    tagName: "table",
3    className: "table table-hover",
4    template: "#contact-list",
5    itemView: List.Contact,
6    itemViewContainer: "tbody",
7
8    onCompositeCollectionRendered: function(){
9      this.appendHtml = function(collectionView, itemView, index){
10       collectionView.$el.prepend(itemView.el);
11     }
12   }
13 });
```

And here's what we see after adding a new contact:



Prepending the new contact

We still have a little housekeeping to do: if the collection is reset, all the child views need to be rerendered and *appended*. So let's add a listener to our collection when we initialize the view:

Restoring the default `appendHtml` behavior on collection reset (assets/js/apps/contacts/list/list_view.js)

```
1  List.Contacts = Marionette.CompositeView.extend({
2    tagName: "table",
3    className: "table table-hover",
4    template: "#contact-list",
5    itemView: List.Contact,
6    itemViewContainer: "tbody",
7
8    initialize: function(){
9      this.listenTo(this.collection, "reset", function(){
10       this.appendHtml = function(collectionView, itemView, index){
11         collectionView.$el.append(itemView.el);
12       }
13     });
14   },
15
16   onCompositeCollectionRendered: function(){
17     this.appendHtml = function(collectionView, itemView, index){
18       collectionView.$el.prepend(itemView.el);
19     }
20   }
21 });
```

Thanks to lines 9-13, `appendHtml` will have its default behavior restored if the collection is reset.

Note that you can also use Marionette's collectionEvents[152] hash to implement this functionality.

Now that we can create new contacts, you can remove the code initializing the contacts (if the collection is empty) in *assets/js/entities/contact.js*. I've left it in for convenience while developing: delete all contacts, and you have a new "fresh" data set to work with.

Git commit implementing the "new contact" modal window by extending from a common base view:

e77ac15337858f9e991dcedf88ce4dd28669ab57[153]

If you create a new contact, then navigate to another page (e.g. "show" a given contact), and then press your browser's back button, you will *not* see the contact you've just created. This seems to be due to the browser reloading the web storage's state as it was when first arriving on the "#contacts" page, and therefore without the new contact (created later on). If you refresh the page, however, you'll see your contact has been properly persisted and is now displayed. This particular quirk is due to how web storage values are restored when the "back" button is pressed in the browser, and would *not* happen when dealing with a remote API.

[152] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.view.md#viewmodelevents-and-viewcollectionevents

[153] https://github.com/davidsulc/marionette-gentle-introduction/commit/e77ac15337858f9e991dcedf88ce4dd28669ab57

Managing Dialogs with a Dedicated Region

Now that we're using modal dialogs in multiple places, it's time to clean up our code by using a dedicated region to manage them. This region will be responsible for calling the `dialog` function to turn a view into a modal window, and remove its DOM element when the dialog is closed, among other responsibilities.

Let's start by defining this new region in a new file:

Defining the new dialog region (assets/js/apps/config/marionette/regions/dialog.js)

```
Marionette.Region.Dialog = Marionette.Region.extend({
  onShow: function(view){
    this.listenTo(view, "dialog:close", this.closeDialog);

    var self = this;
    this.$el.dialog({
      modal: true,
      title: view.title,
      width: "auto",
      close: function(e, ui){
        self.closeDialog();
      }
    });
  },

  closeDialog: function(){
    this.stopListening();
    this.close();
    this.$el.dialog("destroy");
  }
});
```

What's happening in this new region? First, we listen for the "dialog:close" event on our view using `listenTo`[154], and execute the `closeDialog` function if it gets triggered. This will allow us to close the dialog region from the view by triggering this event *on the view*.

[154] http://backbonejs.org/#Events-listenTo

Then, when we show a view within this region, we want to execute the `dialog` method on the region's `$el` with the proper options, per jQuery UI's documentation[155]. In particular, we provide a `close` function[156] to be called when the dialog is closed (regardless of the means: pressing escape, clicking the "x", etc.) to clean everything up:

- stop listening for events (e.g. the "dialog:close" event)
- close the Marionette region (which in turn closes the view it contains)
- restore the region's DOM element as it was before the dialog was displayed

The `dialog` method is now being called on the *region*'s `$el`, whereas it was called on the *view*'s `$el` previously. Since the view will be managed by the region containing it, we no longer call the `dialog` method directly on the view: it would bypass the region's control of its contained view, and break encapsulation.

With this new region defined, let's use it in our app:

Using our custom region for dialogs (assets/js/app.js)

```
ContactManager.addRegions({
  mainRegion: "#main-region",
  dialogRegion: Marionette.Region.Dialog.extend({
    el: "#dialog-region"
  })
});
```

Naturally, for this to work, we need to include the file defining our custom dialog region *before* we include our app file in `index.html`:

Including our custom dialog region (index.html)

```
<script src="./assets/js/apps/config/marionette/regions/dialog.js">
  </script>
<script src="./assets/js/app.js"></script>
```

Since our custom dialog region will now be turning views into modal dialogs, we can clean up our common `Form` view:

[155] http://api.jqueryui.com/dialog/

[156] http://api.jqueryui.com/dialog/#event-close

Cleaning up the common `Form` view (assets/js/apps/contacts/common/views.js)

```
ContactManager.module("ContactsApp.Common.Views", function(Views,
ContactManager, Backbone, Marionette, $, _){
  Views.Form = Marionette.ItemView.extend({
    template: "#contact-form",

    events: {
      "click button.js-submit": "submitClicked"
    },

    // submitClicked function

    // onFormDataInvalid function
  });
});
```

We've removed the `onShow` function, because this functionality is now in the `dialog` region. But we've also removed the `onRender` function: since we have a dedicated region for our modals, we won't be passing in an `asModal` attribute. Therefore, the onRender code should really be moved to the `Edit` view: our `New` view will only be called as a modal dialog. Here's our new `Edit` view:

Moving `onRender` to the `Edit` module (assets/js/apps/contacts/edit/edit_view.js)

```
ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
Backbone, Marionette, $, _){
  Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
    initialize: function(){
      this.title = "Edit " + this.model.get("firstName") + " ";
      this.title += this.model.get("lastName");
    },

    onRender: function(){
      if(this.options.generateTitle){
        var $title = $("<h1>", { text: this.title });
        this.$el.prepend($title);
      }
    }
  });
});
```

You'll noticed we've also renamed the option's name to something more fitting: `generateTitle`. So let's adapt our controller code with that in mind (line 13):

Generating the title when needed (assets/js/apps/contacts/edit/edit_controller.js)

```
1  ContactManager.module("ContactsApp.Edit", function(Edit,
2  ContactManager, Backbone, Marionette, $, _){
3    Edit.Controller = {
4      editContact: function(id){
5        // displaying the loading view
6
7        var fetchingContact = ContactManager.request("contact:entity", id);
8        $.when(fetchingContact).done(function(contact){
9          var view;
10         if(contact !== undefined){
11           view = new Edit.Contact({
12             model: contact,
13             generateTitle: true
14           });
15
16           // process form submission
17         }
18         else{
19           view = new ContactManager.ContactsApp.Show.MissingContact();
20         }
21
22         ContactManager.mainRegion.show(view);
23       });
24     }
25   };
26 });
```

And since we're no longer using the `asModal` attribute to turn views into modals, we need to update our `List` controller to reflect that:

Removing the `asModal` option (assets/js/apps/contacts/list/list_controller.js)

```
ContactManager.module("ContactsApp.List", function(List, ContactManager,
Backbone, Marionette, $, _){
  List.Controller = {
    listContacts: function(){
      // instantiating the various views

      $.when(fetchingContacts).done(function(contacts){
        // insantiating the Contacts view, various events handlers

        contactsListPanel.on("contact:new", function(){
          var newContact = new ContactManager.Entities.Contact();

          var view = new ContactManager.ContactsApp.New.Contact({
            model: newContact
          });

          view.on("form:submit", function(data){
            // set new contact's id
            if(newContact.save(data)){
              contacts.add(newContact);
              view.trigger("dialog:close");
              contactsListView.children.findByModel(newContact).
                                                  flash("success");
            }
            else{
              view.triggerMethod("form:data:invalid",
                                 newContact.validationError);
            }
          });

          ContactManager.dialogRegion.show(view);
        });

        // "itemview:contact:show" handler

        contactsListView.on("itemview:contact:edit",
                                 function(childView, model){
          var view = new ContactManager.ContactsApp.Edit.Contact({
            model: model
          });
```

```
41
42            view.on("form:submit", function(data){
43              if(model.save(data)){
44                childView.render();
45                view.trigger("dialog:close");
46                childView.flash("success");
47              }
48              else{
49                view.triggerMethod("form:data:invalid",
50                                   model.validationError);
51              }
52            });
53
54            ContactManager.dialogRegion.show(view);
55          });
56
57          // "itemview:contact:delete" handler
58
59          ContactManager.mainRegion.show(contactsListLayout);
60        });
61      }
62    }
63  });
```

As you can see on lines 14 and 39, we no longer provide an `asModal` option when creating a new view instance. Instead, the views are displayed as modals by using the dedicated dialog region to display them on lines 31 and 54, respectively.

In addition, on lines 21 and 45, we trigger the "dialog:close" event on our views when the model was saved, which closes the modal region (thanks to the event listener we've configured).

And there we have it! We've now modified our app to use a dedicated `dialog` region to turn any view it displays into a modal dialog.

Git commit implementing a dedicated dialog region:

b2959c4b93881185fed9c7b929a01b44e864c19d[157]

[157] https://github.com/davidsulc/marionette-gentle-introduction/commit/b2959c4b93881185fed9c7b929a01b44e864c19d

If you want more information on using regions to manage dialogs, refer to this blog post[158] by Derick Bailey, as well as this screencast[159] by Brian Mann. The screencast in particular will show you how to define a more powerful and flexible dialog-managing region, which we don't have room to get into here.

Customizing onRender

Now that we have our views refactored, let's improve our user interface slightly:

- when the user is creating a new contact, the form's button should read "Create contact"
- when the user is updating an existing contact, the form's button should read "Update contact"

Right now, our form has the generic "Save" text on it. We'll leave that, as it will serve as a generic fallback. But now that the `onRender` function is defined in each view, we can specify different behavior adapted to the view's needs.

Changing the 'new' form button's text (assets/js/apps/contacts/new/new_view.js)

```
ContactManager.module("ContactsApp.New", function(New, ContactManager,
Backbone, Marionette, $, _){
  New.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
    title: "New Contact",

    onRender: function(){
      this.$(".js-submit").text("Create contact");
    }
  });
});
```

[158] http://lostechies.com/derickbailey/2012/04/17/managing-a-modal-dialog-with-backbone-and-marionette/
[159] http://www.backbonerails.com/screencasts/building-dialogs-with-custom-regions

Changing the 'edit' form button's text (assets/js/apps/contacts/edit/edit_view.js)

```
ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
Backbone, Marionette, $, _){
  Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
    // initialize function

    onRender: function(){
      if(this.options.generateTitle){
        var $title = $("<h1>", { text: this.title });
        this.$el.prepend($title);
      }

      this.$(".js-submit").text("Update contact");
    }
  });
});
```

We're using `this.$(...)` to select our DOM elements, which is syntactic sugar provided by Backbone: it is equivalent to `this.$el.find(...)`

See how easy that was? Another example in favor of proper application structure!

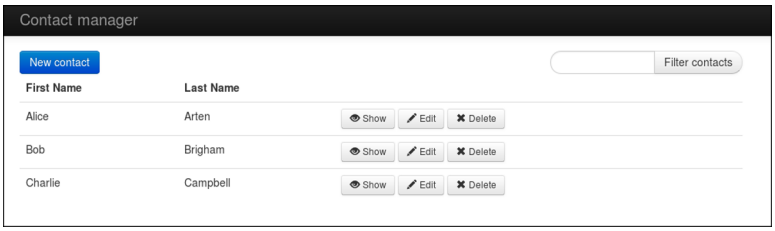
Git commit adapting the submit button's text:

3328db99418603c73d18d2ed8392be70a8df206d[160]

[160] https://github.com/davidsulc/marionette-gentle-introduction/commit/3328db99418603c73d18d2ed8392be70a8df206d

Filtering Contacts

In our `List` view, we'd like to be able to filter contacts and display only those that contain a given string. To allow users to do that, let's add a text input field to our existing panel, so our `List` view will look like this:



Adding a 'filter' text input

We just add the input to our template:

Adding a 'filter' input field to our panel (index.html)

```
<script type="text/template" id="contact-list-panel">
  <button class="btn btn-primary js-new">New contact</button>
  <form id="filter-form" class="form-search form-inline pull-right">
    <div class="input-append">
      <input type="text" class="span2 search-query js-filter-criterion">
      <button type="submit" class="btn">Filter contacts</button>
    </div>
  </form>
</script>
```

And of course we need to get our view to react to the form being submitted, and trigger an event:

Reacting to the form submit (assets/js/apps/contacts/list/list_view.js)

```
1  List.Panel = Marionette.ItemView.extend({
2    template: "#contact-list-panel",
3
4    triggers: {
5      "click button.js-new": "contact:new"
6    },
7
8    events: {
9      "submit #filter-form": "filterContacts"
10   },
11
12   filterContacts: function(e){
13     e.preventDefault();
14     var criterion = this.$(".js-filter-criterion").val();
15     this.trigger("contacts:filter", criterion);
16   }
17 });
```

On line 14, we're fetching the input field's value "manually". We could have used `backbone.syphon`, but it seems excessive to obtain a single value. And this way, we also get to appreciate the work the `syphon` library does for us when fetching many form values.

Finally, our controller has to process that event:

Processing the 'filter' event in the controller (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        // displaying the loading view, and instantiating layout and panel
6
7        $.when(fetchingContacts).done(function(contacts){
8          var contactsListView = new List.Contacts({
9            collection: contacts
10         });
11
```

```
12          contactsListPanel.on("contacts:filter",
13                                  function(filterCriterion){
14            console.log("filter list with criterion ", filterCriterion);
15          });
16
17          // rest of view event handlers
18
19        });
20      }
21    }
22  });
```

If we type text in the input field and press the filter button, we can see the text value is properly forwarded to our controller. What now? We need to filter our `contacts` collection and reset its contents to have only models that match the filtering criterion. But there's a cleaner, more encapsulated way of thinking about the problem: we could create a special collection that would filter itself. Then, we'd simply pass that special collection to the composite view: each time we execute the filtering function, the view will refresh itself automatically (since it listens to the collection's "reset" event).

We'll follow the second approach, using the `FilteredCollection` created in chapter Creating a Filtered Collection. If you want to learn about the implementation details, take a look at them here. Otherwise, you can simply grab the file[161] and save it in *assets/js/entities/common.js*.

Before anything else, let's include this new file in `index.html`:

Including the filtered collection (index.html)

```
1  <script src="./assets/js/apps/config/marionette/regions/dialog.js">
2                                                                  </script>
3  <script src="./assets/js/app.js"></script>
4  <script src="./assets/js/apps/config/storage/localstorage.js"></script>
5  <script src="./assets/js/entities/common.js"></script>
```

Now let's start using a filtered collection:

[161] https://raw.github.com/davidsulc/marionette-gentle-introduction/master/assets/js/entities/common.js

Basic wiring to use a filtered collection (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        // displaying the loading view, and instantiating layout and panel
6
7        $.when(fetchingContacts).done(function(contacts){
8          var filteredContacts = ContactManager.Entities.
9                                          FilteredCollection({
10           collection: contacts
11         });
12
13         var contactsListView = new List.Contacts({
14           collection: filteredContacts
15         });
16
17         contactsListPanel.on("contacts:filter",
18                                    function(filterCriterion){
19           filteredContacts.filter(filterCriterion);
20         });
21
22         // rest of view event handlers
23
24       });
25     }
26   }
27 });
```

Let's explain this new code:

- we create a filtered collection on lines 8-11;
- on lines 13-15, we instantiate our view with the filtered collection (instead of our regular `contacts` collection);
- on lines 17-20, we react to the "contacts:filter" event by (re-)filtering our `FilteredCollection` instance.

Naturally, before we can filter our `filteredContacts` collection, we need to specify a filtering function. We want to filter the contacts that contain the provided text in at least one of the contact's attributes. Here's how that translates into a filtering function:

Providing a `filterFunction` to the filtered collection (assets/js/apps/contacts/list/list_controller.js)

```
1  var filteredContacts = ContactManager.Entities.FilteredCollection({
2    collection: contacts,
3    filterFunction: function(filterCriterion){
4      var criterion = filterCriterion.toLowerCase();
5      return function(contact){
6        if(contact.get("firstName").toLowerCase().indexOf(criterion) !== -1
7         || contact.get("lastName").toLowerCase().indexOf(criterion) !== -1
8         || contact.get("phoneNumber").toLowerCase().
9                                                 indexOf(criterion) !== -1){
10           return contact;
11       }
12     };
13   }
14 });
```

Here's what our app looks like when we filter with "li":

Displaying filtered contacts

Our filtering is working nicely: when we enter a new filtering criterion, the filtered collection gets reset with the contacts that match the criterion, and the composite view rerenders all the child views. Great!

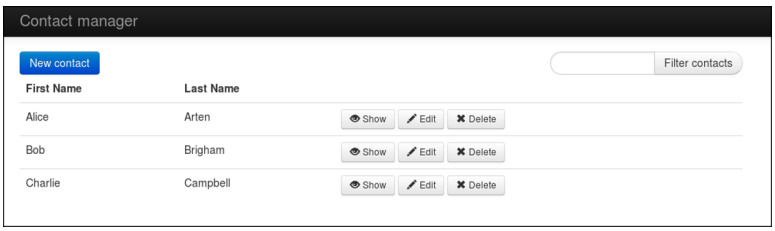
But if we try adding a contact that doesn't match the filtering criterion (e.g. doesn't contain the "li" string in the case above), our app raises an error. This is because we retrieve the new contact's item view to make it flash green (by calling our `success` function on it). However, if the contact doesn't match the filtering criterion, its item view won't be displayed: `children.findByModel` will return `undefined`, and we'll be calling `success` on `undefined`.

Let's correct that (lines 14-20):

Checking for the item view before animating (assets/js/apps/contacts/list/list_controller.js)

```
contactsListPanel.on("contact:new", function(){
  var newContact = new ContactManager.Entities.Contact();

  var view = new ContactManager.ContactsApp.New.Contact({
    model: newContact
  });

  view.on("form:submit", function(data){
    var highestId = contacts.max(function(c){ return c.id; }).get("id");
    data.id = highestId + 1;
    if(newContact.save(data)){
      contacts.add(newContact);
      view.trigger("dialog:close");
      var newContactView = contactsListView.children.
                                                findByModel(newContact);
      // check whether the new contact view is displayed (it could be
      // invisible due to the current filter criterion)
      if(newContactView){
        newContactView.flash("success");
      }
    }
    else{
      view.triggerMethod("form:data:invalid", newContact.validationError);
    }
  });

  ContactManager.dialogRegion.show(view);
});
```

Our filtering now works properly. And if we filter for a blank string, the entire contacts list is displayed again:



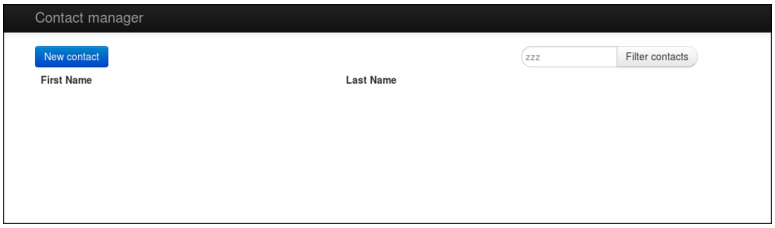
Filtering with a blank string

Git commit implementing filtering functionality:

eb55e34fba15a51c47b677c47814af0ba660345f[162]

Implementing an Empty View

Let's try filtering our contacts with 'zzz':



No filtering matches

That doesn't look very good: users can't tell if no contacts match their filtering criterion, if the app is waiting for data, or if it's broken. Instead, if there are no contacts to display, we want to show users a short message informing them of this fact.

Once again, Marionette has built-in functionality for this purpose: the `emptyView`[163] property. All we need to do is define a view (and template) to be displayed when the collection is empty:

Adding the template (index.html)

```
1 <script type="text/template" id="contact-list-none">
2   <td colspan="3">No contacts to display.</td>
3 </script>
```

Defining the view (assets/js/apps/contacts/list/list_view.js)

```
1 var NoContactsView = Marionette.ItemView.extend({
2   template: "#contact-list-none",
3   tagName: "tr",
4   className: "alert"
5 });
```

And finally, we need to specify the `emptyView` attribute in our composite view on line 5:

[162] https://github.com/davidsulc/marionette-gentle-introduction/commit/eb55e34fba15a51c47b677c47814af0ba660345f

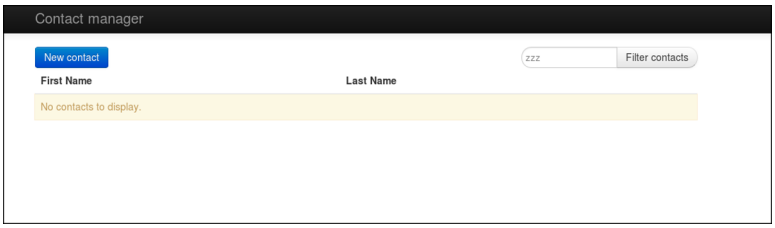
[163] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.collectionview.md#collectionviews-emptyview

Adding the `emptyView` attribute (assets/js/apps/contacts/list/list_view.js)

```
List.Contacts = Marionette.CompositeView.extend({
  tagName: "table",
  className: "table table-hover",
  template: "#contact-list",
  emptyView: NoContactsView,
  itemView: List.Contact,
  itemViewContainer: "tbody",

  // rest of view definition
});
```

We've defined our empty view as a simple `var` (not attached to the `List` module), because we're not going to use it from anywhere else. In other words, the empty view can remain "private" within the `list_view.js` file.

After refreshing, we now have our friendly message displayed when no contacts are diplayed:



"No contacts" message

Git commit implementing the empty view:

41f40de011c2eb18955374e1e41e8ea2c1430cc4[164]

Optional Routes and Query Strings

Our filtering works great, but we should really have the filtering criterion reflected in the URL. That way, users can bookmark their filters, and navigate using the "back" and "forward" browser buttons.

First, let's get the URL updated when we filter. We can add an event listener that will update the URL fragment:

[164] https://github.com/davidsulc/marionette-gentle-introduction/commit/41f40de011c2eb18955374e1e41e8ea2c1430cc4

Event listener to update the URL fragment when filtering (assets/js/apps/contacts/contacts_app.js)

```
1 ContactManager.on("contacts:filter", function(criterion){
2   if(criterion){
3     ContactManager.navigate("contacts/filter/criterion:" + criterion);
4   }
5   else{
6     ContactManager.navigate("contacts");
7   }
8 });
```

If we have a filtering criterion, we add it to the URL fragment (lines 2-4). If no criterion is given (i.e. the user wants to display all contacts by clearing the filtering criterion), we simply want the URL fragment to be "#contacts" instead of "#contacts/filter/criterion:" (lines 5-7).

As of Backbone 1.1.0, query strings are completely ignored by Backbone (see discussion here[165] and here[166], so we can't use query strings to pass the argument around. In other words, it isn't possible to use a URL such as "#contacts?filter=...".

Notice that we're only updating the URL fragment: we're not executing any controller actions, because we're not going to route to a specific filtered page from anywhere in our application. With this listener in place, we just trigger the event from our controller when the users filter:

Triggering the filter event (assets/js/apps/contacts/list/list_controller.js)

```
1 contactsListPanel.on("contacts:filter", function(filterCriterion){
2   filteredContacts.filter(filterCriterion);
3   ContactManager.trigger("contacts:filter", filterCriterion);
4 });
```

Now for the other way around: when users navigate the page with the URL fragment "#contacts/-filter/criterion:li", we need our app to be filtering the contacts with "li". Let's start by updating our routes:

[165] https://github.com/jashkenas/backbone/issues/2801
[166] https://github.com/jashkenas/backbone/issues/891

Updating the `ContactsApp` routes (assets/js/apps/contacts/contacts_app.js)

```
1 ContactsApp.Router = Marionette.AppRouter.extend({
2   appRoutes: {
3     "contacts(/filter/criterion::criterion)": "listContacts",
4     "contacts/:id": "showContact",
5     "contacts/:id/edit": "editContact"
6   }
7 });
```

The portion in parentheses on line 3 means it is optional: the route will match even if that section is absent. Of course, we need to update our associated `listContacts` function to receive the optional filtering criterion:

assets/js/apps/contacts/contacts_app.js

```
1 var API = {
2   listContacts: function(criterion){
3     ContactsApp.List.Controller.listContacts(criterion);
4   },
5
6   // rest of API
7 };
```

Since our routing controller calls our `List` controller with an optional `criterion` argument, we need to update that function also:

Updating the `List` controller to receive a filtering criterion (assets/js/apps/contacts/list/list_controller.js)

```
1 List.Controller = {
2   listContacts: function(criterion){
3     // rest of function
4   }
5 }
```

What will we do with this criterion? If it's provided, we need to filter our contact collection, *and* display the criterion in the filtering input field in our view:

Processing the filtering criterion in the controller (assets/js/apps/contacts/list/list_controller.js)

```
1  List.Controller = {
2    listContacts: function(criterion){
3      // display loading view
4
5      var fetchingContacts = ContactManager.request("contact:entities");
6
7      var contactsListLayout = new List.Layout();
8      var contactsListPanel = new List.Panel();
9
10     $.when(fetchingContacts).done(function(contacts){
11       var filteredContacts = ContactManager.Entities.FilteredCollection({
12         // attributes for filtered collection
13       });
14
15       if(criterion){
16         filteredContacts.filter(criterion);
17         contactsListPanel.once("show", function(){
18           contactsListPanel.triggerMethod("set:filter:criterion",
19                                           criterion);
20         });
21       }
22
23       // rest of code unchanged
```

On lines 17-20, we need to trigger a special method in our view to get the criterion displayed in the filtering input field. But this method should be executed only *once*, not each time the view is shown. Therefore, we use the once[167] method. And naturally, these lines mean we need to define this method in our view (lines 22-24):

[167] http://backbonejs.org/#Events-once

Adapting the panel view (assets/js/apps/contacts/list/list_view.js)

```
1  List.Panel = Marionette.ItemView.extend({
2    template: "#contact-list-panel",
3
4    triggers: {
5      "click button.js-new": "contact:new"
6    },
7
8    events: {
9      "submit #filter-form": "filterContacts"
10   },
11
12   ui: {
13     criterion: "input.js-filter-criterion"
14   },
15
16   filterContacts: function(e){
17     e.preventDefault();
18     var criterion = this.$(".js-filter-criterion").val();
19     this.trigger("contacts:filter", criterion);
20   },
21
22   onSetFilterCriterion: function(criterion){
23     this.ui.criterion.val(criterion);
24   }
25 });
```

We snuck in an extra change: since we're going to refer to the filtering input ui element several times, we've defined a "shortcut" to it on line 13. This `ui` hash is handled by Marionette, and all you need to do is provide a name on the left, and associate a jQuery selector on the right.

On lines 22-24, we can set the input field's value to what we've received from the controller (and was in turn parsed from the URL fragment). All done! Now, if we filter our contacts, the URL gets updated. And if we go directly to URL "#contacts?filter=li", we'll see our contacts list filtered by "li" as if we had navigated there and filtered it manually.

This type of behavior is what was meant when routing was explained earlier: the URL route is triggered when the user uses it at the entry point, and it serves to "configure" the application's state (i.e. how the contacts are filtered, in our case). Once the user is within the application, we control how the application's state gets modified, and only need to keep the URL up to date.

Git commit linking filtering to routes:

6cc77c789801950bfcefc61af2ba9927d84b98bf[168]

[168]https://github.com/davidsulc/marionette-gentle-introduction/commit/6cc77c789801950bfcefc61af2ba9927d84b98bf

The ‘About’ Sub-Application

Since we want to learn about switching sub-applications, we’ll need to add another sub-application to our `ContactManager` application (in addition to our `ContactsApp`). Our new sub-app will be the `AboutApp`, and all it will do is display a static message at the “#about” URL fragment.

Code the Sub-App Yourself

Try and develop the sub-application on your own, by applying what you’ve learned as we developed the `ContactsApp` together. Once you’re done, come back and compare your solution. This is the template we’ll use to display our static message:

```
<script type="text/template" id="about-message">
  <h1>About this application</h1>
  <p>This application was designed to
      accompany you during your learning.</p>
  <p>Hopefully, it has served you well !</p>
</script>
```

Coding the Sub-App

This chapter moves fast and contains very few explanations, because there are no new concepts presented here. If you get confused, refer to the implementation of the `ContactsApp` and the related explantions. In particular, the chapters on application structure and routing should refresh your memory.

First, we add the template to our `index.html`:

Adding the static text template (index.html)

```
<script type="text/template" id="about-message">
  <h1>About this application</h1>
  <p>This application was designed to
      accompany you during your learning.</p>
  <p>Hopefully, it has served you well !</p>
</script>
```

Next, we define a view in a new file to render this template:

Defining a view to display our static message (assets/js/apps/about/show/show_view.js)

```
ContactManager.module("AboutApp.Show", function(Show, ContactManager,
Backbone, Marionette, $, _){
  Show.Message = Marionette.ItemView.extend({
    template: "#about-message"
  });
});
```

What's next? How about a controller to instantiate and display the view:

Implementing the `Show` controller (assets/js/apps/about/show/show_controller.js)

```
1 ContactManager.module("AboutApp.Show", function(Show, ContactManager,
2 Backbone, Marionette, $, _){
3   Show.Controller = {
4     showAbout: function(){
5       var view = new Show.Message();
6       ContactManager.mainRegion.show(view);
7     }
8   };
9 });
```

Notice that on line 5, we don't need to specify any model instance when instantiating our view: our template displays a static message only, so there's no need for a model.

And now for the last piece in our `AboutApp`, the routing code:

Implementing routing for the `AboutApp` (assets/js/apps/about/about_app.js)

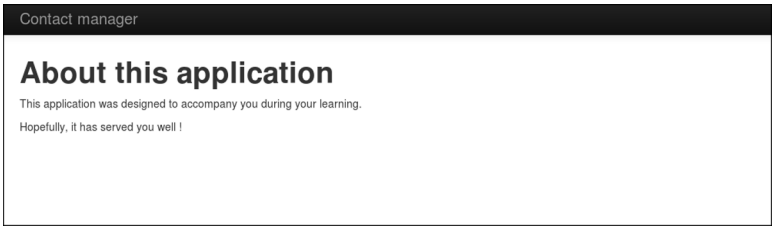
```
ContactManager.module("AboutApp", function(AboutApp, ContactManager,
Backbone, Marionette, $, _){
  AboutApp.Router = Marionette.AppRouter.extend({
    appRoutes: {
      "about" : "showAbout"
    }
  });

  var API = {
    showAbout: function(){
      AboutApp.Show.Controller.showAbout();
    }
  };

  ContactManager.on("about:show", function(){
    ContactManager.navigate("about");
    API.showAbout();
  });

  ContactManager.addInitializer(function(){
    new AboutApp.Router({
      controller: API
    });
  });
});
```

Of course, we need to add all of these files at the end of our `index.html`:

Including our `About` app in index.html

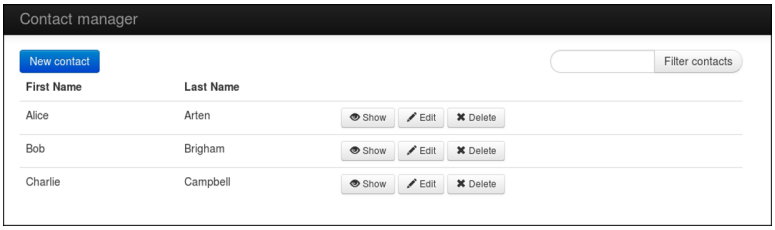
```
<script src="./assets/js/apps/about/about_app.js"></script>
<script src="./assets/js/apps/about/show/show_view.js"></script>
<script src="./assets/js/apps/about/show/show_controller.js"></script>
```

If we refresh the page (to load the new files) and manually enter the "#about" URL fragment, we can see our sub-application:



Displaying our `About` sub-application at "#about"

And if we then enter the "#contacts" URL fragment, we can see our `ContactsApp`:



Displaying our `Contacts` sub-application at "#contacts"

Next up: creating a header app so users can click menu entries to switch applications.

Git commit implementing the About sub-application:

5899800046b2c0bcbca0c4a486b88041726820d6[169]

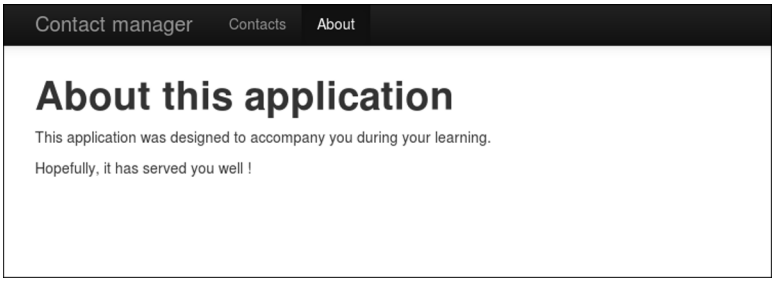
[169] https://github.com/davidsulc/marionette-gentle-introduction/commit/5899800046b2c0bcbca0c4a486b88041726820d6

The 'Header' Sub-Application

We're going to manage the menu in our header with a sub-application. It will control application switching, but will also track the currently active application so its menu entry can be highlighted. All of these features will be handled by manipulating Backbone models, and without nasty HTML `data-` attribute trickery. Instead, we'll use the composite view you've come to know and love.

Setting up the Models

Our models will each represent one entry in our header menu, so we'll need them to have `name` and `url` attributes to store the text we'll display in the link and the link's location, respectively. And in addition to these attributes, we'll also need to determine the active menu entry so it can be highlighted:



Menu with highlighted "About" entry

To do so, we'll use Derick Bailey's Picky[170] Backbone plugin: it will allow us to select a certain item in the collection, and automatically deselect all the others. That's exactly the behavior we want for our header, since only one entry can be active at any given time.

Start by getting the `backbone.picky` plugin here[171] and saving it in *assets/js/vendor/backbone.picky.js*. And we'll need to include it in `index.html` to be able to use it:

Including `backbone.picky` (index.html)

```
1 <script src="./assets/js/vendor/backbone.js"></script>
2 <script src="./assets/js/vendor/backbone.picky.js"></script>
3 <script src="./assets/js/vendor/backbone.syphon.js"></script>
```

Now that we have access to Picky, let's code our header entities (in a new file):

[170] https://github.com/derickbailey/backbone.picky

[171] https://raw.github.com/derickbailey/backbone.picky/master/src/backbone.picky.js

The header entities (assets/js/entities/header.js)

```
ContactManager.module("Entities", function(Entities, ContactManager,
Backbone, Marionette, $, _){
  Entities.Header = Backbone.Model.extend({
    initialize: function(){
      var selectable = new Backbone.Picky.Selectable(this);
      _.extend(this, selectable);
    }
  });

  Entities.HeaderCollection = Backbone.Collection.extend({
    model: Entities.Header,

    initialize: function(){
      var singleSelect = new Backbone.Picky.SingleSelect(this);
      _.extend(this, singleSelect);
    }
  });

  var initializeHeaders = function(){
    Entities.headers = new Entities.HeaderCollection([
      { name: "Contacts", url: "contacts" },
      { name: "About", url: "about" }
    ]);
  };

  var API = {
    getHeaders: function(){
      if(Entities.headers === undefined){
        initializeHeaders();
      }
      return Entities.headers;
    }
  };

  ContactManager.reqres.setHandler("header:entities", function(){
    return API.getHeaders();
  });
});
```

Not much new here: it's very similar to our "contact" entity definition. The only major differences

are the initializers on lines 4-7 and 13-16: they're configuring our model and collection to use the Picky plugin (per the documentation[172]). Also, note that there is no `fetch` call for our header entities since they are static: we simply return them on line 31.

And with this code in place, we' ve got header model instances to handle our menu. Let's add that to our `index.html`:

Adding our header entities (index.html)

```
1  <script src="./assets/js/entities/common.js"></script>
2  <script src="./assets/js/entities/header.js"></script>
3  <script src="./assets/js/entities/contact.js"></script>
```

Adding Templates and Views

With our models set up, we can move on to changing our `index.html` so our header menu will be rendered by Marionette (instead of static HTML). Here's what we have now:

Our current index.html

```
1  <div class="navbar navbar-inverse navbar-fixed-top">
2    <div class="navbar-inner">
3      <div class="container">
4        <span class="brand" href="#">Contact manager</span>
5      </div>
6    </div>
7  </div>
8
9  <div id="main-region" class="container">
10   <p>Here is static content in the web page. You'll notice that it gets
11       replaced by our app as soon as we start it.</p>
12 </div>
13
14 <div id="dialog-region"></div>
```

We need to remove the header menu (lines 1-7) from the displayed HTML and turn it into a template:

[172] https://github.com/derickbailey/backbone.picky#pickysingleselect

Making templates for the header menu (index.html)

```
1  <div id="main-region" class="container">
2    <p>Here is static content in the web page. You'll notice that it gets
3        replaced by our app as soon as we start it.</p>
4  </div>
5
6  <div id="dialog-region"></div>
7
8  <script type="text/template" id="header-template">
9    <div class="navbar-inner">
10     <div class="container">
11       <a class="brand" href="#contacts">Contact manager</a>
12       <div class="nav-collapse collapse">
13         <ul class="nav"></ul>
14       </div>
15     </div>
16   </div>
17 </script>
18
19 <script type="text/template" id="header-link">
20   <a href="#<%= url %>"><%= name %></a>
21 </script>
```

Right below our `dialog` region, we add 2 templates:

- one for the entire header menu navigation bar on lines 8-17 (which was moved from the displayed HTML into a template), which will be used by our composite view
- one for each menu entry, which will be used by the item views (lines 19-21)

You can notice we've slightly changed our `header-template` from what we had before:

- the brand `span` on line 11 became a link, so we can load the `ContactsApp` when the user clicks it
- we've added a `ul` element to contain all of our menu entries (line 13)

Now that we've got templates, let's define some views:

Defining the views for our header menu (assets/js/apps/header/list/list_view.js)

```
1  ContactManager.module("HeaderApp.List", function(List, ContactManager,
2  Backbone, Marionette, $, _){
3    List.Header = Marionette.ItemView.extend({
4      template: "#header-link",
5      tagName: "li"
6    });
7
8    List.Headers = Marionette.CompositeView.extend({
9      template: "#header-template",
10     className: "navbar navbar-inverse navbar-fixed-top",
11     itemView: List.Header,
12     itemViewContainer: "ul"
13   });
14 });
```

Nothing new here: we're simply declaring the new `List` module, along with an item view and a composite view.

We'll naturally need a region to contain our header sub-application, so let's quickly add it (line 1):

Adding a header region (index.html)

```
1  <div id="header-region"></div>
2
3  <div id="main-region" class="container">
4    <p>Here is static content in the web page. You'll notice that it gets
5        replaced by our app as soon as we start it.</p>
6  </div>
7
8  <div id="dialog-region"></div>
```

And to be able to use this new region, we need to declare it within our `ContactManager` application:

Declaring the header region (assets/js/app.js)

```
ContactManager.addRegions({
  headerRegion: "#header-region",
  mainRegion: "#main-region",
  dialogRegion: Marionette.Region.Dialog.extend({
    el: "#dialog-region"
  })
});
```

Implementing the Controller and Sub-Application

With the data and display portions ready, let's add the controller:

Defining the header controller (assets/js/apps/header/list/list_controller.js)

```
ContactManager.module("HeaderApp.List", function(List, ContactManager,
Backbone, Marionette, $, _){
  List.Controller = {
    listHeader: function(){
      var links = ContactManager.request("header:entities");
      var headers = new List.Headers({collection: links});

      ContactManager.headerRegion.show(headers);
    }
  };
});
```

And of course, we need to display our header menu:

Coding the `Header` sub-application file (assets/js/apps/header/header_app.js)

```
1  ContactManager.module("HeaderApp", function(Header, ContactManager,
2  Backbone, Marionette, $, _){
3    var API = {
4      listHeader: function(){
5        Header.List.Controller.listHeader();
6      }
7    };
8
9    ContactManager.commands.setHandler("set:active:header", function(name){
10     ContactManager.HeaderApp.List.Controller.setActiveHeader(name);
11   });
12
13   Header.on("start", function(){
14     API.listHeader();
15   });
16 });
```

We've implemented a command handler on lines 9-11, so that other sub-apps can *command* our header to highlight a different entry.

What's the difference between a command handler[173] and an event handler? They're mainly semantic: we could have written a standard event listener here, and our app would behave exactly the same. But the semantics are different:

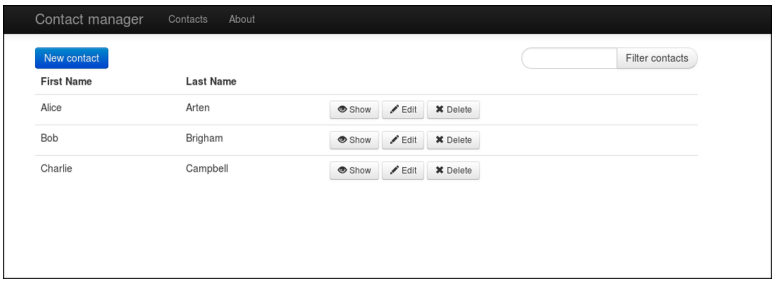
- when we trigger *an event*, we're just notyfing listeners (if any) that some event has taken place (e.g. a button click). It's up to event listeners to react to the event or not;
- when we execute *a command*, we're ordering a remote piece of code to carry out some work.

So why did we use a command here, when we're using events elsewhere in the application? Because we consider the header app's role is to "serve" other sub-applications. Therefore the `ContactsApp` sub-application (e.g.) can order the header to highlight a different entry. In contrast, sub-applications are autonomous, so we just notify the `ContactsApp` (e.g.) that "the user would like to display all the contacts". It is then up to the `ContactsApp` to decide if/when/how to actually display them.

On lines 13-15, we display our header menu as soon as the sub-application starts. And since by

[173] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.commands.md

default Marionette starts all sub-applications as soon as the main application is initialized, our menu will be displayed immediately. If we refresh our page, we'll see the header menu displayed:



Our current menu

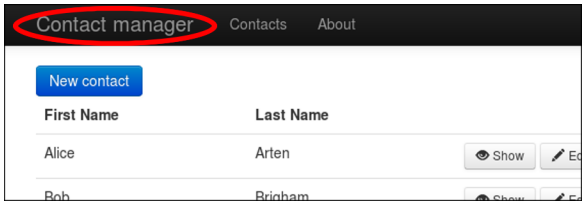
Our menu even works: when we click on a link, we navigate around as expected. But as explained earlier, this isn't the best approach: instead, we'll navigate using events, as we have so far.

Why bother modifying something that's already fully functional?

In the interest of maintainability. By relying on URLs to control our application, we'll eventually start using Backbone as a stateless framework: passing various information in the URLs instead of managing it within our application. Using events to constrain us, we'll automatically enforce the separation between "application behavior" and "URL management".

Navigating with the Brand

First, let's handle our users clicking on the "Contact manager" brand on the left side of the header menu: it should bring them to the page listing all contacts.



The brand in our header menu

Just to refresh your memory, here's the HTML for our brand (from the composite view's `header-template` template):

```
<a class="brand" href="#contacts">Contact manager</a>
```

To implement our new feature, we first add a `click` handler for the brand to trigger an event:

Handling the brand click event in the composite view (assets/js/apps/header/list/list_view.js)

```
List.Headers = Marionette.CompositeView.extend({
  template: "#header-template",
  className: "navbar navbar-inverse navbar-fixed-top",
  itemView: List.Header,
  itemViewContainer: "ul",

  events: {
    "click a.brand": "brandClicked"
  },

  brandClicked: function(e){
    e.preventDefault();
    this.trigger("brand:clicked");
  }
});
```

As you can see, it's a simple matter of capturing the `click` event on our brand link and triggering an event our controller can react to:

Reacting to the 'brand:clicked' event (assets/js/apps/header/list/list_controller.js)

```
List.Controller = {
  listHeader: function(){
    var links = ContactManager.request("header:entities");
    var headers = new List.Headers({collection: links});

    headers.on("brand:clicked", function(){
      ContactManager.trigger("contacts:list");
    });

    ContactManager.headerRegion.show(headers);
  }
};
```

On lines 6-8, we trigger the "contacts:list" event on our application which we've already configured earlier in our `ContactsApp` routing code. Here's the event listener we already have:

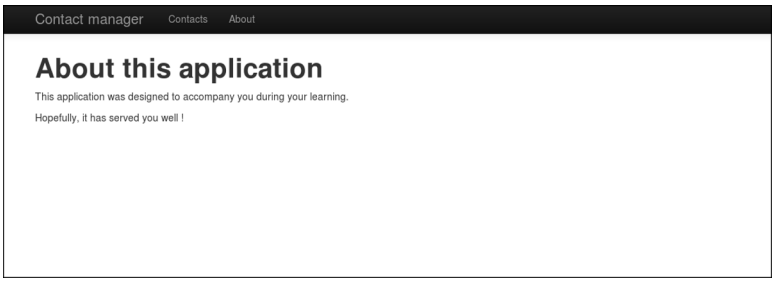
The existing 'contacts:list' event listener in assets/js/apps/contacts/contacts_app.js

```
ContactManager.on("contacts:list", function(){
  ContactManager.navigate("contacts");
  API.listContacts();
});
```

Due to the code we already have in place, our app will react properly to the brand click: updating the URL fragment and triggering the proper controller method.

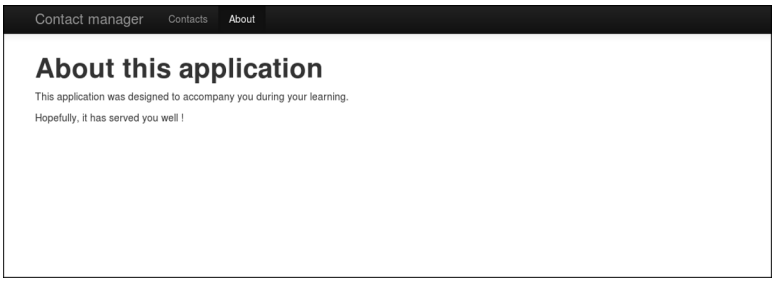
Highlighting the Active Header

Right now, our header menu looks like this:



Our current menu

We'd like it to look like this:



The menu with the active header highlighted

(Depending on your screen, you may not be able to readily notice the difference. In the second image, the "About" header is white, while in the first one, it's grey like the other headers.)

The first thing we need is a method in our header controller to set the currently active header:

Setting the active header (assets/js/apps/header/list/list_controller.js)

```
1  List.Controller = {
2    listHeader: function(){
3      // code omitted for brevity
4    },
5
6    setActiveHeader: function(headerUrl){
7      var links = ContactManager.request("header:entities");
8      var headerToSelect = links.find(function(header){
9        return header.get("url") === headerUrl;
10     });
11     headerToSelect.select();
12     links.trigger("reset");
13   }
14 };
```

What's happening? After getting our header collection (line 7), we determine which model instance should be marked as active. We do that by comparing the model's `url` attribute with the provided function argument. Once we've located the model, we simply call its `select` method. Due to our using the `backbone.picky` plugin to manage our header models, this will select the matching model, and deselect all others (e.g. the previously active header).

Now that we're marking the currently active header, we need to display this information. Since we're using Bootstrap to display the menu, we simply need to add the `active` CSS class to the selected menu entry. We do this in the `onRender` function:

Adding the 'active' class to the selected header (assets/js/apps/header/list/list_view.js)

```
1  List.Header = Marionette.ItemView.extend({
2    template: "#header-link",
3    tagName: "li",
4
5    onRender: function(){
6      if(this.model.selected){
7        // add class so Bootstrap will highlight
8        // the active entry in the navbar
9        this.$el.addClass("active");
10     };
11   }
12 });
```

And last, but not least, we need to set the active headers any time we use our routing controllers to change sub-modules or sub-applications. We achieve this by executing the previously registered command:

Setting the active header in the `ContactsApp` routing controller (assets/js/apps/contacts/contacts_app.js)

```
var API = {
  listContacts: function(criterion){
    ContactsApp.List.Controller.listContacts(criterion);
    ContactManager.execute("set:active:header", "contacts");
  },

  showContact: function(id){
    ContactsApp.Show.Controller.showContact(id);
    ContactManager.execute("set:active:header", "contacts");
  },

  editContact: function(id){
    ContactsApp.Edit.Controller.editContact(id);
    ContactManager.execute("set:active:header", "contacts");
  }
};
```

Setting the active header in the `AboutApp` routing controller (assets/js/apps/about/about_app.js)

```
var API = {
  showAbout: function(){
    AboutApp.Show.Controller.showAbout();
    ContactManager.execute("set:active:header", "about");
  }
};
```

Now, if we enter the "#about" URL fragment, we can see the proper menu header is highlighted. And the proper entry is also highlighted if we enter the "#contacts" fragment.

Just to drive the point home once more: we're setting the active header when the routing controller executes a controller method. This may or may not be due to a new URL being hit. In other words, we do *not* depend on a route being triggered by a URL to highlight the appropriate header entry.

Let's take a look at our `AboutApp` file, to see this in practice:

assets/js/apps/about/about_app.js

```
1  ContactManager.module("AboutApp", function(AboutApp, ContactManager,
2  Backbone, Marionette, $, _){
3    AboutApp.Router = Marionette.AppRouter.extend({
4      appRoutes: {
5        "about" : "showAbout"
6      }
7    });
8
9    var API = {
10     showAbout: function(){
11       AboutApp.Show.Controller.showAbout();
12       ContactManager.execute("set:active:header", "about");
13     }
14   };
15
16   ContactManager.on("about:show", function(){
17     ContactManager.navigate("about");
18     API.showAbout();
19   });
20
21   ContactManager.addInitializer(function(){
22     new AboutApp.Router({
23       controller: API
24     });
25   });
26 });
```

On lines 3-7, we declare routes: if a user uses the "#about" URL fragment as the entry point into our application, we execute `showAbout` (defined on lines 10-13). This, in turn, will call the proper controller method and highlight the desired menu header.

On lines 16-19, we declare an event handler: if our app triggers the "about:show" event, we update the URL fragment and execute `showAbout`. And again, this will call the proper controller method and highlight the desired menu header.

So by using this application structure, we can properly decouple application behavior from URL fragment management: our menu header will be highlighted whether the user arrived directly by a URL, or by navigating within our app.

Handling Menu Clicks

The last thing we need to modify in our header menu is managing user clicks on the various entries. Obviously, we'll start by handling the `click` event in our item view:

Handling user clicks on menu entries (assets/js/apps/header/list/list_view.js)

```
List.Header = Marionette.ItemView.extend({
  template: "#header-link",
  tagName: "li",

  events: {
    "click a": "navigate"
  },

  navigate: function(e){
    e.preventDefault();
    this.trigger("navigate", this.model);
  },

  onRender: function(){
    // code omitted for brevity
  }
});
```

All we do is trigger an event indicating which menu entry was clicked, and pass along the associated header model instance. We then process these events in our controller:

Processing navigation events (assets/js/apps/header/list/list_controller.js)

```
List.Controller = {
  listHeader: function(){
    var links = ContactManager.request("header:entities");
    var headers = new List.Headers({collection: links});

    headers.on("brand:clicked", function(){
      ContactManager.trigger("contacts:list");
    });

    headers.on("itemview:navigate", function(childView, model){
```

```
11        var url = model.get("url");
12        if(url === 'contacts'){
13          ContactManager.trigger("contacts:list");
14        }
15        else if(url === "about"){
16          ContactManager.trigger("about:show");
17        }
18        else{
19          throw "No such sub-application: " + url;
20        }
21      });
22
23      ContactManager.headerRegion.show(headers);
24    },
25
26    setActiveHeader: function(headerUrl){
27      // code omitted for brevity
28    }
29  };
```

On lines 10-21, we listen for the "navigate" event triggered by the child item views. When such an event is triggered, we take a look at the relevant header model's `url` property and trigger the corresponding routing event. Since our sub-applications already listen for these events and know how to react to them, that's all there is for us to do. Our header menu is now fully managed using non-persisted models and events!

But we can still improve this somewhat by moving the triggers within our models, like so:

Adding the trigger to the header entities (assets/js/entities/header.js)

```
1  var initializeHeaders = function(){
2    Entities.headers = new Entities.HeaderCollection([
3      { name: "Contacts", url: "contacts", navigationTrigger: "contacts:list" },
4      { name: "About", url: "about", navigationTrigger: "about:show" }
5    ]);
6  };
```

Having the models store the triggers we want to use simplifies our code:

Processing navigation events (assets/js/apps/header/list/list_controller.js)

```
1 headers.on("itemview:navigate", function(childView, model){
2   var trigger = model.get("navigationTrigger");
3   ContactManager.trigger(trigger);
4 });
```

Git commit implementing the Header sub-application:

fef637d0bfc49e2846afd4b6f3f83480caa3347b[174]

[174] https://github.com/davidsulc/marionette-gentle-introduction/commit/fef637d0bfc49e2846afd4b6f3f83480caa3347b

Closing Thoughts

We've now built a complete javascript application with Backbone Marionette. You should now have a good grasp of how Marionette's various pieces fit together, 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[175]. Help others determine if the book is right for them!

Would you like me to cover another subject in an upcoming book? Let me know by email at davidsulc@gmail.com 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/mailing_list[176]. No spam, I promise!

You can also follow me on Twitter: @davidsulc

[175] https://leanpub.com/marionette-gentle-introduction/feedback
[176] http://davidsulc.com/mailing_list

Other Books I've Written

If you want to take the Marionette knowledge you've acquired with this book a little further, check out my next book: Structuring Backbone Code with RequireJS and Marionette Modules[177]. It takes the Contact Manager application you've developed, and rewrites it to use RequireJS:

- manage dependencies;
- load templates from separate files (templates are no longer in `index.html`!);
- explains typical errors and how you can approach debugging them;
- how to produce a single, optimized, and minifed javascript file of your application that is ready for production.

And best of all, it is written in an exercise style: each chapter introduces what you need to develop, and points out the various things to keep in mind (dependencies, loading order, etc.). You then add the new module to the application by yourself, and can check your answer by reading the step-by-step explanation that follows in the chapter.

[177] https://leanpub.com/structuring-backbone-with-requirejs-and-marionette

Module Architecture

On the next page, you'll find a graphical representation of the application architecture used in this book. What we're doing is breaking down complexity into manageable chunks, i.e. "divide and conquer".

The main application, `ContactManager`, is responsible mainly for:

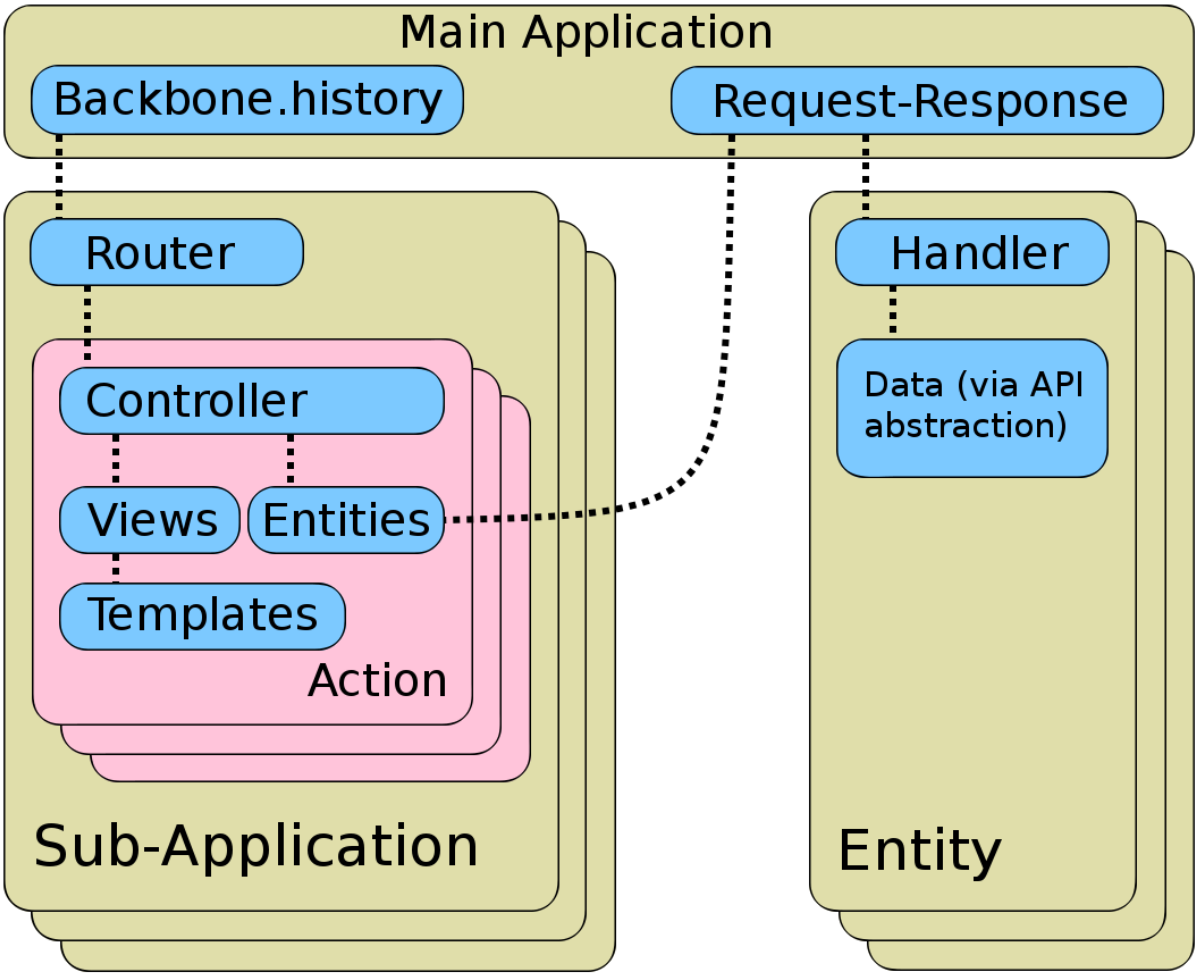
- starting `Backbone.history` which will listen for URL fragment changes (sometimes called "hash changes") and trigger routing code in our sub-applications (e.g. `ContactsApp`), which in turn executes the controller action in the appropriate sub-application's action module;
- providing the request-response mechanism through which we access our entities (such as the `Contact` model).

Each sub-application (e.g. `ContactsApp`) is further divided into actions (`List`, `Show`, etc.) responsible for a single goal (e.g. listing all contacts, displaying a single contact). Each of these actions has a controller to coordinate various elements in order to obtain the desired end result. Among those:

- fetching the required entities (i.e. the data that needs to be displayed) via the request-response mechanism;
- instantiating the proper views (and providing them with the requisite data);
- reacting to view events.

Views are provided the data they need when instantiated, and fetch the templates they need to display the information.

Entities register handlers to respond with requested data. They abstract APIs and transform the received data (usually JSON) into Backbone entities (i.e. models or collections).



Exercise Solutions

Displaying a Single-Item List

This is the solution to the exercise given at the end of chapter "Displaying a Static View".

Here's the relevant portion of `index.html`:

index.html

```
<div id="main-region" class="container">
  <p>Here is static content in the web page. You'll notice that it gets
  replaced by our app as soon as we start it.</p>
</div>

<script type="text/template" id="static-template">
  <p>This is text that was rendered by our Marionette app.</p>
</script>

<script type="text/template" id="list-item-template">
  <li>One item</li>
</script>

<!-- The javascript includes are here -->

<script type="text/javascript">
  var ContactManager = new Marionette.Application();

  ContactManager.addRegions({
    mainRegion: "#main-region"
  });

  ContactManager.StaticView = Marionette.ItemView.extend({
    tagName: "ul",
    template: "#static-template"
  });

  ContactManager.on("initialize:after", function(){
```

```
    var staticView = new ContactManager.StaticView({
      template: "#list-item-template"
    });
    ContactManager.mainRegion.show(staticView);
  });

  ContactManager.start();
</script>
```

We just add the specified template to our HTML (without removing the existing template, since it's refered to in our view definition), and set this new template when instantiating the view. The important part is that we specify a `tagName` attribute in the view, so it gets rendered within a `ul` element instead of the default `div` element.

Note that we can provide any view option at runtime, so we also could have kept the original view definition by defining the `tagName` when instantiating the view:

```
  ContactManager.StaticView = Marionette.ItemView.extend({
    template: "#static-template"
  });

  ContactManager.on("initialize:after", function(){
    var staticView = new ContactManager.StaticView({
      tagName: "ul",
      template: "#list-item-template"
    });
    ContactManager.mainRegion.show(staticView);
  });
```

Displaying a Contact With No Phone Number

This is the solution to the exercise given at the end of chapter "Displaying a Model".

All we need is to alter our model definition to include a default value for the `phoneNumber` attribute:

```
ContactManager.Contact = Backbone.Model.extend({
  defaults: {
    phoneNumber: "No phone number!"
  }
});
```

Then, we create a model instance without a phone number, and pass it to the view:

```
var alice = new ContactManager.Contact({
  firstName: "Alice",
  lastName: "Arten"
});

var aliceView = new ContactManager.ContactView({
  model: alice
});
```

Which gives us the following javascript code in our `index.html`:

```
var ContactManager = new Marionette.Application();

ContactManager.addRegions({
  mainRegion: "#main-region"
});

ContactManager.Contact = Backbone.Model.extend({
  defaults: {
    phoneNumber: "No phone number!"
  }
});

ContactManager.ContactView = Marionette.ItemView.extend({
  template: "#contact-template",

  events: {
    "click p": "alertPhoneNumber"
  },

  alertPhoneNumber: function(){
    alert(this.model.escape("phoneNumber"));
  }
});

ContactManager.on("initialize:after", function(){
  var alice = new ContactManager.Contact({
    firstName: "Alice",
    lastName: "Arten"
  });

  var aliceView = new ContactManager.ContactView({
```

```
32      model: alice
33    });
34
35    ContactManager.mainRegion.show(aliceView);
36  });
37
38  ContactManager.start();
```

Sorting a Collection

This is the solution to the exercise given at the end of chapter "Displaying a Collection of Models".

As indicated in the documentation[178], we can use either a sortBy[179] function or a sort[180] comparator function that expects 2 arguments.

Using SortBy

```
1  ContactManager.ContactCollection = Backbone.Collection.extend({
2    model: ContactManager.Contact,
3
4    comparator: function(contact) {
5      return contact.get("firstName") + " " + contact.get("lastName");
6    }
7  });
```

As you can see, the `comparator` attribute is bound to a function taking a single argument. The collection will then be sorted based on the function's return value.

You'll notice we're using the model's `get` method, and not `escape`: as we're not displaying data, there is no need for escaping HTML, and escaping the model's attributes would interfere with the sorting.

Using Sort

[178] http://backbonejs.org/#Collection-comparator
[179] http://underscorejs.org/#sortBy
[180] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/sort

```
ContactManager.ContactCollection = Backbone.Collection.extend({
  model: ContactManager.Contact,

  comparator: function(a, b) {
    var aFirstName = a.get("firstName");
    var bFirstName = b.get("firstName");
    if(aFirstName === bFirstName){
      var aLastName = a.get("lastName");
      var bLastName = b.get("lastName");
      if(aLastName === bLastName){ return 0; }
      if(aLastName < bLastName){ return -1; }
      else{ return 1; }
    }
    else{
      if(aFirstName < bFirstName){ return -1; }
      else{ return 1; }
    }
  }
});
```

As you can tell, this solution is slightly more involved, but it's also a lot more powerful. Essentialy, the function takes 2 contacts (`a` and `b`), and has to return:

- `-1` if `a` should come before `b`
- `1` if `b` should come before `a`
- `0` if `a` and `b` are equivalent

So what we're doing here is returning `-1` or `1` (as appropriate), and if the first names are identical, we sort based on the last name (again returning the proper integer value). You can learn more about this type of comparator functions at MDN[181].

Declaring a Template Sub-Module

This is the solution to the exercise given at the end of chapter "Structuring Code with Modules".

We'll simply declare a `ContactsApp.List.Templates` sub-module attached to our `ContactManager` app, and attach a `listItemView` attribute to it:

[181] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/sort

```
ContactManager.module("ContactsApp.List.Templates",
          function(Templates, ContactManager, Backbone, Marionette, $, _){
  Templates.listItemView = "<%= firstName %> <%= lastName %>";
});
```

As an aside, this is the code used in chapter "Overriding Marionette's Template Loader" here.

Building your own CompositeView

This is the solution to the exercise given in chapter "Using a CompositeView".

The strategy we'll use to solve this problem is simple: use a composite view containing both the `p` tag and an empty `ul` tag to receive our child views.

Template for the `CompositeView` in index.html

```
<script type="text/template" id="contact-list">
  <p>Here is the list of all the contacts we have information for:</p>
  <ul>
  </ul>
</script>
```

The composite view using this template is defined with:

The `Contacts` view in assets/js/apps/contacts/list/list_view.js

```
List.Contacts = Marionette.CompositeView.extend({
  template: "#contact-list",
  itemView: List.Contact,
  itemViewContainer: "ul"
});
```

Since the entire composite view is rendered within a `div`, we don't even need to specify a `tagName` within our view: `div` is the default tag. We do however, specify the `itemView` to use when rendering the models in our collection (line 3). Also, we instruct our view to put the rendered child views within the template's `ul` element by specifying the `itemViewContainer` on line 4.

The `Contact` item view is essentially the same one we used to display `li` items previously:

The `Contact` view in assets/js/apps/contacts/list/list_view.js

```
List.Contact = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-list-item"
});
```

And the associated template:

Template for the list items in index.html

```
<script type="text/template" id="contact-list-item">
  <%= firstName %> <%= lastName %>
</script>
```

Displaying the Contents of a Clicked Table Cell

This is the solution to the exercise given at the end of chapter "Using a CompositeView".

To achieve the desired functionality, you need to have a `click` event handler for the `td` selector:

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

```
List.Contact = Marionette.ItemView.extend({
  tagName: "tr",
  template: "#contact-list-item",

  events: {
    "click": "highlightName",
    "click td": "alertCellText"
  },

  highlightName: function(e){
    this.$el.toggleClass("warning");
  },

  alertCellText: function(e){
    alert($(e.target).text());
  }
});
```

We define the event listener on line 7, and associate it with the `alertCellText` handler. Our `alertCellText` function defined on lines 14-16 will now be fired any time a `td` element in our view is clicked. Bear in mind that our view here is an item view consisting of a single table row.

When defining event selectors, it's important to be aware of the view you are working with and what DOM elements it contains: event listeners defined on DOM elements that aren't in the view will never get triggered... For example, when you're defining event handlers in an `ItemView`, don't try to register handlers for events on DOM elements located in the parent `CompositeView`.

On line 15, we finish the job by wrapping the event target in a jQuery object, and retrieving its `text` content. This text value is then displayed in an alert.

Event Bubbling from Child Views

This is the solution to the exercise given in chapter "Events, Bubbling, and TriggerMethod".

First, we modify the `highlightName` handler so it triggers an event instead of toggling a CSS class:

Modifying the `highlightName` to trigger an event (assets/js/apps/contacts/list/list_view.js)

```
1  List.Contact = Marionette.ItemView.extend({
2    // tagName and template attributes
3
4    // events object
5
6    highlightName: function(e){
7      this.trigger("contact:highlighting:toggled", this.model);
8    },
9
10   // deleteClicked handler
11 });
```

Then, we add an event listener to our view:

Dumping the highlighted controller (assets/js/apps/contacts/list/list_controller.js)

```
List.Controller = {
  listContacts: function(){
    var contacts = ContactManager.request("contact:entities");

    var contactsListView = new List.Contacts({
      collection: contacts
    });

    contactsListView.on("itemview:contact:delete",
    function(childView, model){
      contacts.remove(model);
    });

    contactsListView.on("itemview:contact:highlighting:toggled",
    function(childView, model){
      console.log("Highlighting toggled on model: ", model);
    });

    ContactManager.mainRegion.show(contactsListView);
  }
}
```

Using the Triggers Object

Since the only thing we wish to do when the user clicks a row is to trigger an event, we can use a `trigger` object[182], which works similarly to the `events` object:

Using the trigger object (assets/js/apps/contacts/list/list_view.js)

```
List.Contact = Marionette.ItemView.extend({
  // tagName and template attributes

  events: {
    "click td a": "deleteClicked"
  },

  triggers: {
```

[182] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.view.md#viewtriggers

```
9       "click": "contact:highlighting:toggled"
10    },
11
12    // deleteClicked handler
13  });
```

The "contact:highlighting:toggled" event is now triggered when the user clicks a row. We still need to adapt our controller slightly:

Processing the event args (assets/js/apps/contacts/list/list_controller.js)

```
1  List.Controller = {
2    listContacts: function(){
3      // code here stays the same
4
5      contactsListView.on("itemview:contact:highlighting:toggled",
6      function(args){
7        console.log("Highlighting toggled on model: ", args.model);
8      });
9
10     // show the view instance
11   }
12 }
```

Events triggered via a view's `trigger` object are provided with a single object with `view`, `model`, and `collection` properties matching the same properties on the view that triggered the event, so:

- on line 6, we need to change our callback to take only one argument
- on line 7, we access our model via the property on the `args` object

You can't have both an event handler *and* a trigger handler responding to the same selector. In that case, simply manually trigger the event from within your event handler.

Getting Back to the Contacts List

This is the solution to the exercise given at the end of chapter "Implementing Routing".

First, let's add the link to our template:

Modifying the `contact-view` template in index.html

```
<script type="text/template" id="contact-view">
  <h1><%= firstName %> <%= lastName %></h1>
  <p><a href="#contacts" class="js-list-contacts">
                              Display contacts list</a></p>
  <p><strong>Phone number:</strong> <%= phoneNumber %></p>
</script>
```

Then, let's respond to this link being clicked, by triggering an event from our view:

Responding to the click (assets/js/apps/contacts/show/show_view.js)

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
Backbone, Marionette, $, _){
  Show.Contact = Marionette.ItemView.extend({
    template: "#contact-view",

    events: {
      "click a.js-list-contacts": "listContactsClicked"
    },

    listContactsClicked: function(e){
      e.preventDefault();
      ContactManager.trigger("contacts:list");
    }
  });
});
```

Thanks to our centralized routing management, our routing controller is already responding to the "contacts:list" event (see here). So all we had to do was trigger the proper event and our app did the rest:

- update the URL to "#contacts"
- change the app's state and the information displayed

Overriding Marionette's Template Loader

In this chapter, we'll cover how to override Marionette functionality to better suit our needs. More specifically, we'll override the way templates are loaded so that we can remove them from our `index.html` and store them within modules.

This modification is carried out as a practical means to demonstrate overriding Marionette functionality. In a production environment, you'll be better off with implementing compiled templates (such as using sprockets[183] with Rails' asset pipeline).

That said, if you absolutely want to remove templates from your HTML and don't want to look into more advanced techniques (or even simple ones such as importing a file containing all of your views into `index.html`), then this technique might be an acceptable trade-off to you, as Marionette will compile the templates only once, and will then serve compiled templates from its cache.

Let's start with the code we had at commit 364c506187c2caa128f01a53493cab5b494ec431[184].

Our end objective is to be able to specify templates using module attributes, each containing our template as a string. Then, we'll need to modify Marionette's `loadTemplate` function to be able to take a template already in string form (instead of needing to use jQuery's `$()` function to select the DOM element.

How did we know what to override in Marionette? Simply by reading through Marionette's annotated source code[185]. The annotated source code for Marionette's `Marionette.Render.render` function states

> Render a template with data. The `template` parameter is passed to the `TemplateCache` object to retrieve the template function.

We're really interested in the "retrieve the template" part, so let's follow the source code and look at the `TemplateCache.get` function, which according to the documentation

> [gets] the specified template by `id`. Either retrieves the cached version, or loads it from the DOM.

[183] https://github.com/sstephenson/sprockets#javascript-templating-with-ejs-and-eco

[184] https://github.com/davidsulc/marionette-gentle-introduction/commit/364c506187c2caa128f01a53493cab5b494ec431

[185] http://marionettejs.com/docs/backbone.marionette.html

At the bottom of this function, we can see that it returns `cachedTemplate.load()`. The `load` function is the "internal method to load the template", and it in turn calls `loadTemplate`. Now, still quoting from the documentation, what `loadTemplate` does is

> load a template from the DOM, by default. Override this method to provide your own template retrieval

Bingo! Providing our own template retrieval is exactly what we want to do.

Declaring a Template Sub-Module

We'll want to access our contact's `listItemView` template from within a dedicated sub-module with

```
ContactManager.ContactsApp.List.Templates.listItemView
```

So we'll need to declare the following sub-module in a new file:

assets/js/apps/contacts/list_templates.js

```
ContactManager.module("ContactsApp.List.Templates",
          function(Templates, ContactManager, Backbone, Marionette, $, _){
  Templates.listItemView = "<%= firstName %> <%= lastName %>";
});
```

If you've been doing the exercises, you'll remember that this is essentially the solution to the exercise given at the end of chapter "Structuring Code with Modules".

Note that for longer templates, you can use an array to display template strings so they're easier to read:

```
ContactManager.module("ContactsApp.List.Templates",
          function(Templates, ContactManager, Backbone, Marionette, $, _){
  Templates.contactView = [
      "<h1><%= firstName %> <%= lastName %></h1>",
      "<p>",
          "<strong>Phone number:</strong><span><%= phonenumber %></span>",
      "</p>"
  ].join("\n");
});
```

This trick allows you to indent the template string without needing to escape line endings. Of course, when you're using Marionette, your templates should all remain very short, so they should remain manageable even without HTML syntax highlighting.

Tackling the Template Loader

First, here's the original Marionette `loadTemplate` code:

Marionette's original `Backbone.Marionette.TemplateCache.prototype.loadTemplate`

```
1  loadTemplate: function(templateId){
2    var template = Marionette.$(templateId).html();
3
4    if (!template || template.length === 0){
5      throwError("Could not find template: '" + templateId + "'",
6                 "NoTemplateError");
7    }
8
9    return template;
10 }
```

Let's quickly explain this. Within our views, we specify the template attribute with a jQuery selector such as "#contact-list-item". This attribute is provided to the `loadTemplate` function as the `templateId` argument. Then, on line 2, the template is selected with jQuery and it's `html()` content is retrieved. If no template is found, an error is raised with lines 4-7. Otherwise, we simply return the template.

The `templateId` argument must be provided as a string, because Marionette uses its values as a storage key in `Marionette.TemplateCache.get`.

Here's what we'll replace it with:

Our new `Backbone.Marionette.TemplateCache.prototype.loadTemplate`

```
1  Backbone.Marionette.TemplateCache.prototype.loadTemplate =
2  function(templateId){
3    var template;
4    if(templateId.charAt( 0 ) == "#"){
5      // If we request the template by providing a jQuery selector,
6      // behave as usual
7      template = Backbone.Marionette.$(templateId).html();
8    }
9    else{
```

```
    // Otherwise, load the template from our special sub-module.
    // We need to evaluate the argument to obtain the template string,
    // as the argument is passed as a string to be used as a
    // storage key in Marionette's template caching mechanism
    template = eval(templateId)
  }

  if (!template || template.length === 0){
      errorMessage = "Could not find template: '" + templateId + "'";
      throwError(errorMessage, "NoTemplateError");
  }

  return template;
}
```

As you can see from the comments, we first check if we're trying to load a template with a jQuery selector. If that's the case, we simply do the same thing as before (namely, fetch the HTML contents in the template DOM element). Otherwise, we `eval` the provided string so that the module attribute's value will be fetched for us. Then, we simply check for errors and return the template, same as before.

We need to provide the full path the the `loadTemplate` function, so that it gets overridden properly.

So now that we have this overridden function, where do we put it? We'll put it in

`assets/js/apps/config/marionette/templateCache.js`

since that's a good location to store files that will override or otherwise modify Marionette's behavior. With that done, we still need to modify our `index.html` to include the new javascript files, and remove the unused template:

index.html

```
<div id="main-region" class="container">
  <p>Here is static content in the web page. You'll notice that it gets
  replaced by our app as soon as we start it.</p>
</div>

<script src="./assets/js/vendor/jquery.js"></script>
<script src="./assets/js/vendor/json2.js"></script>
```

```
<script src="./assets/js/vendor/underscore.js"></script>
<script src="./assets/js/vendor/backbone.js"></script>
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<!-- our file overriding the template cache's loadTemplate function -->
<script src="./assets/js/apps/config/marionette/templateCache.js"></script>

<script src="./assets/js/app.js"></script>
<script src="./assets/js/entities/contact.js"></script>
<!-- our templates sub-module -->
<script src="./assets/js/apps/contacts/list/list_templates.js"></script>
<script src="./assets/js/apps/contacts/list/list_view.js"></script>
<script src="./assets/js/apps/contacts/list/list_controller.js"></script>

<script type="text/javascript">
  ContactManager.start();
</script>
```

Specifying our new Template

Last, but not least, we need to change the template specification within our view:

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

```
ContactManager.module("ContactsApp.List", function(List, ContactManager,
Backbone, Marionette, $, _){
  List.Contact = Marionette.ItemView.extend({
    tagName: "li",
    template: "ContactManager.ContactsApp.List.Templates.listItemView"
  });

  List.Contacts = Marionette.CollectionView.extend({
    tagName: "ul",
    itemView: List.Contact
  });
});
```

Now, if you refresh the page, you'll see our app still working, but now it's using our custom template loading mechanism.

Extending Marionette

Here's where we left our `app.js` file after refactoring our Routings Helpers:

assets/js/app.js

```
var ContactManager = new Marionette.Application();

ContactManager.addRegions({
  mainRegion: "#main-region"
});

ContactManager.navigate = function(route,  options){
  options || (options = {});
  Backbone.history.navigate(route, options);
};

ContactManager.getCurrentRoute = function(){
  return Backbone.history.fragment
};

ContactManager.on("initialize:after", function(){
  if(Backbone.history){
    Backbone.history.start();

    if(this.getCurrentRoute() === ""){
      this.navigate("contacts");
      ContactManager.ContactsApp.List.Controller.listContacts();
    }
  }
});
```

Our plan is now to extend Marionette's `Application` prototype[186] to add the `navigate` and `getCurrentRoute` functions. We'll put this code in a new file to keep our code well organized:

[186] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Inheritance_and_the_prototype_chain

Extending Marionette's `Application` prototype (assets/js/apps/config/marionette/application.js)

```
(function(Backbone){
  _.extend(Backbone.Marionette.Application.prototype, {
    navigate: function(route, options){
      options || (options = {});
      Backbone.history.navigate(route, options);
    },

    getCurrentRoute: function(){
      return Backbone.history.fragment
    }
  });
}(Backbone));
```

We put our extension code within an IIFE[187], and use Underscore's `extend`[188] to add our new functions to the `Application` protoype as discussed above.

Don't forget to include this new file in `index.html`, right after Marionette:

Including our Marionette extension in index.html

```
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/apps/config/marionette/application.js"></script>
```

Now, we can remove those functions from our `app.js` file:

assets/js/app.js

```
var ContactManager = new Marionette.Application();

ContactManager.addRegions({
  mainRegion: "#main-region"
});

ContactManager.on("initialize:after", function(){
  if(Backbone.history){
```

[187] http://en.wikipedia.org/wiki/Immediately-invoked_function_expression

[188] http://underscorejs.org/#extend

```
    Backbone.history.start();

    if(this.getCurrentRoute() === ""){
      this.navigate("contacts");
      ContactManager.ContactsApp.List.Controller.listContacts();
    }
  }
});
```

Using Web Storage for Persistence

As mentioned in chapter Dealing with Persisted Data, we'll be implementing data persistence by leveraging web storage[189]. To accomplish this, we'll be using the `Backbone.localStorage`[190] adapter.

Web storage used to be named *local storage*, hence the adapter's name. To keep things consistent, I'll refer to it in the source code as *local* storage instead of *web* storage, but they are one and the same.

We want to minimize the impact of storing our data locally (as opposed to a remote server). In other words, we'd like collections and models to "look normal" by defining the same attributes they currently do when extending the Backbone base entities. To achieve this minimal intrusion, we'll be using a mixin to define the web storage details.

We'll be using the code base as it was at the begining of chapter Dealing with Persisted Data.

Git commit we'll start with to implement web storage persistence:

4b932caf88f7026e8414d6801ffa6877881a012e[191]

Implementation Strategy

Before proceeding, refer to section Adding Location to our Entities for the explanation on entity `url` and `urlRoot` properties.

Web storage needs a key to store data, just like the entities need a location to be indicated among their properties (via the `url` and/or `urlRoot` values). Since we want to have minimal impact on our entities' API, we'll reuse the `url` and `urlRoot` properties to serve as our storage keys.

Adding to the Entities Module

Since we'll be adding functionality to our entities, let's create a new file to extend our `Entities` module:

[189] http://www.html5rocks.com/en/features/storage

[190] https://github.com/jeromegn/Backbone.localStorage

[191] https://github.com/davidsulc/marionette-gentle-introduction/commit/4b932caf88f7026e8414d6801ffa6877881a012e

Extending the `Entities` module (assets/js/apps/config/storage/localstorage.js)

```
1 ContactManager.module("Entities", function(Entities, ContactManager,
2 Backbone, Marionette, $, _){
3 });
```

And we'll need to add it to our `index.html` (line 2):

Adding our new file to index.html

```
1 <script src="./assets/js/app.js"></script>
2 <script src="./assets/js/apps/config/storage/localstorage.js"></script>
3 <script src="./assets/js/entities/contact.js"></script>
```

Notice we need to load our storage configuration file *after* our `app.js`, since that's where we define the `ContactManager` variable that our `Entities` module belongs to.

Of course, we also need to add the `Backbone.localStorage`[192] adapter to our project. Download it from the provided url, and add it to the *assets/js/vendor* folder. And don't forget to include it in our `index.html` (line 5):

Adding the localstorage adapter to index.html

```
1 <script src="./assets/js/vendor/jquery.js"></script>
2 <script src="./assets/js/vendor/json2.js"></script>
3 <script src="./assets/js/vendor/underscore.js"></script>
4 <script src="./assets/js/vendor/backbone.js"></script>
5 <script src="./assets/js/vendor/backbone.localstorage.js"></script>
6 <script src="./assets/js/vendor/backbone.marionette.js"></script>
```

Using a Mixin with Underscore

Underscore provides an `extend` function[193] that allows "copy/pasting" the properties of a source object into a destination object. This is the same function we use when defining (e.g.) a collection: we extend Backbone's collection so that we can use the functions defined by Backbone.

So let's create a `configureStorage` function that will extend our entity prototypes:

[192] https://github.com/jeromegn/Backbone.localStorage

[193] http://underscorejs.org/#extend

Creating our `configureStorage` function (assets/js/apps/config/storage/localstorage.js)

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    var StorageMixin = {};
4
5    Entities.configureStorage = function(entity){
6      _.extend(entity.prototype, StorageMixin);
7    };
8  });
```

We define our new function on lines 5-7: it receives an entity (e.g. `Entities.Contact`), and extends that entity's prototype with the `StorageMixin` object. Of course the `StorageMixin` mixin object is currently empty, so we won't be extending our entity at all. We need to enrich the mixin object to make the entity use web storage. But what is needed?

If we refer to the documentation for `Backbone.localStorage`[194], we can see we just need to add a property to our entities:

```
// "SomeCollection" must be a unique name within your app.
localStorage: new Backbone.LocalStorage("SomeCollection")
```

So that's what we need to get our `StorageMixin` to do: set the `localStorage` property on the entity, using the proper storage key. As we discussed, this storage key will match the entity's `url` or `urlRoot` value. Since we're going to need to access the values of the specific entity we'll be extending, we're going to need a different `StorageMixin` object for each entity. Therefore, we really need to be able to create `StorageMixin` instances matching a given entity. Let's alter our `StorageMixin` to return a dynamically-created object:

Using the `StorageMixin` to create appropriate objects for extension (assets/js/apps/config/storage/localstorage.js)

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    var StorageMixin = function(entityPrototype){
4      return {
5        localStorage: new Backbone.LocalStorage("SomeCollection")
6      };
7    };
8
```

[194] https://github.com/jeromegn/Backbone.localStorage

```
  Entities.configureStorage = function(entity){
    _.extend(entity.prototype, new StorageMixin(entity.prototype));
  };
});
```

The `StorageMixin` instance always uses the same storage key ("SomeCollection"), but now it's a function: it can be adapted to each entity's prototype.

Determining the Storage Key

We need to determine the storage key according to each entity, so let's create a function for that:

Using `StorageMixin` to create appropriate objects for extension (assets/js/apps/config/storage/localstorage.js)

```
ContactManager.module("Entities", function(Entities, ContactManager,
Backbone, Marionette, $, _){
  var findStorageKey = function(entity){
    // use a model's urlRoot value
    if(entity.urlRoot){
      return entity.urlRoot;
    }
    // use a collection's url value
    if(entity.url){
      return entity.url;
    }

    throw new Error("Unable to determine storage key");
  };

  var StorageMixin = function(entityPrototype){
    var storageKey = findStorageKey(entityPrototype);
    return { localStorage: new Backbone.LocalStorage(storageKey) };
  };

  Entities.configureStorage = function(entity){
    _.extend(entity.prototype, new StorageMixin(entity.prototype));
  };
});
```

That's a great start, but both `url` and `urlRoot` can be defined as functions. So let's use Underscore's `result`[195] function to return the value or function result (as the case may be):

Adapting `StorageMixin` to handle function values (assets/js/apps/config/storage/localstorage.js)

```
ContactManager.module("Entities", function(Entities, ContactManager,
Backbone, Marionette, $, _){
  var findStorageKey = function(entity){
    // use a model's urlRoot value
    if(entity.urlRoot){
      return _.result(entity, "urlRoot");
    }
    // use a collection's url value
    if(entity.url){
      return _.result(entity, "url");
    }

    throw new Error("Unable to determine storage key");
  };

  var StorageMixin = function(entityPrototype){
    var storageKey = findStorageKey(entityPrototype);
    return { localStorage: new Backbone.LocalStorage(storageKey) };
  };

  Entities.configureStorage = function(entity){
    _.extend(entity.prototype, new StorageMixin(entity.prototype));
  };
});
```

We still need to manage the case where a model doesn't have a `urlRoot` defined, but belongs to a collection. In that case, we'll fall back to the collection's `url` value:

[195] http://underscorejs.org/#result

Our completed implementation of `Entities.configureStorage` (assets/js/apps/config/storage/localstorage.js)

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    var findStorageKey = function(entity){
4      // use a model's urlRoot value
5      if(entity.urlRoot){
6        return _.result(entity, "urlRoot");
7      }
8      // use a collection's url value
9      if(entity.url){
10       return _.result(entity, "url");
11     }
12     // fallback to obtaining a model's storage key from
13     // the collection it belongs to
14     if(entity.collection && entity.collection.url){
15       return _.result(entity.collection, "url");
16     }
17
18     throw new Error("Unable to determine storage key");
19   };
20
21   var StorageMixin = function(entityPrototype){
22     var storageKey = findStorageKey(entityPrototype);
23     return { localStorage: new Backbone.LocalStorage(storageKey) };
24   };
25
26   Entities.configureStorage = function(entity){
27     _.extend(entity.prototype, new StorageMixin(entity.prototype));
28   };
29 });
```

Great! Since we create a new `StorageMixin` instance to adapt to the entity we want to modify, we can access the url values we're interested in (see function `findStorageKey`), and return an object containing the proper `localStorage` property (line 23). This new `StorageMixin`, containing the proper `localStorage` attribute value, is then used to extend our entity's prototype.

All that's left to do in our code is call (e.g.) `Entities.configureStorage(Entities.Contact);` and our `Contact` entity will be configured to use web storage.

Git commit implementing the localstorage mixin:

f4e0c7c5034fc367696ed31a3fe0bbe9a84ad39e[196]

But how does it work?

All persistence-related functions get proxied through `Backbone.sync` (see documentation[197]). The `Backbone.localStorage` adapter simply overrides `Backbone.sync` to use web storage instead of a remote RESTful API.

[196] https://github.com/davidsulc/marionette-gentle-introduction/commit/f4e0c7c5034fc367696ed31a3fe0bbe9a84ad39e
[197] http://backbonejs.org/#Sync

Creating a FilteredCollection

Let's create a `FilteredCollection` object in our `Entities` module (based on Derick Bailey's code[198]): it's role will be to manage and encapsulate collection-filtering (we'll use it in chapter Filtering Contacts). When we instantiate a filtered collection, we'll need to provide the base collection (containing all the models), as well as a filtering function[199], like this:

Demonstrating the `FilteredCollection` options

```
var filteredContacts = new ContactManager.Entities.FilteredCollection({
  collection: contacts,
  filterFunction: function(filterCriterion){
    return function(contact){
      if(contact.get("firstName") === filterCriterion){
        return contact;
      }
    };
  }
});
```

Let's start simply, adding basic creation code to a new file:

assets/js/entities/common.js

```
ContactManager.module("Entities", function(Entities, ContactManager,
Backbone, Marionette, $, _){
  Entities.FilteredCollection = function(options){
    var original = options.collection;
    var filtered = new original.constructor();
    filtered.add(original.models);
    filtered.filterFunction = options.filterFunction;

    return filtered;
  };
});
```

[198] http://jsfiddle.net/derickbailey/7tvzF/

[199] http://underscorejs.org/#filter

What we're doing:

1. storing a reference to the original collection (containing all of the unfiltered model instances) in the `original` variable (line 4);
2. creating a `filtered` collection that will hold our filtered models (line 5). We call the original collection's constructor, in case it defined some custom behavior;
3. we add all of the models in the `original` collection to the `filtered` collection (line 6);
4. we store a reference to the `filterFunction` option, for later use (line 7). Attaching it to the `filtered` object instance (which gets returned on line 9), gives us the option to specify the filtering method after initializing our filtered collection (by attaching the function to the returned collection);
5. we return our `filtered` collection on line 9.

We've already mentioned we want to be able to call the `filter` method to filter the models. But Backbone collections already define a `where`[200] method implementing basic filtering, so we want to be able to call a `where` method also. Let's add an `applyFilter` function that will take a filter criterion and a filtering "strategy" (i.e. use the provided `filter` method, or the built-in `where` method). It looks like this:

`applyFilter` **in assets/js/entities/common.js**

```
1  var applyFilter = function(filterCriterion, filterStrategy){
2    var collection = original;
3    var criterion;
4    if(filterStrategy == "filter"){
5      criterion = filterCriterion.trim();
6    }
7    else{
8      criterion = filterCriterion;
9    }
10
11   var items = [];
12   if(criterion){
13     if(filterStrategy == "filter"){
14       if( ! filtered.filterFunction){
15         throw("Attempted to use 'filter' function, but none was defined");
16       }
17       var filterFunction = filtered.filterFunction(criterion);
18       items = collection.filter(filterFunction);
19     }
```

[200] http://backbonejs.org/#Collection-where

```
20      else{
21        items = collection.where(criterion);
22      }
23    }
24    else{
25      items = collection.models;
26    }
27
28    // store current criterion
29    filtered._currentCriterion = criterion;
30
31    return items;
32  };
```

Breaking it down, step by step:

1. on line 2, use the `original` collection as the source containing the unfiltered models;
2. on lines 3-9, we adapt the filter criterion: if we're going to use our custom `filterFunction`, we remove any whitespace from the term. If we're just going to proxy `where`, leave the criterion intact;
3. if we have a filtering criterion:
    1. if we want to filter using the provided `filterFunction`:
        1. we need to ensure it exists (lines 14-16);
        2. we generate a filtering function for our criterion on line 17;
        3. we filter our `collection` using the filtering function (line 18);
    2. otherwise, we're just proxying for the collection's own `where` implementation (on line 21);
4. if there's no filtering criterion, just return all of the unfiltered models from the `collection` (line 25);
5. store the filtering criterion for future reference on line 29;
6. on line 31, return the models that match the filtering criterion.

With this function in place, we can define `filter` and `where` methods on our `FilteredCollection`:

Adding `filter` and `where` methods (assets/js/entities/common.js)

```
1  filtered.filter = function(filterCriterion){
2    filtered._currentFilter = "filter";
3    var items = applyFilter(filterCriterion, "filter");
4
5    // reset the filtered collection with the new items
6    filtered.reset(items);
7    return filtered;
8  };
9
10 filtered.where = function(filterCriterion){
11   filtered._currentFilter = "where";
12   var items = applyFilter(filterCriterion, "where");
13
14   // reset the filtered collection with the new items
15   filtered.reset(items);
16   return filtered;
17 };
```

In each function, we simply store the current filtering strategy in the `_currentFilter` attribute, call `applyFilter` to filter the models, and reset our `filtered` collection with the filtered models. Easy, right?

Note that on lines 7 and 16, we're returning the filtered collection. This will enable chaining filtering calls, e.g. `myCollection.where(...).filter(...)`.

This is what our file looks like so far:

assets/js/entities/common.js

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    Entities.FilteredCollection = function(options){
4      var original = options.collection;
5      var filtered = new original.constructor();
6      filtered.add(original.models);
7      filtered.filterFunction = options.filterFunction;
8
```

```
9       var applyFilter = function(filterCriterion, filterStrategy){
10        var collection = original;
11        var criterion;
12        if(filterStrategy == "filter"){
13          criterion = filterCriterion.trim();
14        }
15        else{
16          criterion = filterCriterion;
17        }
18
19        var items = [];
20        if(criterion){
21          if(filterStrategy == "filter"){
22            if( ! filtered.filterFunction){
23              throw("Attempted to use 'filter' function,
24                      but none was defined");
25            }
26            var filterFunction = filtered.filterFunction(criterion);
27            items = collection.filter(filterFunction);
28          }
29          else{
30            items = collection.where(criterion);
31          }
32        }
33        else{
34          items = collection.models;
35        }
36
37        // store current criterion
38        filtered._currentCriterion = criterion;
39
40        return items;
41      };
42
43      filtered.filter = function(filterCriterion){
44        filtered._currentFilter = "filter";
45        var items = applyFilter(filterCriterion, "filter");
46
47        // reset the filtered collection with the new items
48        filtered.reset(items);
49        return filtered;
50      };
```

```
51
52      filtered.where = function(filterCriterion){
53        filtered._currentFilter = "where";
54        var items = applyFilter(filterCriterion, "where");
55
56        // reset the filtered collection with the new items
57        filtered.reset(items);
58        return filtered;
59      };
60
61      return filtered;
62    };
63  });
```

But we still have some cases to manage. First, if the original collection gets its models reset, we need to adapt the filtered list to reflect that:

Refiltering if the original collection is reset (assets/js/entities/common.js)

```
1  original.on("reset", function(){
2    var items = applyFilter(filtered._currentCriterion,
3                            filtered._currentFilter);
4
5    // reset the filtered collection with the new items
6    filtered.reset(items);
7  });
```

Pretty straightforward: we simply call `applyFilter` with the saved parameters, then reset the `filtered` collection with the new models.

But we also need to manage the case where *new* models are *added* to the original collection. Refiltering the entire collection in this case seems wasteful: instead, we'll filter the new models and add the ones that match the filtering criterion:

Filtering new models to add (assets/js/entities/common.js)

```
1  original.on("add", function(models){
2    var coll = new original.constructor();
3    coll.add(models);
4    var items = applyFilter(filtered._currentCriterion,
5                            filtered._currentFilter, coll);
6    filtered.add(items);
7  });
```

As you can see, we create a new collection with the added models on lines 2-3, and use `applyFilter` to filter out the models we can add to the `filtered` collection. Except `applyFilter` only uses the `original` collection for filtering, so we need to add a new argument to its signature:

Updating `applyFilter` to accept an optional 'collection' argument (assets/js/entities/common.js)

```
1  var applyFilter = function(filterCriterion, filterStrategy, collection){
2    var collection = collection || original;
3    // rest of applyFilter is unchanged
4  };
```

Since the `collection` argument is optional, we use the `original` collection as the fallback value if none is provided. And this is what our `FilteredCollection` now looks like:

assets/js/entities/common.js

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2  Backbone, Marionette, $, _){
3    Entities.FilteredCollection = function(options){
4      var original = options.collection;
5      var filtered = new original.constructor();
6      filtered.add(original.models);
7      filtered.filterFunction = options.filterFunction;
8
9      var applyFilter = function(filterCriterion,
10                                filterStrategy, collection){
11       var collection = collection || original;
12       var criterion;
13       if(filterStrategy == "filter"){
```

```
14          criterion = filterCriterion.trim();
15        }
16        else{
17          criterion = filterCriterion;
18        }
19
20        var items = [];
21        if(criterion){
22          if(filterStrategy == "filter"){
23            if( ! filtered.filterFunction){
24              throw("Attempted to use 'filter' function,
25                      but none was defined");
26            }
27            var filterFunction = filtered.filterFunction(criterion);
28            items = collection.filter(filterFunction);
29          }
30          else{
31            items = collection.where(criterion);
32          }
33        }
34        else{
35          items = collection.models;
36        }
37
38        // store current criterion
39        filtered._currentCriterion = criterion;
40
41        return items;
42      };
43
44      filtered.filter = function(filterCriterion){
45        filtered._currentFilter = "filter";
46        var items = applyFilter(filterCriterion, "filter");
47
48        // reset the filtered collection with the new items
49        filtered.reset(items);
50        return filtered;
51      };
52
53      filtered.where = function(filterCriterion){
54        filtered._currentFilter = "where";
55        var items = applyFilter(filterCriterion, "where");
```

```
56
57        // reset the filtered collection with the new items
58        filtered.reset(items);
59        return filtered;
60      };
61
62      // when the original collection is reset,
63      // the filtered collection will re-filter itself
64      // and end up with the new filtered result set
65      original.on("reset", function(){
66        var items = applyFilter(filtered._currentCriterion,
67                                filtered._currentFilter);
68
69        // reset the filtered collection with the new items
70        filtered.reset(items);
71      });
72
73      // if the original collection gets models added to it:
74      // 1. create a new collection
75      // 2. filter it
76      // 3. add the filtered models (i.e. the models that were added *and*
77      //     match the filtering criterion) to the `filtered` collection
78      original.on("add", function(models){
79        var coll = new original.constructor();
80        coll.add(models);
81        var items = applyFilter(filtered._currentCriterion,
82                                filtered._currentFilter, coll);
83        filtered.add(items);
84      });
85
86      return filtered;
87    };
88  });
```

Git commit creating FilteredCollection:

6ccc57810f8cd1327429bed4154b0d9b792db53c[201]

[201] https://github.com/davidsulc/marionette-gentle-introduction/commit/6ccc57810f8cd1327429bed4154b0d9b792db53c