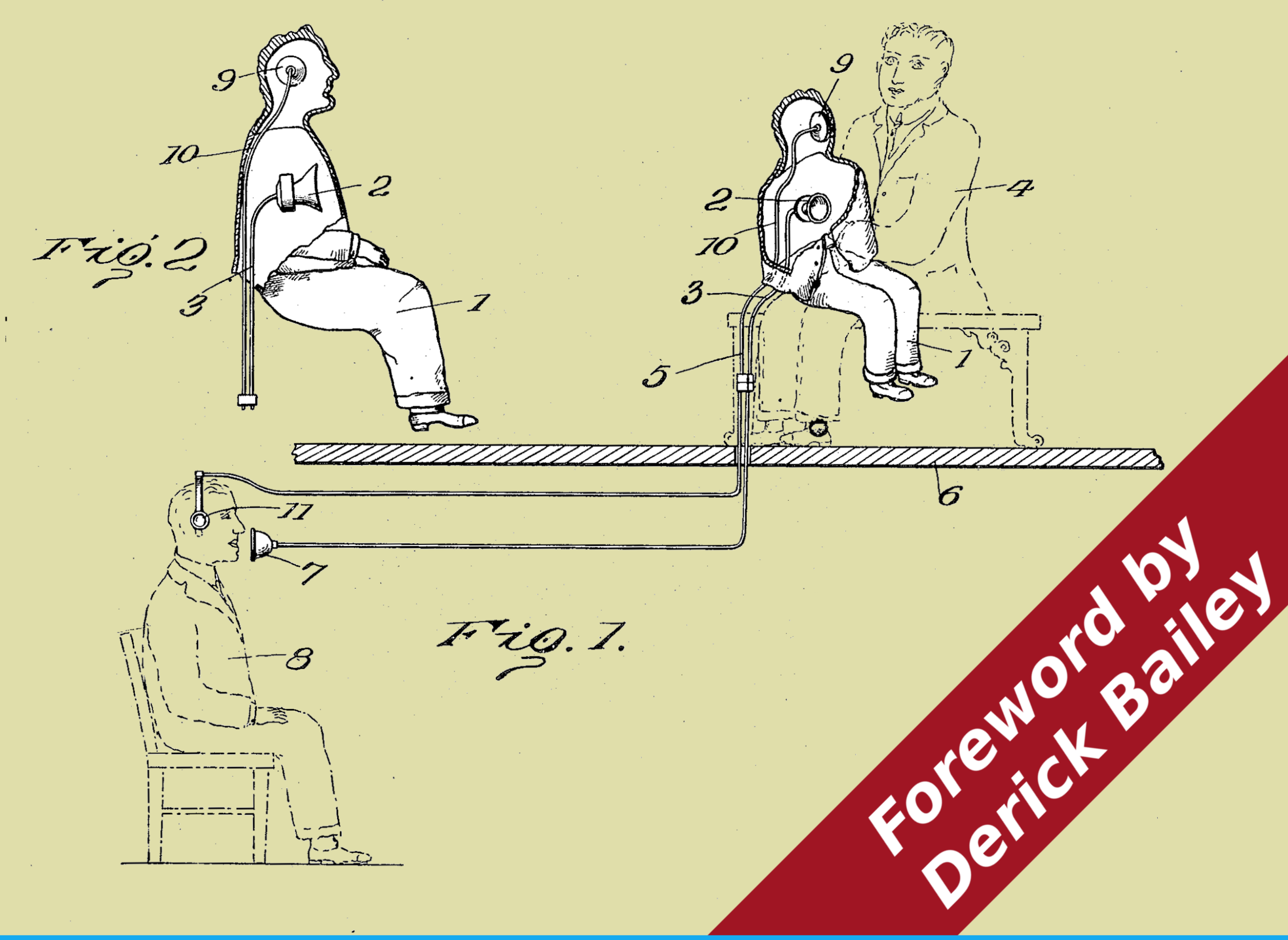
Build a Marionette.js app, one step at a time

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

©2013 - 2014 David Sulc

Also By David Sulc

Structuring Backbone Code with RequireJS and Marionette Modules

Backbone.Marionette.js: A Serious Progression

Contents

Implementing Routing **1**
How to Think About Routing 1
Adding a Router to ContactsApp 2
Routing Helpers 6
End of Preview Chapter 8

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`[1] 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.

[1] 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`[2], 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 });
```

[2]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[3] functionality by changing your `history` starting code to `Backbone.history.start({pushState: true});` (see documentation[4]).

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[5].

A great resource to read up on HTML5's History API is Dive Into HTML5[6], 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:

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

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

```
ContactManager.on("initialize:after", function(){
  if(Backbone.history){
    Backbone.history.start();

    if(Backbone.history.fragment === ""){
      Backbone.history.navigate("contacts");
      ContactManager.ContactsApp.List.Controller.listContacts();
    }
  }
});
```

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[7] 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:

[7] 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)

```
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();
    }
  }
});
```

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.

End of Preview Chapter

The chapter in the book contains additionalcontent, not included here due to spaceconstraints.