Cascading Select Boxes with Backbone.js: A Step-by-Step Tutorial

Posted on July 25, 2011 by Ben Teese

What happens when you don't use Backbone

Up until recent years, client-side Javascript development has resembled the wild-west from a software design perspective. Libraries like jQuery have certainly helped, but with the rise of Single-Page Applications, jQuery alone doesn't provide enough of an overall framework for large-scale client-side development. Fortunately, there's been a recent proliferation of Javascript MVC frameworks, both large and small. Backbone.js is one of these. It's lightweight, works with jQuery (although it doesn't need it) and seems to have some momentum behind it at the moment.

Backbone.js isn't particularly large or opinionated in the manner of say, Rails. For an expert, that might be a good thing. But for a beginner, it's not so good. The API documentation is complete, yet joining the dots can be a little intimidating at times to a newby. Simple tutorials abound that describe how to hook up a single view to a single model, but it's unclear what approach to use for more complex UIs. What should be in a model and what should be in a view? How should models and views interact with each other?

I have recently had the opportunity to work on a non-trivial Backbone.js application. In this entry I'll try to present an example that is slightly bigger than your average single model and view. Furthermore, I'll present it in a manner that is iterative, rather than just dropping the whole thing on you in one hit. You'll see that Backbone.js provides a good basis for building apps in an MVC style, although you will be faced with the same design decisions you'd have to make with any other MVC framework.

Note that this isn't an introduction to Backbone.js, and assumes a little background knowledge of how the framework works.

The Example

We'll use an example that's a little bigger than just a single view and model: the classic "cascading select" user interface. This is where we have a series of HTML `select` elements that represent data in a hierarchy. When you

select an item in one select box, the list of options in the next select box down the hierarchy is repopulated. Consequently, it's an example where – one way or another – views *have* to interact with one another. It's also an example that is surprisingly difficult to do correctly.

In this example, our hierarchy of select boxes will represent locations: specifically, countries, cities within those countries, and suburbs within those cities. This is what it'll look like in action:

Sure, it's not going to win any beauty contests, but that's not why we're here, is it?

Under the hood we've got a hierarchy of data structures, where a Country has many Cities, and a City has many Suburbs. Furthermore, we'll also assume that all of this data can be accessed as JSON via RESTful URLs in the Rails style. More specifically:

- `/countries` returns a list of all countries, each of which comprises an `id` and a `name`
- `/countries/[id]/cities` returns a list of all the cities within a country that has a particular `[id]`. Each city will have an `id` and `name`.
- `/cities/[id]/suburbs` returns a list of all suburbs within a city that has a particular `[id]`. Each suburb will have a `name` and `id`.

Getting Started

First, let's warm up with the basics: loading and displaying a list of countries. Let's start with the HTML:

```
<html>
    <head>
        <script type='text/javascript' src='javascripts/jquery-1.6.2.js'></script>
        <script type='text/javascript' src='javascripts/underscore.js'></script>
        <script type='text/javascript' src='javascripts/backbone.js'></script>
        <script type='text/javascript' src='javascripts/application.js'></script>
    </head>
    <body>
        <form>
            Country:<select id="country"><option value=''>Select</option></select>
            City:<select id="city" disabled="disabled"><option value=''>Select</option></select>
            Suburb:<select id="suburb" disabled="disabled"><option value=''>Select</option></select>
        </form>
    </body>
</html>
```

Now let's look at `application.js`, which uses Backbone to populate the 'Country' field:

```
$(function(){
    var Country = Backbone.Model.extend();
    var Countries = Backbone.Collection.extend({
        url: 'countries',
        model: Country
    });

    var CountryView = Backbone.View.extend({
        tagName: "option",

        initialize: function(){
            _.bindAll(this, 'render');
        },
        render: function(){
            $(this.el).attr('value', this.model.get('id')).html(this.model.get('name'));
            return this;
        }
```

```
    });
    var CountriesView = Backbone.View.extend({
        initialize: function(){
            _.bindAll(this, 'addOne', 'addAll');
            this.collection.bind('reset', this.addAll);
        },
        addOne: function(city){
            $(this.el).append(new CountryView({ model: city }).render().el);
        },
        addAll: function(){
            this.collection.each(this.addOne);
        }
    });

    var countries = new Countries();
    new CountriesView({el: $("#country"), collection: countries});
    countries.fetch();
});
```

I'm not going to drill right into this, as this isn't an introduction to Backbone.js. However, if you're having trouble understanding where the whole thing starts, a good place to begin is when `countries.fetch()` gets called. When this happens, the `countries` collection calls the `/countries` URL, takes the returned array of JSON objects, transforms them into `Country` model objects, populates itself with these objects, and then triggers a 'reset' event.

But that's only part of the story. The `CountriesView` object that has been created on the second-last line is listening for this 'reset' event, and invokes it's own `addAll` method in response. For each `Country` in the collection, this method creates a new `CountryView`, `render`s it and appends the result to the HTML element with ID 'country'. And that's how our UI actually gets updated.

Note that in this case, `CountryView#render` simply returns an HTML `option` element that contains the ID and name of the country. We could have used a HTML template to generate it, but in this instance it would have been overkill.

> Note for more advanced Backbone.js users*: Yes, I am aware that the JSON from `/countries` could instead just be embedded directly into the page and bootstrapped into the `countries` collection using `Backbone.Collection.reset()`, thus saving a call to the server. However I'm trying to keep things simple here and gently ease the reader into the general mechanism we will be using for loading data from the server – namely, `Backbone.Collection.fetch()`.*

Moving Right Along

The good news about our views is that they can be used to render the other select boxes as well. Let's take baby steps and introduce an additional collection for cities, generalising our view along the way. I've highlighted the lines that are changed or new:

```
$(function(){
    var Country = Backbone.Model.extend();
    var City = Backbone.Model.extend();


    var Countries = Backbone.Collection.extend({
        url: 'countries',
        model: Country
    });

    var Cities = Backbone.Collection.extend({
        model: City
    });

    var LocationView = Backbone.View.extend({
        tagName: "option",

        initialize: function(){
            _.bindAll(this, 'render');
```

```
        },
        render: function(){
            $(this.el).attr('value', this.model.get('id')).html(this.model.get('name'));
            return this;
        }
    });

    var LocationsView = Backbone.View.extend({
        initialize: function(){
            _.bindAll(this, 'addOne', 'addAll');
            this.collection.bind('reset', this.addAll);
        },
        addOne: function(location){
            $(this.el).append(new LocationView({ model: location }).render().el);
        },
        addAll: function(){
            this.collection.each(this.addOne);
        }
    });

    var countries = new Countries();

    new LocationsView({el: $("#country"), collection: countries});
    new LocationsView({el: $("#city"), collection: new Cities()});

    countries.fetch();
});
```

Of course, the list of cities still isn't being populated because we don't know what to populate it with. To do that, we need to detect when something is selected in the countries list, and take action. Furthermore, the country view is going to need to be able to access the city view.

So, without any further ado, I give you the next step in this process (note that I haven't included all of the code here – only those bits that are new, plus a little bit of context):

```
...
var LocationsView = Backbone.View.extend({
    events: {
        "change": "changeSelected"
    },
    ...
    changeSelected: function(){
        this.setSelectedId($(this.el).val());
    }
});

var CountriesView = LocationsView.extend({
    setSelectedId: function(countryId) {
        this.citiesView.collection.url = "countries/" + countryId + "/cities";
        this.citiesView.collection.fetch();
        $(this.citiesView.el).attr('disabled', false);
    }
});

var CitiesView = LocationsView.extend({
    setSelectedId: function(cityId) {
        // Do nothing - for now
    }
});

var countries = new Countries();

var countriesView = new CountriesView({el: $("#country"), collection: countries});
var citiesView = new CitiesView({el: $("#city"), collection: new Cities()});

countriesView.citiesView = citiesView;
...
```

In this updated code, we configure the `LocationView` to listen for 'change' events on its element. When this event is triggered, it calls the `setSelectedId` method with the new value. The implementation of this method is deferred to the subclass. In the case of the `CitiesView` subclass, the method does nothing. However, in the case of the `CountriesView` subclass, `setSelectedId` grabs the collection of cities and forces it to load its contents from the appropriate URL – which in-turn triggers a render of the cities via the process we discussed in the previous section. Finally, we actually enable the cities select box so that you can see the contents.

*Disclaimer: The code in this post is **not** optimized to minimize calls to the server. However,*

that's not the purpose – I'll leave it up to the reader to decide how they'd optimize it.

So to sum up, by giving one view access to another, we were able to get some basic interaction going. There's nothing magic about this – after all, the views are just Javascript objects that can have arbitrary properties set on them. If we wanted, we could access the views as global variables, but in my experience that can lead to all sorts of difficult-to-track bugs, and also means you can't really use the code anywhere else in your application. By minimizing the scope of my variables, I can, for example, have a completely different set of select boxes for navigating countries and cities, without having any cross-over effects with the existing select boxes:

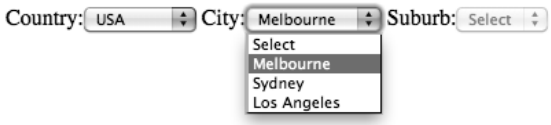
```
...
var anotherCountriesCollection = new Countries();

var anotherCountriesView = new CountriesView({el: $("#anotherCountry"), collection: anotherCountriesCollection});
var anotherCitiesView = new CitiesView({el: $("#anotherCity"), collection: new Cities()});

anotherCountriesView.citiesView = anotherCitiesView;

anotherCountriesCollection.fetch();
...
```

To my mind, that's one of the great things about a framework like Backbone.js – by giving you a simple pattern for grouping together the code for views and models, it also enables you to control scope, which in-turn facilities reuse.

However, before we get too carried away with the reuse possibilities , we need to do some more work to make sure these select boxes actually work properly.

Clearing And Repopulating

Unfortunately, the solution as it stands won't hold up to much scrutiny. For example, if you try select another country, then the list of cities will just have more entries added to it, rather than being cleared out and repopulated. As a result, the select boxes can end up looking like this:



That's not cool.

To get around this, whenever we render some locations, we need to keep track of each location view whose elements we add to the DOM. That way, when we have to render a new set of location views, we can remove the old elements first. It looks something like this:

```
...
var LocationsView = Backbone.View.extend({
    ...
    addOne: function(location){
        var locationView = new LocationView({ model: location });
        this.locationViews.push(locationView);
        $(this.el).append(locationView.render().el);
    },
    addAll: function(){
        _.each(this.locationViews, function(locationView) { locationView.remove() });
        this.locationViews = [];
        this.collection.each(this.addOne);
    },
    ...
});
...
```

That was easy! In case you're wondering, note that `_.each` won't do anything if `this.locationViews` hasn't been defined yet.

Extending out to the Suburbs

Now that we've got the basics going, let's add support for setting the suburbs whenever a city is selected:

```
...
var CitiesView = LocationsView.extend({
    setSelectedId: function(cityId) {
        this.suburbsView.collection.url = "cities/" + cityId + "/suburbs";
        this.suburbsView.collection.fetch();
        $(this.suburbsView.el).attr('disabled', false);
    }
});

var SuburbsView = LocationsView.extend({
    setSelectedId: function(cityId) {
        // Do nothing
    }
});

var countries = new Countries();

var countriesView = new CountriesView({el: $("#country"), collection: countries});
var citiesView = new CitiesView({el: $("#city"), collection: new Cities()});
var suburbsView = new SuburbsView({el: $("#suburb"), collection: new Suburbs()});

countriesView.citiesView = citiesView;
citiesView.suburbsView = suburbsView;
...
```

Nothing new to report here really. It is kind of cool though that we get the logic for populating – and repopulating – the cities select box so easily by subclassing the `LocationsView` again.

However, there's still one part of this that's a bit dodgy – if we select a country and a city, then go back and select a *new* country, the city select box will get repopulated, but the suburb box will stay just as it is:

I told you it was hard to do this right, didn't I?

To do it properly, we should probably both clear out the suburbs collection *and* disable the suburb select box:

```
...
var CountriesView = LocationsView.extend({
    setSelectedId: function(countryId) {
        this.citiesView.collection.url = "countries/" + countryId + "/cities";
        this.citiesView.collection.fetch();
        $(this.citiesView.el).attr('disabled', false);

        this.suburbsView.collection.reset();
        $(this.suburbsView.el).attr('disabled', true);
    }
});
...
countriesView.citiesView = citiesView;
countriesView.suburbsView = suburbsView;
citiesView.suburbsView = suburbsView;
...
```

Note the use of the `Backbone.Collection.reset()` function. This clears out the collection and triggers a 'reset' event – which we already know the views are listening for.

Removing Duplication

At this point you're probably starting to notice some duplication in the view code. There are many options to

remove this duplication, and they can grow more complex as the UI grows more complex. In this case, it would probably suffice for us to just to pull a couple of methods up to the `Locations` superclass:

```
...
var LocationsView = Backbone.View.extend({
    ...
    populateFrom: function(url) {
        this.collection.url = url;
        this.collection.fetch();
        this.setDisabled(false);
    },
    setDisabled: function(disabled) {
        $(this.el).attr('disabled', disabled);
    }
});

var CountriesView = LocationsView.extend({
    setSelectedId: function(countryId) {
        this.citiesView.populateFrom("countries/" + countryId + "/cities");

        this.suburbsView.collection.reset();
        this.suburbsView.setDisabled(true);
    }
});

var CitiesView = LocationsView.extend({
    setSelectedId: function(cityId) {
        this.suburbsView.populateFrom("cities/" + cityId + "/suburbs");
    }
});
...
```

Bonus Points: Setting a Suburb Directly

It's all well and good for us to be able to enable the user to select a suburb by drilling-down through countries and cities, but what if we want to pre-select a suburb for them? Not only do we need to select the suburb, we also need to select the city that the suburb is in, and the country that the city is in. Furthermore, we need to populate the suburbs list with all of the other suburbs that are in the city, and we need to populate the cities list with all of the cities that are in the country.

To implement this, let's introduce a couple more RESTful URLS:

- `/cities/[id]` returns the city that has a particular `[id]`. The city will have an `id`, `name` and `country_id`.
- `/suburbs/[id]` returns the suburb that has a particular `[id]`. The suburb will have a `name`, `id` and `city_id`.

Once we've got these in place, we can adopt a brute-force approach to the problem: given a particular suburb ID, look up the full record for the suburb and get its city ID. Then look up all of the other suburbs in that city and display them, selecting the suburb whilst we're at it. Then repeat this process up the hierarchy: look up the full record for the city, get it's country ID, lookup all the other cities in the country, and display them too. Let's have a first crack at automatically loading the suburb with the ID `3`:

```
$(function(){
    var Country = Backbone.Model.extend();
    var City = Backbone.Model.extend({urlRoot:'cities'});
    var Suburb = Backbone.Model.extend({urlRoot:'suburbs'});
    ...
    // Lots of unchanged code here
    ...
    countries.fetch();

    var suburbId = 3;

    new Suburb({id:suburbId}).fetch({success: function(suburb){
        var cityId = suburb.get('city_id');
        suburbsView.populateFrom("cities/" + cityId + "/suburbs");
        $(suburbsView.el).val(suburbId);

        new City({id: cityId}).fetch({success: function(city){
            var countryId = city.get('country_id');
            citiesView.populateFrom("countries/" + countryId + "/cities");
```

```
            $(citiesView.el).val(cityId);
            $(countriesView.el).val(countryId)
        }));
    }));
});
```

You can see that our changes are at the top and bottom of the file. At the top, we had to set the `urlRoot` property on `City` and `Suburb` so that instances of these models could be loaded from the back-end on their own, instead of having to be part of a collection. At the bottom of the file, we fetch individual instances of the suburb and city by ID, populating the suburbs and cities lists as we go. Note that each of the `fetch` calls is provided with a callback, as they trigger an asynchronous call to the back-end server.

We also attempt to set the selected value in each of the views. However, when we run it, we see the following:

What's happening? The city and suburb select boxes have been populated with the correct lists, but the correct values haven't been selected. Why is this?

The reason is that the calls to `populateFrom` *also* trigger asynchronous calls to the back end. However, we're not waiting for these asynchronous calls to return before we attempt to select the values in the lists. Consequently, there's nothing in the lists, and our call to `val(...)` has no effect. In fact, the same can be said for the countries list; there's no guarantee that it'll be populated by the time we set the selected value in it. It's a race condition; in the screen-shot above, we got lucky and it happened to work.

To get around this, we have to be a bit more careful:

```
    ...
    var LocationsView = Backbone.View.extend({
        addAll: function(){
            ...
            this.collection.each(this.addOne);

            if (this.selectedId) {
                $(this.el).val(this.selectedId);
            }
        }
        ...
    });

    var CountriesView = LocationsView.extend({
        setSelectedId: function(countryId) {
            this.citiesView.selectedId = null;
            this.citiesView.populateFrom("countries/" + countryId + "/cities");

            this.suburbsView.collection.reset();
            this.suburbsView.setDisabled(true);
        }
    });

    var CitiesView = LocationsView.extend({
        setSelectedId: function(cityId) {
            this.suburbsView.selectedId = null;
            this.suburbsView.populateFrom("cities/" + cityId + "/suburbs");

        }
    });

    ...

    new Suburb({id:suburbId}).fetch({success: function(suburb){
        suburbsView.selectedId = suburb.id;
        var cityId = suburb.get('city_id');
        suburbsView.populateFrom("cities/" + cityId + "/suburbs");

        new City({id: cityId}).fetch({success: function(city){
            citiesView.selectedId = city.id;
            var countryId = city.get('country_id');
            citiesView.populateFrom("countries/" + countryId + "/cities");

```

```
            countriesView.selectedId = countryId;
            countries.fetch();
        }});
    }});
});
```

In this case, we set a `selectedId` property on each `LocationView`, *before* we attempt to populate it. Then, *after* the underlying collection has been populated and the rest of the view rendered, we use the `selectedId` to set the selected item. Finally, we make sure that we clear out the `selectedId` in a child view whenever somebody selects an element in its parent.

Now things works correctly when we load the page:

JOY!

Cleaning Up

We introduced a some duplication when we added this functionality, so let's go back and refactor it:

```
    ...
    var CountriesView = LocationsView.extend({
        setSelectedId: function(countryId) {
            this.suburbsView.selectedId = null;
            this.citiesView.setCountryId(countryId);

            this.suburbsView.collection.reset();
            this.suburbsView.setDisabled(true);
        }
    });

    var CitiesView = LocationsView.extend({
        setSelectedId: function(cityId) {
            this.suburbsView.selectedId = null;
            this.suburbsView.setCityId(cityId);
        },
        setCountryId: function(countryId) {
            this.populateFrom("countries/" + countryId + "/cities");
        }
    });

    var SuburbsView = LocationsView.extend({
        setSelectedId: function(cityId) {
            // Do nothing
        },
        setCityId: function(cityId) {
            this.populateFrom("cities/" + cityId + "/suburbs");
        }
    });
    ...
    new Suburb({id:suburbId}).fetch({success: function(suburb){
        suburbsView.selectedId = suburb.id;
        var cityId = suburb.get('city_id');
        suburbsView.setCityId(cityId);

        new City({id: cityId}).fetch({success: function(city){
            citiesView.selectedId = city.id;
            var countryId = city.get('country_id');
            citiesView.setCountryId(countryId);

            countriesView.selectedId = countryId;
            countries.fetch();
        }});
    }});
});
```

In this case we've extracted out the `CitiesView.setCountryId` and `SuburbsView.setCityId` functions. There's probably more scope for refactoring here, but let's not get too tricky.

Where to from here?

In this post we've seen how Backbone.js can be used to add some much-needed order to the problem of dynamically populating cascading-select boxes from the back-end.

There are many ways in-which this solution might be improved. One technique is to push more of the interaction logic into the collection classes, and make it that views only ever interact indirectly via their underlying collections. This means that it's the collections that have references to each other, rather than the views. In my experience, this results in less code, but the increased indirection makes it harder to predict and track the side-effects of changes.

Another option is to decouple components further by only interacting via both custom and pre-defined events – a capability built into the very core of Backbone.js. However, this again can make the code harder to follow. For a simple user interface like that in our example, it could well be overkill.

Those familiar with programming user interfaces will recognize that these sort of design decisions are universal to the process of building software with the MVC pattern, and not specific to Backbone.js. What Backbone.js does do, however, is provide an ideal base on-which to build complex user interfaces with Javascript; it's geared towards the peculiarities of the browser event and DOM model, but without locking you into any particular user interface library. By understanding the basic tools and patterns that it puts at your disposal, your code can scale to meet the complexity of your user interface. I look forward to seeing more people share their experiences building complex interfaces using frameworks like Backbone.js.

> *I've posted the code for the final version of this example, complete with the underlying Rails project.*

About Ben Teese
I'm a Senior Consultant at Shine Technologies. Find me on Twitter at @benteese.
View all posts by Ben Teese →

This entry was posted in AJAX, Javascript and tagged backbone.js. Bookmark the permalink.

6 Responses to *Cascading Select Boxes with Backbone.js: A Step-by-Step Tutorial*

Martin Drapeau *says:*
August 3, 2011 at 12:17 pm

Why didn't you use events to link your different collections? Wouldn't that have been cleaner?
For instance, on your Collections you could have defined a function `select(country)` to pick a country and then trigger an event `select`. That way on your Cities collection, you could have bound to the `select` event change on the Countries collection like this:
countries.bind('select', function(country) {
this.url = "countries/" + country.id + "/cities";
this.fetch();

});
Allows you to completely decouple your code and makes things much simpler. Don't you think?

Reply

Ben Teese *says:*
August 4, 2011 at 1:41 am

Hi Martin,

Thanks for your comment. I did experiment with having a custom 'select' event on my collections, but ran into a couple of issues, mainly with getting the UI to correctly clear out and disable child select boxes when a parent value is selected. That said, I didn't spend a huge amount of time on it before resorting to a more direct approach. I'd be interested to see it working with the approach you're advocating – feel free to fork https://github.com/shinetech/CascadingSelectsWithBackboneJS and let me know if you get it going. I can always follow up with a subsequent blog showing a better way to do it!

Cheers,

Ben

Reply

Nguyen Chien Cong (@nguyenchiencong) *says:*
August 17, 2011 at 3:25 am

To disable chile and correctly clear out, i think you should use.

event.stopImmediatePropagation();

cheers

martindrapeau *says:*
August 4, 2011 at 2:47 am

Hi Ben,
Well in http://www.planbox.com I did extend Backbone Model and Collection objects to do just that. Works pretty well. What I've learned is that it is paramount to keep each object as separate and independent. Otherwise you end up maintaining spaghetti code. Food for thought (no pun intended).
–Martin

Reply

jlleblanclanc *says:*
August 11, 2011 at 11:40 pm

Thanks for writing this tutorial! I just started using Backbone on a project. After getting it up and running on something basic, I had this nagging feeling of missing the code reusability I was expecting. Although I'm not specifically trying to solve the cascading dropdowns problem, this has been a help nonetheless.

Reply

Pingback: *Building a shared calendar with Backbone.js and FullCalendar: A step-by-step tutorial | The Shine Technologies Blog*