Backbone.js Wine Cellar Tutorial — Part 1: Getting Started

by **Christophe Coenraets** on December 6, 2011 in **Backbone.js**, **JQuery**, **REST**

One of the challenges when building nontrivial Web applications is that JavaScript's non-directive nature can initially lead to a lack of structure in your code, or in other words, a lack of… backbone. JavaScript is often written as a litany of free-hanging and unrelated blocks of code, and it doesn't take long before it becomes hard to make sense of the logic and organization of your own code.

Backbone.js is a lightweight framework that addresses this issue by adding structure to JavaScript-heavy Web applications.

Self-contained building blocks

Backbone.js provides several classes (Model, Collection, View, Router) that you can extend to define the building blocks of your application. To build an app with Backbone.js, you first create the Models, Collections, and Views of your application. You then bring these components to life by defining a "Router" that provides the entry points of your application through a set of (deep-linkable) URLs.

With Backbone.js, your code is organized in self-contained entities (Models, Collections, Views): No more free-hanging and unrelated blocks of code.

Data Binding

With Backbone.js, you bind Views to Models so that when a Model’s data changes, all the Views bound to that Model automatically re-render. No more complex UI synchronization code.

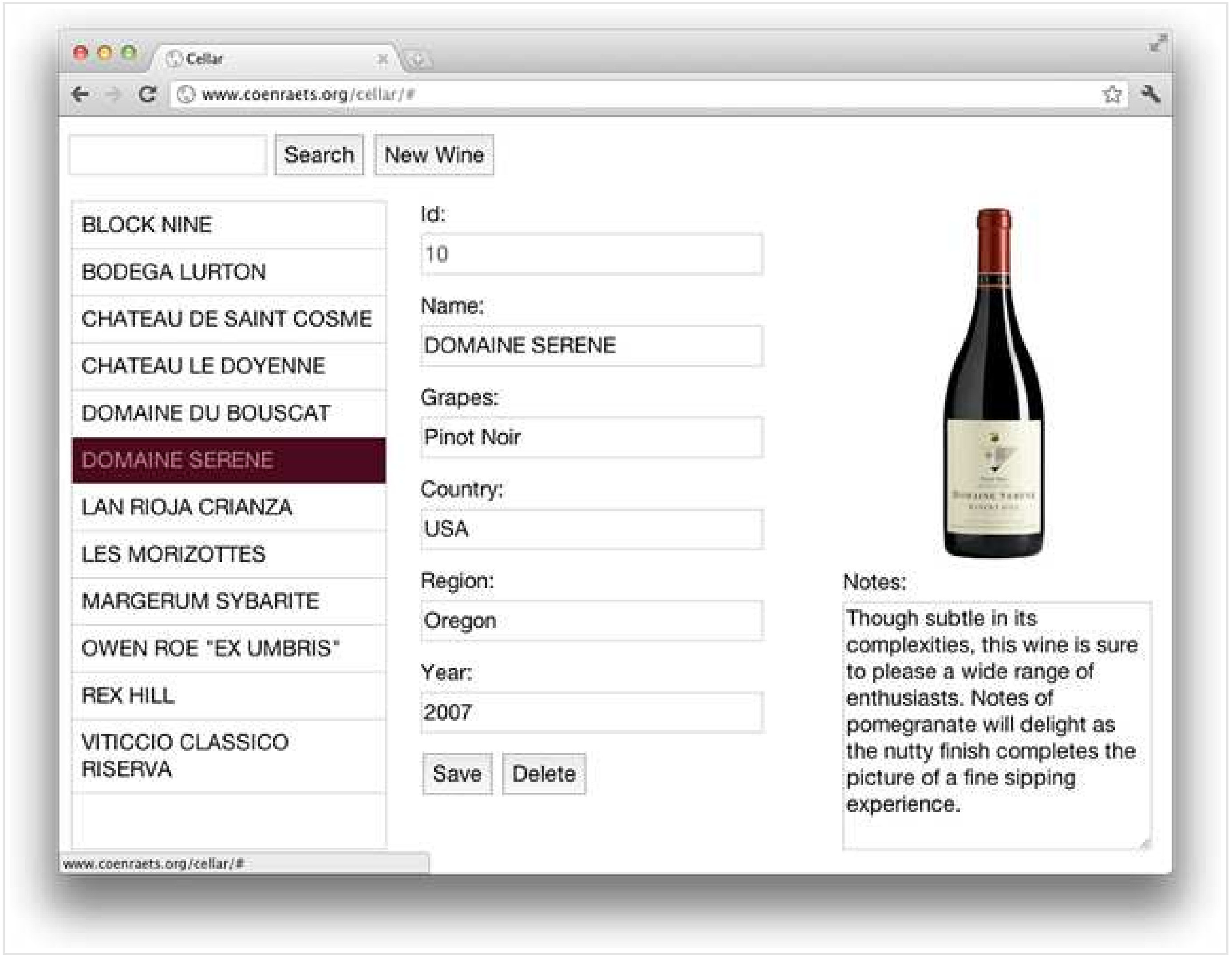
Elegant REST Integration

Backbone.js also provides a natural / magical / elegant integration with RESTful services. If your back-end data is exposed through a pure RESTful API, retrieving (GET), creating (POST), updating (PUT), and deleting (DELETE) models is incredibly easy using the Backbone.js simple Model API.

Sample Application

In this three-part tutorial, you’ll create a Wine Cellar application. You can browse through a list of wines, as well as add, update, and delete wines.

- In Part 1 (this post), you define the basic infrastructure. You create a “read-only” version of the application: you’ll be able to retrieve a list of wine and get the details of each wine.
- In Part 2, you add the code to add, update and delete wines. You leverage Backbone’s powerful REST integration.
- In Part 3, you add complete support for history management and deep linking.



BLOCK NINE
BODEGA LURTON
CHATEAU DE SAINT COSME
CHATEAU LE DOYENNE
DOMAINE DU BOUSCAT
DOMAINE SERENE
LAN RIOJA CRIANZA
LES MORIZOTTES
MARGERUM SYBARITE
OWEN ROE "EX UMBRIS"
REX HILL
VITICCIO CLASSICO RISERVA

Id:	10
Name:	DOMAINE SERENE
Grapes:	Pinot Noir
Country:	USA
Region:	Oregon
Year:	2007
Notes:	Though subtle in its complexities, this wine is sure to please a wide range of enthusiasts. Notes of pomegranate will delight as the nutty finish completes the picture of a fine sipping experience.

NOTE: I also blogged a non-Backbone version of the application here (Java back-end) and here (PHP back-end), which you can look at for comparison.

Part 1: The Read-Only Wine Cellar Application

You can run the application (Part 1) here.

Here is the code:

```
// Models
window.Wine = Backbone.Model.extend();

window.WineCollection = Backbone.Collection.extend({
    model:Wine,
    url:"../api/wines"
});

// Views
window.WineListView = Backbone.View.extend({

    tagName:'ul',

    initialize:function () {
        this.model.bind("reset", this.render, this);
    },

    render:function (eventName) {
        _.each(this.model.models, function (wine) {
            $(this.el).append(new WineListItemView({model:wine}).render().el);
        }, this);
        return this;
    }

});

window.WineListItemView = Backbone.View.extend({

    tagName:"li",

    template:_.template($('#tpl-wine-list-item').html()),

    render:function (eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    }

});

window.WineView = Backbone.View.extend({

    template:_.template($('#tpl-wine-details').html()),

    render:function (eventName) {
```

```
45            $(this.el).html(this.template(this.model.toJSON()));
46            return this;
47        }
48
49    });
50
51    // Router
52    var AppRouter = Backbone.Router.extend({
53
54        routes:{
55            "":"list",
56            "wines/:id":"wineDetails"
57        },
58
59        list:function () {
60            this.wineList = new WineCollection();
61            this.wineListView = new WineListView({model:this.wineList});
62            this.wineList.fetch();
63            $('#sidebar').html(this.wineListView.render().el);
64        },
65
66        wineDetails:function (id) {
67            this.wine = this.wineList.get(id);
68            this.wineView = new WineView({model:this.wine});
69            $('#content').html(this.wineView.render().el);
70        }
71    });
72
73    var app = new AppRouter();
74    Backbone.history.start();
```

Code Highlights:

1. **WineModel** (line 2): Notice that we don't need to explicitly define the attributes (name, country, year, etc). You could add validation, default values, etc. More on that in Part 2.
2. **WineCollection** (lines 4 to 7): "model" indicates the nature of the collection. "url" provides the endpoint for the RESTFul API. This is all that's needed to retrieve, create, update, and delete wines with Backbone's simple Model API.
3. **WineListView** (lines 10 to 25): The render() function iterates through the collection, instantiates a WineListItemView for each wine in the collection, and adds it to the wineList.

4. **WineListItemView** (lines 27 to 38): The render() function merges the model data into the "wine-list-item" template (defined in index.html). By defining a separate View for list items, you will make it easy to update (re-render) a specific list item when the backing model changes without re-rendering the entire list. More on that in Part 2.
5. **WineView** (lines 40 to 49): The view responsible for displaying the wine details in the Wine form. The render() function merges the model data (a specific wine) into the "wine-details" template retrieved from index.html.
6. **AppRouter** (lines 52 to 71): Provides the entry points for the application through a set of (deep-linkable) URLs. Two routes are defined: The default route ("") displays the list of wine. The "wines/:id" route displays the details of a specific wine in the wine form. Note that in Part 1, this route is not deep-linkable. You have to start the application with the default route and then select a specific wine. In Part 3, you will make sure you can deep-link to a specific wine.

Download

The source code for this application is hosted on GitHub here. And here is a quick link to the download.

You will need the RESTful services to run this application. A PHP version (using the Slim framework) is available as part of the download.

UPDATE (1/11/2012): A version of this application with a Java back-end (using JAX-RS and Jersey) is also available on GitHub here. You can find more information on the Java version of this application here.

Part 2 is available here.

Share this Article:

Subscribe

Related Posts:

Sample iOS Application with Xcode, Objective-C, Storyboard, and Core Data
Keypoint: PhoneGap-Based HTML Slide Decks
The Create Now Tour starts this Thursday in San Francisco
Building Modular Mobile/PhoneGap Apps with Backbone.js, RequireJS & Topcoat — Sample App
Building Modular Web Applications with Backbone.js and RequireJS — Sample App

‹ RESTful services with jQuery, PHP and the Slim Framework

Backbone.js Wine Cellar Tutorial — Part 2: CRUD ›