Getting Started with Backbone.js

Unlike its web development peers, JavaScript doesn't really have much in the way of frameworks to provide structure to its code. Thankfully, in recent years, that's beginning to change.

Today, I'd like to introduce you to Backbone.JS, a sweet little library that makes the process of creating complex, interactive and data driven apps so much easier. It provides a clean way to surgically separate your data from your presentation.

Overview of Backbone.JS

Created by Jeremy Ashkenas, the JS ninja who built CoffeeScript, Backbone is a super light-weight library that lets you create easy to maintain front ends. It's backend agnostic and works well with any of the modern JavaScript libraries you're already using.

Backbone is a collection of cohesive objects, weighing in at a shade under *4kb,* that lend structure to your code and basically helps you build a proper MVC app in the browser. The official site describes its purpose as so:

> Backbone supplies structure to JavaScript-heavy applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing application over a RESTful JSON interface.

Let's face it: the above is a little hard to parse and make sense of. So let's go ahead and deconstruct the jargon a bit, with help from Jeremy.

Key-value binding and custom events

When a model's contents or state is changed, other objects that have subscribed to the model are notified so they can proceed accordingly. Here, the views listen to changes in the model, and update themselves accordingly instead of the model having to deal with the views manually.

Rich API of enumerable functions

Backbone ships with a number of very useful functions for handling and working with your data. Unlike other implementation, arrays in JavaScript are pretty neutered, which really is a hampering problem when you have to deal with data.

Views with declarative event handling

Your days of writing spaghetti bind calls are over. You can programmatically declare which callback needs to be associated with specific elements.

RESTful JSON interface

Even though the default method is to use a standard AJAX call when you want to talk to the server, you can easily switch it out to anything you need. A number of adapters have sprung up covering most of the favorites including Websockets and local storage.

To break it down into even simpler terms:

> Backbone provides a clean way to surgically separate your data from your presentation. The model that works with the data is only concerned with synchronizing with a server while the view's primary duty is listening to changes to the subscribed model and rendering the HTML.

A Quick FAQ

I'm guessing you're probably a little fazed at this point, so let's clear a few things up:

Does it replace jQuery?

No. They're pretty complementary in their scopes with almost no overlaps in functionality. Backbone handles all the higher level abstractions, while jQuery – or similar libraries – work with the DOM, normalize events and so on.

Their scopes and use cases are pretty different and because you know one doesn't mean that you shouldn't learn the other. As a JavaScript developer, you should know how to effectively work with both.

Why should I be using this?

Because more often than not, the front end code devolves into a steaming, dirty pile of nested callbacks, DOM manipulations, HTML for the presentation amidst other unsayable acts.

Backbone offers a significantly clean and elegant way of managing this chaos.

Where should I be using this?

Backbone is ideally suited for creating front end heavy, data driven applications. Think the GMail interface, new Twitter or any other revelation of the past few years. It makes creating complex *apps* easier.

While you can shoehorn it for more mainstream web *pages*, this is really a library that is tailored for web apps.

Is it similar to Cappuccino or Sproutcore?

Yes and no.

Yes, because like the above mentioned frameworks, this is primarily intended for creating complex front ends for web applications.

It's dissimilar in that Backbone is quite lean, and ships with none of the widget that the others ship with.

> Backbone is incredibly light weight, at under 4kb.

There's also the fact that Cappuccino forces you to write your code in Objective-J, while Sproutcore's views have to be declared programmatically in JS. While none of those approaches are wrong, with Backbone, normal JavaScript is harnessed by your run of the mill HTML and CSS to get things done, leading to a gentler learning curve.

I can still use other libraries on the page, right?

Absolutely. Not only your typical DOM accessing, AJAX wrapping kind, but also the rest of your templating and script loading kind. It's very, very loosely coupled, which means you can use almost all of your tools in conjunction with Backbone.

Will it usher in world peace?

No, sorry. But here's something to cheer you up.

Ok, now with that out of the way, let's dive in!

Getting to Know Backbone's Backbone

The MVC in Backbone originally stood for Models, Views and Collections, since there were no controllers in the framework. This has since changed.

Backbone's core consists of four major classes:

- Model
- Collection
- View
- Controller

Since we're a little strapped for time, let's take a look at just the core classes today.

Model

Models can mean different things in different implementations of MVC. In Backbone, a model represents a singular entity — a record in a database if you will. But there are no hard and fast rules

here. From the Backbone website:

> Models are the heart of any JavaScript application, containing the interactive data as well as a large part of the logic surrounding it: conversions, validations, computed properties, and access control.

The model merely gives you a way to read and write arbitrary properties or attributes on a data set. With that in mind, the single liner below is completely functional:

```
var Game = Backbone.Model.extend({});
```

Let's build on this a bit.

```
var Game = Backbone.Model.extend({
initialize: function(){
alert("Oh hey! ");
},
defaults: {
name: 'Default title',
releaseDate: 2011,
}
});
```

`initialize` will be fired when an object is instantiated. Here, I'm merely alerting out inanities — in your app you should probably be bootstrapping your data or performing other housekeeping. I'm also defining a bunch of defaults, in case no data is passed.

Let's take a look at how to read and write attributes. But first, let's create a new instance.

```
// Create a new game
var portal = new Game({ name: "Portal 2", releaseData: 2011});

// release will hold the releaseDate value -- 2011 here
var release = portal.get('releaseDate');

// Changes the name attribute
portal.set({ name: "Portal 2 by Valve"});
```

If you noticed the get/set mutators, have a cookie! A model's attributes can't be read through your typical object.attribute format. You'll have to go through the getter/setter since there's a lower chance of you changing data by mistake.

At this point, all the changes are only kept in memory. Let's make these changes permanent by talking to the server.

```
portal.save();
```

That's it. Were you expecting more? The one-liner above will now send a request to your server. Keep in mind that the type of request will change intelligently. Since this is a fresh object, POST will be used. Otherwise, PUT is used.

There are a lot more features, Backbone models give you by default but this should definitely get you started. Hit the documentation for more information.

Collection

Collections in Backbone are essentially just a collection of models. Going with our database analogy from earlier, collections are the results of a query where the results consists of a number of records [models]. You can define a collection like so:

```
var GamesCollection = Backbone.Collection.extend({
model : Game,
}
});
```

The first thing to make note of is that we're defining which model this is a collection of. Expanding on our example earlier, I'm making this collection a collection of games.

Now you can go ahead and play around with your data to your hearts contents. For example, let's extend the collection to add a method that returns only specific games.

```
var GamesCollection = Backbone.Collection.extend({
model : Game,
old : function() {
return this.filter(function(game) {
return game.get('price') < 2009;
});
}
}
});
```

That was easy, wasn't it? We merely check if a game was released before 2009 and if so, return the game.

You can also directly manipulate the contents of a collection like so:

```
var games = new GamesCollection
games.get(0);
```

The above snippet instantiates a new collection and then retrieves the model with an ID of 0. You can find an element at a specific position through referencing the index to the *at* method like so: *games.at(0);*

And finally, you can dynamically populate your collection like so:

```
var GamesCollection = Backbone.Collection.extend({
model : Game,
url: '/games'
}
});

var games = new GamesCollection
games.fetch();
```

We're merely letting Backbone where to get the data from through the *url* property. With that done, we're merely creating a new object and calling the *fetch* method which fires of an asynchronous call to the server and populates the collection with the results.

That should cover the basics of collections with Backbone. As I mentioned, there are tons of goodies here what with Backbone aliasing a lot of nifty utilities from the Underscore library. A quick read through of the official documentation should get you started.

View

Views in Backbone can be slightly confusing at first glance. To MVC purists, they resemble a controller rather than a view itself.

A view handles two duties fundamentally:

- Listen to events thrown by the DOM and models/collections.
- Represent the application's state and data model to the user.

Let's go ahead and create a very simple view.

```
GameView= Backbone.View.extend({
tagName : "div",
className: "game",
render : function() {
// code for rendering the HTML for the view
}
});
```

Fairly simple if you've been following this tutorial so far. I'm merely specifying which HTML element should be used to wrap the view through the *tagName* attribute as well as the ID for it through *className*.

Let's move ahead to the rendering portion.

```
render : function() {
this.el.innerHTML = this.model.get('name');

//Or the jQuery way
$(this.el).html(this.model.get('name'));
}
```

el refers to the DOM element referenced by the view. We're simply accessing the game's name to the element's *innerHTML* property. To put it simply, the *div* element now contains the name of our game. Obviously, the jQuery way is simpler if you've used the library before.

With more complicated layouts, dealing with HTML within JavaScript is not only tedious but also foolhardy. In these scenarios, templating is the way to go.

Backbone ships with a minimal templating solution courtesy of Underscore.JS but you're more than welcome to use any of the excellent templating solutions available.

Finally, let's take a look at how views listen to events. DOM events first.

```
events: {
'click .name': 'handleClick'
},

handleClick: function(){
alert('In the name of science... you monster');

// Other actions as necessary
}
```

Should be simple enough if you've worked with events before. We're basically defining and hooking up *events* through the events object. As you can see above, the first part refers to the event, the next specifies the triggering elements while the last part refers to the function that should be fired.

And now onto binding to models and collections. I'll cover binding to models here.

```
GameView= Backbone.View.extend({
initialize: function (args) {
_.bindAll(this, 'changeName');
this.model.bind('change:name', this.changeName);
},
});
```

The first thing to note is how we're placing the binding code in the *initialize* functions. Naturally, it's best to do this from the get go.

bindAll is a utility provided by Underscore that persists the value of a function's *this* value. This is specially useful since we're passing a bunch of functions around and functions specified as callbacks have this value erased.

Now whenever a model's *name* attribute is changed, the *changeName* function is called. You can also make use of the *add* and *remove* verbs to poll for changes.

Listening to changes in a collections is as simple as replacing *model* with *collection* while binding the handler to the callback.

Controller

Controllers in Backbone essentially let you create bookmarkable, stateful apps by using hashbangs.

```
var Hashbangs = Backbone.Controller.extend({
routes: {
"!/": "root",
"!/games": "games",
},
root: function() {
// Prep the home page and render stuff
},

games: function() {
// Re-render views to show a collection of books
},
});
```

This is very familiar to routing in traditional serverside MVC frameworks. For example, */!/games* will map to the *games* function while the URL in the browser itself will be *domain/#!/games*.

Through intelligent use of hashbangs, you can create apps that are heavily JS based but also bookmarkable.

If you're worried about breaking the back button, Backbone has you covered too.

```
// Init the controller like so
var ApplicationController = new Controller;

Backbone.history.start();
```

With the above snippet, Backbone can monitor your hashbangs and in conjunction with the routes you've specified earlier, make your app bookmarkable.

What I Learned from Backbone

Overall, here are some lessons I learned from the Backbone way of creating applications:

- We really need MVC for the front end. Traditional methods leave us with code that's too coupled, messy and incredibly hard to maintain.
- Storing data and state in the DOM is a bad idea. This started making more sense after creating apps that needed different parts of the app to be updated with the same data.
- Fat models and skinny controllers are the way to go. Workflow is simplified when business logic is taken care of by models.
- Templating is an absolute necessity. Putting HTML inside your JavaScript gives you bad karma.

It's sufficient to say that Backbone has caused a paradigm shift in how front ends should be constructed, at least for me. Given the very broad scope of today's article, I'm sure you have a ton of questions. Hit the comments section below to chime in. Thank you so much for reading and expect a ton more Backbone tutorials in the future!

Original URL:

http://net.tutsplus.com/tutorials/javascript-ajax/getting-started-with-backbone-js/