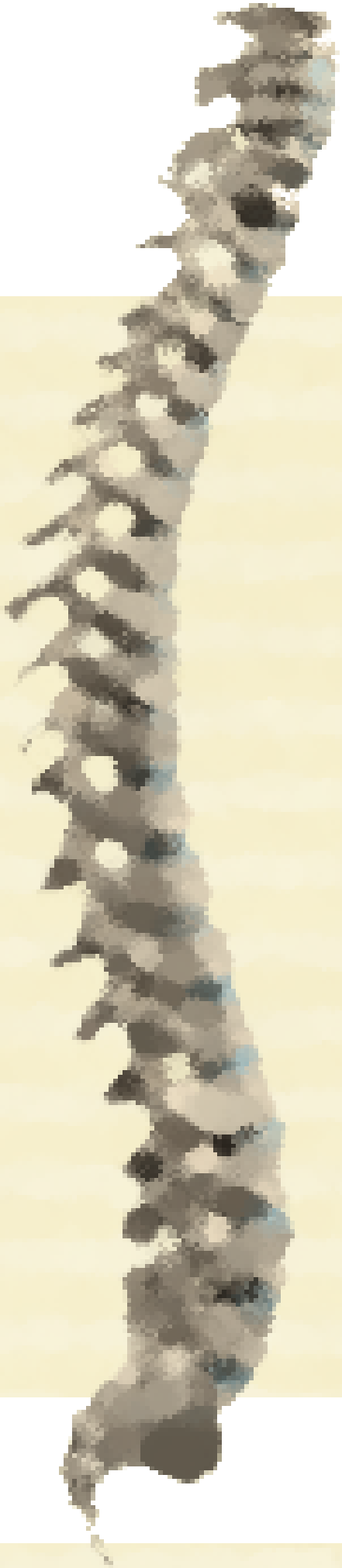
Backbone Tutorials



Thomas Davis

Backbone Tutorials

Beginner, Intermediate and Advanced

Thomas Davis

This book is for sale at http://leanpub.com/backbonetutorials

This version was published on 2012-10-23

This is a Leanpub book. Leanpub helps authors to self-publish in-progress ebooks. We call this idea Lean Publishing.

To learn more about Lean Publishing, go to http://leanpub.com/manifesto.

To learn more about Leanpub, go to http://leanpub.com.

Tweet This Book!

Please help Thomas Davis by spreading the word about this book on Twitter!

The suggested hashtag for this book is #backbonetutorials.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#backbonetutorials

Contents

Why do you need Backbone.js?

Building single-page web apps or complicated user interfaces will get extremely difficult by simply using jQuery[1] or MooTools[2]. The problem is standard JavaScript libraries are great at what they do - and without realizing it you can build an entire application without any formal structure. You will with ease turn your application into a nested pile of jQuery callbacks, all tied to concrete DOM elements.

I shouldn't need to explain why building something without any structure is a bad idea. Of course you can always invent your own way of structuring your application but you miss out on the benefits of the open source community.

Why single page applications are the future

Backbone.js enforces that communication to the server should be done entirely through a RESTful API. The web is currently trending such that all data/content will be exposed through an API. This is because the browser is no longer the only client, we now have mobile devices, tablet devices, Google Goggles and electronic fridges etc.

So how does Backbone.js help?

Backbone is an incredibly small library for the amount of functionality and structure it gives you. It is essentially MVC for the client and allows you to make your code modula r. If you read through some of the beginner tutorials the benefits will soon become self evident and due to Backbone.js light nature you can incrementally include it in any current or future projects.

Other frameworks

If you are looking for comparisons to build your single page application, try some of these resourceful links.

- A feature comparison of different frontend frameworks[3]
- Todo MVC - Todo list implemented in the many different types of frontend frameworks[4]

Contributors

- FND[5]

If you questions regarding why you should choose Backbone.js as your framework, please leave a comment below

[1] http://jquery.com
[2] http://mootools.net
[3] http://codebrief.com/2012/01/the-top-10-javascript-mvc-frameworks-reviewed/
[4] http://addyosmani.github.com/todomvc/
[5] https://github.com/FND

What is a view?

Backbone views are used to reflect what your applications' data models look like. They are also used to listen to events and react accordingly. This tutorial will not be addressing how to bind models and collections to views but will focus on view functionality and how to use views with a JavaScript templating library, specifically Underscore.js's _.template[6].

We will be using jQuery 1.8.2[7] as our DOM manipulator. It's possible to use other libraries such as MooTools[8] or Sizzle[9], but official Backbone.js documentation endorses jQuery. Backbone.View events may not work with other libraries other than jQuery.

For the purposes of this demonstration, we will be implementing a search box. A live example[10] can be found on jsFiddle.

```
    SearchView = Backbone.View.extend({
        initialize: function(){
            alert("Alerts suck.");
        }
    });

    // The initialize function is always called when instantiating a Backbo\
ne View.
    // Consider it the constructor of the class.
    var search_view = new SearchView();
```

The "el" property

The "el" property references the DOM object created in the browser. Every Backbone.js view has an "el" property, and if it not defined, Backbone.js will construct its own, which is an empty div element.

Let us set our view's "el" property to div#search_container, effectively making Backbone.View the owner of the DOM element.

```
    <div id="search_container"></div>

    <script type="text/javascript">
        SearchView = Backbone.View.extend({
            initialize: function(){
                alert("Alerts suck.");
            }
        });

        var search_view = new SearchView({ el: $("#search_container") });
    </script>
```

Note: Keep in mind that this binds the container element. Any events we trigger must be in this element.

[6] http://documentcloud.github.com/underscore/#template
[7] http://jquery.com/
[8] http://mootools.net/
[9] http://sizzlejs.com/
[10] http://jsfiddle.net/tBS4X/1/

Loading a template

Backbone.js is dependent on Underscore.js, which includes its own micro-templating solution. Refer to Underscore.js's documentation[11] for more information.

Let us implement a "render()" function and call it when the view is initialized. The "render()" function will load our template into the view's "el" property using jQuery.

```
    <script type="text/template" id="search_template">
      <label>Search</label>
      <input type="text" id="search_input" />
      <input type="button" id="search_button" value="Search" />
    </script>

    <div id="search_container"></div>

    <script type="text/javascript">
        SearchView = Backbone.View.extend({
            initialize: function(){
                this.render();
            },
            render: function(){
                // Compile the template using underscore
                var template = _.template( $("#search_template").html(), {}\
);
                // Load the compiled HTML into the Backbone "el"
                this.$el.html( template );
            }
        });

        var search_view = new SearchView({ el: $("#search_container") });
    </script>
```

Tip: Place all your templates in a file and serve them from a CDN. This ensures your users will always have your application cached.

Listening for events

To attach a listener to our view, we use the "events" attribute of Backbone.View. Remember that event listeners can only be attached to child elements of the "el" property. Let us attach a "click" listener to our button.

[11] http://documentcloud.github.com/underscore/

```
    <script type="text/template" id="search_template">
      <label>Search</label>
      <input type="text" id="search_input" />
      <input type="button" id="search_button" value="Search" />
    </script>

    <div id="search_container"></div>

    <script type="text/javascript">
        SearchView = Backbone.View.extend({
            initialize: function(){
                this.render();
            },
            render: function(){
                var template = _.template( $("#search_template").html(), {}\
 );
                this.$el.html( template );
            },
            events: {
                "click input[type=button]": "doSearch"
            },
            doSearch: function( event ){
                // Button clicked, you can access the element that was clic\
ked with event.currentTarget
                alert( "Search for " + $("#search_input").val() );
            }
        });

        var search_view = new SearchView({ el: $("#search_container") });
    </script>
```

Tips and Tricks

Using template variables

```
    <script type="text/template" id="search_template">
        <!-- Access template variables with <%= %> -->
        <label><%= search_label %></label>
        <input type="text" id="search_input" />
        <input type="button" id="search_button" value="Search" />
    </script>

    <div id="search_container"></div>

    <script type="text/javascript">
         SearchView = Backbone.View.extend({
            initialize: function(){
                this.render();
```

```
14              },
15              render: function(){
16                  //Pass variables in using Underscore.js Template
17                  var variables = { search_label: "My Search" };
18                  // Compile the template using underscore
19                  var template = _.template( $("#search_template").html(), va\
20  riables );
21                  // Load the compiled HTML into the Backbone "el"
22                  this.$el.html( template );
23              },
24              events: {
25                  "click input[type=button]": "doSearch"
26              },
27              doSearch: function( event ){
28                  // Button clicked, you can access the element that was clic\
29  ked with event.currentTarget
30                  alert( "Search for " + $("#search_input").val() );
31              }
32          });
33
34          var search_view = new SearchView({ el: $("#search_container") });
35      </script>
```

Relevant Links

- This example implemented with google API[12]
- This examples exact code on jsfiddle.net[13]
- Another semi-complete example on jsFiddle[14]

Contributors

- Michael Macias[15]
- Alex Lande[16]

[12] http://thomasdavis.github.com/2011/02/05/backbone-views-and-templates.html
[13] http://jsfiddle.net/thomas/C9wew/4/
[14] http://jsfiddle.net/thomas/dKK9Y/6/
[15] https://github.com/zaeleus
[16] https://github.com/lawnday

What is a model?

Across the internet the definition of MVC[17] is so diluted that it's hard to tell what exactly your model should be doing. The authors of backbone.js have quite a clear definition of what they believe the model represents in backbone.js.

> 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.

So for the purpose of the tutorial let's create a model.

```
1   Person = Backbone.Model.extend({
2       initialize: function(){
3           alert("Welcome to this world");
4       }
5   });
6
7   var person = new Person;
```

So *initialize()* is triggered whenever you create a new instance of a model(models, collections and views work the same way). You don't have to include it in your model declaration but you will find yourself using it more often than not.

Setting attributes

Now we want to pass some parameters when we create an instance of our model.

```
1   Person = Backbone.Model.extend({
2       initialize: function(){
3           alert("Welcome to this world");
4       }
5   });
6
7   var person = new Person({ name: "Thomas", age: 67});
8   // or we can set afterwards, these operations are equivelent
9   var person = new Person();
10  person.set({ name: "Thomas", age: 67});
```

So passing a JavaScript object to our constructor is the same as calling *model.set()*. Now that these models have attributes set we need to be able to retrieve them.

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

Getting attributes

Using the *model.get()* method we can access model properties at anytime.

```
    Person = Backbone.Model.extend({
        initialize: function(){
            alert("Welcome to this world");
        }
    });

    var person = new Person({ name: "Thomas", age: 67, child: 'Ryan'});

    var age = person.get("age"); // 67
    var name = person.get("name"); // "Thomas"
    var child = person.get("child"); // 'Ryan'
```

Setting model defaults

Sometimes you will want your model to contain default values. This can easily be accomplished by setting a property name ‘defaults’ in your model declaration.

```
    Person = Backbone.Model.extend({
        defaults: {
            name: 'Fetus',
            age: 0,
            child: ''
        },
        initialize: function(){
            alert("Welcome to this world");
        }
    });

    var person = new Person({ name: "Thomas", age: 67, child: 'Ryan'});

    var age = person.get("age"); // 67
    var name = person.get("name"); // "Thomas"
    var child = person.get("child"); // 'Ryan'
```

Manipulating model attributes

Models can contain as many custom methods as you like to manipulate attributes. By default all methods are public.

```
    Person = Backbone.Model.extend({
        defaults: {
            name: 'Fetus',
            age: 0,
            child: ''
        },
        initialize: function(){
            alert("Welcome to this world");
        },
        adopt: function( newChildsName ){
            this.set({ child: newChildsName });
        }
    });

    var person = new Person({ name: "Thomas", age: 67, child: 'Ryan'});
    person.adopt('John Resig');
    var child = person.get("child"); // 'John Resig'
```

So we can implement methods to get/set and perform other calculations using attributes from our model at any time.

Listening for changes to the model

Now onto one of the more useful parts of using a library such as backbone. All attributes of a model can have listeners bound to them to detect changes to their values. In our initialize function we are going to bind a function call everytime we change the value of our attribute. In this case if the name of our "person" changes we will alert their new name.

```
    Person = Backbone.Model.extend({
        defaults: {
            name: 'Fetus',
            age: 0
        },
        initialize: function(){
            alert("Welcome to this world");
            this.on("change:name", function(model){
                var name = model.get("name"); // 'Stewie Griffin'
                alert("Changed my name to " + name );
            });
        }
    });

    var person = new Person({ name: "Thomas", age: 67});
    person.set({name: 'Stewie Griffin'}); // This triggers a change and wil\
l alert()
```

So we can bind the change listener to individual attributes or if we like simply '*this.on("change", function(model){});*' to listen for changes to all attributes of the model.

Interacting with the server

Models are used to represent data from your server and actions you perform on them will be translated to RESTful operations.

The `id` attribute of a model identifies how to find it on the database usually mapping to the surrogate key[18].

For the purpose of this tutorial imagine that we have a mysql table called `Users` with the columns `id`, `name`, `email`.

The server has implemented a RESTful URL `/user` which allows us to interact with it.

Our model definition shall thus look like;

```
    var UserModel = Backbone.Model.extend({
        urlRoot: '/user',
        defaults: {
            name: '',
            email: ''
        }

    });
```

Creating a new model

If we wish to create a new user on the server then we will instantiate a new UserModel and call `save`. If the `id` attribute of the model is `null`, Backbone.js will send send of POST request to the server to the urlRoot.

```
    var UserModel = Backbone.Model.extend({
        urlRoot: '/user',
        defaults: {
            name: '',
            email: ''
        }
    });
    var user = new Usermodel();
    // Notice that we haven't set an `id`
    var userDetails = {
        name: 'Thomas',
        email: 'thomasalwyndavis@gmail.com'
    };
    // Because we have not set a `id` the server will call
    // POST /user with a payload of {name:'Thomas', email: 'thomasalwyndavi\
s@gmail.com'}
    // The server should save the data and return a response containing the\
 new `id`
    user.save(userDetails, {
```

[18] http://en.wikipedia.org/wiki/Surrogate_key

```
        success: function (user) {
            alert(user.toJSON());
        }
    })
```

Our table should now have the values

1, 'Thomas', 'thomasalwyndavis@gmail.com'

Getting a model

Now that we have saved a new user model, we can retrieve it from the server. We know that the `id` is 1 from the above example.

If we instantiate a model with an `id`, Backbone.js will automatically perform a get request to the urlRoot + '/id' (conforming to RESTful conventions)

```
    // Here we have set the `id` of the model
    var user = new Usermodel({id: 1});

    // The fetch below will perform GET /user/1
    // The server should return the id, name and email from the database
    user.fetch({
        success: function (user) {
            alert(user.toJSON());
        }
    })
```

Updating a model

Now that we model that exist on the server we can perform an update using a PUT request. We will use the `save` api call which is intelligent and will send a PUT request instead of a POST request if an `id` is present(conforming to RESTful conventions)

```
    // Here we have set the `id` of the model
    var user = new Usermodel({
        id: 1,
        name: 'Thomas',
        email: 'thomasalwyndavis@gmail.com'
    });

    // Let's change the name and update the server
    // Because there is `id` present, Backbone.js will fire
    // PUT /user/1 with a payload of `{name: 'Davis', email: 'thomasalwynda\
vis@gmail.com'}`
    user.save({name: 'Davis'}, {
        success: function (model) {
            alert(user.toJSON());
        }
    });
```

Deleting a model

When a model has an `id` we know that it exist on the server, so if we wish to remove it from the server we can call `destroy`. `destroy` will fire off a DELETE /user/id (conforming to RESTful conventions).

```
    // Here we have set the `id` of the model
    var user = new Usermodel({
        id: 1,
        name: 'Thomas',
        email: 'thomasalwyndavis@gmail.com'
    });

    // Because there is `id` present, Backbone.js will fire
    // DELETE /user/1
    user.destroy({
        success: function () {
            alert('Destroyed');
        }
    });
```

Tips and Tricks

Get all the current attributes

```
    var person = new Person({ name: "Thomas", age: 67});
    var attributes = person.toJSON(); // { name: "Thomas", age: 67}
    /* This simply returns a copy of the current attributes. */

    var attributes = person.attributes;
    /* The line above gives a direct reference to the attributes and you sh\
ould be careful when playing with it.   Best practise would suggest that yo\
u use .set() to edit attributes of a model to take advantage of backbone li\
steners. */
```

Validate data before you set or save it

```
    Person = Backbone.Model.extend({
        // If you return a string from the validate function,
        // Backbone will throw an error
        validate: function( attributes ){
            if( attributes.age < 0 && attributes.name != "Dr Manhatten" ){
                return "You can't be negative years old";
            }
        },
        initialize: function(){
            alert("Welcome to this world");
            this.bind("error", function(model, error){
                // We have received an error, log it, alert it or forget it\
```

```
13  :)
14                  alert( error );
15              });
16          }
17      });
18
19      var person = new Person;
20      person.set({ name: "Mary Poppins", age: -1 });
21      // Will trigger an alert outputting the error
22
23      var person = new Person;
24      person.set({ name: "Dr Manhatten", age: -1 });
25      // God have mercy on our souls
```

Contributors

- Utkarsh Kukreti[19]

[19] https://github.com/utkarshkukreti

What is a collection?

Backbone collections are simply an ordered set of models[20]. Such that it can be used in situations such as;

- Model: Student, Collection: ClassStudents
- Model: Todo Item, Collection: Todo List
- Model: Animals, Collection: Zoo

Typically your collection will only use one type of model but models themselves are not limited to a type of collection;

- Model: Student, Collection: Gym Class
- Model: Student, Collection: Art Class
- Model: Student, Collection: English Class

Here is a generic Model/Collection example.

```
var Song = Backbone.Model.extend({
    initialize: function(){
        console.log("Music is the answer");
    }
});

var Album = Backbone.Collection.extend({
    model: Song
});
```

Building a collection

Now we are going to populate a collection with some useful data.

[20] http://backbonetutorials.com/what-is-a-model

```
    var Song = Backbone.Model.extend({
        defaults: {
            name: "Not specified",
            artist: "Not specified"
        },
        initialize: function(){
            console.log("Music is the answer");
        }
    });

    var Album = Backbone.Collection.extend({
        model: Song
    });

    var song1 = new Song({ name: "How Bizarre", artist: "OMC" });
    var song2 = new Song({ name: "Sexual Healing", artist: "Marvin Gaye" })\
;
    var song3 = new Song({ name: "Talk It Over In Bed", artist: "OMC" });

    var myAlbum = new Album([ song1, song2, song3]);
    console.log( myAlbum.models ); // [song1, song2, song3]
```

What is a router?

Backbone routers are used for routing your applications URL's when using hash tags(#). In the traditional MVC sense they don't necessarily fit the semantics and if you have read "What is a view?[21]" it will elaborate on this point. Though a Backbone "router" is still very useful for any application/feature that needs URL routing/history capabilities.

Defined routers should always contain at least one route and a function to map the particular route to. In the example below we are going to define a route that is always called.

Also note that routes interpret anything after "#" tag in the URL. All links in your application should target "#/action" or "#action". (Appending a forward slash after the hashtag looks a bit nicer e.g. http://example.com/#/user/help)

```
    <script>
        var AppRouter = Backbone.Router.extend({
            routes: {
                "*actions": "defaultRoute" // matches http://example.com/#a\
nything-here
            }
        });
        // Initiate the router
        var app_router = new AppRouter;

        app_router.on('route:defaultRoute', function(actions) {
            alert(actions);
        })

        // Start Backbone history a necessary step for bookmarkable URL's
        Backbone.history.start();

    </script>
```

Activate route

Activate another route

Notice the change in the url

Dynamic Routing

Most conventional frameworks allow you to define routes that contain a mix of static and dynamic route parameters. For example you might want to retrieve a post with a variable id with a friendly URL string. Such that your URL would look like "http://example.com/#/posts/12". Once this route was activated you would want to access the id given in the URL string. This example is implemented below.

[21] http://backbonetutorials.com/what-is-a-view

```
    <script>
        var AppRouter = Backbone.Router.extend({
            routes: {
                "posts/:id": "getPost",
                "*actions": "defaultRoute" // Backbone will try match the r\
oute above first
            }
        });
        // Instantiate the router
        var app_router = new AppRouter;
        app_router.on('route:getPost', function (id) {
            // Note the variable in the route definition being passed in he\
re
            alert( "Get post number " + id );
        });
        app_router.on('route:defaultRoute', function (actions) {
            alert( actions );
        });
        // Start Backbone history a necessary step for bookmarkable URL's
        Backbone.history.start();

    </script>
```

Post 120

Post 130

Notice the change in the url

Dynamic Routing Cont. ":params" and "*splats"

Backbone uses two styles of variables when implementing routes. First there are ":params" which match any URL components between slashes. Then there are "*splats" which match any number of URL components. Note that due to the nature of a* "splat" it will always be the last variable in your URL as it will match any and all components.

Any "*splats" or ":params" in route definitions are passed as arguments (in respective order) to the associated function. A route defined as "/:route/:action" will pass 2 variables ("route" and "action") to the callback function. (If this is confusing please post a comment and I will try articulate it better)

Here are some examples of using ":params" and "*splats"

```
    routes: {

        "posts/:id": "getPost",
        // <a href="http://example.com/#/posts/121">Example</a>

        "download/*path": "downloadFile",
        // <a href="http://example.com/#/download/user/images/hey.gif">Down\
load</a>

        ":route/:action": "loadView",
        // <a href="http://example.com/#/dashboard/graph">Load Route/Action\
 View</a>

    },

    app_router.on('route:getPost', function( id ){
        alert(id); // 121
    });
    app_router.on('route:downloadFile', function( path ){
        alert(path); // user/images/hey.gif
    });
    app_router.on('route:loadView', function( route, action ){
        alert(route + "_" + action); // dashboard_graph
    });
```

Routes are quite powerful and in an ideal world your application should never contain too many. If you need to implement hash tags with SEO in mind, do a google search for "google seo hashbangs". Also check out Seo Server[22]

Remember to do a pull request for any errors you come across.

Relevant Links

- Backbone.js official router documentation[23]
- Using routes and understanding the hash tag[24]

Contributors

- Herman Schistad[25] - (Backbone 0.5 rename from Controller to Router)
- Paul Irish[26]

[22] http://seo.apiengine.io
[23] http://documentcloud.github.com/backbone/#Router
[24] http://thomasdavis.github.com/2011/02/07/making-a-restful-ajax-app.html
[25] http://schistad.info
[26] http://paulirish.com

Organizing your application using Modules (require.js)

Unfortunately Backbone.js does not tell you how to organize your code, leaving many developers in the dark regarding how to load scripts and lay out their development environments.

This was quite a different decision to other JavaScript MVC frameworks who were more in favor of setting a development philosophy.

Hopefully this tutorial will allow you to build a much more robust project with great separation of concerns between design and code.

This tutorial will get you started on combining Backbone.js with AMD[27] (Asynchronous Module Definitions).

What is AMD?

Asynchronous Module Definitions[28] designed to load modular code asynchronously in the browser and server. It is actually a fork of the Common.js specification. Many script loaders have built their implementations around AMD, seeing it as the future of modular JavaScript development.

This tutorial will use Require.js[29] to implement a modular and organized Backbone.js.

I highly recommend using AMD for application development

Quick Overview

- Modular
- Scalable
- Compiles well(see r.js[30])
- Market Adoption(Dojo 1.6 converted fully to AMD[31])

Why Require.js?

a. Require.js has a great community and it is growing rapidly. James Burke[32] the author is married to Require.js and always responds to user feedback. He is a leading expert in script loading and a contributer to the AMD specification.

Getting started

To easily understand this tutorial you should jump straight into the example code base.

Example Codebase[33]

[27] https://github.com/amdjs/amdjs-api/wiki/AMD
[28] https://github.com/amdjs/amdjs-api/wiki/AMD
[29] http://requirejs.org
[30] http://requirejs.org/docs/optimization.html
[31] http://dojotoolkit.org/reference-guide/releasenotes/1.6.html
[32] http://tagneto.blogspot.com/
[33] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/modular-backbone

Example Demo[34]

The tutorial is only loosely coupled with the example and you will find the example to be more comprehensive.

If you would like to see how a particular use case would be implemented please visit the GitHub page and create an issue.(Example Request: How to do nested views).

The example isn't super fleshed out but should give you a vague idea.

Example File Structure

There are many different ways to lay out your files and I believe it is actually dependent on the size and type of the project. In the example below views and templates are mirrored in file structure. Collections and Models are categorized into folders kind of like an ORM.

```
/* File Structure
├── imgs
├── css
│   └── style.css
├── templates
│   ├── projects
│   │   ├── list.html
│   │   └── edit.html
│   └── users
│       ├── list.html
│       └── edit.html
├── js
│   ├── libs
│   │   ├── jquery
│   │   │   ├── jquery.min.js
│   │   ├── backbone
│   │   │   ├── backbone.min.js
│   │   └── underscore
│   │   │   ├── underscore.min.js
│   ├── models
│   │   ├── users.js
│   │   └── projects.js
│   ├── collections
│   │   ├── users.js
│   │   └── projects.js
│   ├── views
│   │   ├── projects
│   │   │   ├── list.js
│   │   │   └── edit.js
│   │   └── users
│   │       ├── list.js
│   │       └── edit.js
```

[34] http://backbonetutorials.com/examples/modular-backbone

```
33      |   ├── router.js
34      |   ├── app.js
35      |   ├── main.js  // Bootstrap
36      |   ├── order.js //Require.js plugin
37      |   └── text.js  //Require.js plugin
38      └── index.html
```

*/

To continue you must really understand what we are aiming towards as described in the introduction.

Bootstrapping your application

Using Require.js we define a single entry point on our index page. We should setup any useful containers that might be used by our Backbone views.

Note: The data-main attribute on our single script tag tells Require.js to load the script located at "js/main.js". It automatically appends the ".js"

```
1       <!doctype html>
2       <html lang="en">
3       <head>
4         <title>Jackie Chan</title>
5         <!-- Load the script "js/main.js" as our entry point -->
6         <script data-main="js/main" src="js/libs/require/require.js"></script\
7   >
8       </head>
9       <body>
10
11      <div id="container">
12        <div id="menu"></div>
13        <div id="content"></div>
14      </div>
15
16      </body>
17      </html>
```

You should most always end up with quite a light weight index file. You can serve this off your server and then the rest of your site off a CDN ensuring that everything that can be cached, will be. (You can also now serve the index file off the CDN using Cloudfront)

What does the bootstrap look like?

Our bootstrap file will be responsible for configuring Require.js and loading initially important dependencies.

In the example below we configure Require.js to create a shortcut alias to commonly used scripts such as jQuery, Underscore and Backbone.

Unfortunately Backbone.js isn't AMD enabled so I downloaded the community managed repository and patched it on amdjs[35].

Hopefully if the AMD specification takes off these libraries will add code to allow themselves to be loaded asynchronously. Due to this inconvenience the bootstrap is not as intuitive as it could be.

We also request a module called "app", this will contain the entirety of our application logic.

Note: Modules are loaded relatively to the boot strap and always append with ".js". So the module "app" will load "app.js" which is in the same directory as the bootstrap.

```
1   // Filename: main.js
2
3   // Require.js allows us to configure shortcut alias
4   // There usage will become more apparent further along in the tutorial.
5   require.config({
6     paths: {
7       jquery: 'libs/jquery/jquery',
8       underscore: 'libs/underscore/underscore',
9       backbone: 'libs/backbone/backbone'
10    }
11
12  });
13
14  require([
15
16    // Load our app module and pass it to our definition function
17    'app',
18  ], function(App){
19    // The "app" dependency is passed in as "App"
20    App.initialize();
21  });
```

How should we lay out external scripts?

Any modules we develop for our application using AMD/Require.js will be asynchronously loaded.

We have a heavy dependency on jQuery, Underscore and Backbone, unfortunately this libraries are loaded synchronously and also depend on each other existing in the global namespace.

A boiler plate module

So before we start developing our application, let's quickly look over boiler plate code that will be reused quite often.

For convenience sake I generally keep a "boilerplate.js" in my application root so I can copy it when I need to.

[35] https://github.com/amdjs

```
//Filename: boilerplate.js

define([
  // These are path alias that we configured in our bootstrap
  'jquery',     // lib/jquery/jquery
  'underscore', // lib/underscore/underscore
  'backbone'    // lib/backbone/backbone
], function($, _, Backbone){
  // Above we have passed in jQuery, Underscore and Backbone
  // They will not be accessible in the global scope
  return {};
  // What we return here will be used by other modules
});
```

The first argument of the define function is our dependency array, in the future we can pass in any modules we like.

App.js Building our applications main module

Our applications main module should always remain light weight. This tutorial only covers setting up a Backbone Router and initializing it in our main module.

The router will then load the correct dependencies depending on the current URL.

```
// Filename: app.js
define([
  'jquery',
  'underscore',
  'backbone',
  'router', // Request router.js
], function($, _, Backbone, Router){
  var initialize = function(){
    // Pass in our Router module and call it's initialize function
    Router.initialize();
  }

  return {
    initialize: initialize
  };
});

// Filename: router.js
define([
  'jquery',
  'underscore',
  'backbone',
  'views/projects/list',
  'views/users/list'
```

```
25      ], function($, _, Backbone, Session, ProjectListView, UserListView){
26        var AppRouter = Backbone.Router.extend({
27          routes: {
28            // Define some URL routes
29            '/projects': 'showProjects',
30            '/users': 'showUsers',
31
32            // Default
33            '*actions": 'defaultAction'
34          }
35        });
36
37        var initialize = function(){
38          var app_router = new AppRouter;
39          app_router.on('showProjects', function(){
40            // Call render on the module we loaded in via the dependency arra\
41  y
42            // 'views/projects/list'
43            var projectListView = new ProjectListView();
44            projectListView.render();
45          });
46            // As above, call render on our loaded module
47            // 'views/users/list'
48          app_router.on('showUsers', function(){
49            var userListView = new UserListView();
50            userListView.render();
51          });
52          app_router.on('defaultAction', function(actions){
53            // We have no matching route, lets just log what the URL was
54            console.log('No route:', actions);
55          });
56          Backbone.history.start();
57        };
58        return {
59          initialize: initialize
60        };
61      });
```

Modularizing a Backbone View

Backbone views usually interact with the DOM. Using our new modular system we can load in JavaScript templates using the Require.js text! plug-in.

```
// Filename: views/project/list
define([
  'jquery',
  'underscore',
  'backbone',
  // Using the Require.js text! plugin, we are loaded raw text
  // which will be used as our views primary template
  'text!templates/project/list.html'
], function($, _, Backbone, projectListTemplate){
  var ProjectListView = Backbone.View.extend({
    el: $('#container'),
    render: function(){
      // Using Underscore we can compile our template with data
      var data = {};
      var compiledTemplate = _.template( projectListTemplate, data );
      // Append our compiled template to this Views "el"
      this.$el.append( compiledTemplate );
    }
  });
  // Our module now returns our view
  return ProjectListView;
});
```

JavaScript templating allows us to separate the design from the application logic by placing all our HTML in the templates folder.

Modularizing a Collection, Model and View

Now we put it altogether by chaining up a Model, Collection and View which is a typical scenario when building a Backbone.js application.

First we will define our model

```
// Filename: models/project
define([
  'underscore',
  'backbone'
], function(_, Backbone){
  var ProjectModel = Backbone.Model.extend({
    defaults: {
      name: "Harry Potter"
    }
  });
  // Return the model for the module
  return ProjectModel;
});
```

Now that we have a model, our collection module can depend on it. We will set the “model” attribute of our collection to the loaded module. Backbone.js offers great benefits when doing this.

> Collection.model: Override this property to specify the model class that the collection contains. If defined, you can pass raw attributes objects (and arrays) to add, create, and reset, and the attributes will be converted into a model of the proper type.

```
// Filename: collections/projects
define([
  'underscore',
  'backbone',
  // Pull in the Model module from above
  'models/project'
], function(_, Backbone, ProjectModel){
  var ProjectCollection = Backbone.Collection.extend({
    model: ProjectModel
  });
  // You don't usually return a collection instantiated
  return ProjectCollection;
});
```

Now we can simply depend on our collection in our view and pass it to our JavaScript template.

// Filename: views/projects/list define(['jquery', 'underscore', 'backbone', // Pull in the Collection module from above 'collections/projects', 'text!templates/projects/list], function($, _, Backbone, ProjectsCollection, projectsListTemplate){ var ProjectListView = Backbone.View.extend({ el: $("#container"), initialize: function(){ this.collection = new ProjectsCollection(); this.collection.add({ name: "Ginger Kid"}); // Compile the template using Underscores micro-templating var compiledTemplate = _.template(projectsListTemplate, { projects: this.collection.models }); this.$el.html(compiledTemplate); } }); // Returning instantiated views can be quite useful for having "state" return ProjectListView; });

Conclusion

Looking forward to feedback so I can turn this post and example into quality references on building modular JavaScript applications.

Get in touch with me on twitter, comments or GitHub!

Relevant Links

- Organizing Your Backbone.js Application With Modules[36]

Contributors

- Jakub Kozisek[37] (created modular-backbone-updated containing updated libs with AMD support)

[36] http://weblog.bocoup.com/organizing-your-backbone-js-application-with-modules
[37] https://github.com/dzejkej

Lightweight Infinite Scrolling using Twitter API

Getting started

In this example we are going to build a widget that pulls in tweets and when the user scrolls to the bottom of the widget Backbone.js will re-sync with the server to bring down the next page of results.

Example Demo[38]

Example Source[39]

Note: This tutorial will use AMD[40] for modularity.

The Twitter Collection

Twitter offers a jsonp API for browsing tweets. The first thing to note is that we have to append '&callback?' to allow cross domain Ajax calls which is a feature of jsonp[41].

Using the 'q' and 'page' query parameters we can find the results we are after. In the collection definition below we have set some defaults which can be overridden at any point.

Twitter's search API actually returns a whole bunch of meta information alongside the results. Though this is a problem for Backbone.js because a Collection expects to be populated with an array of objects. So in our collection definition we can override the Backbone.js default parse function to instead choose the correct property to populate the collection.

```
1     // collections/twitter.js
2     define([
3       'jquery',
4       'underscore',
5       'backbone'
6     ], function($, _, Backbone){
7       var Tweets = Backbone.Collection.extend({
8         url: function () {
9           return 'http://search.twitter.com/search.json?q=' + this.query + \
10 '&page=' + this.page + '&callback=?'
11        },
12        // Because twitter doesn't return an array of models by default we \
13 need
14        // to point Backbone.js at the correct property
15        parse: function(resp, xhr) {
16          return resp.results;
17        },
18        page: 1,
```

[38] http://backbonetutorials.com/examples/infinite-scroll/

[39] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/infinite-scroll

[40] http://backbonetutorials.com/organizing-backbone-using-modules

[41] http://en.wikipedia.org/wiki/JSONP

```
        query: 'backbone.js tutorials'
      });

      return Tweets;
    });
```

Note: Feel free to attach the meta information returned by Twitter to the collection itself e.g.

```
    parse: function(resp, xhr) {
      this.completed_in = resp.completed_in
      return resp.results;
    },
```

Setting up the View

The first thing to do is to load our Twitter collection and template into the widget module. We should attach our collection to our view in our `initialize` function. `loadResults` will be responsible for calling fetch on our Twitter collection. On success we will append the latest results to our widget using our template. Our Backbone.js `events` will listen for `scroll` on the current `el` of the view which is '.twitter-widget'. If the current `scrollTop` is at the bottom then we simply increment the Twitter collections current page property and call `loadResults` again.

```
    // views/twitter/widget.js
    define([
      'jquery',
      'underscore',
      'backbone',
      'vm',
      'collections/twitter',
      'text!templates/twitter/list.html'
    ], function($, _, Backbone, Vm, TwitterCollection, TwitterListTemplate)\
{
      var TwitterWidget = Backbone.View.extend({
        el: '.twitter-widget',
        initialize: function () {
          // isLoading is a useful flag to make sure we don't send off more\
 than
          // one request at a time
          this.isLoading = false;
          this.twitterCollection = new TwitterCollection();
        },
        render: function () {
          this.loadResults();
        },
        loadResults: function () {
          var that = this;
          // we are starting a new load of results so set isLoading to true
          this.isLoading = true;
```

```
27          // fetch is Backbone.js native function for calling and parsing t\
28 he collection url
29          this.twitterCollection.fetch({
30            success: function (tweets) {
31              // Once the results are returned lets populate our template
32              $(that.el).append(_.template(TwitterListTemplate, {tweets: tw\
33 eets.models, _:_}));
34              // Now we have finished loading set isLoading back to false
35              that.isLoading = false;
36            }
37          });
38        },
39        // This will simply listen for scroll events on the current el
40        events: {
41          'scroll': 'checkScroll'
42        },
43        checkScroll: function () {
44          var triggerPoint = 100; // 100px from the bottom
45            if( !this.isLoading && this.el.scrollTop + this.el.clientHeight\
46  + triggerPoint > this.el.scrollHeight ) {
47              this.twitterCollection.page += 1; // Load next page
48              this.loadResults();
49            }
50        }
51      });
52      return TwitterWidget;
53    });
```

Note: `triggerPoint` *will allow you to set an offset where the user has to scroll to before loading the next page*

The widget template

Our view above passes into our underscore template the variable tweets which we can simply iterate over with using underscore's `each` method.

```
<!-- templates/twitter/list.html -->
<ul class="tweets">
<% _.each(tweets, function (tweet) { %>

  <li><%= tweet.get('text') %></li>

<% }); %>
</ul>
```

Conclusion

This is a very lightweight but robust infinite scroll example. There are caveats to using infinite scroll in UI/UX so make sure to only use it when applicable.

Example Demo[42]

Example Source[43]

[42] http://backbonetutorials.com/examples/infinite-scroll/

[43] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/infinite-scroll

Simple example - Node.js, Restify, MongoDb and Mongoose

Before I start, the Backbone.js parts of this tutorial will be using techniques described in "Organizing your application using Modules[44] to construct a simple guestbook.

Getting started

To easily understand this tutorial you should jump straight into the example code base.

Example Codebase[45]

Example Demo[46]

This tutorial will assist you in saving data(Backbone.js Models) to MongoDb and retrieving a list(Backbone.js Collections) of them back.

The technologies

This stack is great for rapid prototyping and highly intuitive. Personal note: I love using JavaScript as my only language for the entire application (FrontEnd/BackEnd/API/Database). Restify is still in early development but is essentially just an extension of Express. So for anyone needing more stability you can easily just substitute Express in.

Node.js

"Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

Restify

"Restify is a node.js module built specifically to enable you to build correct REST web services. It borrows heavily from express (intentionally) as that is more or less the de facto API for writing web applications on top of node.js."

MongoDb

"MongoDB (from "humongous") is a scalable, high-performance, open source NoSQL database."

Mongoose

"Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment."

[44] http://backbonetutorials.com/organizing-backbone-using-modules/
[45] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/nodejs-mongodb-mongoose-restify
[46] http://backbonetutorials.com/examples/nodejs-mongodb-mongoose-restify/

Building the server

In the example repository there is a server.js example which can be executed by running `node server.js`. If you use this example in your own applications make sure to update the Backbone.js Model[47] and Collection[48] definitions to match your server address.

Restify configuration

The first thing to do is require the Restify module. Restify will be in control of handling our restful endpoints and returning the appropriate JSON.

```
1    var restify = require('restify');
2    var server = restify.createServer();
3    server.use(restify.bodyParser());
```

Note: bodyParser() takes care of turning your request data into a JavaScript object on the server automatically.

MongoDb/Mongoose configuration

We simply want to require the MongoDb module and pass it a MongoDb authentication URI e.g. mongodb://username:server@mongoserver:10059/somecollection

The code below presupposes you have another file in the same directory called `config.js`. Your config should never be public as it contains your credentials. So for this repository I have added `config.js` to my `.gitignore` but added in a sample config[49].

```
1    var mongoose = require('mongoose/');
2    var config = require('./config');
3    db = mongoose.connect(config.creds.mongoose_auth),
4    Schema = mongoose.Schema;
```

Mongoose Schema

Mongoose introduces a concept of model/schema[50] enforcing types which allow for easier input validation etc

[47] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/models/message.js

[48] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/collections/messages.js

[49] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/config-sample.js

[50] http://mongoosejs.com/docs/model-definition.html

```
    // Create a schema for our data
    var MessageSchema = new Schema({
      message: String,
      date: Date
    });
    // Use the schema to register a model with MongoDb
    mongoose.model('Message', MessageSchema);
    var Message = mongoose.model('Message');
```

Note: `Message` *can now be used for all things CRUD related.*

Setting up the routes

Just like in Backbone, Restify allows you to configure different routes and their associated callbacks. In the code below we define two routes. One for saving new messages and one for retrieving all messages. After we have created our function definitions, we attach them to either GET/POST/PUT/DELETE on a particular restful endpoint e.g. GET /messages

```
    // This function is responsible for returning all entries for the Messa\
ge model
    function getMessages(req, res, next) {
      // Resitify currently has a bug which doesn't allow you to set defaul\
t headers
      // This headers comply with CORS and allow us to server our response \
to any origin
      res.header("Access-Control-Allow-Origin", "*");
      res.header("Access-Control-Allow-Headers", "X-Requested-With");
      // .find() without any arguments, will return all results
      // the `-1` in .sort() means descending order
      Message.find().sort('date', -1).execFind(function (arr,data) {
        res.send(data);
      });
    }



    function postMessage(req, res, next) {
      res.header("Access-Control-Allow-Origin", "*");
      res.header("Access-Control-Allow-Headers", "X-Requested-With");
      // Create a new message model, fill it up and save it to Mongodb
      var message = new Message();
      message.message = req.params.message;
      message.date = new Date();
      message.save(function () {
        res.send(req.body);
      });
    }

```

```
31      // Set up our routes and start the server
32      server.get('/messages', getMessages);
33      server.post('/messages', postMessage);
```

This wraps up the server side of things, if you follow the example[51] then you should see something like

http://backbonetutorials.nodejitsu.com/messages[52]

Note: Again you must remember to change the Model[53] and Collection[54] definitions to match your server address.

Setting up the client (Backbone.js)

I've actually used the latest copy of http://backboneboilerplate.com[55] to set up the example page.

The important files you will want to check out are;

- views/dashboard/page.js
- views/guestbook/form.js
- views/guestbook/list.js
- models/message.js
- collections/messages.js
- templates/guestbook/

Saving a message

First of all we want to setup a template[56] for showing our form that creates new messages.

```
1       <textarea class="message"></textarea>
2       <button class="post-message">Post Message</button>
```

This template gets inserted into the DOM by `views/guestbook/form.js`, this Backbone view also handles the interaction of the form and the posting of the new data.

Let us create a Backbone Model that has the correct URL for our restful interface.

[51] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/server.js
[52] http://backbonetutorials.nodejitsu.com/messages
[53] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/models/message.js
[54] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/collections/messages.js
[55] http://backboneboilerplate.com
[56] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/templates/guestbook/form.html

```
define([
  'underscore',
  'backbone'
], function(_, Backbone) {
  var Message = Backbone.Model.extend({
      url: 'http://localhost:8080/messages'
  });
  return Message;
});
```

We can see how we require our predefined model for messages and also our form template.

```
define([
  'jquery',
  'underscore',
  'backbone',
  'models/message',
  'text!templates/guestbook/form.html'
], function($, _, Backbone, MessageModel, guestbookFormTemplate){
  var GuestbookForm = Backbone.View.extend({
    el: '.guestbook-form-container',
    render: function () {
      $(this.el).html(guestbookFormTemplate);

    },
    events: {
      'click .post-message': 'postMessage'
    },
    postMessage: function() {
      var that = this;

      var message = new MessageModel();
      message.save({ message: $('.message').val()}, {
        success: function () {
          that.trigger('postMessage');
        }
      });
    }
  });
  return GuestbookForm;
});
```

Note: `trigger` *is from Backbone Events, I binded a listener to this view in* `views/dashboard/page.js` *so when a new message is submitted, the list is re-rendered. We are setting the date of the POST on the server so there is no need to pass it up.*

Retrieving a list of messages

We setup a route on our server to generate a list of all available messages at `GET /messages`. So we need to define a collection with the appropriate `url` to fetch this data down.

```
    define([
      'jquery',
      'underscore',
      'backbone',
      'models/message'
    ], function($, _, Backbone, MessageModel){
      var Messages = Backbone.Collection.extend({
        model: MessageModel, // Generally best practise to bring down a Mod\
el/Schema for your collection
        url: 'http://localhost:8080/messages'
      });

      return Messages;
    });
```

Now that we have a collection to use we can setup our `views/list.js` to require the collection and trigger a fetch. Once the fetch is complete we want to render our returned data to a template and insert it into the DOM.

```
    define([
      'jquery',
      'underscore',
      'backbone',
      'collections/messages',
      'text!templates/guestbook/list.html'
    ], function($, _, Backbone, MessagesCollection, guestbookListTemplate){
      var GuestbookList = Backbone.View.extend({
        el: '.guestbook-list-container',
        render: function () {
          var that = this;
          var messages = new MessagesCollection();
          messages.fetch({
            success: function(messages) {
              $(that.el).html(_.template(guestbookListTemplate, {messages: \
messages.models, _:_}));
            }
          });
        }
      });
      return GuestbookList;
    });
```

The template file should iterate over `messages.models` which is an array and print out a HTML fragment for each model.

```
<% _.each(messages, function(message) { %>

<p><%= message.get('message') %></p>
<em><%= message.get('date') %></em>

<% }); %>
```

This actually sums up everything you need to know to implement this simple example.

Conclusion

Example Codebase[57]

Example Demo[58]

In this example you should really be using relative URL's in your collections/models and instead setting a baseUrl in a config file or by placing your index.html file on the restful server.

This example is hosted on GitHub therefore we had to include the absolute URL to the server which is hosted on nodejitsu.com

On a personal note, I have of recent used the Joyent, Nodejitsu, MongoDbHq stack after they have now partnered up and I have nothing but good things to say. Highly recommend you check it out!

As always I hope I made this tutorial easy to follow!

Get in touch with me on twitter, comments or GitHub!

Relevant Links

Organizing Your Backbone.js Application With Modules[59]

[57] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/nodejs-mongodb-mongoose-restify
[58] http://backbonetutorials.com/examples/nodejs-mongodb-mongoose-restify/
[59] http://weblog.bocoup.com/organizing-your-backbone-js-application-with-modules

Cross-domain Backbone.js with sessions using CORS

** This tutorial is a proof of concept and needs to be checked for security flaws **

This tutorial will teach you how to completely separate the server and client allowing for developers to work with freedom in their respective areas.

On a personal note, I consider this development practice highly desirable and encourage others to think of the possible benefits but the security still needs to be proved.

> Cross-Origin Resource Sharing (CORS) is a specification that enables a truly open access across domain-boundaries. - enable-cors.org[60]

Some benefits include

- The client and back end exist independently regardless of where they are each hosted and built.
- Due to the separation of concerns, testing now becomes easier and more controlled.
- Develop only one API on the server, your front-end could be outsourced or built by a in-house team.
- As a front-end developer you can host the client anywhere.
- This separation enforces that the API be built robustly, documented, collaboratively and versioned.

** Cons of this tutorial **

- This tutorial doesn't explain how to perform this with cross browser support. CORS headers aren't supported by Opera and IE 6/7. Though it is do-able using easyXDM[61]
- Security is somewhat addressed but maybe a more thorough security expert can chime in.

Security

- Don't allow GET request to change data, only retrieve.
- Whitelist your allowed domains (see server.js[62])
- Protect again JSON padding[63]

[60] http://enable-cors.org/
[61] http://easyxdm.net/wp/
[62] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/cross-domain/server.js
[63] http://blog.opensecurityresearch.com/2012/02/json-csrf-with-parameter-padding.html

Getting started

To easily understand this tutorial you should jump straight into the example code base.

Host the codebase on a simple HTTP server such that the domain is `localhost` with port 80 hidden.

Example Codebase[64]

Example Demo[65]

This tutorial focuses on building a flexible Session model to control session state in your application.

Checking session state at first load

Before starting any routes, we should really know whether the user is authenticated. This will allow us to load the appropriate views. We will simply wrap our `Backbone.history.start` in a callback that executes after `Session.getAuth` has checked the server. We will jump into our Session model next.

```
    define([
      'jquery',
      'underscore',
      'backbone',
      'vm',
      'events',
      'models/session',
      'text!templates/layout.html'
    ], function($, _, Backbone, Vm, Events, Session, layoutTemplate){
      var AppView = Backbone.View.extend({
        el: '.container',
        initialize: function () {
            $.ajaxPrefilter( function( options, originalOptions, jqXHR ) {
            // Your server goes below
            //options.url = 'http://localhost:8000' + options.url;
            options.url = 'http://cross-domain.nodejitsu.com' + options.url\
;
          });

        },
        render: function () {
          var that = this;
          $(this.el).html(layoutTemplate);
          // This is the entry point to your app, therefore
          // when the user refreshes the page we should
          // really know if they're authed. We will give it
          // A call back when we know what the auth status is
          Session.getAuth(function () {
            Backbone.history.start();
          })
```

[64] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/cross-domain
[65] http://backbonetutorials.com/examples/cross-domain/

```
31          }
32        });
33        return AppView;
34      });
```

Note: We have used jQuery `ajaxPrefilter` *to hook into all AJAX requests before they are executed. This is where we specify what server we want the application to hit.*

An example Session model

This is a very light weight Session model which handles most situations. Read through the code and comments below. The model simply has a login, logout and check function. Again we have hooked into jQuery `ajaxPrefilter` to allow for csrf tokens and also telling jQuery to send cookies with the `withCredentials` property. The model relies heavily on it's `auth` property. Throughout your application, each view can simply bind to `change:auth` on the Session model and react accordingly. Because we return this AMD module instantiated using the new keyword, then it will keep state throughout the page. (This may not be best practice but it's highly convenient)

```
1       // views/app.js
2       define([
3         'underscore',
4         'backbone'
5       ], function(_, Backbone) {
6         var SessionModel = Backbone.Model.extend({
7
8           urlRoot: '/session',
9           initialize: function () {
10            var that = this;
11            // Hook into jquery
12            // Use withCredentials to send the server cookies
13            // The server must allow this through response headers
14            $.ajaxPrefilter( function( options, originalOptions, jqXHR ) {
15              options.xhrFields = {
16                withCredentials: true
17              };
18              // If we have a csrf token send it through with the next reques\
19  t
20              if(typeof that.get('_csrf') !== 'undefined') {
21                jqXHR.setRequestHeader('X-CSRF-Token', that.get('_csrf'));
22              }
23            });
24          },
25          login: function(creds) {
26            // Do a POST to /session and send the serialized form creds
27            this.save(creds, {
28               success: function () {}
29            });
30          },
```

```
31          logout: function() {
32            // Do a DELETE to /session and clear the clientside data
33            var that = this;
34            this.destroy({
35              success: function (model, resp) {
36                model.clear()
37                model.id = null;
38                // Set auth to false to trigger a change:auth event
39                // The server also returns a new csrf token so that
40                // the user can relogin without refreshing the page
41                that.set({auth: false, _csrf: resp._csrf});
42
43              }
44            });
45          },
46          getAuth: function(callback) {
47            // getAuth is wrapped around our router
48            // before we start any routers let us see if the user is valid
49            this.fetch({
50                success: callback
51            });
52          }
53        });
54        return new SessionModel();
55
56      });
```

Note: This session model is missing one useful feature. If a user looses auth when navigating your application then the application should set {auth: false} on this model. To do this, in the `ajaxPrefilter` *edit outgoing* `success` *functions to check if the server response was {auth: false} and then call the original* `success()` *function.*

Hooking up views to listen to changes in `auth`

Now that we have a Session model, let's hook up our `login/logout` view to listen to changes in `auth`. When creating the view we use `on` to bind a listener to the `auth` attribute of our model. Everytime it changes we will re-render the view which will conditionally load a template depending on the value of `Session.get('auth')`.

```
    // models/session.js
    define([
      'jquery',
      'underscore',
      'backbone',
      'models/session',
      'text!templates/example/login.html',
      'text!templates/example/logout.html'
    ], function($, _, Backbone, Session, exampleLoginTemplate, exampleLogou\
tTemplate){
      var ExamplePage = Backbone.View.extend({
        el: '.page',
        initialize: function () {
          var that = this;
          // Bind to the Session auth attribute so we
          // make our view act recordingly when auth changes
          Session.on('change:auth', function (session) {
              that.render();
          });
        },
        render: function () {
          // Simply choose which template to choose depending on
          // our Session models auth attribute
          if(Session.get('auth')){
            this.$el.html(_.template(exampleLogoutTemplate, {username: Sess\
ion.get('username')}));
          } else {
            this.$el.html(exampleLoginTemplate);
          }
        },
        events: {
          'submit form.login': 'login', // On form submission
          'click .logout': 'logout'
        },
        login: function (ev) {
          // Disable the button
          $('[type=submit]', ev.currentTarget).val('Logging in').attr('disa\
bled', 'disabled');
          // Serialize the form into an object using a jQuery plgin
          var creds = $(ev.currentTarget).serializeObject();
          Session.login(creds);
          return false;
        },
        logout: function (ev) {
          // Disable the button
          $(ev.currentTarget).text('Logging out').attr('disabled', 'disable\
d');

          Session.logout();
        }
```

```
50        });
51        return ExamplePage;
52      });
```

Note: `.serializeObject` *is not a native jQuery function and I have included it as app.js*[66] *in the demo folder.* `creds` *can be an object of any variation of inputs, regardless it will be converted to JSON and posted to the server like any normal Backbone model.*

Here are the templates we are using for our login view

```
1     <!-- templates/example/login.html -->
2     <form class="login">
3         <label for="">Username</label>
4         <input name="username" type="text" required autofocus>
5         <input type="submit" id="submit" value="Login">
6     </form>
7
8     <!-- templates/example/logout.html -->
9     <p>Hello, <%= username %>. Time to logout?</p>
10    <button class="logout">Logout</button>
```

This wraps up setting up the client, there are some notable points to make sure this technique works.

- You must use `withCredentials` supplied by jQuery - session.js
- You must send your request with csrf tokens for security - session.js
- You should wrap your applications entry pointer (router in this example) in a check auth function - app.js
- You must point your application at the right server - app.js

Building a compatible server

This tutorial uses node.js, express.js and a modified csrf.js library. An example server.js file exist in the `examples/cross-domain` folder. When inside the folder simply type `npm install -d` to install the dependencies and then `node server.js` to start the server. Again, make sure your `app.js` points at the correct server.

The server has to do a few things;

- Allow CORS request
- Implement csrf protection
- Allow jQuery to send credentials
- Set a whitelist of allowed domains

[66] https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/cross-domain/js/views/app.js

- Configure the correct response headers

To save you sometime here are some gotchas;

- When sending `withCredentials` you must set correct response header `Access-Control-Allow-Credentials: true`. Also as a security policy browsers do not allow `Access-Control-Allow-Origin` to be set to `*`. So the origin of the request has to be known and trusted, so in the example below we use an of white listed domains.
- jQuery ajax will trigger the browser to send these headers to enforce security `origin, x-requested-with, accept` so our server must allow them.
- The browser might send out a `pre-flight` request to verify that it can talk to the server. The server must return `200 OK` on these `pre-flight` request.

Be sure to read this Mozilla documentation[67] on the above.

Example node server

This server below implements everything we have talked about so far. It should be relatively easy to see how would translate into other frameworks and languages. `app.configure` runs the specified libraries against every request. We have told the server that on each request it should check the csrf token and check if the origin domain is white-listed. If so we edit each request to contain the appropriate headers.

This server has 3 endpoints, that are pseudo-restful;

- POST /session - Login - Sets the session username and returns a csrf token for the user to use
- DELETE /session - Logout - Destroys the session and regenerates a new csrf token if the user wants to re-login
- GET /session - Checks Auth - Simply returns if auth is true or false, if true then also returns some session details

```
1   var express = require('express');
2
3   var connect = require('connect');
4   // Custom csrf library
5   var csrf = require('./csrf');
6
7   var app = express.createServer();
8
9   var allowCrossDomain = function(req, res, next) {
10    // Added other domains you want the server to give access to
11    // WARNING - Be careful with what origins you give access to
12    var allowedHost = [
13      'http://backbonetutorials.com',
14      'http://localhost'
```

[67] http://hacks.mozilla.org/2009/07/cross-site-xmlhttprequest-with-cors/

```
15      ];
16
17      if(allowedHost.indexOf(req.headers.origin) !== -1) {
18        res.header('Access-Control-Allow-Credentials', true);
19        res.header('Access-Control-Allow-Origin', req.headers.origin)
20        res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE,OPTIO\
21  NS');
22        res.header('Access-Control-Allow-Headers', 'X-CSRF-Token, X-Requested\
23  -With, Accept, Accept-Version, Content-Length, Content-MD5, Content-Type, D\
24  ate, X-Api-Version');
25        next();
26      } else {
27        res.send({auth: false});
28      }
29    }
30
31    app.configure(function() {
32        app.use(express.cookieParser());
33        app.use(express.session({ secret: 'thomasdavislovessalmon' }));
34        app.use(express.bodyParser());
35        app.use(allowCrossDomain);
36        app.use(csrf.check);
37    });
38
39    app.get('/session', function(req, res){
40      // This checks the current users auth
41      // It runs before Backbones router is started
42      // we should return a csrf token for Backbone to use
43      if(typeof req.session.username !== 'undefined'){
44        res.send({auth: true, id: req.session.id, username: req.session.usern\
45  ame, _csrf: req.session._csrf});
46      } else {
47        res.send({auth: false, _csrf: req.session._csrf});
48      }
49    });
50
51    app.post('/session', function(req, res){
52      // Login
53      // Here you would pull down your user credentials and match them up
54      // to the request
55      req.session.username = req.body.username;
56      res.send({auth: true, id: req.session.id, username: req.session.usernam\
57  e});
58    });
59
60    app.del('/session/:id', function(req, res, next){
61      // Logout by clearing the session
62      req.session.regenerate(function(err){
63        // Generate a new csrf token so the user can login again
```

```
64        // This is pretty hacky, connect.csrf isn't built for rest
65        // I will probably release a restful csrf module
66        csrf.generate(req, res, function () {
67          res.send({auth: false, _csrf: req.session._csrf});
68        });
69      });
70    });
71
72    app.listen(8000);
```

Note: I wrote a custom csrf module for this which can be found in the example directory. It's based of connects and uses the `crypto` *library. I didn't spend much time on it but other traditional csrf modules won't work because they aren't exactly built for this implementation technique.*

Conclusion

This approach really hammers in the need for a well documented and designed API. A powerful API will let you do application iterations with ease.

Again, it would be great for some more analysis of the security model.

Enjoy using Backbone.js cross domain!

I cannot get passed the spam filter on HackerNews so feel free to submit this tutorial

Example Codebase[68]

Example Demo[69]

Relevant Links

- cross-site xmlhttprequest with CORS[70]
- Cross-Origin Resource Sharing[71]
- Using CORS with All (Modern) Browsers[72]

[68] https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/cross-domain
[69] http://backbonetutorials.com/examples/cross-domain/
[70] http://hacks.mozilla.org/2009/07/cross-site-xmlhttprequest-with-cors/
[71] http://www.w3.org/TR/cors/
[72] http://www.kendoui.com/blogs/teamblog/posts/11-10-04/using_cors_with_all_modern_browsers.aspx

SEO for single page apps

This tutorial will show you how to index your application on search engines. As the author I believe that servers should be completely independent of the client in the age of API's. Which speeds up development for the ever increasing array of clients. It is on the shoulders of the search engines to conform and they should not dictate how the web is stored and accessed.

In 2009 Google released the idea of escaped fragments[73].

The idea simply stating that if a search engine should come across your JavaScript application then you have the permission to redirect the search engine to another URL that serves the fully rendered version of the page (The current search engines cannot execute much JavaScript (Some people speculate that Google Chrome was born of Google Search wishing to successfully render every web page to retrieve ajaxed content)).

How does redirecting bots work?

Using modern headless browsers, we can easily return the fully rendered content per request by redirecting bots using our web servers configuration. Here is an image made by Google depicting the setup.

headless seo

Implementation using Phantom.js

Phantom.js[74] is a headless webkit browser. We are going to setup a node.js server that given a URL, it will fully render the page content. Then we will redirect bots to this server to retrieve the correct content.

You will need to install node.js and phantom.js onto a box. Then start up this server below. There are two files, one which is the web server and the other is a phantomjs script that renders the page.

```
    // web.js

    // Express is our web server that can handle request
    var express = require('express');
    var app = express();


    var getContent = function(url, callback) {
      var content = '';
      // Here we spawn a phantom.js process, the first element of the
      // array is our phantomjs script and the second element is our url
      var phantom = require('child_process').spawn('phantomjs', ['phantom-s\
erver.js', url]);
      phantom.stdout.setEncoding('utf8');
```

[73] http://googlewebmastercentral.blogspot.com.au/2009/10/proposal-for-making-ajax-crawlable.html
[74] http://phantomjs.org/

```
    // Our phantom.js script is simply logging the output and
    // we access it here through stdout
    phantom.stdout.on('data', function(data) {
      content += data.toString();
    });
    phantom.on('exit', function(code) {
      if (code !== 0) {
        console.log('We have an error');
      } else {
        // once our phantom.js script exits, let's call out call back
        // which outputs the contents to the page
        callback(content);
      }
    });
  };

  var respond = function (req, res) {
    // Because we use [P] in htaccess we have access to this header
    url = 'http://' + req.headers['x-forwarded-host'] + req.params[0];
    getContent(url, function (content) {
      res.send(content);
    });
  }

  app.get(/(.*)/, respond);
  app.listen(3000);
```

The script below is `phantom-server.js` and will be in charge of fully rendering the content. We don't return the content until the page is fully rendered. We hook into the resources listener to do this.

```
  var page = require('webpage').create();
  var system = require('system');

  var lastReceived = new Date().getTime();
  var requestCount = 0;
  var responseCount = 0;
  var requestIds = [];
  var startTime = new Date().getTime();

  page.onResourceReceived = function (response) {
      if(requestIds.indexOf(response.id) !== -1) {
          lastReceived = new Date().getTime();
          responseCount++;
          requestIds[requestIds.indexOf(response.id)] = null;
      }
  };
  page.onResourceRequested = function (request) {
      if(requestIds.indexOf(request.id) === -1) {
          requestIds.push(request.id);
```

```
            requestCount++;
        }
    };

    // Open the page
    page.open(system.args[1], function () {});

    var checkComplete = function () {
      // We don't allow it to take longer than 5 seconds but
      // don't return until all requests are finished
      if((new Date().getTime() - lastReceived > 300 && requestCount === res\
ponseCount) || new Date().getTime() - startTime > 5000)  {
        clearInterval(checkCompleteInterval);
        console.log(page.content);
        phantom.exit();
      }
    }
    // Let us check to see if the page is finished rendering
    var checkCompleteInterval = setInterval(checkComplete, 1);
```

Once we have this server up and running we just redirect bots to the server in our client's web server configuration.

Redirecting bots

If you are using apache we can edit out `.htaccess` such that Google requests are proxied to our middle man phantom.js server.

```
    RewriteEngine on
    RewriteCond %{QUERY_STRING} ^_escaped_fragment_=(.*)$
    RewriteRule (.*) http://webserver:3000/%1? [P]
```

We could also include other `RewriteCond`, such as `user agent` to redirect other search engines we wish to be indexed on.

Though Google won't use `_escaped_fragment_` unless we tell it to by either including a meta tag; `<meta name="fragment" content="!">` or using `#!` URLs in our links.

You will most likely have to use both.

I have released an open source npm package called seo server[75] for anyone wanting to jump straight in.

And also via request I have setup a service currently in beta, which takes care of the whole process and even generates sitemap.xml's for your JavaScript applications. It is also called Seo Server[76]. (currently in beta)

This has been tested with Google Webmasters fetch tool. Make sure you include `#!` on your URLs when using the fetch tool.

[75] http://seo.apiengine.io

[76] http://seoserver.apiengine.io

Relevant Links

- Open source node.js Seo Server[77]

[77] http://seo.apiengine.io