FREE eBook

LEARNING backbone.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#backbone.j

S

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: backbone-js

It is an unofficial and free backbone.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official backbone.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with backbone.js

Remarks

Backbone is a simple but robust client-side JavaScript library for building applications. Data is represented as Models, which can be gathered into Collections. Model state is displayed with Views.

Backbone attempts to provide the minimal set of data structure and user interface primitives that would be useful in a JavaScript web application. Its goal is to provide these tools without dictating how to use them or what your use-case should look like. This means that the developer is given a lot of freedom to design the full experience of their application.

Examples

Basic Setup

Backbone requires Underscore and (optionally) jQuery - for DOM manipulation (using Backbone.View) and RESTful persistence.

The quickest way to get up and running with Backbone is to create an `index.html` file with simple script tags in the HTML `<head>`:

```
<html>
    <head>
        <script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.8.3/underscore-
min.js"></script>

        <script src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.3.3/backbone-
min.js"></script>
    </head>
    <body>
    </body>
</html>
```

Backbone is now available for use in the page.

Example showcasing the basic concepts

The following example is an introduction to:

- Template compilation using underscore
- Accessing variables in a template
- Creating a view
- Rendering a view
- Showing a view

```
<html>
<head>
    <script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.8.3/underscore-
min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.3.3/backbone-
min.js"></script>
</head>
<body>

    <div id="example_container"></div>

    <script type="text/template" id="example_template">
        <label><%= example_label %></label>
        <input type="text" id="example_input" />
        <input type="button" id="example_button" value="Search" />
    </script>
    <script type="text/javascript">
        var ExampleView = Backbone.View.extend({
            // Compile the template using underscore
            template: _.template($("#example_template").html()),
            events: {
                "click #example_button": "onButtonClick"
            },

            initialize: function(options) {
                this.customOption = options.customOption;
            },

            render: function() {
                // Load the compiled HTML into the Backbone "el"
                this.$el.html(this.template({
                    example_label: "My Search"
                }));

                return this; // for chaining, a Backbone's standard for render
            },

            onButtonClick: function(event) {
                // Button clicked, you can access the button that
                // was clicked with event.currentTarget
                console.log("Searching for " + $("#example_input").val());
            }
        });
        $(function() {
            //show the view inside the div with id 'example_container'
            var exampleView = new ExampleView({
                el: $("#example_container"),
                customOption: 41,
            });
            exampleView.render();
        });
    </script>
</body>
</html>
```

Example of Using Backbone (Primarily Backbone.Model)

Backbone.js is made up of four separate components: Collections, Models, Routers, and Views.

Each of these serve different purposes:

- `Model` - represents a single data object, but adds additional functionalities not provided by native JavaScript objects, such as an event system and a more convenient way to retrieve and send data to a remote server
- `Collection` - represents a set or "collection" of Models and provides functionalities to manage its models.
- `View` - represents a single part of the user interface; each View wraps an HTML DOM element, and provides structure for working with that element as well as convenience features like simple event binding.
- `Router` - enables a "single page application" by allowing an application to trigger different logic (e.g. show different pages) in response to the URL changes.

Create your own flavors

Before we look at how to use each of these components, let's first take a quick look at Backbone's class system. To create a new sub-class of a Backbone class, you simply call the `extend` method of the original class, and pass it the instance properties and (static) class properties as objects:

```
const MyModelClass = Backbone.Model.extend({
    instanceMethod: function() { console.log('Instance method!'); },
}, {
    staticMethod: function() { console.log('Static method!'); },
});
```

Just as with any other class system, instance methods can be called on instances (objects) of the class, while static methods are called directly on the class itself (the constructor):

```
var myInstance = new MyModelClass();

// Call an instance method on our instance
myInstance.instanceMethod(); // logs "Instance method!"

// Call a static method on our class
MyModelClass.staticMethod(); // logs "Static method!"
```

Using a class

Now, let's look at a quick example of how you can use each class. We'll start with a `Model` of a book.

```
const Book = Backbone.Model.extend({
    idAttribute: 'isbn',
    urlRoot: '/book'
});
```

Let's break down what just happened there. First, we created a `Book` subclass of `Model`, and we gave it two instance properties.

- `idAttribute` tells Backbone to use the "isbn" attribute of the model as its ID when performing AJAX operations.
- `urlRoot`, tells Backbone to look for book data on `www.example.com/book`.

Now let's create an instance of a book, and get its data from the server:

```
var huckleberryFinn = new Book({ isbn: '0486403491' });
huckleberryFinn.fetch({
    // the Backbone way
    success: (model, response, options) => {
       console.log(model.get('name')); // logs "Huckleberry Finn"
    }
}).done(() => console.log('the jQuery promise way'));
```

When we created a new `Book` we passed it an object, and Backbone uses this object as the initial "attributes" (the data) of the `Model`. Because Backbone knows the `idAttribute` is `isbn`, it knows that the URL for our new Book is `/book/0486403491`. When we tell it to `fetch`, Backbone will use jQuery to make an AJAX request for the book's data. `fetch` returns a promise (just like `$.ajax`), which you can use to trigger actions once the fetch has completed.

Attributes can be accessed or modified by using the `get` or `set` methods:

```
huckleberryFinn.get('numberOfPages'); // returns 64

huckleberryFinn.set('numberOfPages', 1); // changes numberOfPages to 1
```

`Models` also have an event system that you can use to react when things happen to a `Model`. For instance, to log a message whenever the `numberOfPages` changes, you could do:

```
huckleberryFinn.on('change:numberOfPages', () => console.log('Page change!'));
```

For a more detailed introduction to the other Backbone classes, view their individual documentation pages.

Hello Web (Basic "Hello World"-type setup)

```
<html>
    <head>
        <script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.8.3/underscore-
min.js"></script>

        <script src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.3.3/backbone-
min.js"></script>

        <script>
            $( function(){
                ( function(){
                    var View = Backbone.View.extend( {
```

```
                        "el": "body",
                        "template": _.template( "<p>Hello, Web!</p>" ),

                        "initialize": function(){
                            this.render();
                        },
                        "render": function(){
                            this.$el.html( this.template() );
                        }
                    } );

                    new View();
                })()
            } );
        </script>
    </head>
    <body>
    </body>
</html>
```

Read Getting started with backbone.js online: https://riptutorial.com/backbone-js/topic/1619/getting-started-with-backbone-js

Chapter 2: Collection

Syntax

- // New custom collection
  var MyCollection = Backbone.Collection.extend(properties, [classProperties]);
- // New collection instance
  var collection = new Backbone.Collection([models], [options]);

Parameters

Parameter	Details
properties	Instance properties.
classProperties	*Optional.* Properties that exist and are shared with every collection instance of this type.
models	*Optional.* The initial array of models (or objects). If this parameter is left out, the collection will be empty.
options	*Optional.* Object which serves to configure the collection and is then passed to the `initialize` function.

Remarks

Collections are ordered sets of models. You can bind `"change"` events to be notified when any model in the collection has been modified, listen for `"add"` and `"remove"` events, `fetch` the collection from the server, and use a full suite of Underscore.js methods.

Any event that is triggered on a model in a collection will also be triggered on the collection directly, for convenience. This allows you to listen for changes to specific attributes in any model in a collection.

Examples

Create a custom collection

To create a new collection "class":

```
var Books = Backbone.Collection.extend({
    // books will be sorted by title
    comparator: "title",
```

```
    initialize: function(models, options) {
        options = options || {};

        // (Optional) you can play with the models here
        _.each(models, function(model) {
            // do things with each model
        }, this);

        this.customProperty = options.property;
    },
});
```

All the properties are optional and are there only as a demonstration. A `Backbone.Collection` can be used as-is.

Then using it is as simple as:

```
var myBookArray = [
    { id: 1, title: "Programming frontend application with backbone" },
    { id: 2, title: "Backbone for dummies" },
];


var myLibrary = new Books(myBookArray, {
    property: "my custom property"
});


myLibrary.each(function(book){
    console.log(book.get('title'));
});
```

Will output:

> Programming frontend application with backbone
> Backbone for dummies

Fetching and rendering data from the server

We need to define a collection with a `url` property. This is the url to an API endpoint which should return a json formatted array.

```
var Books = Backbone.Collection.extend({
    url: "/api/book",
    comparator: "title",
});
```

Then, within a view, we'll fetch and render asynchronously:

```
var LibraryView = Backbone.View.extend({
    // simple underscore template, you could use
    // whatever you like (mustache, handlebar, etc.)
    template: _.template("<p><%= title %></p>"),
```

```
    initialize: function(options) {
        this.collection.fetch({
            context: this,
            success: this.render,
            error: this.onError
        });
    },

    // Remember that "render" should be idempotent.
    render: function() {
        this.$el.empty();
        this.addAll();

        // Always return "this" inside render to chain calls.
        return this;
    },

    addAll: function() {
        this.collection.each(this.addOne, this);
    },

    addOne: function(model) {
        this.$el.append(this.template(model.toJSON()));
    },

    onError: function(collection, response, options) {
        // handle errors however you want
    },
});
```

Simplest way to use this view:

```
var myLibrary = new LibraryView({
    el: "body",
    collection: new Books(),
});
```

Collection.url()

By default, the `url` property is not defined. Calling `fetch()` (while using the default `Backbone.sync`) will result in a GET request to the results of `url`.

```
var Users = Backbone.Collection.extend({

  url: '/api/users',

  // or

  url: function () {
    return '/api/users'
  }

});

var users = new Users();
users.fetch() // GET http://webroot/api/users
```

Read Collection online: https://riptutorial.com/backbone-js/topic/6022/collection

Chapter 3: Model

Syntax

- var MyModel = Backbone.Model.extend(properties, [classProperties]); // Create a custom model
- var model = new Backbone.Model([attributes], [options]); // Instanciate a model object

Parameters

Parameter	Details
properties	Instance properties.
classProperties	*Optional.* Properties that exist and are shared with every model instance of this type.
attributes	*Optional.* Initial values of the model's `attributes`. If this parameter is left out, the model will be initialized with the values specified by the model's `defaults` property.
options	*Optional.* Object which serves to configure the model and is then passed to the `initialize` function.

Examples

Creating models

Backbone models describe how data is stored using JavaScript objects. Each model is a hash of fields called attributes and the behaviour of the model including validation is described by options.

A model of Todo item in a TodoApp would be

```
var ToDo = Backbone.Model.extend({
  defaults: {
    assignee: '',
    task: ''
  },

  validate: function(attrs) {
    var errors = {},
        hasError = false;

    if(!attrs.assignee) {
      errors.assignee = 'assignee must be set';
      hasError = true;
    }
```

```
    if(!attrs.task) {
      errors.task = 'task must be set';
      hasError = true;
    }

    if(hasError) {
      return errors;
    }
  }
});
```

Extending models

```
var Vehicle = Backbone.Model.extend({

  description: function () {
    return 'I have ' + this.get('wheels') + ' wheels';
  }

});

var Bicycle = Vehicle.extend({

  defaults: {
    wheels: 2
  }

});

var Car = Vehicle.extend({

  defaults: {
    wheels: 4
  }

});

var bike = new Bicycle();
bike.description() // I have 2 wheels;

var car = new Car();
car.description() // I have 4 wheels;;
```

Model.urlRoot & Model.url()

By default, the `urlRoot` property is not defined. This `urlRoot` property is used by the `url` method to create a relative URL where the model's resource would be located on the server.

```
var User = Backbone.Model.extend({

  urlRoot: '/api/users',

  // or

  urlRoot: function () {
    return '/api/users'
```

```
  }

});

var user = new User();
```

The `url` method will firstly check if the model's `idAttribute` (defaulted at 'id') has been defined. If not, the model `isNew` and `url` will simply return the results of `urlRoot`.

```
user.url() // /api/users
```

If the model's `idAttribute` has been defined, `url` will return the `urlRoot` + the model's `idAttribute`

```
user.set('id', 1);
user.url() // /api/users/1
```

Calling `save` on a new model will result in a POST request to the results of `url`

```
var user = new User({ username: 'johngalt' });
user.save() // POST http://webroot/api/users
```

Calling `save` on an existing model will result in a PUT request to the results of `url`

```
user.set('id', 1);
user.set('username', 'dagnytaggart');
user.save() // PUT http://webroot/api/users/1
```

Calling `fetch` on an existing model will result in a GET request to the results of `url`

```
user.fetch() // GET http://webroot/api/users/1
```

Calling `destroy` on an existing model will result in a DELETE request to the results of `url`

```
user.destroy() // DELETE http://webroot/api/users/1
```

Read Model online: https://riptutorial.com/backbone-js/topic/4056/model

Chapter 4: Router

Examples

Creating a router

The web server serves the user based on the request sent by the browser but how the user will tell the browser what he/she is looking for, that's when we need URL. Every web page on the internet has got a URL that can be bookmarked, copied, shared, and saved for future reference. In single page Backbone app, everything we see is a view, views are formed and rendered to show specific page but what if the user want's to see the same information again. To achieve this we need to implement a Backbone router based app which will render views based on the route name and parameters.

A very simple example of a Backbone router:

```
var UserList = Backbone.Router.extend({
   routes: {//List of URL routes with the corresponding function name which will get called
when user will visit a page having URL containing this route
       "list":                "list",    // localhost:8080/#list
       "search/:name":        "search",  // localhost:8080/#search/saurav
       "search/:name/p:page": "search",  // localhost:8080/#search/kiwis/p7
       "profile/:userId":     "profile" // localhost:8080/#profile/92
   },
   list: function() {
       var userCollection = new UserCollection();
       var userCollectionView = new UserCollectionView();
       userCollection.fetch({remove : true, data:{}, success: function(){
           for(var i = 0; i < userCollection.length; i++ ){
               var userModel = userCollection.at(i);
               var userView = new UserView({ model: userModel });
               userView.render();
               userCollectionView.$el.append(userView.$el);
           }
        }});
   },
   search: function(name, page) {
       var userCollection = new UserCollection();
       var userCollectionView = new UserCollectionView();
       userCollection.fetch({remove : true, data:{pageNo: page, name: name}, success:
function(){
           for(var i = 0; i < userCollection.length; i++ ){
               var userModel = userCollection.at(i);
               var userView = new UserView({ model: userModel });
               userView.render();
               userCollectionView.$el.append(userView.$el);
           }
        }});
   },
   profile: function(userId){
       var userModel = new UserModel({id: userId});
       userModel.fetch({success: function(){
             var userView = new UserView({model: userModel});
             userView.render();
```

```
        }});
    }
});
var userList = new UserList();
Backbone.history.start();
```

The above code is only an example code which demonstrates how you can create a Backbone router and get parameters from URL to render corresponding views.

Explanation of how the above router will work and behave:

Cases by URLs:

- **localhost:8080/#search/saurav** - router's "search" route (function) will get triggered with parameters name = "saurav" and page = null, now userCollection.fetch() function will fetch all the users having name = "saurav" from backend and it will render each user's details one by one.
- **localhost:8080/#search/saurav/p6** - router's "search" route (function) will get triggered with parameters name = "saurav" and page = 6, now userCollection.fetch() function will fetch all the users having name = "saurav" of page 6 from backend and it will render each user's details one by one.
- **localhost:8080/#list** - router's "list" route (function) will get triggered, now userCollection.fetch() function will fetch all the users from backend and it will render each user's details one by one.
- **localhost:8080/#profile/92** - router's "profile" route (function) will get triggered, we will create a new instance of userModel with id = userId i.e. 92 and we will fetch the user's details from backend and render the userView with that data.

An easy to experiment example:

Visit http://backbonejs.org in chrome browser, open the developer tools console and paste the below code-

```
var Workspace = Backbone.Router.extend({

    routes: {
        "help":                 "help",    // #help
        "search/:query":        "search",  // #search/kiwis
        "search/:query/p:page": "search"   // #search/kiwis/p7
    },

    help: function() {
        console.log("help");
    },

    search: function(query, page) {
        console.log("searched " + query + " " + page);
    }
});
var work = new Workspace();
```

```
Backbone.history.start();
```

Now, replace the URL in the browser with "http://backbonejs.org/#search/kiwis/p9" and hit the enter key. This will trigger "search" route (function) with parameters query = "kiwis" and page = 9 and you will see an output in the browser console i.e. "searched kiwis 9".

Changing the route through code:

- **Case 1**: execute code `work.navigate("search/kiwis/p7", {trigger: true});` in the console and it will print output "searched kiwis 7" but if you will try to execute the same code with the same parameter then nothing will happen, see next case.
- **Case 2**: execute code `work.navigate("search/kiwis/p7", {trigger: false});` in the console it will not print anything because route will not get triggered.
- **Case 3**: In case if you need to reload the current route once again then you will need to execute this code `Backbone.history.loadUrl("search/kiwis/p7");`.
- **Case 4**: Executing the code `work.navigate("search/kiwis/p15");` will just change the URL but it will not trigger the corresponding route (function).
- **Case 5**: Executing the code `work.navigate("search/kiwis/p11", {trigger: true});` will change the URL and trigger the route.
- **Case 6**: Executing code `work.navigate("search/kiwis/p17", {trigger: true, replace: true})` will replace the existing route with this route hence clicking browser's back button will take you 2 routes back to "search/kiwis/p15".

Read Router online: https://riptutorial.com/backbone-js/topic/7566/router

Chapter 5: Sync

Introduction

`sync` is a function that Backbone uses to handle all sending or receiving of data to/from a remote server. The default implementation uses jQuery (or Zepto) to perform AJAX operations when data is synced. However, this method can be overriden to apply different syncing behavior, such as: - Using `setTimeout` to batch multiple updates into a single request - Sending model data as XML instead of JSON - Using WebSockets instead of Ajax

Syntax

- sync(method, model, options)

Parameters

parameter	details
method	create , read , update , delete
model	the model to be saved (or collection to be read)
options	success and error callbacks, and all other jQuery request options

Examples

Basic Example

The sync() method reads and fetched the model data

```
     Backbone.sync = function(method, model) {
        document.write("The state of the model is:");
        document.write("<br>");

        //The 'method' specifies state of the model
        document.write(method + ": " + JSON.stringify(model));
     };

     //'myval' is a collection instance and contains the values which are to be fetched in the
collection
     var myval = new Backbone.Collection({
        site:"mrfarhad.ir",
        title:"Farhad Mehryari Official Website"
     });

     //The myval.fetch() method displays the model's state by delegating to sync() method
     myval.fetch();
```

this code will outputs :

```
The state of the model is:
read: [{"site":"mrfarhad.ir","title":"Farhad Mehryari Official Website"}]
```

Read Sync online: https://riptutorial.com/backbone-js/topic/8178/sync

Chapter 6: url and urlRoot

Examples

Modifying Model.url()

`Model.url` and `Collection.url` are only used internally by the default `Backbone.sync` method. The default method assumes you are tying into a RESTful API. If you are using a different endpoint design, you will want to override the `sync` method and may want utilize the `url` method.

```
var Model = Backbone.Model.extend({

  urlRoot: '/path-to-model',

  url: function (path) {
    var url = this.urlRoot + '/' + path;
    if (this.isNew()) {
      return url;
    }
    return url + '/' + this.get(this.idAttribute);
  }

});

var model = new Model();
model.url('create'); // /path-to-model/create
model.set('id', 1);
model.url('read'); // /path-to-model/read/1
model.url('update'); // /path-to-model/update/1
model.url('delete'); // /path-to-model/delete/1
```

Read url and urlRoot online: https://riptutorial.com/backbone-js/topic/6430/url-and-urlroot

Chapter 7: View

Syntax

- Create: `var View = Backbone.View.extend( { /* properties */ } );`
- Construct: `var myView = new View( /* options */ );`
- `initialize`: method automatically called after construction
- `render`: method used to update `this.el` with new content

Examples

A View Bound to Existing HTML

Assuming this HTML in the page:

```
<body>
    <div id="myPage">
    </div>
</body>
```

A view can be bound to it with:

```
var MyPageView = Backbone.View.extend( {
    "el": "#myPage",
    "template": _.template( "<p>This is my page.</p>" ),

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

    "render": function(){
        this.$el.html( this.template() );
    }
} );

new MyPageView();
```

The HTML in the browser will now show:

```
<body>
    <div id="myPage">
        <p>This is my page.</p>
    </div>
</body>
```

View's initialize function

`initialize` is called by Backbone right after a View is constructed.

Optional parameters

The `initialize` function receives any arguments passed to the view's constructor. Commonly, the options hash that is used to pass the view's default options:

```
['model', 'collection', 'el', 'id', 'attributes', 'className', 'tagName', 'events']
```

You can add any custom attributes to the options hash, and/or custom parameters.

```
var MyView = Backbone.View.extend({
    initialize: function(options, customParam){
        // ensure that the 'options' is a hash to avoid errors if undefined.
        options = options || {};
        this.customAttribute = options.customAttribute;
        this.customParam = customParam;
    },
});
```

And constructing the view:

```
var view = new MyView({
    model: new Backbone.Model(),
    template: "<p>a template</p>",
    customAttribute: "our custom attribute"
}, "my custom param");
```

Note that the all the default view options are automatically added to the view object, so it's unnecessary to do that in the `initialize` function.

Immediately render pattern

One common pattern for the `initialize` method is to call the `render` method so that any newly constructed View is immediately rendered

This pattern should only be used in instances where constructing the object should immediately render it to the HTML document, bind all of the event listeners, and perform all the other actions associated with placing content in the DOM.

```
var MyView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },

    render: function() {
        this.$el.html("<p>I'm running!</p>");
    }
});
```

It should be noted, however, that *some* Views should not be immediately rendered until `.render` is

called manually (or by some other method).

Another common `initialize` pattern is to add things to the View object that will be needed later:

```
var MyView = Backbone.View.extend({
    el: "body",
    template: _.template( "<p>This is <%= name %>'s page</p>" ),

    initialize: function(){
        this.name = "Bill";

        this.render();
    },

    render: function(){
        var viewTemplateData = {
            name: this.name
        };

        this.$el.html( this.template( viewTemplateData ) );
    }
});
```

The DOM will now contain `<p>This is Bill's page</p>` in the `body`.

Read View online: https://riptutorial.com/backbone-js/topic/2728/view

Credits

S. No	Chapters	Contributors
1	Getting started with backbone.js	Community, Emile Bergeron, machineghost, Nighon, nikhil mehta, Peter, rockerest
2	Collection	Cakes, Emile Bergeron
3	Model	Cakes, Emile Bergeron, Louis, Peter, RamenChef
4	Router	saurav
5	Sync	Braiam, Farhad, machineghost
6	url and urlRoot	Cakes, Emile Bergeron
7	View	Emile Bergeron, Louis, rockerest, T J