Backbone

Intro

Source: http://addyosmani.github.io/backbone-fundamentals

Old-style apps did heavy-lifting of data in back end

New client applications pull raw data from the server and render it into the browser when and where it is needed.

The need for fast, complex, and responsive Ajax-powered web applications demands storage of logic on the client side.

MVC

For an application to show real-time data, any change to the data in its Model should result in the View being refreshed instantly. Imagine a stock market app.

Client-side MVC vs Single Page Apps

SPAs can also take advantage of browser features like the History API to update the address seen in the location bar when moving from one view to another. These URLs also make it possible to bookmark and share a particular application state, without the need to navigate to completely new pages.

Quick Summary of Concepts

Models

- validate attributes
- Persistence - save the state of a model somehwere
- announces updates to its data to any Views that are listening
- can be grouped together in a Collection

Views

- edit Model data
- a `render` function in the View renders the contents of the **Model** using a JS templating engine
- can trigger an update when the **Model** changes
- it is not the responsibility of the **View** to deal with User Interaction (e.g. clicks). The

Controller should handle this, which in the case of Backbone is achieved with an event handler.

Controllers

A **Controller** is responsible for handling changes the User makes in the **edit View** for a particular item, updating its **Model** when the User has finished editing.

Backbone does not really have **Controllers** as such. Backbone's **Views** typically contain **Controller** logic, and **Routers** are used to help manage application state.

Controllers facilitate **Views'** responses to different user input and are an example of the Strategy pattern.

Backbone Facts

- Event-driven communication between **Views** and **Models**. Easy to add event-listeners to **Model Attributes**, giving developers fine-grained control over what changes in the **View.**
- Extensive eventing system. It's trivial to add support for pub/sub in Backbone
- Prototypes are instantiated with the new keyword
- Agnostic about templating frameworks, however Underscore's micro-templating is available by default
- Supports data bindings through manual events or a separate Key-value observing (KVO) library
- Support for RESTful interfaces out of the box, so Models can be easily tied to a backend

Concepts in Depth

Models

Backbone models contain data for an application as well as the logic around this data.

We can use a model to represent the concept of a Todo item including its attributes like title (Todo content) and completed (current state of the Todo).

`Initialize` is like a constructor; it runs on instantiation of a Model

```
initialize: function(){
      console.log('This model has been initialized.');
}
```

`Default` values allow you to ensure that certain values are set on every Model that is instantiated

`Model.get()` allows you to retrieve a value from a Model object

`Model.set()` allows you to set the value of an attribute on a Model object. It also triggers a change event, which can be listened to.

`modelObject.toJSON()` returns all the values of the object

Listening to Events

You can silence change events on Models with `{silent:true}`

```
var Person = new Backbone.Model();
Person.on("change:name", function() { console.log('Name changed'); });
Person.set({name: 'Jeremy'}, {silent: true});
// no log entry
```

You can check if an attribute has changed with `Model.hasChanged(attribute)` or if *anything* has changed with `Model.hasChanged(null)`

To listen for *any* change to your Model, place an `this.on('change', function(){})` in your `initialize` function.

```
initialize: function(){
    console.log('This model has been initialized.');
    this.on('change', function(){
        console.log('- Values for this model have changed.');
    });
}
```

FYI - Changing more than one attribute at the same time only triggers the listener once.

To listed for changes to specific attributes, place an `this.on('change:attribute', function(){})` in your `initialize` function.

```
initialize: function(){
    console.log('This model has been initialized.');
    this.on('change:title', function(){
        console.log('Title value for this model has changed.');
    });
}
```

Validation

Backbone supports model validation through `model.validate()`, which allows checking the attribute values for a model prior to setting them. By default, validation occurs when the model is persisted using the `save()` method or when `set()` is called if `{validate:true}` is passed as an argument.

If an error is returned:

- An `invalid` event will be triggered, setting the validationError property on the model with the value which is returned by this method.
- `.save()` will not continue and the attributes of the model will not be modified on the server.

You can catch the `invalid` event in your `initialize` function (similar to above code) with

```
this.on("invalid", function(model, error){
    console.log(error);
});
```

Note: the `attributes` object passed to the `validate` function represents what the attributes would be after completing the current `set()` or `save()`.

It is not possible to change any Number, String, or Boolean attribute of the the **model attributes** within the function, but it is possible to change attributes in nested objects.

Views

Views contain the logic behind the presentation of the model's data to the user.

A view's `render()` method can be bound to a model's `change()` event, enabling the view to instantly reflect model changes without requiring a full page refresh.

In Backbone 1.1.0, if you want to access passed options in your view, you will need to save them as follows:

```
initialize: function (options) {
    this.options = options || {};
}
```

The El Element

`el` is a DOM element that you build in the View.

You give it its tag name with `tagName`, its id with `id`, and its class name with `className`.

Once you give it those attributes, you can fill it with content in the `render` function by using `this.$el.html()`.

```
var Test = Backbone.View.extend({
  tagName: 'p',
  className: 'brook',
  id: 'shmuk',

  render: function() {
    this.$el.html('ummmm..... yeah');
  }
});

var myTest = new Test();
myTest.render();
console.log(myTest.el);
console.log(myTest.$el);
```

The above code creates the DOM element below but doesn't append it to the DOM.

`myTest.$el` will give you the actual jQuery object itself.

There are two ways to associate a DOM element with a view: a new element can be created for the view and subsequently added to the DOM **OR** a reference can be made to an element which already exists on the page.

If the element already exists on the page, you can set `el` as a CSS selector that matches the element.

```
el: '#footer'
```

Alternatively, you can set `el` to an existing element when creating the view:

```
var todosView = new TodosView({el: $('#footer')});
```

NOTE: When declaring a View, options, `el`, `tagName`, `id` and `className` may be defined as functions, if you want their values to be determined at runtime.

NOTE 2: `this.$el` is equivalent to `$(this.el)` and `this.$()` is used to find subelements with a matching identifier - e.g. `this.$('.edit')` finds elements with a class of `edit`.

If you wish to target a different element on the page, or change the element you were targeting, you can use `setElement`

The Render Function

`render()` is an optional function that defines the logic for rendering a template.

The `_.template` method in Underscore compiles JavaScript templates into functions which can be evaluated for rendering.

In the following template, `example-target` is used to pass the template to `_.template()`, which then compiles and stores it in the `myTemplate` variable in the View.

```
// index.html
<script id="example-target" type="text/template">
    <span>Name: <%= name %></span>
</script>

// myView.js
myTemplate = _.template()
```

The `render()` method uses this template by passing it the `toJSON()` encoding of the attributes of the model associated with the view.

A common Backbone convention is to return `this` at the end of `render()`.

Going Through a Full Example

main.js

```
// Model
var ExampleModel = Backbone.Model.extend({
    defaults: {
        name: 'Ben'
    }
});

// View
var DisplayView = Backbone.View.extend({
    el: $('.example-content'),
    template: _.template( $('#example-target').html() ),

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

var example_model = new ExampleModel({name: 'Jack'});
var display_view = new DisplayView({ model: example_model });
display_view.render();
```

index.html

```
<div class="example-content">
    <script id="example-target" type="text/template">
        <span>Name: <%= name %></span>
    </script>
</div>
```

- `el` - is the element in the DOM that we are targeting
- `_.template()` - is the template inthe DOM that we are targeting
- `this.template(this.model.toJSON())` - sends the attributes of this model to the template
- `this.$el.html(exampleTemplate);` - populate this element with the populated template. i.e. render the template on the page

View.remove

`View.remove` - Removes a view from the DOM, and calls `stopListening` to remove any bound events that the view has `listenTo`'d.

Events

Event Listeners can ben attached to `el`-relative selectors

An event takes the form of a key-value pair `'eventName selector': 'callbackFunction'` and a number of DOM event-types are supported, including `click`, `submit`, `mouseover`, and `dblclick`. There has to be a function with the exact name of `callbackFunction` for every event listener.

```
// A sample view
var TodoView = Backbone.View.extend({
    tagName:  'li',

    events: {
      'click .toggle': 'toggleCompleted',
      'blur .edit': 'close'
    },
});
```

Collections

Collections are groupings of Model instances.

Generally, you define the type of model that you want to target in the collection, and you then assign an array of model instances to the collection.

```
// Collection
var ExampleCollection = Backbone.Collection.extend({
  model: ExampleModel
});

var myEgModel = new ExampleModel({name: 'Mark'});
var myEgCollection = new ExampleCollection([myEgModel]);
```

Using add and remove

You can add and remove **Model instances** from a Collection using `add` and `remove`.

```
var a = new ExampleModel({name: 'Mark'}),
    b = new ExampleModel({name: 'Bob'}),
    c = new ExampleModel({name: 'Paul'});

var ExampleCollection = Backbone.Collection.extend({
  model: Todo,
});

var myEgCollection = new ExampleCollection([a,b]);
myEgCollection.add(c);
myEgCollection.remove([b,c]);
```

If you pass `{merge: true}` when using `add`, duplicate models will have their attributes merged into existing models, instead of being ignored (default behavior).

```
var items = new Backbone.Collection;
items.add([{ id : 1, name: "Dog" , age: 3}, { id : 2, name: "cat" , age: 2}]);
items.add([{ id : 1, name: "Bear" }], {merge: true });
items.add([{ id : 2, name: "lion" }]); // merge: false

console.log(JSON.stringify(items.toJSON()));
// [{"id":1,"name":"Bear","age":3},{"id":2,"name":"cat","age":2}]
```

Retrieving Models

`Collection.get()` retrieves a model from a collection, taking a single id.

```
var myTodo = new Todo({title:'Read the whole book', id: 2});
var todos = new TodosCollection([myTodo]);
var todo2 = todos.get(2);
console.log(todo2 === myTodo); // true
```

Anytime you're exchanging data between the client and a server, you will need a way to uniquely identify models. In Backbone, this is done using the `id`, `cid`, and `idAttribute` properties.

Each model in Backbone has an id, which is a unique identifier that is either an integer or string (e.g., a UUID). Models also have a `cid` (client id) which is automatically generated by Backbone when the model is created. Either identifier can be used to retrieve a model from a collection.

The main difference between them is that the `cid` is generated by Backbone; it is helpful when you don’t have a true id - this may be the case if your model has yet to be saved to the server or you aren’t saving it to a database.

The idAttribute is the identifying attribute name of the model returned from the server (i.e., the id in your database). This tells Backbone which data field from the server should be used to populate the id property. By default, it assumes `id`, but this can be customized as needed. For instance, if your server sets a unique attribute on your model named `userId` then you would set idAttribute to `userId` in your model definition.

The value of a model’s idAttribute should be set by the server when the model is saved. After this point you shouldn’t need to set it manually.

Internally, Backbone.Collection contains an array of models enumerated by their id property, if the model instances happen to have one. When collection.get(id) is called, this array is checked for existence of the model instance with the corresponding id.

Event Listeners

As collections represent a group of items, we can listen for `add` and `remove` events which occur when models are added to or removed from a collection.

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("add", function(todo) {
  console.log("I should " + todo.get("title") + ". Have I done it before? "  +
(todo.get("completed") ? 'Yeah!': 'No.' ));
});
```

In addition, we’re also able to bind to a `change` event to listen for changes to any of the models in the collection.

```
// log a message if a model in the collection changes
TodosCollection.on("change:title", function(model) {
    console.log("Changed my mind! I should " + model.get('title'));
});

myTodo.set('title', 'go fishing');
// Logs: Changed my mind! I should go fishing
```

jQuery-style event maps of the form `obj.on({click: action})` can also be used.

```
myTodo.on({
   'change:title' : titleChanged,
   'change:completed' : stateChanged
});
```

Backbone events also support a `once()` method, which ensures that a callback only fires **one time** when a notification arrives. Similar to jQuery's `one` method.

Resetting/Refreshing Collections

Rather than adding or removing models individually, you might want to update an entire collection at once. `Collection.set()` takes an array of models and performs the necessary add, remove, and change operations required to update the collection.

```
TodosCollection.set([
    { id: 1, title: 'go to Jamaica.', completed: true },
    { id: 2, title: 'go to China.', completed: false },
    { id: 4, title: 'go to Disney World.', completed: false }
]);
```

`set` can be used for smart updating sets of models. When a model in this list isn't present in the collection, it is added. If it's present, its attributes will be merged. Models which are present in the collection but not in the list are removed.

```
// Create models for some members of the Beatles
var john = new Beatle({ firstName: 'John', lastName: 'Lennon'});
var paul = new Beatle({ firstName: 'Paul', lastName: 'McCartney'});
var george = new Beatle({ firstName: 'George', lastName: 'Harrison'});

var someBeatles = new Backbone.Collection([john, paul]);

// Removes paul and adds george... phew!
someBeatles.set([john, george]);
```

Use `Collection.reset()` to simply overwrite the entire collection.

```
// Only 'go to Cuba', completed: false will be in the collection
TodosCollection.reset([
  { title: 'go to Cuba.', completed: false }
]);
```

Another useful tip is to use `reset` with no arguments to clear out a collection completely. This is handy when dynamically loading a new page of results where you want to blank out the current page of results.

Note that using `Collection.reset()` doesn’t fire any `add` or `remove` events. A `reset` event is fired instead.

Also note that listening to a reset event, the list of previous models is available in `options.previousModels`, for convenience.

```
.on('reset', function(todos, options) {
  console.log(options.previousModels);
});
```

Underscore Utility Functions

We have access to Underscore utility functions in Backbone.

forEach: iterate over collections

```
var todos = new Backbone.Collection();

todos.add([
  { title: 'go to Belgium.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Austria.', completed: true }
]);

// iterate over models in the collection
todos.forEach(function(model){
  console.log(model.get('title'));
});
```

sortBy: sort a collection on a specific attribute

```
// sort collection
var sortedByAlphabet = todos.sortBy(function (todo) {
    return todo.get("title").toLowerCase();
});

// Now you can check with forEach
sortedByAlphabet.forEach(function(model){
  console.log(model.get('title'));
});
```

Also, check `map`, `min/max`, `pluck`, `filter`, `indexOf`, `any`, `size`, `isEmpty`, `groupBy`, `pick`, `omit`, `keys` and `values`, `pairs`, `invert`, and other functions at Underscorejs.org

Chaining Methods

Some methods can be chained (e.g. `add`). Others that don't return `this` can be changed using the `chain` function.

```
collection
    .add({ name: 'John', age: 23 })
    .add({ name: 'Harry', age: 33 })
    .add({ name: 'Steve', age: 41 });
```

RESTful Persistence

Backbone dramatically simplifies the code you need to write to perform RESTful synchronization with a server through a simple API on its models and collections.

Fetching models from the server

`Collections.fetch()` retrieves a set of models from the server in the form of a JSON array by sending an HTTP GET request to the URL specified by the collection's `url` property (which may be a function). When this data is received, a `set()` will be executed to update the collection.

Saving Models to the Server

While Backbone can retrieve an entire collection of models from the server at once, updates to models are performed individually using the model's `save()` method. When `save()` is called on a model that was fetched from the server, it constructs a URL by appending the model's id to the collection's URL and sends an `HTTP PUT` to the server.

If the model is a new instance that was created in the browser (i.e., it doesn't have an id) then an

`HTTP POST` is sent to the collection's URL.

`Collections.create()` can be used to create a new model, add it to the collection, and send it to the server in a single method call.

As mentioned earlier, a model's `validate()` method is called automatically by `save()` and will trigger an invalid event on the model if validation fails.

Deleting models from the server

A model can be removed from the containing collection and the server by calling its `destroy()` method. Unlike `Collection.remove()` which only removes a model from a collection, `Model.destroy()` will also send an `HTTP DELETE` to the collection's URL.

Calling `destroy` on a Model will return `false` if the model `isNew`

```
var todo = new Backbone.Model();
console.log(todo.destroy());
// false
```

Applying Patches

Specifying the `{patch: true}` option to `Model.save()` will cause it to use `HTTP PATCH` to send only the changed attributes (i.e partial updates) to the server instead of the entire model i.e `model.save(attrs, {patch: true})`

Similarly, passing the `{reset: true}` option to `Collection.fetch()` will result in the collection being updated using `reset()` rather than `set()`.

Events

Events are an inversion of control, giving the power to call functions to event handlers. Events are available to all the Backbone classes (e.g. Models, Collections, Views, etc).

```
Backbone.on('event', function() {console.log('Hi');});
Backbone.trigger('event');
```

on(), off(), and trigger()

`Backbone.Events` can give any object the ability to bind and trigger custom events.

var ourObject = {};

```
// Mixin
_.extend(ourObject, Backbone.Events);

// Add a custom event
ourObject.on('dance', function(msg){
  console.log('We triggered ' + msg);
});

// Trigger the custom event
ourObject.trigger('dance', 'our event');
```

The official Backbone.js documentation recommends namespacing event names using colons if you end up using quite a few of these on your page.

```
// Add namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");

// This one triggers nothing as no listener listens for it
ourObject.trigger("dance", "break dancing. Yeah!");
```

`on` binds a callback function to an object. The callback is invoked whenever the event is triggered.

`off` removes callback functions that were previously bound to an object.

```
// Removes event bound to the object
ourObject.off("dance:tap");
```

To remove all callbacks for the event we pass an event name (e.g., `dance`) to the `off()` method on the object the event is bound to. If we wish to remove a specific callback, we can pass that callback as the second parameter:

`trigger` triggers a callback for a specified event (or a space-separated list of events).

```
// Single event
ourObject.trigger("dance", 'just dancing.');

// Multiple events
ourObject.trigger("dance jump skip", 'very tired from so much action.');
```

`trigger` can also pass multiple arguments to the callback function.

A special `all` event is made available in case you would like notifications for every event that occurs on the object

```
ourObject.on("all", function(eventName){
  console.log("The name of the event passed was " + eventName);
});
```

listenTo() and stopListening()

While `on()` and `off()` add callbacks directly to an observed object, `listenTo()` tells an object to listen for events on another object. `stopListening()` can be called on the listener to tell it to stop listening for events.

```
var a = _.extend({}, Backbone.Events);
var b = _.extend({}, Backbone.Events);

a.listenTo(b, 'anything', function(event){ console.log("anything happened"); });

a.stopListening();
```

`stopListening()` can also be used to selectively stop listening based on the event, model, or callback handler.

Practically, every `on` called on an object also requires an `off` to be called in order for the garbage collector to do its job. `listenTo()` changes that, allowing Views to bind to Model notifications and unbind from all of them with just one call - `stopListening()`.

The default implementation of `View.remove()` makes a call to `stopListening()`, ensuring that any listeners bound using `listenTo()` are unbound before the view is destroyed.

Events and Views

Within a View, there are two types of events you can listen for: DOM events and events triggered using the Event API.

DOM events can be bound to using the View's events property or using jQuery.on(). Within callbacks bound using the events property, this refers to the View object; whereas any callbacks bound directly using jQuery will have this set to the handling DOM element by jQuery.

In Event API events, if the event is bound using on() on the observed object, a context parameter can be passed as the third argument. If the event is bound using listenTo() then within the callback this refers to the listener.

Routers

In Backbone, routers provide a way for you to connect URLs to parts of your application. Any piece of your application that you want to be bookmarkable, shareable, and back-button-able, needs a URL.

```
routes: {
    "about" : "showAbout",
    /* Sample usage: http://example.com/#about */

    "todo/:id" : "getTodo",
    /* This is an example of using a ":param" variable which allows us to match
    any of the components between two URL slashes */
    /* Sample usage: http://example.com/#todo/5 */

    "search/:query" : "searchTodos",
    /* We can also define multiple routes that are bound to the same map
function,
    in this case searchTodos(). Note below how we're optionally passing in a
    reference to a page number if one is supplied */
    /* Sample usage: http://example.com/#search/job */

    "search/:query/p:page" : "searchTodos",
    /* As we can see, URLs may contain as many ":param"s as we wish */
    /* Sample usage: http://example.com/#search/job/p1 */

    "todos/:id/download/*documentPath" : "downloadDocument",
    /* This is an example of using a *splat. Splats are able to match any number
of
    URL components and can be combined with ":param"s*/
    /* Sample usage:
http://example.com/#todos/5/download/files/Meeting_schedule.doc */

    /* If you wish to use splats for anything beyond default routing, it's
probably a good
    idea to leave them at the end of a URL otherwise you may need to apply
regular
    expression parsing on your fragment */

    "*other"    : "defaultRoute",
    /* This is a default route that also uses a *splat. Consider the
    default route a wildcard for URLs that are either not matched or where
    the user has incorrectly typed in a route path manually */
    /* Sample usage: http://example.com/# <anything> */

    "optional(/:item)": "optionalItem",
    "named/optional/(y:z)": "namedOptionalItem"
    /* Router URLs also support optional parts via parentheses, without having
       to use a regex.  */
}
```

Usually you only need a single router for your app.

Backbone offers an opt-in for HTML5 pushState support via `window.history.pushState`.

We need to initialize `Backbone.history` as it handles hashchange events in our application, handling routes that have been defined and triggering callbacks.

The `Backbone.history.start()` method will simply tell Backbone that it's okay to begin monitoring all hashchange events.

If you would like to update the URL to reflect the application state at a particular point, you can use the router's `.navigate()` method. By default, it simply updates your URL fragment without triggering the hashchange event.

```
viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit');
    // updates the fragment for us, but doesn't trigger the route
},
```

It is also possible for `Router.navigate()` to trigger the route along with updating the URL fragment by passing the `trigger:true` option, but this usage is discouraged.

A `route` event is also triggered on the router in addition to being fired on `Backbone.history`. See book for more info on this.