Getting started with Backbone

How to bring model-view-controller structure to Ajax web applications

Sebastiano Armeli-Battana

December 13, 2011

Efficient management of the numerous lines of JavaScript code in web applications can be a challenge. Asynchronous JavaScript and XML (Ajax) interactions heavily populate pages to provide a better experience to the user. Single page interfaces, which are becoming more common, are driven by Ajax. Backbone is a JavaScript framework that can be used to create model-view-controller (MVC)-like applications and single page interfaces. In this article, learn how useful Backbone can be for creating Ajax applications or single page interfaces.

Introduction

Increasingly, web applications focus on the front end, using client-side scripting and Ajax interactions. As the complexity of JavaScript applications increases, writing efficient, non-repetitive, and maintainable JavaScript code can be challenging without the right tools and patterns. Model-View-Controller (MVC) is a common pattern used in server-side development to produce code that is organized and easy to maintain. MVC, which allows the separation of data (such as JavaScript Object Notation (JSON) objects, often used in Ajax interactions) from the presentation layer, or document object model (DOM) of a page, is also applicable to client-side development.

Backbone (also known as Backbone.js), a lightweight library created by Jeremy Ashkenas, is useful for creating MVC-like applications. Backbone:

- Has a hard dependency with Underscore.js, a utility-belt library
- Has soft dependencies with jQuery/Zepto
- Updates the HTML of your application automatically, depending upon how the model changes, favoring code maintainability
- Promotes the use of client-side templating, making it unnecessary to embed HTML code inside JavaScript

Models, views, collections, and routers are the main components inside the Backbone framework. In Backbone, models store data retrieved from the server over a RESTful JSON interface. Models are associated with the views, which are in charge of rendering the HTML for a specific UI component and handling events triggered on elements that are part of the view itself.
In this article, learn about the different components of the Backbone.js framework. Explore how MVC applies to Backbone. Through examples, see how useful Backbone can be when creating Ajax applications or single page interfaces (SPIs).

Download the source code used in this article.

**SPI applications: Backbone.Router and Backbone.history**

Applications that have a lot of Ajax interactions are becoming more like applications in which no page refresh occurs. These applications often try to limit interaction to a single page. This SPI approach increases efficiency and speed and the overall application becomes more responsive.

The concept of state replaces the concept of pages. Hash fragments are used to identify a specific state. A hash fragment is the part of a URL following the hash tag (#) and is the key element for this type of application. **Listing 1** shows two different states inside an SPI application using two different hash fragments.

**Listing 1. Two different states inside an SPI or Ajax application**

http://www.example.com/#/state1
http://www.example.com/#/state2

Backbone offers a component called a router (named controller prior to version 0.5) to route client-side states. A router extends the Backbone.Router function and contains a hash map (routes attribute) that associates a state with an action. A specific action is triggered when the application reaches a correlated state. **Listing 2** shows an example of a Backbone router.

**Listing 2. Backbone.Router example: routers.js**

```javascript
App.Routers.Main = Backbone.Router.extend({
    routes : {
        "" : "index",
        "/teams" : "getTeams",
        "/teams/:country" : "getTeamsCountry",
        "/teams/:country/:name : "getTeam"
    },

    index: function(){
        // Homepage
    },

    getTeams: function() {
        // List all teams
    },

    getTeamsCountry: function(country) {
        // Get list of teams for specific country
    },

    getTeam: function(country, name) {
        // Get the teams for a specific country and with a specific name
    },
});
```
Each state created can be bookmarked. The five actions (index, getTeams, getTeamsCountry, getTeamCountry, and fourOfour) are called when the URLs are similar to the following.

- http://www.example.com triggers index()
- http://www.example.com/#/teams triggers getTeams()
- http://www.example.com/#/teams/country1 triggers getTeamsCountry() passing country1 as a parameter
- http://www.example.com/#/teams/country1/team1 triggers getTeamCountry() passing country1 and team1 as parameters
- http://www.example.com/#/something triggers fourOfour() as an * (asterisk) was used.

To start Backbone, instantiate the router on page load and monitor any change on the hash fragment through the instruction Backbone.history.start() method, as in Listing 3.

Listing 3. Application initialization (using jQuery)
```javascript
$(function(){
    var router = new App.Routers.Main();
    Backbone.history.start({pushState : true});
});
```

The Backbone.history object is generated when the router is instantiated; it is an automatic reference to the Backbone.History function. Backbone.history takes care of matching a route to the action defined in the router object. After the start() method is triggered, the fragment attribute inside Backbone.history is created. It contains the value of the hash fragment. This sequence is helpful in managing the browser history according to the sequence of states. To direct the user to the previous state, click the back button of the browser.

In the example in Listing 3, the start() method is invoked with a configuration enabling the HTML5 feature pushState. For a browser supporting pushState, Backbone will monitor the popstate event to trigger a new state. If the browser doesn't support that HTML5 feature, the onhashchange event is monitored. If the browser does not support this event, a polling technique monitors any change on the hash fragment of the URL.

Models and collections
Models and collections are important components of Backbone.js. Models hold the data (often coming from the server) in key-value pairs. To create a model, extend Backbone.Model, as in Listing 4.

Listing 4. Backbone.Model creation
```javascript
App.Models.Team = Backbone.Model.extend({
    defaults : {
        // default attributes
    },
    // Domain-specific methods go here
});
```
The `App.Models.Team` function is a new model function, but an instance of it must be created to use a specific model in an application, as in **Listing 5**.

**Listing 5. Model instantiation**

```javascript
var team1 = new App.Models.Team();
```

The variable `team1` now has the field named `cid`, which is a client identifier in the form of "c" plus a number (for example, c0, c1, c2). Models are defined by attributes that are stored in a hash map. Attributes can be set at instantiation time or with the `set()` method. Attribute values are retrievable through the `get()` method. **Listing 6** shows how to set and get attributes through instantiation or `get()/set()`.

**Listing 6. Model instantiation and get/set methods**

```javascript
// "name" attribute is set into the model
var team1 = new App.Models.Team({
    name : "name1"
});
console.log(team1.get("name")); // prints "name1"

// "name" attribute is set with a new value
team1.set({
    name : "name2"
});
console.log(team1.get("name")); // prints "name2"
```

When working with JavaScript objects, the reason for using the `set()` method to create or set the value of an attribute might not be obvious. One reason is for updating the value, as in **Listing 7**.

**Listing 7. Updating attributes in a wrong way**

```javascript
team1.attributes.name = "name2";
```

*Avoid* using the code in **Listing 7**. Using `set()` is the only way to change the state of the model and to trigger the change event on it. Using `set()` promotes the principle of encapsulation. **Listing 8** below shows how to bind an event handler to the change event. The event handler contains an alert that is triggered when the `set()` method is called, as in **Listing 6**, but is not triggered using the code in **Listing 7**.

**Listing 8. Change event handler in the App.Models.Team model**

```javascript
App.Models.Team = Backbone.Model.extend({
    initialize : function(){
        this.bind("change", this.changed);
    },
    changed : function(){
        alert("changed");
    }
});
```

Another benefit of Backbone is the ease of communicating with the server through Ajax interactions. Invoking a `save()` method on a model will asynchronously save the current state (characterized by the hash map of attributes) to the server through REST JSON APIs. **Listing 9** shows an example.
Listing 9. save method invoked on a model object

```javascript
barca.save();
```

In the background, the `save()` function delegates to `Backbone.sync`, which is the component in charge of making RESTful requests, using the jQuery function `$().ajax()` by default. Because REST style architecture is involved, each Create, Read, Update, or Delete (CRUD) action is associated with a different type of HTTP request (`POST`, `GET`, `PUT`, `DELETE`). The first time the model object is saved, a `POST` request is used and an identifier ID is created. For subsequent attempts to send the object to the server, a `PUT` request is used.

When a model needs to be retrieved from the server, a Read action is requested and an Ajax `GET` request is used. This type of request uses the `fetch()` method. To determine the location on the server where the model data is pushed or pulled from:

- If the model belongs to a `collection`, the `url` attribute from a collection object will be the base of the location and the model ID (not the cid) will be appended to complete the full URL
- If the model is not inside a collection, the `urlroot` attribute of the model is used as the base of the location

Listing 10 shows how to fetch a model.

Listing 10. Fetch() method for a model object

```javascript
var teamNew = new App.Models.Team({
    urlRoot : '/specialTeams'
});
teamNew.save(); // returns model's ID equal to '222'
teamNew.fetch(); // Ajax request to '/specialTeams/222'
```

The `validate()` method can be used to validate a model, as in Listing 11. The `validate()` method, which is triggered when the `set()` method is called, needs to be overridden to contain the validation logic for the model. The only parameter passed to this function is a JavaScript object containing the attributes updated by the `set()` method so conditions on those attributes can be verified. If nothing is returned from the `validate()` method, the validation is successful. If an error message is returned, the validation fails and the `set()` method will not execute.

Listing 11. Validate method for a model

```javascript
App.Models.Team = Backbone.Model.extend({
    validate : function(attributes){
        if (!!attributes && attributes.name === "teamX") {
            // Error message returned if the value of the "name"
            // attribute is equal to "teamX"
            return "Error!";
        }
    }
});
```

Sets of models are grouped into collections that extend the function `Backbone.Collection`. Collections are characterized by a model attribute defining the type of models composing the
collection. Add and remove models to a collection using the add() / remove() method. **Listing 12** shows how to create and populate a collection.

**Listing 12. Backbone collection**

```
App.Collections.Teams = Backbone.Collection.extend({
    model : App.Models.Team
});
var teams = new App.Collections.Teams();
// Add e model to the collection object "teams"
teams.add(team1);
teams.add(new App.Models.Team({
    name : "Team B"
}));
teams.add(new App.Models.Team());
teams.remove(team1);
console.log(teams.length) // prints 2
```

The `teams` collection that is created contains an array of two models stored in the models attribute. In a typical Ajax application, though, the collection will be populated dynamically (not manually) from the server. The fetch() method helps accomplish the task, as in **Listing 13**, and stores data into the array of models.

**Listing 13. Fetch() method**

```
teams.fetch();
```

Collections in Backbone have a `url` attribute that defines the location on the server from which JSON data is pulled with an Ajax GET request, as in **Listing 14**.

**Listing 14. url attribute and fetch() method for a collection**

```
teams.url = '/getTeams';
teams.fetch(); // Ajax GET Request to '/getTeams'
```

The `fetch()` method is an asynchronous call, so the application does not hang while waiting for the response from the server. In some cases, to manipulate the raw data returned from the server, the parse() method of a collection can be used, as in **Listing 15**.

**Listing 15. parse() method**

```
App.Collections.Teams = Backbone.Collection.extend({
    model : App.Models.Team,
    parse : function(data) {
        // 'data' contains the raw JSON object
        console.log(data);
    }
});
```

Another interesting method available for collections is `reset()`, which allows several models to be set into a collection. The `reset()` method is very handy for bootstrapping data into a collection, such as on page load, to avoid the user waiting for asynchronous calls to return.
Views and client-side templating

Views in Backbone are not the same as views in a classic MVC approach. A Backbone view extends the `Backbone.View` function and displays data stored in models. A view supplies an HTML element defined by the `el` attribute. This attribute can be made by combining the values of the `tagName`, `className`, and `id` attributes, or by the value of `el` itself. **Listing 16** shows two different views with different ways of composing the `el` attribute.

**Listing 16. Backbone view samples**

```javascript
// In the following view, el value is 'UL.team-element'
App.Views.Teams = Backbone.View.extend({
  el : 'UL.team-list'
});

// In the following view, el value is 'div.team-element'
App.Views.Team = Backbone.View.extend({
  className : '.team-element',
  tagName : 'div'
});
```

If the `el`, `tagName`, `className` and `id` attributes are empty, an empty DIV is assigned to `el` by default.

As mentioned, a view must be associated with a model. The model attributes come in handy, as in **Listing 17**. The `App.View.Team` view is tied with an instance of the `App.Models.Team` model.

**Listing 17. Model attribute in a Backbone view**

```javascript
// In the following view, el value is 'UL.team-element'
App.Views.Team = Backbone.View.extend({
  ...;
  model : new App.Models.Team
});
```

To render data, which is the main purpose of a view, override the `render()` method with the logic for displaying attributes of a model inside the DOM element referred by the `el` attribute. **Listing 18** shows a sample of how the `render` method will update the user interface.

**Listing 18. Render () method**

```javascript
App.Views.Team = Backbone.View.extend({
  className : '.team-element',
  tagName : 'div',
  model : new App.Models.Team,
  render : function() {
    // Render the 'name' attribute of the model associated
    // inside the DOM element referred by 'el'
    $(this.el).html("<span>" + this.model.get("name") + "</span>");
  }
});
```

Backbone also promotes the use of client-side templating, making it unnecessary to embed HTML code inside JavaScript, as in **Listing 18**. (With templating, templates encapsulate functions that are common among views; specify that function only once.) Backbone comes with a template engine...
inside underscore.js (a required library), though it is not necessary to use this template engine. The example in Listing 19 uses the underscore.js HTML template.

**Listing 19. HTML containing the template**

```html
<script id="teamTemplate" type="text/template">
  <%= name %>
</script>
```

Listing 20 shows another sample using the underscore.js HTML template.

**Listing 20. View using _.template() function**

```javascript
App.Views.Team = Backbone.View.extend({
  className : '.team-element',
  tagName : 'div',
  model : new App.Models.Team
  render : function() {
    // Compile the template
    var compiledTemplate = _.template($('#teamTemplate').html());
    // Model attributes loaded into the template. Template is appended to the DOM element referred by the el attribute
    $(this.el).html(compiledTemplate(this.model.toJSON()));
  }
});
```

One of the most useful and interesting features in Backbone is the ability to bind the render() method to the change event of a model, as in Listing 21.

**Listing 21. Render() method bound to the change event of a model**

```javascript
// In the following view, el value is 'div.team-element'
App.Views.Team = Backbone.View.extend({
  model : new App.Models.Team,
  initialize : function() {
    this.model.bind("change", this.render, this);
  }
});
```

The code binds the render() method to the change event of a model. When the model changes, the render() method is triggered automatically—thereby saving many lines of code. Starting with Backbone 0.5.2, the bind() method accepts a third parameter to define the object of the callback function. (In the previous example, the current view will be the object inside the callback render()). Prior to Backbone 0.5.2, it was necessary to leverage the bindAll function from underscore.js, as in Listing 22.

**Listing 22. _.bindAll() usage**

```javascript
// In the following view, el value is 'div.team-element'
App.Views.Team = Backbone.View.extend({
  initialize : function() {
    _.bindAll(this, "render");
    this.model.bind("change", this.render);
  }
});
```
In a Backbone view, it is easy to listen to the events thrown by DOM elements inside the view. The `events` attribute becomes very handy for achieving this, as in Listing 23.

**Listing 23. Events attribute**

```javascript
App.Views.Team = Backbone.View.extend({
  className : '.team-element',
  tagName : 'div',
  events : {
    "click a.more" : "moreInfo"
  },
  moreInfo : function(e){
    // Logic here
  }
});
```

Each item in the `events` attributes has two parts:

- The left part indicates the event type and the selector triggering the event.
- The right part defines the event handler function.

In Listing 23, when the user clicks on the links with class `more` inside the DIV with class `team-element`, the function `moreInfo` is called.

**Conclusion**

MVC patterns can provide the organized code that large JavaScript applications need. Backbone is a JavaScript MVC framework that is lightweight and has a small learning curve. Models, views, collections, and routers divide an application in different layers and take care of several specific matters. Backbone can be the right solution when dealing with Ajax applications or SPI applications.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>IBM_Backbone.zip</td>
<td>45KB</td>
</tr>
</tbody>
</table>
Related topics

- **Getting Started with Backbone.js**: Get an introduction to Backbone.js with this short tutorial.
- **Test-Driving Backbone Views With JQuery Templates, The Jasmine Gem, and Jasmine-JQuery** (Derick Bailey, September 2011): Find out more about testing your Backbone views.
- **SPI Manifesto**: Learn more about Single Page Interface.
- **Model-view-controller**: Get an overview of the MVC software architecture from Wikipedia.
- **W3C DOM Level 2 Core Specification**: Read the Document Object Model Core specification.
- **Using the Document Object Model from JavaScript**: Read more about using the JavaScript DOM.
- **XML area on developerWorks**: Find resources for advancing skills in the XML arena, including DTDs, schemas, and XSLT. See the XML technical library for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **developerWorks on Twitter**: Join today to follow developerWorks tweets.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)