BackboneJS 教程

作者: since1027 http://since1027.iteye.com

一套完整的BackboneJS教程

目 录

1.1 Part 1: Introduction to Backbone.Js

发表时间: 2016-06-21 关键字: backbone, javascript, mvc

It was a long time ago (almost a decade back) when most software applications were getting built as standalone applications. These applications were targeted at a single user and ran on their operating systems. Then came the need to share data across multiple users and a need to store data at a central location. This need gave birth to distributed applications and web applications. Distributed applications ran as standalone applications on the user’ s machine giving him a rich user interface (desktop like experience) to work with, and behind the scenes, these applications sent data to a server. Web applications, on the contrary, were sandboxed in a web browser and HTML and HTTP were used to let the user perform operations on the data which is stored on the remote server.

Link to complete series:

The major differentiating factor for these two applications was that the distributed applications provided an interactive user experience whereas web applications provided very limited features (due to technology limitations). The downside of distributed applications was that it was very difficult to roll-out and ensure the application updated across all users. Web applications had no such problems because once the application is updated on the server, all users got the updated applications.

Both these approaches had pros and cons and something needed to be done to get the best of both worlds. This is where browser plugin based applications like Flash applications and Silverlight applications came into picture. These technologies filled in the gap for all the functionality not possible with HTML. They provided the possibility of creating rich internet applications that ran in the browser. The only downside of this was that the user needed to install the browser plug-in to get these applications to work.

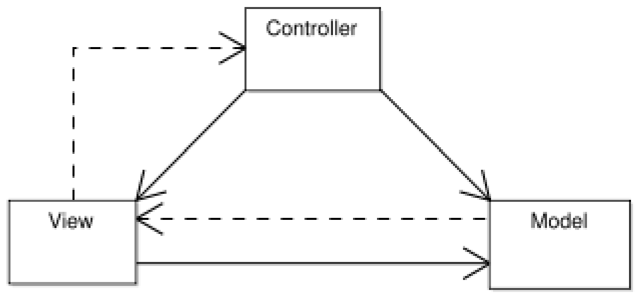
Then came the time when browsers became more capable and HTML became more mature. Creating a rich internet application became possible only using browser based client side technologies. This led developers to write client side code using HTML and JavaScript to create rich internet applications. No need for plugins like Flash and Silverlight. But since HTML and JavaScript were never meant to be used for writing full fledged web applications, these applications had all the HTML and JavaScript code intermingled. This led to spaghetti code and these client side HTML/JavaScript applications (Single Page Applications or SPAs) became a maintenance nightmare.

So why should we write single page apps when they lead to bad code? The main reason for wanting to create a single page application is that they allow us to create a more native-like/desktop-like/ device-like application experience to the user. So the need was to create SPAs in a structured manner and this created a need for JavaScript frameworks and libraries that provided some structure to single page applications.

Currently there are quite a few Open Source JavaScript frameworks available that help us solve the problem of spaghetti code. These frameworks let us design our applications in a structured manner.

Separation of Concerns and MVC

One of the best things about a good application architecture is the Separation of Concerns (SoC). The best part of the Backbone Marionette framework is the ability to provide this separation of concerns using the MVC pattern. The Model will represent the business objects needed to implement the solution. The view is the part that is visible to the user. The user can simply consume the data using views or act upon the data, i.e., CRUD operations. The controller is the one that provides the mechanism for interaction between models and views. The controller’ s responsibility is to react on the user’ s input and orchestrate the application by updating the models and views. The Models and Views remain loosely coupled, i.e., the Models don’ t know anything about the View and the View has a Model object (association) to extract information and display it to the user.



What is BackBone.js

Backbone.js is a lightweight framework that lets us create single page applications in a structured manner. It is based on the Model-View-Controller (MV*) pattern. It is best suited for creating single page applications using a RESTful service for persisting data.

Now one might wonder what is this MV*. we have talked about the MVC architecture at length and then we are saying that backbone is a MV* framework. Well the reason for this is that backbone does not want to enforce the use of controllers. Applications can choose to hook their own code to be used as controllers, use some plugin that can provide a controller or perhaps use MVVM pattern instead of MVC pattern.

What will Backbone Provide

Model: Every application need some way of organizing their data structures. Backbone will provide us with the possibility of creating Models to manage all our entities. Backbone models by default provide the ways to persist themselves. The persistence can be on a localStorage or even on a server via a restful API. These models also provide ways to validate the data in the model before persisting it.

Collections: Collections are simply a group of models together. Backbone also provide the possibility of creating the ordered set of models i.e. collections.

Views: The major problem in JavaScript applications is to handle the UI elements on views. listening to their events and changing their values based on the data received. Backbone ease up this problem by providing an abstraction over the HTML elements i.e. views. Backbone view are like observers on some HTML. We can define the HTML and associate with a view. The view will then take care of handing the events of these HTML elements and populating and rendering these views based on data. The HTML is totally separate from the view object. It can be associated with the view object either directly or via some templating engine(hang on tight, we will get to see all these in action in due course of time).

Routers: Even though we want to create a single page web application, requirements might dictate the need of copying, bookmarking the URLs. So for a single page application, if we want the URL to determine the view/views to be rendered, routers are very helpful. routers will look at the requested URL and then execute code based on the requested URL and then render the view to the user.

Why this Article series

So the idea behind writing this tutorial series is to understand backbone.js framework in a step by step manner by looking at small chunks of features. This article is mainly to get the user acquainted with the backbone.js framework features. from next article, we will dig deeper start looking at the backbone framework in detail. Later we will create a complete application to understand all these concepts.

In the very later stages we will look at Marionette.js plugin. This plugin makes it much more easier to create backbone applications. Also from the architectural perspective backbone.js + Marionette.js application are even better structured than backbone.js applications. We will then create a complete application to see the backbone marionette in action.

So just sit tight and start enjoying this backbone journey with me.

原文链接：http://rahulrajatsingh.com/2014/07/backbone-tutorial-part-1-introduction-to-backbone-js/

1.2 Part 2: Understanding the basics of Backbone Models

发表时间: 2016-06-21 关键字: backbone, javascript, mvc

When we talk about any MV* pattern, model is undoubtedly the most important part of the architecture/application. Its the model that contains all the application data. Along with keeping the data the model class performs various set of actions on the data. Actions like possibility to validate the data, possibility to persist the data, defining access to various parts of data contained in the model (access control).

Backbone.js models are also the most important building blocks when it comes to building backbone.js applications. It keeps track of application data, perform validations on data and provide a mechanism to persist the data either locally on localstorage or remotely on a server using a web service.

Link to complete series:

- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

Creating a simple backbone.js Model

To create a backbone model, we simply need to extend the backbone model class. Following code snippet shows how this can be done.

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


});
```

Furthermore, if we want to create a model that inherits from our model class then we just need to extend from our model class.

```
var ChildrensBook = Book.extend({

});
```

Instantiating a Model

Backbone models can simply be instantiated by using the new keyword.

```
var book = new Book();
```

Deleting a model

To delete a model, we just need to call the destroy function on the model.

```
book.destroy();
```

Sometimes deleting a model could take some time(depending on the size of the model). In such cases we can define a function that will be called when the model get successfully deleted.

```
book.destroy({
    success: function () {
        alert("The model has been destroyed successfully");
    }
});
```

Cloning a model

Often times we would want to have a deep copied object or clone of a model. To create clone of a backbone model we simply need to call the clone method.

```
function cloneModel() {
    var book = new Book();

    var book2 = book.clone();
}
```

How to specify the model attributes

Backbone models does not enforce defining the attributes in the model definition itself i.e. one can create a model and specify the attributes on the fly. Lets say we want to create 2 attributes in our Book model. lets try to create them on the fly.

```
var book = new Book({
    ID: 1,
    BookName: "Sample book"
});
```

Default values of model attributes

Now creating the attributes on the fly is supported by the backbone models and it is a very powerful feature. But this feature actually becomes proves to be a maintenance nightmare when it comes to working with large scale application. From a maintainable application perspective and also from a best practices perspective, I would like the possibility to define my models attributes in my model definition itself.

To accomplish this, the default function can be used. The default function is used to specify the default attributes of the model and their default values. Lets try to move the attributes in the model definition now.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
});
```

This way just instantiating the model will be enough and the created models will have these attributes associated with them.

Setting and getting model attributes

Once we specify the model attributes, we need to be able to get and set their values too. To do this we can use the get and set functions on the model.

```
var book = new Book();

book.set("ID", 3);
book.set("BookName", "C# in a nutshell");

var bookId = book.get('ID');
var bookName = book.get('BookName');
```

How to check attribute existence

Since backbone allows us to add attributes on the fly, we need some way to identify whether a particular attribute exist in the model or not. To do this we can use the has function on model.

```
book.has('ID');     // true
book.has('author');  // false
```

Defining Functions in a Model

We can also define our functions in the model classes. Lets try to create a simple function in our model class.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },

    showAlert: function () {
        alert('ID: ' + this.get('ID') + ', BookName: ' + this.get('BookName'));
    }
});
```

The initialize function

Whenever we create a model, the backbone will call its initialize function. We can override this function to provide custom behavior to it.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },

    initialize: function(){
        console.log('Book has been intialized');
    },

    showAlert: function () {
        alert('ID: ' + this.get('ID') + ', BookName: ' + this.get('BookName'));
    }
});
```

Listening Model attribute changes

We can also use the events to listen to the model changes. This can be done by listening to the change event. backbone raises a change event whenever any model attribute is changed. For each attribute we can use hasChanged method to check if that attribute has been changed or not. Lets try to hook up the event handler to listen to the model change in our current model.

[size=large]

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },

    initialize: function(){
        console.log('Book has been intialized');

        // Lets hook up some event handers to listen to model change
        this.on('change',  function() {
            if(this.hasChanged('ID')){
                console.log('ID has been changed');
            }
            if(this.hasChanged('BookName')){
                console.log('BookName has been changed');
            }
        });
    },

    showAlert: function () {
        alert('ID: ' + this.get('ID') + ', BookName: ' + this.get('BookName'));
    }
});
```

[/size]

If we have a lot of attributes and we are interested in listening to change for any specific attribute then perhaps we can specify that too in the change event binding. Lets try to listen to the BookName change only.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },

    initialize: function () {
        console.log('Book has been intialized');

        // Lets hook up some event handers to listen to model change
        this.on('change:BookName', function () {
            console.log('Message from specific listener: BookName has been changed');
        });
    },

    showAlert: function () {
        alert('ID: ' + this.get('ID') + ', BookName: ' + this.get('BookName'));
    }
});
```

Point of Interest

So that is for this blog. the idea behind this article was to get familiar with the basic concepts of the backbone model. In next article of this series, we will look at more advanced topics associated with the backbone model. This article has been written from beginner’ s perspective, I hope this has been informative.

原文链接：http://rahulrajatsingh.com/2014/07/backbone-tutorial-part-2-understanding-the-basics-of-backbone-models/
附件下载:

- backboneModelsSampleApp.zip (127.3 KB)
- dl.iteye.com/topics/download/b78c93cb-eb6b-36a2-b9e1-bcb4d9b208aa

1.3 Part 3: More about Backbone Models

发表时间: 2016-06-21 关键字: backnone, javascript, mvc

In this article we will look at some more concepts related to backbone models. We will try to see how we can override the default model behavior. We will look at the signification of model IDs, how we can validate a model and finally how a model can be persisted either locally or on a server.

Link to complete series:

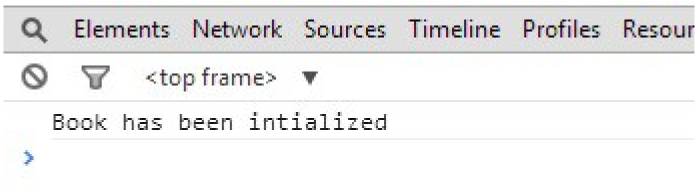
- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

The Initialize function and the Constructor

Whenever we create a model, the backbone will call its initialize function. We can override this function to provide custom behavior to it.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    initialize: function () {
        console.log('Book has been intialized');
    },
});
```

So when we create this object the output will be:

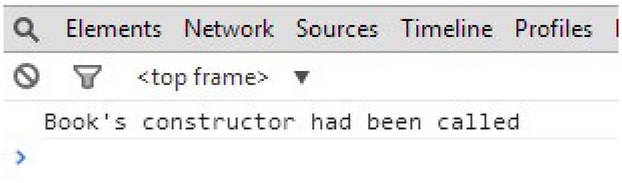


```
Book has been intialized
```

Internally what happens is that whenever a backbone model is created, its constructor gets called. The constructor will call the initialize function. It is also possible to provide out own constructor and provide the custom behavior.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    initialize: function () {
        console.log('Book has been intialized');
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
    },
});
```

The output when creating this model:

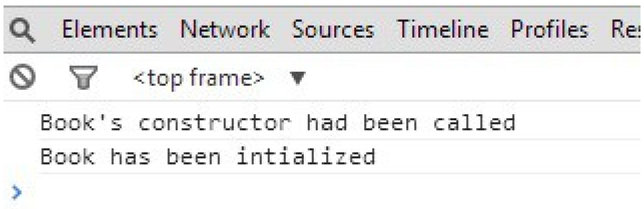


```
Book's constructor had been called
```

Now the problem with this constructor is that whenever the backbone model is getting created our constructor will be called. But the default constructor also does a lot of other activities at the time of object construction like calling the initialize function. So to make sure that our custom constructor works in unison with all that default behavior we need to let the backbone framework know that we still wants that default behavior. This can be done by calling Backbone.Model.apply(this, arguments); at the end of our custom constructor. This will make sure that our custom constructor will be called and then all the other activities that the default constructor is supposed to do are also done.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    initialize: function () {
        console.log('Book has been intialized');
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
});
```

Now the output will be:



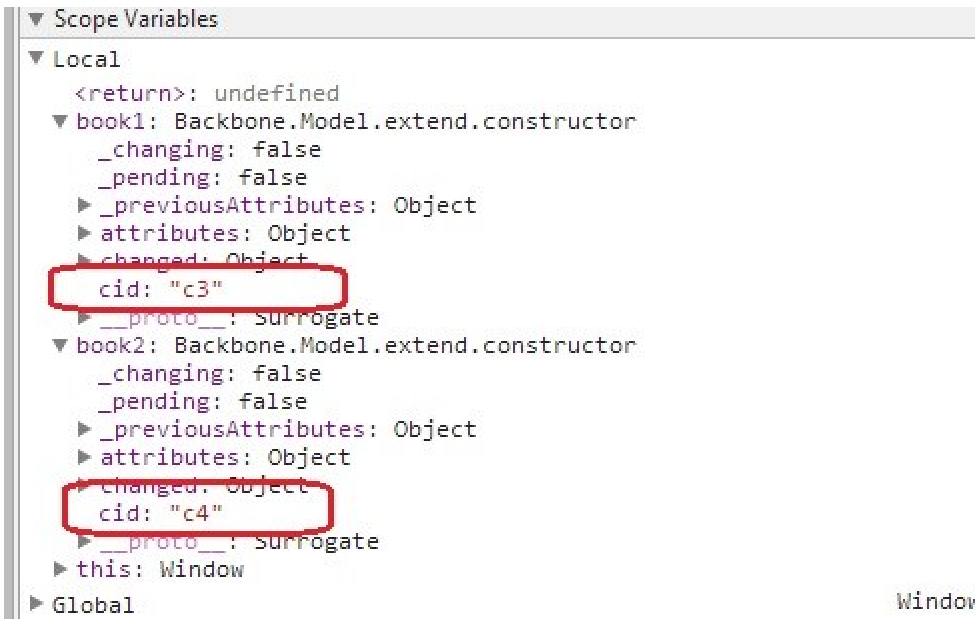
```
Book's constructor had been called
Book has been intialized
```

Note: For most of practical purposes overriding the initialize function will suffice. There is seldom a need to override the constructor but in case one decide to override the constructor, this should be the way to do it.

Model identifiers – id, cid and idAttribute

Every model needs to be uniquely identified. For this backbone gives us the model identifiers. The first one to look at is the cid. The cid or the client id is the auto-generated by backbone so that every model can be uniquely identified on the client.

```
var book1 = new Book();
var book2 = new Book();
```



```
▼ Scope Variables
▼ Local
    <return>: undefined
  ▼ book1: Backbone.Model.extend.constructor
      _changing: false
      _pending: false
    ▶ _previousAttributes: Object
    ▶ attributes: Object
    ▶ changed: Object
      cid: "c3"
    ▶ __proto__: Surrogate
  ▼ book2: Backbone.Model.extend.constructor
      _changing: false
      _pending: false
    ▶ _previousAttributes: Object
    ▶ attributes: Object
    ▶ changed: Object
      cid: "c4"
    ▶ __proto__: Surrogate
  ▶ this: Window
▶ Global                                                Window
```

Backbone also provides an identifier id to uniquely identify the model entity. This is the id that will be used to identify the model when the model data is actually being synced with server i.e. getting persisted. the cid is more useful for debugging purpose but the id attribute will determine the uniqueness of the model when it comes to CRUD operations on the model. Its fairly straight forward to set and get the id property.

```
var book2 = new Book();
book2.id = 3;
console.log(book2.id);
```

Output for the above code will be: 3.

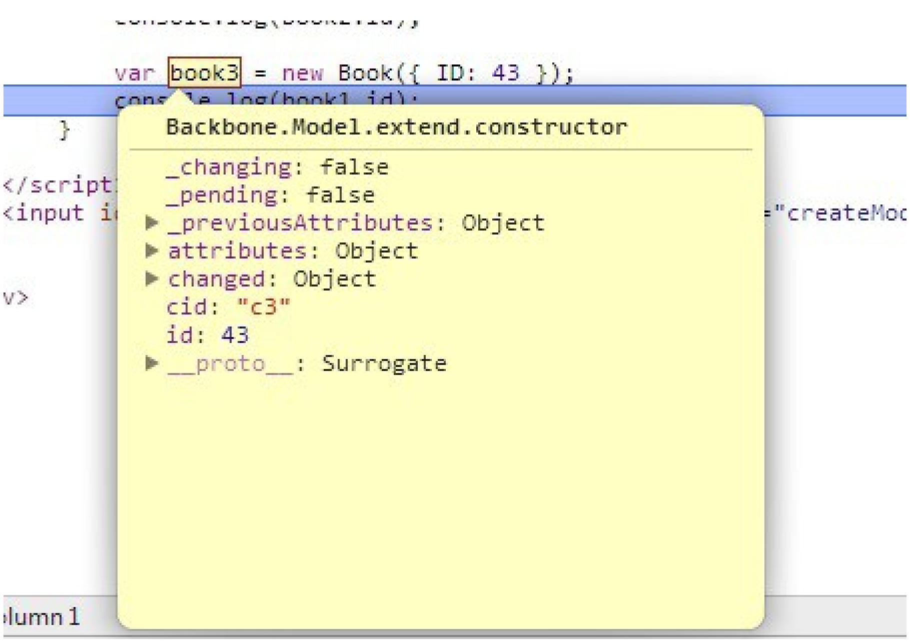
Now it gets a little confusing at this point. Since most of our models will have an attribute that will correspond to the primary key/unique identifier of the entity. Do we need to explicitly set the id value to that attribute. The answer if yes and no. We have to somehow indicate the backbone model what attribute should be used as id but we don’ t have to set the id explicitly. we can use the idAttribute to accomplish this.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",
    initialize: function () {
        console.log('Book has been intialized');
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
});
```

Now in the above code we have specified that the ID should be used as id by specifying the idAttribute. Lets try to create a new model with ID value now.

```
var book3 = new Book({ ID: 43 });
console.log(book1.id);
```

And we can see that the id value is taken from the specified attribute.



```
var book3 = new Book({ ID: 43 });
cons  le.log(book1.id);
}
</script
<input i                                   ="createMod
v>
```

```
Backbone.Model.extend.constructor
  _changing: false
  _pending: false
▶ _previousAttributes: Object
▶ attributes: Object
▶ changed: Object
  cid: "c3"
  id: 43
▶ __proto__: Surrogate
```

and thus this makes it very easier for the backbone models to work with server side entities and makes the model identification seamless.

Validating the model

When we are working on business applications it is often required that we validate the model before persisting the data. Backbone provides a very easy way of validating the model data. We just need to implement the models validate function.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",
    initialize: function () {
        console.log('Book has been intialized');
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
    validate: function (attr) {
        if (attr.ID <= 0) {
            return "Invalid value for ID supplied."
        }
    }
});
```

What happens here is that whenever we try to save the model(which we will see in next section), the Validate function will get called. It will check the custom
validation logic that we have put in place and validate the model. To test the validate method, we can use models isValid function.

```
var book4 = new Book({ ID: -4 });
var result = book4.isValid(); // false
```

Another way to prevent the invalid values in the model attributes is by passing the validate:true while setting the models attribute. This will also trigger the validate function

```
var book5 = new Book();
book5.set("ID", -1, {validate:true});
```

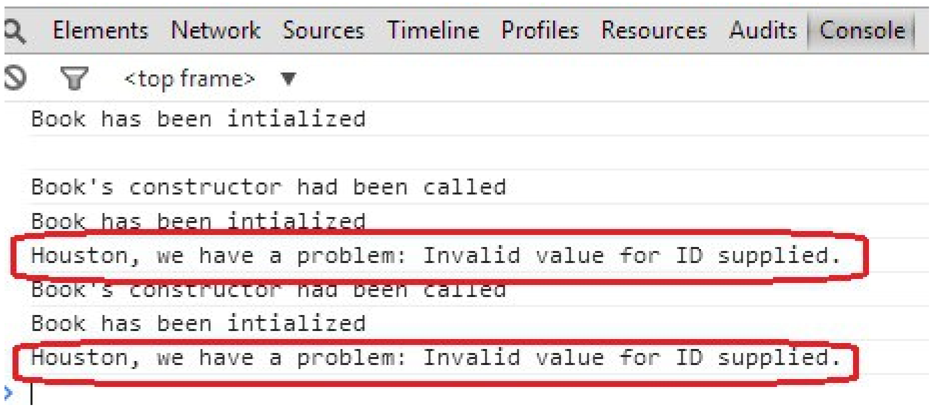
What this will do is that this will not even allow setting of invalid values if the value that we are trying to set is invalid as per our custom logic.

How this validation works is that whenever the user chooses to save the model, the validate function will be called. if there is any validation error then the model save will fail. alternatively, the user can choose to pass validate:true whenever he want to restrict the setting of invalid values in the model attributes.

If we want to check the validity of the model at any particular instance, we can use the isValid function to test this. Having said that one important thing to know here is that whenever our validation function fails to validate the model an event invalid is raised by backbone. If we want to listen to this event, we can subscribe to this.

Lets try to hook up to this event and see the validation errors. We will do this in the initialize function of the model.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",
    initialize: function () {
        console.log('Book has been intialized');
        this.on("invalid", function (model, error) {
            console.log("Houston, we have a problem: " + error)
        });
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
    validate: function (attr) {
        if (attr.ID <= 0) {
            return "Invalid value for ID supplied."
        }
    }
});
```



```
Book has been intialized

Book's constructor had been called
Book has been intialized
Houston, we have a problem: Invalid value for ID supplied.
Book's constructor had been called
Book has been intialized
Houston, we have a problem: Invalid value for ID supplied.
```

Saving the model

The backbone models inherently supports saving on the server using a restful web api. To save the model using a HTTP REST service, we need to specify the urlRoot in the backbone model. To actually save the model, we can call the save on the backbone model. The save method will trigger the validations and if the validations are successful, it will try to identify the action to be performed i.e. create or update and based on that action, it will use urlRoot and call the appropriate REST API to perform the operation.

So if I have a service running on my local machine, i first need to specify the urlRoot for the service in my model.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",
    initialize: function () {
        console.log('Book has been initialized');
        this.on("invalid", function (model, error) {
            console.log("Houston, we have a problem: " + error)
        });
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
    validate: function (attr) {
        if (attr.ID <= 0) {
            return "Invalid value for ID supplied."
        }
    },
    urlRoot: 'http://localhost:51377/api/Books'
});
```

and to save this model using this service, I could do something like:

```
var book = new Book({ BookName: "Backbone Book 43" });
    book.save({}, {
        success: function (model, response, options) {
            console.log("The model has been saved to the server");
        },
        error: function (model, xhr, options) {
            console.log("Something went wrong while saving the model");
        }
    });
```

The save function also accepts success and error callback functions so that appropriate action can be taken based on the response from the server.

Now if we want to save the model on local storage rather than on a server, we just need to keep in mind that save function actually calls sync function to actually save/retrieve the model information. So if we need to save the model on a local storage, we need to override the sync function and provide the custom code to save on local storage.

Note: The code shown above(for save model) is syntactically correct but it will not work unless we have a REST service running at mentioned urlRoot. In coming articles, I will explain the model save using the REST service in details along with the sample service and HTML code.

Point of interest

So we saw a few more details about the backbone models. We have not yet looked at how to save the model either locally or via using a service. Perhaps in my next articles, we will talk about that only.

原文链接：http://rahulrajatsingh.com/2014/07/backbone-tutorial-part-3-more-about-backbone-models/
附件下载:

- backboneModelsSample2.zip (127.1 KB)
- dl.iteye.com/topics/download/826ddad4-d04e-3493-8b73-75feea9bf09c

1.4 Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service

发表时间: 2016-06-21 关键字: backnone, javascript, mvc

In this article we will discuss how we can perform CRUD operations on a backbone model using a REST based HTTP service.

Background

Earlier we have discussed about the benefits of using backbone.js and we also looked at the backbone models.

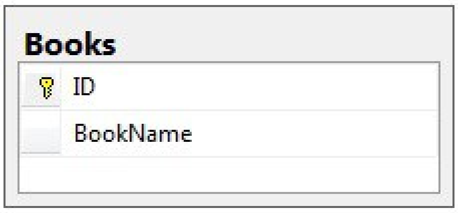
In this article we will look at performing the CRUD operations on backbone models using a REST based web service.

Link to complete series:

- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

Using the code

The first thing we will do is that we will create a simple REST based web api that can be used to save the data on the server using our simple backbone application. For this I have created a simple database with a single table as:



The ID field is configured to auto increment and this is the primary key of the table. so while creating a new model we don’ t have to provide this to the server. Now on top of this model, I have written a simple ASP.NET web api that will provide us the RESTful api. This API is configured to run on my local machine at: http://localhost:51377/. The API details are as follows:

- **Create:** POST http://localhost:51377/api/values
- **Read:** GET http://localhost:51377/api/values/{id}
- **Update:** PUT http://localhost:51377/api/values/{id}
- **Delete:** DELETE http://localhost:51377/api/values/{id}

Once we have the API running, we can start working on our backbone model. We had create the backbone model in our previous article as:

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",
    initialize: function () {
        console.log('Book has been initialized');
        this.on("invalid", function (model, error) {
            console.log("Houston, we have a problem: " + error)
        });
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
    validate: function (attr) {
        if (!attr.BookName) {
            return "Invalid BookName supplied."
        }
    }
});
```

The backbone models inherently supports saving on the server using a restful web api. To save the model using a HTTP REST service, we need to specify the urlRoot in the backbone model. To actually save the model, we can call the save on the backbone model.The save method will trigger the validations and if the validations are successful, it will try to identify the action to be performed i.e. create or update and based on that action, it will use urlRoot and call the appropriate REST API to perform the operation. Let us specify the URL root to enable this model to use our web api service.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",
    initialize: function () {
        console.log('Book has been initialized');
        this.on("invalid", function (model, error) {
            console.log("Houston, we have a problem: " + error)
        });
    },
    constructor: function (attributes, options) {
        console.log('Book\'s constructor had been called');
        Backbone.Model.apply(this, arguments);
    },
    validate: function (attr) {
        if (!attr.BookName) {
            return "Invalid BookName supplied."
        }
    },
    urlRoot: 'http://localhost:51377/api/Books'
});
```

Now let us try to perform CRUD operations on this model.

Create

To create a new entity on the server, we need to populate the non identity fields in the model (other than ID in this case) and then call the Save method on the model.

```
// Lets perform a create operation [CREATE]
var book = new Book({ BookName: "Backbone Book 43" });
book.save({}, {
    success: function (model, respose, options) {
        console.log("The model has been saved to the server");
    },
    error: function (model, xhr, options) {
        console.log("Something went wrong while saving the model");
    }
});
```

Read

To read a single book entity, we need to create the book entity with the identity attribute populated, i.e., the ID of the book we want to read. Then we need to call the fetch method on the model object.

```
// Now let us try to retrieve a book [READ]
var book1 = new Book({ ID: 40 });
book1.fetch({
    success: function (bookResponse) {
        console.log("Found the book: " + bookResponse.get("BookName"));
    }
});
```

Update

Now let’ s say we want to update the name of the book retrieved in the earlier fetch call. All we need to do is set the attributes we need to update and call the save method again.

```
// Lets try to update a book [UPDATE]
var book1 = new Book({ ID: 40 });
book1.fetch({
    success: function (bookResponse) {
        console.log("Found the book: " + bookResponse.get("BookName"));
        // Let us update this retreived book now (doing it in the callback) [UPDATE]
        bookResponse.set("BookName", bookResponse.get("BookName") + "_updated");
        bookResponse.save({}, {
            success: function (model, respose, options) {
                console.log("The model has been updated to the server");
            },
            error: function (model, xhr, options) {
                console.log("Something went wrong while updating the model");
            }
        });
    }
});
```

Delete

Now to delete a Model, we just need to call the destroy method of the model object.

```
// Let us delete the model with id 13 [DELETE]
var book2 = new Book({ ID: 40 });
book2.destroy({
    success: function (model, respose, options) {
        console.log("The model has deleted the server");
    },
    error: function (model, xhr, options) {
        console.log("Something went wrong while deleting the model");
    }
});
```

Custom URLs to perform CRUD operation on models

There are few scenarios where we might want to have provide custom URLs for the individual operations. This can be achieved by overriding the sync function and providing custom URL for each action. Let us create one more model BookEx to see how this can be done.

```
var BookEx = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",

    // Lets create function which will return the custom URL based on the method type
    getCustomUrl: function (method) {
        switch (method) {
            case 'read':
                return 'http://localhost:51377/api/Books/' + this.id;
                break;
            case 'create':
                return 'http://localhost:51377/api/Books';
                break;
            case 'update':
                return 'http://localhost:51377/api/Books/' + this.id;
                break;
            case 'delete':
                return 'http://localhost:51377/api/Books/' + this.id;
                break;
        }
    },
    // Now lets override the sync function to use our custom URLs
    sync: function (method, model, options) {
        options || (options = {});
        options.url = this.getCustomUrl(method.toLowerCase());

        // Lets notify backbone to use our URLs and do follow default course
        return Backbone.sync.apply(this, arguments);
    }
});
```

Now we can perform the CRUD operations on this model in the same way as we did for the previous model.

Point of interest

In this article we have looked at how to perform CRUD operations on backbone models using HTTP based REST service. This has been written from a beginner’ s perspective. I hope this has been informative.

原文链接：http://rahulrajatsingh.com/2014/07/backbone-tutorial-part-4-crud-operations-on-backbonejs-models-using-http-rest-service/
附件下载:

- backboneSample.zip (127.7 KB)
- dl.iteye.com/topics/download/4c24f274-afd4-3694-8e4c-d92f15a17eb4

- WebAPISample.zip (3.1 MB)
- dl.iteye.com/topics/download/2a464f76-cfa6-3050-820d-d6b391431387

1.5 Part 5: Understanding Backbone.js Collections

发表时间: 2016-06-21 关键字: backbone, javascript, mvc

In this article we will discuss about Backbone.js collections. We will see how we can use collections to manipulate a group of models and how we can use restul API to easily fetch and save collections.

Background

Every application need to create a collection of models which can be ordered, iterated and perhaps sorted and searched when a need arises. Keeping this in mind, backbone also comes with a collection type which makes dealing with collection of models fairly easy and straight forward.

Link to complete series:

- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

Using the code

Let us start looking at the backbone collections in details.

Creating a collection

Creating a backbone collection is similar to creating a model. We just need to extend the backbone's collection class to create our own collection. Let us continue working with the same example where we created a Book model and lets try to create a simple BooksCollection.

```
var BooksCollection = Backbone.Collection.extend({
});
```

This collection will hold the Book model we have created in our previous articles.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    },
    idAttribute: "ID",

    urlRoot: 'http://localhost:51377/api/Books'
});
```

Specifying the model for a collection

To specify which model this collection should hold, we need to specify/override the model property of the collection class.

```
var BooksCollection = Backbone.Collection.extend({
    model: Book,
});
```

Once we specify the model property of a collection what will happen internally is that whenever we create this collection, internally it will create an array of the specified models. Then all the operations on this collection object will result in the actual operations on that array.

Instantiating a collection

A collection can be instantiated by using the new keyword. We can create an empty collection and then add the model objects to it later or we can pass a few model objects in the collection while creating it.

```
// Lets create an empty collection
var collection1 = new BooksCollection();
//Lets create a pre-populated collection
var book1 = new Book({ ID: 1,  BookName: "Book 1" });
var book2 = new Book({ ID: 2, BookName: "Book 2" });
var collection2 = new BooksCollection([book1, book2]);
```

Adding models to collection

To add an item to a collection, we can use the add method on the collection. The important thing to notice here is that if the item with the same id exist in the collection, the add will simply be ignored.

```
    var book3 = new Book({ ID: 3, BookName: "Book 3" });
    collection2.add(book3);
```

Now there might be a scenario where we actually want to update an existing added model in a collection. If that is the case, then we need to pass the {merge: true} option in the add function.

```
var book3 = new Book({ ID: 3, BookName: "Book 3" });
collection2.add(book3);
var book3_changed = new Book({ ID: 3, BookName: "Changed Model" });
collection2.add(book3_changed, { merge: true });
```

Another important point to consider here is that the collection keep a shallow copy of the actual models. So if we change a model attribute after adding it to a collection, the attribute value will also get changed inside the collection.

Also, if we want to add multiple models, we can do that by passing the model array in the add method.

```
var book4 = new Book({ ID: 4, BookName: "Book 4" });
var book5 = new Book({ ID: 5, BookName: "Book 5" });
collection2.add([book4, book5]);
```

It is also possible to add the model at a specific index in the collection. To do this we need to pass the {at: location} in the add options.

```
var book0 = new Book({ ID: 0, BookName: "Book 0" });
collection2.add(book0, {at:0});
```

Note: push and unshift function can also be used to add models to collection.

Removing models from collection

To remove the model from the collection, we just need to call the remove method on the collection. The remove method simply removes this model from the collection.

```
    collection2.remove(book0);
```

Also, if we want to empty the model, we can call the reset method on the collection.

```
  collection1.reset();
```

It is also possible to reset a collection and populate it with new models by passing an array of models in the reset function.

```
    collection2.reset([book4, book5]); // this will reset the collection and add book4 and book
```

Note: pop and shift function can also be used to remove model from collection.

Finding the number of items in collection

The total number of items in a collection can be found using the length property.

```
    var collection2 = new BooksCollection([book1, book2]);
    console.log(collection2.length); // prints 2
```

Retrieving models from collection

To retrieve a model from a specific location, we can use the at function by passing a 0 based index.

```
    var bookRecieved = collection2.at(3);
```

Alternatively, to get the index of a known model in the collection, we can use the indexOf method.

```
    var index = collection2.indexOf(bookRecieved);
```

We can also retreive a model from a collection if we know its id or cid. this can be done by using the get function.

```
    var bookFetchedbyId = collection2.get(2); // get the book with ID=2
    var bookFetchedbyCid = collection2.get("c3"); // get the book with cid=c3
```

If we want to iterate through all the models in a collection, we can simply use the classic for loop or the each function provided by collections which is very similar to the foreach loop of underscore.js.

```
for (var i = 0; i < collection2.length; ++i) {
    console.log(collection2.at(i).get("BookName"));
}
```

```
collection2.each(function (item, index, all) {
    console.log(item.get("BookName"));
});
```

Listening to collection events

Backbone collection raises events whenever an item is added removed to updated in the collection. We can subscribe to these events by listening to add, remove and change event respectively. Let us subscribe to these events in our model to see how this can be done.

```
var BooksCollection = Backbone.Collection.extend({
    model: Book,
    initialize: function () {

        // This will be called when an item is added. pushed or unshifted
        this.on('add', function(model) {
            console.log('something got added');
        });
        // This will be called when an item is removed, popped or shifted
        this.on('remove',  function(model) {
            console.log('something got removed');
        });
        // This will be called when an item is updated
        this.on('change', function(model) {
            console.log('something got changed');
        });
    },
});
```

The set function

The set function can be used to update all the items in a model. If we use set function, it will check for all the existing models and the models being passed in set. If any new model is found in the models being passed, it will be added. If some are not present in the new models list, they will be removed. If there are same models, they will be updated.

```
var collection3 = new BooksCollection();
collection3.add(book1);
collection3.add(book2);
collection3.add(book3);
collection3.set([book1, { ID: 3, BookName: "test sort"}, book5]);
```

The above shown set function will call remove for book2, change for book3 and add for book5.

Sorting a collection

Backbone keeps all the models in the collection in a sorted order. We can call the sort function to forcefully sort it again but the models are always stored in sorted order. By default these items are sorted in the order they are added to the collection. But we can customize this sorting behavior by providing a simple comparator to our collection.

```
var BooksCollection = Backbone.Collection.extend({
    model: Book,

    comparator: function (model) {
        return model.get("ID");
    },
});
```

What this comparator does is that it overrides the default sorting behavior by specifying the attribute that should be used for sorting. We can even used a custom expression in this comparator too.

Fetch collection using HTTP REST service

To be able to fetch the collection from the server, we need to specify the url for the api that returns the collection.

```
var BooksCollection = Backbone.Collection.extend({
    model: Book,

    url: "http://localhost:51377/api/Books",
});
```

Now to fetch the collection from the server, lets call the fetch function.

```
var collection4 = new BooksCollection();
collection4.fetch();
```

Save collection using HTTP REST service

Lets see how we can save the items of a collection on the server.

```
var collection4 = new BooksCollection();
collection4.fetch({
    success: function (collection4, response) {
        // fetch successful, lets iterate and update the values here
        collection4.each(function (item, index, all) {
            item.set("BookName", item.get("BookName") + "_updated"); // lets update all book na
            item.save();
        });
    }
});
```

In the above code we are calling save on each model object. this can be improved by either overriding the sync function on a collection or perhaps creating a wrapper model for collection and saving the data using that.

Note: The web api code for can be downloaded from the previous article of the series.

Point of interest

In this article we have discusses about the backbone collections. This has been written from a beginner’ s perspective. I hope this has been informative.

原文链接：http://rahulrajatsingh.com/2014/07/backbone-tutorial-part-5-understanding-backbone-js-collections/
附件下载:

- backboneSample1.zip (127.4 KB)
- dl.iteye.com/topics/download/0a00040e-b2e9-37a3-b53f-0ae7804ce8bc

1.6 Part 6: Understanding Backbone.js Views

发表时间: 2016-06-22 关键字: backbone, javascript, mvc

In this article, we will try to look at the View classes in Backbone.js and see how view classes help us in updating the relevant parts of the application easily.

Background

The biggest problem while writing JavaScript applications is the spaghetti code that one needs to write just for HTML DOM manipulation. Every element on the UI will need some actions to happen when the user interacts with them. Some UI elements would want to automatically update the values based on the new/updated data. Doing all this using plain HTML and JavaScript/jQuery is a big problem (doable but nightmare) specially from maintenance perspective.

Backbone.js view greatly helps us when it comes to creating large scale manageable applications. The view classes are more like a glue that holds an HTML template with the model object. Also, this provides the mechanism to handle the events raised from the model and update the UI and handle UI events and act on them (perform some operations on the model). So in a way we can say that the views are just observers who are listening to the model and UI events which makes them a perfect place to handle all the events and act upon them. Backbone views can be thought of as:

Observers that keep listening to the DOM events and in case the event fires taking the appropriate actions.
Objects backed by models that are responsible for rendering the model data on the screen.
Let us see how we can use backbone.js views to efficiently manage the applications.

Link to complete series:

- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

Using the code

Creating a simple view

Let is start the discussion by looking at how we can create backbone views. Like backbone models and collections, creating a backbone view is also as easy as extending the existing Viewclass of backbone.

```
var sampleView = Backbone.View.extend({


});
```

Like models and collections, we can also override the initialize and constructor of the backbone views. Lets try to see how we can override the initializefunction.

```
var sampleView = Backbone.View.extend({
    initialize: function() {
        console.log('sampleView has been created');
    }
});
```

Instantiating the view is also straight forward. A view can simply be instantiated using the newkeyword.

```
var view1 = new sampleView();
```

Associating model with a view

Every view will be backed by a model. This modelcan be passed to the view in the constructor.

```
var book1 = new Book({ ID: 1, BookName: "Book 1" });
var m_bookView = new bookView({model: book1});
```

This model can either be a backbone model or a backbone collection. The view can extract the information from its model and render the HTML accordingly.

Understanding the el property

Now we are saying that the views are responsible for listening to DOM element’ s events and also for updating the DOM elements. For this to happen the view class should be associated/attached to a DOM element. Backbone views are always associated to a DOM element. This associated DOM element can be accessed/manipulated using the elproperty.

Now there are 2 ways to create view:

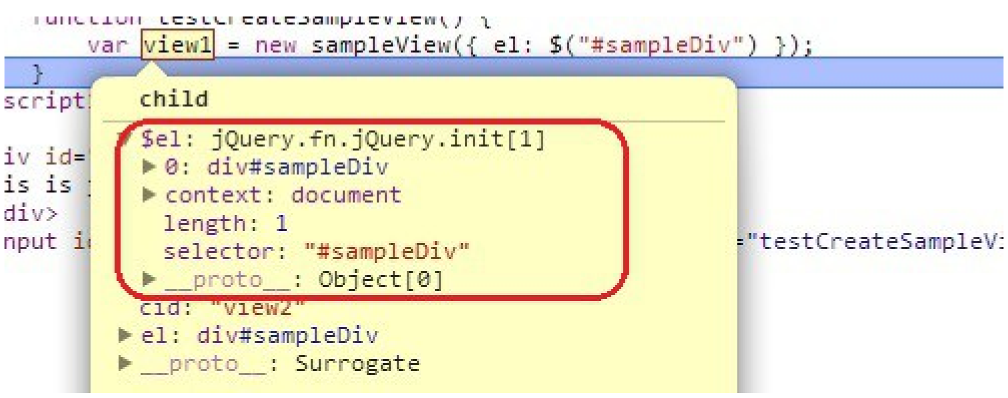
1.Creating a view that will get associated with an existing DOM element.
2.Creating a view that will create its own DOM element on the fly.

So lets start by looking at how we can create a view that will get associated with an existing DOM element. The views constructor is capable of accepting a lot of parameters. It can accept models, collections and even the DOM element that this view should associate itself to.

Lets say we want to create a view for an existing DOM element i.e. a div with id="sampleDiv".

```
var view1 = new sampleView({ el: $("#sampleDiv") });
```

When we run the application and try to watch the elproperty, we can see that the el property contains the div element.

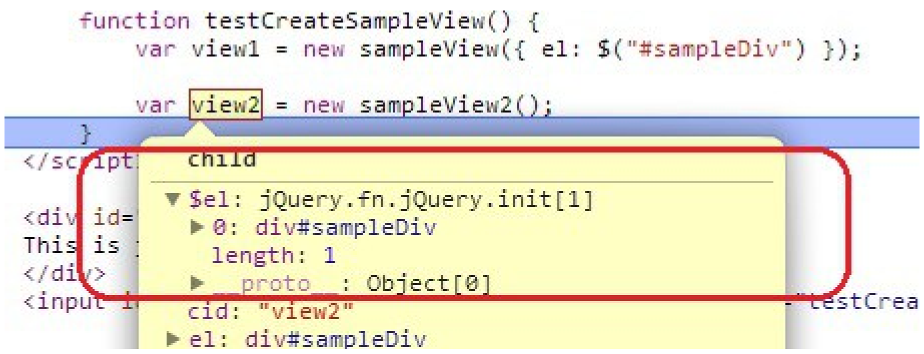


```
    var view1 = new sampleView({ el: $("#sampleDiv") });
  }
script    child
            $el: jQuery.fn.jQuery.init[1]
iv id=       ▶ 0: div#sampleDiv
is is        ▶ context: document
div>          length: 1
nput i        selector: "#sampleDiv"                    ="testCreateSampleV:
             ▶ __proto__: Object[0]
            cid: "view2"
           ▶ el: div#sampleDiv
           ▶ __proto__: Surrogate
```

Now lets see how we can create a view that will create a DOM element for it dynamically. The way it works is that we can specify tagName, className, id and attributes in a backbone view. Based on these values the el will be created by backbone. Lets try to create a simple div with id using this approach.

```
var sampleView2 = Backbone.View.extend({

    tagname: 'div',

    id: 'sampleDiv'

});
```

When we create this view, we can see that the view is associated with a div which was created using our specified tagName and id values.



```
    function testCreateSampleView() {
        var view1 = new sampleView({ el: $("#sampleDiv") });

        var view2 = new sampleView2();
    }
  </script    child
               ▼ $el: jQuery.fn.jQuery.init[1]
  <div id=       ▶ 0: div#sampleDiv
  This is          length: 1
  </div>         ▶ __proto__: Object[0]
  <input i       cid: "view2"                          "testCrea
               ▶ el: div#sampleDiv
```

Now these two approached provides a lot of flexibility while developing backbone applications. Let us try to look at a simple example to understand the complete picture. lets say we need to create a list of books. We know the area where these items should be rendered but we the actual items will be added at runtime. This can easily be achieved by creating an empty list and using JavaScript to add list items at runtime. Lets see how we can use backbone views to achieve this.

First let us create the a simple view that will render the book data as a list element. Do do this we will use the dynamically generated DOM element.

```
var bookView = Backbone.View.extend({
    tagname: "li",
    model: Book,
    render: function (){
        this.$el.html('<li>' + this.model.get("BookName") + '</li>');
        return this;
    }
});
```

What this view is doing is that, it is overiding the render function to render the book name in a list element. We have overridden the renderfunction to render the book as a list element.

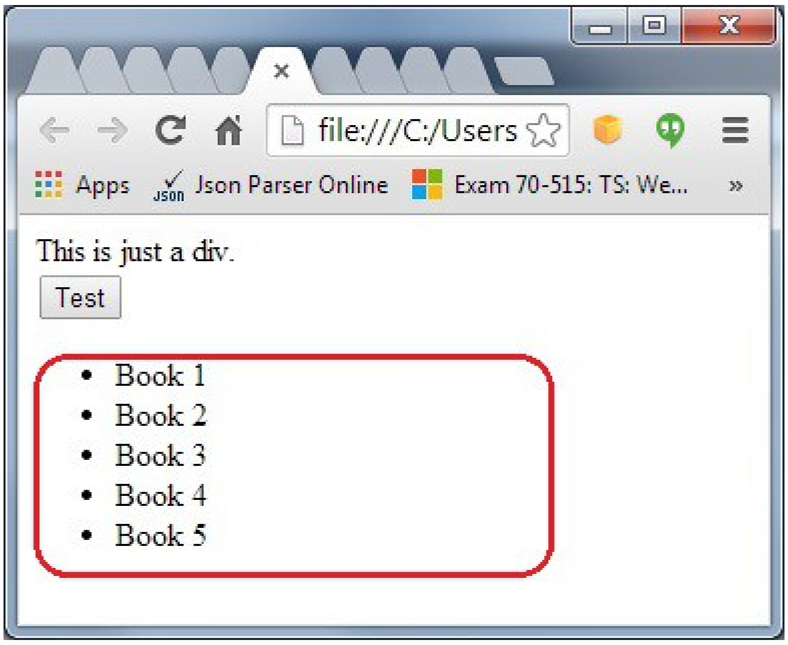
Now we need a view that will contain this list elements i.e. the list view. For this lets create a simple list element on my HTML and then lets use this view class to use that el.

```
var bookListView = Backbone.View.extend({
    model: BooksCollection,

    render: function() {
        this.$el.html(); // lets render this view

        var self = this;

        for(var i = 0; i < this.model.length; ++i) {
            // lets create a book view to render
            var m_bookView = new bookView({model: this.model.at(i)});

            // lets add this book view to this list view
            this.$el.append(m_bookView.$el);
            m_bookView.render(); // lets render the book
        }

         return this;
    },
});
```

What this view is doing is that, it is accepting a collection of books and in the render function it is using the bookViewto render the books inside the associated el. Now the next thing we need to do is to associated the list created on the HTML page with this view as its el and pass the books collection to this view as model.

```
var book1 = new Book({ ID: 1, BookName: "Book 1" });
var book2 = new Book({ ID: 2, BookName: "Book 2" });
var book3 = new Book({ ID: 3, BookName: "Book 3" });
var book4 = new Book({ ID: 4, BookName: "Book 4" });
var book5 = new Book({ ID: 5, BookName: "Book 5" });
var bookCollection = new BooksCollection([book1, book2, book3, book4, book5]);
var bookList = null;

$(document).ready(function () {
    bookList = new bookListView({ el: $("#bookList"), model: bookCollection });
    bookList.render();
});
```

Calling the render function of this view will use our backbone views and render the list of books in an unordered list.



This is just a div.
Test

- Book 1
- Book 2
- Book 3
- Book 4
- Book 5

Note: A view’ s elcan be changed anytime by calling the setElementmethod of the view.

Using templates

Now in our example we have overridden the render function of our views and took charge of rendering the HTML is our own code. This is still better than plain JavaScript/jquery based approach because here our JavaScript code is not intermingled with HTML and there is a logical structure to our views.

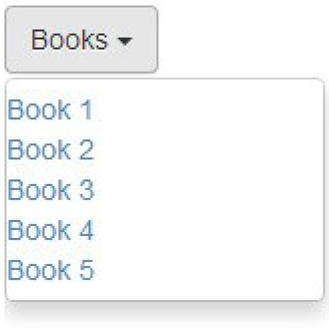
But the problem is that our view HTML could become very complex and it might always not be possible to spit out that HTML from our render functions. To ease this problem backbone supports view templates. Any template engine can be used with backbone view. To understand the concept of templates, let us use the simple JavaScript style templates.

Lets say that every book needs to be rendered as a drop down menu. This can be achived by using bootstrap very easily. But creating all that HTML in the render function might not be a very good idea. So let us create one more set of views that will use the template to render the books in a drop-down.

```
var bookView2 = Backbone.View.extend({


    model: Book,
    tagName: 'li',
    template: '',

    initialize: function() {
        this.template = _.template($('#bookItem').html());
    },

    render: function() {
        this.$el.html(this.template(this.model.attributes));
        return this;
    }
});

var bookListView2 = Backbone.View.extend({
    model: BooksCollection,

    render: function() {
        this.$el.html(); // lets render this view

        for(var i = 0; i < this.model.length; ++i) {
            // lets create a book view to render
            var m_bookView = new bookView2({model: this.model.at(i)});

            // lets add this book view to this list view
            this.$el.append(m_bookView.$el);
            m_bookView.render(); // lets render the book
        }

         return this;
    },
});
```

And the template is defined in the HTML file itself as:

```
 <script type="text/template" id="bookItem">
     <li role="presentation"><a role="menuitem" tabindex="-1" href="#"> <%= BookName %> </a></l
    </script>
```



What will happen here is that the bookView2will use this template to render the books as list elements. Backbone can work on any view engine. Also the example taken here was little contrived but very complex templates can also be created and rendered using this approach very easily.

Listening to DOM events

Now there is one important thing remaining. how can a view object listen to DOM elements and perform needed actions. To understand this let us add a simple button on our list view and try to listen to its click action.

```
var bookView2 = Backbone.View.extend({

    model: Book,
    tagName: 'li',
    template: '',

    events: {
        'click': "itemClicked"
    },

    itemClicked: function () {
        alert('clicked: ' + this.model.get('BookName'));
    },

    initialize: function() {
        this.template = _.template($('#bookItem').html());
    },

    render: function() {
        this.$el.html(this.template(this.model.attributes));
        return this;
    }
});
```

Now whenever an a DOM element raises an event the associated view will look for its handler in the events section. If the handler exists, it calls that handler. this is very useful when we need to listen to DOM events and take some actions. we can use {"event selector": "callback"}format to declare our DOM event handlers. the selector are are usual jquery/css selectors.

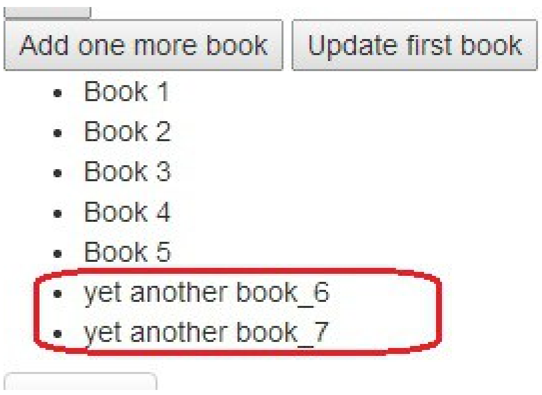
Listening to Model changes

In large scale applications there might be multiple views rendering the same data. what if one view changes the data? should other views continue to show the stale data? Probably no. Thus we also need to listen to the model changes too. this can easily be achieved by listening to model changes as:

```
var bookListView = Backbone.View.extend({
    model: BooksCollection,

    initialize: function() {
        // lets listen to model change and update ourselves
        this.listenTo(this.model, "add", this.modelUpdated);
    },

    modelUpdated: function() {
        this.render();
    },
});
```

What we did here is that whenever new books are added to the collection. The associated view will be listening to the add event. On recieving this event it will simply renders the view again. This can be tested by simply adding few more books in the already rendering collection

```
function AddMoreBooks() {
    var i = bookCollection.length + 1;
    var newBook = new Book({ID: i, BookName: 'yet another book_' + i});

    bookCollection.add(newBook);
}
```



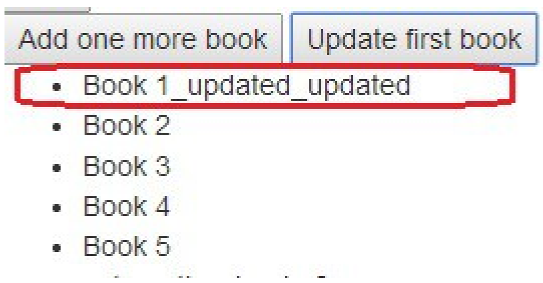
- Book 1
- Book 2
- Book 3
- Book 4
- Book 5
- yet another book_6
- yet another book_7

On same lines, we can also listen to change event to listen to model updates.

```
var bookView = Backbone.View.extend({
    tagName: "li",
    model: Book,

    initialize: function() {
        // lets listen to model change and update ourselves
        this.listenTo(this.model, "change", this.render);
    }
});
```

To test this, lets just try to update a book that is already being rendered on screen.

```
book1.set('BookName', book1.get('BookName') + '_updated');
```



- Book 1_updated_updated
- Book 2
- Book 3
- Book 4
- Book 5

```
Removing a view from DOM
```

Removing a view from DOM can be easily achieved by calling the removefunction on the view.

```
bookList.remove();
```

Point of interest

In this article we looked at the backbone views. We looked at how we can use backbone views to implement better structured applications that can easily perform DOM manipulations.

原文链接：http://rahulrajatsingh.com/2014/07/backbone-tutorial-part-6-understanding-backbone-js-views/
附件下载:

- backboneViewsSample.zip (323.1 KB)
- dl.iteye.com/topics/download/d86a6e9f-8da6-3786-950c-1a8148013bcc

1.7 Part 7: Understanding Backbone.js Routes and History

发表时间: 2016-06-22 关键字: backbone, javascript, mvc

In this article, we will try to look at Routes in Backbone.js. We will try to understand how routes can be useful in a large scale single page applications and how we can use routes to perform action based on requested URL.

Background

We have been using web application for more than 2 decades now. This has made us tuned to some of the functionalities that the websites provide. One such functionality is to be able to copy the URL and use it for the viewing the exact application area that we were viewing before. Another example is the use of browser navigation buttons to navigate back and forth the pages.

When we create single page applications, there is only one page being rendered on the screen. There is no separate URL for each page. The browser is not loading the separate pages for separate screens. So how can we still perform the above mentioned operations even with a single page application. The answer is backbone routes.

Link to complete series:

- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

Using the code

Backbone routes and history provides us the mechanism by which we can copy the URLs and use them to reach the exact view. It also enables us to use browser navigation with single page applications. Actually routes facilitate the possibility of having deep copied URLs and history provides the possibility of using the browser navigation.

Life without Router

Let us try to create a simple application that is not using the router. Lets create three simple views and these views will be rendered in the same area on our application based on user selection. Let create 3 very simple views.

```
var View1 = Backbone.View.extend({

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

    render: function() {
        this.$el.html(this.model.get('Message') + " from the View 1");
        return this;
    }
});

var View2 = Backbone.View.extend({

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

    render: function() {
        this.$el.html(this.model.get('Message') + " from the View 2");
        return this;
    }
});

var View3 = Backbone.View.extend({

    initialize: function() {
        this.render();
    },
    render: function() {
        this.$el.html(this.model.get('Message') + " from the View 3");
        return this;
    }
});
```

Now we need a view that will contain the view and render it whenever the user makes a choice on the screen.

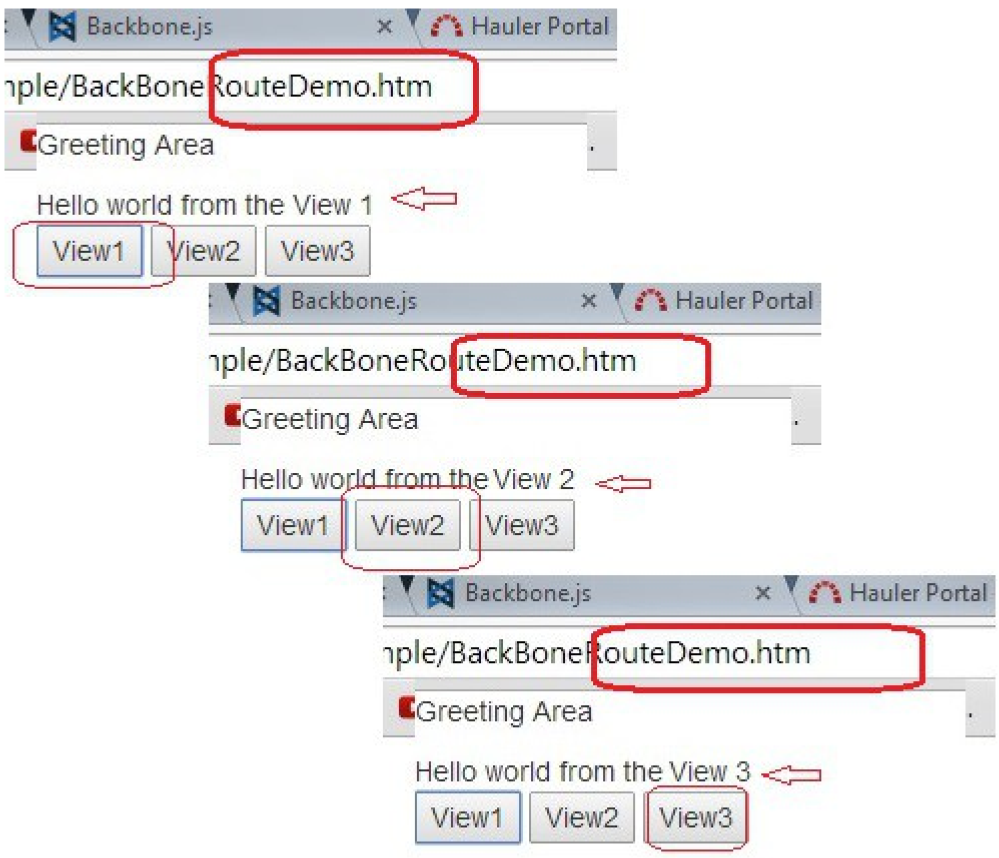
```
var ContainerView = Backbone.View.extend({
     myChildView: null,

     render: function() {
        this.$el.html("Greeting Area");

        this.$el.append(this.myChildView.$el);
        return this;
    }
});
```

Now lets create a simple div on the UI which will be used as elto this ContainerView. We will then position three buttons on the UI which will let the user to change the view. Below code shows the application setup that is creating the container view and the functions that will get invoked when the user selects the view from screen.

```
var greeting = new GreetModel({ Message: "Hello world" });


var container = new ContainerView({ el: $("#AppContainer"), model: greeting });
var view1 = null;
var view2 = null;
var view3 = null;
```

```
function showView1() {
    if (view1 == null) {
        view1 = new View1({ model: greeting });
    }

    container.myChildView = view1;
    container.render();
}

function showView2() {
    if (view2 == null) {
        view2 = new View2({ model: greeting });
    }

    container.myChildView = view2;
    container.render();
}

function showView3() {
    if (view3 == null) {
        view3 = new View3({ model: greeting });
    }

    container.myChildView = view3;
    container.render();
}
```

Now lets run the application and see the results.


Backbone.js
Hauler Portal
nple/BackBoneRouteDemo.htm
Greeting Area
Hello world from the View 1
View1
View2
View3
Hello world from the View 2
Hello world from the View 3

When we click on the buttons we can see that the actual view is getting changes but the URL is not getting changes. That would mean that there is no way, I can copy a URL and directly go to any view. Also, the second thing to note here is that if we press the browser back button, the application will go away(since its still on the same single page from the browser’ s perspective).

Note: Please download and run the sample code to see this in action.

Hello Backbone Routes

Now the above problem can very easily be solved using Backbone routesand History. So lets try to first look at what are backbone routes.

Backbone routes are simple objects that are handle the incoming route value from the URL and the invoke any function. Lets create a very simple route class for our application.

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


});
```

In our route class we will have to define the routes that our application will support and how we want to handle them. So first lets create a simple route where only the URL is present. This usually is the starting page of our application. For our application lets just open view1 whenever nothing is present in the route. Then if the request is for any specific view we will simply invoke the function which will take care of rendering the appropriate view.

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

    greeting: null,
    container: null,
    view1: null,
    view2: null,
    view3: null,

    initialize: function() {
        this.greeting = new GreetModel({ Message: "Hello world" });
        this.container = new ContainerView({ el: $("#rAppContainer"), model: this.greeting });
    },

    routes: {
        "": "handleRoute1",
        "view1": "handleRoute1",
        "view2": "handleRoute2",
        "view3": "handleRoute3"
    },

    handleRoute1: function () {
        if (this.view1 == null) {
            this.view1 = new View1({ model: this.greeting });
        }

        this.container.myChildView = this.view1;
        this.container.render();
    },

    handleRoute2: function () {
        if (this.view2 == null) {
            this.view2 = new View2({ model: this.greeting });
        }

        this.container.myChildView = this.view2;
        this.container.render();
    },

    handleRoute3: function () {
        if (this.view3 == null) {
            this.view3 = new View3({ model: this.greeting });
        }

        this.container.myChildView = this.view3;
        this.container.render();
    }
});
```

Now this route class contains the complete logic of handling the URL requests and rendering the view accordingly. Not only this, we can see that the code which was written in a global scope earlier i.e. the controller and view creation all that is put inside the route now. This would also mean that routes not only provide us deep copyable URLs but also could provide more options to have better structured code(since we can have multiple route classes and each route class can handle all the respective views for the defined routes).

BackBone History and Instantiating Routes

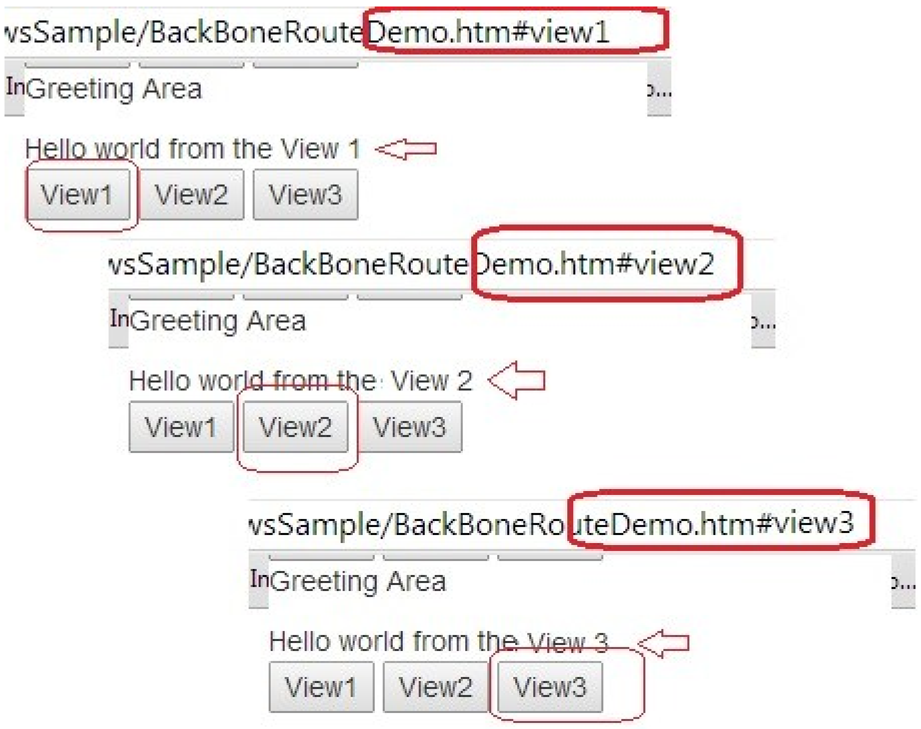
Backbone history is a global router that will keep track of the history and let us enable the routing in the application. To Instantiate a route and start tracking the navigation history, we need to simply create the router class and call Backbone.history.start for let the backbone start listening to routes and manage history.

```
$(document).ready(function () {
    router = new myRouter();
    Backbone.history.start();
})
```

Invoking and requesting Routes

A route can either be invoked from the other parts of the application or it can simply be requested by the user.

Invoking Route: Application wants to navigate to a specific route (this can be done by navigating to a route by calling the navigate function: router.navigate('view1');
Route Request: User Enters the fully qualified URL (this will work seamlessly)
Let us run the application and see the result.

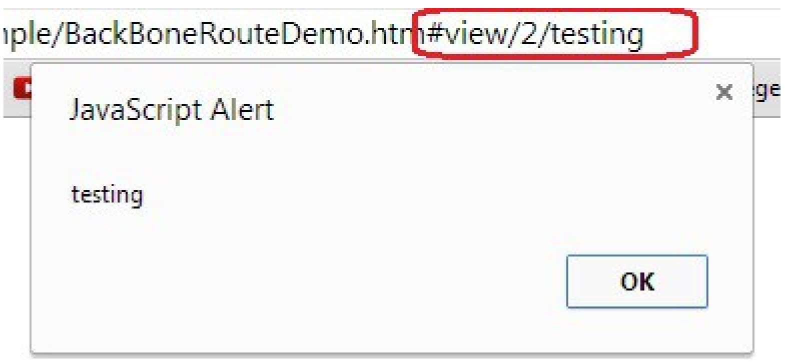

vsSample/BackBoneRouteDemo.htm#view1
InGreeting Area
Hello world from the View 1
View1
View2
View3
vsSample/BackBoneRouteDemo.htm#view2
InGreeting Area
Hello world from the View 2
View1
View2
View3
vsSample/BackBoneRouteDemo.htm#view3
InGreeting Area
Hello world from the View 3
View1
View2
View3

Passing parameters in the Routes

We can also pass parameters in the route. Lets us try to create a new route where the user will request for a view in a parameterized manner. Parameters can be defined as “ route/:param”

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

    greeting: null,
    container: null,
    view1: null,
    view2: null,
    view3: null,

    initialize: function () {
        this.greeting = new GreetModel({ Message: "Hello world" });
        this.container = new ContainerView({ el: $("#rAppContainer"), model: this.greeting });
    },

    routes: {
        "": "handleRoute1",
        "view/:viewid": "handleRouteAll"
    },

     handleRouteAll: function (viewid) {

        if (viewid == 1) {
            this.handleRoute1();
        }
        else if (viewid == 2) {
            this.handleRoute2();
        }
        else if (viewid == 3) {
            this.handleRoute3();
        }
    },

    handleRoute1: function () {
        if (this.view1 == null) {
            this.view1 = new View1({ model: this.greeting });
        }

        this.container.myChildView = this.view1;
        this.container.render();
    },

    handleRoute2: function () {
        if (this.view2 == null) {
            this.view2 = new View2({ model: this.greeting });
        }

        this.container.myChildView = this.view2;
        this.container.render();
    },

    handleRoute3: function () {
        if (this.view3 == null) {
            this.view3 = new View3({ model: this.greeting });
        }

        this.container.myChildView = this.view3;
        this.container.render();
    }
});
```

The above route can be invoked by passing view/2 as URL. the viewId passed to the router will be 2.


ıple/BackBoneRouteDemo.htm#view/2/testing

JavaScript Alert

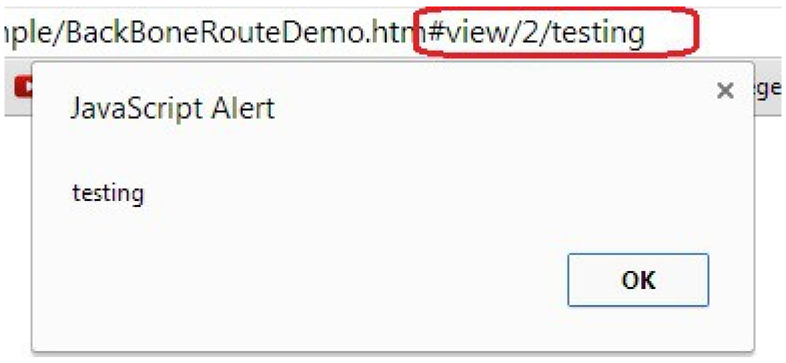
testing

Having optional Parameters in Routes

We can also pass optional parameters in the routes, Lets try to pass a simple parameter in the above defined route and see how it works. optional parameters can be defined as “ route(/:param)” .

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

    routes: {
        "": "handleRoute1",
        "view1": "handleRoute1",
        "view2": "handleRoute2",
        "view3": "handleRoute3",
        "view/:viewid(/:msg)": "handleRouteAll"
    },

    handleRouteAll: function (viewid, msg) {

        if (msg) {
            alert(msg);
        }
    }
});
```

In the above code, if we pass the second parameter i.e. view/2/test, the alert will be shown else not.


ıple/BackBoneRouteDemo.htm#view/2/testing

JavaScript Alert

testing

Note: The route definition can also contain complex regex based patterns if we need one route to handle multiple URLs based on some regular expression.

Point of interest

In this article we saw backbone.js routes. We saw how routes enable us to create bookmarkable URLs and will let the user request a view based on URL. We also looked at how we can use browser navigation by using backbone history. This has been written from a beginner’ s perspective. I hope this has been informative.

原文链接：http://rahulrajatsingh.com/2014/08/backbone-tutorial-part-7-understanding-backbone-js-routes-and-history/
附件下载:

- backboneRoutesSample.zip (323.6 KB)
- dl.iteye.com/topics/download/1143b212-bcae-3af4-b487-b3c4fc124e47

1.8 Part 8: Understanding Backbone.js Events

发表时间: 2016-06-22 关键字: backbone, javascript, mvc

In this article, we will look at events in Backbone.js. We will see how backbone provides us events and how we can use backbone events in our application.

Background

Events are a vital part of any application framework. Events are useful mainly in two scenarios when it comes to applications. Events can be particularly very useful when we want to implement a publisher subscriber model in our application where any change in one area of application(the publisher) will trigger a notification to everyone who is interested(subscribers).

There are a plethora of JavaScript libraries that can be used to implement eventing in our JavaScript application. Backbone also provides a very nice implementation of eventing mechanism which makes the use of publisher subscriber model in our application seamless. Let us now look at how we can use events in a backbone application.

Link to complete series:

- Part 1: Introduction to Backbone.Js
- Part 2: Understanding the basics of Backbone Models
- Part 3: More about Backbone Models
- Part 4: CRUD Operations on BackboneJs Models using HTTP REST Service
- Part 5: Understanding Backbone.js Collections
- Part 6: Understanding Backbone.js Views
- Part 7: Understanding Backbone.js Routes and History
- Part 8: Understanding Backbone.js Events

Using the code

Backbone provides a very simple, clean and elegant way to use events. What backbone does is that it let any object to be associated with backbone events simply by extending from the Backbone.Events. This can be done by calling _.extend method on the object instance. Let us create a simple object that extends from Backbone.Event.

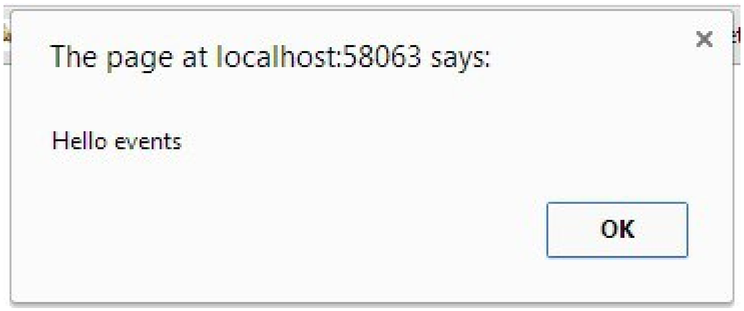
```
var testObj = {};
_.extend(testObj, Backbone.Events);
```

Once we have extended from Backbone.Event, we can use onto hook the callback functions with the event. Lets define a simple function and subscribe to an event.

```
function ShowMeWhenSomethingHappens(message) {
    alert(message);
}

testObj.on('something', ShowMeWhenSomethingHappens);
```

Now the simplest way to invoke this event is by calling the triggeron the event name. Lets invoke the function and see the results.

```
testObj.trigger('something', 'Hello events');
```


The page at localhost:58063 says:

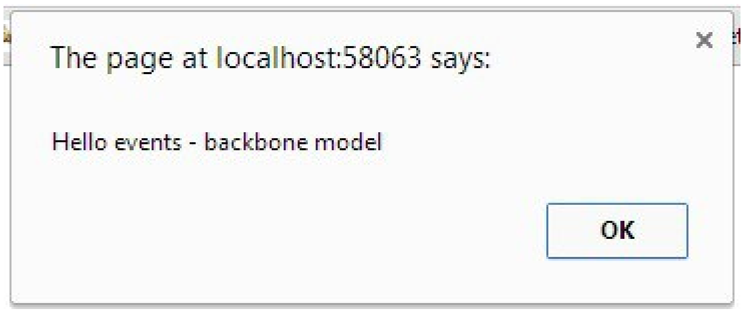
Hello events

So we can see how simple it is to use triggers with backbone. Or is it? The only pain point I see in this event mechanism is the hookup of Backbone events with objects using _.extend. Fortunately, we don’ t have to call this extend to hook backbone eventing mechanism with any object. If we have an object that extends from a backbone type i.e. model, view, collection then we have the event mechanism hooked with it by default. So lets say of we have a simple model like following.

```
var Book = Backbone.Model.extend({
    defaults: {
        ID: "",
        BookName: ""
    }
});
```

It if already hooked with the backbone events. So if if I run the following code, it should just work fine and show us the alert like the previous code.

```
var book = new Book();

book.on('something', ShowMeWhenSomethingHappens);

book.trigger('something', 'Hello events - backbone model');
```


The page at localhost:58063 says:

Hello events - backbone model

In the same way events can be used with any of the backbone object seamlessly. Let us now look at the functions that can be used to take full control over these events. The first function that is of interest to us is on. onattaches a callback function with an event.

```
var book = new Book();
book.on('something', ShowMeWhenSomethingHappens);
```

The callback functions that are attached to an event using onfunction can be removed by using off. So if we want to remove the callback function attached to the event in the previous step, we just have to use the offfunction.

```
book.off('something', ShowMeWhenSomethingHappens);
```

If we want to associate a callback with an event that should only be called once, we can use once.

```
book.once('something', ShowMeWhenSomethingHappens);
```

And finally, how can we trigger an event? The events can be triggered by using the triggerfunction.

```
book.trigger('something', 'Hello events - backbone model');
```

These are the basic functions that we can use to subscribe to events. The important thing to note here is that these functions are being used on the object that is publishing the events. What if we want to use another object i.e. the subscriber then there is another set of functions that we can use. Before looking at the function lets try to understand this scenario better. Lets say I have another model Catalogin my application.

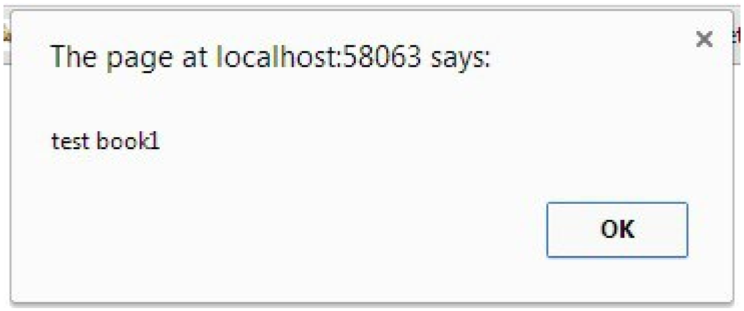
```
var Catalog = Backbone.Model.extend({
    defaults: {
        ID: "",
        CatalogName: ""
    },

    // code ommitted for brevity

    bookChanged : function(book) {
        alert(book.get("BookName"));
    }
});
```

What this model is expecting is that whenever a book is changed, the bookChangedfunction will get called. One way to do that is by using the on function as:

```
var catalog = new Catalog();

var book = new Book({BookName : "test book1"});

book.on('changed', catalog.bookChanged);

book.trigger('changed', book);
```

Another way to perform the same thing is by using the event association functions on the subscriber i.e. the catalog object. To achieve this, we need to use the listenTofunction. Lets see how this can be done.

```
var catalog = new Catalog();

var book = new Book({BookName : "test book1"});

catalog.listenTo(book, 'changed', catalog.bookChanged);

book.trigger('changed', book);
```


The page at localhost:58063 says:

test book1

In the above code, we are still subscribing the catalog object with the book’ s event but using listenTofunction on the catalog object instead. If we want to remove the associated object the we can use the stopListeningfunction.

```
catalog.stopListening(book);
```

If we want the equivalnt of once in this scenario, we can use listenToOncefunction instead.

```
catalog.listenToOnce(book, 'changed', catalog.bookChanged);
```

The benefit of using these set of functions over the previously shown one is that the subscribers can keep track of the all the events they are subscribed to and then selectively add and remove their subscriptions.

Note. The above code example is very contrived and far from the real world example. Its only purpose is to show how the given set of functions work.

There are a lot of build in events that backbone framework triggers that we can subscribe to in our application. For example “change” will be triggered on a model when its state is changed. “add” and “remove” will be called whenever an item is added or removed from the collection respectively. There are a lot of such built in events that the framework triggers on models, view, collections and routes. I recommend referring the backbone documentation for the exhaustive list.

Point of interest

In this small article we looked at events in backbone. How backbone makes it extremely easy to work with events and how it provides a lot of built in events that can readily be subscribed from the application. This article has been written from a beginner’s perspective. I hope this has been informative.

原文链接：http://rahulrajatsingh.com/2015/02/backbone-tutorial-part-8-understanding-backbone-js-events/
附件下载:

- backboneSample.zip (126.9 KB)
- dl.iteye.com/topics/download/2dccc982-c5b4-37b3-a5ef-35b7b5d1f47b