EBook Gratis

APRENDIZAJE backbone.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#backbone.j

S

Tabla de contenido

Acerca de

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

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

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

Capítulo 1: Empezando con backbone.js

Observaciones

Backbone es una biblioteca de JavaScript del lado del cliente simple pero robusta para crear aplicaciones. Los datos se representan como modelos, que se pueden recopilar en colecciones. El estado del modelo se muestra con vistas .

Backbone intenta proporcionar el conjunto mínimo de estructuras de datos y primitivas de interfaz de usuario que serían útiles en una aplicación web de JavaScript. Su objetivo es proporcionar estas herramientas sin dictar cómo usarlas o cómo debería ser su caso de uso. Esto significa que al desarrollador se le da mucha libertad para diseñar la experiencia completa de su aplicación.

Examples

Configuración básica

Backbone requiere un guión bajo (y, opcionalmente) jQuery - para la manipulación de DOM (usando Backbone.View) y la persistencia RESTful.

La forma más rápida de comenzar a utilizar Backbone es crear un archivo `index.html` con etiquetas de script simples en el HTML `<head>` :

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

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

Backbone ya está disponible para su uso en la página.

Ejemplo que muestra los conceptos básicos.

El siguiente ejemplo es una introducción a:

- Compilación de plantillas usando guiones bajos.
- Acceso a variables en una plantilla
- Creando una vista
- Renderizar una vista
- Mostrando una vista

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

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

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

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

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

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

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

Ejemplo de uso de Backbone (principalmente Backbone.Model)

Backbone.js se compone de cuatro componentes separados: colecciones, modelos, enrutadores

y vistas. Cada uno de estos sirve para diferentes propósitos:

- `Model` : representa un solo objeto de datos, pero agrega funcionalidades adicionales que no son proporcionadas por objetos nativos de JavaScript, como un sistema de eventos y una forma más conveniente de recuperar y enviar datos a un servidor remoto.
- `Collection` : representa un conjunto o "colección" de Modelos y proporciona funcionalidades para administrar sus modelos.
- `View` : representa una parte única de la interfaz de usuario; cada Vista envuelve un elemento HTML DOM, y proporciona una estructura para trabajar con ese elemento, así como características de conveniencia como el enlace simple de eventos.
- `Router` : permite una "aplicación de una sola página" al permitir que una aplicación active una lógica diferente (p. Ej., Mostrar páginas diferentes) en respuesta a los cambios de la URL.

Crea tus propios sabores.

Antes de ver cómo usar cada uno de estos componentes, primero echemos un vistazo rápido al sistema de clases de Backbone. Para crear una nueva subclase de una clase Backbone, simplemente llame al método `extend` de la clase original y pásele las propiedades de instancia y las propiedades de la clase (estáticas) como objetos:

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

Al igual que con cualquier otro sistema de clases, los métodos de instancia se pueden llamar en instancias (objetos) de la clase, mientras que los métodos estáticos se llaman directamente en la propia clase (el constructor):

```
var myInstance = new MyModelClass();

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

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

Usando una clase

Ahora, veamos un ejemplo rápido de cómo puedes usar cada clase. Comenzaremos con un `Model` de un libro.

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

Vamos a desglosar lo que acaba de pasar allí. Primero, creamos una subclase de `Book` del `Model` , y le dimos dos propiedades de instancia.

- `idAttribute` le dice a Backbone que use el atributo "isbn" del modelo como su ID al realizar operaciones AJAX.
- `urlRoot` , le dice a Backbone que busque datos de libros en `www.example.com/book` .

Ahora vamos a crear una instancia de un libro y obtener sus datos del servidor:

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

Cuando creamos un nuevo `Book` le pasamos un objeto y Backbone usa este objeto como los "atributos" iniciales (los datos) del `Model` . Debido a que Backbone sabe que el `idAttribute` es `isbn` , sabe que la URL de nuestro nuevo Libro es `/book/0486403491` . Cuando le decimos que lo `fetch` , Backbone usará jQuery para realizar una solicitud AJAX para los datos del libro. `fetch` devuelve una promesa (como `$.ajax`), que puede usar para desencadenar acciones una vez que se haya completado la búsqueda.

Se puede acceder o modificar los atributos utilizando los métodos `get` o `set` :

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

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

`Models` también tienen un sistema de eventos que puede utilizar para reaccionar cuando le ocurren cosas a un `Model` . Por ejemplo, para registrar un mensaje cada vez que cambie el número de `numberOfPages` , puede hacer:

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

Para una introducción más detallada de las otras clases de Backbone, vea sus páginas de documentación individuales.

Hello Web (configuración básica de tipo "Hello World")

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

```
min.js"></script>

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

        <script>
            $( function(){
                ( function(){
                    var View = Backbone.View.extend( {
                        "el": "body",
                        "template": _.template( "<p>Hello, Web!</p>" ),

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

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

Lea Empezando con backbone.js en línea: https://riptutorial.com/es/backbone-js/topic/1619/empezando-con-backbone-js

Capítulo 2: Colección

Sintaxis

- // Nueva colección personalizada
  var MyCollection = Backbone.Collection.extend (propiedades, [classProperties]);
- // Nueva instancia de colección
  var collection = new Backbone.Collection ([modelos], [opciones]);

Parámetros

Parámetro	Detalles
propiedades	Propiedades de instancia.
propiedades de clase	*Opcional.* Las propiedades que existen y se comparten con cada instancia de colección de este tipo.
modelos	*Opcional.* La matriz inicial de modelos (u objetos). Si este parámetro se omite, la colección estará vacía.
opciones	*Opcional.* Objeto que sirve para configurar la colección y luego se pasa a la función de `initialize` .

Observaciones

Las colecciones son conjuntos ordenados de modelos. Puede vincular los eventos de `"change"` para recibir notificaciones cuando se modifique cualquier modelo de la colección, escuchar los eventos de `"add"` y `"remove"` , `fetch` la colección del servidor y usar un conjunto completo de métodos Underscore.js .

Cualquier evento que se active en un modelo en una colección también se activará en la colección directamente, para su conveniencia. Esto le permite escuchar los cambios a atributos específicos en cualquier modelo en una colección.

Examples

Crear una colección personalizada

Para crear una nueva colección "clase":

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

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

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

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

Todas las propiedades son opcionales y están ahí solo como una demostración. Un `Backbone.Collection` se puede utilizar como está.

Entonces usarlo es tan simple como:

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


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


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

Saldrá:

> Programación de aplicaciones frontend con backbone.
> Columna vertebral para los maniquíes

Obteniendo y renderizando datos del servidor.

Necesitamos definir una colección con una propiedad `url` . Esta es la url a un punto final de API que debería devolver una matriz formateada json.

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

Luego, dentro de una vista, buscaremos y renderizaremos de forma asíncrona:

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

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

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

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

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

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

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

La forma más sencilla de usar esta vista:

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

Colección.url ()

Por defecto, la propiedad `url` no está definida. Al llamar a `fetch()` (mientras se usa el `Backbone.sync` predeterminado) se obtendrá una solicitud GET para los resultados de `url` .

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

  url: '/api/users',

  // or

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

});

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

Lea Colección en línea: https://riptutorial.com/es/backbone-js/topic/6022/coleccion

Capítulo 3: Enrutador

Examples

Creando un enrutador

El servidor web sirve al usuario según la solicitud enviada por el navegador, pero cómo el usuario le dirá al navegador lo que está buscando, es cuando necesitamos la URL. Cada página web en Internet tiene una URL que se puede marcar, copiar, compartir y guardar para futuras referencias. En la aplicación Backbone de una sola página, todo lo que vemos es una vista, las vistas se forman y se representan para mostrar una página específica, pero ¿qué pasa si el usuario desea ver la misma información nuevamente? Para lograr esto, necesitamos implementar una aplicación basada en enrutador Backbone que genere vistas basadas en el nombre de la ruta y los parámetros.

Un ejemplo muy simple de un enrutador Backbone:

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

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

El código anterior es solo un código de ejemplo que demuestra cómo puede crear un enrutador Backbone y obtener parámetros de la URL para representar las vistas correspondientes.

Explicación de cómo funcionará y se comportará el enrutador anterior:

Casos por URLs:

- **localhost: 8080 / # search / saurav** - la ruta de "búsqueda" del enrutador (función) se activará con los parámetros name = "saurav" y page = null, ahora la función userCollection.fetch () buscará a todos los usuarios que tengan name = "saurav" desde el backend y se procesarán los detalles de cada usuario uno por uno.
- **localhost: 8080 / # search / saurav / p6** - la ruta de "búsqueda" del enrutador (función) se activará con los parámetros nombre = "saurav" y página = 6, ahora la función userCollection.fetch () buscará a todos los usuarios que tengan nombre = " saurav "de la página 6 del servidor y mostrará los detalles de cada usuario uno por uno.
- **localhost: 8080 / # list** : la ruta (lista) de la "lista" del enrutador se activará, ahora la función userCollection.fetch () capturará a todos los usuarios desde el backend y rendirá los detalles de cada usuario uno por uno.
- **localhost: 8080 / # perfil / 92** - la ruta (función) del "perfil" del enrutador se activará, crearemos una nueva instancia de userModel con id = userId ie 92 y buscaremos los detalles del usuario desde el backend y renderizaremos el userView con eso datos.

Un ejemplo fácil de experimentar:

Visite http://backbonejs.org en el navegador Chrome, abra la consola de herramientas del desarrollador y pegue el siguiente código:

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

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

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

    search: function(query, page) {
       console.log("searched " + query + " " + page);
    }
});
```

```
var work = new Workspace();
Backbone.history.start();
```

Ahora, reemplace la URL en el navegador con " http://backbonejs.org/#search/kiwis/p9 " y presione la tecla enter. Esto activará la ruta de "búsqueda" (función) con los parámetros consulta = "kiwis" y página = 9 y verá una salida en la consola del navegador, es decir, "kiwis buscados 9".

Cambiando la ruta a través del código:

- **Caso 1** : ejecute el código `work.navigate("search/kiwis/p7", {trigger: true});` en la consola e imprimirá la salida "kiwis buscados 7", pero si intenta ejecutar el mismo código con el mismo parámetro, entonces no ocurrirá nada, vea el siguiente caso.
- **Caso 2** : ejecute el código `work.navigate("search/kiwis/p7", {trigger: false});` en la consola no imprimirá nada porque la ruta no se activará.
- **Caso 3** : en caso de que necesite volver a cargar la ruta actual, deberá ejecutar este código `Backbone.history.loadUrl("search/kiwis/p7");` .
- **Caso 4** : Ejecutar el código `work.navigate("search/kiwis/p15");` solo cambiará la URL pero no activará la ruta correspondiente (función).
- **Caso 5** : Ejecutando el código `work.navigate("search/kiwis/p11", {trigger: true});` Cambiará la URL y activará la ruta.
- **Caso 6** : la ejecución del código `work.navigate("search/kiwis/p17", {trigger: true, replace: true})` reemplazará la ruta existente con esta ruta, por lo tanto, hacer clic en el botón de retroceso del navegador lo llevará a 2 rutas para "buscar / kiwis / p15 ".

Lea Enrutador en línea: https://riptutorial.com/es/backbone-js/topic/7566/enrutador

Capítulo 4: Modelo

Sintaxis

- var MyModel = Backbone.Model.extend (propiedades, [classProperties]); // Crear un modelo personalizado
- var model = new Backbone.Model ([atributos], [opciones]); // Instancia un objeto modelo

Parámetros

Parámetro	Detalles
propiedades	Propiedades de instancia.
propiedades de clase	*Opcional.* Propiedades que existen y se comparten con cada instancia de modelo de este tipo.
atributos	*Opcional.* Valores iniciales de los `attributes` del modelo. Si este parámetro se `defaults` , el modelo se inicializará con los valores especificados por la propiedad `defaults` del modelo.
opciones	*Opcional.* Objeto que sirve para configurar el modelo y luego se pasa a la función de `initialize` .

Examples

Creando modelos

Los modelos de red troncal describen cómo se almacenan los datos utilizando objetos JavaScript. Cada modelo es un hash de campos llamados atributos y el comportamiento del modelo, incluida la validación, se describe mediante opciones.

Un modelo de elemento Todo en un TodoApp sería

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

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

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

```
    }

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

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

Ampliación de modelos

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

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

});

var Bicycle = Vehicle.extend({

  defaults: {
    wheels: 2
  }

});

var Car = Vehicle.extend({

  defaults: {
    wheels: 4
  }

});

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

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

Model.urlRoot & Model.url ()

Por defecto, la propiedad `urlRoot` no está definida. El método `url` utiliza esta propiedad `urlRoot` para crear una URL relativa donde el recurso del modelo estaría ubicado en el servidor.

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

  urlRoot: '/api/users',

  // or

  urlRoot: function () {
```

```
    return '/api/users'
  }

});

var user = new User();
```

El método `url` primero verificará si se ha definido el `idAttribute` del modelo (predeterminado en 'id'). Si no, el modelo es `isNew` y la `url` simplemente devolverá los resultados de `urlRoot` .

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

Si se ha definido el `idAttribute` del modelo, `url` devolverá `urlRoot` + el `idAttribute` del modelo

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

Llamar a `save` en un nuevo modelo dará como resultado una solicitud POST a los resultados de `url`

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

Llamar a `save` en un modelo existente dará como resultado una solicitud PUT a los resultados de `url`

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

Llamar a `fetch` un modelo existente dará como resultado una solicitud GET para los resultados de `url`

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

Llamar a `destroy` en un modelo existente dará como resultado una solicitud DELETE a los resultados de `url`

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

Lea Modelo en línea: https://riptutorial.com/es/backbone-js/topic/4056/modelo

Capítulo 5: Sincronizar

Introducción

`sync` es una función que Backbone utiliza para manejar todo el envío o recepción de datos a / desde un servidor remoto. La implementación predeterminada utiliza jQuery (o Zepto) para realizar operaciones AJAX cuando se sincronizan los datos. Sin embargo, este método puede anularse para aplicar diferentes comportamientos de sincronización, como: - Usar `setTimeout` para `setTimeout` varias actualizaciones en una sola solicitud - Enviar datos del modelo como XML en lugar de JSON - Usar WebSockets en lugar de Ajax

Sintaxis

- sincronización (método, modelo, opciones)

Parámetros

parámetro	detalles
método	crear, leer, actualizar, eliminar
modelo	El modelo a guardar (o colección a leer).
opciones	devoluciones de llamada de éxito y error, y todas las demás opciones de solicitud de jQuery

Examples

Ejemplo básico

El método sync () lee y obtiene los datos del modelo.

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

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

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

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

este código producirá:

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

Lea Sincronizar en línea: https://riptutorial.com/es/backbone-js/topic/8178/sincronizar

Capítulo 6: url y urlRoot

Examples

Modificando Model.url ()

`Model.url` y `Collection.url` solo se usan internamente por el método `Backbone.sync` predeterminado. El método predeterminado asume que estás enlazando con una API RESTful. Si está utilizando un diseño de punto final diferente, deseará anular el método de `sync` y es posible que desee utilizar el método de `url` .

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

  urlRoot: '/path-to-model',

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

});

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

Lea url y urlRoot en línea: https://riptutorial.com/es/backbone-js/topic/6430/url-y-urlroot

Capítulo 7: Ver

Sintaxis

- Crear: `var View = Backbone.View.extend( { /* properties */ } );`
- Construir: `var myView = new View( /* options */ );`
- `initialize` : método llamado automáticamente después de la construcción
- `render` : método utilizado para actualizar `this.el` con nuevo contenido

Examples

Una vista enlazada a HTML existente

Asumiendo este HTML en la página:

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

Una vista puede estar vinculada a ella con:

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

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

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

new MyPageView();
```

El HTML en el navegador ahora mostrará:

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

Función de inicialización de la vista.

Backbone llama a la `initialize` justo después de que se construye una vista.

Parámetros opcionales

La función de `initialize` recibe todos los argumentos pasados al constructor de la vista. Comúnmente, el hash de opciones que se usa para pasar las opciones predeterminadas de la vista:

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

Puede agregar cualquier atributo personalizado al hash de opciones y / o parámetros personalizados.

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

Y construyendo la vista:

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

Tenga en cuenta que todas las opciones de vista predeterminadas se agregan automáticamente al objeto de vista, por lo que no es necesario hacerlo en la función de `initialize` .

Inmediatamente hacer patrón

Un patrón común para el método de `initialize` es llamar al método de `render` para que se genere inmediatamente cualquier vista recién construida.

Este patrón solo se debe usar en casos en los que la construcción del objeto debe representarse inmediatamente en el documento HTML, vincular todos los detectores de eventos y realizar todas las demás acciones asociadas con la colocación de contenido en el DOM.

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

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

Sin embargo, se debe tener en cuenta que *algunas* Vistas no deben representarse de forma inmediata hasta que se `.render` manualmente (o por algún otro método).

Otro patrón de `initialize` común es agregar cosas al objeto Ver que se necesitará más adelante:

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

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

        this.render();
    },

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

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

El DOM ahora contendrá `<p>This is Bill's page</p>` en el `body` .

Lea Ver en línea: https://riptutorial.com/es/backbone-js/topic/2728/ver

Creditos

S. No	Capítulos	Contributors
1	Empezando con backbone.js	Community, Emile Bergeron, machineghost, Nighon, nikhil mehta, Peter, rockerest
2	Colección	Cakes, Emile Bergeron
3	Enrutador	saurav
4	Modelo	Cakes, Emile Bergeron, Louis, Peter, RamenChef
5	Sincronizar	Braiam, Farhad, machineghost
6	url y urlRoot	Cakes, Emile Bergeron
7	Ver	Emile Bergeron, Louis, rockerest, T J