Marionette.Application

The **Application** is a container for the rest of your code. It is recommended  
that every Marionette app have at least one instance of Application.

By creating an Application you get three important things:

* A start method to kick off your application.  
  This allows you an opportunity to do things that may need to occur before, say, you  
  begin routing. An example would be making an AJAX call to request data that your app  
  needs before starting.
* A namespace to keep things off of the window.  
  If you are not using a module loader like ES6 modules, CommonJS, or AMD, then  
  you can use the Application to store your Javascript objects. And if you are  
  using one of those module systems, then you can still attach things to the  
  application to aid in debugging.
* Integration with the Marionette Inspector. The Marionette Inspector is a fantastic tool  
  that makes it easy to understand and debug your application. Using the Application Class  
  will automatically hook up your application to that extension.

Note that the Application is undergoing many changes to become more lightweight. While it  
still includes many more features beyond what has been listed here, such as a Radio Channel and Regions,  
these features are now deprecated. Refer to the relevant sections below to learn what to use  
instead of these deprecated features.

Documentation Index

* [Getting Started](http://marionettejs.com/docs/v2.4.1/marionette.application.html#getting-started)
* [initialize](http://marionettejs.com/docs/v2.4.1/marionette.application.html#initialize)
* [Application Events](http://marionettejs.com/docs/v2.4.1/marionette.application.html#application-events)
* [Starting An Application](http://marionettejs.com/docs/v2.4.1/marionette.application.html#starting-an-application)
* [Application Regions (deprecated)](http://marionettejs.com/docs/v2.4.1/marionette.application.html#application-regions)
  + [jQuery Selector](http://marionettejs.com/docs/v2.4.1/marionette.application.html#jquery-selector)
  + [Custom Region Class](http://marionettejs.com/docs/v2.4.1/marionette.application.html#custom-region-class)
  + [Custom Region Class And Selector](http://marionettejs.com/docs/v2.4.1/marionette.application.html#custom-region-class-and-selector)
  + [Region Options](http://marionettejs.com/docs/v2.4.1/marionette.application.html#region-options)
  + [Overriding the default RegionManager](http://marionettejs.com/docs/v2.4.1/marionette.application.html#overriding-the-default-regionmanager)
  + [Get Region By Name](http://marionettejs.com/docs/v2.4.1/marionette.application.html#get-region-by-name)
  + [Removing Regions](http://marionettejs.com/docs/v2.4.1/marionette.application.html#removing-regions)
* [Application.mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.application.html#applicationmergeoptions)
* [Application.getOption](http://marionettejs.com/docs/v2.4.1/marionette.application.html#applicationgetoption)
* [Adding Initializers (deprecated)](http://marionettejs.com/docs/v2.4.1/marionette.application.html#adding-initializers)
* [The Application Channel (deprecated)](http://marionettejs.com/docs/v2.4.1/marionette.application.html#the-application-channel)
  + [Event Aggregator](http://marionettejs.com/docs/v2.4.1/marionette.application.html#event-aggregator)
  + [Request Response](http://marionettejs.com/docs/v2.4.1/marionette.application.html#request-response)
  + [Commands](http://marionettejs.com/docs/v2.4.1/marionette.application.html#commands)
  + [Accessing the Application Channel](http://marionettejs.com/docs/v2.4.1/marionette.application.html#accessing-the-application-channel)

Getting Started

A common pattern in Backbone apps is the following:

var app = {};

Two notable examples of this pattern are  
[DocumentCloud's source](https://github.com/documentcloud/documentcloud/blob/master/public/javascripts/application.js#L3) and  
[Backbone Boilerplate](https://github.com/backbone-boilerplate/backbone-boilerplate/blob/master/app/app.js#L1-L6). DocumentCloud  
is notable because it is the codebase that Backbone was abstracted from. If such a thing as a quintessential Backbone application  
existed, then that app would certainly be a candidate. Backbone Boilerplate is notable as perhaps the most popular library  
for bootstrapping a Backbone application. Do note that in the Backbone Boilerplate code the exported object is implicit.

The pattern of creating a Javascript object is so popular because it provides you with a location to  
put the pieces of your application. For instance, attaching a Router to this object is common practice.

Using a raw Javascript object is great, but Marionette provides a light wrapper for a plain Javascript object, which is the  
Application. One benefit to using the Application is that it comes with a start method. This can be used to accomplish  
tasks before the rest of your application begins. Let's take a quick look at an example:

// Create our Application

var app = new Mn.Application();

// Start history when our application is ready

app.on('start', function() {

Backbone.history.start();

});

// Load some initial data, and then start our application

loadInitialData().then(app.start);

In the simple example above, we could have just as easily started history after our initial data had loaded. This  
pattern becomes more useful as the startup phase of your application becomes more complex.

Initialize

Like other objects in Backbone and Marionette, Applications have an initialize method.  
It is called immediately after the Application has been instantiated, and is invoked with  
the same arguments that the constructor received.

var app = Marionette.Application.extend({

initialize: function(options) {

console.log('My container:', options.container);

}

});

// Although applications will not do anything

// with a `container` option out-of-the-box, you

// could build an Application Class that does use

// such an option.

var app = new app({container: '#app'});

Application Events

The Application object raises a few events during its lifecycle, using the  
[Marionette.triggerMethod](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethod) function. These events  
can be used to do additional processing of your application. For example, you  
may want to pre-process some data just before initialization happens. Or you may  
want to wait until your entire application is initialized to start  
Backbone.history.

The events that are currently triggered, are:

* **"before:start" / onBeforeStart**: fired just before the Application starts and before the initializers are executed.
* **"start" / onStart**: fires after the Application has started and after the initializers have been executed.

MyApp.on("before:start", function(options){

options.moreData = "Yo dawg, I heard you like options so I put some options in your options!";

});

MyApp.on("start", function(options){

if (Backbone.history){

Backbone.history.start();

}

});

The options parameter is passed through the start method of the application  
object (see below).

Starting An Application

Once you have your application configured, you can kick everything off by  
calling: MyApp.start(options).

This function takes a single optional parameter. This parameter will be passed  
to each of your initializer functions, as well as the initialize events. This  
allows you to provide extra configuration for various parts of your app throughout the  
initialization sequence.

var options = {

something: "some value",

another: "#some-selector"

};

MyApp.start(options);

Application Regions

Warning: deprecated  
This feature is deprecated. Instead of using the Application as the root  
of your view tree, you should use a Layout View. To scope your Layout View to the entire  
document, you could set its el to 'body'. This might look something like the following:

var RootView = Marionette.LayoutView.extend({

el: 'body'

});

Later, you can attach an instance of the RootView to your Application instance.

app.rootView = new RootView();

Application instances have an API that allow you to manage [Regions](http://marionettejs.com/docs/v2.4.1/marionette.region.html).  
These Regions are typically the means through which your views become attached to the document.

You can create Regions through the addRegions method by passing in an object  
literal or a function that returns an object literal.

There are three syntax forms for adding a region to an application object.

jQuery Selector

The first is to specify a jQuery selector as the value of the region  
definition. This will create an instance of a Marionette.Region directly,  
and assign it to the selector:

MyApp.addRegions({

someRegion: "#some-div",

anotherRegion: "#another-div"

});

Custom Region Class

The second is to specify a custom region class, where the region class has  
already specified a selector:

var MyCustomRegion = Marionette.Region.extend({

el: "#foo"

});

MyApp.addRegions(function() {

return {

someRegion: MyCustomRegion

};

});

Custom Region Class And Selector

The third method is to specify a custom region class, and a jQuery selector  
for this region instance, using an object literal:

var MyCustomRegion = Marionette.Region.extend({});

MyApp.addRegions({

someRegion: {

selector: "#foo",

regionClass: MyCustomRegion

},

anotherRegion: {

selector: "#bar",

regionClass: MyCustomRegion

}

});

Region Options

You can also specify regions per Application instance.

new Marionette.Application({

regions: {

fooRegion: '#foo-region'

}

});

Overriding the default RegionManager

If you need the RegionManager's class chosen dynamically, specify getRegionManager:

Marionette.Application.extend({

// ...

getRegionManager: function() {

// custom logic

return new MyRegionManager();

}

This can be useful if you want to attach Application's regions to your own instance of RegionManager.

Get Region By Name

A region can be retrieved by name, using the getRegion method:

var app = new Marionette.Application();

app.addRegions({ r1: "#region1" });

var myRegion = app.getRegion('r1');

Regions are also attached directly to the Application instance, **but this is not recommended usage**.

Removing Regions

Regions can also be removed with the removeRegion method, passing in  
the name of the region to remove as a string value:

MyApp.removeRegion('someRegion');

Removing a region will properly empty it before removing it from the  
application object.

For more information on regions, see [the region documentation](http://marionettejs.com/docs/v2.4.1/marionette.region.html) Also, the API that Applications use to  
manage regions comes from the RegionManager Class, which is documented [over here](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html).

Application.mergeOptions

Merge keys from the options object directly onto the Application instance.

var MyApp = Marionette.Application.extend({

initialize: function(options) {

this.mergeOptions(options, ['myOption']);

console.log('The option is:', this.myOption);

}

})

More information at [mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettemergeoptions)

Application.getOption

Retrieve an object's attribute either directly from the object, or from the object's this.options, with this.options taking precedence.

More information [getOption](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettegetoption)

Adding Initializers

Warning: deprecated

This feature is deprecated, and is scheduled to be removed in version 3 of Marionette. Instead  
of Initializers, you should use events to manage start-up logic. The start event is an ideal  
substitute for Initializers.

If you were relying on the deferred nature of Initializers in your app, you should instead  
use Promises. This might look something like the following:

doAsyncThings().then(app.start);

Your application needs to do useful things, like displaying content in your  
regions, starting up your routers, and more. To accomplish these tasks and  
ensure that your Application is fully configured, you can add initializer  
callbacks to the application.

MyApp.addInitializer(function(options){

// do useful stuff here

var myView = new MyView({

model: options.someModel

});

MyApp.mainRegion.show(myView);

});

MyApp.addInitializer(function(options){

new MyAppRouter();

Backbone.history.start();

});

These callbacks will be executed when you start your application,  
and are bound to the application object as the context for  
the callback. In other words, this is the MyApp object inside  
of the initializer function.

The options argument is passed from the start method (see below).

Initializer callbacks are guaranteed to run, no matter when you  
add them to the app object. If you add them before the app is  
started, they will run when the start method is called. If you  
add them after the app is started, they will run immediately.

The Application Channel

Warning: deprecated

This feature is deprecated, and is scheduled to be removed in the next major release of Marionette.  
Instead of accessing Channels through the Application, you should use the Wreqr (or Radio) API.  
By default the application's channel is named 'global'. To access this channel, you can use  
the following code, depending on whether you're using Wreqr or Radio:

// Wreqr

var globalCh = Backbone.Wreqr.radio.channel('global');

// Radio

var globalCh = Backbone.Radio.channel('global');

Marionette Applications come with a [messaging system](http://en.wikipedia.org/wiki/Message_passing) to facilitate communications within your app.

The messaging system on the Application is the radio channel from Backbone.Wreqr, which is actually comprised of three distinct systems.

Marionette Applications default to the 'global' channel, but the channel can be configured.

var MyApp = new Marionette.Application({ channelName: 'appChannel' });

This section will give a brief overview of the systems; for a more in-depth look you are encouraged to read  
the [Backbone.Wreqr documentation](https://github.com/marionettejs/backbone.wreqr).

Event Aggregator

The Event Aggregator is available through the vent property. vent is convenient for passively sharing information between  
pieces of your application as events occur.

var MyApp = new Backbone.Marionette.Application();

// Alert the user on the 'minutePassed' event

MyApp.vent.on("minutePassed", function(someData){

alert("Received", someData);

});

// This will emit an event with the value of window.someData every minute

window.setInterval(function() {

MyApp.vent.trigger("minutePassed", window.someData);

}, 1000 \* 60);

Request Response

Request Response is a means for any component to request information from another component without being tightly coupled. An instance of Request Response is available on the Application as the reqres property.

var MyApp = new Backbone.Marionette.Application();

// Set up a handler to return a todoList based on type

MyApp.reqres.setHandler("todoList", function(type){

return this.todoLists[type];

});

// Make the request to get the grocery list

var groceryList = MyApp.reqres.request("todoList", "groceries");

// The request method can also be accessed directly from the application object

var groceryList = MyApp.request("todoList", "groceries");

Commands

Commands are used to make any component tell another component to perform an action without a direct reference to it. A Commands instance is available under the commands property of the Application.

Note that the callback of a command is not meant to return a value.

var MyApp = new Backbone.Marionette.Application();

MyApp.model = new Backbone.Model();

// Set up the handler to call fetch on the model

MyApp.commands.setHandler("fetchData", function(reset){

MyApp.model.fetch({reset: reset});

});

// Order that the data be fetched

MyApp.commands.execute("fetchData", true);

// The execute function is also available directly from the application

MyApp.execute("fetchData", true);

Accessing the Application Channel

To access this application channel from other objects within your app you are encouraged to get a handle of the systems  
through the Wreqr API instead of the Application instance itself.

// Assuming that we're in some class within your app,

// and that we are using the default 'global' channel

// it is preferable to access the channel like this:

var globalCh = Backbone.Wreqr.radio.channel('global');

globalCh.vent;

// This is discouraged because it assumes the name of your application

window.app.vent;

Marionette.AppRouter

Reduce the boilerplate code of handling route events and then calling a single method on another object.  
Have your routers configured to call the method on your object, directly.

Documentation Index

* [Configure Routes](http://marionettejs.com/docs/v2.4.1/marionette.approuter.html#configure-routes)
* [Configure Routes In Constructor](http://marionettejs.com/docs/v2.4.1/marionette.approuter.html#configure-routes-in-constructor)
* [Add Routes At Runtime](http://marionettejs.com/docs/v2.4.1/marionette.approuter.html#add-routes-at-runtime)
* [Specify A Controller](http://marionettejs.com/docs/v2.4.1/marionette.approuter.html#specify-a-controller)
* [onRoute](http://marionettejs.com/docs/v2.4.1/marionette.approuter.html#onroute)

Configure Routes

Configure an AppRouter with appRoutes. The route definition is passed on to Backbone's standard routing handlers. This means that you define routes like you normally would. However, instead of providing a callback method that exists on the router, you provide a callback method that exists on the controller, which you specify for the router instance (see below.)

var MyRouter = Backbone.Marionette.AppRouter.extend({

// "someMethod" must exist at controller.someMethod

appRoutes: {

"some/route": "someMethod"

},

/\* standard routes can be mixed with appRoutes/Controllers above \*/

routes : {

"some/otherRoute" : "someOtherMethod"

},

someOtherMethod : function(){

// do something here.

}

});

You can also add standard routes to an AppRouter with methods on the router.

Configure Routes In Constructor

Routes can be defined through the constructor function options, as well.

var MyRouter = new Marionette.AppRouter({

controller: myController,

appRoutes: {

"foo": "doFoo",

"bar/:id": "doBar"

}

});

This allows you to create router instances without having to .extend  
from the AppRouter. You can just create the instance with the routes defined  
in the constructor, as shown.

Add Routes At Runtime

In addition to setting the appRoutes for an AppRouter, you can add app routes  
at runtime, to an instance of a router. This is done with the appRoute()  
method call. It works the same as the built-in router.route() call from  
Backbone's Router, but has all the same semantics and behavior of the appRoutes  
configuration.

var MyRouter = Marionette.AppRouter.extend({});

var router = new MyRouter();

router.appRoute("/foo", "fooThat");

Also you can specify a controller with the multiple routes at runtime with method  
processAppRoutes. However, In this case the current controller of AppRouter will not change.

var MyRouter = Marionette.AppRouter.extend({});

var router = new MyRouter();

router.processAppRoutes(myController, {

"foo": "doFoo",

"bar/:id": "doBar"

});

Specify A Controller

App routers can only use one controller object. You can either specify this  
directly in the router definition:

var someController = {

someMethod: function(){ /\*...\*/ }

};

Backbone.Marionette.AppRouter.extend({

controller: someController

});

... or in a parameter to the constructor:

var myObj = {

someMethod: function(){ /\*...\*/ }

};

new MyRouter({

controller: myObj

});

The object that is used as the controller has no requirements, other than it will  
contain the methods that you specified in the appRoutes.

It is recommended that you divide your controller objects into smaller pieces of related functionality  
and have multiple routers / controllers, instead of just one giant router and controller.

onRoute

If it exists, AppRouters will call the onRoute method whenever a user navigates within your app. The  
callback receives three arguments: the name, path, and arguments of the route.

Marionette.Behavior

A Behavior is an isolated set of DOM / user interactions that can be mixed into any View or another Behavior. Behaviors allow you to blackbox View specific interactions into portable logical chunks, keeping your Views simple and your code DRY.

Documentation Index

* [Motivation](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#the-motivation)
* [Using Behaviors](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#using)
* [API](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#api)
  + [Event proxy](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#the-event-proxy)
  + [Triggers](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#triggers)
  + [Model Events](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#model-events)
  + [Collection Events](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#model-events)
  + [Grouped Behaviors](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#grouped-behaviors)
  + [$](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#$)
  + [$el and el](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#$el-and-el)
  + [Defaults](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#defaults)
  + [View](http://marionettejs.com/docs/v2.4.1/marionette.behavior.html#view)

The Motivation

As you build more and more complex Views, you will find that your View becomes less about displaying model data, and more about interactions.

These interactions tend to be chunks of logic that you want to use in multiple views.

Using

Here is an example of a simple ItemView. Let's take a stab at simplifying it, and abstracting Behaviors from it.

var MyView = Marionette.ItemView.extend({

ui: {

"destroy": ".destroy-btn"

},

events: {

"click @ui.destroy": "warnBeforeDestroy"

},

warnBeforeDestroy: function() {

alert("you are destroying all your data is now gone!");

this.destroy();

},

onShow: function() {

this.ui.destroy.tooltip({

text: "what a nice mouse you have"

});

}

});

Interaction points, such as tooltips and warning messages, are generic concepts. There is no need to recode them within your Views. They are prime for abstraction into a higher level non-coupled concept, which is exactly what Behaviors provide you with.

Here is the syntax for declaring which behaviors get used within a View.  
The keys in the hash are passed to getBehaviorClass which looks up the correct Behavior class.  
The options for each Behavior are also passed through to the Behavior during initialization. The options are then stored within each Behavior under options.

var MyView = Marionette.ItemView.extend({

ui: {

"destroy": ".destroy-btn"

},

behaviors: {

DestroyWarn: {

message: "you are destroying all your data is now gone!"

},

ToolTip: {

text: "what a nice mouse you have"

}

}

});

Now let's create the DestroyWarn Behavior.

var DestroyWarn = Marionette.Behavior.extend({

// you can set default options

// just like you can in your Backbone Models

// they will be overriden if you pass in an option with the same key

defaults: {

"message": "you are destroying!"

},

// behaviors have events that are bound to the views DOM

events: {

"click @ui.destroy": "warnBeforeDestroy"

},

warnBeforeDestroy: function() {

alert(this.options.message);

// every Behavior has a hook into the

// view that it is attached to

this.view.destroy();

}

});

And onto the Tooltip behavior.

var ToolTip = Marionette.Behavior.extend({

ui: {

tooltip: '.tooltip'

},

onShow: function() {

this.ui.tooltip.tooltip({

text: this.options.text

});

}

});

Finally, the user must define a location for where their Behaviors are stored.  
A simple example of this would look like this:

Marionette.Behaviors.behaviorsLookup = function() {

return window.Behaviors;

}

In this example you would then store your Behaviors like this:

window.Behaviors.ToolTip = ToolTip;

window.Behaviors.DestroyWarn = DestroyWarn;

Note than in addition to extending a View with Behavior, a Behavior can itself use other Behaviors. The syntax is identical to that used for a View:

var Modal = Marionette.Behavior.extend({

behaviors: {

DestroyWarn: {

message: "Whoa! You sure about this?"

}

}

});

Nested Behaviors act as if they were direct Behaviors of the parent Behavior's view instance.

API

The Event Proxy

Behaviors are powered by an event proxy. What this means is that any events that are triggered by the view's triggerMethod function are passed to each Behavior on the view as well.

As a real world example, whenever in your View you would define a click event in the events hash, you can define the same event listeners and callbacks in the behavior's events hash. The same follows for modelEvents and collectionEvents. Think of your behavior as a receiver for all of the events on your view instance.

This concept also allows for a nice decoupled method to communicate to behaviors from your view instance.  
You can just call from within your view this.triggerMethod("SomeEvent", {some: "data"}). then your behavior class would look like this:

Marionette.Behavior.extend({

events: {

'click .foo' : 'onClick'

},

onClick: function(data) {

console.log("wow such data", data);

}

});

Model Events

modelEvents will respond to the view's model events.

Marionette.Behavior.extend({

modelEvents: {

"change:doge": "onDogeChange"

},

onDogeChange: function() {

// buy more doge...

}

});

Collection Events

collectionEvents will respond to the view's collection events.

Marionette.Behavior.extend({

collectionEvents: {

add: "onCollectionAdd"

},

onCollectionAdd: function() {

}

});

Life Cycle Methods

In addition to providing the same event hashes as Views, Behaviors allow you to use the same life cycle functions that you find on Views.  
That means methods like initialize, onRender, onBeforeShow, and onBeforeDestroy are all valid as long as the View that implements the Behavior fires the relevant events.

Marionette.Behavior.extend({

onRender: function() {

//apply a jQuery plugin to every .foo item within the view

this.$('.foo').bar();

}

});

Triggers

Any triggers you define on the Behavior will be triggered in response to the  
appropriate event on the view.

Marionette.Behavior.extend({

triggers: {

'click .label': 'click:label'

}

});

Grouped Behaviors

Then behaviors key allows a behavior to group multiple behaviors together.

Marionette.Behavior.extend({

behaviors: {

SomeBehavior: {}

}

});

$

$ is a direct proxy of the view's $ lookup method.

Marionette.Behavior.extend({

onShow: function() {

this.$('.zerg')

}

});

$el and el

el is a direct proxy of the view's el.  
Similarly, $el is a direct proxy of the view's el cached as a jQuery selector.

Marionette.Behavior.extend({

onShow: function() {

this.$el.fadeOut('slow')

}

});

defaults

defaults can be a hash or function to define the default options for your Behavior.  
The default options will be overridden depending on what you set as the options per Behavior (this works just like a Backbone.Model).

Marionette.Behavior.extend({

defaults: function() {

return {

'deepSpace': 9

}

}

});

Marionette.Behavior.extend({

defaults: {

'dominion': 'invasion',

'doge': 'amaze'

}

});

view

The view is a reference to the view instance that the Behavior is on.

Marionette.Behavior.extend({

handleDestroyClick: function() {

this.view.destroy();

}

});

ui

Behaviors can have their own ui hash, which will be mixed into the ui hash of its associated view instance.  
ui elements defined on either the Behavior or the View will be made available within events, and triggers. They  
also are attached directly to the Behavior and can be accessed within Behavior methods as this.ui.

Marionette.Behavior.extend({

ui: {

'foo' : 'li.foo'

},

doStuff: function() {

this.ui.foo.trigger('something');

}

})

Marionette.Behaviors

'Marionette.Behaviors' is a utility class that takes care of gluing your Behavior instances to their given View.  
The most important thing to understand when using this class is that you **MUST** override the class level behaviorsLookup method or set the option behaviorClass for things to work properly.

Documentation Index

* [API](http://marionettejs.com/docs/v2.4.1/marionette.behaviors.html#api)
  + [Behaviors Lookup](http://marionettejs.com/docs/v2.4.1/marionette.behaviors.html#behaviorslookup)
  + [getBehaviorClass](http://marionettejs.com/docs/v2.4.1/marionette.behaviors.html#getbehaviorclass)
  + [behaviorClass](http://marionettejs.com/docs/v2.4.1/marionette.behaviors.html#behaviorclass)

API

There are two class level methods that you can override on the Behaviors class. The rest of the class is tied to under the hood implementation details of Views.

behaviorsLookup

This method defines where your Behavior classes are stored. A simple implementation might look something like this.

Marionette.Behaviors.behaviorsLookup = function() {

return window.Behaviors;

}

By default the Behaviors are looked up by their key value in a given View's behavior hash.

In this sample (using the default getBehaviorClass implementation) your code will expect the following Behaviors to be present in window.Behaviors.DestroyWarn and window.Behaviors.ToolTip

var MyView = Marionette.ItemView.extend({

behaviors: {

DestroyWarn: {

message: "you are destroying all your data is now gone!"

},

ToolTip: {

text: "what a nice mouse you have"

}

}

});

getBehaviorClass

This method has a default implementation that is simple to override. It is responsible for the lookup of single Behavior from within the Behaviors.behaviorsLookup or elsewhere.

getBehaviorClass: function(options, key) {

if (options.behaviorClass) {

return options.behaviorClass;

}

return Behaviors.behaviorsLookup[key];

}

behaviorClass

This property lets you pass a class in for the Behavior to use (bypassing the normal key based lookup). This is nice to have when the Behavior is a dependency of the View in a module system like [requirejs](http://requirejs.org/) or [browserify](http://browserify.org/). Properties passed in this way will be used in getBehaviorClass.

define(['marionette', 'lib/tooltip'], function(Marionette, Tooltip) {

var View = Marionette.ItemView.extend({

behaviors: {

Tooltip: {

behaviorClass: Tooltip,

message: "hello world"

}

}

});

});

Marionette.Callbacks

Warning: deprecated

Marionette.Callbacks are deprecated, and are scheduled to be removed in the next major release of the library. Instead  
of Callbacks, you should use promises or events to manage asynchronous logic.

The Callbacks object assists in managing a collection of callback  
methods, and executing them, in an async-safe manner.

There are only two methods:

* add
* run

The add method adds a new callback to be executed later.

The run method executes all current callbacks in, using the  
specified context for each of the callbacks, and supplying the  
provided options to the callbacks.

Documentation Index

* [Basic Usage](http://marionettejs.com/docs/v2.4.1/marionette.callbacks.html#basic-usage)
* [Specify Context Per-Callback](http://marionettejs.com/docs/v2.4.1/marionette.callbacks.html#specify-context-per-callback)
* [Advanced / Async Use](http://marionettejs.com/docs/v2.4.1/marionette.callbacks.html#advanced--async-use)

Basic Usage

var callbacks = new Backbone.Marionette.Callbacks();

callbacks.add(function(options){

alert("I'm a callback with " + options.value + "!");

});

callbacks.run({value: "options"}, someContext);

This example will display an alert box that says "I'm a callback  
with options!". The executing context for each of the callback  
methods has been set to the someContext object, which is an optional  
parameter that can be any valid JavaScript object.

Specify Context Per-Callback

You can optionally specify the context that you want each callback to be  
executed with, when adding a callback:

var callbacks = new Backbone.Marionette.Callbacks();

callbacks.add(function(options){

alert("I'm a callback with " + options.value + "!");

// specify callback context as second parameter

}, myContext);

// the `someContext` context is ignored by the above callback

callbacks.run({value: "options"}, someContext);

This will run the specified callback with the myContext object set as  
this in the callback, instead of someContext.

Advanced / Async Use

The Callbacks executes each callback in an async-friendly  
manner, and can be used to facilitate async callbacks.  
The Marionette.Application object uses Callbacks  
to manage initializers (see above).

It can also be used to guarantee callback execution in an event  
driven scenario, much like the application initializers.

Marionette.CollectionView

The CollectionView will loop through all of the models in the  
specified collection, render each of them using a specified childView,  
then append the results of the child view's el to the collection view's  
el. By default the CollectionView will maintain a sorted collection's order  
in the DOM. This behavior can be disabled by specifying {sort: false} on initialize.

CollectionView extends directly from Marionette.View. Please see  
[the Marionette.View documentation](http://marionettejs.com/docs/v2.4.1/marionette.view.html)  
for more information on available features and functionality.

Additionally, interactions with Marionette.Region  
will provide features such as onShow callbacks, etc. Please see  
[the Region documentation](http://marionettejs.com/docs/v2.4.1/marionette.region.html) for more information.

Documentation Index

* [CollectionView's childView](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-childview)
  + [CollectionView's getChildView](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-getchildview)
  + [CollectionView's childViewOptions](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-childviewoptions)
  + [CollectionView's childViewEventPrefix](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-childvieweventprefix)
  + [CollectionView's childEvents](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-childevents)
  + [CollectionView's buildChildView](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-buildchildview)
  + [CollectionView's addChild](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-addchild)
  + [CollectionView's reorderOnSort](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-reorderonsort)
* [CollectionView's emptyView](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-emptyview)
  + [CollectionView's getEmptyView](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-getemptyview)
  + [CollectionView's isEmpty](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-isempty)
  + [CollectionView's emptyViewOptions](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-emptyviewoptions)
* [Callback Methods](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#callback-methods)
  + [onBeforeRender callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onbeforerender-callback)
  + [onRender callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onrender-callback)
  + [onBeforeReorder callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onbeforereorder-callback)
  + [onReorder callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onreorder-callback)
  + [onBeforeDestroy callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforedestroy-callback)
  + [onDestroy callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#ondestroy-callback)
  + [onBeforeAddChild callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onbeforeaddchild-callback)
  + [onAddChild callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onaddchild-callback)
  + [onBeforeRemoveChild callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onbeforeremovechild-callback)
  + [onRemoveChild callback](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#onremovechild-callback)
* [CollectionView Events](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionview-events)
  + ["before:render" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforerender-event)
  + ["render" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#render-event)
  + ["before:reorder" / "reorder" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforereorder--reorder-event)
  + ["before:destroy" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforedestroy-event)
  + ["destroy" / "destroy:collection" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#destroy--destroycollection-event)
  + ["before:add:child" / "add:child" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforeaddchild--addchild-event)
  + ["before:remove:child event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforeremovechild-event)
  + ["remove:child" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#removechild-event)
  + ["childview:\*" event bubbling from child views](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#childview-event-bubbling-from-child-views)
  + ["before:render:collection" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#beforerendercollection-event)
  + ["render:collection" event](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#rendercollection-event)
* [CollectionView render](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionview-render)
* [CollectionView: Automatic Rendering](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionview-automatic-rendering)
* [CollectionView: Re-render Collection](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionview-re-render-collection)
* [CollectionView's attachHtml](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-attachhtml)
* [CollectionView's resortView](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-resortview)
* [CollectionView's viewComparator](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-viewcomparator)
* [CollectionView's filter](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-filter)
* [CollectionView's children](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-children)
* [CollectionView destroy](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionview-destroy)

CollectionView's childView

Specify a childView in your collection view definition. This must be  
a Backbone view object definition, not an instance. It can be any  
Backbone.View or be derived from Marionette.ItemView.

var MyChildView = Marionette.ItemView.extend({});

Marionette.CollectionView.extend({

childView: MyChildView

});

Child views must be defined before they are referenced by the  
childView attribute in a collection view definition. Use getChildView  
to lookup the definition as child views are instantiated.

Alternatively, you can specify a childView in the options for  
the constructor:

var MyCollectionView = Marionette.CollectionView.extend({...});

new MyCollectionView({

childView: MyChildView

});

If you do not specify a childView, an exception will be thrown  
stating that you must specify a childView.

CollectionView's getChildView

The value returned by this method is the ChildView class that will be instantiated when a Model needs to be initially rendered.  
This method also gives you the ability to customize per Model ChildViews.

var FooBar = Backbone.Model.extend({

defaults: {

isFoo: false

}

});

var FooView = Marionette.ItemView.extend({

template: '#foo-template'

});

var BarView = Marionette.ItemView.extend({

template: '#bar-template'

});

var MyCollectionView = Marionette.CollectionView.extend({

getChildView: function(item) {

// Choose which view class to render,

// depending on the properties of the item model

if (item.get('isFoo')) {

return FooView;

}

else {

return BarView;

}

}

});

var collectionView = new MyCollectionView();

var foo = new FooBar({

isFoo: true

});

var bar = new FooBar({

isFoo: false

});

// Renders a FooView

collectionView.collection.add(foo);

// Renders a BarView

collectionView.collection.add(bar);

CollectionView's childViewOptions

There may be scenarios where you need to pass data from your parent  
collection view in to each of the childView instances. To do this, provide  
a childViewOptions definition on your collection view as an object  
literal. This will be passed to the constructor of your childView as part  
of the options.

var ChildView = Marionette.ItemView.extend({

initialize: function(options) {

console.log(options.foo); // => "bar"

}

});

var CollectionView = Marionette.CollectionView.extend({

childView: ChildView,

childViewOptions: {

foo: "bar"

}

});

You can also specify the childViewOptions as a function, if you need to  
calculate the values to return at runtime. The model will be passed into  
the function should you need access to it when calculating  
childViewOptions. The function must return an object, and the attributes  
of the object will be copied to the childView instance's options.

var CollectionView = Marionette.CollectionView.extend({

childViewOptions: function(model, index) {

// do some calculations based on the model

return {

foo: "bar",

childIndex: index

}

}

});

CollectionView's childViewEventPrefix

You can customize the event prefix for events that are forwarded  
through the collection view. To do this, set the childViewEventPrefix  
on the collection view. For more information on the childViewEventPrefix see  
["childview:\*" event bubbling from child views](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#childview-event-bubbling-from-child-views)

var CV = Marionette.CollectionView.extend({

childViewEventPrefix: "some:prefix"

});

var c = new CV({

collection: myCol

});

c.on("some:prefix:render", function(){

// child view was rendered

});

c.render();

The childViewEventPrefix can be provided in the view definition or  
in the constructor function call, to get a view instance.

CollectionView's childEvents

You can specify a childEvents hash or method which allows you to capture all bubbling childEvents without having to manually set bindings. The keys of the hash can either be a function or a string that is the name of a method on the collection view.

// childEvents can be specified as a hash...

var MyCollectionView = Marionette.CollectionView.extend({

// This callback will be called whenever a child is rendered or emits a `render` event

childEvents: {

render: function() {

console.log("a childView has been rendered");

}

}

});

// ...or as a function that returns a hash.

var MyCollectionView = Marionette.CollectionView.extend({

childEvents: function() {

return {

render: this.onChildRendered

}

});

This also works for custom events that you might fire on your child views.

// The child view fires a custom event, `show:message`

var ChildView = new Marionette.ItemView.extend({

events: {

'click .button': 'showMessage'

},

showMessage: function () {

console.log('The button was clicked.');

this.triggerMethod('show:message');

}

});

// The parent uses childEvents to catch that custom event on the child view

var ParentView = new Marionette.CollectionView.extend({

childView: ChildView,

childEvents: {

'show:message': function () {

console.log('The show:message event bubbled up to the parent.');

}

}

});

CollectionView's buildChildView

When a custom view instance needs to be created for the childView that  
represents a child, override the buildChildView method. This method  
takes three parameters and returns a view instance to be used as the  
child view.

buildChildView: function(child, ChildViewClass, childViewOptions){

// build the final list of options for the childView class

var options = \_.extend({model: child}, childViewOptions);

// create the child view instance

var view = new ChildViewClass(options);

// return it

return view;

},

CollectionView's addChild

The addChild method is responsible for rendering the childViews and adding them to the HTML for the collectionView instance. It is also responsible for triggering the events per ChildView. In most cases you should not override this method. However if you do want to short circuit this method, it can be accomplished via the following.

Marionette.CollectionView.extend({

addChild: function(child, ChildView, index){

if (child.shouldBeShown()) {

Marionette.CollectionView.prototype.addChild.apply(this, arguments);

}

}

});

CollectionView's reorderOnSort

This option is useful when you have performance issues when you resort your CollectionView.  
Without this option, your CollectionView will be completely re-rendered, which can be  
costly if you have a large number of elements or if your ChildViews are complex. If this option  
is activated, when you sort your Collection, there will be no re-rendering, only the DOM nodes  
will be reordered. This can be a problem if your ChildViews use their collection's index  
in their rendering. In this case, you cannot use this option as you need to re-render each  
ChildView.

If you combine this option with a [filter](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-filter) that changes the views that are  
to be displayed, reorderOnSort will be bypassed to render new children and remove those that are rejected by the filter.

CollectionView's emptyView

When a collection has no children, and you need to render a view other than  
the list of childViews, you can specify an emptyView attribute on your  
collection view.

var NoChildrenView = Marionette.ItemView.extend({

template: "#show-no-children-message-template"

});

Marionette.CollectionView.extend({

// ...

emptyView: NoChildrenView

});

CollectionView's getEmptyView

If you need the emptyView's class chosen dynamically, specify getEmptyView:

Marionette.CollectionView.extend({

// ...

getEmptyView: function() {

// custom logic

return NoChildrenView;

}

});

CollectionView's isEmpty

If you want to control when the empty view is rendered, you can override  
isEmpty:

Marionette.CollectionView.extend({

isEmpty: function(collection) {

// some logic to calculate if the view should be rendered as empty

return someBoolean;

}

});

CollectionView's emptyViewOptions

Similar to childView and childViewOptions, there is an emptyViewOptions property that will be passed to the emptyView constructor. It can be provided as an object literal or as a function.

If emptyViewOptions aren't provided the CollectionView will default to passing the childViewOptions to the emptyView.

var EmptyView = Marionette.ItemView({

initialize: function(options){

console.log(options.foo); // => "bar"

}

});

var CollectionView = Marionette.CollectionView({

emptyView: EmptyView,

emptyViewOptions: {

foo: "bar"

}

});

Callback Methods

There are several callback methods that can be provided on a  
CollectionView. If they are found, they will be called by the  
view's base methods. These callback methods are intended to be  
handled within the view definition directly.

onBeforeRender callback

A onBeforeRender callback will be called just prior to rendering  
the collection view.

Marionette.CollectionView.extend({

onBeforeRender: function(){

// do stuff here

}

});

onRender callback

After the view has been rendered, a onRender method will be called.  
You can implement this in your view to provide custom code for dealing  
with the view's el after it has been rendered:

Marionette.CollectionView.extend({

onRender: function(){

// do stuff here

}

});

onBeforeReorder callback

If reorderOnSort is set to true, onBeforeReorder will be called just  
prior to reordering the collection view.

Marionette.CollectionView.extend({

onBeforeReorder: function(){

// do stuff here

}

});

onReorder callback

If reorderOnSort is set to true, after the view has been reordered,  
a onReorder method will be called.

Marionette.CollectionView.extend({

onReorder: function(){

// do stuff here

}

});

onBeforeDestroy callback

This method is called just before destroying the view.

Marionette.CollectionView.extend({

onBeforeDestroy: function(){

// do stuff here

}

});

onDestroy callback

This method is called just after destroying the view.

Marionette.CollectionView.extend({

onDestroy: function(){

// do stuff here

}

});

onBeforeAddChild callback

This callback function allows you to know when a child / child view  
instance is about to be added to the collection view. It provides access to  
the view instance for the child that was added.

Marionette.CollectionView.extend({

onBeforeAddChild: function(childView){

// work with the childView instance, here

}

});

onAddChild callback

This callback function allows you to know when a child / child view  
instance has been added to the collection view. It provides access to  
the view instance for the child that was added.

Marionette.CollectionView.extend({

onAddChild: function(childView){

// work with the childView instance, here

}

});

onBeforeRemoveChild callback

This callback function allows you to know when a childView  
instance is about to be removed from the collectionView. It provides access to  
the view instance for the child that was removed.

Marionette.CollectionView.extend({

onBeforeRemoveChild: function(childView){

// work with the childView instance, here

}

});

onRemoveChild callback

This callback function allows you to know when a child / childView  
instance has been deleted or removed from the  
collection.

Marionette.CollectionView.extend({

onRemoveChild: function(childView){

// work with the childView instance, here

}

});

CollectionView Events

There are several events that will be triggered during the life  
of a collection view. Each of these events is called with the  
[Marionette.triggerMethod](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethod) function,  
which calls a corresponding "on{EventName}" method on the  
view instance (see [above](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#callback-methods)).

"before:render" event

Triggers just prior to the view being rendered. Also triggered as  
"collection:before:render" / onCollectionBeforeRender.

var MyView = Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("before:render", function(){

alert("the collection view is about to be rendered");

});

myView.render();

"render" event

A "render:collection" / onRenderCollection event will also be fired. This allows you to  
add more than one callback to execute after the view is rendered,  
and allows parent views and other parts of the application to  
know that the view was rendered.

var MyView = Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("render", function(){

alert("the collection view was rendered!");

});

myView.on("collection:rendered", function(){

alert("the collection view was rendered!");

});

myView.render();

"before:reorder" / "reorder" events

When reorderOnSort is set to true, these events are fired  
respectfully just prior/just after the reordering of the collection.

var MyView = Marionette.CollectionView.extend({...});

var myCol = new Backbone.Collection({ comparator: ... })

var myView = new MyView({ reorderOnSort: true });

myView.render();

myCol.comparator = function () { return this.get('foo'); };

myView.on("before:reorder", function(){

alert("the collection view is about to be reordered");

});

myView.on("reorder", function(){

alert("the collection view has been reordered following its collection");

});

myCol.sort()

"before:destroy" event

Triggered just before destroying the view. A "before:destroy:collection" /  
onBeforeDestroyCollection event will also be fired

var MyView = Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("before:destroy:collection", function(){

alert("the collection view is about to be destroyed");

});

myView.destroy();

"destroy" / "destroy:collection" event

Triggered just after destroying the view, both with corresponding  
method calls.

var MyView = Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("destroy:collection", function(){

alert("the collection view is now destroyed");

});

myView.destroy();

"before:add:child" / "add:child" event

The "before:add:child" event and corresponding onBeforeAddChild  
method are triggered just after creating a new childView instance for  
a child that was added to the collection, but before the  
view is rendered and added to the DOM.

The "add:child" event and corresponding onAddChild  
method are triggered after rendering the view and adding it to the  
view's DOM element.

var MyCV = Marionette.CollectionView.extend({

// ...

onBeforeAddChild: function(){

// ...

},

onAddChild: function(){

// ...

}

});

var cv = new MyCV({...});

cv.on("before:add:child", function(viewInstance){

// ...

});

cv.on("add:child", function(viewInstance){

// ...

});

"before:remove:child"

This is triggered after the childView instance has been  
removed from the collection, but before it has been destroyed.

cv.on("before:remove:child", function(childView){

// ...

});

"remove:child" event

Triggered after a childView instance has been destroyed and  
removed, when its child was deleted or removed from the  
collection.

cv.on("remove:child", function(viewInstance){

// ...

});

"childview:\*" event bubbling from child views

When a child view within a collection view triggers an  
event, that event will bubble up through the parent  
collection view with "childview:" prepended to the event  
name.

That is, if a child view triggers "do:something", the  
parent collection view will then trigger "childview:do:something".

// set up basic collection

var myModel = new MyModel();

var myCollection = new MyCollection();

myCollection.add(myModel);

var MyItemView = Backbone.Marionette.ItemView.extend({

triggers: {

'click button': 'do:something'

}

});

// get the collection view in place

var colView = new CollectionView({

collection: myCollection,

childView: MyItemView,

onChildviewDoSomething: function() {

alert("I said, 'do something!'");

}

});

colView.render();

Now, whenever the button inside the attached childView is clicked, an alert box  
will appear that says: I said, 'do something!'

It's also possible to attach the event manually using the usual  
view.on('childview:do:something').

before:render:collection event

The before:render:collection event is triggered before the collectionView's children have been rendered and buffered. It differs from the collectionsView's before:render in that it is **only** emitted if the collection is not empty.

render:collection event

The render:collection event is triggered after a collectionView's children have been rendered and buffered. It differs from the collectionViews's render event in that it happens **only** if the collection is not not empty.

CollectionView render

The render method of the collection view is responsible for  
rendering the entire collection. It loops through each of the  
children in the collection and renders them individually as an  
childView.

var MyCollectionView = Marionette.CollectionView.extend({...});

// all of the children views will now be rendered.

new MyCollectionView().render();

CollectionView: Automatic Rendering

The collection view binds to the "add", "remove" and "reset" events of the  
collection that is specified.

When the collection for the view is "reset", the view will call render on  
itself and re-render the entire collection.

When a model is added to the collection, the collection view will render that  
one model in to the collection of child views.

When a model is removed from a collection (or destroyed / deleted), the collection  
view will destroy and remove that model's child view.

CollectionView: Re-render Collection

If you need to re-render the entire collection, you can call the  
view.render method. This method takes care of destroying all of  
the child views that may have previously been opened.

CollectionView's attachHtml

By default the collection view will append the HTML of each ChildView  
into the element buffer, and then call jQuery's .append once at the  
end to move the HTML into the collection view's el.

You can override this by specifying an attachHtml method in your  
view definition. This method takes three parameters and has no return  
value.

Marionette.CollectionView.extend({

// The default implementation:

attachHtml: function(collectionView, childView, index){

if (collectionView.isBuffering) {

// buffering happens on reset events and initial renders

// in order to reduce the number of inserts into the

// document, which are expensive.

collectionView.\_bufferedChildren.splice(index, 0, childView);

}

else {

// If we've already rendered the main collection, append

// the new child into the correct order if we need to. Otherwise

// append to the end.

if (!collectionView.\_insertBefore(childView, index)){

collectionView.\_insertAfter(childView);

}

}

},

// Called after all children have been appended into the elBuffer

attachBuffer: function(collectionView, buffer) {

collectionView.$el.append(buffer);

},

// called on initialize and after attachBuffer is called

initRenderBuffer: function() {

this.elBuffer = document.createDocumentFragment();

}

});

The first parameter is the instance of the collection view that  
will receive the HTML from the second parameter, the current child  
view instance.

The third parameter, index, is the index of the  
model that this childView instance represents, in the collection  
that the model came from. This is useful for sorting a collection  
and displaying the sorted list in the correct order on the screen.

Overrides of attachHtml that don't take into account the element  
buffer will work fine, but won't take advantage of the 60x performance  
increase the buffer provides.

CollectionView's resortView

By default the CollectionView will maintain the order of its collection  
in the DOM. However on occasions the view may need to re-render to make this  
possible, for example if you were to change the comparator on the collection.  
By default CollectionView will call render when this happens, but there are  
cases where this may not be suitable. For instance when sorting the children  
in a CompositeView, you want to only render the internal collection.

var cv = new Marionette.CollectionView({

collection: someCollection,

resortView: function() {

// provide custom logic for rendering after sorting the collection

}

});

CollectionView's viewComparator

CollectionView allows for a custom viewComparator option if you want your CollectionView's children to be rendered with a different sort order than the underlying Backbone collection uses.

var cv = new MarionetteCollectionView({

collection: someCollection,

viewComparator: 'otherFieldToSortOn'

});

The viewComparator can take any of the acceptable Backbone.Collection [comparator formats](http://backbonejs.org/#Collection-comparator) -- a sortBy (pass a function that takes a single argument), as a sort (pass a comparator function that expects two arguments), or as a string indicating the attribute to sort by.

CollectionView's filter

CollectionView allows for a custom filter option if you want to prevent some of the  
underlying collection's models from being rendered as child views.  
The filter function takes a model from the collection and returns a truthy value if the child should be rendered,  
and a falsey value if it should not.

var cv = new Marionette.CollectionView({

childView: SomeChildView,

emptyView: SomeEmptyView,

collection: new Backbone.Collection([

{ value: 1 },

{ value: 2 },

{ value: 3 },

{ value: 4 }

]),

// Only show views with even values

filter: function (child, index, collection) {

return child.get('value') % 2 === 0;

}

});

// renders the views with values '2' and '4'

cv.render();

// change the filter

cv.filter = function (child, index, collection) {

return child.get('value') % 2 !== 0;

};

// renders the views with values '1' and '3'

cv.render();

// remove the filter

// note that using `delete cv.filter` will cause the prototype's filter to be used

// which may be undesirable

cv.filter = null;

// renders all views

cv.render();

CollectionView's children

The CollectionView uses [Backbone.BabySitter](https://github.com/marionettejs/backbone.babysitter)  
to store and manage its child views. This allows you to easily access  
the views within the collection view, iterate them, find them by  
a given indexer such as the view's model or collection, and more.

var cv = new Marionette.CollectionView({

collection: someCollection

});

cv.render();

// retrieve a view by model

var v = cv.children.findByModel(someModel);

// iterate over all of the views and process them

cv.children.each(function(view){

// process the `view` here

});

For more information on the available features and functionality of  
the .children, see the [Backbone.BabySitter documentation](https://github.com/marionettejs/backbone.babysitter).

CollectionView destroy

CollectionView implements a destroy method, which is called by the  
region managers automatically. As part of the implementation, the  
following are performed:

* unbind all listenTo events
* unbind all custom view events
* unbind all DOM events
* unbind all child views that were rendered
* remove this.el from the DOM
* call an onDestroy event on the view, if one is provided
* the CollectionView is returned

By providing an onDestroy event in your view definition, you can  
run custom code for your view that is fired after your view has been  
destroyed and cleaned up. This lets you handle any additional clean up  
code without having to override the destroy method.

Marionette.CollectionView.extend({

onDestroy: function() {

// custom cleanup or destroying code, here

}

});

Marionette.CompositeView

A CompositeView extends from CollectionView to be used as a  
composite view for scenarios where it should represent both a  
branch and leaf in a tree structure, or for scenarios where a  
collection needs to be rendered within a wrapper template. By default the  
CompositeView will maintain a sorted collection's order  
in the DOM. This behavior can be disabled by specifying {sort: false} on initialize.

Please see  
[the Marionette.CollectionView documentation](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html)  
for more information on available features and functionality.

Additionally, interactions with Marionette.Region  
will provide features such as onShow callbacks, etc. Please see  
[the Region documentation](http://marionettejs.com/docs/v2.4.1/marionette.region.html) for more information.

Example Usage: Tree View

For example, if you're rendering a tree-view control, you may  
want to render a collection view with a model and template so  
that it will show a parent child with children in the tree.

You can specify a modelView to use for the model. If you don't  
specify one, it will default to the Marionette.ItemView.

var CompositeView = Backbone.Marionette.CompositeView.extend({

template: "#leaf-branch-template"

});

new CompositeView({

model: someModel,

collection: someCollection

});

For more examples, see my blog post on  
[using the composite view.](http://lostechies.com/derickbailey/2012/04/05/composite-views-tree-structures-tables-and-more/)

Documentation Index

* [Composite Model template](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#composite-model-template)
* [CompositeView's childView](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#compositeviews-childview)
* [CompositeView's childViewContainer](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#compositeviews-childviewcontainer)
* [CompositeView's attachHtml](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#compositeviews-attachhtml)
* [Recursive By Default](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#recursive-by-default)
* [Model And Collection Rendering](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#model-and-collection-rendering)
* [Events And Callbacks](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#events-and-callbacks)
* [Organizing UI elements](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#organizing-ui-elements)
* [modelEvents and collectionEvents](http://marionettejs.com/docs/v2.4.1/marionette.compositeview.html#modelevents-and-collectionevents)

Composite Model template

When a CompositeView is rendered, the model will be rendered  
with the template that the view is configured with. You can  
override the template by passing it in as a constructor option:

new MyComp({

template: "#some-template"

});

The collection option is not passed to the template context by  
default. If your template needs access to the collection, you'll  
need to pass it via templateHelpers:

new MyComp({

template: "#some-template",

templateHelpers: function() {

return { items: this.collection.toJSON() };

}

})

CompositeView's childView

Each childView will be rendered using the childView's template. The CompositeView's  
template is rendered and the childView's templates are added to this.

var ChildView = Backbone.Marionette.ItemView.extend({});

var CompView = Backbone.Marionette.CompositeView.extend({

childView: ChildView

});

CompositeView's childViewContainer

By default the composite view uses the same attachHtml method that the  
collection view provides. This means the view will call jQuery's .append  
to move the HTML contents from the child view instance in to the collection  
view's el.

This is typically not very useful as a composite view will usually render  
a container DOM element in which the child views should be placed.

For example, if you are building a table view, and want to append each  
child from the collection in to the <tbody> of the table, you might  
do this with a template:

<script id="row-template" type="text/html">

<td><%= someData %></td>

<td><%= moreData %></td>

<td><%= stuff %></td>

</script>

<script id="table-template" type="text/html">

<table>

<thead>

<tr>

<th>Some Column</th>

<th>Another Column</th>

<th>Still More</th>

</tr>

</thead>

<!-- want to insert collection children, here -->

<tbody></tbody>

<tfoot>

<tr>

<td colspan="3">some footer information</td>

</tr>

</tfoot>

</table>

</script>

To get your childView instances to render within the <tbody> of this  
table structure, specify an childViewContainer in your composite view,  
like this:

var RowView = Backbone.Marionette.ItemView.extend({

tagName: "tr",

template: "#row-template"

});

var TableView = Backbone.Marionette.CompositeView.extend({

childView: RowView,

// specify a jQuery selector to put the `childView` instances into

childViewContainer: "tbody",

template: "#table-template"

});

This will put all of the childView instances into the <tbody> tag of  
the composite view's rendered template, correctly producing the table  
structure.

Alternatively, you can specify a function as the childViewContainer. This  
function needs to return a jQuery selector string, or a jQuery selector  
object.

var TableView = Backbone.Marionette.CompositeView.extend({

// ...

childViewContainer: function(){

return "#my-tbody"

}

});

Using a function allows for logic to be used for the selector. However,  
only one value can be returned. Upon returning the first value, it will  
be cached and that value will be used for the remainder of that view  
instance' lifecycle.

Alternatively, the childViewContainer can be supplied in the constructor  
function options:

var myComp = new Marionette.CompositeView({

// ...,

childViewContainer: "#my-tbody"

});

CompositeView's attachHtml

Sometimes the childViewContainer configuration is insufficient for  
specifying where the childView instance should be placed. If this is the  
case, you can override the attachHtml method with your own implementation.

For more information on this method, see the [CollectionView's documentation](http://marionettejs.com/docs/v2.4.1/marionette.collectionview.html#collectionviews-attachhtml).

CompositeView's childView container selection

The getChildViewContainer method is passed a second childView parameter which, when overridden, allows for a finer tuned container selection by being able to access the childView which is about to be appended to the containerView returned by getChildViewContainer.

Recursive By Default

The default rendering mode for a CompositeView assumes a  
hierarchical, recursive structure. If you configure a composite  
view without specifying an childView, you'll get the same  
composite view class rendered for each child in the collection.

Model And Collection Rendering

The model and collection for the composite view will re-render  
themselves under the following conditions:

* When the collection's "reset" event is fired, it will only re-render the collection within the composite, and not the wrapper template
* When the collection has a model added to it (the "add" event is fired), it will render that one child into the list
* When the collection has a model removed (the "remove" event is fired), it will remove that one child from the rendered list

As with item view instances, the composite view instance is passed as the  
third argument to the Renderer object's render method, which is  
useful in custom Renderer implementations.

Events And Callbacks

During the course of rendering a composite, several events will  
be triggered. These events are triggered with the [Marionette.triggerMethod](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethod)  
function, which calls a corresponding "on{EventName}" method on the view.

* "before:render:template" / onBeforeRenderTemplate - before the model has been rendered
* "render:template" / onRenderTemplate - after the model has been rendered
* "before:render:collection" / onBeforeRenderCollection - before the collection of models is rendered
* "render:collection" / onRenderCollection - after the collection of models has been rendered
* "before:render" / onBeforeRender - before anything has been rendered
* "render" / onRender - after everything has been rendered

Additionally, after the composite view has been rendered, an  
onRender method will be called. You can implement this in  
your view to provide custom code for dealing with the view's  
el after it has been rendered:

Backbone.Marionette.CompositeView.extend({

onRender: function(){

// do stuff here

}

});

Organizing UI elements

Similar to ItemView, you can organize the UI elements inside the  
CompositeView by specifying them in the UI hash. It should be  
noted that the elements that can be accessed via this hash are  
the elements that are directly rendered by the composite view  
template, not those belonging to the collection.

The UI elements will be accessible as soon as the composite view  
template is rendered (and before the collection is rendered),  
which means you can even access them in the onBeforeRender method.

modelEvents and collectionEvents

CompositeViews can bind directly to model events and collection events  
in a declarative manner:

Marionette.CompositeView.extend({

modelEvents: {

"change": "modelChanged"

},

collectionEvents: {

"add": "modelAdded"

}

});

For more information, see the [Marionette.View](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewmodelevents-and-viewcollectionevents)  
documentation.

Marionette Configuration

Marionette has a few globally configurable settings that will  
change how the system works. While many of these subjects are covered  
in other docs, this configuration doc should provide a list of the  
most common items to change.

Marionette.Deferred

Warning: deprecated

This feature is deprecated, and is scheduled to be removed in version 3 of Marionette. It is used to configure  
Marionette.Callbacks, which is also deprecated and scheduled to be removed in version 3. Instead of proxying  
the Deferred property on Marionette, use the native Promise object directly, and include a polyfill such as  
<https://github.com/jakearchibald/es6-promise> if you are supporting older browsers. $.Deferred can also be used, but  
it is not compliant with the ES6 Promise and is not recommended.

By default, Marionette makes use of Backbone.$.Deferred to create  
thenable objects.

Overriding Marionette.Deferred

If you are using Marionette without jQuery you must first shim Backbone.$.Deferred with a following object that adheres to these properties:

1. promise: a Promises/A+ thenable, or a function that returns one
2. resolve: a function that resolves the provided promise with a value

For example:

var deferred = Marionette.Deferred();

\_.result(deferred, 'promise').then(function (target) {

console.log("Hello, " + target + "!");

});

deferred.resolve("world"); // asynchronous "Hello, world!"

If you wish to use a specific promise library, you can override the default via:

Marionette.Deferred = myDeferredLib;

Marionette.Controller

Warning: deprecated. The Controller object is deprecated. Instead of using the Controller  
class with the AppRouter, you should specify your callbacks on a plain Javascript object.

A Controller is an object used in the Marionette Router. Controllers are where you store  
your Router's callbacks.

Documentation Index

* [Basic Use](http://marionettejs.com/docs/v2.4.1/marionette.controller.html#basic-use)
* [Destroying A Controller](http://marionettejs.com/docs/v2.4.1/marionette.controller.html#destroying-a-controller)
* [mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.controller.html#mergeoptions)
* [getOption](http://marionettejs.com/docs/v2.4.1/marionette.controller.html#getoption)
* [Prior Usage](http://marionettejs.com/docs/v2.4.1/marionette.controller.html#prior-usage)

Basic Use

A Marionette.Controller is intended to solely be used within the Router.

// Create a Controller, giving it the callbacks for our Router.

var MyController = Marionette.Controller.extend({

home: function() {},

profile: function() {}

});

// Instantiate it

var myController = new MyController();

// Pass it into the Router

var myRouter = new Marionette.AppRouter({

controller: myController,

appRoutes: {

"home": "home",

"profile": "profile"

}

});

mergeOptions

Merge keys from the options object directly onto the instance. This is the preferred way to access options  
passed into the Controller.

More information at [mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettemergeoptions)

getOption

Retrieve an object's attribute either directly from the object, or from the object's this.options, with this.options taking precedence.

More information [getOption](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettegetoption)

Destroying A Controller

Each Controller instance has a built in destroy method that handles  
unbinding all of the events that are directly attached to the controller  
instance, as well as those that are bound using the EventBinder from  
the controller.

Invoking the destroy method will trigger the "before:destroy" and "destroy" events and the  
corresponding onBeforeDestory and onDestroy method calls. These calls will be passed any arguments destroy  
was invoked with.

// define a controller with an onDestroy method

var MyController = Marionette.Controller.extend({

onBeforeDestroy: function(arg1, arg2){

// put custom code here, before destroying this controller

}

onDestroy: function(arg1, arg2){

// put custom code here, to destroy this controller

}

});

// create a new controller instance

var contr = new MyController();

// add some event handlers

contr.on("before:destroy", function(arg1, arg2){ ... });

contr.on("destroy", function(arg1, arg2){ ... });

contr.listenTo(something, "bar", function(){...});

// destroy the controller: unbind all of the

// event handlers, trigger the "destroy" event and

// call the onDestroy method

contr.destroy(arg1, arg2);

Prior Usage

Before Marionette 2.1, the Controller had another use, which was a general-purpose, white-label object. This was confusing given its other use within the Router, and its name, which carries so much meaning in the context of MVC frameworks.

As of v2.1, a new Class is available for your use: Marionette.Object. We recommend using Marionette.Object instead of Marionette.Controller in all situations outside of the Router.

Marionette functions

Marionette provides a set of utility / helper functions that are used to  
facilitate common behaviors throughout the framework. These functions may  
be useful to those that are building on top of Marionette, as they provide  
a way to get the same behaviors and conventions from your own code.

Documentation Index

* [Marionette.extend](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteextend)
* [Marionette.isNodeAttached](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteisnodeattached)
* [Marionette.mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettemergeoptions)
* [Marionette.getOption](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettegetoption)
* [Marionette.proxyGetOption](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteproxygetoption)
* [Marionette.triggerMethod](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethod)
* [Marionette.bindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettebindentityevents)
* [Marionette.triggerMethodOn](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethodon)
* [Marionette.bindEntityEvent](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettebindentityevents)
* [Marionette.unbindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteunbindentityevents)
* [Marionette.proxyBindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteproxybindentityevents)
* [Marionette.proxyUnbindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteproxyunbindentityevents)
* [Marionette.normalizeMethods](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettenormalizemethods)
* [Marionette.normalizeUIKeys](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettenormalizeuikeys)
* [Marionette.normalizeUIValues](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettenormalizeuivalues)
* [Marionette.actAsCollection](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionetteactascollection)

Marionette.extend

Backbone's extend function is a useful utility to have, and is used in  
various places in Marionette. To make the use of this method more consistent,  
Backbone's extend has been aliased to Marionette.extend. This allows  
you to get the extend functionality for your object without having to  
decide if you want to use Backbone.View or Backbone.Model or another  
Backbone object to grab the method from.

var Foo = function(){};

// use Marionette.extend to make Foo extendable, just like other

// Backbone and Marionette objects

Foo.extend = Marionette.extend;

// Now Foo can be extended to create a new class, with methods

var Bar = Foo.extend({

someMethod: function(){ ... }

// ...

});

// Create an instance of Bar

var b = new Bar();

Marionette.isNodeAttached

Determines whether the passed-in node is a child of the document or not.

var div = document.createElement('div');

Marionette.isNodeAttached(div);

// => false

$('body').append(div);

Marionette.isNodeAttached(div);

// => true

Marionette.mergeOptions

A handy function to pluck certain options and attach them directly to an instance.  
Most Marionette Classes, such as the Views, come with this method.

var MyView = ItemView.extend({

myViewOptions: ['color', 'size', 'country'],

initialize: function(options) {

this.mergeOptions(options, this.myViewOptions);

},

onRender: function() {

// The merged options will be attached directly to the prototype

this.$el.addClass(this.color);

}

});

Marionette.getOption

Retrieve an object's attribute either directly from the object, or from  
the object's this.options, with this.options taking precedence.

var M = Backbone.Model.extend({

foo: "bar",

initialize: function(attributes, options){

this.options = options;

var f = Marionette.getOption(this, "foo");

console.log(f);

}

});

new M(); // => "bar"

new M({}, { foo: "quux" }); // => "quux"

This is useful when building an object that can have configuration set  
in either the object definition or the object's constructor options.

Falsey values

The getOption function will return any falsey value from the options,  
other than undefined. If an object's options has an undefined value, it will  
attempt to read the value from the object directly.

For example:

var M = Backbone.Model.extend({

foo: "bar",

initialize: function(){

var f = Marionette.getOption(this, "foo");

console.log(f);

}

});

new M(); // => "bar"

var f;

new M({}, { foo: f }); // => "bar"

In this example, "bar" is returned both times because the second  
example has an undefined value for f.

Marionette.proxyGetOption

This method proxies Marionette.getOption so that it can be easily added to an instance.

Say you've written your own Pagination class and you always pass options to it.  
With proxyGetOption, you can easily give this class the getOption function.

\_.extend(Pagination.prototype, {

getFoo: function(){

return this.getOption("foo");

},

getOption: Marionette.proxyGetOption

});

Marionette.triggerMethod

Trigger an event and a corresponding method on the target object.

When an event is triggered, the first letter of each section of the  
event name is capitalized, and the word "on" is tagged on to the front  
of it. Examples:

* triggerMethod("render") fires the "onRender" function
* triggerMethod("before:destroy") fires the "onBeforeDestroy" function

All arguments that are passed to the triggerMethod call are passed along to both the event and the method, with the exception of the event name not being passed to the corresponding method.

triggerMethod("foo", bar) will call onFoo: function(bar){...})

Note that triggerMethod can be called on objects that do not have  
Backbone.Events mixed in to them. These objects will not have a trigger  
method, and no attempt to call .trigger() will be made. The on{Name}  
callback methods will still be called, though.

Marionette.triggerMethodOn

Invoke triggerMethod on a specific context.

This is useful when it's not clear that the object has triggerMethod defined. In the case of views, Marionette.View defines triggerMethod, but Backbone.View does not.

Marionette.triggerMethodOn(ctx, "foo", bar);

// will invoke `onFoo: function(bar){...})`

// will trigger "foo" on ctx

Marionette.bindEntityEvents

This method is used to bind a backbone "entity" (e.g. collection/model)  
to methods on a target object.

Backbone.View.extend({

modelEvents: {

"change:foo": "doSomething"

},

initialize: function(){

Marionette.bindEntityEvents(this, this.model, this.modelEvents);

},

doSomething: function(){

// the "change:foo" event was fired from the model

// respond to it appropriately, here.

}

});

The first parameter, target, must have the Backbone.Events module mixed in.

The second parameter is the entity (Backbone.Model, Backbone.Collection or  
any object that has Backbone.Events mixed in) to bind the events from.

The third parameter is a hash of { "event:name": "eventHandler" }  
configuration. Multiple handlers can be separated by a space. A  
function can be supplied instead of a string handler name.

Marionette.unbindEntityEvents

This method can be used to unbind callbacks from entities' (e.g. collection/model) events. It's  
the opposite of bindEntityEvents, described above. Consequently, the APIs are identical for each method.

// Just like the above example we bind our model events.

// This time, however, we unbind them on close.

Backbone.View.extend({

modelEvents: {

"change:foo": "doSomething"

},

initialize: function(){

Marionette.bindEntityEvents(this, this.model, this.modelEvents);

},

doSomething: function(){

// the "change:foo" event was fired from the model

// respond to it appropriately, here.

},

onClose: function() {

Marionette.unbindEntityEvents(this, this.model, this.modelEvents);

}

});

Marionette.proxyBindEntityEvents

This method proxies Marionette.bindEntityEvents so that it can easily be added to an instance.

Say you've written your own Pagination class and you want to easily listen to some entities events.  
With proxyBindEntityEvents, you can easily give this class the bindEntityEvents function.

\_.extend(Pagination.prototype, {

bindSomething: function() {

this.bindEntityEvents(this.something, this.somethingEvents)

},

bindEntityEvents: Marionette.proxyBindEntityEvents

});

Marionette.proxyUnbindEntityEvents

This method proxies Marionette.unbindEntityEvents so that it can easily be added to an instance.

It's the opposite of proxyBindEntityEvents, described above. Consequently, the APIs are identical for each method.

Say you've written your own Pagination class and you want to easily unbind callbacks from some entities events.  
With proxyUnbindEntityEvents, you can easily give this class the unbindEntityEvents function.

\_.extend(Pagination.prototype, {

bindSomething: function() {

this.bindEntityEvents(this.something, this.somethingEvents)

},

unbindSomething: function() {

this.unbindEntityEvents(this.something, this.somethingEvents)

},

bindEntityEvents: Marionette.proxyBindEntityEvents,

unbindEntityEvents: Marionette.proxyUnbindEntityEvents

});

Marionette.normalizeMethods

Receives a hash of event names and functions and/or function names, and returns the  
same hash with the function names replaced with the function references themselves.

This function is attached to the Marionette.View prototype by default. To use it from non-View classes you'll need to attach it yourself.

var View = Marionette.ItemView.extend({

initialize: function() {

this.someFn = function() {};

this.someOtherFn = function() {};

var hash = {

eventOne: "someFn", // This will become a reference to `this.someFn`

eventTwo: this.someOtherFn

};

this.normalizedHash = this.normalizeMethods(hash);

}

});

Marionette.normalizeUIKeys

This method allows you to use the @ui. syntax within a given key for triggers and events hashes. It  
swaps the @ui. reference with the associated selector.

var hash = {

'click @ui.list': 'myCb'

};

var ui = {

'list': 'ul'

};

// This sets 'click @ui.list' to be 'click ul' in the newHash object

var newHash = Marionette.normalizeUIKeys(hash, ui);

Marionette.normalizeUIValues

This method allows you to use the @ui. syntax within a given hash value (for example region hashes). It  
swaps the @ui. reference with the associated selector.

var hash = {

'foo': '@ui.bar'

};

var ui = {

'bar': '.quux'

};

// This sets 'foo' to be '.quux' in the newHash object

var newHash = Marionette.normalizeUIValues(hash, ui);

Marionette.actAsCollection

Utility function for mixing in underscore collection behavior to an object.

It works by taking an object and a property field, in this example 'list',  
and appending collection functions to the object so that it can  
delegate collection calls to its list.

Object Literal

var obj = {

list: [1, 2, 3]

}

Marionette.actAsCollection(obj, 'list');

var double = function(v){ return v\*2};

console.log(obj.map(double)); // [2, 4, 6]

Function Prototype

var Func = function(list) {

this.list = list;

};

Marionette.actAsCollection(Func.prototype, 'list');

var func = new Func([1,2,3]);

var double = function(v){ return v\*2};

console.log(func.map(double)); // [2, 4, 6]

The first parameter is the object that will delegate underscore collection methods.

The second parameter is the object field that will hold the list.

# Marionette.ItemView

An ItemView is a view that represents a single item. That item may be a  
Backbone.Model or may be a Backbone.Collection. Whichever it is though, it  
will be treated as a single item.

ItemView extends directly from Marionette.View. Please see  
[the Marionette.View documentation](http://marionettejs.com/docs/v2.4.1/marionette.view.html)  
for more information on available features and functionality.

Additionally, interactions with Marionette.Region  
will provide features such as onShow callbacks, etc. Please see  
[the Region documentation](http://marionettejs.com/docs/v2.4.1/marionette.region.html) for more information.

Documentation Index

* [ItemView render](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#itemview-render)
* [Rendering A Collection In An ItemView](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#rendering-a-collection-in-an-itemview)
* [Template-less ItemView](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#template-less-itemview)
* [Events and Callback Methods](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#events-and-callback-methods)
  + ["before:render" / onBeforeRender event](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#beforerender--onbeforerender-event)
  + ["render" / onRender event](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#render--onrender-event)
  + ["before:destroy" / onBeforeDestroy event](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#beforedestroy--onbeforedestroy-event)
  + ["destroy" / onDestroy event](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#destroy--ondestroy-event)
* [ItemView serializeData](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#itemview-serializedata)
* [Organizing ui elements](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#organizing-ui-elements)
* [modelEvents and collectionEvents](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#modelevents-and-collectionevents)

ItemView render

Unlike Backbone Views, all Marionette views come with a powerful render method.  
In fact, the primary differences between the views are the differences in their  
render methods. It goes without saying that it is unwise to override the render  
method of any Marionette view. Instead, you should use the [onBeforeRender and onRender callbacks](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#events-and-callback-methods)  
to layer in additional functionality to the rendering of your view.

The ItemView defers to the Marionette.Renderer object to do the actual  
rendering of the template.

The item view instance is passed as the third argument to the  
Renderer object's render method, which is useful in custom  
Renderer implementations.

You should provide a template attribute on the item view, which  
will be either a jQuery selector:

var MyView = Backbone.Marionette.ItemView.extend({

template: "#some-template"

});

new MyView().render();

.. or a function taking a single argument: the object returned by [ItemView.serializeData](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html#itemview-serializedata):

var my\_template\_html = '<div><%= args.name %></div>'

var MyView = Backbone.Marionette.ItemView.extend({

template : function(serialized\_model) {

var name = serialized\_model.name;

return \_.template(my\_template\_html)({

name : name,

some\_custom\_attribute : some\_custom\_key

});

}

});

new MyView().render();

Note that using a template function allows passing custom arguments into the *.template function and allows for more control over how the* .template function is called.

For more information on the \_.template function see the [Underscore docs](http://underscorejs.org/#template).

Rendering A Collection In An ItemView

While the most common way to render a Backbone.Collection is to use  
a CollectionView or CompositeView, if you just need to render a  
simple list that does not need a lot of interaction, it does not  
always make sense to use these. A Backbone.Collection can be  
rendered with a simple ItemView, using the templates to iterate  
over an items array.

<script id="some-template" type="text/html">

<ul>

<% \_.each(items, function(item){ %>

<li> <%= item.someAttribute %> </li>

<% }); %>

</ul>

</script>

The important thing to note here, is the use of items as the  
variable to iterate in the \_.each call. This will always be the  
name of the variable that contains your collection's items.

Then, from JavaScript, you can define and use an ItemView with this  
template, like this:

var MyItemsView = Marionette.ItemView.extend({

template: "#some-template"

});

var view = new MyItemsView({

collection: someCollection

});

// show the view via a region or calling the .render method directly

Rendering this view will convert the someCollection collection in to  
the items array for your template to use.

For more information on when you would want to do this, and what options  
you have for retrieving an individual item that was clicked or  
otherwise interacted with, see the blog post on  
[Getting The Model For A Clicked Element](http://lostechies.com/derickbailey/2011/10/11/backbone-js-getting-the-model-for-a-clicked-element/).

Template-less ItemView

An ItemView can be attached to existing elements as well. The primary benefit of this is to attach behavior and events to static content that has been rendered by your server (typically for SEO purposes). To set up a template-less ItemView, your template attribute must be false.

<div id="my-element">

<p>Hello World</p>

<button class="my-button">Click Me</button>

</div>

var MyView = Marionette.ItemView.extend({

el: '#my-element',

template: false,

ui: {

paragraph: 'p',

button: '.my-button'

},

events: {

'click @ui.button': 'clickedButton'

},

clickedButton: function() {

console.log('I clicked the button!');

}

});

var view = new MyView();

view.render();

view.ui.paragraph.text(); // returns 'Hello World'

view.ui.button.trigger('click'); // logs 'I clicked the button!'

Another use case is when you want to attach a Marionette.ItemView to a SVG graphic or canvas element, to provide a uniform view layer interface to non-standard DOM nodes. By not having a template this allows you to also use a view on pre-rendered DOM nodes, such as complex graphic elements.

Events and Callback Methods

There are several events and callback methods that are called  
for an ItemView. These events and methods are triggered with the  
[Marionette.triggerMethod](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethod) function, which  
triggers the event and a corresponding "on{EventName}" method.

"before:render" / onBeforeRender event

Triggered before an ItemView is rendered.

Backbone.Marionette.ItemView.extend({

onBeforeRender: function(){

// set up final bits just before rendering the view's `el`

}

});

"render" / onRender event

Triggered after the view has been rendered.  
You can implement this in your view to provide custom code for dealing  
with the view's el after it has been rendered.

Backbone.Marionette.ItemView.extend({

onRender: function(){

// manipulate the `el` here. it's already

// been rendered, and is full of the view's

// HTML, ready to go.

}

});

"before:destroy" / onBeforeDestroy event

Triggered just prior to destroying the view, when the view's destroy()  
method has been called.

Backbone.Marionette.ItemView.extend({

onBeforeDestroy: function(){

// manipulate the `el` here. it's already

// been rendered, and is full of the view's

// HTML, ready to go.

}

});

"destroy" / onDestroy event

Triggered just after the view has been destroyed.

Backbone.Marionette.ItemView.extend({

onDestroy: function(){

// custom destroying and cleanup goes here

}

});

ItemView serializeData

Item views will serialize a model or collection, by default, by  
calling .toJSON on either the model or collection. If both a model  
and collection are attached to an item view, the model will be used  
as the data source. The results of the data serialization will be passed to the template  
that is rendered.

If the serialization is a model, the results are passed in directly:

var myModel = new MyModel({foo: "bar"});

new MyItemView({

template: "#myItemTemplate",

model: myModel

});

MyItemView.render();

<script id="myItemTemplate" type="template">

Foo is: <%= foo %>

</script>

If the serialization is a collection, the results are passed in as an  
items array:

var myCollection = new MyCollection([{foo: "bar"}, {foo: "baz"}]);

new MyItemView({

template: "#myCollectionTemplate",

collection: myCollection

});

MyItemView.render();

<script id="myCollectionTemplate" type="template">

<% \_.each(items, function(item){ %>

Foo is: <%= foo %>

<% }); %>

</script>

If you need custom serialization for your data, you can provide a  
serializeData method on your view. It must return a valid JSON  
object, as if you had called .toJSON on a model or collection.

Backbone.Marionette.ItemView.extend({

serializeData: function(){

return {

"some attribute": "some value"

}

}

});

Organizing UI Elements

As documented in [Marionette.View](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewbindentityevents), you can specify a ui hash in your view that  
maps UI elements by their jQuery selectors. This is especially useful if you access the  
same UI element more than once in your view's code. Instead of  
duplicating the selector, you can simply reference it by  
this.ui.elementName:

You can also use the ui hash values from within events and trigger keys using the "@ui.elementName": syntax

Backbone.Marionette.ItemView.extend({

tagName: "tr",

ui: {

checkbox: "input[type=checkbox]"

},

onRender: function() {

if (this.model.get('selected')) {

this.ui.checkbox.addClass('checked');

}

}

});

modelEvents and collectionEvents

ItemViews can bind directly to model events and collection events  
in a declarative manner:

Marionette.ItemView.extend({

modelEvents: {

"change": "modelChanged"

},

collectionEvents: {

"add": "modelAdded"

}

});

For more information, see the [Marionette.View](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewmodelevents-and-viewcollectionevents) documentation.

# Marionette.LayoutView

A LayoutView is a hybrid of an ItemView and a collection of Region objects. They  
are ideal for rendering application layouts with multiple sub-regions  
managed by specified region managers.

A layoutView can also act as a composite-view to aggregate multiple  
views and sub-application areas of the screen allowing applications to  
attach multiple region managers to dynamically rendered HTML.

You can create complex views by nesting layoutView managers within Regions.

For a more in-depth discussion on LayoutViews, see the blog post  
[Manage Layouts And Nested Views With Backbone.Marionette](http://lostechies.com/derickbailey/2012/03/22/managing-layouts-and-nested-views-with-backbone-marionette/)

Please see  
[the Marionette.ItemView documentation](http://marionettejs.com/docs/v2.4.1/marionette.itemview.html)  
for more information on available features and functionality.

Additionally, interactions with Marionette.Region  
will provide features such as onShow callbacks, etc. Please see  
[the Region documentation](http://marionettejs.com/docs/v2.4.1/marionette.region.html) for more information.

## Documentation Index

* [Basic Usage](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#basic-usage)
* [Region Options](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#region-options)
* [LayoutView.childEvents](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#layoutview-childevents)
* [Specifying Regions As A Function](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#specifying-regions-as-a-function)
* [Overriding the default RegionManager](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#overriding-the-default-regionmanager)
* [Region Availability](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#region-availability)
* [Re-Rendering A LayoutView](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#re-rendering-a-layoutview)
  + [Avoid Re-Rendering The Entire LayoutView](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#avoid-re-rendering-the-entire-layoutview)
* [Nested LayoutViews And Views](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#nested-layoutviews-and-views)
  + [Efficient Nested View Structures](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#efficient-nested-view-structures)
    - [Use of the attach Event](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#use-of-the-attach-event)
* [Destroying A LayoutView](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#destroying-a-layoutview)
* [Custom Region Class](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#custom-region-class)
* [Adding And Removing Regions](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#adding-and-removing-regions)
* [Region Naming](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html#region-naming)

## Basic Usage

The LayoutView extends directly from ItemView and adds the ability  
to specify regions which become Region instances that are attached  
to the layoutView.

<script id="layout-view-template" type="text/template">

<section>

<navigation id="menu">...</navigation>

<article id="content">...</article>

</section>

</script>

var AppLayoutView = Backbone.Marionette.LayoutView.extend({

template: "#layout-view-template",

regions: {

menu: "#menu",

content: "#content"

}

});

var layoutView = new AppLayoutView();

layoutView.render();

Once you've rendered the layoutView, you now have direct access  
to all of the specified regions as region managers.

layoutView.getRegion('menu').show(new MenuView());

layoutView.getRegion('content').show(new MainContentView());

There are also helpful shortcuts for more concise syntax.

layoutView.showChildView('menu', new MenuView());

layoutView.showChildView('content', new MainContentView());

### Region Options

A LayoutView can take a regions hash that allows you to specify regions per LayoutView instance.

new Marionette.LayoutView({

regions: {

"cat": ".doge",

"wow": {

selector: ".such",

regionClass: Coin

}

}

})

### LayoutView childEvents

You can specify a childEvents hash or method which allows you to capture all  
bubbling childEvents without having to manually set bindings.

The keys of the hash can either be a function or a string  
that is the name of a method on the layout view.

The function is called in the context of the view. The first parameter is  
the child view, which emitted the event, the remainder are the arguments  
associated with the event.

// childEvents can be specified as a hash...

var MyLayoutView = Marionette.LayoutView.extend({

// This callback will be called whenever a child is rendered or emits a `render` event

childEvents: {

render: function(childView) {

console.log("a childView has been rendered");

}

}

});

// ...or as a function that returns a hash.

var MyLayoutView = Marionette.LayoutView.extend({

childEvents: function() {

return {

render: this.onChildRender

}

},

onChildRender: function(childView) {

}

});

This also works for custom events that you might fire on your child views.

// The child view fires a custom event, `show:message`

var ChildView = new Marionette.ItemView.extend({

events: {

'click .button': 'showMessage'

},

showMessage: function (e) {

console.log('The button was clicked.');

this.triggerMethod('show:message', msg);

}

});

// The parent uses childEvents to catch that custom event on the child view

var ParentView = new Marionette.LayoutView.extend({

childEvents: {

'show:message': function (childView, msg) {

console.log('The show:message event bubbled up to the parent.');

}

},

// Alternatively we can use the trigger notation with childview: as the

// prefix

onChildviewShowMessage: function (childView, msg) {

console.log('The show:message event bubbled up to the parent.');

}

});

### Specifying Regions As A Function

Regions can be specified on a LayoutView using a function that returns  
an object with the region definitions. The returned object follow the  
same rules for defining a region, as outlined above.

Marionette.LayoutView.extend({

// ...

regions: function(options){

return {

fooRegion: "#foo-element"

};

},

// ...

});

Note that the function receives the view's options arguments that  
were passed in to the view's constructor. this.options is not yet  
available when the regions are first initialized, so the options  
must be accessed through this parameter.

### Overriding the default RegionManager

If you need the RegionManager's class chosen dynamically, specify getRegionManager:

Marionette.LayoutView.extend({

// ...

getRegionManager: function() {

// custom logic

return new MyRegionManager();

}

This can be useful if you want to attach LayoutView's regions to your own instance of  
RegionManager.

## Region Availability

Any defined regions within a layoutView will be available to the  
View or any calling code immediately after instantiating the  
View. This allows a View to be attached to an existing  
DOM element in an HTML page, without the need to call a render  
method or anything else, to create the regions.

However, a region will only be able to populate itself if the  
View has access to the elements specified within the region  
definitions. That is, if your view has not yet rendered, your  
regions may not be able to find the element that you've  
specified for them to manage. In that scenario, using the  
region will result in no changes to the DOM.

## Re-Rendering A LayoutView

A layoutView can be rendered as many times as needed, but renders  
after the first one behave differently than the initial render.

The first time a layoutView is rendered, nothing special happens. It just  
delegates to the ItemView prototype to do the render. After the  
first render has happened, though, the render function is modified to  
account for re-rendering with regions in the layoutView.

After the first render, all subsequent renders will force every  
region to be emptied by calling the empty method on them. This will  
force every view in the region, and sub-views if any, to be destroyed  
as well. Once the regions are emptied, the regions will also be  
reset so that they are no longer referencing the element of the previous  
layoutView render.

Then after the layoutView is finished re-rendering itself,  
showing a view in the layoutView's regions will cause the regions to attach  
themselves to the new elements in the layoutView.

### Avoid Re-Rendering The Entire LayoutView

There are times when re-rendering the entire layoutView is necessary. However,  
due to the behavior described above, this can cause a large amount of  
work to be needed in order to fully restore the layoutView and all of the  
views that the layoutView is displaying.

Therefore, it is suggested that you avoid re-rendering the entire  
layoutView unless absolutely necessary. Instead, if you are binding the  
layoutView's template to a model and need to update portions of the layoutView,  
you should listen to the model's "change" events and only update the  
necessary DOM elements.

## Nested LayoutViews And Views

Since the LayoutView extends directly from ItemView, it  
has all of the core functionality of an item view. This includes  
the methods necessary to be shown within an existing region manager.

In the following example, we will use the Application's Regions  
as the base of a deeply nested view structure.

// Create an Application

var MyApp = new Marionette.Application();

// Add a region

MyApp.addRegions({

main: "main"

});

// Create a new LayoutView

var layoutView = new Marionette.LayoutView({

// This option removes the layoutView from

// the DOM before destroying the children

// preventing repaints as each option is removed.

// However, it makes it difficult to do close animations

// for a child view (false by default)

destroyImmediate: true

});

// Lastly, show the LayoutView in the App's mainRegion

MyApp.getRegion('main').show(layoutView);

You can nest LayoutViews as deeply as you want. This provides for a well organized,  
nested view structure.

For example, to nest 3 layouts:

var layout1 = new Layout1();

var layout2 = new Layout2();

var layout3 = new Layout3();

MyApp.getRegion('main').show(layout1);

layout1.showChildView('region1', layout2);

layout2.showChildView('region2', layout3);

### Efficient Nested View Structures

The above example works great, but it causes three separate paints: one for each layout that's being  
shown. Marionette provides a simple mechanism to infinitely nest views in a single paint: just render all  
of the children in the onBeforeShow callback.

var ParentLayout = Marionette.LayoutView.extend({

onBeforeShow: function() {

this.showChildView('header', new HeaderView());

this.showChildView('footer', new FooterView());

}

});

myRegion.show(new ParentLayout());

In this example, the doubly-nested view structure will be rendered in a single paint.

This system is recursive, so it works for any deeply nested structure. The child views  
you show can render their own child views within their onBeforeShow callbacks!

#### Use of the attach event

Often times you need to know when your views in the view tree have been attached to the document,  
like when using certain jQuery plugins. The attach event, and associated onAttach callback, are perfect for this  
use case. Start with a Region that's a child of the document and show any LayoutView in it: every view in the tree  
(including the parent LayoutView) will have the attach event triggered on it when they have been  
attached to the document.

Note that inefficient tree rendering will cause the attach event to be fired multiple times. This  
situation can occur if you render the children views after the parent has been rendered, such as using  
onShow to render children. As a rule of thumb, most of the time you'll want to render any nested views in  
the onBeforeShow callback.

## Destroying A LayoutView

When you are finished with a layoutView, you can call the  
destroy method on it. This will ensure that all of the region managers  
within the layoutView are destroyed correctly, which in turn  
ensures all of the views shown within the regions are destroyed correctly.

If you are showing a layoutView within a parent region manager, replacing  
the layoutView with another view or another layoutView will destroy the current  
one, the same it will destroy a view.

All of this ensures that layoutViews and the views that they  
contain are cleaned up correctly.

When calling destroy on a layoutView, the layoutView will be returned. This can be useful for  
chaining.

## Custom Region Class

If you have the need to replace the Region with a region class of  
your own implementation, you can specify an alternate class to use  
with the regionClass property of the LayoutView.

var MyLayoutView = Backbone.Marionette.LayoutView.extend({

regionClass: SomeCustomRegion

});

You can also specify custom Region classes for each region:

var AppLayoutView = Backbone.Marionette.LayoutView.extend({

template: "#layout-view-template",

regionClass: SomeDefaultCustomRegion,

regions: {

menu: {

selector: "#menu",

regionClass: CustomRegionClassReference

},

content: {

selector: "#content",

regionClass: CustomRegionClass2Reference

}

}

});

## Adding And Removing Regions

Regions can be added and removed as needed, in a  
LayoutView instance. Use the following methods:

* addRegion
* addRegions
* removeRegion

addRegion:

var layoutView = new MyLayoutView();

// ...

layoutView.addRegion("foo", "#foo");

layoutView.getRegion('foo').show(new someView());

addRegions:

var layoutView = new MyLayoutView();

// ...

// Object literal

layoutView.addRegions({

foo: "#foo",

bar: "#bar"

});

// Or, function that returns an object literal

layoutView.addRegions(function() {

return {

baz: "#baz",

quux: "#quux"

};

});

removeRegions:

var layoutView = new MyLayoutView();

// ...

layoutView.removeRegion("foo");

Any region can be removed, whether it was defined  
in the regions attribute of the region definition,  
or added later.

For more information on using these methods, see  
the regionManager documentation.

## Region Naming

A LayoutViews' Regions are attached directly to the LayoutView instance with the name of the region  
as the key and the region itself as the value. Because of this, you need to be careful  
to avoid conflicts with existing properties on the LayoutView when you name your Region.

The prototype chain of LayoutViews is:

Backbone.View > Marionette.View > Marionette.ItemView > Marionette.LayoutView

Consequently, every property on each of those Classes must be avoided as Region names. The most  
common issue people run into is trying to name their Region "attributes". Be aware  
that you are **not** able to do this.

The following is an abbreviated list of other names that can't be used as Region names. For a more  
complete list refer to the API documentation for each Class on the prototype chain:

* attributes
* constructor
* regionClass
* render
* destroy
* addRegion
* addRegions
* removeRegion

Note: this is a known issue that is flagged for being fixed in v2

# Marionette.Module

Marionette Modules allow you to create modular encapsulated logic.  
They can be used to split apart large applications into multiple files,  
and to build individual components of your app.

## Documentation Index

* [Basic Usage](http://marionettejs.com/docs/v2.4.1/marionette.module.html#basic-usage)
* [Module Definitions](http://marionettejs.com/docs/v2.4.1/marionette.module.html#module-definitions)
  + [Callback Function Definition](http://marionettejs.com/docs/v2.4.1/marionette.module.html#callback-function-definition)
  + [Object Literal Definition](http://marionettejs.com/docs/v2.4.1/marionette.module.html#object-literal-definition)
* [Module Classes](http://marionettejs.com/docs/v2.4.1/marionette.module.html#module-classes)
* [Defining Sub-Modules](http://marionettejs.com/docs/v2.4.1/marionette.module.html#defining-sub-modules)
* [Starting and Stopping Modules](http://marionettejs.com/docs/v2.4.1/marionette.module.html#starting-and-stopping-modules)
* [Starting Modules](http://marionettejs.com/docs/v2.4.1/marionette.module.html#starting-modules)
  + [Start Events](http://marionettejs.com/docs/v2.4.1/marionette.module.html#start-events)
  + [Preventing Auto-Start Of Modules](http://marionettejs.com/docs/v2.4.1/marionette.module.html#preventing-auto-start-of-modules)
  + [Starting Sub-Modules With Parent](http://marionettejs.com/docs/v2.4.1/marionette.module.html#starting-sub-modules-with-parent)
* [Stopping Modules](http://marionettejs.com/docs/v2.4.1/marionette.module.html#stopping-modules)
  + [Stop Events](http://marionettejs.com/docs/v2.4.1/marionette.module.html#stop-events)
* [Module Initializers (deprecated)](http://marionettejs.com/docs/v2.4.1/marionette.module.html#module-initializers)
* [Module Finalizers (deprecated)](http://marionettejs.com/docs/v2.4.1/marionette.module.html#module-finalizers)

## Basic Usage

A module is defined directly from an Application object. To create a module all  
you need to do is give it a name.

var MyApp = new Backbone.Marionette.Application();

// Creates a new module named "MyModule"

var myModule = MyApp.module("MyModule");

myModule === MyApp.MyModule; // => true

Modules cannot be overwritten once they are created. Subsequent  
calls to module with the same name argument will not create  
a new module, but instead return the already-created instance.

var MyApp = new Backbone.Marionette.Application();

// Instantiates a new Marionette.Module

var myModule = MyApp.module("MyModule");

// Returns the module you just created

var theSameModule = MyApp.module("MyModule");

## Module Definitions

You can provide a definition for your module when you instantiate it.  
Definitions can either be a callback function or an object literal.

### Callback Function Definition

The callback function definition will be invoked immediately on calling  
the module method.

It will receive 6 parameters, in this order:

* The module itself
* The Application object
* Backbone
* Backbone.Marionette
* jQuery
* Underscore
* Any custom arguments

Within the callback you can attach both private and public  
functions and data directly to your module.

MyApp.module("MyModule", function(MyModule, MyApp, Backbone, Marionette, $, \_){

// The context of the function is also the module itself

this === MyModule; // => true

// Private Data And Functions

// --------------------------

var myData = "this is private data";

var myFunction = function(){

console.log(myData);

}

// Public Data And Functions

// -------------------------

MyModule.someData = "public data";

MyModule.someFunction = function(){

console.log(MyModule.someData);

}

});

console.log(MyApp.MyModule.someData); //=> public data

MyApp.MyModule.someFunction(); //=> public data

#### Additional Arguments

You can provide additional arguments to the definition  
function, allowing you to import 3rd party libraries  
and other resources that you want to have locally scoped to  
your module.

Pass the additional arguments after the  
definition itself in the call to module.

MyApp.module("MyModule", function(MyModule, MyApp, Backbone, Marionette, $, \_, Lib1, Lib2, LibEtc){

// Lib1 === LibraryNumber1;

// Lib2 === LibraryNumber2;

// LibEtc === LibraryNumberEtc;

}, LibraryNumber1, LibraryNumber2, LibraryNumberEtc);

#### Splitting A Module Definition Apart

Sometimes a module definition can become quite long. You can split  
apart the definition by making subsequent calls to the module  
function.

This can used to split the definition of your module  
across multiple files.

MyApp.module("MyModule", function(MyModule){

MyModule.definition1 = true;

});

// The following could be in a separate file

MyApp.module("MyModule", function(MyModule){

MyModule.definition2 = true;

});

MyApp.MyModule.definition1; //=> true

MyApp.MyModule.definition2; //=> true

### Object Literal Definition

The object literal definition of a module allows for more flexibility  
than the callback method. It allows you to, for instance, specify  
a custom class for your module.

Through the object literal definition you can still set a definition  
function through the define property.

MyApp.module("MyModule", {

define: function(MyModule, MyApp, Backbone, Marionette, $, \_) {

// Define your module here

}

});

#### Specifying a Custom Module Class

One of the more useful features of the object literal definition is specifying a custom  
module class. You can make a new class using the extend function.

var CustomModule = Marionette.Module.extend({

// Custom module properties

});

MyApp.module("Foo", {

moduleClass: CustomModule,

define: function() {} // You can still use the definition function on custom modules

});

When moduleClass is omitted Marionette will default to instantiating a new Marionette.Module.

#### Initialize Function

Modules have an initialize function which is immediately called when the Module is invoked. You can think of the initialize function as an extension of the constructor.

The initialize function is only available through the object literal definition of a Module.

MyApp.module("Foo", {

startWithParent: false,

initialize: function( moduleName, app, options ) {

this.someProperty = 'someValue';

},

// You can still set a define function

define: function( Foo ) {

console.log( this.someProperty ); // Logs 'someValue'

}

});

The initialize function is passed the same arguments as the constructor.

* The moduleName
* The app
* The object literal definition of the Module itself (which allows you to pass arbitrary values to your Module)

MyApp.module("Foo", {

initialize: function( moduleName, app, options ) {

console.log( options.someVar ); // Logs 'someString'

},

someVar: 'someString'

});

The initialize function is distinct from the define function. The primary difference between the two is that initialize is on the prototype chain, whereas define is not. What this means is that initialize can be inherited.

var CustomModule = Marionette.Module.extend({

define: function() {}, // This is not inherited and will never be called

initialize: function() {} // This, on the other hand, will be inherited

});

## Module Classes

Module classes can be used as an alternative to the define pattern.

The extend function of a Module is identical to the extend functions on other Backbone and Marionette classes. This allows module lifecycle events like onStart and onStop to be called directly.

var FooModule = Marionette.Module.extend({

startWithParent: false,

initialize: function(options, moduleName, app) {

},

onStart: function(options) {

},

onStop: function(options) {

},

});

MyApp.module("Foo", FooModule);

If all of the module's functionality is defined inside its class, then the class can be passed in directly. MyApp.module("Foo", FooModule)

## Defining Sub-Modules

Sub-Modules (or 'child' Modules) can be defined in a single call by passing  
a period-separated list of Modules to be created.

MyApp.module("Parent.Child.GrandChild");

MyApp.Parent; // => a valid module object

MyApp.Parent.Child; // => a valid module object

MyApp.Parent.Child.GrandChild; // => a valid module object

When defining sub-modules using the dot-notation, the  
parent modules do not need to exist; they'll be created for you. If a parent  
has already been instantiated then that instance will be used.

## Accessing Modules

Although modules are attached directly to the Application instance we don't recommend accessing them this way. Instead,  
use the .module() function to access your modules.

Let's look at two examples of accessing a module named MyModule.Submodule.

// Not recommended

var myModule = App.MyModule.Submodule;

// Recommended

var MyModule = App.module('MyModule.Submodule');

## Starting And Stopping Modules

Modules can be started and stopped independently of the application and  
of each other. This allows them to be loaded asynchronously, and also allows  
them to be shut down when they are no longer needed.

This also facilitates unit testing of modules as you can start only the  
module that you need in your tests.

## Starting Modules

Modules will, by default, start with the parent application. They also have a  
.start function that can be used to start a stopped module, or a module that's  
been configured to start independently from its parent.

In this example, the module will exhibit the default behavior and start automatically  
with the parent application object's start call:

MyApp = new Backbone.Marionette.Application();

MyApp.module("Foo", function(){

// module code goes here

});

MyApp.start();

Note that modules loaded after the MyApp.start() call will be  
immediately started.

### Start Events

When starting a module, a "before:start" event will be triggered prior  
to any of the initializers being run. A "start" event will then be  
triggered after they have been run.

var mod = MyApp.module("MyMod");

mod.on("before:start", function(){

// do stuff before the module is started

});

mod.on("start", function(){

// do stuff after the module has been started

});

#### Passing Data to Start Events

.start takes a single options parameter that will be passed to start events and their equivalent methods (onStart and onBeforeStart.)

var mod = MyApp.module("MyMod");

mod.on("before:start", function(options){

// do stuff before the module is started

});

mod.on("start", function(options){

// do stuff after the module has been started

});

var options = {

// any data

};

mod.start(options);

### Preventing Auto-Start Of Modules

The default behavior of modules is that they start with the application.  
If you wish to manually start a module instead, you can change this behavior  
with the startWithParent property.

var fooModule = MyApp.module("Foo", function(){

// prevent starting with parent

this.startWithParent = false;

// ... module code goes here

});

// start the app without starting the module

MyApp.start();

// later, start the module

fooModule.start();

The same behavior can be accomplished with the object literal definition:

var fooModule = MyApp.module("Foo", {

startWithParent: false

});

When splitting a module across multiple files, it is recommended that you set  
startWithParent to be false.

### Starting Sub-Modules With Parent

As you might expect, submodules default to starting with their parent module.  
The starting of sub-modules is done in a depth-first hierarchy traversal.  
That is, a hierarchy of Foo.Bar.Baz will start Baz first, then Bar,  
and finally Foo.

MyApp.module("Foo", function(){...});

MyApp.module("Foo.Bar", function(){...});

MyApp.start();

In this example, the "Foo.Bar" module will be started with the call to  
MyApp.start() because the parent module, "Foo" is (by default) set to start  
with the app.

A sub-module can override this behavior by setting its startWithParent  
to false. This prevents it from being started by the parent's start call.

MyApp.module("Foo", function(){...});

MyApp.module("Foo.Bar", function(){

this.startWithParent = false;

})

MyApp.start();

Now the module "Foo" will be started, but the sub-module "Foo.Bar" will  
not be started.

A sub-module can still be started manually, with this configuration:

MyApp.module("Foo.Bar").start();

## Stopping Modules

A module can be stopped, or shut down, to clear memory and resources when  
the module is no longer needed. Like the starting of modules, stopping is done  
in a depth-first hierarchy traversal. That is, a hierarchy of modules like  
Foo.Bar.Baz will stop Baz first, then Bar, and finally Foo.

To stop a module and its children, call the stop method of a module.

MyApp.module("Foo").stop();

Modules are not automatically stopped by the application. If you wish to  
stop one you must call the stop method on it, or stop its parent module.  
When you stop any parent module, all of its children will be stopped as well.

MyApp.module("Foo.Bar.Baz");

MyApp.module("Foo").stop();

This call to stop causes the Bar and Baz modules to both be stopped  
as they are sub-modules of Foo. For more information on defining  
sub-modules, see the section "Defining Sub-Modules".

### Stop Events

When stopping a module, a "before:stop" event will be triggered prior  
to any of the finalizers being run. A "stop" event will then be triggered  
after they have been run.

var mod = MyApp.module("MyMod");

mod.on("before:stop", function(){

// do stuff before the module is stopped

});

mod.on("stop", function(){

// do stuff after the module has been stopped

});

### Module Initializers

Warning: deprecated

This feature is deprecated, and is scheduled to be removed in version 3 of Marionette. Instead  
of Initializers, you should use events to manage start-up logic. The start event is an ideal  
substitute for Initializers.

If you were relying on the deferred nature of Initializers in your app, you should instead  
use Promises. This might look something like the following:

doAsyncThings().then(myModule.start);

Modules, like Application objects, can be configured to have initializers. And just like  
an Application's initializers, module's initializers are run anytime that  
the module is started. Further, there is no limit to the number of initializers it can have.

Initializers can be added in the module's definition function.

MyApp.module("Foo", function(Foo){

Foo.addInitializer(function(){

// Do things once the module has started

});

Foo.addInitializer(function(){

// You can have more than one initializer

});

});

### Module Finalizers

Warning: deprecated

This feature is deprecated, and is scheduled to be removed in version 3 of Marionette. Instead  
of Finalizers, you should use events to manage start-up logic. The stop event is an ideal  
substitute for Finalizers.

If you were relying on the deferred nature of Initializers in your app, you should instead  
use Promises. This might look something like the following:

doAsyncThings().then(myModule.stop);

Modules also have finalizers that work in an opposite manner to  
initializers: they are called whenever a module is stopped via the stop method.  
You can have as many finalizers as you'd like.

MyApp.module("Foo", function(Foo){

Foo.addFinalizer(function(){

// Tear down, shut down and clean up the module in here

});

Foo.addFinalizer(function(){

// Do more things

});

});

# Marionette.Object

A base class which other classes can extend from.  
Object incorporates many backbone conventions and utilities  
like initialize and Backbone.Events.

## Documentation Index

* [initialize](http://marionettejs.com/docs/v2.4.1/marionette.object.html#initialize)
* [events](http://marionettejs.com/docs/v2.4.1/marionette.object.html#events)
* [Destroying An Object](http://marionettejs.com/docs/v2.4.1/marionette.object.html#destroying-a-object)
* [mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.object.html#mergeoptions)
* [getOption](http://marionettejs.com/docs/v2.4.1/marionette.object.html#getoption)
* [bindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.object.html#bindentityevents)
* [Basic Use](http://marionettejs.com/docs/v2.4.1/marionette.object.html#basic-use)

### Initialize

Initialize is called immediately after the Object has been instantiated,  
and is invoked with the same arguments that the constructor received.

var Friend = Marionette.Object.extend({

initialize: function(options){

console.log(options.name);

}

});

new Friend({name: 'John'});

### Events

Marionette.Object extends Backbone.Events and includes triggerMethod.  
This makes it easy for Objects to emit events that other objects can listen for  
with on or listenTo.

var Friend = Marionette.Object.extend({

graduate: function() {

this.triggerMethod('announce', 'I graduated!!!');

}

});

var john = new Friend({name: 'John'});

john.on('announce', function(message) {

console.log(message); // I graduated!!!

})

john.graduate();

## mergeOptions

Merge keys from the options object directly onto the instance. This is the preferred way to access options  
passed into the Object.

More information at [mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettemergeoptions)

### getOption

Retrieve an object's attribute either directly from the object, or from the object's this.options, with this.options taking precedence.

More information [getOption](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettegetoption).

### bindEntityEvents

Helps bind a backbone "entity" to methods on a target object. More information [bindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettebindentityevents).

### Destroying A Object

Objects have a destroy method that unbind the events that are directly attached to the  
instance.

Invoking the destroy method will trigger a "before:destroy" event and corresponding  
onBeforeDestroy method call. These calls will be passed any arguments destroy  
was invoked with. Invoking destroy will return the object, this can be useful for chaining.

// define a object with an onDestroy method

var MyObject = Marionette.Object.extend({

onBeforeDestroy: function(arg1, arg2){

// put custom code here, to destroy this object

}

});

// create a new object instance

var obj = new MyObject();

// add some event handlers

obj.on("before:destroy", function(arg1, arg2){ ... });

obj.listenTo(something, "bar", function(){...});

// destroy the object: unbind all of the

// event handlers, trigger the "destroy" event and

// call the onDestroy method

obj.destroy(arg1, arg2);

### Basic Use

Selections is a simple Object that manages a selection of things.  
Because Selections extends from Object, it gets initialize and Events  
for free.

var Selections = Marionette.Object.extend({

initialize: function(options){

this.selections = {};

},

select: function(key, item){

this.triggerMethod("select", key, item);

this.selections[key] = item;

},

deselect: function(key, item) {

this.triggerMethod("deselect", key, item);

delete this.selections[key];

}

});

var selections = new Selections({

filters: Filters

});

// use the built in EventBinder

selections.listenTo(selections, "select", function(key, item){

console.log(item);

});

selections.select('toy', Truck);

# Marionette.Region

Regions provide consistent methods to manage, show and destroy  
views in your applications and layouts. They use a jQuery selector  
to show your views in the correct place.

Using the LayoutView class you can create nested regions.

## Documentation Index

* [Defining An Application Region](http://marionettejs.com/docs/v2.4.1/marionette.region.html#defining-an-application-region)
  + [Region Configuration Types](http://marionettejs.com/docs/v2.4.1/marionette.region.html#region-configuration-types)
* [Initialize A Region With An el](http://marionettejs.com/docs/v2.4.1/marionette.region.html#initialize-a-region-with-an-el)
* [Basic Use](http://marionettejs.com/docs/v2.4.1/marionette.region.html#basic-use)
* [Showing a view](http://marionettejs.com/docs/v2.4.1/marionette.region.html#showing-a-view)
* [Checking whether a region is showing a view](http://marionettejs.com/docs/v2.4.1/marionette.region.html#checking-whether-a-region-is-showing-a-view)
* [reset A Region](http://marionettejs.com/docs/v2.4.1/marionette.region.html#reset-a-region)
* [Set How View's el Is Attached](http://marionettejs.com/docs/v2.4.1/marionette.region.html#set-how-views-el-is-attached)
* [Attach Existing View](http://marionettejs.com/docs/v2.4.1/marionette.region.html#attach-existing-view)
  + [Set currentView On Initialization](http://marionettejs.com/docs/v2.4.1/marionette.region.html#set-currentview-on-initialization)
  + [Call attachView On Region](http://marionettejs.com/docs/v2.4.1/marionette.region.html#call-attachview-on-region)
* [Region Events And Callbacks](http://marionettejs.com/docs/v2.4.1/marionette.region.html#region-events-and-callbacks)
  + [Events raised during show](http://marionettejs.com/docs/v2.4.1/marionette.region.html#events-raised-during-show)
* [Custom Region Classes](http://marionettejs.com/docs/v2.4.1/marionette.region.html#custom-region-classes)
  + [Attaching Custom Region Classes](http://marionettejs.com/docs/v2.4.1/marionette.region.html#attaching-custom-region-classes)
  + [Instantiate Your Own Region](http://marionettejs.com/docs/v2.4.1/marionette.region.html#instantiate-your-own-region)

## Defining An Application Region

You can add regions to your applications by calling the addRegions method on  
your application instance. This method expects a single hash parameter, with  
named regions and either jQuery selectors or Region objects. You may  
call this method as many times as you like, and it will continue adding regions  
to the app.

MyApp.addRegions({

mainRegion: "#main-content",

navigationRegion: "#navigation"

});

As soon as you call addRegions, your regions are available on your  
app object. In the above, example MyApp.mainRegion and MyApp.navigationRegion  
would be available for use immediately.

If you specify the same region name twice, the last one in wins.

You can also add regions via LayoutViews:

var AppLayoutView = Backbone.Marionette.LayoutView.extend({

template: "#layout-view-template",

regions: {

menu: "#menu",

content: "#content"

}

});

var layoutView = new AppLayoutView();

layoutView.render();

layoutView.menu.show(new MenuView());

layoutView.content.show(new MainContentView());

### Region Configuration Types

Marionette supports multiple ways to define regions on your Application or LayoutView.

#### String Selector

You can use a jQuery string selector to define regions.

App.addRegions({

mainRegion: '#main'

});

#### Region Class

If you've created a custom region class, you can use it to  
define your region.

**NOTE:** Make sure the region class has an el  
property set or it won't work!

var MyRegion = Marionette.Region.extend({

el: '#main-nav'

});

App.addRegions({

navigationRegion: MyRegion

});

#### Object Literal

Finally, you can define regions with an object literal. Object  
literal definitions normally expect a selector or el  
property. The selector property is a selector string, and  
the el property can be a selector string, a jQuery object,  
or an HTML node.

You may also supply a regionClass property for a custom region  
class. If your regionClass already has el set, then you do  
not need to supply a selector or el property on the object  
literal.

Any other properties you set on the object literal will be  
used as options passed to the region instance, including the  
allowMissingEl option.

Ordinarily regions enforce the presence of a backing DOM element.  
In some instances it may be desirable to allow regions to be  
instantiated and used without an element, such as when regions  
defined by a parent LayoutView class are used by only some of its  
subclasses. In these instances, the region can be defined with the  
allowMissingEl option, suppressing the missing element error and  
causing show calls to the region to be treated as no-ops.

var MyRegion = Marionette.Region.extend();

var MyOtherRegion = Marionette.Region.extend();

var MyElRegion = Marionette.Region.extend({ el: '#footer' });

App.addRegions({

contentRegion: {

el: '#content',

regionClass: MyRegion

},

navigationRegion: {

el: '#navigation',

regionClass: MyOtherRegion,

// Options passed to instance of `MyOtherRegion` for

// the `navigationRegion` on `App`

navigationOption: 42,

anotherNavigationOption: 'foo'

},

footerRegion: {

regionClass: MyElRegion

}

});

Take note that one of the primary benefits of using regionClass  
with an el already set is to also provide options to the region  
instance. This isn't possible when using the region class directly  
like earlier.

var MyRegion = Marionette.Region.extend({

el: '#content',

});

App.addRegions({

contentRegion: {

regionClass: MyRegion,

myRegionOption: 'bar',

myOtherRegionOption: 'baz'

}

});

#### Mix-and-match

Of course you can mix-and-match the region configuration types.

var MyRegion = Marionette.Region.extend({

el: '#content'

});

var MyOtherRegion = Marionette.Region.extend();

App.addRegions({

contentRegion: MyRegion,

navigationRegion: '#navigation',

footerRegion: {

el: '#footer',

regionClass: MyOtherRegion

}

});

## Initialize A Region With An el

You can specify an el for the region to manage at the time  
that the region is instantiated:

var mgr = new Backbone.Marionette.Region({

el: "#someElement"

});

The el option can also be a raw DOM node reference:

var mgr = new Backbone.Marionette.Region({

el: document.querySelector("body")

});

Or the el can also be a jQuery wrapped DOM node:

var mgr = new Backbone.Marionette.Region({

el: $("body")

});

## Basic Use

### Showing a View

Once a region is defined, you can call its show  
and empty methods to display and shut-down a view:

var myView = new MyView();

// render and display the view

MyApp.mainRegion.show(myView);

// empties the current view

MyApp.mainRegion.empty();

#### preventDestroy

If you replace the current view with a new view by calling show,  
by default it will automatically destroy the previous view.  
You can prevent this behavior by passing {preventDestroy: true} in the options  
parameter. Several events will also be triggered on the views; see  
[Region Events And Callbacks](http://marionettejs.com/docs/v2.4.1/marionette.region.html#region-events-and-callbacks) for details.

// Show the first view.

var myView = new MyView();

MyApp.mainRegion.show(myView);

// Replace the view with another. The

// `destroy` method is called for you

var anotherView = new AnotherView();

MyApp.mainRegion.show(anotherView);

// Replace the view with another.

// Prevent `destroy` from being called

var anotherView2 = new AnotherView();

MyApp.mainRegion.show(anotherView2, { preventDestroy: true });

NOTE: When using preventDestroy: true you must be careful to cleanup your old views  
manually to prevent memory leaks.

#### forceShow

If you re-call show with the same view, by default nothing will happen  
because the view is already in the region. You can force the view to be re-shown  
by passing in {forceShow: true} in the options parameter.

var myView = new MyView();

MyApp.mainRegion.show(myView);

// the second show call will re-show the view

MyApp.mainRegion.show(myView, {forceShow: true});

#### Emptying a region

You can empty a region of its view and contents by invoking .empty() on the region instance.  
If you would like to prevent the view currently shown in the region from being destroyed you can pass {preventDestroy: true} to the empty method to prevent the default destroy behavior.  
The empty method returns the region instance from the invocation of the method.

#### onBeforeAttach & onAttach

Regions that are attached to the document when you execute show are special in that the  
views that they show will also become attached to the document. These regions fire a pair of triggerMethods on all  
of the views that are about to be attached – even the nested ones. This can cause a performance issue if you're  
rendering hundreds or thousands of views at once.

If you think these events might be causing some lag in your app, you can selectively turn them off  
with the triggerBeforeAttach and triggerAttach properties.

// No longer trigger attach

myRegion.triggerAttach = false;

You can override this on a per-show basis by passing it in as an option to show.

// This region won't trigger beforeAttach...

myRegion.triggerBeforeAttach = false;

// Unless we tell it to

myRegion.show(myView, {triggerBeforeAttach: true});

### Checking whether a region is showing a view

If you wish to check whether a region has a view, you can use the hasView  
function. This will return a boolean value depending whether or not the region  
is showing a view.

### reset A Region

A region can be reset at any time. This destroys any existing view  
being displayed, and deletes the cached el. The next time the  
region shows a view, the region's el is queried from  
the DOM.

myRegion.reset();

This is useful when regions are re-used across view  
instances, and in unit testing.

### Set How View's el Is Attached

Override the region's attachHtml method to change how the view is attached  
to the DOM. This method receives one parameter - the view to show.

The default implementation of attachHtml is:

Marionette.Region.prototype.attachHtml = function(view){

this.$el.empty().append(view.el);

}

This replaces the contents of the region with the view's  
el / content. You can override attachHtml for transition effects and more.

Marionette.Region.prototype.attachHtml = function(view){

this.$el.hide();

this.$el.html(view.el);

this.$el.slideDown("fast");

}

It is also possible to define a custom render method for a single region by  
extending from the Region class and including a custom attachHtml method.

This example will make a view slide down from the top of the screen instead of just  
appearing in place:

var ModalRegion = Marionette.Region.extend({

attachHtml: function(view){

// Some effect to show the view:

this.$el.empty().append(view.el);

this.$el.hide().slideDown('fast');

}

})

MyApp.addRegions({

mainRegion: '#main-region',

modalRegion: {

regionClass: ModalRegion,

selector: '#modal-region'

}

})

### Attach Existing View

There are some scenarios where it's desirable to attach an existing  
view to a region , without rendering or showing the view, and  
without replacing the HTML content of the region. For example, SEO and  
accessibility often need HTML to be generated by the server, and progressive  
enhancement of the HTML.

There are two ways to accomplish this:

* set the currentView in the region's constructor
* call attachView on the region instance

#### Set currentView On Initialization

var myView = new MyView({

el: $("#existing-view-stuff")

});

var region = new Backbone.Marionette.Region({

el: "#content",

currentView: myView

});

#### Call attachView On Region

MyApp.addRegions({

someRegion: "#content"

});

var myView = new MyView({

el: $("#existing-view-stuff")

});

MyApp.someRegion.attachView(myView);

## Region Events And Callbacks

### Events raised during show:

A region will raise a few events when showing  
and destroying views:

* "before:show" / onBeforeShow - Called on the view instance after the view has been rendered, but before its been displayed.
* "before:show" / onBeforeShow - Called on the region instance after the view has been rendered, but before its been displayed.
* "show" / onShow - Called on the view instance when the view has been rendered and displayed.
* "show" / onShow - Called on the region instance when the view has been rendered and displayed.
* "before:swap" / onBeforeSwap - Called on the region instance before a new view is shown. NOTE: this will only be called when a view is being swapped, not when the region is empty.
* "before:swapOut" / onBeforeSwapOut - Called on the region instance before a new view swapped in. NOTE: this will only be called when a view is being swapped, not when the region is empty.
* "swap" / onSwap - Called on the region instance when a new view is shown. NOTE: this will only be called when a view is being swapped, not when the region is empty.
* "swapOut" / onSwapOut - Called on the region instance when a new view swapped in to replace the currently shown view. NOTE: this will only be called when a view is being swapped, not when the region is empty.
* "before:empty" / onBeforeEmpty - Called on the region instance before the view has been emptied.
* "empty" / onEmpty - Called when the view has been emptied.

These events can be used to run code when your region  
opens and destroys views.

MyApp.mainRegion.on("before:show", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

MyApp.mainRegion.on("show", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

MyApp.mainRegion.on("before:swap", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

MyApp.mainRegion.on("swap", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

MyApp.mainRegion.on("before:swapOut", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

MyApp.mainRegion.on("swapOut", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

MyApp.mainRegion.on("empty", function(view, region, options){

// manipulate the `view` or do something extra

// with the `region`

// you also have access to the `options` that were passed to the Region.show call

});

var MyRegion = Backbone.Marionette.Region.extend({

// ...

onBeforeShow: function(view, region, options) {

// the `view` has not been shown yet

},

onShow: function(view, region, options){

// the `view` has been shown

}

});

var MyView = Marionette.ItemView.extend({

onBeforeShow: function(view, region, options) {

// called before the `view` has been shown

},

onShow: function(view, region, options){

// called when the `view` has been shown

}

});

var MyRegion = Backbone.Marionette.Region.extend({

// ...

onBeforeSwap: function(view, region, options) {

// the `view` has not been swapped yet

},

onSwap: function(view, region, options){

// the `view` has been swapped

},

onBeforeSwapOut: function(view, region, options) {

// the `view` has not been swapped out yet

},

onSwapOut: function(view, region, options){

// the `view` has been swapped out

}

});

## Custom Region Classes

You can define a custom region by extending from  
Region. This allows you to create new functionality,  
or provide a base set of functionality for your app.

### Attaching Custom Region Classes

Once you define a region class, you can attach the  
new region class by specifying the region class as the  
value. In this case, addRegions expects the constructor itself, not an instance.

var FooterRegion = Backbone.Marionette.Region.extend({

el: "#footer"

});

MyApp.addRegions({

footerRegion: FooterRegion

});

You can also specify a selector for the region by using  
an object literal for the configuration.

var FooterRegion = Backbone.Marionette.Region.extend({

el: "#footer"

});

MyApp.addRegions({

footerRegion: {

selector: "#footer",

regionClass: FooterRegion

}

});

Note that a region must have an element to attach itself to. If you  
do not specify a selector when attaching the region instance to your  
Application or LayoutView, the region must provide an el either in its  
definition or constructor options.

### Instantiate Your Own Region

There may be times when you want to add a region to your  
application after your app is up and running. To do this, you'll  
need to extend from Region as shown above and then use  
that constructor function on your own:

var SomeRegion = Backbone.Marionette.Region.extend({

el: "#some-div",

initialize: function(options){

// your init code, here

}

});

MyApp.someRegion = new SomeRegion();

MyApp.someRegion.show(someView);

You can optionally add an initialize function to your Region  
definition as shown in this example. It receives the options  
that were passed to the constructor of the Region, similar to  
a Backbone.View.

# Marionette.RegionManager

Region managers provide a consistent way to manage  
a number of Marionette.Region objects within an  
application. The RegionManager is intended to be  
used by other objects, to facilitate the addition,  
storage, retrieval, and removal of regions from  
that object. For examples of how it can be used,  
see the [Marionette.Application](http://marionettejs.com/docs/v2.4.1/marionette.application.html) and [Marionette.LayoutView](http://marionettejs.com/docs/v2.4.1/marionette.layoutview.html)  
objects.

## Documentation Index

* [Basic Use](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#basic-use)
* [Constucting](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#constructing)
* [RegionManager.addRegion](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanageraddregion)
* [RegionManager.addRegions](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanageraddregions)
  + [addRegions default options](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#addregions-default-options)
* [RegionManager.get](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanagerget)
* [RegionManager.getRegions](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanagergetregions)
* [RegionManager.removeRegion](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanagerremoveregion)
* [RegionManager.removeRegions](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanagerremoveregions)
* [RegionManager.emptyRegions](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanageremptyregions)
* [RegionManager.destroy](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanagerdestroy)
* [RegionManager Events](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanager-events)
  + [before:add:region event](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#beforeaddregion-event)
  + [add:region event](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#addregion-event)
  + [before:remove:region event](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#beforeremoveregion-event)
  + [remove:region event](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#removeregion-event)
* [RegionManager Iterators](http://marionettejs.com/docs/v2.4.1/marionette.regionmanager.html#regionmanager-iterators)

## Basic Use

RegionManagers can be instantiated directly, and can  
have regions added and removed via several methods:

var rm = new Marionette.RegionManager();

var region = rm.addRegion("foo", "#bar");

var regions = rm.addRegions({

baz: "#baz",

quux: "ul.quux"

});

regions.get('baz').show(myView);

rm.removeRegion("foo");

## Constructing

The RegionManager take an optional region option in their constructor. the regions are passed directly into addRegions for the region manager instance.

var mananger = new Marionette.RegionManager({

regions: {

"aRegion": "#bar"

}

});

mananger.get('aRegion').show(new MyView);

## RegionManager.addRegion

Regions can be added individually using the addRegion  
method. This method takes two parameters: the region name  
and the region definition.

var rm = new Marionette.RegionManager();

var region = rm.addRegion("foo", "#bar");

In this example, a region named "foo" will be added  
to the RegionManager instance. It is defined as a  
jQuery selector that will search for the #bar  
element in the DOM.

There are a lot of other ways to define a region,  
including object literals with various options, and  
instances of Region objects. For more information  
on this, see the Region documentation.

## RegionManager.addRegions

Regions can also be added en-masse through the use  
of the addRegions method. This method takes an object  
literal or a function that returns an object literal.  
The object literal must contain region names as keys  
and region definitions as values. The return value  
is an object literal with all the created regions.

var rm = new Marionette.RegionManager();

// With an object literal

var regions = rm.addRegions({

main: '#main-content',

navigation: {

selector: '#navigation',

regionClass: MyNavRegion

}

});

// With a function

var otherRegions = rm.addRegions(function(regionDefinition) {

return {

footer: '#footer'

};

});

regions.get('main'); //=> 'main' region instance

regions.get('navigation'); //=> 'navigation' region instance

otherRegions.get('footer'); //=> 'footer' region instance

If you supply a function to addRegions, it will be  
called with the RegionManager instance context and  
all the arguments passed to addRegions.

var rm = new Marionette.RegionManager();

var regionDefaults = {

regionClass: MyRegionClass

};

rm.addRegions(function(regionDefinition, defaults) {

console.log(this); // `rm` instance of `RegionManager`

console.log(regionDefinition); // the region definition function

console.log(defaults); // `{ regionClass: MyRegionClass }`

// ...return the region definiton object literal

}, regionDefaults);

### addRegions default options

When adding multiple regions it may be useful to  
provide a set of defaults that get applied to all  
of the regions being added. This can be done through  
the use of a defaults parameter. Specify this  
parameter as an object literal with key: value  
pairs that will be applied to every region added.

var rm = new Marionette.RegionManager();

var defaults = {

regionClass: MyRegionClass

};

var regions = {

foo: "#bar",

baz: "#quux"

};

rm.addRegions(regions, defaults);

In this example, all regions will be added as  
instances of MyRegionClass.

## RegionManager.get

A region instance can be retrieved from the  
RegionManager instance using the get method and  
passing in the name of the region.

var rm = new Marionette.RegionManager();

rm.addRegion("foo", "#bar");

var region = rm.get("foo");

## RegionManager.getRegions

Get all the regions from the region manager.  
Returns an object literal with named regions  
as attributes.

var rm = new Marionette.RegionManager();

rm.addRegion("foo", "#foo");

rm.addRegion("bar", "#bar");

var regions = rm.getRegions();

regions.foo; //=> foo region

regions.bar; //=> bar region

## RegionManager.removeRegion

A region can be removed by calling the removeRegion  
method and passing in the name of the region.

var rm = new Marionette.RegionManager();

rm.addRegion("foo", "#bar");

rm.removeRegion("foo");

A region will have its empty method called before  
it is removed from the RegionManager instance and  
stopListening is called.

## RegionManager.removeRegions

You can quickly remove all regions from the  
RegionManager instance by calling the removeRegions  
method.

var rm = new Marionette.RegionManager();

rm.addRegions({

foo: "#foo",

bar: "#bar",

baz: "#baz"

});

rm.removeRegions();

This will empty all regions, and remove them.

## RegionManager.emptyRegions

You can quickly empty all regions from the RegionManager  
instance by calling the emptyRegions method.

var rm = new Marionette.RegionManager();

rm.addRegions({

foo: "#foo",

bar: "#bar",

baz: "#baz"

});

rm.emptyRegions();

This will empty the regions without removing them  
from the RegionManager instance.

## RegionManager.destroy

A RegionManager instance can be destroyed entirely by  
calling the destroy method. This will both destroy  
and remove all regions from the RegionManager instance.

var rm = new Marionette.RegionManager();

rm.addRegions({

foo: "#foo",

bar: "#bar",

baz: "#baz"

});

rm.destroy();

## RegionManager Events

A RegionManager will trigger various events as it  
is being used.

### before:add:region event

The RegionManager will trigger a "before:add:region"  
event before a region is added to the manager. This  
allows you to perform some actions on the region before it is added.

var rm = new Marionette.RegionManager();

rm.on("before:add:region", function(name, region){

// do something with the region instance

});

rm.addRegion("foo", "#bar");

### add:region event

The RegionManager will trigger a "add:region"  
event when a region is added to the manager. This  
allows you to use the region instance immediately,  
or attach the region to an object that needs a  
reference to it:

var rm = new Marionette.RegionManager();

rm.on("add:region", function(name, region){

// add the region instance to an object

myObject[name] = region;

});

rm.addRegion("foo", "#bar");

### before:remove:region event

The RegionManager will trigger a "before:remove:region"  
event before a region is removed from the manager.  
This allows you to perform any cleanup operations before the region is removed.

var rm = new Marionette.RegionManager();

rm.on("before:remove:region", function(name, region){

// do something with the region instance here

});

rm.addRegion("foo", "#bar");

rm.removeRegion("foo");

### remove:region event

The RegionManager will trigger a "remove:region"  
event when a region is removed from the manager.  
This allows you to use the region instance one last  
time, or remove the region from an object that has a  
reference to it:

var rm = new Marionette.RegionManager();

rm.on("remove:region", function(name, region){

// add the region instance to an object

delete myObject[name];

});

rm.addRegion("foo", "#bar");

rm.removeRegion("foo");

## RegionManager Iterators

The RegionManager has several methods for iteration  
attached to it, from underscore.js. This works in the  
same way as the Backbone.Collection methods that have  
been imported. For example, you can easily iterate over  
the entire collection of region instances by calling  
the each method:

var rm = new Marionette.RegionManager();

rm.each(function(region){

// do stuff w/ the region instance here

});

The list of underscore methods include:

* [forEach](http://underscorejs.org/#each)
* [each](http://underscorejs.org/#each)
* [map](http://underscorejs.org/#map)
* [find](http://underscorejs.org/#find)
* [detect](http://underscorejs.org/#find)
* [filter](http://underscorejs.org/#filter)
* [select](http://underscorejs.org/#filter)
* [reject](http://underscorejs.org/#reject)
* [every](http://underscorejs.org/#every)
* [all](http://underscorejs.org/#every)
* [some](http://underscorejs.org/#some)
* [any](http://underscorejs.org/#some)
* [include](http://underscorejs.org/#contains)
* [contains](http://underscorejs.org/#contains)
* [invoke](http://underscorejs.org/#invoke)
* [toArray](http://underscorejs.org/#toArray)
* [first](http://underscorejs.org/#first)
* [initial](http://underscorejs.org/#initial)
* [rest](http://underscorejs.org/#rest)
* [last](http://underscorejs.org/#last)
* [without](http://underscorejs.org/#without)
* [isEmpty](http://underscorejs.org/#isEmpty)
* [pluck](http://underscorejs.org/#pluck)

# Marionette.Renderer

The Renderer object was extracted from the ItemView rendering  
process, in order to create a consistent and re-usable method of  
rendering a template with or without data.

## Documentation Index

* [Basic Usage](http://marionettejs.com/docs/v2.4.1/marionette.renderer.html#basic-usage)
* [Pre-compiled Templates](http://marionettejs.com/docs/v2.4.1/marionette.renderer.html#pre-compiled-templates)
* [Custom Template Selection And Rendering](http://marionettejs.com/docs/v2.4.1/marionette.renderer.html#custom-template-selection-and-rendering)
* [Using Pre-compiled Templates](http://marionettejs.com/docs/v2.4.1/marionette.renderer.html#using-pre-compiled-templates)

## Basic Usage

The basic usage of the Renderer is to call the render method.  
This method returns a string containing the result of applying the  
template using the data object as the context.

var template = "#some-template";

var data = {foo: "bar"};

var html = Backbone.Marionette.Renderer.render(template, data);

// do something with the HTML here

If you pass a template that coerces to a falsey value, the  
render method will throw an exception stating that there was no  
template provided.

## Pre-compiled Templates

If the template parameter of the render function is itself a function,  
the renderer treats this as a pre-compiled template and does not try to  
compile it again. This allows any view that supports a template parameter  
to specify a pre-compiled template function as the template setting.

var myTemplate = \_.template("<div>foo</div>");

Backbone.Marionette.ItemView.extend({

template: myTemplate

});

The template function does not have to be any specific template engine. It  
only needs to be a function that returns valid HTML as a string from the  
data parameter passed to the function.

## Custom Template Selection And Rendering

By default, the renderer will take a jQuery selector object as  
the first parameter, and a JSON data object as the optional  
second parameter. It then uses the TemplateCache to load the  
template by the specified selector, and renders the template with  
the data provided (if any) using Underscore.js templates.

If you wish to override the way the template is loaded, see  
the TemplateCache object.

If you wish to override the template engine used, change the  
render method to work however you want:

Backbone.Marionette.Renderer.render = function(template, data){

return $(template).tmpl(data);

};

This implementation will replace the default Underscore.js  
rendering with jQuery templates rendering.

If you override the render method and wish to use the  
TemplateCache mechanism, remember to include the code necessary to  
fetch the template from the cache in your render method:

Backbone.Marionette.Renderer.render = function(template, data){

var template = Marionette.TemplateCache.get(template);

// Do something with the template here

};

## Using Pre-compiled Templates

You can easily replace the standard template rendering functionality  
with a pre-compiled template, such as those provided by the JST or TPL  
plugins for AMD/RequireJS.

To do this, just override the render method to return your executed  
template with the data.

Backbone.Marionette.Renderer.render = function(template, data){

return template(data);

};

Then you can specify the pre-compiled template function as your view's  
template attribute:

var myPrecompiledTemplate = \_.template("<div>some template</div>");

Backbone.Marionette.ItemView.extend({

template: myPrecompiledTemplate

});

# Marionette.TemplateCache

The TemplateCache provides a cache for retrieving templates  
from script blocks in your HTML. This will improve  
the speed of subsequent calls to get a template.

## Documentation Index

* [Basic Usage](http://marionettejs.com/docs/v2.4.1/marionette.templatecache.html#basic-usage)
* [Clear Items From cache](http://marionettejs.com/docs/v2.4.1/marionette.templatecache.html#clear-items-from-cache)
* [Customizing Template Access](http://marionettejs.com/docs/v2.4.1/marionette.templatecache.html#customizing-template-access)
* [Override Template Retrieval](http://marionettejs.com/docs/v2.4.1/marionette.templatecache.html#override-template-retrieval)
* [Override Template Compilation](http://marionettejs.com/docs/v2.4.1/marionette.templatecache.html#override-template-compilation)

## Basic Usage

To use the TemplateCache, call the get method on TemplateCache directly.  
Internally, instances of the TemplateCache class will be created and stored  
but you do not have to manually create these instances yourself. get will  
return a compiled template function.

var template = Backbone.Marionette.TemplateCache.get("#my-template", {some: options});

// use the template

template({param1:'value1', paramN:'valueN'});

Making multiple calls to get the same template will retrieve the  
template from the cache on subsequence calls.

### Clear Items From cache

You can clear one or more, or all items from the cache using the  
clear method. Clearing a template from the cache will force it  
to re-load from the DOM (via the loadTemplate  
function which can be overridden, see below) the next time it is retrieved.

If you do not specify any parameters, all items will be cleared  
from the cache:

Backbone.Marionette.TemplateCache.get("#my-template");

Backbone.Marionette.TemplateCache.get("#this-template");

Backbone.Marionette.TemplateCache.get("#that-template");

// clear all templates from the cache

Backbone.Marionette.TemplateCache.clear()

If you specify one or more parameters, these parameters are assumed  
to be the templateId used for loading / caching:

Backbone.Marionette.TemplateCache.get("#my-template");

Backbone.Marionette.TemplateCache.get("#this-template");

Backbone.Marionette.TemplateCache.get("#that-template");

// clear 2 of 3 templates from the cache

Backbone.Marionette.TemplateCache.clear("#my-template", "#this-template")

## Customizing Template Access

If you want to use an alternate template engine while  
still taking advantage of the template caching functionality, or want to customize  
how templates are stored and retrieved, you will need to customize the  
TemplateCache object. The default operation of TemplateCache, is to  
retrieve templates from the DOM based on the containing element's id  
attribute, and compile the html in that element with the underscore.js  
template function.

### Override Template Retrieval

The default template retrieval is to select the template contents  
from the DOM using jQuery. If you wish to change the way this  
works, you can override the loadTemplate method on the  
TemplateCache object.

Backbone.Marionette.TemplateCache.prototype.loadTemplate = function(templateId, options){

// load your template here, returning the data needed for the compileTemplate

// function. For example, you have a function that creates templates based on the

// value of templateId

var myTemplate = myTemplateFunc(templateId);

// send the template back

return myTemplate;

}

### Override Template Compilation

The default template compilation passes the results from  
loadTemplate to the compileTemplate function, which returns  
an underscore.js compiled template function. When overriding compileTemplate  
remember that it must return a function which takes an object of parameters and values  
and returns a formatted HTML string.

Backbone.Marionette.TemplateCache.prototype.compileTemplate = function(rawTemplate, options) {

// use Handlebars.js to compile the template

return Handlebars.compile(rawTemplate);

}

# Marionette.View

Marionette has a base Marionette.View class that other views extend from.  
This base view provides some common and core functionality for  
other views to take advantage of.

**Note:** The Marionette.View class is not intended to be  
used directly. It exists as a base view for other view classes  
to be extended from, and to provide a common location for  
behaviors that are shared across all views.

## Documentation Index

* [Binding To View Events](http://marionettejs.com/docs/v2.4.1/marionette.view.html#binding-to-view-events)
* [View onShow](http://marionettejs.com/docs/v2.4.1/marionette.view.html#view-onshow)
* [View destroy](http://marionettejs.com/docs/v2.4.1/marionette.view.html#view-destroy)
* [View onBeforeDestroy](http://marionettejs.com/docs/v2.4.1/marionette.view.html#view-onbeforedestroy)
* [View "attach" / onAttach event](http://marionettejs.com/docs/v2.4.1/marionette.view.html#view-attach--onattach-event)
* [View "before:attach" / onBeforeAttach event](http://marionettejs.com/docs/v2.4.1/marionette.view.html#view-beforeattach--onbeforeattach-event)
* [View "dom:refresh" / onDomRefresh event](http://marionettejs.com/docs/v2.4.1/marionette.view.html#view-domrefresh--ondomrefresh-event)
* [View.triggers](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewtriggers)
* [View.events](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewevents)
* [View.modelEvents and View.collectionEvents](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewmodelevents-and-viewcollectionevents)
* [View.serializeModel](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewserializemodel)
* [View.bindUIElements](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewbinduielements)
* [View.mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewmergeoptions)
* [View.getOption](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewgetoption)
* [View.bindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewbindentityevents)
* [View.templateHelpers](http://marionettejs.com/docs/v2.4.1/marionette.view.html#viewtemplatehelpers)
  + [Basic Example](http://marionettejs.com/docs/v2.4.1/marionette.view.html#basic-example)
  + [Accessing Data Within The Helpers](http://marionettejs.com/docs/v2.4.1/marionette.view.html#accessing-data-within-the-helpers)
  + [Object Or Function As templateHelpers](http://marionettejs.com/docs/v2.4.1/marionette.view.html#object-or-function-as-templatehelpers)
* [Change Which Template Is Rendered For A View](http://marionettejs.com/docs/v2.4.1/marionette.view.html#change-which-template-is-rendered-for-a-view)

## Binding To View Events

Marionette.View extends Backbone.View. It is recommended that you use  
the listenTo method to bind model, collection, or other events from Backbone  
and Marionette objects.

var MyView = Backbone.Marionette.ItemView.extend({

initialize: function(){

this.listenTo(this.model, "change:foo", this.modelChanged);

this.listenTo(this.collection, "add", this.modelAdded);

},

modelChanged: function(model, value){

},

modelAdded: function(model){

}

});

The context (this) will automatically be set to the view. You can  
optionally set the context by using \_.bind.

// Force the context of the "reconcileCollection" callback method to be the collection

// itself, for this event handler only (does not affect any other use of the

// "reconcileCollection" method)

this.listenTo(this.collection, "add", \_.bind(this.reconcileCollection, this.collection));

## View onShow

* "show" / onShow - Called on the view instance when the view has been rendered and displayed.

This event can be used to react to when a view has been shown via a [region](http://marionettejs.com/docs/v2.4.1/marionette.region.html).  
All views that inherit from the base Marionette.View class have this functionality, notably ItemView, CollectionView, CompositeView, and LayoutView.

Backbone.Marionette.ItemView.extend({

onShow: function(){

// react to when a view has been shown

}

});

A common use case for the onShow method is to use it to add children views.

var LayoutView = Backbone.Marionette.LayoutView.extend({

regions: {

Header: 'header',

Section: 'section'

},

onShow: function() {

this.Header.show(new Header());

this.Section.show(new Section());

}

});

## View destroy

View implements a destroy method, which is called by the region  
managers automatically. As part of the implementation, the following  
are performed:

* call an onBeforeDestroy event on the view, if one is provided
* call an onDestroy event on the view, if one is provided
* unbind all custom view events
* unbind all DOM events
* remove this.el from the DOM
* unbind all listenTo events
* returns the view.

By providing an onDestroy method in your view definition, you can  
run custom code for your view that is fired after your view has been  
destroyed and cleaned up. The onDestroy method will be passed any arguments  
that destroy was invoked with. This lets you handle any additional clean  
up code without having to override the destroy method.

var MyView = Backbone.Marionette.ItemView.extend({

onDestroy: function(arg1, arg2){

// custom cleanup or destroying code, here

}

});

var v = new MyView();

v.destroy(arg1, arg2);

## View onBeforeDestroy

When destroying a view, an onBeforeDestroy method will be called, if it  
has been provided, just before the view destroys. It will be passed any arguments  
that destroy was invoked with.

### View "attach" / onAttach event

Every view in Marionette has a special event called "attach," which is triggered anytime that showing  
the view in a Region causes it to be attached to the document. Like other Marionette events, it also  
executes a callback method, onAttach, if you've specified one. The "attach" event is great for jQuery  
plugins or other logic that must be executed after the view is attached to the document.

The attach event is only fired when the view becomes a child of the document. If the Region you're showing the view in is not a child of the document at the time that you call show then the attach event will not fire until the Region is a child of the document.

This event is unique in that it propagates down the view tree. For instance, when a CollectionView's  
attach event is fired, all of its children views will have the attach event fired as well. In  
addition, deeply nested Layout View structures will all have their attach event fired at the proper  
time, too.

For more on efficient, deeply-nested view structures, refer to the LayoutView docs.

### View "before:attach" / onBeforeAttach event

This is just like the attach event described above, but it's triggered right before the view is  
attached to the document.

### View "dom:refresh" / onDomRefresh event

Triggered after the view has been rendered, has been shown in the DOM via a Marionette.Region, and has been  
re-rendered.

This event / callback is useful for  
[DOM-dependent UI plugins](http://lostechies.com/derickbailey/2012/02/20/using-jquery-plugins-and-ui-controls-with-backbone/) such as  
[jQueryUI](http://jqueryui.com/) or [KendoUI](http://kendoui.com).

Backbone.Marionette.ItemView.extend({

onDomRefresh: function(){

// manipulate the `el` here. it's already

// been rendered, and is full of the view's

// HTML, ready to go.

}

});

For more information about integration Marionette w/ KendoUI (also applicable to jQueryUI and other UI  
widget suites), see [this blog post on KendoUI + Backbone](http://www.kendoui.com/blogs/teamblog/posts/12-11-26/backbone_and_kendo_ui_a_beautiful_combination.aspx).

## View.events

Since Views extend from backbone's view class, you gain the benefits of the [events hash](http://backbonejs.org/#View-delegateEvents).

Some preprocessing sugar is added on top to add the ability to cross utilize the ui hash.

var MyView = Backbone.Marionette.ItemView.extend({

// ...

ui: {

"cat": ".dog"

},

events: {

"click @ui.cat": "bark" //is the same as "click .dog":

}

});

## View.triggers

Views can define a set of triggers as a hash, which will  
convert a DOM event into a  
[view.triggerMethod](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettetriggermethod) call.

The left side of the hash is a standard Backbone.View DOM  
event configuration, while the right side of the hash is the  
view event that you want to trigger from the view.

var MyView = Backbone.Marionette.ItemView.extend({

// ...

triggers: {

"click .do-something": "something:do:it"

}

});

var view = new MyView();

view.render();

view.on("something:do:it", function(args){

alert("I DID IT!");

});

// "click" the 'do-something' DOM element to

// demonstrate the DOM event conversion

view.$(".do-something").trigger("click");

The result of this is an alert box that says, "I DID IT!" Triggers can also be  
executed using the 'on{EventName}' attribute.

By default all triggers are stopped with preventDefault and  
stopPropagation methods. But you can manually configure the triggers using  
hash instead of event name. Example below triggers an event and prevents  
default browser behaviour using preventDefault method.

Backbone.Marionette.CompositeView.extend({

triggers: {

"click .do-something": {

event: "something:do:it",

preventDefault: true, // this param is optional and will default to true

stopPropagation: false

}

}

});

You can also specify the triggers as a function that  
returns a hash of trigger configurations

Backbone.Marionette.CompositeView.extend({

triggers: function(){

return {

"click .that-thing": "that:i:sent:you"

};

}

});

Trigger keys can be configured to cross utilize the ui hash.

Backbone.Marionette.ItemView.extend({

ui: {

'monkey': '.guybrush'

},

triggers: {

'click @ui.monkey': 'see:LeChuck' // equivalent of "click .guybrush"

}

});

Triggers work with all View classes that extend from the base  
Marionette.View.

### Trigger Handler Arguments

A trigger event handler will receive a single argument that  
includes the following:

* view
* model
* collection

These properties match the view, model, and collection properties of the view that triggered the event.

var MyView = Backbone.Marionette.ItemView.extend({

// ...

triggers: {

"click .do-something": "some:event"

}

});

var view = new MyView();

view.on("some:event", function(args){

args.view; // => the view instance that triggered the event

args.model; // => the view.model, if one was set on the view

args.collection; // => the view.collection, if one was set on the view

});

Having access to these allows more flexibility in handling events from  
multiple views. For example, a tab control or expand/collapse widget such  
as a panel bar could trigger the same event from many different views  
and be handled with a single function.

## View.modelEvents and View.collectionEvents

Similar to the events hash, views can specify a configuration  
hash for collections and models. The left side is the event on  
the model or collection, and the right side is the name of the  
method on the view.

Backbone.Marionette.CompositeView.extend({

modelEvents: {

"change:name": "nameChanged" // equivalent to view.listenTo(view.model, "change:name", view.nameChanged, view)

},

collectionEvents: {

"add": "itemAdded" // equivalent to view.listenTo(view.collection, "add", view.itemAdded, view)

},

// ... event handler methods

nameChanged: function(){ /\* ... \*/ },

itemAdded: function(){ /\* ... \*/ },

})

These will use the memory safe listenTo, and will set the context  
(the value of this) in the handler to be the view. Events are  
bound at the time of instantiation, and an exception will be thrown  
if the handlers on the view do not exist.

The modelEvents and collectionEvents will be bound and  
unbound with the Backbone.View delegateEvents and undelegateEvents  
method calls. This allows the view to be re-used and have  
the model and collection events re-bound.

### Multiple Callbacks

Multiple callback functions can be specified by separating them with a  
space.

Backbone.Marionette.CompositeView.extend({

modelEvents: {

"change:name": "nameChanged thatThing"

},

nameChanged: function(){ },

thatThing: function(){ },

});

This works in both modelEvents and collectionEvents.

### Callbacks As Function

A single function can be declared directly in-line instead of specifying a  
callback via a string method name.

Backbone.Marionette.CompositeView.extend({

modelEvents: {

"change:name": function(){

// handle the name changed event here

}

}

});

This works for both modelEvents and collectionEvents.

### Event Configuration As Function

A function can be used to declare the event configuration as long as  
that function returns a hash that fits the above configuration options.

Backbone.Marionette.CompositeView.extend({

modelEvents: function(){

return { "change:name": "someFunc" };

}

});

This works for both modelEvents and collectionEvents.

## View.serializeModel

The serializeModel method will serialize a model that is passed in as an argument.

## View.bindUIElements

In several cases you need to access ui elements inside the view  
to retrieve their data or manipulate them. For example you have a  
certain div element you need to show/hide based on some state,  
or other ui element that you wish to set a css class to it.  
Instead of having jQuery selectors hanging around in the view's code  
you can define a ui hash that contains a mapping between the  
ui element's name and its jQuery selector. Afterwards you can simply  
access it via this.ui.elementName.  
See ItemView documentation for examples.

This functionality is provided via the bindUIElements method.  
Since View doesn't implement the render method, then if you directly extend  
from View you will need to invoke this method from your render method.  
In ItemView and CompositeView this is already taken care of.

## View.mergeOptions

The preferred way to manage your view's options is with mergeOptions. It accepts two arguments: the options object  
and the keys to merge onto the instance directly.

var ProfileView = Marionette.ItemView.extend({

profileViewOptions: ['user', 'age'],

initialize: function(options) {

this.mergeOptions(options, this.profileViewOptions);

console.log('The merged options are:', this.user, this.age);

}

});

More information [mergeOptions](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettemergeoptions)

## View.getOption

Retrieve an object's attribute either directly from the object, or from the object's this.options, with this.options taking precedence.

More information [getOption](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettegetoption)

## View.bindEntityEvents

Helps bind a backbone "entity" to methods on a target object. bindEntityEvents is used to support modelEvents and collectionEvents.

More information [bindEntityEvents](http://marionettejs.com/docs/v2.4.1/marionette.functions.html#marionettebindentityevents)

## View.templateHelpers

There are times when a view's template needs to have some  
logic in it and the view engine itself will not provide an  
easy way to accomplish this. For example, Underscore templates  
do not provide a helper method mechanism while Handlebars  
templates do.

A templateHelpers attribute can be applied to any View object that  
renders a template. When this attribute is present its contents  
will be mixed in to the data object that comes back from the  
serializeData method. This will allow you to create helper methods  
that can be called from within your templates. This is also a good place  
to add data not returned from serializeData, such as calculated values.

### Basic Example

<script id="my-template" type="text/html">

I <%= percent %>% think that <%= showMessage() %>

</script>

var MyView = Backbone.Marionette.ItemView.extend({

template: "#my-template",

templateHelpers: function () {

return {

showMessage: function(){

return this.name + " is the coolest!";

},

percent: this.model.get('decimal') \* 100

};

}

});

var model = new Backbone.Model({

name: "Backbone.Marionette",

decimal: 1

});

var view = new MyView({

model: model

});

view.render(); //=> "I 100% think that Backbone.Marionette is the coolest!";

The templateHelpers can also be provided as a constructor parameter  
for any Marionette view class that supports the helpers.

var MyView = Marionette.ItemView.extend({

// ...

});

new MyView({

templateHelpers: {

doFoo: function(){ /\* ... \*/ }

}

});

### Accessing Data Within The Helpers

In order to access data from within the helper methods, you  
need to prefix the data you need with this. Doing that will  
give you all of the methods and attributes of the serialized  
data object, including the other helper methods.

templateHelpers: {

something: function(){

return "Do stuff with " + this.name + " because it's awesome.";

}

}

### Object Or Function As templateHelpers

You can specify an object literal (as shown above), a reference  
to an object literal, or a function as the templateHelpers.

If you specify a function, the function will be invoked  
with the current view instance as the context of the  
function. The function must return an object that can be  
mixed in to the data for the view.

Backbone.Marionette.ItemView.extend({

templateHelpers: function(){

return {

foo: function(){ /\* ... \*/ }

}

}

});

## Change Which Template Is Rendered For A View

There may be some cases where you need to change the template that is  
used for a view, based on some simple logic such as the value of a  
specific attribute in the view's model. To do this, you can provide  
a getTemplate function on your views and use this to return the  
template that you need.

var MyView = Backbone.Marionette.ItemView.extend({

getTemplate: function(){

if (this.model.get("foo")){

return "#some-template";

} else {

return "#a-different-template";

}

}

});

This applies to all view classes.