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

Marionette.isNodeAttached

Marionette.mergeOptions

Marionette.getOption

Marionette.proxyGetOption

Marionette.triggerMethod

Marionette.bindEntityEvents

Marionette.triggerMethodOn

Marionette.bindEntityEvent

Marionette.unbindEntityEvents

Marionette.proxyBindEntityEvents

Marionette.proxyUnbindEntityEvents

Marionette.normalizeMethods

Marionette.normalizeUIKeys

Marionette.normalizeUIValues

Marionette.actAsCollection

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

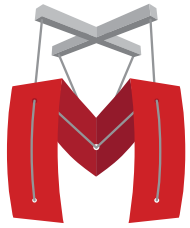
This method is used to bind a backbone "entity" (e.g. collection/model) to methods on a target object.

```
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.



v2.4.4

Application
AppRouter
Behavior
Behaviors
Callbacks
CollectionView
CompositeView
Configuration
Controller
Functions
ItemView

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);
  },
```

LayoutView
Module
Object

```
  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.