

v2.4.4

[Application](#)[AppRouter](#)[Behavior](#)[Behaviors](#)[Callbacks](#)[CollectionView](#)[CompositeView](#)[Configuration](#)[Controller](#)[Functions](#)[ItemView](#)[LayoutView](#)[Module](#)[Object](#)

## ► **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](#)[View onShow](#)[View destroy](#)[View onBeforeDestroy](#)[View "attach" / onAttach event](#)[View "before:attach" / onBeforeAttach event](#)[View "dom:refresh" / onDomRefresh event](#)[View.triggers](#)[View.events](#)[View.modelEvents and View.collectionEvents](#)

[View.serializeModel](#)[View.bindUIElements](#)[View.mergeOptions](#)[View.getOption](#)[View.bindEntityEvents](#)[View.templateHelpers](#)[Basic Example](#)[Accessing Data Within The Helpers](#)[Object Or Function As `templateHelpers`](#)[Change Which Template Is Rendered For A View](#)[UI Interpolation](#)

## ► 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 = 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](#).

All views that inherit from the base `Marionette.View` class have this functionality, notably `ItemView`, `CollectionView`, `CompositeView`, and `LayoutView`.

```
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 = 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 = 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 such as jQueryUI or KendoUI.

```
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](#).

## ► View.events

Since Views extend from backbone's view class, you gain the benefits of the [events hash](#).

Some preprocessing sugar is added on top to add the ability to cross utilize the `ui` hash.

```
var MyView = 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](#) 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 = 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.

```
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

```
Marionette.CompositeView.extend({
  triggers: function(){
    return {
      "click .that-thing": "that:i:sent:you"
    };
  }
});
```

Trigger keys can be configured to cross utilize the `ui` hash.

```
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 = 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.


```
Marionette.CompositeView.extend({

  modelEvents: {
    "change:name": "nameChanged" // equivalent to view.listenTo(view.model, "change:name", v:
  },

  collectionEvents: {
    "add": "itemAdded" // equivalent to view.listenTo(view.collection, "add", view.itemAdded.
  },

  // ... 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.

```
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.

```
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.

```
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](#)

## ► 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](#)

## ► View.bindEntityEvents

Helps bind a backbone "entity" to methods on a target object. `bindEntityEvents` is used to support `modelEvents` and `collectionEvents`.

More information [bindEntityEvents](#)

## ► 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 = 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: "Marionette",  
  decimal: 1  
});  
  
var view = new MyView({  
  model: model  
});  
  
view.render(); //=> "I 100% think that 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.

```
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 = Marionette.ItemView.extend({
  getTemplate: function(){
    if (this.model.get("foo")){
      return "#some-template";
    } else {
      return "#a-different-template";
    }
  }
});
```

This applies to all view classes.

## ► UI Interpolation

Marionette UI offers a convenient way to reference jQuery elements. UI elements can also be interpolated into event and region selectors.

In this example, the buy button is referenced in a DOM event and the checkout section is referenced in the region selector.

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

```
ui: {  
  buyButton: '.buy-button',  
  checkoutSection: '.checkout-section'  
},  
  
events: {  
  'click @ui.buyButton': 'onClickBuyButton'  
},  
  
regions: {  
  checkoutSection: '@ui.checkoutSection'  
},  
  
onShow: function() {  
  this.getRegion('checkoutSection').show(new CheckoutSection({  
    model: this.checkoutModel  
  }));  
}  
});
```