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

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 `LayoutView`s:

```
var AppLayoutView = 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 Marionette.Region({
  el: "#someElement"
});
```

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

```
var mgr = new Marionette.Region({
  el: document.querySelector("body")
});
```

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

```
var mgr = new 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, options);

// empties the current view
MyApp.mainRegion.empty();
```

The `options` object is optional. If provided, it will be passed to the events raised during `show` (except for `before:empty` and `empty`). Special properties that change the behavior of `show` include `preventDestroy` and `forceShow`.

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 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 or `show()` options.

```
// 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});
```

Or you can leave the events on by default but disable them for a single show.

```
// This region will trigger attach events by default but not for this particular show.
myRegion.show(myView, {triggerBeforeAttach: false, triggerAttach: false});
```

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 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

A region will raise a few events on itself and on the target view when showing and destroying views.

Events Raised on the Region During `show()`

`before:show` / `onBeforeShow` - Called after the view has been rendered, but before its been displayed.

`show` / `onShow` - Called when the view has been rendered and displayed.

`before:swap` / `onBeforeSwap` – Called before a new view is shown. NOTE: this will only be called when a view is being swapped, not when the region is empty.

`swap` / `onSwap` – Called when 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 before a new view swapped in. NOTE: this will only be called when a view is being swapped, not when the region is empty.

`swapOut` / `onSwapOut` – Called 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 before the view has been emptied.

`empty` / `onEmpty` – Called when the view has been emptied.

▸ Events Raised on the View During `show()`

`before:render` / `onBeforeRender` – Called before the view is rendered.

`render` / `onRender` – Called after the view is rendered, but before it is attached to the DOM.

`before:show` / `onBeforeShow` – Called after the view has been rendered, but before it has been bound to the region.

`before:attach` / `onBeforeAttach` – Called before the view is attached to the DOM. This will not fire if the Region itself is not attached.

`attach` / `onAttach` – Called after the view is attached to the DOM. This will not fire if the Region itself is not attached.

`show` / `onShow` – Called when the view has been rendered and bound to the region.

`dom:refresh` / `onDomRefresh` – Called when the view is both rendered and shown, but only if it is attached to the DOM. This will not fire if the Region itself is not attached.

`before:destroy` / `onBeforeDestroy` – Called before destroying a view.

`destroy` / `onDestroy` – Called after destroying a view.

Note: `render`, `destroy`, and `dom:refresh` are triggered on pure Backbone Views during a show, but for a complete implementation of these events the Backbone View should fire `render` within `render()` and `destroy` within `remove()` as well as set the following flags:

```
view.supportsRenderLifecycle = true;
view.supportsDestroyLifecycle = true;
```

Example Event Handlers

```
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){
  // manipulate the `view` or do something extra
  // with the `region`
```

```
});

var MyRegion = 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 = 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 = 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 = 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 = Marionette.Region.extend({
  el: "#some-div",

  initialize: function(options){
    // your init code, here
  }
});

MyApp.someRegion = new SomeRegion();

MyApp.someRegion.show(someView, options);
```

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.