Backbone.Marionette Documentation

Release 1.0.0-beta3

Derick Bailey

October 31, 2012

CONTENTS

Contents:

CHAPTER

ONE

MARIONETTE.APPLICATION.MODULE

Marionette allows you to define a module within your application, including sub-modules hanging from that module. This is useful for creating modular, encapsulated applications that are split apart in to multiple files.

Marionette's module allow you to have unlimited sub-modules hanging off your application, and serve as an event aggregator in themselves.

1.1 Basic Usage

A module is defined directly from an Application object as the specified name:

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

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

MyApp.MyModule; // => a new Marionette.Application object

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

If you specify the same module name more than once, the first instance of the module will be retained and a new instance will not be created.

1.2 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 easier unit testing of modules in isolation as you can start only the module that you need in your tests.

1.2.1 Starting Modules

Starting a module is done in one of two ways:

1. Automatically with the parent module (or Application) `.start()` method
2. Manually call the `.start()` method on the module

In this example, the module will be started 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 and defined after the `app.start()` call will still be started automatically.

1.2.2 Preventing Auto-Start Of Modules

If you wish to manually start a module instead of having the application start it, you can tell the module definition not to start with the parent:

```
var fooModule = MyApp.module("Foo", { startWithParent: false, define:
    function(){ // module code goes here } });

// start the app without starting the module MyApp.start();

// later, start the module fooModule.start();
```

Note the use of an object literal instead of just a function to define the module, and the presence of the `startWithParent` attribute, to tell it not to start with the application. Then to start the module, the module's `start` method is manually called.

You can also grab a reference to the module at a later point in time, to start it:

```
MyApp.module("Foo", { startWithParent: false, define: function(){
    /*...*/ }
});

// start the module by getting a reference to it first
MyApp.module("Foo").start();
```

1.2.3 Starting Sub-Modules With Parent

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.

Submodules default to starting with their parent module.

```
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 set to start with the app.

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

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

MyApp.module("Foo.Bar", { startWithParent: true, define: function(){...}});

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();
```

1.2.4 Stopping Modules

A module can be stopped, or shut down, to clear memory and resources when the module is no longer needed. Like 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 it's 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. The exception to this is that stopping a parent module will stop all of it's sub-modules.

```
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 With . Notation".

1.3 Defining Sub-Modules With . Notation

Sub-modules or child modules can be defined as a hierarchy of modules and sub-modules all at once:

```
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 will be created for you if they don't exist. If they do exist, though, the existing module will be used instead of creating a new one.

1.4 Module Definitions

You can specify a callback function to provide a definition for the module. Module definitions are invoked immediately on calling `module` method.

The module definition callback will receive 6 parameters:

- The module itself
- The Parent module, or Application object that `.module` was called from

- Backbone
- Backbone.Marionette
- jQuery
- Underscore
- Any custom arguments

You can add functions and data directly to your module to make them publicly accessible. You can also add private functions and data by using locally scoped variables.

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

    // 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
```

1.4.1 Module Initializers

Modules have initializers, similarly to `Application` objects. A module's initializers are run when the module is started.

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

    Foo.addInitializer(function(){
        // initialize and start the module's running code, here.
    });

});
```

Any way of starting this module will cause it's initializers to run. You can have as many initializers for a module as you wish.

1.4.2 Module Finalizers

Modules also have finalizers that are run when a module is stopped.

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

    Foo.addFinalizer(function(){
        // tear down, shut down and clean up the module, here
    });
});
```

Calling the `stop` method on the module will run all that module's finalizers. A module can have as many finalizers as you wish.

1.5 The Module's `this` Argument

The module's `this` argument is set to the module itself.

```
MyApp.module("Foo", function(Foo){    this === Foo; //=> true });
```

1.6 Custom Arguments

You can provide any number of custom arguments to your module, after the module definition function. This will allow you to import 3rd party libraries, and other resources that you want to have locally scoped to your module.

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

    // Lib1 === LibraryNumber1; // Lib2 === LibraryNumber2; // LibEtc ===
    LibraryNumberEtc;

}, LibraryNumber1, LibraryNumber2, LibraryNumberEtc);
```

1.7 Splitting A Module Definition Apart

Sometimes a module gets to be too long for a single file. In this case, you can split a module definition across multiple files:

```
MyApp.module("MyModule", function(MyModule){
    MyModule.definition1 = true; });

MyApp.module("MyModule", function(MyModule){ MyModule.definition2 =
true; });

MyApp.MyModule.definition1; //=> true MyApp.MyModule.definition2; //=>
true
```

CHAPTER

TWO

MARIONETTE.APPLICATION

The `Backbone.Marionette.Application` object is the hub of your composite application. It organizes, initializes and coordinate the various pieces of your app. It also provides a starting point for you to call into, from your HTML script block or from your JavaScript files directly if you prefer to go that route.

The `Application` is meant to be instantiated directly, although you can extend it to add your own functionality.

`js MyApp = new Backbone.Marionette.Application();`

2.1 Documentation Index

2.2 Adding Initializers

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.

‘‘‘js 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` parameters 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.

2.3 Application Event

The `Application` object raises a few events during its lifecycle, using the Marionette.triggerMethod 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 the `Backbone.history`.

The events that are currently triggered, are:

- **"initialize:before" / `onInitializeBefore` **: fired just before the initializers kick off
- **"initialize:after" / `onInitializeAfter` **: fires just after the initializers have finished
- **"start" / ``onStart``**: fires after all initializers and after the initializer events

```js MyApp.on("initialize:before", function(options){ options.moreData = "Yo dawg, I heard you like options so I put some options in your options!" });

MyApp.on("initialize:after", function(options){ if (Backbone.history){ Backbone.history.start(); } }); ```

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

## 2.4 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, at initialization/start of the app, instead of just at definition.

```js var options = { something: "some value", another: "#some-selector" };

MyApp.start(options); ```

## 2.5 app.vent: Event Aggregator

Every application instance comes with an instance of `Marionette.EventAggregator` called `app.vent`.

```js MyApp = new Backbone.Marionette.Application();

MyApp.vent.on("foo", function(){ alert("bar"); });

MyApp.vent.trigger("foo"); // => alert box "bar" ```

See the '`Marionette.EventAggregator` <./marionette.eventaggregator.md>'_ documentation for more details.
```

2.6 Regions And The Application Object

Marionette's `Region` objects can be directly added to an application by calling the `addRegions` method.

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

2.6.1 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:

`js MyApp.addRegions({ someRegion: "#some-div", anotherRegion: "#another-div" });`

2.6.2 Custom Region Type

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

‘‘‘js MyCustomRegion = Marionette.Region.extend({ el: “#foo” });

MyApp.addRegions({ someRegion: MyCustomRegion }); ‘‘‘

2.6.3 Custom Region Type And Selector

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

‘‘‘js MyCustomRegion = Marionette.Region.extend({});

MyApp.addRegions({

someRegion: { selector: “#foo”, regionType: MyCustomRegion },

anotherRegion: { selector: “#bar”, regionType: MyCustomRegion }

}); ‘‘‘

2.6.4 Removing Regions

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

`js MyApp.removeRegion('someRegion');`

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

For more information on regions, see the region documentation

CHAPTER

THREE

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.

3.1 Documentation Index

3.2 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. Instead of providing a callback method that exists on the router, though, you provide a callback method that exists on the `controller` that you specify for the router instance (see below).

‘‘‘js 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.

3.3 Specify A Controller

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

‘‘‘js someController = { someMethod: function(){ /.../ } };

Backbone.Marionette.AppRouter.extend({ controller: someController }); ‘‘‘

Or in a parameter to the constructor:

‘‘‘js myObj = { someMethod: function(){ /.../ } };

new MyRouter({ controller: myObj }); ‘‘‘

Or

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.

CHAPTER

FOUR

MARIONETTE.CALLBACKS

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.

4.1 Documentation Index

4.2 Basic Usage

‘‘‘js 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.

4.3 Specify Context Per-Callback

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

‘‘‘js 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 ignore 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`.

4.4 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.

CHAPTER

FIVE

MARIONETTE.COLLECTIONVIEW

The `CollectionView` will loop through all of the models in the specified collection, render each of them using a specified `itemView`, then append the results of the item view's `el` to the collection view's `el`.

5.1 Documentation Index

- CollectionView close

5.2 CollectionView's `itemView`

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

‘‘‘js MyItemView = Backbone.Marionette.ItemView.extend({});

Backbone.Marionette.CollectionView.extend({ itemView: MyItemView }); ‘‘‘

Alternatively, you can specify an `itemView` in the options for the constructor:

‘‘‘js MyCollectionView = Backbone.Marionette.CollectionView.extend({...});

new MyCollectionView({ itemView: MyItemView }); ‘‘‘

If you do not specify an `itemView`, an exception will be thrown stating that you must specify an `itemView`.

If you need a view specific to your model, you can override `getItemView`:

`js Backbone.Marionette.CollectionView.extend({ getItemView: function(item) { // some logic to calculate which view to return return someItemSpecificView; } })`

5.3 CollectionView's `itemViewOptions`

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

‘‘‘js ItemView = Backbone.Marionette.ItemView({ initialize: function(options){ console.log(options.foo); // => “bar” } });

CollectionView = Backbone.Marionette.CollectionView({ itemView: ItemView,

itemViewOptions: { foo: “bar” } }); ‘‘‘

You can also specify the `itemViewOptions` 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 `itemViewOptions`. The function must return an object, and the attributes of the object will be copied to the itemView instance’ options.

`js CollectionView = Backbone.Marionette.CollectionView({ itemViewOptions: function(model) { // do some calculations based on the model return { foo: "bar" } } });`

5.4 CollectionView's `emptyView`

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

‘‘‘js NoItemsView = Backbone.Marionette.ItemView.extend({ template: “#show-no-items-message-template” });

Backbone.Marionette.CollectionView.extend({ // ...

emptyView: NoItemsView }); ‘‘‘

This will render the `emptyView` and display the message that needs to be displayed when there are no items.

5.5 CollectionView's `buildItemView`

When a custom view instance needs to be created for the `itemView` that represents an item, override the `buildItemView` method. This method takes three parameters and returns a view instance to be used as the item view.

`js buildItemView: function(item, ItemViewType, itemViewOptions){ // build the final list of options for the item view type var options = _.extend({model: item}, itemViewOptions); // create the item view instance var view = new ItemViewType(options); // return it return view; },`

5.6 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.

5.6.1 onBeforeRender callback

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

`js Backbone.Marionette.CollectionView.extend({ onBeforeRender: function(){ // do stuff here } });`

5.6.2 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:

`js Backbone.Marionette.CollectionView.extend({ onRender: function(){ // do stuff here } });`

5.6.3 onItemAdded callback

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

`js Backbone.Marionette.CollectionView.extend({ onItemAdded: function(itemView){ // work with the itemView instance, here } });`

5.6.4 onBeforeClose callback

This method is called just before closing the view.

`js Backbone.Marionette.CollectionView.extend({ onBeforeClose: function(){ // do stuff here } });`

5.6.5 onClose callback

This method is called just after closing the view.

`js Backbone.Marionette.CollectionView.extend({ onClose:  function(){ // do stuff here } });`

5.7 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 function, which calls a corresponding "on{EventName}" method on the view instance.

5.7.1 "before:render" / onBeforeRender event

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

```js MyView = Backbone.Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("before:render", function(){ alert("the collection view is about to be rendered"); });

myView.render(); ```

### 5.7.2 "render" / onRender event

A "collection:rendered" / `onCollectionRendered` 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.

```js MyView = Backbone.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(); ```

### 5.7.3 "before:close" / onBeforeClose event

Triggered just before closing the view. A "collection:before:close" / `onCollectionBeforeClose` event will also be fired

```js MyView = Backbone.Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("collection:before:close", function(){ alert("the collection view is about to be closed"); });

myView.close(); ```
```

5.7.4 "closed" / "collection:closed" event

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

‘‘‘js MyView = Backbone.Marionette.CollectionView.extend({...});

var myView = new MyView();

myView.on("collection:closed", function(){ alert("the collection view is now closed"); });

myView.close(); ‘‘‘

5.7.5 "item:added" / onItemAdded

Triggered just after creating a new itemView instance for an item that was added to the collection, but before the view is rendered and added to the DOM.

`js cv.on("item:added", function(viewInstance){ // ...  });`

5.7.6 "item:removed" / onItemRemoved

Triggered after an itemView instance has been closed and removed, when it's item was deleted or removed from the collection.

`js cv.on("item:removed", function(viewInstance){ // ...  });`

5.7.7 "itemview:*" event bubbling from child views

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

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

‘‘‘js // set up basic collection var myModel = new MyModel(); var myCollection = new MyCollection(); myCollection.add(myModel);

// get the collection view in place colView = new CollectionView({ collection: myCollection }); colView.render();

// bind to the collection view's events that were bubbled // from the child view colView.on("itemview:do:something", function(childView, msg){ alert("I said, '" + msg + "'"); });

// hack, to get the child view and trigger from it var childView = colView.children[myModel.cid]; childView.trigger("do:something", "do something!"); ‘‘‘

The result of this will be an alert box that says "I said, 'do something!'".

Also note that you would not normally grab a reference to the child view the way this is showing. I'm merely using that hack as a way to demonstrate the event bubbling. Normally, you would have your item view listening to DOM events or model change events, and then triggering an event of it's own based on that.

5.8 CollectionView render

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

‘‘‘js MyCollectionView = Backbone.Marionette.CollectionView.extend({...});

new MyCollectionView().render().done(function(){ // all of the children are now rendered. do stuff here. }); ‘‘‘

5.9 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 item views.

When a model is removed from a collection (or destroyed / deleted), the collection view will close and remove that model’s item view.

5.10 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 closing all of the child views that may have previously been opened.

5.11 CollectionView’s appendHtml

By default the collection view will call jQuery’s `.append` to move the HTML contents from the item view instance in to the collection view’s `el`.

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

‘‘‘js Backbone.Marionette.CollectionView.extend({

appendHtml: function(collectionView, itemView, index){ collectionView.$el.prepend(itemView.el); }

}); ‘‘‘

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

The third parameter, `index`, is the index of the model that this itemView 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.

5.12 CollectionView close

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

- unbind all `bindTo` events
- unbind all custom view events
- unbind all DOM events
- unbind all item views that were rendered
- remove `this.el` from the DOM

- call an `onClose` event on the view, if one is provided

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

`js Backbone.Marionette.CollectionView.extend({ onClose:  function(){ // custom cleanup or closing code, here } });`

CHAPTER

SIX

MARIONETTE.COMMANDS

An application level command execution system. This allows components in an application to state that some work needs to be done, but without having to be explicitly coupled to the component that is performing the work.

No response is allowed from the execution of a command. It's a "fire-and-forget" scenario.

Facilitated by Backbone.Wreqr's Commands object.

6.1 Documentation Index

6.2 Register A Command

To register a command, call `App.commands.addHandler` and provide a name for the command to handle, and a callback method.

‘‘‘js var App = new Marionette.Application();

App.commands.addHandler("foo", function(bar){ console.log(bar); }); ‘‘‘

6.3 Execute A Command

To execute a command, either call `App.commands.execute` or the more direct route of `App.execute`, providing the name of the command to execute and any parameters the command needs:

`js App.execute("foo", "baz"); // outputs "baz" to the console, from command registered above`

6.4 Remove / Replace Commands

To remove a command, call `App.commands.removeHandler` and provide the name of the command to remove.

To remove all commands, call `App.commands.removeAllHandlers()`.

To replace a command, simply register a new handler for an existing command name. There can be only one command handler for a given command name.

CHAPTER

SEVEN

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.

For example, if you're rendering a treeview control, you may want to render a collection view with a model and template so that it will show a parent item 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`.

'''js 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

7.1 Documentation Index

7.2 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:

`js new MyComp({ template:  "#some-template" });`

7.3 CompositeView's `itemViewContainer`

By default the composite view uses the same `appendHtml` method that the collection view provides. This means the view will call jQuery's `.append` to move the HTML contents from the item 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 item views should be placed.

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

‘‘‘html

‘‘‘

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

‘‘‘js RowView = Backbone.Marionette.ItemView.extend({ tagName: "tr", template: "#row-template" });

TableView = Backbone.Marionette.CompositeView.extend({ itemView: RowView,

// specify a jQuery selector to put the itemView instances in to itemViewContainer: "tbody",

template: "#table-template" }); ‘‘‘

This will put all of the `itemView` instances in to the `<tbody>` tag of the composite view's rendered template, correctly producing the table structure.

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

‘‘‘js TableView = Backbone.Marionette.CompositeView.extend({ // ...

itemViewContainer: function(){ return "#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.

7.4 CompositeView's `appendHtml`

Sometimes the `itemViewContainer` configuration is insuficient for specifying where the itemView instance should be placed. If this is the case, you can override the `appendHtml` method with your own implementation.

For example:

‘‘‘js TableView = Backbone.Marionette.CompositeView.extend({ itemView: RowView,

template: "#table-template",

appendHtml: function(collectionView, itemView, index){ collectionView.$("tbody").append(itemView.el); } }); ‘‘‘

For more information on the parameters of this method, see the CollectionView's documentation.

7.5 Recursive By Default

The default rendering mode for a `CompositeView` assumes a hierarchical, recursive structure. If you configure a composite view without specifying an `itemView`, you'll get the same composite view type rendered for each item in the collection. If you need to override this, you can specify a `itemView` in the composite view's definition:

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

var CompView = Backbone.Marionette.CompositeView.extend({ itemView: ItemView }); ‘‘‘

7.6 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 item to the rendered list
- When the collection has a model removed (the “remove” event is fired), it will remove that one item from the rendered list

You can also manually re-render either or both of them:

- If you want to re-render everything, call the `.render()` method
- If you want to re-render the model’s view, you can call `.renderModel()`
- If you want to re-render the collection’s views, you can call `.renderCollection()`

7.7 Events And Callbacks

During the course of rendering a composite, several events will be triggered. These events are triggered with the Marionette.triggerMethod function, which calls a corresponding “on{EventName}” method on the view.

- “composite:item:rendered” / `onCompositeItemRendered` - after the `modelView` has been rendered
- “composite:collection:rendered” / `onCompositeCollectionRendered` - after the collection of models has been rendered
- “render” / `onRender` and “composite:rendered” / `onCompositeRendered` - 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:

`js Backbone.Marionette.CompositeView.extend({ onRender: function(){ // do stuff here } });`

7.8 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.

7.9 modelEvents and collectionEvents

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

‘‘‘js Marionette.CompositeView.extend({ modelEvents: { “change”: “modelChanged” },

collectionEvents: { “add”: “modelAdded” } }); ‘‘‘

For more information, see the Marionette.View documentation.

CHAPTER

EIGHT

MARIONETTE.CONTROLLER

A multi-purpose object to use as a controller for modules and routers, and as a mediator for workflow and coordination of other objects, views, and more.

8.1 Documentation Index

8.2 Basic Use

A `Marionette.Controller` can be extended, like other Backbone and Marionette objects. It supports the standard `initialize` method, has a built-in `EventBinder`, and can trigger events, itself.

‘‘‘js // define a controller var MyController = Marionette.Controller.extend({

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

doStuff: function(){ this.trigger(“stuff:done”, this.stuff); }

});

// create an instance var c = new MyController({ stuff: “some stuff” });

// use the built in EventBinder c.bindTo(c, “stuff:done”, function(stuff){ console.log(stuff); });

// do some stuff c.doStuff(); ‘‘‘

8.3 On The Name 'Controller'

The name `Controller` is bound to cause a bit of confusion, which is rather unfortunate. There was some discussion and debate about what to call this object, the idea that people would confuse this with an MVC style controller came up a number of times. In the end, we decided to call this a controller anyways, as the typical use case is to control the workflow and process of an application and / or module.

But the truth is, this is a very generic, multi-purpose object that can serve many different roles in many different scenarios. We are always open to suggestions, with good reason and discussion, on renaming objects to be more descriptive, less confusing, etc. If you would like to suggest a different name, please do so in either the mailing list or the github issues list.

CHAPTER

NINE

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.

9.1 Documentation Index

9.2 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.

‘‘‘js var template = “#some-template”; var data = {foo: “bar”}; var html = Backbone.Marionette.Renderer.render(template, data);

// do something with the HTML here ‘‘‘

9.3 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.

`js 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.

9.4 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:

```
js 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:

```
js Backbone.Marionette.Renderer.render = function(template, data){ var
template = Marionette.TemplateCache.get(template); // Do something with the
template here };
```

9.5 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.

```
js 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:

‘‘‘js var myPrecompiledTemplate = _.template(“

some template

”);

Backbone.Marionette.ItemView.extend({ template: myPrecompiledTemplate }); ‘‘‘

CHAPTER

TEN

MARIONETTE.EVENTAGGREGATOR

An event aggregator is an application level pub/sub mechanism that allows various pieces of an otherwise segmented and disconnected system to communicate with each other.

Marionette's EventAggregator is facilitated by Backbone.Wreqr's EventAggregator object and Backbone.EventBinder. It combines an EventBinder in to the EventAggregator instance.

10.1 Documentation Index

10.2 Basic Usage

Marionette provides an event aggregator with each application instance: `MyApp.vent`. You can also instantiate your own event aggregator:

`js myVent = new Marionette.EventAggregator();`

Passing an object literal of options to the constructor function will extend the event aggregator with those options:

`js myVent = new Marionette.EventAggregator({foo: "bar"}); myVent.foo // => "bar"`

10.3 BindTo

The `EventAggregator` mixes in an EventBinder object to easily track and unbind all event callbacks, including inline callback functions.

‘‘‘js vent = new Marionette.EventAggregator();

vent.bindTo(vent, “foo”, function(){ alert(“bar”); });

vent.unbindAll();

vent.trigger(“foo”); // => nothing. all events have been unbound. ‘‘‘

10.4 Decoupling With An Event-Driven Architecture

You can use an event aggregator to communicate between various modules of your application, ensuring correct decoupling while also facilitating functionality that needs more than one of your application's modules.

‘‘‘js var vent = new Marionette.EventAggregator();

vent.on(“some:event”, function(){ alert(“Some event was fired!!!!”); });

vent.trigger(“some:event”); ‘‘‘

For a more detailed discussion and example of using an event aggregator with Backbone applications, see the blog post: References, Routing, and The Event Aggregator: Coordinating Views In Backbone.js.

CHAPTER

ELEVEN

MARIONETTE.EVENTBINDER

The `EventBinder` object provides event binding management for related events, across any number of objects that trigger the events. This allows events to be grouped together and unbound with a single call during the clean-up of an object that is bound to the events.

11.1 Documentation Index

11.2 Bind Events

‘‘‘js var binder = new Backbone.Marionette.EventBinder();

var model = new MyModel();

var handler = { doIt: function(){} } binder.bindTo(model, “change:foo”, handler.doIt); ‘‘‘

You can optionally specify a 4th parameter as the context in which the callback method for the event will be executed:

`js binder.bindTo(model, "change:foo", someCallback, someContext);`

11.3 Unbind A Single Event

When you call `bindTo`, it returns a “binding” object that can be used to unbind from a single event with the `unbindFrom` method:

‘‘‘js var binding = binder.bindTo(model, “change:foo”, someCallback, someContext);

// later in the code binder.unbindFrom(binding); ‘‘‘

This will unbind the event that was configured with the binding object, and remove it from the EventBinder bindings.

11.4 Unbind All Events

You can call `unbindAll` to unbind all events that were bound with the `bindTo` method:

`js binder.unbindAll();`

This even works with in-line callback functions.

11.5 When To Use EventBinder vs `on` Handlers

See the wiki: When to use the EventBinder

CHAPTER

TWELVE

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 the provide a way to get the same behaviors and conventions from your own code.

12.1 Documentation Index

12.2 Marionette.addEventBinder

Add a Backbone.EventBinder instance to any target object. This method attaches an `eventBinder` to the target object, and then copies the necessary methods to the target while maintaining the event binder in it's own object.

‘‘‘js myObj = {};

Marionette.addEventBinder(myObj);

myObj.bindTo(aModel, “foo”, function(){...}); ‘‘‘

This allows the event binder's implementation to vary independently of it being attached to the view. For example, the internal structure used to store the events can change without worry about it interfering with Marionette's views.

12.3 Marionette.createObject

Marionette provides a method called `Marionette.createObject`. This method is a simple wrapper around / shim for a native `Object.create`, allowing simple prototypal inheritance for various purposes.

There is an intended limitation of only allowing the first parameter for the Object.create method. Since ES “properties” cannot be back-filled in to old versions, the second parameter is not supported.

12.3.1 CAVEAT EMPTOR

This method is not intended to be a polyfill or shim used outside of Marionette. Use at your own risk.

If you need a true polyfill or shim for older browser support, we recommend you include one of the following in your project:

- Modernizr
- cujojs/poly
- ES5-Shim
- Any other proper shim / polyfill for backward compatibility

Be sure to include your preferred shim / polyfill BEFORE any other script files in your app. This will ensure Marionette uses your polyfill instead of the built in `Marionette.createObject`.

12.4 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.

'''js 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 type, with methods var Bar = Foo.extend({

someMethod: function(){ ... }

// ... });

// Create an instance of Bar var b = new Bar(); '''

12.5 Marionette.getOption

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

'''js var M = Backbone.Model.extend({ foo: "bar",

initialize: function(){ 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.

12.6 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:close")` fires the "onBeforeClose" 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){...})`

CHAPTER

THIRTEEN

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.

13.1 Documentation Index

13.2 ItemView render

An item view has a `render` method built in to it, and uses the `Renderer` object to do the actual rendering.

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

‘‘‘js MyView = Backbone.Marionette.ItemView.extend({ template: "#some-template" });

new MyView().render(); ‘‘‘

13.3 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 function, which triggers the event and a corresponding "on{EventName}" method.

13.3.1 "before:render" / onBeforeRender event

Triggered before an ItemView is rendered. Also triggered as "item:before:render" / `onItemBeforeRemder`.

`js Backbone.Marionette.ItemView.extend({ onBeforeRender: function(){ // set up final bits just before rendering the view's 'el' } });`

13.3.2 "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.

Also triggered as "item:render" / `onItemRender`.

`js 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. } });`

13.3.3 "before:close" / onBeforeClose event

Triggered just prior to closing the view, when the view's `close()` method has been called. Also triggered as "item:before:close" / `onItemBeforeClose`.

`js Backbone.Marionette.ItemView.extend({ onBeforeClose: function(){ // manipulate the 'el' here. it's already // been rendered, and is full of the view's // HTML, ready to go. } });`

13.3.4 "close" / onClose event

Triggered just after the view has been closed. Also triggered as "item:close" / `onItemClose`.

`js Backbone.Marionette.ItemView.extend({ onClose: function(){ // custom closing and cleanup goes here } });`

13.4 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:

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

new MyItemView({ template: "#myItemTemplate", model: myModel });

MyItemView.render(); ‘‘‘

`html <script id="myItemTemplate" type="template"> Foo is: <%= foo %> </script>`

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

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

new MyItemView({ template: "#myCollectionTemplate", collection: myCollection });

MyItemView.render(); ‘‘‘

`html <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.

`js Backbone.Marionette.ItemView.extend({ serializeData: function(){ return { "some attribute": "some value" } } });`

13.5 Organizing ui elements

As documented in View, you can specify a `ui` hash in your view that maps between a ui element’s name and its jQuery selector, similar to how regions are organized. This is especially useful if you access the same ui element more than once in your view’s code, so instead of duplicating the selector you can simply reference it by `this.ui.elementName`:

‘‘‘js Backbone.Marionette.ItemView.extend({ tagName: “tr”,

ui: { checkbox: “input[type=checkbox]” },

onRender: function() { if (this.model.get(‘selected’)) { this.ui.checkbox.addClass(‘checked’); } } }); ‘‘‘

13.6 modelEvents and collectionEvents

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

‘‘‘js Marionette.ItemView.extend({ modelEvents: { “change”: “modelChanged” },

collectionEvents: { “add”: “modelAdded” } }); ‘‘‘

For more information, see the Marionette.View documentation.

CHAPTER

FOURTEEN

MARIONETTE.LAYOUT

A `Layout` is a specialized hybrid between an `ItemView` and a collection of `Region` objects, used for rendering an application layout with multiple sub-regions to be managed by specified region managers.

A layout manager can also be used as a composite-view to aggregate multiple views and sub-application areas of the screen where multiple region managers need to be attached to dynamically rendered HTML.

For a more in-depth discussion on Layouts, see the blog post Manage Layouts And Nested Views With Backbone.Marionette

14.1 Documentation Index

14.2 Basic Usage

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

`html <script id="layout-template" type="text/template"> <section> <navigation id="menu">...</navigation> <article id="content">...</article> </section> </script>`

‘‘‘js AppLayout = Backbone.Marionette.Layout.extend({ template: “#layout-template”,

regions: { menu: “#menu”, content: “#content” } });

var layout = new AppLayout(); layout.render(); ‘‘‘

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

‘‘‘js layout.menu.show(new MenuView());

layout.content.show(new MainContentView()); ‘‘‘

14.3 Region Availability

Any defined regions within a layout will be available to the layout or any calling code immediately after instantiating the layout. This allows a layout 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 layout 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.

14.4 Re-Rendering A Layout

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

The first time a layout 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 layout.

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

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

14.4.1 Avoid Re-Rendering The Entire Layout

There are times when re-rendering the entire layout 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 layout and all of the views that the layout is displaying.

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

14.5 Nested Layouts And Views

Since the `Layout` 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.

‘‘‘js MyApp = new Backbone.Marionette.Application(); MyApp.addRegions({ mainRegion: “#main” });

var layout = new AppLayout(); MyApp.mainRegion.show(layout);

layout.show(new MenuView()); ‘‘‘

You can nest layouts into region managers as deeply as you want. This provides for a well organized, nested view structure.

14.6 Closing A Layout

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

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

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

14.7 Custom Region Type

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 `regionType` propery of the `Layout`.

`js MyLayout = Backbone.Marionette.Layout.extend({ regionType: SomeCustomRegion });`

You can also specify custom `Region` classes for each `region`:

‘‘‘js AppLayout = Backbone.Marionette.Layout.extend({ template: “#layout-template”,

regionType: SomeDefaultCustomRegion,

regions: { menu: { selector: “#menu”, regionType: CustomRegionTypeReference }, content: { selector: “#content”, regionType: CustomRegionType2Reference } } }); ‘‘‘

CHAPTER

FIFTEEN

MARIONETTE.REGION

Region managers provide a consistent way to manage your views and show / close them in your application. They use a jQuery selector to show your views in the correct place. They also call extra methods on your views to facilitate additional functionality.

15.1 Documentation Index

15.2 Defining An Application Region

Regions can be added to the application 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.

`js MyApp.addRegions({ mainRegion: "#main-content", navigationRegion: "#navigation" });`

As soon as you call `addRegions`, your region managers 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.

15.3 Initialize A Region With An `el`

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

`js var mgr = new Backbone.Marionette.Region({ el: "#someElement" });`

15.4 Basic Use

Once a region manager has been defined, you can call the `show` and `close` methods on it to render and display a view, and then to close that view:

‘‘‘js var myView = new MyView();

// render and display the view MyApp.mainRegion.show(myView);

// closes the current view MyApp.mainRegion.close(); ‘‘‘

If you replace the current view with a new view by calling `show`, it will automatically close the previous view.

‘‘‘js // show the first view var myView = new MyView(); MyApp.mainRegion.show(myView);

// replace view with another. the // `close` method is called for you var anotherView = new AnotherView(); MyApp.mainRegion.show(anotherView); ‘‘‘

15.5 `reset` A Region

A region can be `reset` at any time. This will close any existing view that is being displayed, and delete the cached `el`. The next time the region is used to show a view, the region’s `el` will be queried from the DOM.

`js myRegion.reset();`

This is useful for scenarios where a region is re-used across view instances, or in unit testing.

15.6 Set How View’s `el` Is Attached

If you need to change how the view is attached to the DOM when showing a view via a region, override the `open` method of the region. This method receives one parameter - the view to show.

The default implementation of `open` is:

```
js Marionette.Region.prototype.open = function(view){ this.$el.html(view.el);
}
```

This will replace the contents of the region with the view’s `el` / content. You can change to this be anything you wish, though, facilitating transition effects and more.

```
js Marionette.Region.prototype.open = function(view){ this.$el.hide();
this.$el.html(view.el); this.$el.slideDown("fast"); }
```

This example will cause a view to slide down from the top of the region, instead of just appearing in place.

15.7 Attach Existing View

There are some scenarios where it's desirable to attach an existing view to a region manager, without rendering or showing the view, and without replacing the HTML content of the region. For example, SEO and accessibiliy 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 manager's constructor
- call `attachView` on the region manager instance

15.7.1 Set `currentView` On Initialization

‘‘‘js var myView = new MyView({ el: $(“#existing-view-stuff”) });

var manager = new Backbone.Marionette.Region({ el: “#content”, currentView: myView }); ‘‘‘

15.7.2 Call `attachView` On Region

‘‘‘js MyApp.addRegions({ someRegion: “#content” });

var myView = new MyView({ el: $(“#existing-view-stuff”) });

MyApp.someRegion.attachView(myView); ‘‘‘

15.8 Region Events And Callbacks

A region manager will raise a few events during it's showing and closing of views:

- “show” / `onShow` - called on the view instance when the view has been rendered and displayed
- “show” / `onShow` - called on the region isntance when the view has been rendered and displayed
- “close” / `onClose` - when the view has been closed

You can bind to these events and add code that needs to run with your region manager, opening and closing views.

‘‘‘‘js MyApp.mainRegion.on(“show”, function(view){ // manipulate the‘‘view‘‘or do something extra // with the region manager via‘‘this‘ });

MyApp.mainRegion.on(“closed”, function(view){ // manipulate the `view` or do something extra // with the region manager via `this` });

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

onShow: function(view){ // the `view` has been shown } });

MyView = Marionette.ItemView.extend({ onShow: function(){ // called when the view has been shown } }); ‘‘‘

15.8.1 View Callbacks And Events For Regions

The region manager will call an `onShow` method on the view that was displayed. It will also trigger a “show” event from the view:

‘‘‘js MyView = Backbone.View.extend({ onShow: function(){ // the view has been shown } });

view = new MyView();

view.on("show", function(){ // the view has been shown. });

MyApp.mainRegion.show(view); ```

15.9 Custom Region Types

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

15.9.1 Attaching Custom Region Types

Once you define a region manager type, you can attach the new region type by specifying the region type as the value - not an instance of it, but the actual constructor function.

```js 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.

```js var FooterRegion = Backbone.Marionette.Region.extend({ el: "#footer" });

MyApp.addRegions({ footerRegion: { selector: "#footer", type: 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 Layout, the region must provide an `el` either in it's definition or constructor options.

### 15.9.2 Instantiate Your Own Region

There may be times when you want to add a region manager 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:

```js 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.
```

CHAPTER

SIXTEEN

MARIONETTE.REQUESTRESPONSE

An application level request/response system. This allows components in an application to request some information or work be done by another part of the app, but without having to be explicitly coupled to the component that is performing the work.

A return response is expected when making a request.

Facilitated by Backbone.Wreqr's RequestResponse object.

16.1 Documentation Index

16.2 Register A Request Handler

To register a command, call `App.reqres.addHandler` and provide a name for the command to handle, and a callback method.

```js var App = new Marionette.Application();

App.reqres.addHandler("foo", function(bar){ return bar + "-quux"; }); ```

## 16.3 Request A Response

To execute a command, either call `App.reqres.request` or the more direct route of `App.request`, providing the name of the command to execute and any parameters the command needs:

`js App.request("foo", "baz"); // => returns "baz-quux"`

## 16.4 Remove / Replace A Request Handler

To remove a request handler, call `App.reqres.removeHandler` and provide the name of the request handler to remove.
```

To remove all request handlers, call `App.reqres.removeAllHandlers()`.

To replace a request handler, simply register a new handler for an existing request handler name. There can be only one request handler for a given request name.

CHAPTER

SEVENTEEN

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.

17.1 Documentation Index

17.2 Basic Usage

To use the `TemplateCache`, call the `get` method on TemplateCache directly. Internally, instances of the TemplateCache type will be created and stored but you do not have to manually create these instances yourself.

`js var promise = Backbone.Marionette.TemplateCache.get("#my-template"); promise.done(function(template){ // use the template here });`

Making multiple calls to get the same template will retrieve the template from the cache on subsequence calls.

17.3 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.

‘‘‘js Backbone.Marionette.TemplateCache.prototype.loadTemplate = function(templateId){ // load your template here, returning a compiled template or function // that returns the rendered HTML var myTemplate = compileMyTemplate(“some template”);

// send the template back return myTemplate; } ‘‘‘

17.4 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 (or from the overriden `loadTemplate` function) the next time it is retrieved.

If you do not specify any parameters, all items will be cleared from the cache:

‘‘‘js 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:

‘‘‘js 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”) ‘‘‘

CHAPTER

EIGHTEEN

MARIONETTE.VIEW

Marionette has a base `Marionette.View` type 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` type is not intended to be used directly. It exists as a base view for other view types to be extended from, and to provide a common location for behaviors that are shared across all views.

18.1 Documentation Index

18.2 Binding To View Events

Marionette.View extends `Marionette.BindTo`. It is recommended that you use the `bindTo` method to bind model, collection, or other events from Backbone and Marionette objects.

‘‘‘js MyView = Backbone.Marionette.ItemView.extend({ initialize: function(){ this.bindTo(this.model, “change:foo”, this.modelChanged); this.bindTo(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 passing in the context object as the 4th parameter of `bindTo`.

18.3 ItemView close

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

- unbind all `bindTo` events
- unbind all custom view events
- unbind all DOM events
- remove `this.el` from the DOM
- call an `onClose` event on the view, if one is provided

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

`js Backbone.Marionette.ItemView.extend({ onClose:  function(){ // custom cleanup or closing code, here } });`

18.4 View.triggers

Views can define a set of `triggers` as a hash, which will convert a DOM event in to a `view.trigger` event.

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.

‘‘‘js MyView = Backbone.Marionette.ItemView.extend({ // ...

triggers: { “click .do-something”: “something:do:it” } });

view = new MyView(); view.render();

view.on(“something:do:it”, function(){ 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!”

You can also specify the `triggers` as a function that returns a hash of trigger configurations

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

Triggers work with all View types that extend from the base Marionette.View.

18.5 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.

‘‘‘js Backbone.Marionette.CompositeView.extend({

modelEvents: { “change:name”: “nameChanged” // equivalent to view.bindTo(view.model, “change:name”, view.nameChanged, view) }, collectionEvents: { “add”: “itemAdded” // equivalent to view.bindTo(view.collection, “add”, collection.itemAdded, view) },

// ... event handler methods nameChanged: function(){ /* ... / }, *itemAdded: function(){ / ... */ },

}) ‘‘‘

These will use the memory safe `bindTo`, and will set the context (the value of `this`) in the handler to be the view. Events are bound at the time of instantiation instanciation, and an exception will be thrown if the handlers on the view do not exist.

18.6 View.serializeData

The `serializeData` method will serialize a view’s model or collection - with precedence given to collections. That is, if you have both a collection and a model in a view, calling the `serializeData` method will return the serialized collection.

18.7 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.

18.8 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, it’s 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.

18.8.1 Basic Example

`html <script id="my-template" type="text/html"> I think that <%= showMessage() %> </script>`

‘‘‘js MyView = Backbone.Marionette.ItemView.extend({ template: “#my-template”,

templateHelpers: { showMessage: function(){ return this.name + ” is the coolest!” } }

});

model = new Backbone.Model({name: “Backbone.Marionette”}); view = new MyView({ model: model });

view.render(); //=> “I think that Backbone.Marionette is the coolest!”; ‘‘‘

18.8.2 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.

`js templateHelpers: { something: function(){ return "Do stuff with " + this.name + " because it's awesome."; } }`

18.8.3 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.

`js Backbone.Marionette.ItemView.extend({ templateHelpers: function(){ return { foo: function(){ /* ... */ } } } });`

18.9 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.

`js 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 types.

CHAPTER

NINETEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*