JS

Better Backbone Applications with MarionetteJS

by Joseph Zimmerman

SMASHING MAGAZINE

Imprint

Published 2014 by Smashing Magazine GmbH, Freiburg, Germany
ISBN: 978-3-94454072-6 (PDF)
Cover Design: Veerle Pieters
Proofreader: Owen Gregory
Editing and Quality Control: Vitaly Friedman
eBook Production: Cosima Mielke
Tools: Elja Friedman. Syntax Highlighting: Prism by Lea Verou.

Better Backbone Applications With MarionetteJS was written by Joseph Zimmerman and reviewed by Derick Bailey.

TABLE OF CONTENTS

Introduction

Backbone.js[1] is a growing library that has been around for a few years. It provides some of the key elements necessary for building JavaScript-based web applications in an organized way. Instead of procedural or functional programming, Backbone gives us the object-oriented tools to build our application using patterns similar to MVC.

Though Backbone offers routers, models, and views, it doesn't technically enforce an MVC architecture because these components were designed to work independently: you can pick and choose the components of Backbone you wish to use, and you can even use them in ways that are quite contrary to their intended use. Also, since Backbone doesn't contain anything related to a controller, there is no way it can be considered an MVC framework. Most people refer to it as MV*, since it also doesn't follow any other patterns, such as MVVM or MVP.

Go Learn Backbone First

If you didn't already know the majority of this information, though, you may want to rethink reading this book. There are several great resources out there teaching people how to use Backbone — including a series of articles on my own blog[2] — but this book isn't about Backbone.

After you've obtained a working knowledge of Backbone, then come on back to this book to learn about MarionetteJS[3] (currently at version 1.3.0): an extension to Backbone that can take your Backbone applications to the next level, because that is what this book is about.

What's Marionette?

MarionetteJS — formerly known as Backbone.Marionette — builds on top of Backbone's event-driven architecture, and adds a myriad of tools for building larger, more structured Backbone applications while implementing a lot of functionality that is commonly used across most applications. As its creator, Derick Bailey[4], said himself: Marionette "make[s] your Backbone.js apps dance with a composite application architecture!"

1. http://www.backbonejs.org
2. http://www.joezimjs.com/javascript/give-your-apps-a-backbone-js/
3. http://marionettejs.com/
4. https://twitter.com/derickbailey

Enough with the general mumbo jumbo jargon nonsense! What does Marionette really offer? Well… read the book! Seriously though, if you want a quick look at what Marionette really has to offer before you decide to spend your precious time reading this book, only to find out Marionette doesn't have what you want, then here's a quick overview:

- `ItemView`, `CollectionView` and `CompositeView` give you powerful view classes that can handle most rendering needs. Just provide the collection/model and a template, and you'll have a view that already knows how to render itself. In the case of `CollectionView` and `CompositeView`, you may also want to provide the class of a view that renders the individual items in the collection.
- Applications, modules, regions, layouts and more allow you to organize your application hierarchically and take control of how your views are displayed.
- Event aggregators and request/response systems allow you to decouple your components by having them communicate through a mediator.
- A new base router that makes it easier to move the logic out to the rest of the system and keep your routers clean.

Marionette also follows a similar ideology to Backbone in that it allows you to pick and choose the components you want to use. You are not forced to use anything you don't want to. This is both good and bad. If your application doesn't need to use it, then it's obviously nice to not need to use it, but you can also end up making some bad decisions by excluding some of the components and taking your own path instead. With this book, I hope to teach you to make good decisions about using Marionette to its full potential in your JavaScript applications.

Terminology and Conventions

You will hear me toss the word *class* around a lot in this book. In fact, I've already used the term in this introduction. JavaScript doesn't have classes in the traditional sense. Instead, it uses prototypes, and even though the next version of JavaScript has a `class` structure, it is just syntactical sugar built on top of the current prototypal inheritance system. If you have no idea what I'm talking about, then you should educate yourself on the matter. Mozilla Developer Network has a decent explanation of prototypal inheritance[5] and loads of other JavaScript learning materials.

MarionetteJS is the official name for the JavaScript library that this book is about, but you'll generally see me refer to it as simply *Mari-*

onette. You might see this as laziness on my part — and it partially is — but it's also to make it easier for you to read it. Marionette is difficult enough to read or say without needing to throw another two syllables on the end. Be grateful :) I'm sure Derick Bailey really doesn't mind the casual shortening of the name.

The same goes for Backbone.js. I'll refer to it as just *Backbone* most of the time. I'm saving three characters this time! And I'm saving you two to three syllables depending on whether you pronounce the dot or not. And finally, I do it with Underscore.js, shortening it to plain old *Underscore*. There are potentially more examples of this, but I'm sure you'll spot them and know that I'm just trying to be nice to you.

Getting Started with Marionette

The first thing we need to do is download Marionette. Marionette also has a few dependencies that we'll need to download, including:

- Backbone
- Underscore
- jQuery or Zepto
- Backbone.Wreqr
- Backbone.Babysitter

The first three are pretty obvious, but what are those other two? They are separate Backbone plugins that Derick Bailey built. He felt their usefulness extended beyond just being used in Marionette, so they come separately. You only need to include these last two dependencies if you use the core Marionette file. You can use the full file, which includes Wreqr and Babysitter inline. In any case, you need all of those plus the Marionette library itself.

If you're using Bower[6], you can just type `bower install marionette` or `bower install backbone.marionette` into your console and it'll install Marionette along with all of its dependencies, with the exception of jQuery/Zepto.

If you haven't started using Bower for managing your dependencies, and you don't feel like learning anything new (at least until you're done learning about Marionette), then you can simply go to the Mari-

5. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain
6. http://bower.io/

onetteJS website[7], click the download link, and you'll see links for everything you need.

Setup

Once you've downloaded everything, make sure you import all of the scripts into your HTML in this order:

- jQuery or Zepto
- Underscore
- Backbone
- Wreqr
- Babysitter
- Marionette

The order of jQuery/Zepto and Underscore can be swapped; and similiarly with Wreqr and Babysitter. Once all of these scripts have been loaded in the correct order, you can access Marionette either through the `Backbone.Marionette` or the `Marionette` globals in your own scripts. Throughout this book, we'll just be using `Marionette` because it's shorter. ❧

7. http://marionettejs.com/

Chapter 1: Application

In the introduction, the library's author mentions something about "a composite application architecture". This composite architecture refers to two different things:

- Application components built in a hierarchical module structure.
- Hierarchical view structures through nested (sub)views.

This chapter will discuss the root piece of that first point: the `Application` object. We won't be going into details about how to add subcomponents to your applications until the next chapter, when we talk about modules, but we'll learn just about everything else there is to know about `Application`.

The Central Application Object

Most of the time, when someone creates a Backbone application, they make a central object that everything is attached to, which is often referenced as `App` or `Application`. Backbone doesn't offer anything to make this object from, so *most people just create a main router or view* and make that the app object. While it's great that people are attaching things to a central object so that the global namespace isn't so convoluted, the router was not intended to handle this task and it violates the single responsibility principle.

This can be a problem. Without a central application to use as a namespace (or a module system that keeps variables out of the global scope), most objects and variables will be made globally accessible. While this may seem like it makes things simpler, it poses a substantial risk because variables can easily and unwittingly be overwritten, which can cause bugs that are difficult to track down.

This is why so many people convert their routers and views into central application objects and attach all of their components as properties of this central application. As I mentioned earlier, though, this violates the single responsibility principle. A router is designed to convert URLs to actions, and a view is designed to display information — and that is all they should do. Those are their single responsibilities. Turning them into the central hubs of activity for your application is giving them new responsibilities, which makes them more verbose, makes them more difficult to understand and maintain, and convolutes their roles. You can find more information on this principle around the web[8].

Derick Bailey decided to give users a prebuilt solution to these issues: the `Application` class. `Application` takes over the namespacing responsibility and is a mediator for your entire application. Now your routers can be routers and your views can be views.

Extending Application

As you'll find out quite soon, `Application` has plenty to offer right out of the box, but every application is different, so you'll probably want to add your own functionality to your `Application` object. You have two options to accomplish this:

- Use the built-in `extend` function to create a subclass of `Application`, the same way you do with Backbone's components.
- Add properties to the already instantiated object.

APPLICATION.EXTEND

Just like almost everything in Backbone – and also nearly everything in Marionette – you can call `extend` on the class to create a subclass with your own functionality. `Application` is no different.

```
var MyApp = Marionette.Application.extend({
    // add your own properties and methods here
});
```

Unlike some classes, when you extend `Application`, there are no properties you can add that will be used in a special way (such as the `events` property for `Backbone.View`). This simply allows you to create a new subclass of `Application`.

ADDING PROPERTIES TO THE OBJECT

Just like any other object in JavaScript, you can dynamically add properties to an instantiated `Application` object any time you want. Simply plop a period and a property name onto the object and assign something to it. Obviously, you can use the square bracket notation too, but the point is there's nothing stopping you from adding functionality after your `Application` has already been instantiated.

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

8. https://docs.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/edit

```
// Add property with dot notation
App.foo = function() {...};
// Add property with square bracket notation
App['bar'] = function() {...};
```

Either of these ways of extending `Application` works just fine. While using `extend` seems more correct, the fact that an `Application` generally only has a single instance and is rarely reused makes it seem superfluous. Do whatever suits you.

Initializers

One of the coolest things about Marionette's `Application` is the initializers. When your code is modular, several pieces will need to be initialized when the application starts. Rather than filling a single file with a load of code to initialize all of these objects, you can just set the modules up for initialization within the code for the module. You do this using `addInitializer`. For example:

```
var SomeModule = function(o){
    // Constructor for SomeModule
};

// App is an instantiated Application object
App.addInitializer(function(options) {
    App.someModule = new SomeModule(options);
});
```

All of the initializers added this way will be run when `App.start` is called. Notice the `options` argument being passed into the initializer. This is the very same object that is passed in when you call `App.start(options)`. This is great for allowing a configuration to be passed in so that every module can use it.

A few events are also fired when running through these initializers:

- `initialize:before` fires just before the initializers are run.
- `initialize:after` fires just after the initializers have all finished.
- `start` fires after `initialize:after`.

You can listen for these events and exert even more control. Listen for these events like this:

```
App.on('initialize:before', function(options) {
    options.anotherThing = true; // Add more data to your options
});
App.on('initialize:after', function(options) {
    console.log('Initialization Finished');
});
App.on('start', function(options) {
    Backbone.history.start(); // Great time to do this
});
```

Pretty simple, and it gives you a ton of flexibility in how you start up your applications.

Event Aggregator

The `Application` object brings even more possibilities for decoupling a Backbone application through the use of an event aggregator. A while back I wrote an article about scalable JavaScript applications[9], in which I mentioned that modules of a system should be completely ignorant of one another, and that the only way they should be able to communicate with each other is *through application-wide events* provided by a *mediator*, which in this case is the `Application` object. This way, every module that cares can listen for the changes and events they need to, so that they can react to them without anything else in the system even realizing it exists.

Marionette makes this kind of decoupling largely possible via the event aggregator (provided by the Backbone.Wreqr plugin) that is automatically attached to the application object. While this is only one of the mechanisms I wrote about in that article, it is a start and can be very useful even in small applications.

The event aggregator is available through a property in the application called `vent`. You can subscribe and unsubscribe to events simply via the `on` and `off` methods, respectively (or `bind` and `unbind`, if you prefer). These functions might sound familiar, and that's because the event aggregator is simply an extension of Backbone's `Event` object[10]. Really, the only thing new here that you need to worry about is that we're using the events on an object that should be accessible everywhere within your app, so that every piece of your application can communicate through it. The event aggregator is available as a separate component too, so you can add it to any object you want, just like Backbone's `Event`.

9. http://www.joezimjs.com/javascript/scalable-javascript-applications/
10. http://backbonejs.org/#Events

There are a few other components that Backbone.Wreqr provides that are attached to `Application` objects, but we'll discuss those more in chapter 8. For now, just realize that `Application` can be used as a mediator for communication throughout the system.

Regions

`Region` is another component from Marionette that enables you to easily attach views to different regions of an HTML document. I won't go into detail about how regions work here — they will be discussed in greater detail in chapter 7 — but I'll cover it briefly and explain how they integrate with `Application`.

A region is an object — normally created with `new Marionette.Region({el: 'selector'})` — that manages the insertion and removal of views from a certain location in the DOM. Add a view and automatically render it by using `show` and then close out that view (meaning it will remove it from the DOM and clean up any event bindings) and render a different view simply by calling `show` again, or you can just close the view by calling `close`. Regions can do more than that, but the fact that they handle the rendering and closing for you with a single function call makes them extremely useful. Here's a code sample for those who speak better in code than in English:

```
// Create a region. It will control what's in the #container
// element.
var region = new Marionette.Region({
    el: "#container"
});

// Add a view to the region. Immediately renders the view.
region.show(new MyView());

// Close out the current view and render a different view.
region.show(new MyOtherView());

// Close out the view and display nothing in #container.
region.close();
```

If you want a `Region` directly on your application object (e.g. `App.someRegion`), `Application` provides a simple way to add one quickly: `addRegions`. There are three ways to use `addRegions`. In every case, you would pass in an object whose property names will be added to the application as regions, but the value of each of these may be different depending on which way you wish to accomplish this.

SELECTOR

Simply supply a selector, and a standard region will be created that manages views being attached to the DOM node that matches that selector.

```
App.addRegions({
    container: "#container",
    footer:    "#footer"
});

// This is equivalent to
App.container = new Marionette.Region({el:"#container"});
App.footer    = new Marionette.Region({el:"#footer"});
```

CUSTOM REGION TYPE

You can extend `Region` to create your own types of regions. If you want to use your own type of region, you can use the syntax below. Note that, with this syntax, `el` must already be defined within your region type, which is a selector used to determine which DOM node the region manages.

```
var ContainerRegion = Marionette.Region.extend({
    el: "#container", // Must be defined for this syntax
    // Whatever other custom stuff you want
});

var FooterRegion = Marionette.Region.extend({
    el: "#footer", // Must be defined for this syntax
    // Whatever other custom stuff you want
});

// Use these new Region types on App.
App.addRegions({
    container: ContainerRegion,
    footer:    FooterRegion
});

// This is equivalent to:
App.container = new ContainerRegion();
App.footer    = new FooterRegion();
```

CUSTOM REGION TYPE WITH SELECTOR

If you don't define `el` — or you want to override it — in your custom region type, then you can use this syntax, which specifies both the region type and the selector:

```
var ContainerRegion = Marionette.Region.extend({});

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

// Use these new Region types on App.
App.addRegions({
    container: {
        regionType: ContainerRegion,
        selector:   "#container"
    },
    footer: {
        regionType: FooterRegion,
        selector:   "#footer"
    }
});

// This is equivalent to:
App.container = new ContainerRegion({el:"#container"});
App.footer    = new FooterRegion({el:"#footer"});
```

As you can see, adding application-wide regions is dead simple (especially if you're using the normal `Region` type). You'll get a better sense of how useful this is when we discuss regions in more detail and put them to work in an application.

Summary

We've learned about the `Application` object, which is used as the root of an application and a mediator between the other modules of the system. These are powerful features that offer a simple means for organizing and decoupling your JavaScript applications.

Marionette also adds many other great features to make Backbone development simpler. So far we've only covered one of the many components Marionette offers — though we have quickly touched on a couple other components used by `Application` — and there is a lot more to learn, so keep reading! ❧

Chapter 2: Modules

At the beginning of chapter 1, I mentioned the two meanings that composite architecture had in the context of Marionette. The first meaning referred to application components built in a hierarchical module structure. Now that we've covered the root node of that hierarchy, let's look at how we start to branch out into a hierarchy of modules.

What Are Modules?

Before we get into the details of how to use Marionette's module system, we should make sure we have a decent definition of what a module is. A module is *an independent unit of code that ideally does one thing*. A module is used in conjunction with other modules to create an entire system. The more independent a unit of code is, the more easily it can be exchanged or internally modified without affecting other parts of the system, and the more easily it can be tested.

For this book, that's about as far as we need to define modules, but if you want to learn more about writing modular code and its benefits, plenty of resources exist throughout the internet, of which the "Modularity"[11] chapter of the MSDN book, *Project Silk: Client-Side Web Development for Modern Browsers*, is one of the better resources out there.

The JavaScript language doesn't currently have any built-in means for defining or loading modules (the next version should change that[12]), but several libraries have arisen to provide support for defining and loading modules. Marionette's module system is quite different from all of these in a few ways:

1. It doesn't provide a means of loading modules from other script files. A module must already be loaded and defined before you can use it.
2. Modules are not available completely independently. They are attached to the `Application` object as namespaced objects.
3. Marionette modules have built-in functionality. Most module systems have you define the entire module, but Marionette's modules are instantiated `Marionette.Module` objects.

11. http://msdn.microsoft.com/en-us/library/hh404079.aspx
12. http://wiki.ecmascript.org/doku.php?id=harmony:modules

I won't elaborate on the first point, but the details of the other two will become clear as you read more. First, let's learn how to define a module, and we'll move on from there.

Module Definition

Let's start with the most basic module definition. As mentioned, modules are accessible through the `Application`, so first we need one of those. Then we can use its `module` method to define a module.

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

var myModule = App.module('myModule');
```

That's pretty simple, right? Well, it is, but that's the simplest module we can create. What exactly did we create, though? Essentially, we told the application that we want a barebones module, with no custom functionality, and that it will be named `myModule` (according to the argument we passed to the `module` function). But what is a barebones module? In short, it's an instantiation of the `Marionette.Module` class.

`Module` comes with a bit of functionality baked in, such as: events (through `EventAggregator`, which we'll discuss thoroughly in chapter 8); starting; *initializers* (just like `Application` has); stopping; and *finalizers* (we'll go over these in the "Starting and Stopping Modules" section later in this chapter).

STANDARD MODULE DEFINITION

Now let's look at how to define a module with some of our own functionality.

```
App.module("myModule", function(myModule, App, Backbone,
Marionette, $, _){
    // Private Data And Functions
    var privateData = "this is private data";

    var privateFunction = function(){
        console.log(privateData);
    };

    // Public Data And Functions
    myModule.someData = "public data";

    myModule.someFunction = function(){
        privateFunction();
        console.log(myModule.someData);
    };
});
```

As you can see, there's a lot of stuff in there. Let's look at the top line and work our way down. Just like before, we call `App.module` and provide a name for the module. But now we're also passing in a function, too. The function is passed several arguments. I bet you can figure out what they are, based on the names I've given them, but I'll still explain them all:

- `myModule` is the very module you're trying to create. Remember, it's already created for you, and it's a new instantiation of `Module`. You're probably going to want to extend this with some new properties or methods; otherwise, you might as well stick with the short syntax that doesn't require you to pass in a function.
- `App` is the `Application` object that you called `module` on.
- `Backbone` is, of course, the reference to the Backbone library.
- `Marionette` is the reference to the Backbone.Marionette library. It is actually available through `Backbone`, but this allows you to alias it and make it a shorter name.
- `$` is your DOM library, which will be either jQuery or Zepto (or possibly something else in the future).
- `_` is a reference to Underscore or Lodash, whichever you're using.

After that, you can actually pass in and use custom arguments. We'll go over this in a moment.

Normally, I would say that most of these arguments are unnecessary; after all, why wouldn't you already have access to these references? However, I can see these being useful in a couple of situations:

- A minifier can shorten the names of the arguments, saving some bytes.
- If you're using RequireJS or some other module loader, you only need to pull in the `Application` object as a dependency. The rest will be available through the arguments given to you by `Module`.

Anyway, let's get back to explaining the rest of what's going on in the code above. Inside the function, you can use the closure to create private variables and functions, which is what we've done. You can also expose data and functions publicly by adding them as properties of `my-`

`Module`. This is how we create and extend our module. There is no need to return anything because the module will be accessible directly through `App`, as I'll explain in the "Accessing a Module" section below.

Note: Make sure that you only add properties to your `module` variable and do not set it equal to something (for example, `myModule = {…}`), because when you set your `module` variable to something, that changes what the variable's name references, and none of the changes you specify will show up in your module later.

Overall, we're essentially just using a mechanism similar to the *module pattern*[13], except first, the anonymous function is being sent as a parameter to a different function instead of just being called immediately; and second, the module we're defining is already instantiated and is passed into our function instead of being returned from the function and assigned to an outside variable.

Getting Help From Underscore

Since the module we're trying to define is already an instantiated object, we obviously can't just assign an object literal to the module — as I've already mentioned — but there is an alternative to messes such as the one below:

```
App.module("myModule", function(myModule, App, Backbone,
Marionette, $, _){
    myModule.data1 = "public data";
    myModule.data2 = "some data";
    myModule.method1 = function() {
        // Do Stuff
    };
    myModule.method2 = function(){
        // Do Stuff
    };
});
```

That just requires writing `myModule` too many times. Instead, we can use Underscore's `extend` method to simplify things for us. The great thing is that since Underscore is already a required dependency for Backbone, it's not something extra we need to add to the project.

Below you'll find the same code as above, except it uses Underscore to make it easier to read and faster to type:

13. http://www.joezimjs.com/javascript/javascript-closures-and-the-module-pattern/

```
App.module("myModule", function(myModule, App, Backbone,
Marionette, $, _){
    _.extend(myModule, {
        data1: "public data",
        data2: "some data",
        method1: function() {
            // Do Stuff
        },
        method2: function(){
            // Do Stuff
        }
    });
});
```

You aren't required to use Underscore to do it this way, but it's a simple and useful technique, so feel free to have at it.

Custom Arguments

Earlier, I noted that you can send in custom arguments. In fact, you can send in as many custom arguments as you want. Take a look at the code below to see how it's done.

```
App.module("myModule", function(myModule, App, Backbone,
Marionette, $, _, customArg1, customArg2){
    // Create Your Module
}, customArg1, customArg2);
```

As you can see, if you pass additional arguments to `module`, they will be passed in to the function that you are defining your module in. As with the other arguments passed into your definition function, the biggest benefit I see from this is *saving some bytes after minification*.

Another potential benefit can come from isolating global variable changes. If the custom arguments that you pass in are primitive values (not objects), then when the variable is changed either outside or inside the definition function, the *changes don't affect the value in the other scope*. I don't see this issue coming up often, but it's an option if you need it.

Overall, though, these spare arguments aren't all that useful and you'll be hard pressed to find real use cases for them. Even Derick Bailey doesn't recommend using them.

This is the Module

Another thing to note is that the `this` keyword is available within the function and actually refers to the module. This means you don't even need the first `myModule` argument, but you would lose the advantage of minification. Let's rewrite that first code using `this` so that you can see that it's exactly the same as `myModule`.

```
App.module("myModule", function(){
    // Private Data And Functions
    var privateData = "this is private data";

    var privateFunction = function(){
        console.log(privateData);
    }

    // Public Data And Functions
    this.someData = "public data";

    this.someFunction = function(){
        privateFunction();
        console.log(this.someData);
    }
});
```

As you can see, because I'm not using any of the arguments, I decided not to list any of them this time. It should also be obvious that you can skip the first argument and just use `this`. This is entirely a stylistic choice and you can decide for yourself exactly what you would like to do. If someone reading your code understands Marionette, then it should be just as readable either way.

SPLIT DEFINITIONS

The final thing I'll mention about defining modules is that we can split up the definitions. This can be useful for others to extend your module, or it can also help keep files sizes smaller by splitting the definitions up. If you're using a module loader – like RequireJS – then this can be done in a such way that split definitions are completely pointless, but without the module loader it's probably the best way to keep things organized. Here's an example of split definitions:

```
// File 1
App.module("myModule", function(){
    this.someData = "public data";
});

// File 2
App.module("myModule", function(){
    // Private Data And Functions
    var privateData = "this is private data";

    var privateFunction = function(){
        console.log(privateData);
    }

    this.someFunction = function(){
        privateFunction();
        console.log(this.someData);
    }
});
```

This gives us the same result as the previous example, but it's split up. This works because in `File 2`, the module that we defined in `File 1` is being given to us (assuming that `File 1` was run before `File 2`).

Of course, if you're trying to access a private variable or function, it has to be defined in the module definition where it is used, because it's only available within the scope where it is defined.

As you'll find out in the next section, you access modules the exact same way you define them, and, in fact, if the module is not defined at the point it is being accessed, a plain module will be created for the definition.

```
// I want to use the module, but it isn't defined
var mod = App.module('myModule');

// `mod` is now a barebones module, so we can call built-in
// functions
// you'll see what these are later in the chapter
mod.addInitializer(function(){...});

// Later we have the module definition and everything still works
App.module("myModule", function(){
    this.someData = "public data";

    var privateData = "this is private data";

    var privateFunction = function(){
        console.log(privateData);
    }

    this.someFunction = function(){
        privateFunction();
        console.log(this.someData);
    }
});
```

Since we're only extending the module given to us, people can use the module before we even define our additions to it. They'll run into errors if they try to access any of our functionality before it's defined, but when will that ever happen?

Accessing A Module

What good is creating modules if we can't access them? We need to be able to access them in order to use them. Well, in the very first code snippet of this article, you saw that when I called `module`, I assigned its return value to a variable. That's because we use the very same method to both define *and* retrieve modules.

```
var myModule = App.module("myModule");
```

Normally, if you're just trying to retrieve the module, you'll pass in the first argument, and `module` will go out and grab the module with that name for you. But if you pass in a function as the second argument, the module will be augmented with your new functionality, *and* it will still return your newly created or modified module. This means you can define your module and retrieve it all with a single method call.

This isn't the only way to retrieve modules, though. When a module is created, it is attached directly to the `Application` object that it was constructed with. This means you can also use the normal dot notation to access your module; but this time, it *must* be defined beforehand, otherwise you'll get `undefined` back.

```
// Works but I don't recommend it
var myModule = App.myModule;
```

While this syntax is shorter, it doesn't convey the same meaning to other developers. I would recommend using `module` to access your modules so that it is obvious you are accessing a module and not some other property of `App`. The convenience and very slight danger here is that it will create the module if it doesn't already exist. The danger comes if you misspell the name of the module: you won't have any way of knowing that you didn't get the correct module until you try to access a property on it that doesn't exist.

Submodules

Modules can also have submodules. Sadly, `Module` doesn't have its own `module` method, so you can't add submodules to it directly, but that won't stop us. Instead, to create submodules you call `module` on `App`, just like you used to; but for the name of the module, you need to put a dot (`.`) after the parent module's name and then put the name of the submodule.

```
App.module('myModule.newModule', function(){
    ...
});
```

By using the dot separator in the module's name, Marionette knows that it should create (or retrieve) a module as a submodule of the module named before the dot. The cool (and potentially dangerous) part is that if the parent module isn't created at the time that you call this, it will be created along with its submodule. This can be dangerous because of the same potential for misspelling that I just mentioned. You could end up creating a module you didn't mean to, and the submodule would be attached to it, instead of the module you intended.

ACCESSING SUBMODULES

As before, submodules can be accessed the same way they are defined, or you can access them as properties of the module.

```
// These all work. The first example is recommended
var newModule = App.module('myModule.newModule');
var newModule = App.module('myModule').newModule;
var newModule = App.myModule.newModule;

// These don't work. Modules don't have a 'module' function
var newModule = App.myModule.module('newModule');
var newModule = App.module('myModule').module('newModule');
```

Any of these methods of accessing the submodule will work equally well if both the module and submodule have already been created. As noted in the code's comment, though, the first method is recommended

Starting And Stopping Modules

If you're not jumping around between this book's chapters randomly, you will know that you can start an `Application` with its `start` method. Starting modules works in the same way, and they can also be stopped with the `stop` method.

If you recall, you can add initializers with `addInitializer` to an `Application`, and they will run when it is started (or will run immediately if the `Application` has already started). A few other things happen when you start an `Application`. Here are all of the events, in order:

- fires the `initialize:before` event,
- starts all of the defined modules,
- runs all of the initializers in the order they were added,
- fires the `initialize:after` event,
- fires the `start` event.

A `Module` behaves in a very similar way. The number of events and some of the names of the events are different, but overall it is the same process. When a module is started, it:

- fires the `before:start` event,
- starts all of its defined submodules,
- runs all of its initializers in the order they were added,
- fires the `start` event.

The `stop` method is also very similar. Instead of adding initializers, though, you need to add finalizers. You do this with `addFinalizer` and by passing in a function to run when `stop` is called. Unlike with initializers, no data or options are passed along to each of the functions. When `stop` is called, it:

- fires the `before:stop` event,
- stops its submodules,
- runs its finalizers in the order they were added,
- fires the `stop` event.

Initializers and finalizers are intended to be used within the module definitions for setup and tear-down, rather than by external components.

It can be helpful to use them outside, but generally, listening for the events mentioned should be done instead.

When using initializers and finalizers inside the module definitions, you can define a module without actually instantiating any objects, but then write your initializers to start instantiating the objects and setting them up, such as in the example below.

```
App.module("myModule", function(myModule){
    myModule.startWithParent = false;

    var UsefulClass = function() {...}; // Constructor definition
    UsefulClass.prototype ... // Finish defining UsefulClass
    ...

    myModule.addInitializer(function() {
        myModule.useful = new UsefulClass();
        // More setup
    });

    myModule.addFinalizer(function() {
        myModule.useful = null;
        // More tear down
    });
});
```

Normally, `UsefulClass` should be defined outside of the module definition (and probably loaded in via a module loader), but the idea is the same. Do instantiations and setup in the initializers and clear out the now useless objects in the finalizers to free up memory.

AUTOMATIC AND MANUAL STARTING

When a module is defined, by default it will automatically start at the same time that its parent starts (either the root `Application` object or a parent module). If a module is defined on a parent that has already started, it will start immediately.

You can tell a module not to start automatically by changing its definition in one of two ways: inside the definition, you can set a module's `startWithParent` property to `false`; or you can pass an object (instead of a function) to `module` that has a `startWithParent` property set to `false` and a `define` property to replace the usual function.

```
// Set 'startWithParent' inside function
App.module("myModule", function(){
    // Assign 'startWithParent' to false
    this.startWithParent = false;
});

// -- or --

// Pass in object
App.module("myModule", {
    startWithParent: false,

    define: function(){
        // Define module here
    }
});

App.start();

// myModule wasn't started, so we need to do it manually
App.module('myModule').start("Data that will be passed along");
```

Now the module won't automatically start with its parent. You must call `start` manually to start it, as I did in the example above. The data that is passed to `start` could be anything of any type, and it will be passed along to the submodules when they're started, to the initializers, and to the `before:start` and `start` events.

As I mentioned earlier, though, data isn't passed along like this when you call `stop`. Also, `stop` *must* be called manually on a module (since `Application` doesn't have a `stop` method, and it will always call `stop` on submodules – there is no way around this. This makes sense because a submodule shouldn't be running when its parent isn't running, although there are cases when a submodule should be off when its parent is running.

Other Events And Built-In Functionality

I mentioned that `Module` comes with some baked-in functionality, such as the `EventAggregator`. As discussed, we can use the `on` method on a module to watch for events related to starting and stopping. That's not all. There are no other built-in events, but a module can define and trigger its own events. Take a look:

```
App.module('myModule', function(myModule) {
    myModule.doSomething = function() {
        // Do some stuff
        myModule.trigger('something happened', randomData);
    }
});
```

Now, whenever we call `doSomething` on the module, it will trigger the `something happened` event, which you can subscribe to:

```
App.module('myModule').on('something happened', function(data) {
    // Whatever arguments were passed to `trigger` after the name
    // of the event will show up as arguments to this function

    // Respond to the event
});
```

This is very similar to the way we do things with events on collections, models, and views in normal Backbone code.

How Should We Use Modules?

The modules in Marionette can definitely be used to define modules very similarly to any other module definition library, but that's actually not how it was intended to be used. The built-in `start` and `stop` methods are an indication of this. The modules that Marionette includes are for creating and managing relatively *large* subapplications of the main larger application. As an example, let's look at Gmail.

Gmail is a single application that actually contains several smaller applications: email client, chat client, phone client, and contact manager. Each of these is independent – it can exist on its own – but they are all found within the same application and are able to interact with one another. When we first start up Gmail, the contact manager isn't up, and neither is the chat window. If we were to represent this with a Marionette application, each of those subapplications would be a module. When a user clicks the button to open the contact manager, we would stop the email application (because it becomes hidden – although, for speed, we could keep it running and just make sure it doesn't show in the DOM), and start the contacts manager.

Another example would be an application built largely out of widgets. Each widget would be a module that you can start and stop in order to show or hide it. This would be like a customizable dashboard such as iGoogle or the dashboard in the back-end of WordPress.

Of course, we're not limited to using Marionette's modules in this way, but it's tedious to use them in the traditional sense. This is because Marionette's modules are fully instantiated objects with built-in functionality, while traditional modules are generally classes that are meant for instantiation later.

Marionette's modules have caused a lot of confusion, since they lack much resemblance with anything else related to modules (with the exception of the module pattern, though there are still many differences). Personally, I think the name "SubApplication" would be more semantic.

No matter what the name is, though, Derick Bailey is looking to revamp them in version 2.0 (whenever he gets there). He doesn't like that these modules are responsible for namespacing, encapsulation, and starting and stopping subapplications. We both agree that it is unfortunate that these three concerns are crammed together in this way, especially considering how simple it is to do namespacing and encapsulation by yourself.

Before 2.0 is released, this chapter is perfectly usable and can be used with abandon, but beware the usefulness of this chapter once Mr. Bailey gets his hands dirty and resolves these issues.

Summary

Despite the convoluted mess that Marionette's modules bring to the table, they still offer us a great way of easily creating powerful subapplications that can be started and stopped. This is a great way to organize and architect your large applications – splitting them into smaller, more manageable applications.

They also provide a built-in event system to continually help decouple the components of your applications. Overall, if your applications are growing large, modules will likely be helpful to you. ❧

Chapter 3: Views

So we've gotten the big guys — `Application` and `Module` — taken care of, so now we'll get into the bits you can see: views. Backbone already has views, but they really don't do very much for you. Marionette fills in the feature gaps, so you can skip over the vast amounts of boilerplate code, and avoid the pitfalls you could run into if you don't know what to look out for. So let's take a gander at what Marionette provides.

Event Binding

Up until recently, Backbone views were often mishandled, causing a horrible problem known as zombie views. The issue was caused by the views listening to events on the model, which in itself is completely harmless. The problem was that when the views were no longer needed and were discarded, they didn't stop listening to the events on the model, which meant that the model still had a reference to the view, keeping it from being garbage-collected. This caused the amount of memory used by the application to constantly grow, and the view would still be responding to events from the model, even though it wouldn't render anything because it was removed from the DOM.

Many Backbone extensions and plugins — including Marionette — remedied this early on. I won't go into any detail on that, though, because Backbone's developers fixed this problem themselves (finally!) when they released Backbone 1.0, by adding the `listenTo` and `stopListening` methods to `Events`, which Backbone's `View` class inherits some methods from. Marionette's developers have since removed their own implementation of this feature, but that doesn't mean Marionette doesn't help us out with some other things related to event binding.

To make binding to events on the view's models and collections simpler, Marionette's base `View` class gives us a couple properties to use when extending Marionette's views: `modelEvents` and `collectionEvents`. Simply pass in an object hash where the keys are the name of the event we're listening to on the model or collection, and the property is the name(s) of the function(s) to call when that event is triggered. Look at this simple example:

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    modelEvents: {
        'change:attribute': 'attributeChanged render', // call
        // 2 functions
        'destroy': 'modelDestroyed'
    },

    render: function(){ … },
    attributeChanged: function(){ … },
    modelDestroyed: function(){ … }
});
```

This accomplishes the same thing as using `listenTo`, except it requires less code. Here's the equivalent code using `listenTo`.

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    initialize: function() {
        this.listenTo(this.model, 'change:attribute',
        this.attributeChanged);
        this.listenTo(this.model, 'change:attribute',
        this.render);
        this.listenTo(this.model, 'destroy', this.modelDestroyed);
    },

    render: function(){ … },
    attributeChanged: function(){ … },
    modelDestroyed: function(){ … }
});
```

There are a couple key things to note. First, `modelEvents` is used to listen to the view's model, and `collectionEvents` is used to listen to the view's collection (`this.model` and `this.collection`, respectively). Second, you may have noticed that there are two callbacks for the `change:attribute` event. When you specify a string for the callbacks, you can have as many callback function names as you want, separated by spaces. All of these functions will be invoked when the event is triggered. Any function name that you specify in the string must be a method of the view.

There are alternative ways to specify `modelEvents` and `collectionEvents`, too. First, instead of using a string to specify the names of methods on the view, you can assign anonymous functions:

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    modelEvents: {
        'change': function() {
            …
        }
    }
});
```

This isn't a good way of doing this, but the option is there if you absolutely need it. Also, instead of simply assigning an object literal to `modelEvents` or `collectionEvents`, you may also assign a function. The function will need to return an object hash that has the events and callbacks.

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    modelEvents: function() {
        return {'destroy': 'modelDestroyed'};
    },

    modelDestroyed: function(){ … }
});
```

There are very few reasons that you would need to use `modelEvents` like this, but it will sure come in handy when that occasion arises.

As often as possible, the `modelEvents` and `collectionEvents` feature follows the pattern that Backbone and Marionette use: relegate code to simple configuration. Backbone itself did this with the `events` hash, which enables you to easily set up DOM event listeners. Marionette's `modelEvents` and `collectionEvents` are directly inspired by the original `events` configuration in Backbone. You'll see this configuration concept show up a lot, especially in a couple chapters, when we get into `ItemView`, `CollectionView` and `CompositeView`.

Destroying A View

As I mentioned at the beginning of the previous section, sometimes a view needs to be discarded or removed because a model was destroyed, or because we need to show a different view in its place. With `stopListening`, we have the power to clean up all of those event bindings — assuming they were set up using `listenTo`. But what about destroying the rest of the view? Backbone has a `remove` function that calls `stopListening` for us and also removes the view from the DOM.

Generally, this would be all you need, but Marionette takes it a step further by adding the `close` function. When using Marionette's views, you'll want to call `close` instead of `remove` because it will clean up all of the things that Marionette's views set up in the background, in addition to calling Backbone's `remove` function for you.

Another benefit offered by Marionette's `close` method is that it fires off some events. At the start of closing the view, it'll fire off the `before:close` event, and then the `close` event when it's finished. In addition to the events, you can specify methods on the view that will run just before these events are triggered.

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    onBeforeClose: function() {
        // This will run just before the before:close event is
        // fired
    },

    onClose: function(){
        // This will run just before the close event is fired
    }
});
```

If you want to run some code before the view disappears completely, you can use the `onBeforeClose` and `onClose` view methods to automatically have it run without needing to set up listeners for these events. Simply declare the methods, and Marionette will make sure they are invoked. Of course, other objects will still need to listen to the events on the view.

DOM Refresh

Back when we discussed `Application`, I mentioned `Region` a bit. I won't get into this much here (they will be covered in detail in chapter 7), but know that a `Region` is an object that handles the showing and hiding or discarding of views in a particular part of the DOM. Look at the code below to see how to render a view in a `Region`.

```
var view = new FooView(); // Assume FooView has already been
// defined
region.show(view); // Assume the region was already instantiated.
// Just use "show" to render the view.
```

When you use `show`, it will render the view, attach it to the DOM and then show the view, which simply means that a `show` event is fired so that components will know that the view was rendered via a `Region`. (All view classes that Marionette implements that are based on this base `View` class will also call the `onRender` function if you've defined it and will fire a `render` event when `render` is invoked.) After a view has been rendered and then shown, if the view is rendered again, it will trigger a DOM refresh.

This actually isn't true at the moment because of a bug, but it's on the developer's to-do list, and I'll work with him to fix it if we can. Currently, when a view is rendered, it will set a flag saying that it was rendered. Then, when the view is shown, it will set a flag saying that it was shown. The moment when both of these flags have been activated, it will trigger a DOM refresh. Then, any time after that, the DOM refresh will be triggered when the view is rendered or shown. Keep this in mind if you need to use this functionality. This will be essentially the same in most circumstances, but it may cause the DOM refresh event to fire twice if a view is shown again (perhaps in a different region).

When a DOM refresh is triggered, first it will run the `onDomRefresh` method of the view (if you defined one), and then trigger the `dom:refresh` event on the view. This is mostly useful for UI plugins (such as jQuery UI, Kendo UI, etc.) because some widgets depend on the DOM element they are working with to already exist in the actual DOM. Often, when a view is rendered, it won't be appended to the DOM until after the rendering has finished. This means that you can't initialize the widget during `render` or in your `onRender` function.

However, you can use it in `onShow` (which is invoked just before the `show` event is triggered) because a region is supposed to be attached to an existing DOM node (as we'll see in chapter 7). Now, since the view has been shown, you will know that the view is in the DOM; so, every time `render` is called, a DOM refresh will take place immediately after the rendering, and you can call the UI plugin's functionality safely again.

DOM Triggers

Sometimes, when a user clicks a button, you want to respond to the event, but you don't want the view to handle the work. Instead, you want the view to trigger an event so that other modules listening for this event can respond to it. Suppose you have code that looks like this:

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    events: {
        'click .awesomeButton': 'buttonClicked'
    },
    buttonClicked: function() {
        this.trigger('awesomeButton:clicked', this);
    }
});
```

The function for handling the click event just triggers an event on the view. Marionette has a feature that allows you to specify a hash of these events to simplify this code. By specifying the `triggers` property when extending a `View`, you can assign a hash very similar to the `events` property. But instead of giving it the name of one of the view's methods to invoke, you give it the name of an event to fire. So, we can convert the previous snippet to this:

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    triggers: {
        'click .awesomeButton': ' awesomeButton:clicked '
    }
});
```

And it'll do nearly the same thing. There is one major difference between these two snippets: the arguments passed to the listening functions. In the first snippet, all we passed to the functions listening for the event was `this`, which was the view. Using `triggers`, Marionette will pass a single object with three properties as the argument to each of the functions. These three properties are the following:

- `view`: a reference to the view object that triggered the event.
- `model`: a reference to the view's `model` property, if it has one.
- `collection`: a reference to the view's `collection` property, if it has one.

So, if you were subscribing to the event from the previous snippet, it would look like this:

```
// 'view' refers to an instance of the previously defined View
// type
view.on('awesomeButton:clicked', function(arg) {
    arg.view; // The view instance
    arg.model; // The view's model
    arg.collection; // The view's collection
}
```

As with many of the Marionette features, there isn't a surplus of use cases for this, but in the few situations where this applies, it makes things much simpler.

DOM Element Caching

Often, `this.$el` isn't the only element that you'll need to directly manipulate. In such cases, many people will do something like this:

```
Backbone.Marionette.View.extend({ // We don't normally directly
// extend this view
    render: function() {
        this.list = this.$('ul');
        this.listItems = this.$('li');
        . . .
        // Now we use them and use them in other methods, too.
    }
});
```

Once again, Marionette makes this simpler by converting this all into a simple configuration. Just specify a `ui` property that contains a hash of names and their corresponding selectors:

```
Backbone.Marionette.View.extend({ // We don't normally directly
// extend this view
    ui: {
        list: 'ul',
        listItems: 'li'
    }
});
```

You can access these elements with `this.ui.x`, where `x` is the name specified in the hash, such as `this.ui.list`. This `ui` property is converted into the cached jQuery objects by the `bindUIElements` method. If you're extending `Marionette.View`, instead of one of the other view types that Marionette offers, then you'll need to call this method yourself; otherwise, the other view types will call it for you automatically.

```
Marionette.View.extend({ // We don't normally directly extend
// this view
    ui: {
        list: 'ul',
        listItems: 'li'
    },
    render: function() {
        // render template or generate your HTML, then…
        this.bindUIElements();
        // now you can manipulate the elements
        this.ui.list.hide();
        this.ui.listItems.addClass('someCoolClass');
    }
});
```

Summary

We've already seen many features that Marionette brings to views that cut down on the complexity and amount of code required for common tasks—but we haven't even touched on the most important part. `Marionette.View` doesn't handle any of the rendering responsibilities for us, but Marionette has three other view types that do: `ItemView`, `CollectionView`, and `CompositeView`. These view types, which are what you'll actually extend in your code (note the "We don't normally directly extend this view" comment in all of the code snippets), will take a few minor configuration details and then handle the rest of the rendering for you. The next chapter is about `ItemView` and then the next chapters will cover `CollectionView` and `CompositeView`. ❧

Chapter 4: ItemView

The previous chapter on Marionette's `View` was only about the base class that each of Marionette's usable view classes inherit from. If you're jumping around between chapters, make sure you read chapter 3 before this chapter, because now we are getting into the meat and potatoes of what Marionette's views have to offer, and the previous chapter will help you understand this one, along with telling you more about the features available for your views.

In this chapter, we'll specifically discuss the `ItemView`. The `ItemView` is designed to show data from a single model, or just a template without any data. The next chapter is about `CollectionView` and `CompositeView`, which – each in their own ways – provide you the means for displaying a collection of models. `ItemView` can also display a collection, but your template will need to iterate over it, whereas the other view types will iterate over the models and render subviews for each one.

Extending An ItemView

The first thing we need to do to use the awesome power of the `ItemView` is create our own view class that inherits from it. Just like practically everything in Backbone and Marionette, the `ItemView` is easily extendable using `Marionette.ItemView.extend`. This works in exactly the same manner as using `Backbone.View.extend` except, of course, `ItemView` provides you with a lot more built-in functionality.

```
var MyView = Marionette.ItemView.extend({
    // Your functionality
});
```

Congratulations! You have now extended `ItemView`! But wait… if you try to instantiate it, you get an error! This is because `ItemView` expects some things to be specified so it knows how to render the view. Let's talk about what's required to get the `ItemView` to really work.

Rendering

Part of the point of Marionette creating `ItemView`, and its other view types, is to remove the boilerplate related to rendering a view. The vast majority of views do the exact same thing: compile a template; run the templating function with the data (usually from a model or collection); and render it.

Marionette realizes that this is how things work, so it strives to help you out by eliminating that repetitive code. When using an `ItemView`, all you need to provide is a template, and if your template is using dynamic data, then make sure a model or collection is provided when instantiating the view.

```
var MyView = Marionette.ItemView.extend({
    template: "#my-template"
});
```

If the view just displays content and doesn't listen for events of any kind, then this is all that you'll need; but no matter what, the template is required for an `ItemView` to work. There are two ways you can specify a template.

1. INLINE TEMPLATES

A CSS selector that corresponds to an inline element that contains the template. For the above example, you could write something like this:

```
<script type="template" id="my-template">
<p>This is an Underscore.js template with <%= data ></p>
</script>
```

By default, Marionette will compile the template using Underscore's templating mechanism[14]. If you prefer to use inline templates and like this option of specifying the selector to find the template content, but you don't want to use Underscore, you can override the `Marionette.TemplateCache.prototype.compileTemplate` function. For example, if you'd prefer to use Handlebars[15], you can write the new `compileTemplate` function like this:

```
Marionette.TemplateCache.prototype.compileTemplate =
function(rawTemplate) {
    return Handlebars.compile(rawTemplate);
}
```

14. http://underscorejs.org/#template
15. http://handlebarsjs.com/

2. PRECOMPILED TEMPLATES

You can also specify a function. This function should be the compiled template that you receive when running `_.template(rawTemplate)` or `Handlebars.compile(rawTemplate)`, or something similar from whatever templating library you are using. In most of my projects I use external template files and RequireJS's tpl plugin[16]. This plugin will pull in the external template and compile it using Underscore's mechanism (although it doesn't require Underscore to be loaded because it hard-coded the implementation into the plugin) and give back the result of the compilation. If you're a fan of Handlebars, you may want the RequireJS plugin for Handlebars[17] instead.

If you're not using RequireJS with one of those plugins and you're using Underscore, you specify a template as a function like this:

```
var MyView = Marionette.ItemView.extend({
    template: _.template('<div>Your Template Here</div>');
    /* or */
    template: _.template($('#template').html());
});
```

There are several ways to put that string into the `template` function, but it's all essentially the same. As long as you give it a precompiled template function, Marionette will take care of the rest.

Provided that one of these things is supplied, you should have a working view that will display the template whenever you call `render` on it.

Rendering With Data

Obviously, it wouldn't be much of an application if your views all used static templates. We want our templates to render data from our models and collections, right? Well, defining a view would look exactly the same, but when you create the view, you need to pass a model in:

```
var MyView = Marionette.ItemView.extend({
    template: "#my-template"
});

var model = new Backbone.Model({name: "Joe"});
```

16. https://github.com/ZeeAgency/requirejs-tpl
17. https://github.com/SlexAxton/require-handlebars-plugin

```
var view = new MyView({model:model});
```

There's no difference between the definition of a view class that renders a static template and one that renders a template with dynamic data. The difference is entirely in the template. Your view just tells the template to render, and if there is any data attached to `this.model` or `this.collection`, it'll be passed into the template function.

`ItemView` passes the data to the template differently for collections and models. To retrieve the data to pass to the template, `ItemView` uses its `serializeData` function, which looks like this:

```
serializeData: function(){
    var data = {};

    if (this.model) {
        data = this.model.toJSON();
    }
    else if (this.collection) {
        data = { items: this.collection.toJSON() };
    }

    return data;
}
```

If your view has a model, the function will serialize it with `toJSON()` and use that. If there's no model, the function will check to see if it has a collection. If it has a collection, the function will use an object with one property, called `items`, that contains an array of serialized models (retrieved from the collection's `toJSON` function). If neither model nor collection is provided, the function will just use an empty object literal.

This function can easily be overridden if you'd prefer to send different data to the template, or if your view has both a model and a collection but you'd rather have the collection sent to the template instead of the model (since the model is prioritized over the collection in the default implementation).

Template Helpers

If you've used Handlebars or other powerful templating libraries, you may have heard about helpers, which are essentially functions that you can invoke within your templates. Underscore doesn't have helpers built in, but Marionette offers you a simple way to add helpers to your Underscore templates.

Simply provide `templateHelpers` properties containing a hash of functions that you want to be accessible within your template. Here's a quick example:

```
MyView = Backbone.Marionette.ItemView.extend({
    template: _.template('<div><%= showName() %></div>'),
    templateHelpers: {
        showName: function(){
            return this.name;
        }
    }
    //...

});
```

In the template, simply call the name of the function, like I did in the example. Within the template helper, `this` refers to the data object that was passed into template (such as the model), so you can retrieve the data you need.

Here's how it works:

1. The `serializeData` method is called to retrieve the data for the template.
2. The `templateHelpers` object's properties are added to the data.

Essentially, these template helper functions become part of the data, which is why they are accessible from within the template. These helpers are especially handy when you need to format data (such as currency) that is stored in a raw state in your models. Using a helper, you can ensure that you always have the dollar sign in front and two decimals showing:

```
MyView = Backbone.Marionette.ItemView.extend({
    template: _.template('<div><%= showMeTheMoney(cost)
    %></div>'),
    templateHelpers: {
        showMeTheMoney: function(amount){
            return "$" + amount.toFixed(2);
        }
    },
    //...

});
```

Dynamic Templates

Sometimes (though probably not often), you need to render a different template depending on the state of an object or other circumstance. In these instances, specifying a single `template` property won't get you what you desire. We can change this, though, by changing the way Marionette retrieves the template.

All of Marionette's views have a method called `getTemplate`, which we can override to retrieve the desired template for the situation:

```
var MyView = Marionette.ItemView.extend({
    defaultTemplate: '#default-template',
    specialTemplate: '#special-template',

    getTemplate: function(){
        if (this.model.get("special")){
            return this.specialTemplate;
        } else {
            return this.defaultTemplate;
        }
    },
    //...

});
```

Marionette tends to do things like this pretty well. If there is a tiny bit of functionality that you want to change, generally it is separated into its own little function that you can override. This way, we don't need to override Marionette's entire `render` function to determine which template to use each time, because we can just override a tiny chunk of what it does.

Events And Callbacks

Most of the built-in events were touched on in the previous chapter, but I want to lump them all together here so that they're more prominent, since they can be important. You see, since Marionette provides so much functionality for you, it would normally be difficult to inject your own functionality where you need it without rewriting and overriding Marionette's functions. For example, because Marionette handles rendering the view for you, how would you add some functionality that also happens during the rendering process? That's where these events come into play.

Also, Marionette has a special mechanism that allows you to trigger a method on an object that will call a function of a specific name on that object before triggering the registered event callbacks. We'll discuss this mechanism in more detail in chapter 8, but for now, you should know that for each of these built-in events, there is also a callback function you can specify that will be called before the event is triggered.

Here's a table of the events and their associated callbacks that are called on `ItemView` automatically.

Event Name	Callback Name	Description
`before:render`	`onBeforeRender`	When `render` is called, this event is fired before any of the rendering happens.
`render`	`onRender`	Immediately after rendering has finished, this event will be fired.
`before:close`	`onBeforeClose`	When `close` is called, this event is the first thing that happens. If `onBeforeClose` returns `false`, it will cancel closing the view.
`close`	`onClose`	Just before the view is removed from the DOM and all of its bound listeners are removed, this event is fired.
`show`	`onShow`	When a view is shown via a region's `show` method, the region will trigger this event after the view has been added to the DOM.

Table 1: Events and their associated callbacks that are called on `ItemView` *automatically.*

The first four events are automatically triggered by the view, and the final one is triggered by a region using the view. All of these events can come in useful at times.

You may have also noticed that `before:close` has a special capability, too. If the `onBeforeClose` method returns `false`, the view will not be closed. This can be handy with views that contain forms so we can prompt the user to make sure that they want them closed without finishing or submitting the form.

Summary

As you can see, `ItemView` is a wonderful tool in Marionette's toolbox that allows you to skip the boilerplate and simply provide some configuration and any application-specific functionality you need. It doesn't seem to specify a lot of new functionality by itself, but it handles the rendering for us simply by providing it a template, which is great. Also, you have to remember all the functionality it inherits from `Marionette.View`. Combined, this provides a very powerful view class for us. ❧

Chapter 5: CollectionView

By this point, you should have learnt a few things about `View` and `ItemView`. Now we'll be diving into another view type: `CollectionView`. While it is possible to use an `ItemView` to display data from a collection, you need to loop over the collection's models manually in the template, so you end up with a single view that displays all the models in the collection.

`CollectionView` does things a bit differently. It is designed to automatically loop through the collection for you, and create a new child view to represent each model. Since each model has a view that is solely in charge of that model, the views have simpler, more fine-grained control over displaying changes to their individual models. In the next chapter, we'll be discussing `CompositeView`, which is very similar to `CollectionView`, but has a few differences. Let's look at an overview of the main differences between all of these views:

- As mentioned, `ItemView` is designed for showing single models. While it can display collections, the template will need to loop over the collection to show all the data. `CollectionView` is designed to display collections of data and they do it by creating child views that represent the individual models.
- `CollectionView` will append the child views directly into its `this.$el` element, while `CompositeView` can render a template and append the child views into a specified element within the template.
- `CollectionView` requires you to specify the view type that it will use to render the models, whereas a `CompositeView` will default to using its own type, so a `CompositeView` (or your own view class that extends `CompositeView`) will be used to both represent the collection and the individual models. This is why it's name includes "Composite": by default it follows the composite design pattern[18]. `CompositeView` can be told to use a different view for the child views, though, so it can act more like `CollectionView`.
- `CompositeView` extends `CollectionView`, so it can essentially do anything that `CollectionView` can do, but adds more capabilities.

18. http://www.joezimjs.com/javascript/javascript-design-patterns-composite/

This chapter will dive into `CollectionView` and we'll give `CompositeView` some love in the next chapter. We'll kick off `CollectionView` by discussing what we need to get one of these views to render.

Rendering

Like `ItemView`, `CollectionView` has some configuration options that you'll need to state before it'll work correctly. For `CollectionViews`, the only thing you must specify is an `itemView`, which is the view class that you want to instantiate for each of the child views, but of course there are additional properties you can specify to customize rendering further. Here's a table of all of the properties you can set that will have an effect during rendering:

Property Name	What it's for
`itemView`	As mentioned, this is the property that lets the `CollectionView` know what type of view to instantiate for each child.
`itemViewOptions`	In addition to the model, these options will be passed into the constructor of each of the child views as it is instantiated.
`emptyView`	If this isn't specified, then, if the collection is empty (or there is no collection), the `CollectionView` will only render its root element (`el`) and won't render any child views. But if you specify a view class here, it will be displayed instead, so your users aren't presented with a blank screen.

Table 2: Properties that will have an effect during rendering.

Here's a code snippet showing each of these options:

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

});

var EmptyView = Marionette.ItemView.extend({
   ...

});

var ManyModelsView = Marionette.CollectionView.extend({
    itemView: ModelView,
    itemViewOptions: {
        someOption: true
    },
    emptyView: EmptyView
});
```

There are a few key points about `itemViewOptions` that can be helpful to know:

1. These options are also passed into the `emptyView` view's constructor.
2. It can also be defined as a function that returns an object.

If you use `itemViewOptions` as a function, it will be called for each `itemView` that is created and is passed two arguments: the model that is going to be given to view; and the index of that model in the collection – the same two arguments that would be passed to your callback function if you were using the `each` function on a collection: `collection.each(function(model, index){ ... });`).

This can be used in several ways:

- Pass different parameters into the view depending on model data/state.
- Pass the index into the view so it can render the index.
- Pass different options along depending on whether we're rendering the `emptyView` or `itemView`.

You can determine whether it's going to render the empty view in one of two ways. First of all, you can check if `this._showingEmptyView` is true, which is the simplest way. But since it's an internal mechanism, there is little guarantee that it will work in future versions of Marionette. You can also check to see if the collection exists and how many models it contains, which is how `render` decides which type of view(s) to render:

```
var MyCollectionView = Marionette.CollectionView.extend({
    itemView: MyItemView,
    emptyView: MyEmptyView,
    itemViewOptions: function(model, index) {
        // I didn't use the arguments here, but you could.

        // I just wanted you to see them
        if (this.collection && this.collection.length > 0) {
            return { /* itemView options */ };
        }
        else {
            return { /* emptyView options */ };
        }
    }
});
```

Note: if the `emptyView` is going to be rendered, then the first argument passed to the `itemViewOptions` function will simply be a new base model (`new Backbone.Model()`) and the second argument will be `0`.

Automatic Rendering

Once you've got all that done and you've instantiated the view, you can just call `render` on it like any other view and add it to the DOM, and you'll be all set. If you run `render` again, it'll close all the current views and set everything back up again with new views. It's not the most efficient way to do things but it's the simplest way to keep the views synchronized with the state of the collection/models, and if you're calling `render` again, it generally means that you want it to start over and re-render everything anyway. One cool thing to note, though, is that in a recent update to Marionette, this was updated to render everything in memory before actually being placed in the DOM, which greatly helps performance, so don't feel *too* bad about re-rendering an entire collection.

But what if something happens with the collection before you tell it to render again? Marionette has your back and won't re-render the whole collection unless it makes sense to. `CollectionView`s automatically listen for certain events on a collection and react accordingly. See the table below to see what events the views listen for and what they do in response.

Event	Action
`reset`	The view will be re-rendered
`add`	A new view will be created for the newly added model and appended at the end of the list. If the empty view was showing, it will be closed first.
`remove`	The view that corresponds to that model will be closed and removed, leaving the rest of the views intact.

Table 3: Events the `CollectionView`*s listen for and what they do in response.*

Closing

Just like any other view from Marionette, `CollectionView` has a `close` method. It's a little more involved, since it needs to iterate through all of its child views and close them all first, but it is essentially the same thing as all the other views. All you need to know about this is that the `CollectionView` will take care of all the other views for you, so you don't have to manually account for countless zombie views lying around everywhere.

Events and Callbacks

Also like the other Marionette views, `CollectionView` will automatically fire different events and call specially named callbacks for those events in certain situations. These can be used to extend and customize your views and also give other objects in your system a heads-up on what is going on inside your views.

`CollectionView` has a similar set of events — and related callbacks — to `ItemView`, but it also has many differences. All of the built-in events and callbacks are listed below:

Event Name	Callback Name	Description
`before:render`	`onBeforeRender`	When `render` is called, this event is fired before any of the rendering happens.
`render`	`onRender`	Immediately after rendering has finished, this event will be fired.
`collection:before:close`	`onCollectionBeforeClose`	When `close` is called, this event is the first thing that happens. It happens just before closing all of the child views.
`collection:closed`	`onCollectionClosed`	This event fires immediately after all of the child views have been closed. Interestingly, this

		is the only `close`-related event with a past tense "closed". Might catch you by surprise.
`before:close`	`onBeforeClose`	After the child views are closed, we get into the normal procedure for closing the parent view. This event fires before the view actually starts closing. Sadly, unlike `ItemView`, you can't have the callback return `false` in order to cancel closing a `CollectionView`.
`close`	`onClose`	Just before the view is removed from the DOM and all of its bound listeners are removed, this event is fired.
`before:item:added`	`onBeforeItemAdded`	This event is fired when a child view is about to be added to the `CollectionView`. This happens no matter how the view is added (whether during rendering, or when a model is added to the collection, or manually).
`after:item:added`	`onAfterItemAdded`	This event is fired when a child view is finished being

		added to the CollectionView.
item:removed	onItemRemoved	This event is fired when a child view is removed from the CollectionView.
show	onShow	When a view is shown via a region's show method, the region will trigger this event after the view has been added to the DOM.

Table 4: Built-in events and callbacks in CollectionView.

The CollectionView also listens to all of the events of each child view. When a child view fires an event, the CollectionView will be notified and will rebroadcast that event itself with a prefix of "itemview:". For example, if one of the child views fires the render event, CollectionView will then fire the itemview:render event. CollectionView uses the same mechanism when firing these events as it does when it fires its other events, so in the current itemview:render example, CollectionView would invoke the onItemviewRender function just before firing the event. For more information on the mechanism that provides this functionality and how the naming works, look for Marionette.triggerMethod in chapter 8.

Here's a sample of the events emitted from a CollectionView that is displaying two child ItemViews:

- before:render
- collection:before:render
- before:item:added
- itemview:before:render
- itemview:item:before:render
- itemview:render
- itemview:item:rendered
- after:item:added

- `before:item:added`
- `itemview:before:render`
- `itemview:item:before:render`
- `itemview:render`
- `itemview:item:rendered`
- `after:item:added`
- `render`
- `collection:rendered`

That is quite a number of events triggered simply by rendering a `CollectionView` but, luckily, you shouldn't need to memorize them. Just look through Marionette's source code or documentation to find out where you want to inject your functionality and add your event handlers and callbacks in there.

Customization

While the event callbacks offer a decent amount of customization, there's always room for more customization. We're programmers after all — custom is in our blood! So, let's discuss a few more ways we can customize `CollectionView`.

ITEMVIEWEVENTPREFIX

In the previous section, we discussed how the `CollectionView` will refire all of the events from its child views with a prefix. With `itemViewEventPrefix`, we can change what that prefix is. Just specify a string value to replace "itemview" as the prefix for events.

I don't recommend doing this without a bit of thought, since some may automatically subscribe to the itemview-prefixed events. If you're going to change this, then you'll either want to change this to the same value for every `CollectionView` (and `CompositeView`) in your application, or you'll want to have a very obvious naming scheme that determines what the prefix will be for each view. Overall, though, I wouldn't bother changing the prefix unless you have a very good reason for doing so. This feature was added for a very special case that you aren't likely to run into.

METHOD OVERRIDING

Of course, as with any object-oriented programming language, you can override any method of a parent class. While it isn't likely you'll do it often, overriding methods can afford you some powerful customization that isn't achievable any other way. Of course, in order to know exactly which methods to override and how to override them, you'll need to look through Marionette's source code, which is actually very good practice anyway. Not only will you learn a lot, you may even spot a bug that you can help the developer fix.

Whenever you decide that you want to override a method, there are a few things you should keep in mind.

- Make sure you completely understand what is going on inside the original method. You may want to override a method just to change the order in which some operations take place, or remove a function call. But unless you know why things are done in that specific order or what that function call is for, you could end up causing more problems than you are trying to fix.
- You'll likely be overriding a method that is used internally and isn't intended to be part of the public API. This means that the library author may change implementations of that method, or even remove or rename it without any notice. If you override a method, make sure that you check to make sure that there weren't any changes that could affect it when you upgrade to newer versions of Marionette.
- Other code authors might not understand how the overridden method is being used, so make sure you explain that it overrides a built-in method and include why you are overriding it.

In other words, overriding methods is a powerful tool in your arsenal, but it should be done with a measure of caution, and your intentions should be made clear so that other developers (and potentially your future self) know exactly what is going on.

Summary

Marionette does a wonderful job of cutting complex processing down to a few pieces of configuration and the `CollectionView` is one of the prime examples. Specify a child view type, give it a collection, and you're ready to spit out some HTML. ❧

Chapter 6: CompositeView

In the last chapter, we discussed `CollectionView`. This chapter is about `CompositeView`, which inherits most of its functionality from `CollectionView`, so most of what I've mentioned in chapter 5 is also true for `CompositeView`. There are differences, obviously – otherwise we wouldn't need both view classes – and instead of going through everything about `CompositeView`, this chapter will only explain what's different between `CollectionView` and `CompositeView`.

Rendering

`CollectionView` cannot render templates (it simply appends the child views to its `el`), but `CompositeView` *can* render a template, so rendering – and the properties that affect it – is probably the biggest difference between the two view types. There are also some other aspects that affect the differences in how the two views render, so let's take a closer look.

THE TEMPLATE

`CompositeView` renders a template in much the same way `ItemView` does. The first and most obvious thing you need to do is provide a `template` property. Go back and look at `ItemView` in chapter 4 for more information about how that works.

An example of a `CollectionView` that would need a template is building a table with headers. Each child view is a row in the table, but the first row of the table – the headers – isn't built from data, so it should be in a template. Having a template would also allow you to add in the `thead` and `tbody` elements for more semantic goodness.

WHERE DO WE PUT THE KIDS?

Now, since we'll be rendering child views, we need to inform the view where child views will be shown within our template. By default, if you don't specify where the child views should be added, they will be appended to the view's root `el`, right after the template. This can be what you want sometimes, but generally you will need to place them somewhere inside your template. To do this, we use the `itemViewContainer` property to specify a selector for the element we want the child elements placed into.

COMPOSITE PRINCIPLES

There is something about the way `CompositeView` renders the template that is slightly different from `ItemView`. `ItemView` sends either the model or the collection to the template for rendering, depending on what exists (or an empty object if neither exists). `CompositeView` will only send the model (or an empty object if there is no model) to the template. It will then render the collection (if there is one) as child views.

This means we need both a model *and* a collection in order to make the `CompositeView` useful (though usually we only need to provide the model, as the collection will be extracted from it, as you'll see later on). Also, if your template doesn't require any data (such as when it's just building the shell of a table or list), you can do without the model as well. It renders the data from the model within the template and then iterates through the collection, creating a child view for each model in that collection. If all you have is a model, it'll pretty much act just like an `ItemView`. If all you have is a collection, then I hope your template doesn't require any data.

Another interesting and unique thing about `CompositeView` is that although you *can*, you don't *need* to supply an `itemView` property. If an `itemView` is provided, it'll act like a `CollectionView` and render each child view as the type of view specified. But if `itemView` isn't specified, it will default to recursively using its own class for each child view.

Recursion is often difficult for people to wrap their heads around, even if they are familiar with the concept and have seen it in practice. This is more than a simple function calling itself, so I'll try to give some concrete examples, which I'll steal from a post by Derick Bailey[19], since he has a simple example that does a good job at focusing on the concept. However, as some people pointed out in the comments of that article, there are some flaws in the way he implemented the example, so I'll actually use the code sample[20] that one of those commenters offered as a replacement.

Recursive Data

Before I show how `CompositeView` recursively creates the views, though, I need to show you how recursive data works. Without recursive data, there is no reason to make the views recursive, since views are just visual representations of the data. When I talk about recursive

19. http://lostechies.com/derickbailey/2012/04/05/composite-views-tree-structures-tables-and-more/
20. http://jsfiddle.net/dmitri14/St2dB/1/

data, I'm really talking about a tree of data in which a single node can have child nodes. There are plenty of great places to learn about composite data structures online[21], and it's a bit outside the scope of this book to go into it too much. If you don't already know much about it, then I suggest you learn more about it online before continuing. I'll still show examples of this sort of data structure so you know how it's implemented in Backbone.

We'll start with a collection of models. The models will simply be called "nodes" so I don't need to contrive some strange reason for having a tree.

```
var Node = Backbone.Model.extend({
    initialize: function(){
        var nodes = this.get("nodes");
        // Covert nodes to a NodeCollection
        this.set('nodes', new NodeCollection(nodes));
    },

    toJSON: function() {
        // Call parent's toJSON method
        var data = Backbone.Model.prototype.toJSON.call(this);
        if (data.nodes && data.nodes.toJSON) {
            // If nodes is a collection, convert it to JSON
            data.nodes = data.nodes.toJSON();
        }

        return data;
    }
});

NodeCollection = Backbone.Collection.extend({
    model: Node
});
```

As you can see, when a model is created, it will convert the `nodes` property's data to a `NodeCollection` from that data and assign it to the `nodes` property. Now you have a node that contains a collection of nodes. The nodes in that collection could also have their own collections, but not necessarily (no one likes infinite recursion). We also customize the `toJSON` method to also call `toJSON` on the child `NodeCollection`.

21. http://www.joezimjs.com/javascript/javascript-design-patterns-composite/

Here's some sample data that we could use plus the code to set it up:

```
var nodeData = [
    {
        nodeName: "1",
        nodes: [
            {
                nodeName: "1.1",
                nodes: [
                    { nodeName: "1.1.1" },
                    { nodeName: "1.1.2" },
                    { nodeName: "1.1.3" }
                ]
            },
            {
                nodeName: "1.2",
                nodes: [
                    { nodeName: "1.2.1" },
                    {
                        nodeName: "1.2.2",
                        nodes: [
                            { nodeName: "1.2.2.1" },
                            { nodeName: "1.2.2.2" },
                            { nodeName: "1.2.2.3" }
                        ]
                    },
                    { nodeName: "1.2.3" }
                ]
            }
        ]
    },
    {
        nodeName: "2",
        nodes: [
            {
                nodeName: "2.1",
                nodes: [
                    { nodeName: "2.1.1" },
                    { nodeName: "2.1.2" },
                    { nodeName: "2.1.3" }
                ]
            },
            {
                nodeName: "2.2",
                nodes: [
                    { nodeName: "2.2.1" },
                    { nodeName: "2.2.2" },
                    { nodeName: "2.2.3" }
                ]
            }
        ]
    }

];

var nodes = new NodeCollection(nodeData);
```

So that's the data that will be used for the example. Let's prepare the views.

RECURSIVE VIEWS

We're going to create a `CompositeView` that creates a simple list out of this data, like this:

- 1
    - 1.1
        - 1.1.1
        - 1.1.2
        - 1.1.3
    - 1.2
        - 1.2.1
        - 1.2.2
            - 1.2.2.1
            - 1.2.2.2
            - 1.2.2.3
        - 1.2.3
- 2

- 2.1
  - 2.1.1
  - 2.1.2
  - 2.1.3
- 2.2
  - 2.2.1
  - 2.2.2
  - 2.2.3

To do this, each model will be represented by a `li` element, and each collection of models will be represented by a `ul` element. Since the `CompositeView` itself actually represents the model, we need a way to put all of them into an initial `ul` element. For this, we'll also create the following `CollectionView`:

```
var TreeRoot = Marionette.CollectionView.extend({
    tagName: "ul",
    itemView: TreeView
});
```

`TreeView` will be our `CompositeView`. `TreeRoot` will take the `nodes` collection we created earlier and create the two main `CompositeView`s, which will take it from there. First of all, we should have a template for the `CompositeView`. Let's add that to the HTML right now:

```
<script id="node-template" type="text/template">
    <%= nodeName %>
    <ul></ul>
</script>
```

It's a very simple template. It just displays the `nodeName` from the model and has a `ul` to add the child views to. Alright, let's stop beating around the bush and finally create that `CompositeView`.

```
var TreeView = Marionette.CompositeView.extend({
    template: "#node-template",
    tagName: "li",
    itemViewContainer: "ul",

    initialize: function(){
        // grab the child collection from the parent model
        // so that we can render the collection as children
        // of this parent node
        this.collection = this.model.get('nodes');
    },

});
```

Let's step through each line:

- Create `TreeView` by extending `Marionette.CompositeView`.
- Assign the template that we created.
- Give it a `tagName` of `li`. This is built into Backbone and changes the root element's (`this.el`) type.
- Set the `itemViewContainer` to `ul`, so that all the child views will be appended to the `ul` in the template.
- Use the `initialize` function to set up our collection. Since each `TreeView` will only be receiving the models, we need to extract the collections out of the models.

None of that should have been new except the last step. That last step is very important, and you may often need to do something similar when using the `CompositeView` in its default recursive mode.

Now the only thing that's left to do is render it all and put it on the page:

```
var tree = new TreeRoot({collection: nodes});
tree.render();
$('body').append(tree.el);
```

Stealing from the ItemView

Practically everything from `ItemView` also applies to `CompositeView`. In particular, `templateHelpers` and dynamic templates via overriding `getTemplate` make an appearance at the `CompositeView` party. There isn't really any reason to go deeper into either one, but I thought I'd show you a way of overriding `getTemplate` that would make our example a little better.

Right now, we only have a single template and that template has the child `ul` element in it. This means that even `CompositeView`s that have no collection will display the `ul` but it will be empty. Let's come up with a way to render a different template when we don't need the `ul`.

First, let's add the new template. It will simply remove the `ul` from the current template.

```
<script id="leaf-template" type="text/template">
    <%= nodeName %>
</script>
```

Now we need to change `TreeView` so that it will use a dynamic template:

```
var TreeView = Backbone.Marionette.CompositeView.extend({
    getTemplate: function() {
        if (_.isUndefined(this.collection)) {
            return "#leaf-template";
        } else {
            return "#node-template";
        }
    },

    tagName: "li",
    itemViewContainer: "ul",

    initialize: function(){
        this.collection = this.model.get('nodes');
    },

});
```

We don't even need to specify a `template` property, but we *do* need to override `getTemplate` to choose which template to use depending on whether or not we have a collection. This may make the JavaScript code a bit more cluttered, but it cleans up the HTML, and depending on your style sheet, clearing those extra `ul` elements out of there may be absolutely necessary.

Events and Callbacks

As usual, `CompositeView` has built-in events and callbacks, most of which are inherited from `CollectionView`. `CompositeView` implements a few of its own events between the ones it inherits. The only events it adds, though, are during rendering. All the other events are inherited. Let's take a look at the new events and callbacks.

Event Name	Callback Name	Description
`composite:model:rendered`	`onCompositeModelRendered`	After all of the `before:render` events, the template and model data will be rendered. This event will fire immediately after that rendering.
`composite:collection:rendered`	`onCompositeCollectionRendered`	This event fires following the child views being rendered.
`composite:rendered`	`onCompositeRendered`	This event fires immediately after `composite:collection:rendered` and just before the `render` and `collection:rendered` events.

Table 5: New events and callbacks in `CompositeView`.

Like `CollectionView`, `CompositeView` also listens for and forwards the events of the child views. *That's a lot of events being fired*. Here's a sample of the events (in order) that you would see from a recursive `CompositeView` that is rendering only two child views:

- `before:render`
- `collection:before:render`
- `composite:model:rendered`
- `before:item:added`
- `itemview:before:render`
- `itemview:collection:before:render`
- `itemview:composite:model:rendered`
- `itemview:composite:collection:rendered`
- `itemview:composite:rendered`
- `itemview:render`

- `itemview:collection:rendered`
- `after:item:added`
- `before:item:added`
- `itemview:before:render`
- `itemview:collection:before:render`
- `itemview:composite:model:rendered`
- `itemview:composite:collection:rendered`
- `itemview:composite:rendered`
- `itemview:render`
- `itemview:collection:rendered`
- `after:item:added`
- `composite:collection:rendered`
- `composite:rendered`
- `render`
- `collection:rendered`

That's noticeably more than `CollectionView`, but as I said earlier, you don't need to memorize them. Just look them up when you need them.

Summary

It's truly amazing that something as intricate and complex as displaying a collection or a composite architecture can — for the most part — be simplified down to just a few pieces of configuration. Together, the view classes are probably the most powerful and useful pieces that Marionette provides. Of course, that doesn't mean that Marionette isn't chock-full of other extremely useful goodies, as you've already seen and will soon see again in the upcoming chapters. ❧

Chapter 7: Region and Layout

In the first chapter, you may remember me mentioning being able to add `Region`s to the `Application` object with some simple configuration. At that point I didn't go into many details about the specifics of a `Region`; nor did I give many details any of the other times it was mentioned throughout the chapters, but finally we've reached the point where I divulge the intimate inner workings of the amazing `Region`! Not only that, but we have a double-header because this chapter also talks about `Layout`.

Why do both in one chapter? Well, `Layout` is simply a special type of `ItemView` that contains `Region`s. We've already spent a lot of time discussing `ItemView` and soon all the secrets of `Region` will be revealed, so there really won't be much to discuss concerning `Layout`, and it fits pretty well into this chapter along side `Region`.

Region

Let's kick this chapter off with the `Region` class that Marionette provides for us. What exactly is a `Region`? A `Region` is an object that manages rendering and closing views in a given DOM element. Each `Region` needs to have a DOM element associated with it; then when you wish to render a view inside that area of the DOM, just tell the `Region` to show it. If you want to show a different view later, the `Region` will close the previous one automatically and render the new one in its place. You can also just close down the current view without showing a new one.

Before we get too far into that, let's go over how we create a `Region`.

CREATING AND EXTENDING

As long as a DOM element is provided, you can use the `Region` provided by Marionette straight out of the box. Most of the time you won't need to perform any kind of special customization because a `Region` generally just does what it does and nothing else. Of course, there may be some situation where you need to override a built-in behavior – maybe to throw in some animation when rendering a new view – or add in some of your own functionality, so let's show how you would do that.

As with practically everything else in Backbone and Marionette, `Region` comes with an `extend` function so you can easily extend the

`Region` class. The only property that does anything special (other than the usual `initialize` method) is `el`, which is the selector for the DOM element where the region will inject the views.

Usually `el` is supplied when instantiating a `Region` object, and specifying the `el` before you instantiate it couples the `Region` with that element. You could just define `el` in your subclass as a backup default, since if an `el` is supplied during instantiating the `Region`, it will override the `el` provided in the prototype. But as you'll see, there are very few instances where you'd even want to bother adding that default.

Anyway, here is a quick example of extending `Region`:

```
var MyRegion = Marionette.Region.extend({
    el: "body", // Default element
    initialize: function(options) {
        // Special initialization stuff
    }
});
```

Now that you have your own custom `Region` type, or you've decided you'll just use the standard one, we need to instantiate some of these bad boys so we can put them to work. There are several ways to do this. Our first option is the old-fashioned way:

```
var region = new Marionette.Region({el:'#selector'});
```

In practice, however, you probably won't need to explicitly instantiate `Region` objects, as you'll see. A quick note, though: if a `Region` does not have an `el` at the time it's created (either from the prototype or being passed in like in the example I just showed you), then it will throw an error.

The other means for instantiating `Region` objects come from `Application` (we touched on this in chapter 1) and `Layout`. To add regions to an `Application` object, you use its `addRegions` method, which has a few signatures so you can easily use custom or standard `Region` types without any hassle.

The first signature requires a name and a selector. The name allows you to reference the `Region` as a property of that name on `Application` and the selector specifying where the `Region` gets attached.

```
// Add some standard regions to an Application
App.addRegions({
    header: '#header',
    body: '#content',
    footer: '#footer'
});
```

```
// Regions can then be retrieved like this
App.header;
// or
App.getRegion('header');
```

The second option is similar, but instead of providing a selector string, you provide a `Region` class. For this method, you'll need to have specified an `el` property when you create the class.

```
// Create custom Region types
var HeaderRegion = Marionette.Region.extend({
    el: '#header'
});

var BodyRegion = Marionette.Region.extend({
    el: '#body'
});

// Add this custom Region type to the Application
App.addRegions({
    header: HeaderRegion,
    body: BodyRegion,
    footer: '#footer' // to show that you can mix signatures
});
```

Finally, you can use a combination of the two where you specify a custom `Region` type *and* a selector. For the example below, we'll continue to use the `HeaderRegion` and `BodyRegion` definitions that we just saw.

```
App.addRegions({
    header: {
        // selector overrides the el in the class definition
        selector: '#header',
        regionType: HeaderRegion
    },
    body: BodyRegion,
    footer: '#footer' // to show that you can mix signatures
});
```

A quick note about `addRegions`: if you use it to add a region that has the same property name of one that already exists, the one that already exists will be replaced with no guarantees about where it will end up. It could potentially be garbage collected if there are no other references to it, but otherwise it could sit around in memory doing nothing; worse, it could still react to events and such without your knowledge. This probably won't end up being an issue because I can't think of any reason anyone would adding regions willy-nilly without knowing what's already there, but I figured I would give you fair warning, just in case some crazy people get their hands on your source code or something.

You can also instantiate `Region` objects with a `Layout`, but we'll discuss that more a little later, when we get to the details of `Layout`.

USING A REGION

Now we actually get down to the fun part of the `Region`: using it! Once a `Region` is created, it greatly simplifies the act of rendering a view in the DOM. Let's take a look to see what I mean.

Show

A `Region`'s `show` method is the magic method. It's the one you'll use most, so get well-acquainted with it. The `show` method takes a single parameter: a `View` object. It will then take that view, render it, add it to the DOM and trigger a show event on both the `Region` and the `View`. If there was already a `View` being shown in the `Region`, it will first close the existing `View` before rending the new one.

```
var region = new Marionette.Region({el:'body'});
var view1 = new MyView1(); // Pretend I already implemented
// MyView1
var view2 = new MyView2(); // Pretend I implemented this too

region.show(view1); // The view's el now exists in the body
region.show(view2); // view1 is gone. view2 took its place
```

This comes in quite handy. For example, when a new route is picked up, we need to replace the main view with a new one. If you have a region on your `Application` for that main view area, you can just call `show` on it with the new view: `App.body.show(new FooView());`. It doesn't get much simpler than that.

Close

If you want to stop showing a view in the `Region` altogether, without replacing it with a new view, you rely on `close`. The `close` method will call the `close` or `remove` method (`remove` is only called if `close` doesn't exist) on the `View`, trigger a close event on itself (calling `close` on a `View` will automatically trigger a close event on it) and remove the `View` from its memory. It doesn't take any parameters.

```
// Continuing the above example, we decide we don't want to see
// any views in this Region anymore
region.close();
```

Attaching an Already Rendered View

If you render your HTML on the server and then just have views attach themselves and manage the DOM elements that already exist, you can still use a `Region` to manage that view. Each `Region` comes with an `attachView` method that takes a `View` as its sole parameter and simply assigns the provided view as the current view. It won't close any views that it may have been holding already and it won't call render or trigger a show event on the view. If you use `attachView` in cases other than the one described, you may need to remember to call `close` on the `Region` first.

```
// Create a View that attaches itself to an existing DOM element.

view1 = new MyView1({el:'#foo'});
// You shouldn't need to render, but you might need to do
// something to set up event listeners and such.
// delegateEvents is provided by Backbone.

view1.delegateEvents();

// Now let's let the Region manage it.

region.attachView(view1);
```

Resetting a Region

Resetting a `Region` is useful for reusing it. If you want to continue using the same `Region` in a new place after the place it previously used no longer exists (maybe in unit testing? It's also used internally by `Layout`), you can reset it, which closes the `View` and uncaches the reference to the DOM element. The next time you call `show` on the `Region`, it'll query the DOM again for that element. This allows you to change the `Region`'s selector when it is reused.

```
App.body.reset();

// Give it a new selector
App.body.el = "#foo";
// Reuse the Region where it is
```

```
App.body.show(fooView);

// Now let's move the Region somewhere else
App.body.reset();
someObject.region = App.body;
App.body = null;

// Use it now with someObject
someObject.region.el = "#bar";
someObject.show(barView);
```

If you look at that and wonder why you would do it, don't worry. The example was just written to show what you *can* do, not what you *should* do.

EVENTS

Once again, let's take a closer look at the events that are built in to the component we're studying. `Region` doesn't give us much, though, so it's a pretty short look.

Event	Action
`show`	When you call `show`, after the `View` is rendered and placed into the DOM, the show event will be triggered on the view and then on the `Region`.
`close`	When you call `close`, after the `View` is closed and removed from the `Region`, the close event will be triggered on the `Region`.

Table 6: Built-in events in `Region`.

Layout

Marionette's `Layout` is the final `View` type given to us. `Layout` inherits from `ItemView` and acts just like it in every way, except that it is designed to have `Region`s attached to it, similar to the way `Application` does. By using `Region`s inside of a `View`, we can have subviews that aren't automatically generated and can be swapped out at any time.

When evaluating the selectors for each `Region` on a `Layout`, the search will be in the context of `Layout.$el`, so the `Region`s can only exist within the `Layout`, not outside of it.

Just like the `Application`, a `Region` will be accessible directly on the `Layout` object via a property matching the name of the region; for example, if you name a region "header", it will be accessible via the `header` property on the `Layout`.

MANAGING REGIONS ON A LAYOUT

There are several ways we can add `Regions` to a `Layout`. The first, simplest, and probably most appropriate way is to define them when extending `Layout` to make your own custom `Layout` type. Do this by adding a `regions` property. (I honestly wish this method was available for `Application` so I'm not required to add the `Regions` after the `Application` is already instantiated.) Using `regions` you can specify the hash in the same way you would specify it in a call to `App.addRegions`.

If you want your `Regions` to be something other than the standard `Marionette.Region`, you can also provide a `regionType` property, which will make any `Regions` without a specified type be of the type mentioned in `regionType`.

```
var FooLayout = Marionette.Layout({
    regionType: CustomRegion,
    regions: {
        header: {
            selector: '#header',
            regionType: CustomHeaderRegion
        },
        body: CustomBodyRegion, // Has an el already
        footer: '#footer' // Will be a CustomRegion
    },
   ...

});
```

After the `Layout` is initialized, these three `Regions` will be available on the `Layout` as the `header`, `body`, and `footer` properties.

Instead of predefining the `Regions`, you can dynamically add `Regions` to the `Layout` after it has been instantiated by using `addRegions`. It works exactly the same way as the `addRegions` method on `Application`, except that if you have a `regionType` specified on the `Layout`, and a `Region` class isn't specified for the `Region` you're adding, it will default to the `regionType` specified rather than `Marionette.Region`.

```
var layout = new FooLayout(); // The same Layout as above
layout.addRegions({
    foo: '#foo', // Will be a CustomRegion
    bar: {
        selector: '#bar',
        regionType: BarRegion // Will be a BarRegion
    }
});
```

Layout has a few more methods — which oddly aren't available on Application — for managing Regions. The first one is addRegion (notice the singular "Region"). Instead of providing an object literal, you must provide two arguments: the name of the Region (like "foo" and "bar" in the previous example with addRegions); and the definition for the Region. The documentation only shows an example of addRegion where you specify a selector for the second parameter, but it also supports the other definition types of specifying a Region class or an object literal with the selector and regionType properties.

I personally only use addRegions since it can be used to add a single Region and I don't need to remember if it's Layout or Application that implements addRegion. I would recommend you do this too, but it's entirely up to you.

Layout also has a removeRegion method whereby you can provide a string with the name of a Region you'd like to remove from the Layout. If the region is currently showing a View that you want closed, you'll have to call close first, since removeRegion doesn't do it for you.

```
layout.foo.close();
layout.removeRegion('foo');
```

REGION AVAILABILITY AND RENDERING BEHAVIOR

The defined Region objects are available to use immediately after the Layout is initialized. This is done just in case you attach your Layout directly to an existing DOM element, in which case you wouldn't need to render the Layout. However, if the Layout isn't attached to the DOM right away, you'll need to render it before the Regions will have any DOM elements to attach to.

The first time a Layout is rendered, it doesn't do anything special for the Regions except make sure they are attached to a DOM element. Every time you render the Layout after that, all of the Regions will be closed and reset, so make sure they are attached to the newly rendered DOM elements. Because of this, it is generally recommended that you do not re-render a Layout because it will require a lot of extra work to get all of the subviews attached to their Regions again and re-rendered, which must all be done manually.

CLOSING THE LAYOUT

The final thing to note about `Layout` is that, like every `View`, it has a `close` method. The `close` method on `Layout` will also call `close` on each of its `Region`s, making sure that all of the subviews are closed down too.

Summary

Well, that's `Region` and `Layout`. `Region`s offer you simple, but powerful management of rendering and closing views. `Layout`s give you the ability to render a `View` and then use the power of `Region` to manage its subviews. You could say that 'simple but effective' is the theme of this chapter, or even the theme for most of the Marionette library. ❧

Chapter 8: Events, Commands, and Requests and Responses

In object-oriented programming, there are countless good reasons (testability, maintenance, and so on) as well as countless good ways (dependency injection, interfaces, and the rest) to decouple objects from one another. One of the ways to decouple objects is by using events. If you've used Backbone for any length of time, I hope you're familiar with events because Backbone uses them quite liberally and consistently throughout the library.

Events belong to a category of decoupling known as messaging. Of course, what good is a category if there is only one item that fits into that category? Pretty pointless, so obviously there are more patterns that fit the bill. Some of those are commands and request/responses. Each of these messaging services is also available in Marionette, thanks to the Backbone.Wreqr[22] extension, also written by Derick Bailey, which is available through `Backbone.Wreqr`.

As you'll see later on, each of these messaging services is available on the `Application` object, so they offer their capabilities to the entire application out of the box.

Events

Events are an implementation of the observer pattern[23] that allows multiple listeners to subscribe to an event. When that event triggers, all of the listeners will be called.

Backbone provides its own `Backbone.Events` object that gives you one of the better event systems out there (though it's missing some of the ultra cool features like prioritizing or namespacing, which are unnecessary most of the time). You can mix `Backbone.Events` into any of your objects to access Backbone's event handling capabilities. Wreqr uses these exact same `Backbone.Events` objects to build its `Backbone.Wreqr.EventAggregator`. The only difference is that `Backbone.Events` is designed as a static object to mix in to other objects, whereas Wreqr's `EventAggregator` is an instantiable object that you

22. https://github.com/marionettejs/backbone.wreqr
23. http://www.joezimjs.com/javascript/javascript-design-patterns-observer/

can also directly extend. In other words you can create a `new EventAggregator()` or you can even extend it with `EventAggregator.extend({...})`.

You can find an instance of the `EventAggregator` on your `Application` objects through `application.vent`, which we discussed in chapter 1. Also, since the `EventAggregator` is essentially the same thing as `Backbone.Events`, I won't go into any more detail about how to use it, since you should have gleaned that from the Backbone documentation when you were learning that.

I will however, mention one more thing related to events that Marionette provides us: `Marionette.triggerMethod`. Normally you would just call `trigger` to fire an event, but Marionette provides a special method that is often mixed into objects such as `Marionette.ItemView` so you can call it directly on that object.

When you call `trigger`, it simply loops through each of the listeners subscribed to the event you passed in and calls them. `triggerMethod`, on the other hand, will first take the event name and transform it into a method name (that is, it would convert the some:event event name to `onSomeEvent`), and invoke the method with that name before moving on to trigger the event handlers. This method should be found on the object that the event is triggered on.

A very simple algorithm is used to generate the method name that it calls before triggering the event. First, it takes the event name (some:event); then it capitalizes the first letter and each letter immediately following a colon; then it removes the colons and prefixes the whole thing with "on". As I pointed out a moment ago, some:event would be converted to `onSomeEvent`, as you can see in the following code example. If the method doesn't exist, it'll just skip ahead to triggering the event.

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

    initialize: function() {
        this.listenTo(this, 'some:event', function(){
            console.log('some:event triggered');
        });
    },
    onSomeEvent: function() {
        console.log('onSomeEvent called');
    },
    ...

});
```

```
view = new MyView();
view.triggerMethod('some:event');
// => logs 'onSomeEvent called' then 'some:event triggered'
```

Notice that `onSomeEvent` was called first when you used `trigger-Method('some:event')`. After that, it actually called the listeners to some:event.

This is a useful tactic that negates the need for objects to listen for their own events, and instead just creates methods on the object following a specific naming scheme to ensure the method is called when it should be. This, of course, only works if you use `triggerMethod` instead of just `trigger`, but that's not too difficult.

Commands

Using commands is very similar to using events, but commands have very different semantics and purpose. The biggest difference is that you can only have one listener subscribed to any one event. That and they aren't called listeners or events. They are called commands and handlers respectively.

Wreqr exposes the `Backbone.Wreqr.Commands` class, which is automatically instantiated on every `Application` object as `application-.commands` to create an application-wide command bus. You may also instantiate your own `Commands` or `extend` it.

SETTING HANDLERS

In order to set a handler for a command, use `setHandler`. Just like the `on` methods for events, `setHandler` can take up to three arguments. The first is the name of the command you are setting the handler for. The second parameter is the function that will be run when the command is executed. The third is optional and it is the context in which the handler function will be executed. If the third argument is specified, then the `this` keyword inside the handler function will refer to the object passed in as the third argument.

```
commands = new Backbone.Wreqr.Commands();
commands.setHandler('foo', someObject.foo, someObject);
```

If you'd like to specify multiple handlers at once, you can use `setHandlers`, which takes an object literal specifying multiple handlers.

```
commands.setHandlers({
    // This first handler is the same as the previous example
    'foo': {
        callback: someObject.foo,
        context: someObject
    },
    // Don't use a context for this one
    'bar': function(){...}
});
```

As you can see, you use the name of the handler as the property name on the object hash. Then you either specify just the handler function, or you specify an object with the `callback` and `context` properties, which refer to the handler function and the context object respectively.

If you ever want to replace a handler, you can just call `setHandler` or `setHandlers` again and specify a new handler for the command you'd like to replace.

```
// Set a handler
commands.setHandler('foo', function() {...});
// Replace the existing handler
commands.setHandler('foo', function() {...});
// Replace it again while also setting other handlers
commands.setHandlers({
    'foo': function() {...},
    'bar': function() {...},
    'baz': {
        callback: someObject.baz,
        context: someObject
    }
});
```

REMOVING HANDLERS

If you no longer want a handler to be specified, you can remove it with `removeHandler` or you can remove all of them at once with `remove-AllHandlers`. You don't need to pass any arguments in for `removeAll-Handlers`, but for `removeHandler`, you need to pass in one: the name of the handler as a string.

```
commands.removeHandler('foo'); // "foo" handler is gone
commands.removeAllHandlers(); // remaining handlers are gone
```

EXECUTING COMMANDS

Now we get to the point where we actually use the handlers. In order to execute a handler, you call `execute`. The first parameter should be the name of the command that you want to execute. Any other parameters that you pass in will be forwarded to the handler.

```
commands.setHandler('foo', function(arg) { console.log(arg); });
commands.execute('foo', "Some Arg");
```

After calling `execute` in this example, "Some Arg" will be printed out in the console.

If there are no handlers set up for a command, the command execution call will be stored until the handler is added.

```
var commands = new Backbone.Wreqr.Commands();

// Nothing happens either time
commands.execute('foo');
commands.execute('foo');

// After the handler is set, it will run twice because execute
// has been called twice before any handler was specified
commands.setHandler('foo', function(){...});

// Calls the handler immediately
commands.execute('foo');

// Will set a new handler, but will not execute it immediately
commands.setHandler('foo', function(){...});

// Calls the new handler
commands.execute('foo');
```

This guarantees that no matter when a handler is added, when you call `execute`, something will respond to it eventually. Granted, if no handler is ever added, then nothing will ever happen. Like events, this is a great way to extend applications. You can build an application that executes these commands all over the place, even though nothing yet implements handlers for those commands. Later, if someone wants to extend your application to add features, they can hook into one of these commands and respond to them. Or, if you would like to override some functionality of the application, you can replace the handler that already exists to do something new.

Requests And Responses

Requests and responses work in the exact same way as commands, except that the handler is expected to return a value (the response). In fact, the `Backbone.Wreqr.RequestResponse` class is so much like the class for commands, that they share the exact same `setHandler`, `setHandlers`, `removeHandler`, and `removeAllHandlers` methods. The only difference in the interface is that `execute` is replaced with `request`.

The `request` method takes the same parameters as `execute`, but operates slightly differently. First, it will return the return value of the handler. Second, if a handler is not set up for this request, it will throw an error instead of queueing up the request for later. This is because when you call `request`, you expect a response right away. You can't afford to wait until later, because then you won't actually receive the response.

Also, once again, you can instantiate your own `Backbone.Wreqr.RequestResponse` object, extend it, or find an instantiation of it already residing on each `Application` object as `application.reqres`.

Summary

Each of these methodologies is great for decoupling modules of your application and allowing for simple extension of your application, or replacement of modules. Each one has its special quirk (multiple handlers, queueing requests until a handler is present, or returning a value) so they are each best used in specific situations – make sure you use them liberally, but wisely. ❧

Chapter 9: AppRouter

Assuming you've used Backbone prior to reading this book — this entire book assumes you have, even if I myself wouldn't — you should be familiar with `Backbone.Router`. Routers are great things that allow single page applications to have multiple URLs as if they were applications spanning multiple pages. This grants users the ability to use the Back and Forward buttons in the browser to navigate around apparent pages of your application that they have seen before, or even bookmark a particular place in your application.

Every good application with more than one screen should take advantage of a router to provide these expected pieces of the user experience. So, let's get started using `Backbone.Router`, right? Nah, I have a better idea: let's use `Marionette.AppRouter`.

The `AppRouter` is exactly the same as Backbone's `Router` — in fact it extends `Router` and you can just replace all instances of `Router` with `AppRouter` and your application will continue to work exactly the same way — except that it has an additional feature that allows you to use the router in a better way. In chapter 1, I mentioned that people often used their router as the central application namespace object, which is outside the responsibilities of a router. Well, apparently, the router comes under lots of misuse — though this time it isn't our fault.

Using A Controller

Backbone's `Router` was designed to handle the full processing of all of the routes within itself, but if you've ever looked at a router on the server, it is little more than configuration used to delegate actions to controllers based on the URL. `AppRouter` is set up more like a back-end router: it allows you to configure it with a controller, and then respond to a URL change by calling a method on the controller rather than on the router itself.

This keeps the concerns of determining a URL change and parsing it separate from controlling the application based on those changes. In other words: let the router route and the controller control.

Derick has an interesting example in a blog post[24] that demonstrates why he created the `AppRouter`. When he was using Backbone's `Router`, his code looked similar to this:

24. http://lostechies.com/derickbailey/2012/01/02/reducing-backbone-routers-to-nothing-more-than-configuration/

```
var MyRouter = Backbone.Router.extend({
    routes: {
        "": "main",
        "/category/:catid": "category",
        "/item/:itemid": "item"
    },

    main: function() {
        App.showMain();
    },
    category: function(catid) {
        App.showCategory(catid);
    },
    item: function(itemid) {
        App.showItem(itemid);
    }
});
```

This is not the same example he gave, but it's very similar – it's really just a simplified and generalized version of his code. With some inspiration from Marionette users' feedback, he decided that instead of just passing requests through to the application manually, he would create a means for himself to add the application as a controller that the router calls methods on directly. Thus `AppRouter` was born. And hence the name "AppRouter" instead of something like "ControllerRouter". Obviously, you don't need to use your application object as the controller, but if you're using the application object in the same way as Derick was, it's not a bad choice.

I prefer to use a separate object as the controller, but as long as you're removing the responsibility of controlling models and views from the router, you're taking good steps toward better structure.

So, how do we add that controller to an `AppRouter`? There are two ways to do it: when extending the `AppRouter`, or when instantiating one. No matter which method you choose to use, you do it by specifying a `controller` property that points to the object you want to use as the controller.

```
var MyRouter = Marionette.AppRouter.extend({
    controller: App, // when extending
    ...

});

router = new MyRouter({controller: App}); // when initializing
```

Note that no `AppRouter` can use more than one controller object.

Routing To The Controller

Of course, it doesn't do us any good to assign a controller to the router if we don't know how to make the router use the controller. Backbone's `Router` uses the `routes` property to specify the routes and map methods to them, which you can also do with `AppRouter`:

```
var MyRouter = Marionette.AppRouter({
    routes: {
        "": "index",
        ...

    },

    index: function() {
        ...

    },
    ...

});
```

This is why you can just replace your old routers with `AppRouter` and not have any changes. But if you want to map routes to methods on the controller, you use the `appRoutes` property:

```
var MyRouter = Marionette.AppRouter.extend({
    controller: App,
    appRoutes: {
        "": "index",
        ...

    },
    ...

});
```

The syntax is exactly the same. The only difference is that the method name refers to a method on the controller, rather than a method on the router. You can use both `routes` and `appRoutes` in the same router, so the routes specified in `routes` will use methods on the router, while the routes specified in `appRoutes` will use methods on the controller. It's important to note that if you have a route that appears in both `routes` and `appRoutes`, the `appRoutes` route will take precedence and the one in `routes` will never be called.

If you'd like to add routes dynamically after the router is initialized, you can use the `appRoute` method. It works in exactly the same way as the `route` method on Backbone's `Router`s, except of course that it adds a route that uses the method on the controller rather than on the router. When using `appRoute` (or just `route`), you need to pass in two arguments: the route as a string; and the name of the method on the controller that you want to call when the URL matches that route. This is also sent in a string.

```
router.appRoute('/list/', 'showList');
```

There are alternative syntaxes for `route` and `appRoute` where you can specify a callback function instead of using a method on the router or controller, but I won't show you those. The use cases are few and far between and aren't very good practice. If you want to learn more about that syntax, take a look at Backbone's documention for the `route` method[25].

Summary

If you're using a router in your web applications, then Marionette's `AppRouter` is a very good choice. Even if you're already using Backbone's `Router`, you can just switch them all up for `AppRouter` without any issues, and you'll be set for the future when or if your application grows to the point where you need better organization and structure. If you're just starting your application, there's no good reason not to use the `AppRouter` over `Router`.

It may only be a small step up from the `Router`, but it can have a noticeable impact on the way you write your code and keep your routers clean. ❧

25. http://backbonejs.org/#Router-route

Chapter 10: Controller

After talking about routers and assigning controllers to them, I bet you're thinking that this chapter is about a Marionette component that is designed to be used as a controller for a router. Well, you'd be partially right. Let me explain by quoting the documentation:

> *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.*

It was intended to be a general-purpose controller to be used in your routers, but Derick's favorite use is as a module controller. For each module he creates, he uses a `Controller` essentially as the API for that module. That's a good use for it, but as you'll see throughout this chapter, `Controller` is a lot more versatile than that.

Extensibility

Once again, we are given a class with an `extend` function to easily allow us to extend it with subclasses. You can use it just like you would any other extendable Backbone or Marionette class.

It also allows us to include an `initialize` method that is called in the constructor, so we can add our own initialization logic as usual. Finally, it takes the first parameter passed into the constructor and saves it as `this.options`. This way you don't need to manually store individual options on your object. You can just access them through the `options` property.

```
var ctrl = Marionette.Controller.extend({
    initialize: function(options) {
        // options === this.options
    },
    ...

});
```

This functionality of assigning options to the object is common in Marionette's view classes, too, in case you weren't aware. Backbone used to do this in `View`, but it has removed that functionality. Marionette threw it back in there for anyone who used it so that nothing would break.

Events

Another mainstay in the Backbone world that is available on `Controller` is events. You get all the standard event methods such as `on`, `off`, `listenTo`, and `stopListening`. In addition to these methods provided by `Backbone.Events`, `triggerMethod` is also added. If you don't know what I'm talking about here, make sure to read the "Events" section of chapter 8.

Finally, a `close` method is provided for us that automatically calls `stopListening` to stop listening for any events it may have subscribed to, triggers a close event (via `triggerMethod`), and then unbinds all the listeners that have subscribed to it.

Use For Anything

That's everything there is to `Controller`. It doesn't have much, but it has the essential functionality that is found just about everywhere in Backbone and Marionette, so it is very versatile. Just extend it and you're on your way to creating a new component that can be used in many Backbone applications.

Because of this versatility, the name for the class has been somewhat controversial, but since it was intended to be used as a controller or mediator, it was branded "Controller" and it has stuck. I'd argue that the name could be "Base", but that implies that it could be the base for all other classes in Marionette (which it certainly is not) and, once again, intended use gets in the way, regardless of the possibilities for it.

A Couple Examples

`Controller` has so many uses that I could never show you all of them, but I'll demonstrate a couple ways it is often used. First, we'll see it being used as a controller for an `AppRouter`.

```
var Router = Marionette.AppRouter.extend({
    appRoutes: {
        "": "showIndex",
        "categories": "showCategories",
        "items": "showAllItems"
    }
});

var Controller = Marionette.Controller.extend({
    showIndex: function() {
        // Show the index
    },
    showCategories: function() {
        // Show the categories
    },
    showAllItems: function() {
        // Show the items
    },
});

var controller = new Controller();
var router = new Router({controller:controller});
```

That was a simple but very common example. The next example will be a little more complex. We're going to use the controller to handle the public API of a module using the commands system.

```
App.module('SomeModule', function(SomeModule, App) {
    SomeModule.Controller = Marionette.Controller.extend({
        doSomething: function() {
            // do something ...

        },
        doSomethingElse: function() {
            // do something else ...

        }
    });

    SomeModule.addInitializer(function() {
        SomeModule.controller = new SomeModule.Controller();

        App.commands.setHandler('do:something',
        controller.doSomething, controller);
        App.commands.setHandler('do:something:else',
        controller.doSomethingElse, controller);
    });
});
```

This is another common use for `Controller` that Derick Bailey likes. The interesting thing to note, though, is that we didn't actually use `Controller` for anything that couldn't be done with a simple object literal. In both cases, though, there are likely cases where you'll at least want the `Controller` to listen for events or even trigger events. This can't be done cleanly with an object literal, but `Controller` provides all the necessary methods for using events and cleaning them up.

You can also use it as the base for any utility class that needs events or the simple `extend` functionality, and match the way that the rest of Backbone's and Marionette's components work.

Summary

Not much to summarize here. `Controller` provides all the common basic functionality found in most Backbone and Marionette classes, so you can get started creating your own objects. It's obviously useful as intended – as a controller for modules, routers, and so on – but go ahead and take advantage of its versatility. ❧

Chapter 11: Introducing Our Example Application

Well, it was great teaching you guys all there is to know about Marionette. I hope you enjoyed it. See you later!

Just kidding. What kind of library or framework discussions come without a sample application? Sure, you now know about all of Marionette's components, but you haven't seen them working together. To really understand how Marionette can work for you, you need to understand how its pieces fit together to form a cohesive framework. So, I've created a sample application that demonstrates how most of the Marionette components could be used. Sadly, since I was attempting to make the application small and simple enough that there wouldn't be too much application logic to distract you from the lessons about using Marionette, I couldn't find a way to squeeze every single component into the application. Both `CollectionView` and `Request/Response` didn't make the cut, though I feel the application is sufficiently large and complex to demonstrate how Marionette should be used.

Derick Bailey created an example application to show how he believes Marionette should be used, and while I think it is a good application for this sort of demonstration, it also misses a few components — and I feel like it's cheating if I steal his application for this book. His application is BBCloneMail[26], which is a simple fake email client (it can't really be used to send and receive emails) that mimics some of GMail's functionality. His application leaves out `CompositeView`, `AppRouter`, and `Layout`. The reason he doesn't use `AppRouter` is because he used a filtered router (which can run a special method before or after each route method). The plugin he used (Backbone RouteFilter[27]) can now be used to enhance `AppRouter` instead of just Backbone's `Router`, but his application hasn't been updated in a while so it doesn't reflect this change. It would be nice to have this functionality built into `AppRouter`, but as long as the plugin exists elsewhere, there is no incentive to add it to Marionette.

Switching our focus back to the example application for this book, we'll be building a quiz engine. And we'll name our wonderful little application, appropriately enough, Quiz Engine. Why make it confusing, right?

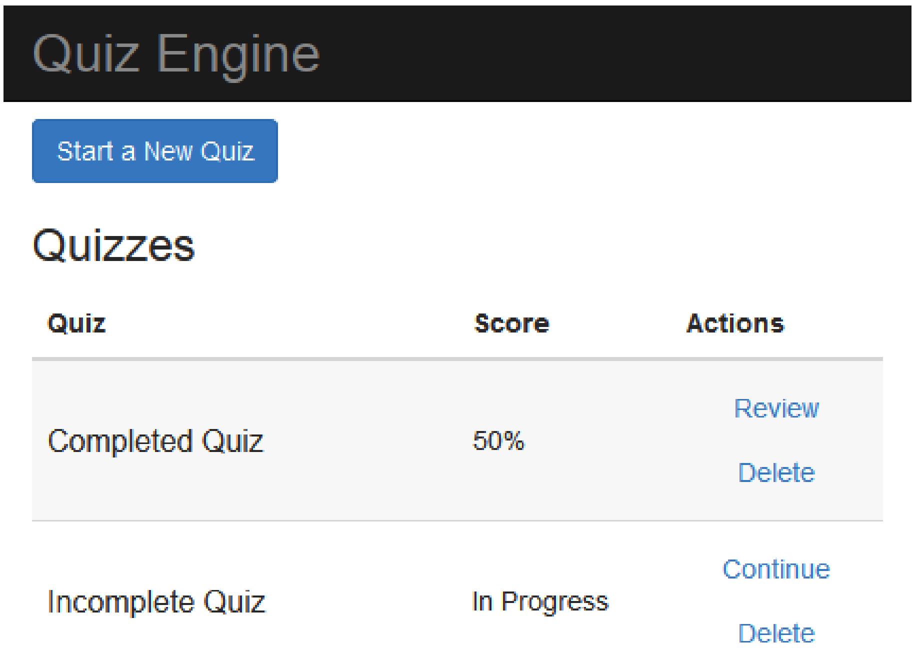
26. https://github.com/marionettejs/bbclonemail/
27. https://github.com/boazsender/backbone.routefilter

You can try out the application on my site[28] or you can get it off of GitHub[29] and run it on your own server. We'll discuss how you can get it running on your own server later.

Before we get into the nuts and bolts of the application in the next few chapters, we'll take a little tour. We'll talk about what the application does and looks like from the perspective of a user.

Quiz List

The first thing you should see is a list of the quizzes you've taken.


Quiz Engine

Start a New Quiz

Quizzes

Quiz	Score	Actions
Completed Quiz	50%	Review Delete
Incomplete Quiz	In Progress	Continue Delete

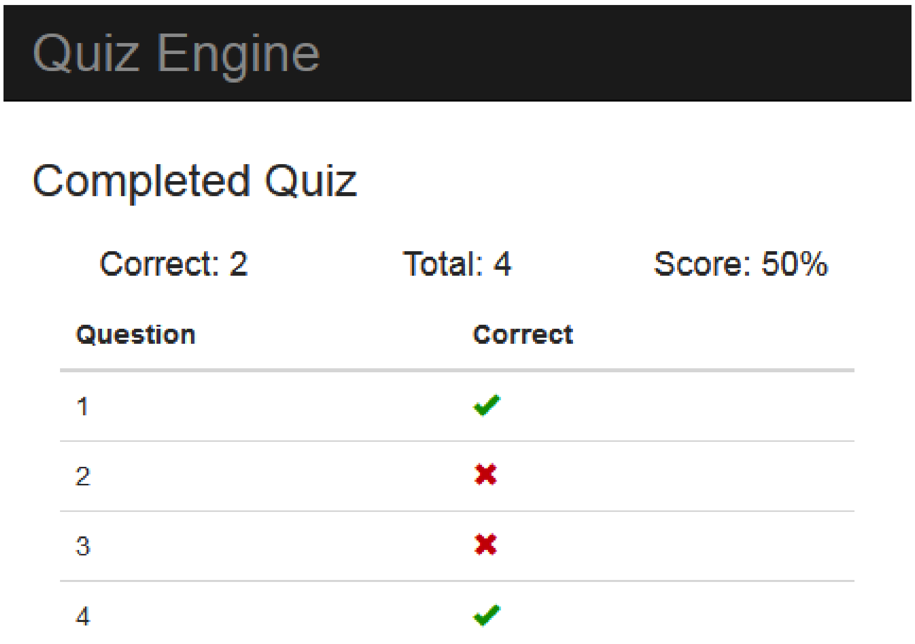
Home Screen: List of Quizzes You've Taken

From here, you can start a new quiz, delete quizzes, review the answers of quizzes that have been completed, or continue a quiz that you haven't finished yet.

So, what happens when we click one of those "Delete" links? Well, it simply removes that test from the list. No big deal. But instead of deleting one of those quizzes right now, let's click the "Review" link for the completed quiz and see what we get.

28. http://www.joezimjs.com/_quizengine/
29. https://github.com/joezimjs/Quiz-Engine

Quiz Review



Completed Quiz

Correct: 2 Total: 4 Score: 50%

Question	Correct
1	✓
2	✗
3	✗
4	✓

Quiz Review Screen

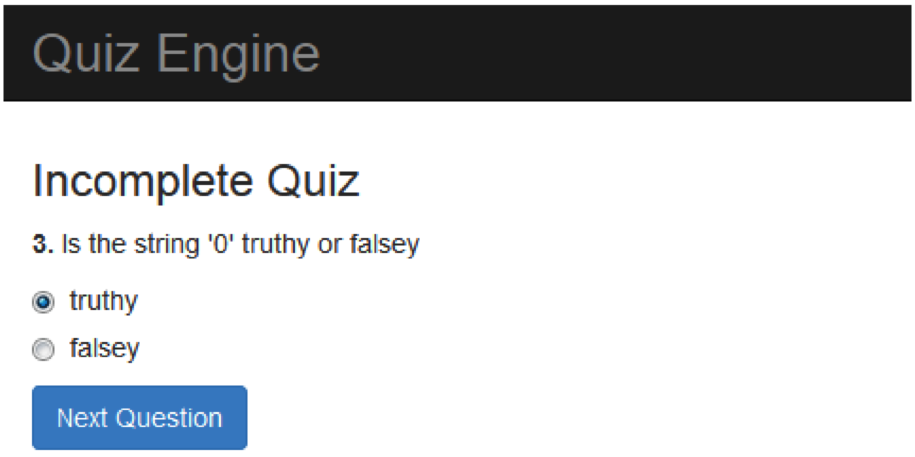
Here we see a few details about the quiz. We have its title, the number of correct questions, the total number of questions on the quiz, the score of the quiz, and then a table showing which of the questions were right and which ones were wrong. Of course, the ever present header is also there so we can get back to the main page, which is good — I was too lazy to add a button that would bring us back there!

You could take this review page one step further and show the user details about each question, such as the question text, the answer they chose, and the correct answer, but I figured I'd keep it simple. This is one of the places where it doesn't pay to add features because I'm just demonstrating how to use Marionette, not creating an application for the masses.

Anyway, head back to the main listing page, and let's click the "Continue" link for the other quiz.

Taking a Quiz

Whether you're continuing a quiz you already started or you're starting a new quiz, the image below is what you'll see. After the quiz name, we see the question number, the question text, and a list of possible answers. Users can select their answer and click the "Next Question" button. If they're on the final question of the quiz, then the button will say "Finish Quiz" instead and it'll take them to the review page.


Quiz Engine

Incomplete Quiz

3. Is the string '0' truthy or falsey

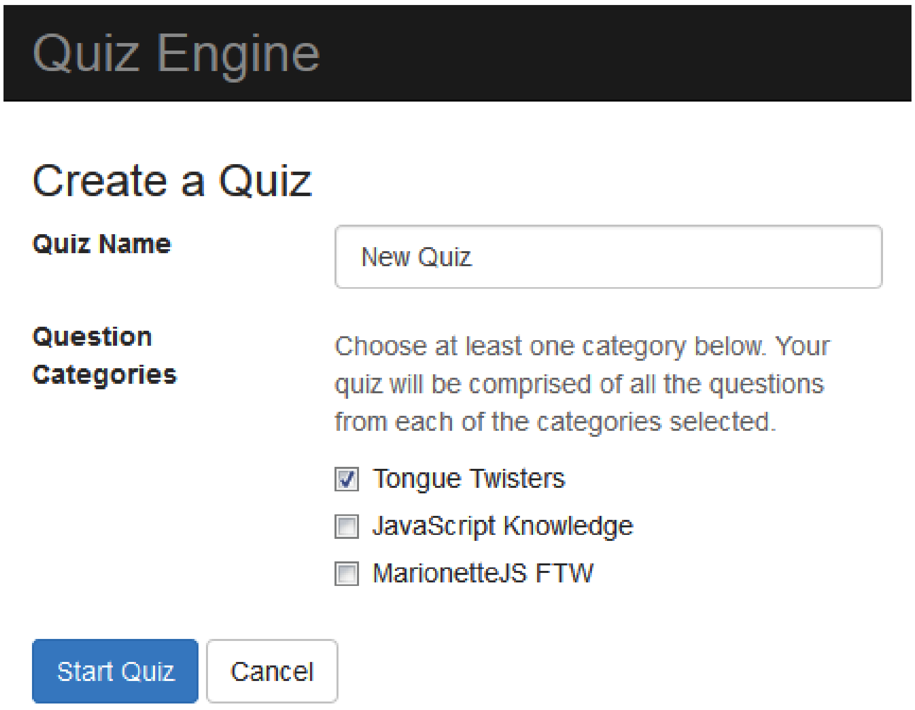
[✓ truthy]

falsey

Answering a Question on a Quiz

Let's see what it looks like to create a new quiz. Head on back to the main screen and click the "Start New Quiz" button.

Creating a Quiz


Quiz Engine

Create a Quiz

Quiz Name: New Quiz

Question Categories

Choose at least one category below. Your quiz will be comprised of all the questions from each of the categories selected.

- ✓ Tongue Twisters
- JavaScript Knowledge
- MarionetteJS FTW

Creating a Quiz

When creating a quiz, you give it a name and choose the categories of questions you'd like to add to the quiz, and click "Start Quiz". After that you get taken to the screen that we looked at just before this one. If you decided that none of the categories interest you, you can always click

“Cancel” and get taken back to the main screen and take a gander at the quizzes you’ve already completed.

Summary

That’s all the major screens that you’ll see, though there are a few bits that we missed: error handling, empty lists and such. We’ll take a look at those in some of the later chapters. For now we’re done with our introduction and we’ll move on to some of the decisions that we need to make before putting our fingers to the keyboard. ❧

Chapter 12: Decisions and Setup

Before you write any application, you need to take some time to sit down, organize your thoughts, and make a few decisions — or, in this example, you need to understand the decisions I made. First and foremost, as I mentioned in the previous chapter, Marionette should be in the forefront of this application, so we need to keep the domain logic and other features simple. To this end, I chose not to have any form of persistence. Data is stored solely in memory. It will also be generated by some static JSON when the application is started up, so we at least have *some* data each time the browser is refreshed. This sidesteps the need to set up a back end and also allows us to skip using a solution for client-side storage.

We'll also keep things simple by using Underscore templates and sticking the templates inside `<script>` tags in the HTML file. That way, we can simply provide an ID for the `<script>` tag holding the template we need, instead of having to worry about loading in additional files or anything else.

In addition, to help make development simpler and quicker, I've decided to use Twitter Bootstrap 3. This allows me to easily make the pages responsive as well, as you can see in the screenshots, which show the app at only 480 pixels wide. This may make the HTML look a bit convoluted at times, but for the most part you can ignore the HTML and just focus on the JavaScript bits.

Since Bootstrap is designed for more modern browsers (it doesn't have support for older versions of Internet Explorer), I've also decided to use jQuery 2.0.3, which also eliminates support for older versions of IE. I figured that web developers tend to have more modern browsers, so leaving out these older browsers would be fine for this application.

I also decided not to use some of the libraries that I almost always use, such as RequireJS[30]. Normally, I wouldn't dream of skipping out on RequireJS — which would change the way I do numerous things, such as making some classes more decoupled, and not storing classes within a namespace since they can just be retrieved with `require` directly — but once again, this is about keeping things simple.

Finally, some conventions should be agreed on:

- Capitalize module names.

30. http://requirejs.org

- Capitalize file names for files that only contain a class definition. Other file names will be all lowercase letters.
- Keep all classes related to a module namespaced on that module. If I were using RequireJS, I wouldn't be namespacing classes.
- All modules have an *index.js* file that handles the core bits and external communication for the module.

Setting Up

Now that we've come to some decisions, let's get started. First let's set up some folders. Of course, we need to start with our root folder where the whole application will be held. Call this whatever you like, but it should have something to do with the application name: Quiz Engine. Inside that folder we'll add three folders: *css*, *js*, and *fonts*. That will be good enough for now. It should look something like this:

```
□ quizengine
    □ css
    □ fonts
    □ js
```

Now let's add a few subfolders into that *js* folder. First off, we'll add a *vendor* folder. This is where we'll put all the third-party scripts and libraries. Next we'll add a *modules* folder in the *js* folder. This application will be divided into a few subapplications, and since Marionette uses `module` for creating these subapps, we'll call this folder *modules* instead of *subapps* or something similar. Finally, we'll throw in a *helpers* folder. You could also call this something like *utils* or *extensions* or whatever takes your fancy, but this is where anything that offers some type of utility functionality will go, especially if it can't be classified as anything else we've come across. The overall folder structure should look like this:

```
□ quizengine
    □ css
    □ fonts
    □ js
        □ helpers
        □ modules
        □ vendor
```

Now that we've got all our folders set up (or if you've downloaded or cloned the GitHub repository), we need to make sure that the root folder is accessible to your web server. We don't need anything special — any standard web server that can deliver files will work for this application since there is no back end. For a quick and simple server, I like to use Polpetta[31], which allows me to use the command line to quickly turn any folder into the root of a website accessible at *http://localhost:1337*. Go ahead and use whatever you'd like. If you're downloading the code from GitHub, then this is all you need to do to get the code working.

Now, in order to start writing our own code, we'll need to first get our dependencies. Download each of these dependencies:

- **Twitter Bootstrap**[32]:
  Go to the Bootstrap site and click "Download Bootstrap" on the homepage. Unzip the file, go into the *dist* folder, and copy the *bootstrap.min.css* file (you could use the non-minified version if you prefer) in the *css* folder directly to our *css* folder and copy all the contents of the *fonts* folder directly to our *fonts* folder. It also comes with some JavaScript, but we won't be using any of it.

- **jQuery**[33]:
  Go to the jQuery site, click "Download jQuery", and click "Download the compressed, production jQuery 2.0.3" under the "jQuery 2.x" section and save it into the *js/vendor* folder.

- **Underscore**[34]:
  Go to the Underscore site, click "Production Version (1.5.2)", and save the file in the same *vendor* folder as jQuery.

- **Backbone**[35]:
  Go to the Backbone site, click "Production Version (1.1.0)". Save this file in the *vendor* folder as well.

- **Marionette**[36]:
  Go to the Marionette site, scroll down to the "Downloads and Documentation" → "Bundled" section and click "backbone.marionette.min.js". Save this file in the same *vendor* folder as the other JavaScript files we've downloaded. This bundled version also includes Wreqr and Babysitter, so we don't need to download them separately.

---

31. https://github.com/WebReflection/polpetta
32. http://getbootstrap.com
33. http://www.jquery.com
34. http://underscorejs.org/
35. http://backbonejs.org/
36. http://marionettejs.com/

Now that we have all of our libraries and frameworks downloaded, we'll need to get our HTML file set up. Create an *index.html* file in the root application folder and add these contents:

index.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
    content="width=device-width,initial-scale=1"/>
    <title>Quiz Engine: A MarionetteJS Example Application</title>
    <link rel="stylesheet" href="css/bootstrap.min.css">
    <link rel="stylesheet" href="css/styles.css">
</head>
<body>
    <div class="container">
        <header data-region="header" class="navbar navbar-inverse
        navbar-fixed-top">
            <div class="container">
                <div class="navbar-header">
                    <a href="#list" class="navbar-brand">Quiz
                    Engine</a>
                </div>
            </div>
        </header>
        <section id="appBody" data-region="body"
class="row"></section>
        <footer data-region="footer" class="row">
            <p class="col-xs-12 text-muted">Copyright &copy; 2013
            Joe Zimmerman. This is an example application used in
            the book, <em>Better Backbone Applications with
            MarionetteJS</em>.</p>
        </footer>
    </div>

<!-- Templates -->

<!-- Dependencies -->
    <script src="js/vendor/jquery.min.js"></script>
    <script src="js/vendor/underscore.min.js"></script>
    <script src="js/vendor/backbone.min.js"></script>
    <script src="js/vendor/backbone.marionette.min.js"></script>
<!-- Our Code -->
</body>
</html>
```

What do we have here? Well there's the obvious stuff, like a `<doctype>` element and such. We also have the `<meta>` elements for the character set and viewport. The viewport `<meta>` element is important for responsive websites, because without it many mobile browsers will initially be zoomed out. Moving past that, we come to our style sheets. We obviously needed to include the Twitter Bootstrap style sheet, but we also have another style sheet there that we'll get to in a moment.

Once we get into the `<body>`, we see the structural HTML for the page with a header, footer, and body. The header and footer should remain unchanged by the application, but the `[data-region="body"]` `<section>` element is where all of our views will sit.

After that, I have a comment denoting where our templates will be. And moving past that, at the end of the `<body>` element is where we'll place our myriad of scripts. I've already put in the dependencies, and just below those is where we'll start sticking our scripts.

Before we get to writing our JavaScript code, though, let's create that *styles.css* file and place it in the *css* folder:

css/styles.css

```
#appBody {
    margin-top: 60px;
    margin-bottom: 30px;
}

.table > thead > tr > th,
.table > tbody > tr > th,
.table > tfoot > tr > th,
.table > thead > tr > td,
.table > tbody > tr > td,
.table > tfoot > tr > td {
    vertical-align: middle;
}

.green {
    color: #090;
}

.red {
    color: #c00;
}

.navbar-brand {
    font-size: 28px;
}
```

Nothing special here. Just some minor styles and tweaks to Bootstrap's styles.

Summary

Well, that's the bulk of the high-level decisions we need to make. Most of the decisions are fairly straightforward, especially with an application of this size. Now that we have our setup done, though, I'm sure you are keen to get into writing the JavaScript, so let's move on to the next chapter. ❧

Chapter 13: Building a Foundation

Before we get into the meat and potatoes of the application, we should start with the basic structural and foundational pieces so the rest of our code has something to lean on. We'll start off by creating our `Application` object.

Application

Create a file called *QuizEngine.js* and place it in the *js* folder. It'll contain the following code:

js/QuizEngine.js

```
var QuizEngine = (function(){

    var Application = Marionette.Application.extend({});

    var application = new Application();

    application.addRegions({
        header: '[data-region=header]', // Not used right now
        body: '[data-region=body]',
        footer: '[data-region=footer]' // Not used right now
    });

    application.on('initialize:after', function() {
        Backbone.history.start();
    });

    return application;
})();
```

We'll use the module pattern to assign an instantiated `Application` object to the global `QuizEngine` object. Normally, I wouldn't capitalize an instantiated object, but for the `Application` object, I tend to make an exception, since it is such an important and central piece.

We use `Application.extend` to create our own class, but I didn't actually put anything inside. This may seem pretty pointless, and for the most part it is, but if anyone ever comes back to this app and wants to add functionality, it's already ready and waiting for that developer.

It's small and simple, so I don't feel like it's much wasted effort to do this sort of forward thinking.

Then, after instantiating our not-so-custom `Application`, we add some regions to it. Once again, we do a bit of forward thinking and add the header and footer as regions even though we never touch them. Of course, we need to add the body region, so we could actually show something to the users.

Finally, before we return our `Application`, we throw in a little event listener. After the app is completely initialized, we'll start up `Backbone.history` so that our routers will work. This tidbit of code could also very easily have been put with the code where we define the main router, but it works just fine here, too.

Router

Speaking of the router, let's set that up next. In the *js* folder, let's create a file named *Router.js* with the following contents:

js/Router.js

```
QuizEngine.Router = Marionette.AppRouter.extend({
    routes: {
        "": "redirectToMain"
    },

    redirectToMain: function() {
        Backbone.history.navigate('list', { trigger: true,
        replace: true});
    }
});

QuizEngine.addInitializer(function() {
    QuizEngine.router = new QuizEngine.Router();
});
```

This router may not be what you were expecting. Let's walk through it and clear up any confusion you may have. First we're creating an extension of `AppRouter` and assigning it to `QuizEngine.Router`. Here we're using `QuizEngine` as a namespace for storing our router class. There are really only three options for storing classes like this:

1. Use a namespace, such as `QuizEngine`.

2. Make it global, which just isn't really an option as storing things in the global namespace is bad form.

3. Use a module loader, such as RequireJS, to load it in without storing it anywhere. Of course, for simplicity's sake, we've already eliminated this option.

Moving on: we extended `AppRouter` even though we didn't make use of `appRoutes`. I mentioned in chapter 9 where we discussed `AppRouter` that we could do this, and I've chosen to do so here. No reason — other than negligible performance gains — to keep track of two different kinds of router when we can just use the one that is simply an enhanced version of the other.

Now we get to the part you may be really confused about. Where are all the routes? Well, most of the routes are going to be split into separate routers, so that each module can control the routes that relate to them. The only route for this application that doesn't pertain to a specific module is the empty/root route.

Since the default screen shows the quizzes list, and that module is using its own router, I decided to redirect to `'list'` instead of using the root URL for that screen. To do a redirect with Backbone's routers, you use `navigate` on either the router or `Backbone.history`, which is what I chose to do. I prefer not to use the routers for this, because it honestly shouldn't be the router's job — they are meant to react to changes in the URL, not cause them. Anyway, with a redirect, we need it to trigger a route, so we need to set `trigger` to `true`. We also use `replace: true` to prevent adding an extra entry in the history, so that people won't hit the Back button and immediately get redirected to `#list` again.

Finally, we use `addInitializer` on `QuizEngine`, so that we can initialize the router when the application starts up. We also could have moved this initialization to a separate file that contains bootstrapping code, but that's a personal decision.

Subapplication Manager

Moving beyond the router, we come to the final piece of our application that is used for setting up the initial structure: a subapplication manager. We will use this object to start and stop subapplications; in particular, we will ensure that when we start a subapp no other subapps are running at the same time, by stopping the one that was currently running.

Create a filed name *SubAppManager.js* in the *js/helpers* folder and fill with these contents:

js/helpers/SubAppManager.js

```
QuizEngine.Helpers = QuizEngine.Helpers || {};
QuizEngine.Helpers.SubAppManager = (function() {

    var SubAppManager = Marionette.Controller.extend({
        startSubApp: function(name, args) {
            var newApp = QuizEngine.module(name);

            if (this.currentApp === newApp) {
                return;
            }

            if (this.currentApp) {
                this.currentApp.stop();
                QuizEngine.vent.trigger('subapp:stopped',
                this.currentAppName);
            }

            this.currentApp = newApp;
            this.currentAppName = name;

            newApp.start(args);
            QuizEngine.vent.trigger('subapp:started', name);
        }
    });

    var manager = new SubAppManager();
    QuizEngine.commands.setHandler('subapp:start',
    manager.startSubApp, manager);
    return manager;

})();
```

We start by adding the `Helpers` namespace to `QuizEngine` so we have somewhere to stick this `SubAppManager`. The `SubAppManager` is defined inside of another instance of the module pattern. We create the `SubAppManager` by extending Marionette's `Controller`, more to demonstrate that it can be used for just about anything than for any other good reason. If you'd like, you can go ahead and just rewrite this with an object literal instead.

We're giving the `SubAppManager` just a single method: `startSubApp`. It takes two arguments: the name of the subapp (which corresponds to the name the module was created with); and any arguments that you want passed into the module's `start` function.

The first thing this method does is check to see if the subapp that is trying to be started is one that is already running; if it is, then don't bother doing anything else, just return. If we're trying to start up a new subapp, we first close down the currently running subapp, assuming there are any running, and fire off an application-wide event. This event won't be used in this Quiz Engine application, but it's an excellent example of the sort of event that could come in handy somewhere else in the application.

After that, we store the current subapp so we can check it the next time `startSubApp` is called. Finally, we start the specified subapplication and fire off another event, informing the entire app that a new subapp has been started.

After we finish defining `SubAppManager`, we instantiate it. Then we register `startSubApp` with Marionette's command system on `QuizEngine`, so we have a clean and easy way to call it. Of course, we could have just let the rest of the system do this by calling `QuizEngine.Helpers.SubAppManager.startSubApp('app')` but that isn't as manageable as `QuizEngine.commands.execute('subapp:start', 'app')`. When using the command system, you make your interface simpler and it's easier to swap out or change the `SubAppManager` later on.

Now, after all of that, we finally return the manager so it can be assigned to its rightful place. The interesting thing to note here is that we don't even need to save `SubAppManager` anywhere, since a reference to it will always exist because of passing it in to the call to `QuizEngine.commands.setHandler`. Considering it is only used for the method called through the command, this would not be a bad way to handle it, but I tend to use more object-oriented programming than functional programming, so I like to keep my objects around. This file could be completely rewritten like this:

```
(function() {
    var startSubApp = function(name, args) {
        var newApp = QuizEngine.module(name);

        if (this.currentApp === newApp) {
            return;
        }

        if (this.currentApp) {
            this.currentApp.stop();
            QuizEngine.vent.trigger('subapp:stopped',
            this.currentAppName);
        }

        this.currentApp = newApp;
        this.currentAppName = name;

        newApp.start(args);
        QuizEngine.vent.trigger('subapp:started', name);
    };

    QuizEngine.commands.setHandler('subapp:start', startSubApp,
{});
})();
```

Now it is just a function, and a call to register it as a command, all wrapped inside an IIFE (immediately invoked function expression) to protect the variables. Once again, JavaScript shows that it can be used in many different ways all of which are correct, and it is largely a matter of preference.

Now that we have all of that done, we need to add these files to the HTML file. Just below the "Our Code" comment, let's add this bit of HTML:

index.html

```
<script src="js/QuizEngine.js"></script>
<script src="js/helpers/SubAppManager.js"></script>
<script src="js/Router.js"></script>
```

Summary

That wraps up the core structural parts of our application. We now have our application, subapplication manager, and core router all set up. Let's move on to coding our data structure. ❧

Chapter 14: Managing Data

Now that we have the foundation laid, we need to create some means of handling data, which will naturally be handled by Backbone's `Model` and `Collection` classes. We'll keep all the data in its own separate module. This isn't necessary, but I like how it makes initialization work. You could also just place all of the data in a namespace on `QuizEngine` instead of bloating it with all of the features that come with a Marionette `Module` that won't be used. I've heard of some developers putting all of their data in an `Entities` namespace, which is probably a good name for it, but I'm sticking with the simple "Data" name as a module.

Module Index

Since this is going to be a module, we'll first add a folder for it in the *js/modules* folder, named *Data*. I mentioned in the tiny section about conventions that each module will have an *index.js* file where we handle the core functionality of the module, and this module is no different. Create an *index.js* file in the *js/modules/Data* folder with the following contents:

js/modules/Data/index.js

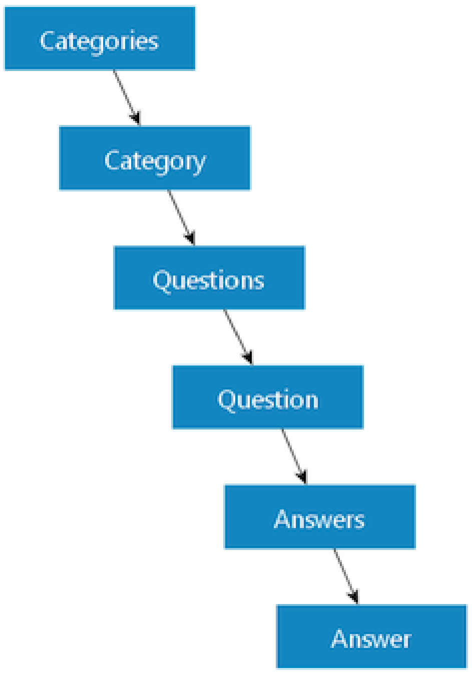
```
QuizEngine.module('Data', function(Data) {

    Data.addInitializer(function(options) {
        // Grab the data from the option passed in to the App's/
        Module's start method
        Data.questions = new Data.Categories((options &&
        options.data && options.data.questions) || []);
        Data.quizzes = new Data.Quizzes((options && options.data
        && options.data.quizzes) || []);
    });

    Data.addFinalizer(function() {
        delete Data.questions;
        delete Data.quizzes;
    });

});
```

The first thing we do is declare the Data module. Then, inside the module definition function we add an initializer and a finalizer. Inside the initializer, we initialize our data (the `Categories` and `Quizzes` classes will be shown soon) using the data that was given to the module. In our application, these options will be passed through the call to `QuizEngine.start`, but they can also be passed in with a call to `QuizEngine.module('Data').start`. Anyway, if we didn't receive any information, an empty array will be used by default. In the finalizer, we clear up some memory by deleting the references to the data, though I don't see this module ever being stopped.

If you really want to keep things decoupled, you could use the Request/Response or the commands system as the API for the data, but we'll forego that in this application.

Questions

Let's discuss how we're handling the data for the questions. Each question belongs to a category and has one or more answers. We will end up needing to define a total of three different models and three different collections to do this. We need a `Category` model and a `Categories` collection for storing them. Each `Category` can have any number of questions, so they will need to contain a `Questions` collection, which can hold `Question` models. Each `Question` has some answers, so they will contain an `Answers` collection to hold each `Answer` model.



Question Data Structure

Owing to this hierarchical structure where models can contain collections, we'll need to override the `toJSON` methods for those models to make sure we actually get JSON instead of a reference to the collection. We'll also need to use the constructors to ensure that the data is indeed a collection, rather than just an array.

Before we write any code for these, I want to create a *js/modules/Data/models* folder to place all these files in. We'll also start by just specifying the barebones models and collections. We'll skip the application-specific API for now.

js/modules/Data/models/Categories.js

```
QuizEngine.module('Data', function(Data) {

    Data.Categories = Backbone.Collection.extend({
        model: Data.Category
    });

});
```

Here we're just creating a `Collection` that uses `Category` as the type of `Model` it instantiates. The class is stored on the module for easy access.

js/modules/Data/models/Category.js

```
QuizEngine.module('Data', function(Data) {

    Data.Category = Backbone.Model.extend({
        defaults: {
            name: "",
            questions: null
        },

        initialize: function() {
            var questions = this.get('questions');
            this.set('questions', new Data.Questions(questions));
        },

        // Custom toJSON to also JSONify 'questions'
        toJSON: function() {
            var data = Backbone.Model.prototype.toJSON.call(this);
            if (data.questions && data.questions.toJSON) {
                data.questions = data.questions.toJSON();
            }
            if (!data.id) {
                data.id = this.cid;
            }

            return data;
        }
    });

});
```

You shouldn't see anything new here if you've already read chapter 5 where I briefly discuss hierarchical data sets. The only real difference is that we're not using a recursive data type: nodes, which have nodes, which have nodes, and so on. Instead, it's just categories, which have questions, which have answers. There is another minor difference: I add the `cid` (an auto-generated unique ID for all models) as the `id` property in the JSON. This gives an identifier to use in the view without us needing to specify one.

It may be nice to know that the models are rather simple, besides their hierarchical nature: a category only has a name and a list of questions.

js/modules/Data/models/Questions.js

```
QuizEngine.module('Data', function(Data) {

    Data.Questions = Backbone.Collection.extend({
        model: Data.Question
    });

});
```

This is another basic `Collection` stuck onto the module.

js/modules/Data/models/Question.js

```
QuizEngine.module('Data', function(Data) {

    Data.Question = Backbone.Model.extend({
        defaults: {
            id: null,
            text: "",
            correctAnswer: 0,
            answers: []
        },
        initialize: function() {
            var answers = this.get('answers');
            this.set('answers', new Data.Answers(answers));
        },

        // Custom toJSON
        toJSON: function() {
            var data = Backbone.Model.prototype.toJSON.call(this);
            if (data.answers && data.answers.toJSON) {
                data.answers = data.answers.toJSON();
            }

            if (!data.id) {
                data.id = this.cid;
            }

            return data;
        }
    });

});
```

The `Question` model has a few more bits of data that it stores compared to the other models. Each `Question` has an ID, the question text, a list of possible answers, and a number that specifies the index of the correct answer. Of course, the list of answers is a `Collection`, so we make sure that is taken care of, just like `Category`.

Questions have an `id` so that quizzes can refer to the question without worrying about the `cid` not remaining constant between refreshes. Otherwise `cid` is perfectly fine to use throughout the rest of the application, especially when there's no persistence. If `id` is falsey, I still use the `cid` for the JSON's `id`, just in case it gets changed.

js/modules/Data/models/Answers.js

```
QuizEngine.module('Data', function(Data) {

    Data.Answers = Backbone.Collection.extend({
        model: Data.Answer
    });

});
```

One more basic `Collection` specific to `Answer` models.

js/modules/Data/models/Answer.js

```
QuizEngine.module('Data', function(Data) {

    Data.Answer = Backbone.Model.extend({
        defaults: {
            text: ""
        }
    });

});
```

Finally, we come to the `Answer` model. All that answers contain is the text displayed to the user. That's all of the files related to the questions, but this is just the basic functionality. Let's take a look at the application-specific logic that I've included in these files.

js/modules/Data/models/Categories.js

```
// ... inside module
Data.Categories = Backbone.Collection.extend({
    // ...

    getQuestion: function(id) {
        var question = null;

        this.each(function(category) {
            var tempQuestion = category.getQuestion(id);

            if (tempQuestion) {
                question = tempQuestion;
                return false;
            }
        });

        return question;
    },

    getQuestionsByCategories: function(categories) {
        var self = this;

        if (_.isArray(categories)) {
            var questions = [];

            _.each(categories, function(categoryId){

questions.push(self.get(categoryId).getQuestions());
            });

            return _.flatten(questions);
        }

        if (_.isNumber(categories) || _.isString(categories)) {
            return this.get(categories).get('questions').models;
        }

        return null;
    }
});
```

There are two methods added here. `Categories` will end up being a bit of a mediator to the rest of the hierarchy of data. Both methods are used to retrieve questions, and you'll see where and how they're used when we get to those parts of the application.

The first method, `getQuestion`, retrieves a single question by the provided ID. It uses a `getQuestion` method on each of the `Category` models to help with the searching. The second method, `getQuestions-ByCategories`, returns an array of all the questions contained in the specified categories. You can either provide a single category ID, or an array of category IDs to this method (an ID can also refer to `cid`).

js/modules/Data/models/Category.js

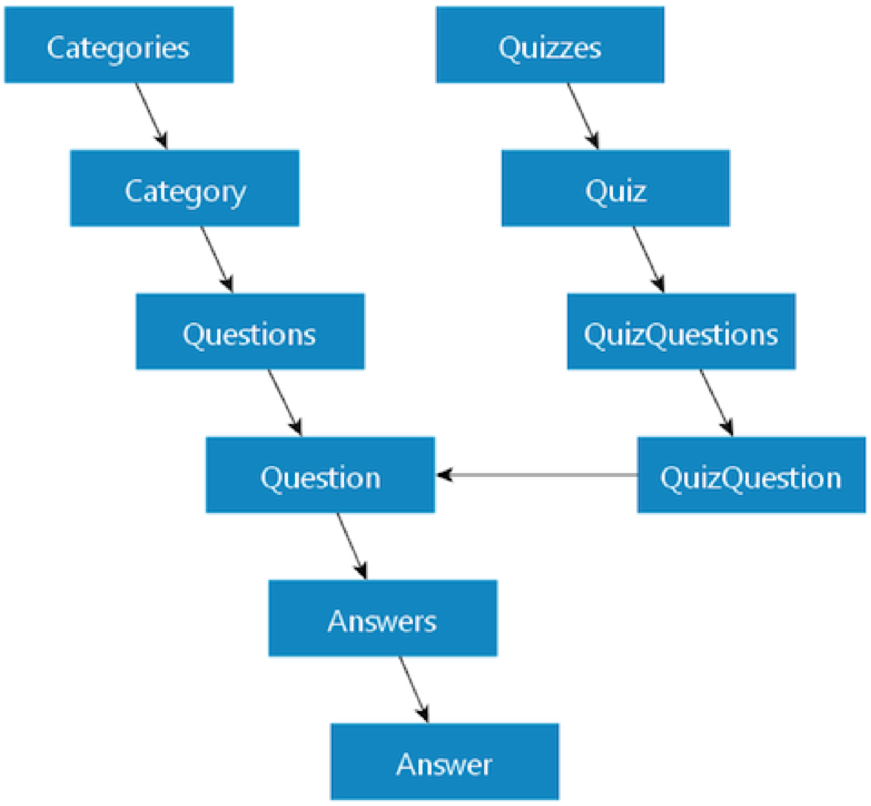
```
// ... inside module
Data.Category = Backbone.Model.extend({
    // ...

    getQuestion: function(id) {
        return this.get('questions').get(id);
    },

    getQuestions: function() {
        return this.get('questions').models;
    },
    // ...
});
```

Here are a couple simple methods. We either get a single question by its ID, or we get all of the questions for the category. Both of these methods are used in `Categories.js`.

That's all the spare methods that weren't mentioned earlier, so let's move on to quizzes.

Quizzes

Quizzes will have a similar structure to the questions hierarchy, but have fewer layers (or more, depending on how you look at it). We have a `Quizzes` collection that holds `Quiz` models. These have a `QuizQuestions` collection to hold `QuizQuestion` models. Each `QuizQuestion` then has a reference to a `Question` rather than copying data from the `Question`. Below is an image representation.



Question Data Structure

Creating the basic hierarchical structure will also be very similar to how we did it with the question data. This time, instead of showing you the basic functionality and then adding the special functionality afterward, I'll just dump it all on you at once. These files also all are in the *js/modules/Data/models* folder.

js/modules/Data/models/Quiz.js

```
QuizEngine.module('Data', function(Data) {

    Data.Quiz = Backbone.Model.extend({
        defaults: {
            name: "",
            questions: null
        },

        initialize: function() {
            this.set('questions', new
            Data.QuizQuestions(this.get('questions')));

            this.listenTo(this.get('questions'),
            'change:chosenAnswer', this._transformEvent);
        },

        getCurrentQuestion: function() {
            // Get the first question that hasn't been answered
            return this.get('questions').find(function(question) {
                if (question.get('chosenAnswer') === null) {
                    return true;
                }
            });
        },

        getCurrentPosition: function() {
            // Get the number for the first question that hasn't
            // been answered
            return this.get('questions').reduce(function(memo,
            question, index) {
                if (question.get('chosenAnswer') === null && memo
                === 0) {
                    return index + 1;
                }

                return memo;
            }, 0);
        },
        isComplete: function() {
            // If no questions have a null 'chosenAnswer', the
            // quiz is complete
            return this.get('questions').where({chosenAnswer:
            null}).length === 0;
        },

        isInProgress: function() {
            return !this.isComplete();
        },

        getScore: function() {
            if (this.isInProgress()) {
                return null;
            }

            return Math.round(this.getCorrect() /
            this.get('questions').length * 100);
        },

        getCorrect: function() {
            return this.get('questions').filter(function(
            question) {
                return question.isCorrect();
            }).length;
        },

        // Custom toJSON to also JSONify 'questions'
        toJSON: function() {
            var data = Backbone.Model.prototype.toJSON.call(this);
            if (data.questions && data.questions.toJSON) {
                data.questions = data.questions.toJSON();
            }

            return data;
        },

        _transformEvent: function(question, index) {
            this.trigger('question:answered', question, index);
            if (this.isComplete()) {
                this.trigger('completed');
            }
```

```
        }

    });

});
```

There's a whole lot going on here, at least compared to pretty much any other file we've seen thus far. Let's take it a step at a time, starting in the `initialize` method.

Just like we've done quite a bit in this chapter, we make sure our list of questions is a collection, a `QuizQuestions` collection to be exact. Then we listen to all of the events on that collection for the sake of transforming and forwarding them on. This way, we can listen for `"question:answered"` and `"completed"` instead of trying to listen to all the questions to work out when one is answered and then using `isComplete` to determine if the final question has been answered.

The `getCurrentQuestion` method retrieves the first `QuizQuestion` in the list that hasn't been answered. The `find` method on the `QuizQuestions` collection makes this really simple. Then we have `isComplete` and `isInProgress`, which determine whether the quiz is still in progress or has been completed, judging by the number of questions that have not been answered.

Then there's `getCurrentPosition`, which tells us which question number we've reached. To determine this, we use `reduce` to loop through all the questions and return the `index + 1` of the first unanswered question we encounter.

Next up is `getScore` which returns null if the quiz is still in progress. Otherwise it'll give you an integer percentage of correct answers (it'll return 50 if you have 50% correct).

Other than the `toJSON` override, the last thing to discuss is `getCorrect`, which will return the number of questions that have been answered correctly. I won't go into any detail about the `toJSON` method, as I'm sure you're used to it by now.

js/modules/Data/models/QuizQuestions.js

```
QuizEngine.module('Data', function(Data) {

    Data.QuizQuestions = Backbone.Collection.extend({
        model: Data.QuizQuestion
    });

});
```

QuizQuestions is just another basic Collection that uses QuizQuestion for its model type.

js/modules/Data/models/QuizQuestion.js

```
QuizEngine.module('Data', function(Data) {

    Data.QuizQuestion = Backbone.Model.extend({
        defaults: {
            question: null,
            chosenAnswer: null
        },

        initialize: function() {
            var q = this.get('question');

            if (_.isNumber(q) || _.isString(q)) {
                // If all we have is an identifier, retrieve the
                // actual question
                this.set('question', QuizEngine.module('Data')
                .questions.getQuestion(q));
            }
        },

        isCorrect: function() {
            return this.get('chosenAnswer') === this.get(
            'question').get('correctAnswer');
        },

        // Custom toJSON to also JSONify 'question'
        toJSON: function() {
            var data = Backbone.Model.prototype.toJSON.call(this);
            if (data.question && data.question.toJSON) {
                data.question = data.question.toJSON();
            }

            if (!data.id) {
                data.id = this.cid;
            }

            return data;
        }
    });
});
```

QuizQuestion is our final model. Its initialize method is a bit different. Since Question models already exist, we want to point to the one that's already there, rather than create a new one. If we were given a reference to the actual model from the start, then we don't need to do anything; but if we're given an ID for the Question, we'll need to retrieve it.

Other than the interesting initializer, and the now standard toJSON override, we have one method: isCorrect. This one is pretty simple: it checks to see if the chosenAnswer is the same as the correctAnswer for the question.

Back To The HTML

Now all we have to do is add all these scripts to *index.html*. Go ahead and open it up, then throw these few lines in after the *Router.js* import:

index.html

```
<script src="js/modules/Data/models/Answer.js"></script>
<script src="js/modules/Data/models/Answers.js"></script>
<script src="js/modules/Data/models/Question.js"></script>
<script src="js/modules/Data/models/Questions.js"></script>
<script src="js/modules/Data/models/Category.js"></script>
<script src="js/modules/Data/models/Categories.js"></script>
<script src="js/modules/Data/models/QuizQuestion.js"></script>
<script src="js/modules/Data/models/QuizQuestions.js"></script>
<script src="js/modules/Data/models/Quiz.js"></script>
<script src="js/modules/Data/models/Quizzes.js"></script>
<script src="js/modules/Data/index.js"></script>
```

The order is important since some of these classes depend on other classes being available right away. There are certainly other ways to order the files that will work, but right now you're following this example, so use this order.

Initializing The Data

Now let's initialize all of those Models and Collections with data, and while we're at it let's start up the application – we actually *need* to boot the application to initialize the data. The cool thing is that since the rest of the application will be built via QuizEngine.module, the Models and

`Collections` automatically hook themselves up to initialization, either just by starting with the application or by using the `SubAppManager`. We won't need to make any modifications to the file after this.

Anyway, let's create an *initialize.js* file in the *js* folder. Here are the contents:

js/initialize.js

```
var init_question_data = [
    {
        name: "Tongue Twisters",
        questions: [
            {
                id: 1,
                text: "How much wood could a woodchuck chuck if a
                woodchuck could chuck wood?",
                correctAnswer: 0,
                answers: [
                    {text: "A woodchuck could chuck lots of wood
                    if a woodchuck could chuck wood."},
                    {text: "Is this really a question?"},
                    {text: "It depends on the woodchuck."}
                ]
            },
            {
                id: 2,
                text: "How many cans can a canner can if a canner
                can can cans?",
                correctAnswer: 2,
                answers: [
                    {text: "A canner can can lots of cans if a
                    canner can can cans."},
                    {text: "Again? Really? Why are we doing
                    this?"},
                    {text: "Irrelevant. There's no reason to can
                    cans."}
                ]
            },
            {
                id: 3,
                text: "If Peter Piper picked a peck of pickled
                peppers, where's the peck of pickled peppers
                Peter Piper picked?",
                correctAnswer: 2,
                answers: [
                    {text: "By the other pecks of pickled peppers
                    that Peter Piper picked."},
                    {text: "They went bad, so I threw them out"},
                    {text: "Oh, those were Peter's? I thought
                    they were mine so I ate them."}
                ]
            },
            {
                id: 4,
                text: "This isn't a question? Wait, or is it?",
                correctAnswer: 0,
                answers: [
                    {text: "Well, technically it is."},
                    {text: "Is this a trick question?"},
                    {text: "Psh. No."}
                ]
            }
        ]
    },
    {
        name: "JavaScript Knowledge",
        questions: [
            {
                id: 5,
                text: "What's the most widely-used JavaScript
                library?",
                correctAnswer: 1,
                answers: [
                    {text: "Backbone.js"},
                    {text: "jQuery"},
                    {text: "MarionetteJS"}
                ]
            },
            {
                id: 6,
                text: "What does `typeof null` evaluate to?",
                correctAnswer: 0,
                answers: [
                    {text: "'object'"},
                    {text: "'string'"},
                    {text: "'null'"}
                ]
            },
            {
                id: 7,
                text: "Is the string '0' truthy or falsey",
                correctAnswer: 0,
                answers: [
                    {text: "truthy"},
                    {text: "falsey"}
                ]
            },
            {
                id: 8,
                text: "What is the keyword to access all of the
                arguments passed to a function?",
                correctAnswer: 2,
                answers: [
                    {text: "args"},
                    {text: "parameters"},
                    {text: "arguments"}
                ]
            }
        ]
    },
    {
        name: "MarionetteJS FTW",
        questions: [
            {
                id: 9,
                text: "Which of these lists contains all of the
                View classes for Marionette?",
                correctAnswer: 1,
                answers: [
                    {text: "ModelView, CollectionView, Layout"},
                    {text: "ItemView, CollectionView,
                    CompositeView, Layout"},
                    {text: "View"}
                ]
            },
            {
                id: 10,
                text: "Who is the main author of Marionette?",
                correctAnswer: 0,
                answers: [
                    {text: "Derick Bailey"},
                    {text: "Addy Osmani"},
                    {text: "Paul Irish"}
                ]
            },
            {
                id: 11,
                text: "What is the main color in the Marionette
                logo?",
                correctAnswer: 0,
                answers: [
                    {text: "red"},
                    {text: "blue"},
                    {text: "green"}
                ]
            },
            {
                id: 12,
                text: "What is chapter 1 of <i>Building
                Better Backbone Applications with
                MarionetteJS</i> about?",
                correctAnswer: 2,
                answers: [
                    {text: "Regions"},
                    {text: "Modules"},
                    {text: "Application"}
                ]
            }
        ]
    }
];

var init_quiz_data = [
    {
        name: "Complete Quiz",
        questions: [
            {
                question: 1,
                chosenAnswer: 0
            },
            {
                question: 2,
                chosenAnswer: 0
            },
            {
                question: 3,
                chosenAnswer: 0
            },
            {
                question: 4,
                chosenAnswer: 0
            }
        ]
    },
    {
        name: "Incomplete Quiz",
        questions: [
            {
                question: 5,
                chosenAnswer: 0
            },
            {
                question: 6,
                chosenAnswer: 0
            },
            {
                question: 7,
                chosenAnswer: null
            },
            {
                question: 8,
                chosenAnswer: null
            }
        ]
    }
];

QuizEngine.start({
    data: {
        questions: init_question_data,
        quizzes: init_quiz_data
    }
});
```

This is a really long file, but there are only three things going on here. First, we create the question data, which will be fed to the `Categories` constructor to create the entire question data hierarchy. Then we do the same thing again, except the data is for the quizzes. Notice that when we assign questions to quizzes we just use the ID number for the question. Remember this little tidbit inside the `QuizQuestion` constructor?

```
if (_.isNumber(q) || _.isString(q)) {
    // If all we have is an identifier, retrieve the actual
    // question
    this.set('question', QuizEngine.module('Data').questions
    .getQuestion(q));
}
```

This little conditional block allows us to specify questions in two ways, one of which is very useful while the app is running, and this one while we are trying to initialize.

After all the question and quiz data is created, we start our app and pass that data in. The data will eventually make it to the `Data` module's initializer where it is used to construct the `Collections` and `Models`. And that's all she wrote; or rather, that's all I wrote in this file. Now we need to add the file to HTML. Add this below the rest of your scripts:

index.html

```
<!-- Fire This Sucker Up -->
<script src="js/initialize.js"></script>
```

Bam! Now we're in business! Our application is running! But we still just have a white screen. Well, all we did was get the data running. You can still play around with it in the console, but in the next chapters let's move on to the interesting bits of the application so that users actually have something to look at.

Summary

There are a lot of layers of data in this application, which makes the application seem more complicated than it really is. Hierarchical data sets are common and this is good practice to help you understand them. Numerous Backbone plugins have been built to make the setup and consumption of hierarchical data easier, but if we keep it simple, they aren't really necessary.

Now that we have `QuizEngine` starting, we just need to hook our modules up to get them started as well. Let's put the data behind us and move forward toward the rest of the application where we can actually construct some of these modules. ❧

Chapter 15: The Home Screen

We start our fun in the area where the user would start their fun: the list of quizzes. We made sure to automatically create a couple quizzes during initialization so that the list isn't empty – nobody likes an empty table, right? The application will have something to show the user when the table is empty, but we'll be nice to them to start.

First things first: let's create a folder where we'll put the files for this subapplication. Inside the *js/modules* folder, let's add a *QuizList* folder, and we might as well add a *views* folder inside that one right away. We'll also create an *index.js* file inside the *js/modules/QuizList* folder with the following contents:

js/modules/QuizList/index.js

```
QuizEngine.module('QuizList', function(QuizList) {
    // QuizList Module Must be Manually Started
    QuizList.startWithParent = false;

    // Router needs to be created immediately, regardless of
    // whether or not the module is started
    QuizList.controller = new QuizList.Controller();
    QuizList.router = new QuizList.Router({controller:
    QuizList.controller});

    QuizList.addInitializer(function(){
        QuizList.controller.show();
    });

    QuizList.addFinalizer(function(){
        QuizList.controller.hide();
        QuizList.stopListening();
    });

});
```

Here we create a new module named `QuizList`. We make sure it doesn't start with the application by setting `startWithParent` to `false`. We do this because it's a subapplication and it will be started and stopped by the `SubAppManager`. Next we construct our `Controller` and `Router`. We're using an `AppRouter` — as you'll see soon — so we pass the new `QuizList.controller` in to it.

Notice that we did not put this bit of initialization into `addInitializer`. This is because we want these up and running whether this subapp is running or not. Without them, the subapp would never start because the application starts when the controller reacts to a route, which you'll see soon.

As we continue through the file, we have an initializer and a finalizer. We will use essentially the same initializer and finalizer for each subapp, even though some of the bits will be pointless. I'm simply doing it as a sort of convention. In the initializer, the call to `show` on the controller is designed to make sure any initial setup is done right away. Interestingly, all of the `show` methods in this application are empty, so this is actually just a formality that you can skip right now. Before starting the application I decided to have setup and teardown methods (`show` and `hide`) for each module's controller. I apparently didn't need any setup for any of the modules in this application, but I'll keep `show` in there anyway.

In the finalizer we call `hide` on the controller, which *is* used by each controller. This cleans up any views that the controller rendered, unbinds any listeners the controller may have set up, and clears up memory for garbage collection by removing any references to views or models it may have obtained. After this we call `stopListening` on the module. Once again, this is technically pointless because none of the modules ever actually set up any event listeners, but it's done as a formality and is already there in case we set up some event listeners later.

Now let's take a look at the *js/modules/QuizList/Router.js* file, which you should create. It should look like this:

js/modules/QuizList/Router.js

```
QuizEngine.module('QuizList', function(QuizList) {

    QuizList.Router = Marionette.AppRouter.extend({
        appRoutes: {
            "list": "showList"
        }
    });

});
```

Short and sweet — I like it. There is one route, which *happens* to be the route that we redirect to in the main router (*js/Router.js*). When we hit that route, we'll call `showList` on the controller. Speaking of the controller, let's go take a look at it. It's located at *js/modules/QuizList/Controller.js*; at least it will be there when you create it.

js/modules/QuizList/Controller.js

```
QuizEngine.module('QuizList', function(QuizList) {

    QuizList.Controller = Marionette.Controller.extend({
        // When the module starts, we need to make sure we have
        // the correct view showing
        show: function() {
            // No Special Setup Needed: no-op
        },

        // When the module stops, we need to clean up our views
        hide: function() {
            QuizEngine.body.close();
            this.data = this.view = null;
        },

        // Show List of quizzes for the user
        showList: function() {
            this._ensureSubAppIsRunning();

            this.data = QuizEngine.module('Data').quizzes;
            this.view = new QuizList.QuizzesView(
            {collection:this.data});

            QuizEngine.body.show(this.view);
        },

        // Makes sure that this subapp is running so that we can
        // perform everything we need to
        _ensureSubAppIsRunning: function() {
            QuizEngine.execute('subapp:start', 'QuizList');
        }
    });

});
```

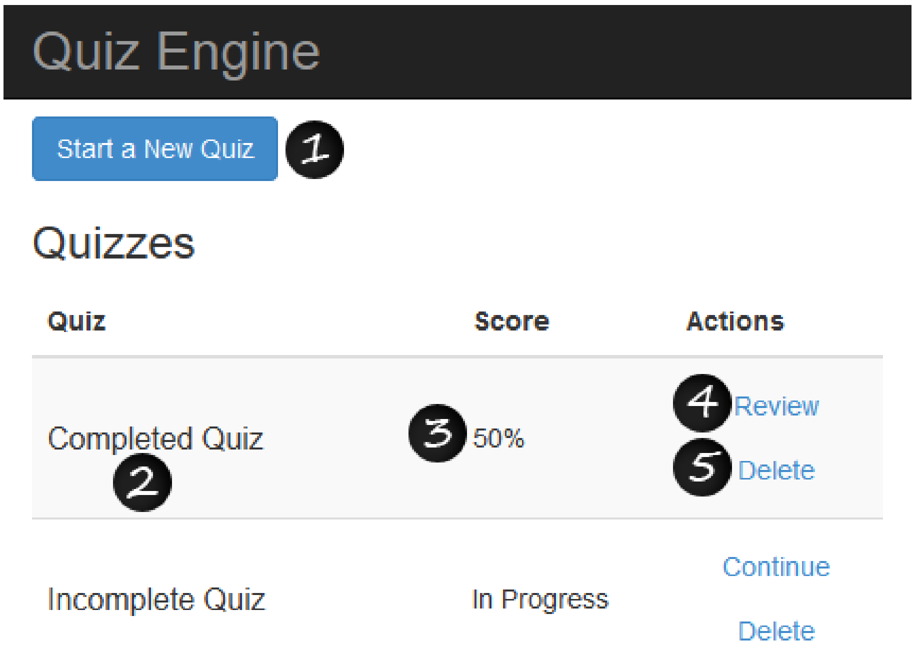
This isn't extremely complicated either. The first things we see at the top, besides the usual `QuizEngine.module` wrapping, are the `show` and `hide` methods. As I mentioned earlier, `show` is empty, but `hide` isn't. Inside `hide`, we know that the only place we'll be attaching views is to the `body` region on `QuizEngine`, so we just close whatever view is there to make sure we clean up after ourselves. We also set our references to the data and view to `null` so that they can be cleaned up by the garbage collector.

Next up is `showList`, which is what the router calls when we hit the `list` route. The first thing we do is make sure that the subapplication is running by calling `_ensureSubAppIsRunning`, which in turn uses the `SubAppManager` via `QuizEngine.execute('subapp:start')`. Normally, I would have added the Backbone RouteFilter plugin to enhance the router to automatically call this for us before executing the route's handler on the controller, but I decided not to add any plugins in to this application.

Beyond that, we get our `quizzes` from the `Data` module and give them to a view to render. Then, finally, we use the `QuizEngine.body` region to show our list of quizzes.

Structurally, this part of the application is important. We see how we can use modules, routers, and controllers in tandem to keep our code separate and organized. It would become even more evident that this structure is useful if the application were larger and had greater complexity, but there is no way to fit that sort of application into this book, so you'll just have to use your imagination.

Now that we've got the foundation for this module set up, let's take a look at bringing that list to the users' eyes. Take another quick peek to remind yourself what exactly we're building here:



Quizzes

Quiz	Score	Actions
Completed Quiz	50%	Review Delete
Incomplete Quiz	In Progress	Continue Delete

Quizzes List Reminder

I assume you noticed that this screenshot is a bit different from the last one: this one has numbered annotations. These are highlights of things we should take note of as we build our views.

1. This button will bring us to the quiz builder. The easiest way to do this is to make it a link to the route for the quiz builder, rather than hooking up any event handlers and using JavaScript to route it.
2. This is the name of the quiz. No big deal. Something to note, though: there is only one button for starting a new quiz, yet there is a list of several quizzes. This means we'll need to use a `CompositeView` so we can have the wrapping template and list the quizzes within it using separate views.
3. This here is the score. Notice two things: when a quiz is in progress, it shows something other than score; and the score isn't an attribute saved on a quiz, so it'll have to be calculated. Both of these points mean that we'll need some template helpers to get it to display what we need.
4. For completed quizzes, this first action will be "Review". Incomplete quizzes get a "Continue" action. Another template helper will be needed.
5. The "Delete" action should immediately remove the quiz from the data as well as remove it from the view. We get this second part for free with `CompositeView`, since it will re-render the child views when a model is added or removed from its collection.

So, we've established that we'll need a `CompositeView` to wrap everything up, and then we'll need some views for showing the individual quizzes in the list. What do we show to the user if the list is empty? We'll have to create another view for that and use the `emptyView` property in `CompositeView` to show it.

Let's start with templates. Inside the HTML file, add this code below the "Templates" comment and before the scripts:

index.html

```
...

<script type="text/template" id="quizlist-quizzes">
    <div class="col-xs-12">
        <div><a href="#new-quiz" class="btn btn-primary">Start a
        New Quiz</a></div>
        <h3>Quizzes</h3>
        <table class="table table-striped">
```

```
                <thead>
                    <tr>
                        <th class="col-xs-6">Quiz</th>
                        <th class="col-xs-3">Score</th>
                        <th class="col-xs-3">Actions</th>
                    </tr>
                </thead>
                <tbody data-item-view-container></tbody>
            </table>
        </div>
</script>

<script type="text/template" id="quizlist-quiz">
    <td class="lead"><%= name %></td>
    <td><%= score() %></td>
    <td>
        <a href="#quiz/<%= id() %>" class="btn btn-link col-xs-12
        col-sm-6"><%= viewAction() %></a>
        <button data-action="delete" class="btn btn-link
        col-xs-12 col-sm-6">Delete</button>
    </td>
</script>

<script type="text/template" id="quizlist-noquizzes">
    <td colspan="3" class="text-center">
        You have no quizzes here. Click the "Start a New Quiz"
        button to kick things off!
    </td>
</script>
...
```

There are three different templates there. Let's start with the first one, which is for the `CompositeView`. Using some HTML designed for Twitter Bootstrap, we show a button, which is actually a link to `"#new-quiz"`, which will bring up the quiz builder. Then we have a simple header saying "Quizzes", which leads us to the table. The `<tbody>` tag has a `data-item-view-container` attribute, which is what we'll use to select the proper place for the `ItemViews`. If you need to add an attribute as an identifier solely for JavaScript, using the `data-` attributes is probably your best bet.

Speaking of the `ItemViews`, our next template is for each individual quiz in the list. The first thing you see is that we're filling the first column with the name of the quiz, which is grabbed simply from the `name` attribute of the model. The rest of the data takes a bit more work to glean, so we use template helpers. The first one is the score. There is no `score` attribute on the model, and since we also want this column to say “In Progress” for incomplete quizzes, it wouldn’t be useful anyway. Next up is `id()`, which is actually used to grab the `cid`, since we don’t have any explicit IDs for quizzes. It is used to build a link that points to the URL for showing a quiz. Both the “Review” and “Continue” links point to the same URL, and the `Quiz` module will sort out whether to continue taking the quiz or show the review page depending on the state of the quiz. We determine whether to show the “Review” or “Continue” text by using another template helper: `viewAction`. Finally, we have the “Delete” button which has the `data-action` attribute set to `"delete"` for both semantics and to help us find it for applying the event handler to it.

The third template is shown when there are no quizzes. Nothing special: just text informing you of the void and how to fill it. Now all we need is some views to use these templates.

js/modules/QuizList/views/QuizzesView.js

```
QuizEngine.module('QuizList', function(QuizList) {

    QuizList.QuizzesView = Marionette.CompositeView.extend({
        template: '#quizlist-quizzes',
        itemView: QuizList.QuizView,
        itemViewContainer: '[data-item-view-container]',

        emptyView: QuizList.NoQuizzesView
    });

});
```

The is the `CompositeView` that contains the heading and table. As you might expect, `QuizzesView` uses the `’#quizlist-quizzes’` template and finds the `itemViewContainer` via the `data-item-view-container` attribute. If you wanted to, since the `data-` attribute is so generic, you could create a custom `CompositeView` that just specifies the `itemView-Container`, and then extend that one every time instead of specifying it each time. Anyway, you can also see that we specified the `itemView` and `emptyView`.

It still amazes me that we could have something so complicated as a `CompositeView` like this and have it working by specifying three or four quick properties. But that’s enough gawking. Let’s move on to `QuizView`.

js/modules/QuizList/views/QuizView.js

```
QuizEngine.module('QuizList', function(QuizList) {

    QuizList.QuizView = Marionette.ItemView.extend({
        tagName: 'tr',
        template: '#quizlist-quiz',
        templateHelpers: function() {
            var model = this.model;

            return {
                score: function() {
                    return model.isInProgress() ? "In Progress" :
                    model.getScore() + "%";
                },
                id: function() {
                    return model.cid;
                },
                viewAction: function() {
                    return model.isComplete() ? "Review" :
                    "Continue";
                }
            };
        },

        events: {
            'click [data-action=delete]': 'deleteQuiz'
        },

        deleteQuiz: function() {
            this.model.destroy();
        }
    });

});
```

Here's our first look at an `ItemView` in the wild. We create our new view as `QuizView` on the `QuizList` module. We start off our view by specifying a `tagName` as `'tr'`. This ensures that we don't add `<div>`s into our `<tbody>` and prevents us from needing to add the `<tr>` tags in our template. Then we specify our template and template helpers.

Notice I'm specifying `templateHelpers` as a function that returns an object. This is because when a template helper is run inside a template, `this` won't refer to the view object. To access `this.model`, we need to make it available to the template helpers through a closure.

As we go through each template helper, we don't really see anything special. We just get data from the model and, based on that data, create some formatted text to display to the user — with the exception of `id`, which just returns the pure `cid` of the model.

After this, we move on to our single event handler. Whenever you click the `[data-action=delete]` button, we'll run `deleteQuiz`, which simply destroys the model. Our `QuizzesView` will detect this deletion and will re-render the list.

I really like using the `data-action` attribute on elements that we'll monitor events on. It's semantic and it allows us to group every element with the same functionality together into a single selector. For example, if we had a larger view with a lot of content in it, plus a button bar on the top *and* bottom (for the user's convenience so instead of scrolling to either just the top or bottom, they can scroll to whichever is closer), we can use the same `data-action` attribute for the matching buttons on top and bottom.

Well, that's enough about `QuizView`; let's move on to `NoQuizzesView`, which is the view that will be rendered when there are no quizzes to list.

js/modules/QuizList/views/NoQuizzesView.js

```
QuizEngine.module('QuizList', function(QuizList) {

    QuizList.NoQuizzesView = Marionette.ItemView.extend({
        tagName: 'tr',
        template: '#quizlist-noquizzes'
    });

});
```

This file is much smaller, because all we really need to do is render a template — specifically the `#quizlist-noquizzes` template — inside a `<tr>` element. There's no functionality to worry about, since it displays only text telling the user to start a new quiz.

We mustn't forget to add the scripts to the HTML. Paste the following code beneath the scripts for the `Data` module, but keep them above the *initialize.js* script:

index.html

```
<!-- ... Data module scripts -->
<script src="js/modules/QuizList/views/QuizView.js"></script>
```

```
<script src="js/modules/QuizList/views/NoQuizzesView.js"></script>
<script src="js/modules/QuizList/views/QuizzesView.js"></script>
<script src="js/modules/QuizList/Controller.js"></script>
<script src="js/modules/QuizList/Router.js"></script>
<script src="js/modules/QuizList/index.js"></script>
<!-- ... js/initialize.js -->
```

Summary

And that wraps up the `QuizList` module. Load up the app and you should see the list of quizzes. You should also be able to delete them. Just refresh the browser to bring back the deleted quizzes. The other functionality is coming up next. More specifically, we'll be looking at the `QuizCreator` module in the next chapter. ❧

Chapter 16: The Quiz Creator

In this chapter we'll move on to our second module: the quiz creator. In this module, we'll give the user the ability to construct a quiz by choosing the categories of questions they want to appear on their quiz.

Just like the previous two modules, we'll start with the *index.js* main module file. Be sure to create a *QuizCreator* folder in the *modules* folder and stick *index.js* in there. Interestingly, we can copy and paste the code from the *index.js* file for the `QuizList` module and just change any reference of `QuizList` to `QuizCreator`:

js/modules/QuizCreator/index.js

```
QuizEngine.module('QuizCreator', function(QuizCreator) {
    // QuizCreator Module Must be Manually Started
    QuizCreator.startWithParent = false;

    // Router needs to be created immediately, regardless of
    // whether or not the module is started
    QuizCreator.controller = new QuizCreator.Controller();
    QuizCreator.router = new QuizCreator.Router({controller:
    QuizCreator.controller});

    QuizCreator.addInitializer(function(){
        QuizCreator.controller.show();
    });

    QuizCreator.addFinalizer(function(){
        QuizCreator.controller.hide();
        QuizCreator.stopListening();
    });

});
```

I won't waste your time going into detail about this file since, as I mentioned, it's the same as *js/modules/QuizList/index.js*. The router and controller also share some similarities with the ones in the last module:

js/modules/QuizCreator/Router.js

```
QuizEngine.module('QuizCreator', function(QuizCreator) {
    QuizCreator.Router = Marionette.AppRouter.extend({
        appRoutes: {
            "new-quiz": "showForm"
        }
    });

});
```

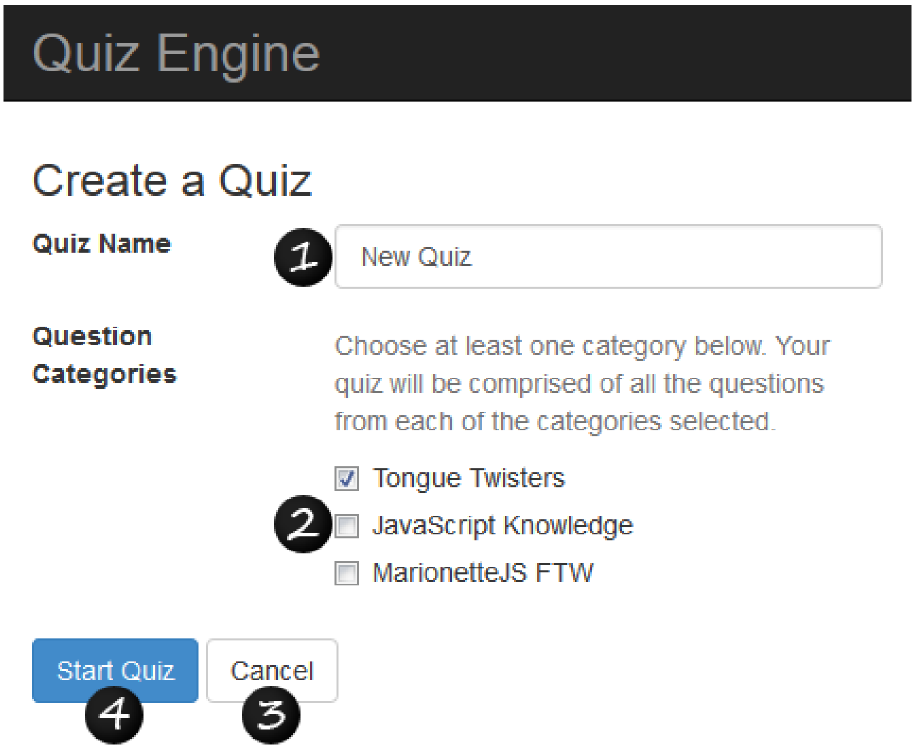
Once again, we have a single route, which runs a method on the controller. This time the route is `"new-quiz"` and the method name is `showForm`. Let's see how the controller handles this:

js/modules/QuizCreator/Controller.js

```
QuizEngine.module('QuizCreator', function(QuizCreator) {

    QuizCreator.Controller = Marionette.Controller.extend({
        // When the module starts, we need to make sure we have
        // the correct view showing
        show: function() {
            // No Special Setup Needed: no-op
        },

        // When the module stops, we need to clean up our views
        hide: function() {
            QuizEngine.body.close();
            this.data = this.view = null;
        },

        // Show List of quizzes for the user
        showForm: function() {
            this._ensureSubAppIsRunning();

            this.data = QuizEngine.module('Data').questions;
            this.view = new QuizCreator.FormView({collection:
            this.data});

            QuizEngine.body.show(this.view);
        },

        // Makes sure that this subapp is running so that we can
        // perform everything we need to
        _ensureSubAppIsRunning: function() {
            QuizEngine.execute('subapp:start', 'QuizCreator');
        }
    });

});
```

We have the same empty `show` method used by the module initializer. We also have an identical `hide` method so the module can clean up after itself. Not even the `showForm` method is all that different than the `showList` method from the previous module. The only differences are that I'm grabbing `questions` from the `Data` module instead of `quizzes`, and I'm using a different view: `FormView`. So there's nothing really new to learn here. If this happens a lot in an application, it might be a good idea to abstract that functionality out into a new class and extend it, to save on code bytes and time copying and pasting.

Let's move on to what we'll actually see and use for this module.



Choose at least one category below. Your quiz will be comprised of all the questions from each of the categories selected.

- [x] Tongue Twisters
- [ ] JavaScript Knowledge
- [ ] MarionetteJS FTW

Quiz Creator Reminder

To elaborate on the numbered points in the image right away:

1. The quiz name. This is what the quiz will be called by humans.
2. Question categories. We can choose one or more question categories. When the quiz is being built, it'll use all of the questions from each of the chosen categories.

3. This "Cancel" button is really just a link back to the quiz list. In a more complicated application, it might be wiser to use `Backbone.history-.history.back()` to go back a page, but that's up to you.

4. This button will fire off an event to get the quiz built, and forward you to the URL where you can start taking the quiz.

Now that we've gotten that explanation out of the way, let's take a look at the template we'll be using. Place this template inside your *index.html* file where the rest of your templates are sitting.

index.html

```
<script type="text/template" id="quizcreator-form">
    <div class="col-xs-12">
        <h3>Create a Quiz</h3>
        <form>
            <div class="form-group row">
                <label for="quizName" class="col-xs-4
                col-sm-2">Quiz Name</label>
                <div class="col-xs-8 col-sm-10">
                    <input class="form-control" id="quizName"
                    type="text" value="Quiz">
                </div>
            </div>
            <div class="form-group row">
                <label class="col-xs-4 col-sm-2">Question
                Categories</label>
                <div class="col-xs-8 col-sm-10">
                    <p class="help-block">Choose at least one
                    category below. Your quiz will be comprised
                    of all the questions from each of the
                    categories selected.</p>
                    <% _.each(items, function(category) { %>
                        <div class="checkbox">
                            <label>
                                <input type="checkbox" value="<%=
                                category.id %>"> <%=
                                category.name %>
                            </label>
                        </div>
                    <% }); %>
                </div>
            </div>
        </form>
        <button data-action="start" class="btn btn-primary">Start
        Quiz</button>
        <a href="#list" class="btn btn-default">Cancel</a>
    </div>
</script>
```

A lot of the `<div>`s and classes are used to let Bootstrap make it look awesome at different screen sizes. There are only a few things worthy of interest at this point:

- For the checkboxes, I looped through the collection. This is an example of using an `ItemView` to display collections. Here, we simply loop through each category to display a checkbox and label that have the ID and name of the category, respectively.
- We once again use the `data-action` attribute on the “Start Quiz” button.
- Like I mentioned earlier, the “Cancel” button is simply a link to the quizzes list.

If you remember, I had this `toJSON` method in the `Category` class:

```
toJSON: function() {
    var data = Backbone.Model.prototype.toJSON.call(this);
    if (data.questions && data.questions.toJSON) {
        data.questions = data.questions.toJSON();
    }

    if (!data.id) {
        data.id = this.cid;
    }

    return data;
}
```

Thanks to that bit about `data.id`, we’re able to access the `cid` of the category directly in the template rather than relying on a helper. You could do this for everything, but I did it specifically for `Category` because in this instance, if we wanted to get the `cid` of a category, we’d need to use the index to grab the category from the list of categories, and then ask for the `cid`. This method simplifies things pretty significantly. Anyway, let’s move on to the view.

js/modules/QuizCreator/views/FormView.js

```
QuizEngine.module('QuizCreator', function(QuizCreator) {

    QuizCreator.FormView = Marionette.ItemView.extend({
        template: '#quizcreator-form',

        events: {
            'click [data-action=start]': 'createQuiz'
        },

        createQuiz: function() {
            var categories = [],
                quizName = this.$el.find('#quizName').val(),
                questions, quiz;

            if (!quizName) {
                alert("Please enter a quiz name");
                return;
            }

this.$el.find('input[type=checkbox]:checked').each(function() {
                categories.push(this.value);
            });

            if (categories.length < 1) {
                alert("Please select at least one category");
                return;
            }

            questions = this.collection.getQuestionsByCategories
            (categories);
            quiz = QuizEngine.module('Data').quizzes.createQuiz
            (quizName, questions);
            Backbone.history.navigate('quiz/' + quiz.cid,
            {trigger:true});
        }
    });

});
```

Here, we specify our `'#quizcreator-form'` template, set up the event listener on the `[data-action=start]` button. Then there's that big `createQuiz` method. It may have been wiser to move much of this logic into the controller, and then use the controller to listen to an event on the view that informs the controller that a quiz should be built, but in this small application, it's fine to keep this bit of logic in the view.

The `createQuiz` method retrieves the values from the form inputs and does some simple validation on them — when the quiz name is empty or no categories are chosen, we pop up an alert box to let the user know they need to fill those in. Then it gets the questions based on the which checkboxes were selected. Next, it creates the quiz and routes us to the URL that will allow us to start taking the quiz.

Cool, now that we have all the `QuizCreator` module's components done, let's make sure the scripts are loaded in the HTML. Load the scripts after the `QuizList` module's script and before the *js/initialize.js* file.

index.html

```
<!-- ... QuizList module scripts -->
<script src="js/modules/QuizCreator/views/FormView.js"></script>
<script src="js/modules/QuizCreator/Controller.js"></script>
<script src="js/modules/QuizCreator/Router.js"></script>
<script src="js/modules/QuizCreator/index.js"></script>
<!-- ... js/initialize.js -->
```

Summary

After all of that work, you can run the app, click "Start a Quiz" and get a quiz created. Now all we need to do is allow you to take that test and review it, which we'll do in our next (and final) chapter. ❧

Chapter 17: Taking and Reviewing Quizzes

In this final chapter, we'll finish up our Quiz Engine application by implementing the `Quiz` module, which handles both taking and reviewing quizzes, depending on the state of the quiz.

Once again we'll start by looking at the *index.js* file for the module, which will be in a new folder called *Quiz* inside the *modules* folder.

js/modules/Quiz/index.js

```
QuizEngine.module('Quiz', function(Quiz) {
    // Quiz Module Must be Manually Started
    Quiz.startWithParent = false;

    // Router needs to be created immediately, regardless of
    // whether or not the module is started
    Quiz.controller = new Quiz.Controller();
    Quiz.router = new Quiz.Router({controller: Quiz.controller});

    Quiz.addInitializer(function(){
        Quiz.controller.show();
    });

    Quiz.addFinalizer(function(){
        Quiz.controller.hide();
        Quiz.stopListening();
    });

});
```

Looks familiar, doesn't it? Yup, it's the same as the other ones with the exception of the module name. Let's move on to the next piece of the puzzle, then: the router.

js/modules/Quiz/Router.js

```
QuizEngine.module('Quiz', function(Quiz) {

    Quiz.Router = Marionette.AppRouter.extend({
        appRoutes: {
            "quiz/:cid": "showQuiz"
        }
    });

});
```

Once again, we have a single route. This time, though, we're actually grabbing a parameter from the route: the `cid` of the quiz. Since there isn't much to see here, let's move on to the controller right away.

js/modules/Quiz/Controller.js

```
QuizEngine.module('Quiz', function(Quiz) {

    Quiz.Controller = Marionette.Controller.extend({
        // When the module starts, we need to make sure we have
        // the correct view showing
        show: function() {
            // No Special Setup Needed: no-op
        },

        // When the module stops, we need to clean up
        hide: function() {
            QuizEngine.body.close();
            this.stopListening();
            this.data = this.view = null;
        },

        showQuiz: function(cid) {
            this._ensureSubAppIsRunning();
            this.stopListening();

            this.data =
            QuizEngine.module('Data').quizzes.get(cid);
            this.view = new Quiz.QuizView({model:this.data});

            QuizEngine.body.show(this.view);
            if (this.data) {
                if (this.data.isComplete()) {
                    this.showReview();
                }
                else {
                    this.listenTo(this.data, 'question:answered',
                    this.showQuestion);
                    this.listenTo(this.data, 'completed',
                    this.showReview);
                    this.showQuestion();
                }
            }
        },

        showReview: function() {
            var subView = new Quiz.QuizReviewView({model:
            this.data});

            this.renderSubView(subView);
        },

        showQuestion: function() {
            var question = this.data.getCurrentQuestion();

            if (question) {
                var subView = new Quiz.QuizQuestionView({
                    model: question,
                    questionNumber:
                    this.data.getCurrentPosition(),
                    quizLength: this.data.get('questions').length
                });

                this.renderSubView(subView);
            }
        },

        renderSubView: function(subView) {
            this.view.quizData.show(subView);
        },

        // Makes sure that this subapp is running so that we can
        // perform everything we need to
        _ensureSubAppIsRunning: function() {
            QuizEngine.execute('subapp:start', 'Quiz');
        }
    });

});
```

Before I go into the controller's details, I need to tell you how I decided to handle this module. First of all, we use the same route for taking the quiz and for reviewing it. The route simply says "show me the quiz" and depending on the state of the data, we could show a question that needs to be answered, or the review screen. Then, when a question is answered, the controller responds to the event by re-rendering with the next question. If the quiz is completed, it will instead show the review screen.

Also, we used a `Layout` here because whether we're showing a single question or showing the full review, there is a part of the view that is constant: the quiz name. So we abstracted the quiz name into its own template, which is rendered by the `Layout` and then the rest of the view is separate and is shown via the `Region` on the `Layout`.

That explanation should help us understand what is going on as we traverse the methods of our controller. We start off with the familiar `show` and `hide` methods. I won't go into them.

After that we come to `showQuiz`, which is the method that the router calls. As usual, we make sure that the current module is running. Since `showQuiz` is only called by the router, we have the controller `stopListening` to any events that it was listening to before, because it only listens to events on a single quiz. If someone went straight to one quiz after looking at a different quiz, then we need to stop listening to the events on the old quiz, otherwise there's a possibility that we could react to changes on a quiz that we don't currently care anything about.

After clearing the event listeners, we retrieve our quiz and create a `QuizView`, which is our layout. We immediately render it onscreen so that the regions are available. Also, if there was no quiz that matched the `cid`, the `Layout` will render a different template stating that the quiz doesn't exist, so that's another reason to have it rendered right away.

After that, we check to see if the quiz does indeed exist. If it does, we check to see if the quiz is completed. If it is, we render the review view; if not, we want to show the first available quiz question. Before we show the question, though, we register some event listeners on the quiz.

Every time a question is answered, we render a new question. This can be a problem after we answer the last question because there won't be any more questions to render. That's why we have a check in `showQuestion` to make sure there is a question available to render before it does anything.

Also, when a quiz is completed, we will show the review screen. In practice, when the final question is answered the `'question:answered'` event will fire, which will call `showQuestion`, which will end up not showing anything. Then immediately afterward, the `'completed'` event will fire and `showReview` will be called.

The `showReview` and `showQuestion` methods do pretty much what you'd expect, based on how view rendering was handled in the previous modules. There's the obvious difference that these views will be rendered through a region on `QuizView` instead of through a region on `QuizEngine`. Also, `QuizQuestionView` requires some interesting information to be sent in since the model we give it is the question that it will render, and we'll need some information from the quiz. Instead of giving both the quiz and the question to the view, we give it the individual pieces that it would need from the quiz. Either approach would work just fine.

Now let's move on to the visual side of things.

We'll start out by showing the templates and the class for `QuizView`. Add these templates into the *index.html* file:

index.html

```
<script type="text/template" id="quiz-quiz">
    <div class="col-xs-12">
        <h3><%= name %></h3>
        <div data-region="quizData"></div>
    </div>
</script>

<script type="text/template" id="quiz-quiz404">
    <div class="col-xs-12 text-center">
        <h3>This Quiz Does Not Exist</h2>
        <p><a href="#list">Return to Your Quizzes</a></p>
    </div>
</script>
```

The first template is what we'll see most of the time. It shows the quiz name and it sets up an area for the region to connect. The second template is shown if the quiz doesn't exist. It's a simple message letting the user know that the quiz doesn't exist. It then also provides a link back to the quiz list.

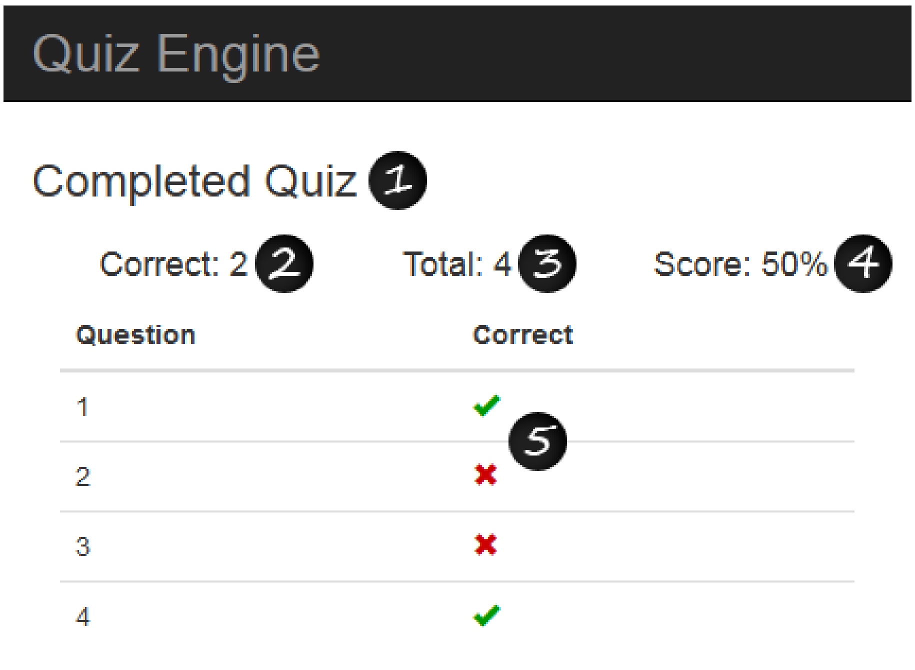
Now let's look at the `QuizView` class itself:

js/modules/Quiz/views/QuizView.js

```
QuizEngine.module('Quiz', function(Quiz) {

    Quiz.QuizView = Marionette.Layout.extend({
        template: '#quiz-quiz',
        emptyTemplate: '#quiz-quiz404',
        getTemplate: function() {
            if (this.model) {
                return this.template;
            }
            else {
                return this.emptyTemplate;
            }
        },

        regions: {
            quizData: '[data-region=quizData]'
        }
    });

});
```

`QuizView` references two templates, then overrides the `getTemplate` method to retrieve the template we need, depending on whether we actually have a model or not. The only other thing we do is define our region. Simple enough.

Let's move on to the view for quiz reviews.



Quiz Engine

Completed Quiz

Correct: 2   Total: 4   Score: 50%

Question	Correct
1	✓
2	✗
3	✗
4	✓

Quiz Review Reminder

1. This the quiz name, which is actually taken care of in the `QuizView`.

2. This is the first of the stats. We display the number of questions that were answered correctly. We can get this number from the `getCorrect` method on a quiz.

3. This is the total number of questions on the quiz. This is retrieved simply by getting the `length` of the `questions` collection on a quiz.

4. This is the score, which is available through the `getScore` method on a quiz.

5. Whether you use a checkmark or an “X” is determined by calling `isCorrect` on each individual question.

Here is the template for the review screen. As usual, add it to the *index.html* file.

index.html

```
<script type="text/template" id="quiz-quizreview">
    <div class="col-xs-12 col-sm-4">
        <div class="row">
            <h4 class="col-xs-4 col-sm-12 text-center">Correct:
            <%= getCorrect() %></h4>
            <h4 class="col-xs-4 col-sm-12 text-center">Total: <%=
            getTotal() %></h4>
            <h4 class="col-xs-4 col-sm-12 text-center">Score: <%=
            getScore() %></h4>
        </div>
    </div>
    <div class="col-xs-12 col-sm-8">
        <table class="table">
            <thead>
                <th class="col-xs-4">Question</th>
                <th class="col-xs-4">Correct</th>
            </thead>
            <tbody>
                <% _.each(questions, function(question, index) {
                %>
                    <tr>
                        <td><%= index + 1 %></td>
                        <td>
                            <% if (isCorrect(question.id)) { %>
                                <span class="glyphicon
                                glyphicon-ok green"></span>
                            <% } else { %>
```

```
                                <span class="glyphicon
                                glyphicon-remove red"></span>
                            <% } %>
                        </td>
                    </tr>
                <% }); %>
            </tbody>
        </table>
    </div>
</script>
```

You'll notice that just about every bit of data we need for this template is retrieved through a template helper. The exceptions are the question number, which is determined by the index of the loop through the questions; and the question's ID, which is used as the parameter of a template helper.

Inside this template you will notice that we're looping through a collection again. This is partly because the collection is actually only a portion of the data, but also that it would be a waste of resources to create a view for each of these individual questions, especially since they have no functionality: they simply show the results.

This is actually true of the entire view. There are no events to listen for or react to. If it weren't for all of the necessary template helpers, the view would be rather small and uninteresting.

js/modules/Quiz/views/QuizReviewView.js

```
QuizEngine.module('Quiz', function(Quiz) {

    Quiz.QuizReviewView = Marionette.ItemView.extend({
        template: '#quiz-quizreview',

        templateHelpers: function() {
            var quiz = this.model;

            return {
                getCorrect: function() {
                    return quiz.getCorrect();
                },
                getTotal: function() {
                    return quiz.get('questions').length;
                },
                getScore: function() {
                    return quiz.getScore() + "%";
                },
                isCorrect: function(qid) {
                    var question = quiz.get('questions').get(qid);

                    return question.isCorrect();
                }
            };
        }
    });

});
```

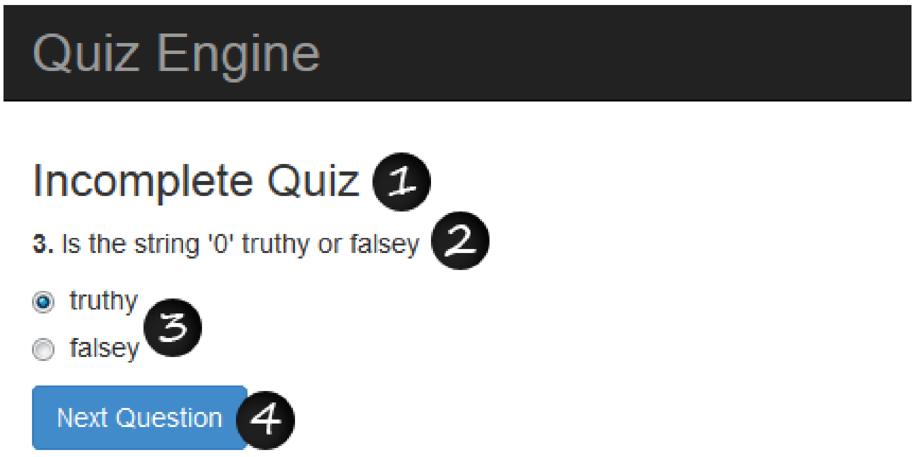
Like I said, there isn't much here, except template helpers. For the most part, the template helpers are straightforward, but I'll look at `isCorrect` in more detail. It seems a bit wasteful to grab the ID from a question just to pass that ID into a function that retrieves a question using that ID. The issue is that the `question` in the template is a JSON representation of the action question object, and therefore does not have the `isCorrect` method, which is what we need.

Alternatively, we could pass `question` into the template helper and change it to this:

```
isCorrect: function(question) {
    return question.chosenAnswer ===
    question.question.correctAnswer;
}
```

We could even just use that conditional in the template instead of using a template helper. The issue with this solution is that I consider comparing `chosenAnswer` with `question.correctAnswer` to be an implementation detail, which should be hidden behind a method (just like it is in the real application). Even if you are OK with using this implementation detail instead of using the public API (the `isCorrect` method), you are duplicating code, which is obviously not a good thing.

Anyway, that was a long dissertation on a single, lowly template helper. Let's leave `isCorrect` alone now and move on to taking a quiz.



Quiz Taking Reminder

1. Once again, we need to show the quiz name, which is handled by the `Layout`.
2. On this line, we have both the question number and the question text. The question number is one of the pieces of data that we passed in to the view from the controller.
3. We need to show some answer options to the user, otherwise it isn't much of a quiz, is it? If the answer is left blank, then we'll also need to inform the user that they made a mistake.
4. This button will simply fire off an event, which will then be used to change the `chosenAnswer` of the question. Then the question will throw around some events, which will lead to the quiz firing off events, which leads to the controller re-rendering with a new question. If we're on the last question of the quiz, this button will say "Finish Quiz" instead of "Next Question".

Here's the template that makes all of this possible:

index.html

```
<script type="text/template" id="quiz-quizquestion">
    <p><strong><%= questionNumber() %>.</strong> <%=
    question.text %></p>
    <form id="form">
        <% _.each(question.answers, function(answer, index) { %>
            <div class="radio">
                <label>
                    <input type="radio" name="answers" value="<%=
                    index %>">
                    <%= answer.text %>
                </label>
            </div>
        <% }); %>
        <button class="btn btn-primary" data-action="next"><%=
        isLastQuestion() ? "Finish Quiz" : "Next Question"
        %></button>
    </form>
</script>
```

Might as well show `QuizQuestionView` right away, too.

js/modules/Quiz/view/QuizQuestionView.js

```
QuizEngine.module('Quiz', function(Quiz) {

    Quiz.QuizQuestionView = Marionette.ItemView.extend({
        template: '#quiz-quizquestion',
        templateHelpers: function() {
            var options = this.options;

            return {
                questionNumber: function() {
                    return options.questionNumber;
                },
                isLastQuestion: function() {
                    return options.questionNumber ===
                    options.quizLength;
                }
            };
        },

        events: {
            'click [data-action=next]': 'submitAnswer'
        },

        submitAnswer: function(event) {
            var selectedAnswer = this.$(':radio:checked').val();

            if (_.isUndefined(selectedAnswer)) {
                alert("Please select an answer");
            }
            else {
                this.model.set('chosenAnswer',
                parseInt(selectedAnswer, 10));
            }

            event.preventDefault();
        }
    });

});
```

Now we can see how the template and view relate to each other more easily because they are right next to one another. The template uses the `questionNumber` helper, which simply returns the `questionNumber` option that was passed to the view's constructor. Remember, the first parameter (which in most cases is an object literal) passed to a view's constructor is stored on the view's `options` property so they can all be accessed later.

The `isLastQuestion` helper uses both the `questionNumber` and the `quizLength` options that were passed in to determine if we are on the final question of the quiz, which is used to decide what the button says.

We also set up an event listener on the `[data-action=next]` button in order to submit the chosen answer. Once again, we do simple validation and use `alert` to alert users to an error. Then, if it passes, we just change the `chosenAnswer` of the question, firing off a series of events, as I've already mentioned.

Well, that's everything. All we need to do is put the final scripts into the HTML, and we should have a fully functional Quiz Engine application. Place the scripts just above the *js/initialize.js* script file:

index.html

```
<script src="js/modules/Quiz/views/QuizView.js"></script>
<script src="js/modules/Quiz/views/QuizReviewView.js"></script>
<script src="js/modules/Quiz/views/QuizQuestionView.js"></script>
<script src="js/modules/Quiz/Controller.js"></script>
<script src="js/modules/Quiz/Router.js"></script>
<script src="js/modules/Quiz/index.js"></script>
```

Now fire it up!

More Complicated Applications

Sadly, this application doesn't show you every type of scenario that you might run into when constructing a Marionette application. There are many larger applications with more complicated layouts and features that would require some more thought and organization than this.

Here are a few different scenarios you might run into just with how you use modules:

- Multiple modules needing to be run at the same time at different locations on the layout. Some of the modules may even be compatible to be used in multiple locations on the layout. Some changes and upgrades would be necessary for the `SubAppManager` at least.
- Modules might need to be controlled by something other than routes. Some may be entirely reliant on the state of data to determine how and when they are used. Others may be controlled by events, requests, or commands. In these instances, you'll likely set up the handlers and listeners in the module's *index.js* file and have them call methods on the controller.
- Some modules don't have a life cycle: they are constantly on, as long as the application is. For example, if you have a module dedicated to navigation, it would likely persist throughout the session.

I wish there was a simple application that could fit into this book (this one barely does) that could show you everything you'll ever need to know about all of Marionette's capabilities, but it is quite likely that such an application does not exist, even outside the constraints of this book.

I can only hope that this Quiz Engine application has taught you everything you need to know right now to get started, and point you in the right direction of where to go in the other circumstances. I also hope you enjoyed learning about Marionette. This is where our journey together ends, so I wish you happy coding and God's blessings. ❧

Additional Resources

This little appendix is here to give you a few more resources to help you out as you explore Marionette more on your own. Some were mentioned in passing earlier in the book, but they're all listed here.

Official MarionetteJS Resources

- MarionetteJS.com[37]:
  Here you can download the library and point yourself to the GitHub repository, documentation, and other resources.
- Documentation[38]:
  This is the official documentation for Marionette, which is a great reference when you're trying to remember a certain detail.
- GitHub Repository[39]:
  If it's not in the documentation, you can find your answer in the code. Also, if you are feeling ambitious, you could fork the code and help implement some new features or fix some bugs.
- BBCloneMail[40]:
  This is a simple example web application designed to demonstrate how to use Marionette the way it was intended to be used (similar to chapter 10).

Blogs, Articles, and Screencasts

- Derick Bailey on Los Techies[41]:
  Los Techies is a blog that Derick Bailey frequently contributes to. He sometimes announces updates to Marionette and occasionally goes into detail about why he built certain Marionette components and how he uses them.
- Brian Mann's "The Tools and Patterns for Building Large-Scale Backbone Applications"[42]:

37. http://marionettejs.com
38. https://github.com/marionettejs/backbone.marionette/wiki
39. https://github.com/marionettejs/backbone.marionette
40. https://github.com/marionettejs/bbclonemail/
41. http://lostechies.com/derickbailey/author/derickbailey/
42. https://www.youtube.com/watch?v=qWr7x9wk6_c

Brian Mann gives a talk about using Marionette to build large-scale Backbone applications. He has a lot of great insight into the theory behind large, scalable architecture.

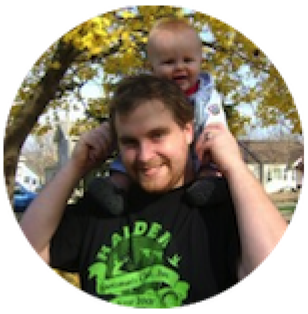
- Joe Zim's JavaScript Blog[43]:
  This is my personal JavaScript blog. I've written a lot about Backbone and plan to write a lot more about Marionette. The blog has fallen a bit to the wayside since I started writing this book, but now that this book is in your hands, it is out of mine, so I'm free to blog more frequently again.

Books

- *Backbone.Marionette.js: A Gentle Introduction*[44]:
  David Sulc takes you through Marionette by slowly building an entire application through the course of the book. The application is larger than the one you find here, and David often uses some components in different ways to me, so you'll get a new perspective on how Marionette can be used.
- *Structuring Backbone Code with RequireJS and Marionette Modules*[45]:
  This is essentially a sequel to David Sulc's *A Gentle Introduction*. It goes into greater depth about how RequireJS and Marionette's modules can be used to structure and organize your code. If you are looking for a follow-up to this book, *Structuring Backbone Code with RequireJS and Marionette Modules* is probably a good choice.
- *Building Backbone Plugins*[46]:
  This book was written by Derick Bailey himself. It gives useful insight into the way he approaches extending Backbone and provides the reasons behind many of the decisions he made when creating Marionette. This book covers more than just Marionette, though, and strictly speaking isn't a book about Marionette at all. ❧

43. http://www.joezimjs.com
44. https://leanpub.com/marionette-gentle-introduction
45. https://leanpub.com/structuring-backbone-with-requirejs-and-marionette
46. https://leanpub.com/building-backbone-plugins

About The Author



Joseph Zimmerman (@joezimjs[47]) is a web developer for Kaplan Professional and runs his own JavaScript blog[48]. When he isn't feeding his coding obsession, he's spending time with his wife and two little boys.

About Smashing Magazine

Smashing Magazine[49] is an online magazine dedicated to web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy[50].

Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised. Since its emergence back in 2006 Smashing Magazine has proven to be a trustworthy online source.

47. http://www.twitter.com/joezimjs
48. http://www.joezimjs.com
49. http://www.smashingmagazine.com
50. http://www.smashingmagazine.com/publishing-policy/