Backbone Basics: Part I

Chapter 1 Introduction

It's PeepCode! Today you'll learn to write rich client-side applications with Backbone.js.

As web browsers have become more capable, people have started to write more powerful applications with them. But applications have also become more complicated. And as code becomes complicated, it becomes harder to read and maintain, and it becomes harder to fix bugs.

Backbone is a very small amount of code that attempts to solve part of that problem. It implements models and views, and a router that you can use to write thoroughly organized, easily testable, and clearly maintainable client-side code.

Backbone itself is barely over 1,000 lines of code, easy to read through in an hour or two. It doesn't provide any UI widgets. You can use your choice of templating engine. You can use the same HTML and CSS that you're already comfortable with; you don't have to learn a new system for creating the user interface.

Using jQuery alone often results in code that's directly tied to the markup of a document. Backbone starts in the opposite direction. Your applications will start with data models. Connected to them are Views that react when model data is loaded, changed, or deleted. The Router interprets URLs and calls sections of your code. So single-page applications that are URL-aware are not only possible with Backbone, but they are the default.

In this screencast, Part I, we'll learn the fundamentals of Backbone Models, Views, and Routers.

Part II uses those concepts to build out the rest of the application, adding minor features and other tips along the way.

A separate Backbone screencast will cover persistence and networking.

As with most PeepCode screencasts, we'll cover a lot of information in a short amount of time. Use the pause button or replay a section if you need to view it again.

Let's get started!

Chapter 10 App Setup

Here's the application we're going to build: a simple music player. There's a library with some albums. You can click to add one to a playlist. The big "play" button plays a song. It highlights the current album and track that's currently playing. You can switch to the

next track or the previous track. If you go forward far enough it will go to the first song on the next album. You can also pause the music.

Finally, you can delete an album from the playlist.

That's the application we're going to build with Backbone.js.

To start, go to the code download from PeepCode and copy the "0-start" project to a new directory. I'll rename it to "backbone-tunes". Open it in your text editor and let's take a brief look at the files you're starting out with.

We need a simple webserver, so I've added a few files to run the Ruby Sinatra webserver. It will serve the static JavaScript, CSS, and HTML files and even send some JSON data to the Backbone client. To run the server, open a terminal and change into the "backbone-tunes" directory. If you use Ruby and have setup the RVM sandboxing tool, you may want to create a new RVM gemset to store this application's code. This is completely optional and the application will run fine either way.

Run "rake server". It will install any additional dependencies and start up the server.

Due to recent changes in Ruby's supporting libraries, you may need to install a newer version of RubyGems with "sudo gem update --system". If you run into problems, email peepcode@topfunky.com and we'll help you get it up and running.

Go to "localhost" port 9292 to view the website. At the moment nothing is visible, but there's some HTML boilerplate underneath. Right click or ctrl click and pick "inspect element" or "view source" to see the HTML boilerplate. I'll use the Google Chrome developer console throughout this screencast. There's also one included with Safari or you can use Firebug with Firefox.

Let's look at the starter code. Most of the code we'll work with is either in the "public" directory or in the "tests-js" directory. I've provided you with an HTML5 "index.html" page that has some directives for serving content to the various browsers. It includes a few stylesheets, and – importantly for our application – it includes a few JavaScript files.

You don't *need* Modernizr but it's useful if you want to target features available in newer browsers. You *will* need jQuery, Underscore and Backbone. Our application will live in tunes.js.

In the markup there's a template that will be used shortly to render a single Album. I have a blank "div" in the body section of the document that will be populated with content generated by Backbone.

The tunes.js file is very basic right now; it's the blank "jQuery" loading function. We'll write our Backbone classes here.

In the next chapter, we'll start writing code with Backbone.js!

Chapter 12 A Simple Model and View

In this chapter, you'll write a very simple Backbone Model and View.

We'll write only what's needed to get content on the screen. It will be the foundation for the Playlist application that we will be building in the rest of the screencast.

Go to "public/js" and open "tunes.js". There's a bit of boilerplate that will give us access to jQuery and run code when the document's assets have loaded.

MODEL

The fundamental data structure in Backbone is the Model. Models work with pure data, separate from any user interface. This particular model won't store data permanently, but it *will* notify other objects when its attributes are modified. We'll use both features here.

Let's create an Album model that will have a title, an artist, and a number of song tracks which each have a title of their own.

The simplest model can be defined in a single line of code.

Start by using an Album variable. I'll capitalize Album as a convention to show that this is an abstract object which won't store data directly but will be used to create other new Album objects with their own data.

I will extend Backbone.Model to define the Album model. A basic Backbone Model needs no more code than this. You only need to add extra code if you want to augment or customize a model's behavior. In this chapter we won't add any other code to this model.

However, we do want this to be usable in the browser console and from tests in other files, so I'll assign the Album to window.Album. This Model is now available externally.

Go back to the browser. Refresh to get the latest code, and right click or ctrl click to pull up the developer console. Let's use this Album object in the console. Create a new Album. Pass in attributes that this Album should be initialized with. I'll specify a title: "Abbey Road", and an artist: "The Beatles".

What would one want to do with a data model? Query and change that data!

Right away, the Backbone Model provides you with methods to modify its attributes. These are named “get” and “set”. “get” takes a string and returns the value if it exists. I can “get” the title and see that it is “Abbey Road”. The artist correctly returns “The Beatles”.

There are also auxiliary methods that provide extra information about the object. For example, this Album is currently not persistent. It hasn’t been saved anywhere, and it hasn’t been retrieved from any saved value. So it’s a new record. If you’ve used Ruby on Rails, this is similar the basic functionality you’ll see in Ruby’s ActiveRecord.

We can also “set” attributes on a Model. The argument to “set” is different from “get”; it takes a key/value object such as the one we initialized this Model with. But you don’t need to pass *every* attribute, just the ones you want to change. I’ll “set” a single key, changing the title of this Album to “Revolver”.

When debugging an application, it’s useful to be able to look at a model’s data. There are several ways to do that.

One is to call the built-in “toJSON” function which will return all the keys and values for this Backbone Model. In the developer console you can also type the name of the “album” variable, and see the entire Backbone Model, including the attributes that you’ve given to it, and also the other built-in properties and methods.

VIEW

That’s a Backbone Model. Let’s use this Model in a basic View, the other fundamental class in Backbone.

The View is responsible for displaying data. Usually that means taking a model and rendering an HTML template with its data, then appending the resulting HTML to the rest of the document. Views also listen to events such as clicks, drags, and keyboard input. They might re-render themselves on such an event, or even modify their model to reflect the new data.

If you’ve only worked with server-based web frameworks, you might think that the Backbone View does a lot of work that you’re used to seeing in a controller. That’s true, but it’s true for all desktop-style MVC frameworks. In a framework like Rails, someone clicks on a link and your controller re-renders an entire HTML document from scratch. But when you can react immediately to user input on the client, it makes more sense to make views smarter. And that’s what Backbone does.

I’ll use similar syntax to define my AlbumView: Backbone.View.extend.

This time I will need to do bit more work before this view is usable. Backbone provides an “initialize” method which you’ll frequently override to setup the view.

It doesn’t take any arguments.

The Backbone “initialize” method often performs the same few tasks. One is to set up a template that will be rendered later. It will specify the template content to use, often located in your HTML document. It will setup a template engine of your choice. Then this “template” function can be called later in the view’s “render” function to populate that template with the model’s data.

You can use your choice of template engines if you already have a JavaScript-based template engine that you prefer. I’m going to use the one that’s already built into the Underscore library. Let’s look at the source briefly.

In “underscore.js” there is a templateSettings method that specifies default delimiters similar to ASP or Ruby’s ERB: angle bracket and percent will start a bit of logic or data in the template. Then there’s the “template” function which takes a template string and optionally some data to render with.

The raw template for the AlbumView can be found in the HTML document that this Backbone app is running from. Use jQuery to ask for all of the HTML content in the node identified by this ID.

What does the template look like? If you’re using the starter project, I’ve given you an AlbumView template already. It’s specified inside script tags with a type of “text/template”, so the browser doesn’t try to do anything else with it. It has an ID so we can refer to it easily with a jQuery selector.

I’ll print out the title and artist of this Album. I’ll use the underscore library to iterate through each of the tracks and print each of their titles.

That’s a basic view template. Most other views will use similar templates within their own script tags.

Back in the view, I must implement one other method: “render”. Most often this will render a template with some data from an associated Model. Alternately, you could use jQuery to modify a document directly. But most often, a view’s “render” method will render an existing template setup in “initialize”. Let’s implement “render”.

For convenience, I’ll make a local variable that will hold the rendered content. I’ll call my new “template” method with the attribute data from my Model as returned by the model’s built-in “toJSON” method. We’ll see in a moment where that Model comes from.

But that content isn't in the document yet. I'll show the line to do that and then I'll explain it. I'll use jQuery's "html" method to replace all the content in an element with the rendered content from the template.

Backbone Views ship with several built-in, frequently used properties. One of those is "el"; the wrapper element represented by this View. Most often, Backbone will create that element for you automatically and you'll add your own content inside the element in creates for you. Let's do that.

Wrap "this.el" in the jQuery $ function so it's a jQuery object. Set its HTML content with the rendered content from the template.

Finally, each "render" method should return itself so you can chain other calls onto the call to render.

Let's try it out. Go back to the browser and refresh. You still won't see anything visible but we can create some content from the console. Use the up arrow to revisit the history where we created an Album with data. But this time I'll add some tracks since the template expects to find some. The "tracks" attribute will hold an array of key/value objects, which each have a title.

Create a new AlbumView with the new album as its model. Backbone recognizes a few attributes as special. One of those is "model"; if you pass an attribute with a key of "model", Backbone automatically sets a root property of "model" on the view. That's what we used in the view's "render" method when rendering the template.

You may have noticed that even now, nothing is visible. Even though the view's "render" method adds some HTML to its own element, we also need to add *that* element to the rest of the page.

So I'll find the *container* ID that I specified as the root element for my HTML document, and I'll append the rendered element from the Album View to it.

If you're following along and you typed everything correctly, you'll see "Abbey Road", "The Beatles", and one album track all rendered from Model data inside a Backbone View.

In the next chapter, we'll add some styling and wire up the view to redisplay itself when the model is modified.

Chapter 14 View Styling & Rendering

In this chapter, we'll modify the view's properties so it uses the CSS styling we set up previously. And more importantly, we'll use a powerful feature of Backbone: model change events.

STYLING

Backbone Views set up quite a few defaults for you but you can override those when necessary. One is the name of the HTML tag that will be used for the view's HTML element. By default it's a "div". I want this to be a list item because, later on, we'll use several Albums in a list.

You can also specify the CSS "className" to apply to the view's HTML element. I'll ask that it be "album".

Go back to the browser. Refresh the page to get the new code. Use your arrow keys to repeat the creation of an Album model, an AlbumView, and add it to the page. Now you'll see that the content is styled. The styling came from the CSS in the starter project, but the proper tags and CSS class names were generated from the Backbone View.

CHANGE EVENTS

What we've done so far has been interesting, but not revolutionary.

A final basic concept of Backbone Models and Views is change detection. When you call "set" on model, it can notify other objects that it has changed. In the case of a view, it will probably re-render itself with the new data.

For server-side web frameworks, the Controller does most of the work of reacting to user interaction. The controller gets a request to show a record and it pull that record from the database, renders a view, and sends the entire output back to the browser.

Backbone Views are driven by data. Instead of manipulating views, you'll just work with pure data models. Your code will define views that watch specific models and know how to render their data. But you'll rarely ever call "render" on a view. Instead, you'll change a model's data. You'll fetch data from a server, add models to lists of other models, or change a specific attribute. The changes will cascade through the rest of the application, which will display the new or updated data.

If you've only done server-based MVC, I think you'll really like this alternate way of thinking. I think it embodies MVC much more sensibly than the handcuffed version that we have to use on the server.

But it doesn't happen automatically. Here's what we need to do:

- Tell a model to notify a view when specific events happen
- Implement custom view methods that update the view accordingly

First, a bit of housekeeping. Because of the nature of the JavaScript language, we need to make sure that the view's "render" method can be called in a way that associates it with

the current view object. Most of your Backbone Views will need to start out with this line: _.bindAll(). Use "this" as the object to bind to, and then list all the methods that will be used as callbacks. Right now, it's just the "render" method.

Next, I want to be notified when the Model changes. Models can fire events when any attribute in the model changes, or when specific attributes change, such as the "title". The coarsest event is the "change" event. If any of the attributes in this Model change by being set with the "set" method, it will trigger the "change" event and call the method we specify here. Complex views might want to change only a part of their view (for efficiency), but we'll start by bluntly re-rendering the entire view.

Those two lines are all that's needed. Let's try it out.

Refresh. And, again, create an Album and an AlbumView, and add the view to the container. That so far is clear. Next, set a new value on the album. If it all worked correctly, you should see the new value automatically displayed in the AlbumView.

I'll set a different artist. As soon as you hit "enter" you'll see that the rendered AlbumView was automatically updated with the new value. We didn't have to touch AlbumView directly. We didn't call "render". We didn't mess around with any jQuery. We just changed the data in the model, and the AlbumView automatically re-rendered itself to display the new data.

For further proof, change other attributes in the model, such as the "title". For complex data structures such as the array of objects assigned to "tracks", Backbone only fires a change event if the root element is changed. A modification to a track's "title" would *not* fire the "change" event. Only a re-assignment of the entire "tracks" array will fire the change event. But most of the time, your Models will be built from root level keys and values and this won't be a problem.

Type just the word "album" and you can inspect the contents of the Backbone Album Model that we've created. The attributes contain the "title", "artist", and "tracks", which show the new values.

Those are the basic features of the Backbone Model and View. Models contain data which can be modified. Views initialize and render themselves. Models can notify Views when they change.

In the next chapter, we'll take a brief excursion into testing Backbone models with the Jasmine test framework.

Chapter 14 Jasmine Specs

In this chapter we'll look briefly at Jasmine, a test framework.

Backbone Basics: Part I

One of the best things about working with a Model View Controller system like Backbone is that you can easily write tests for your code. A system I've started using recently is called Jasmine. It recreates parts of the RSpec framework available for Ruby.

The logic for moving forward and backward through multiple tracks and albums was complicated and had a few bugs. I think it would have taken me much longer to figure it out if I had not used Jasmine to help me. I was able to write a test for each data configuration and run them all together to arrive at an implementation that satisfied all of the edge cases.

We won't talk much about Jasmine in this screencast, but I want to show you how easy it is to use, especially for your Backbone models.

I've included Jasmine in the "test-js" directory. You can create custom specifications in the "spec" directory. I've created one already for you: "TunesSpec.js". The "SpecRunner.html" file lists the code files and runs the tests. Drag that to your browser to run it. One spec has already been implemented for you. It doesn't require any kind of webserver. It just runs straight off an HTML file.

Look at the markup in the "SpecRunner.html" file. It includes all the JavaScript files that we need to run Backbone. I've also included Tunes.js which has the code for our Backbone application. It also lists "TunesSpec.js". If you create other spec files, add them here.

In "TunesSpec.js" I've set up some sample Album data that you can write tests against. There are two albums each which have titles, artists and a few tracks. I've made it straightforward by naming them Album A, Artist A; and Album B, Artist B.

Jasmine tests are made of nested calls to "describe" and "it". Each takes a description and a function. The "describe" function usually describes a class. The "it" usually describes a specific method or use case for that class. Other calls to "describe" can happen within a "describe" function (for organization).

I'll start by describing the Album Model. It will set up a new Album from the sample data and then run expectations on it.

Let's write a very simple expectation to see how this workflow happens. I'm going to write a new "describe" block where I will write examples associated with trying to find the first track of an album. When we build this whole application, we'll want to be able to switch through the different tracks and switch through different Albums. As part of that, we need to know if we're on the first track, or the last track, or neither, which would mean that we're on a track in the middle of an Album.

I'll specify that I want to be able to identify the correct first track. The method to find this information hasn't been written yet but will be called "isFirstTrack". It will take a single parameter: a number that represents an index in an array. It will return true or false based on whether that is indeed the first track. I expect that track zero is in fact the first track. Jasmine's matcher for a true result is "toBeTruthy". It should return a true value.

Go back to the browser and refresh the page for the spec runner. We haven't written the "isFirstTrack" method yet, so an error is shown. That's what we expect at this point in the workflow. We expect to see an error that proves this to be a useful test. Then we can implement the method.

Up until now the Album Model hasn't had any custom methods. But now we can write one in the same style that we used to add methods to a Backbone View. It will take a single argument, the index. It should check to see if that index is zero, so it's a very simple comparison.

Go back to the browser and refresh. Now it passes because it did identify track 0 as the first track. We could do a similar thing for the last track. In my sample data there are two tracks so the track at index 1 is the last track. Here's what an example for that would look like.

The implementation will be this method, which compares the the total number of tracks to the provided index and of course the array of tracks. The array is zero indexed, so it will have to subtract 1 from the total number of tracks in order to find out if the index represents the last track in this album.

Another custom function we'll need to use later is one that will find the URL to the music file for this track on this album. And it will similarly work with an index. I've specified track URLs in the sample data, so I'll attempt to match the exact name of an MP3 file. I'd expect the track URL at index 0 to be "Album A Track A.mp3".

Checking the browser shows that it fails again as expected because I haven't implemented it yet.

For the implementation I'll use the "get" method to find the number of tracks. If there are as many tracks as requested by the index, I'll find the track at that index and return its URL. Otherwise I'll return null.

Back in the browser, it passes.

When I was writing, debugging, and refactoring this application, the ability to write simple tests made a huge improvement in my ability to wrap my mind around the kinds

of things that needed to happen in the code and then ensure that they did in fact happen. This was especially true when trying to move between tracks and across Albums in a Playlist, such as properly wrapping around from the last track of one Album to the first track of another.

We'll only briefly mention tests from here on out. But the completed application includes quite a few tests. I encourage you to look at them and even try writing your own for this or other applications.

In the next chapter, we'll add to the fundamentals by introducing Collections: arrays of model objects.

Chapter 16 Collections

In this chapter, we'll explore the basic features of Backbone Collections.

Models are useful on their own but they become much *more* useful when they're part of a Collection. Collections are also the standard way to load a group of models from data residing on a server. In fact, we'll do just that in this chapter.

Collections manage groups of models. If you're used to Ruby's implementation of ActiveRecord, Collections serve the same role as class methods in ActiveRecord: finding model records, loading them, searching them.

You'll need a Collection for each kind of Model you define.

As you might guess from seeing the implementation of Models and Views, Backbone Collections are be defined with Backbone.Collection.extend. I'll make one and use the plural Albums to refer to it.

A nice convention is to use the plural to refer to Collections (Albums) and the singular to refer to Models (Album).

Collections must define at least one property: the "model" they manage. It won't do any introspection to try to automatically match up to a Model of the same name.

As you may guess, this collection manages the Album model. When this Backbone collection receives new data, it will make a bunch of Album instances.

Second, there is the URL from which this data will be fetched. The easiest strategy is to use something like "/albums". If you're using Ruby on Rails on the server, most of Backbone's defaults work directly with the flavor of REST used by Rails.

The URL "/albums" will be used to load the records in this collection. Models created with it will also get a URL property similar to "/albums/id" which can be used to save Models back to the server.

So the simplest Collection has "model" and "url" properties. For convenience and consistency, I've set up the server portion of this application so if you request "/albums" you'll get back "public/albums.json". If you want to see different album data, edit that JSON file.

Here's what "albums.json" looks like. This is the format Backbone will look for by default. It's an array of key/value objects: "title", "artist", and multiple tracks for each album. In a production application you would also have an ID for each record which Backbone would use for persistence.

Let's use this collection. Go to the browser and refresh it to load the new code. You won't see the list of albums anymore, but we can work with a collection in the console. Let's make a new Albums collection and ask it to "fetch". The "fetch" method will make a request to the server and asynchronously populate itself with content.

You can see the Models it retrieved by looking at "albums.models". There are two. Inspecting them shows one with the attributes of an album from the "Zen Bound" in-game music from the iPhone game. It has several tracks, each with a title the URL of the music file on disk. The second album is from one of my favorite albums, the soothing ambient tuba sounds of Tom Heasley.

Now we can work with the models in this collection. Any of the built-in Underscore.js methods can be called directly on the collection. They will return records from the array of Models. So you don't need to address the "models" array directly. Here's an example: albums.map will process each of the models and return an array with the return value of your custom function. I'll get a handle on each with the function's parameter: "album". Whatever we return from the function will be returned as an array from the "map" function. For the example, I'll return the "title" from each individual Album. The result looks like this.

A shortcut is to use the "pluck" method which does the same thing.

Like Models, Collections also fire events when records are added, removed, or loaded in bulk.

In the next chapter we'll write a View that renders the albums in this Collection.

Backbone Basics: Part I

Chapter 18 Collection Views

We've worked with Models and a simple View. We've discovered Collections. In this chapter, we'll use a collection to drive a View.

Here's what we'll create by the end of this chapter; a section that displays the entire music library with title and artist for each album. As before, our code will reside in Tunes.js.

Before we implement this, let's think briefly about Views. Working with Backbone Views is much more similar to working with the Views of a desktop application than it is to a server-driven web application. With a standard web application, you might have a series of templates that are combined into a single HTML document and are delivered to the browser.

With Backbone, some Views can have other Views nested inside them. A Collection-backed View works exactly this way; we'll define a View for the Collection. Each of the items in the Collection will be rendered using their own View class.

Let's start by defining the View class that will render a single Album in the library. Thanks to object inheritance it only requires a single line of code. We already have the AlbumView that knows how to display an album. We could use that directly, but I know that I'll need to customize it in the future, so I'll make a new LibraryAlbumView class that extends the existing AlbumView.

This works the same as if we were extending Backbone.View. Since Album View already extends Backbone.View, we can extend AlbumView and we'll automatically get all of the "initialize" and "render" functionality defined in AlbumView. It will also use the existing "album-template" defined in "index.html".

So nothing else needs to be written at the moment for LibraryAlbumView. But we *will* need a wrapper view to coordinate the display of each of the albums in the library.

I'll just name this LibraryView. Again it inherits from Backbone.View (there's no special view type for Collections). We will start with an "initialize" method that will look like the one you've already seen. I'll start by binding the "render" method so we can use it as a callback to a binding. I'll again make a template based on content defined in the HTML document: ID "library-template".

We haven't implemented that template yet so let's do it. Go over to "index.html". Start a script tag but rename it to "library-template". Most templates will display and iterate through Model attributes. Others like this one will be extremely simple and won't have any logic or even data in them.

The “library-template” needs a header tag to tell people that this is our music library. It has an unordered list with a class of “albums” that will contain an AlbumView for each of the albums in our music library. We will populate those from the “render” method of the LibraryView.

Back in Tunes.js, let’s write the “render” method. One thing you’ve already seen is that the “render” method often looks at it’s DOM element and renders a template with some data. Because our template doesn’t use data from any Collection or Model, it will be much simpler. I’ll render the template with a blank object since no data is needed by the template.

We could even skip the whole idea of rendering a template and just ask it to use the HTML directly from the library template. However, sending it through the template engine is pretty quick and will leave open the option to add view logic to the template in the future, if needed.

Next, let’s define a few variables for readability. I want to find the unordered list element from the current element’s markup and insert some rendered Views into it. I also want to work with the Collection which I’ll make slightly more convenient with a local variable.

By convention, variables which reference jQuery objects often begin with a “$”, so use that in the name of the albums variable; it will refer to the “albums” list in the document. Here’s a useful feature of Backbone views: If I call jQuery methods on “this”, they will automatically be scoped to the current element. So instead of searching the entire document for the “albums” list, I can quickly search within the current element for this view.

Next, loop through the Models in the collection and render a LibraryAlbumView for each. As we saw in the previous chapter, I can use “each” directly on the collection and Backbone will run it on the Collection’s models.

What do we need to do with each album? We need to make a new LibraryAlbumView. For the purposes of some code that we’ll write later on, I’ll give the View both a Model and a Collection. The LibraryAlbumView will have access to both. Alternately, Models initialized from a Collection have a “collection” method that returns their Collection. But passing it in this way is a clear, explicit way to show what’s happening.

On its own, a View doesn’t insert itself into the DOM, the HTML document. So I will manually do that using my existing “$albums” variable. I’ll append the rendered version of this LibraryAlbumView. And as is conventional, I will return “this” from the “render” method.

We're almost there. We need to trigger a call to "render" the LibraryView after the Collection's data has been loaded.

In the AlbumView, the object to bind to was obvious: the model's "change" event.

There's no "change" event on Collections. The main Collection event that fires when a group of models is loaded is "reset" (this is a new name as of Backbone 0.5).

Monitoring the Collection is the best strategy since the data will be loaded asynchronously. Instead of manually rendering the template when *we* load the data, we'll just ask the Collection to load its data. The LibraryView will observe the Collection. When the Collection's data has arrived, the Collection will fire the "reset" event and the view's "render" method will be called.

It's good to become familiar with these events so you can hook into them and run your own code when needed.

For proper styling, let's also specify a "tag" and a "className" to match up with the CSS in the starter project. I'll use the HTML5 "section" tag for this element and the class "library".

We're still manually adding views to the page, for this one last time.

Refresh the browser. In the console, make a new "library" from our Albums collection. Make a new LibraryView and pass the "library" we just made as its collection. Append the LibraryView to the document. I'll ask it to render, even though there's no content to display yet. We *do* get to see the "Music Library" title.

Let's trigger the model's "reset" event. The public collection method to get the data and trigger that event is "fetch".

Call "library.fetch". If it worked correctly you will see both albums listed: "title" and "artist".

It may seem odd that the view bound itself to the "reset" event but we called the "fetch" method on the collection. It turns out that there are several ways you can load a collection with fresh data. "fetch" will attempt to make a request to the specified server URL and then trigger "reset" afterward. Another way is to pass an array of key/value objects straight to the "reset" method. This is the preferred technique if you're generating the HTML document dynamically on the server. You can deliver both HTML and Backbone data to the client in a single response. This means that as soon as the document loads, people will see the application pre-populated with live data. They won't have to wait for a second request to get data from the server.

This can provide a much better, quicker user experience.

Now you know how to define collections and render them in a view.

In the next chapter we'll look at the router, the last of Backbone's fundamental objects.

Chapter 19: Router

In this chapter, you'll learn how to run your custom Backbone code based on the browser's URL.

One of the problems with JavaScript-powered applications is state. People are used to being able to bookmark a page, reload it the next day, and see the same thing they saw before. It's hard to do that with JavaScript applications that are dynamically built.

At least it *was* difficult before the Backbone Router. The Router is about 200 lines of code that looks at the current URL and calls a corresponding method. You can implement that method to put your application into a predictable state. For example, I have a TODO list that has the date in the URL. I use the URLs to move between tasks for specific days.

The Router is useful for another reason. So far we've created objects in the console, then manually placed views into the DOM. But that's a bit much to ask your customers to do that. By using the Router, we can automatically load data and render templates when a person visits the page.

In previous versions, the Backbone Router was named "Controller" but it has been renamed "Router" to more accurately reflect what it does. Other than the name change, the functionality is the same.

Before we can use the router, let's do a bit of code cleanup and organization. For convenience I'll instantiate a Backbone Collection for the entire list of albums in the "library". Most often, you'll use a single instance of each Collection, so it makes sense to create a new one and set it to a variable that can be used elsewhere in your code.

In order to be clearer about what this represents, I'll call it the "library" and make it globally available by assigning it to "window".

Next, I'll fetch the album data when the page loads.

As mentioned previously, you'll deliver the best user experience if you serve HTML and initial data in a single response. In this case, we don't have that luxury. So I'll do the next best thing. I'll use jQuery to detect when the page has loaded, and I'll call

"window.library.fetch()" to retrieve the data for all the albums. Using "window." is optional, but more clearly communicates where the "library" variable comes from.

Now we're ready to work with the Backbone Router. Let's start by defining a Backbone Router that will setup views for the root page of the application.

I'll call my main router BackboneTunes. A full featured application may have several Routers that each deal with separate URLs, so in a larger application you'll want to name them based on what they do: AdminRouter, SignupRouter, ReportRouter.

Start by extending Backbone.Router, as we did with Model, View, and Collection. Then define a route. The "route" property is an object where the keys are URL paths and the values are methods to call when that URL is visited. A router class will consist of a list of routes, an "initialize" method, and route-handling methods for each URL path.

The most important route is the empty string. This will be called when someone visits the home page of the application. I'll specify that my router's "home" method should be called.

Like the other objects, the Router can optionally implement an "initialize" method. It was only after writing a few Backbone applications that I understood what code should go where in a Router.

Here's the situation that has worked best for me:

The "initialize" method should instantiate the application's root-level view. For most designs, this will be a single view that will create its own subviews from a Collection's data. In this Playlist application, there are two root level views: the PlaylistView (on the left) and the LibraryView (on the right, already implemented).

But the "initialize" method should *not* insert those views into the DOM yet. Individual route handler methods should do that. They need to clear out any other content that's already in the document and replace them with the current route's views. Or, they may be able to reuse existing views. For example, I wrote a TODO list application that displays information for different days. One handler method just re-fetches Collection data for the specified day and lets the View re-render itself with the new data.

For this application, it's much simpler. We will initialize the LibraryView in the "initialize" method and add it to the document in the "home" method.

I'll set the LibraryView's collection to window.library.

In the "home" handler, find the root container in the HTML document. Use jQuery to empty it out so I can start with a fresh state. Then append the LibraryView, being sure to call "render" and pass in its element as we've done several times already.

That is the definition of a simple Backbone Router.

Our Router will not be very useful unless we make an instance of it. Let's use the jQuery "$" function to make a new instance of the Router as soon as the document loads. I'll make a new instance of BackboneTunes and assign it to window.app for future reference. Then kick it off by calling "Backbone.history.start()".

By default this will use a hash mark in the URL, like Twitter. As of 0.5, you can pass "{pushState: true}" as an option to use a feature of newer browsers that uses standard URL paths instead of the hash mark. Using the hash mark is a bit of a hack and doesn't work well with existing web servers, search engines, and the ways people link to things. But older browsers don't support pushState. So if your clients use newer browsers, definitely try using the "pushState" option. See this article by Dan Webb for the philosophy behind using traditional URL paths. http://danwebb.net/2011/5/28/it-is-about-the-hashbangs

Let's try this out. For the first time so far, we can refresh the page and, if it all worked correctly, you should see the music library on the screen with a few albums.

This doesn't really illustrate the flexibility or power of the Backbone Router. Let's do a simple extra route to show how this works. I'll make one named "blank" which will call the "blank" method in the Router. As before I'll clear out the root container with the jQuery "empty" method. For illustration, I will set some text in the container so we can see a visual change when a different URL is visited.

Refresh the browser again and go to "#blank". You'll see the "blank" text on the page.

Go back to the root URL and you'll see the content of our music library. Backbone will automatically route requests from the URL to the proper function in the Router. You can develop complex applications easily using only this simple functionality.

It can also be navigated programmatically. Open the console and use the App variable that we set the router to. "App.navigate('blank', true)" with "true" as the second argument will change the URL and trigger the "blank" method in the router (by default it just changes the URL but doesn't trigger the method). Go back home with a blank string as the destination: "App.navigate('', true)".

Or, enable pushState and try it out with slashes instead of the hash.

That's the Backbone Router. A simple but effective way to drive the state of your application with URLs.

Conclusion to Part II

In Part I, you've learned how to use the basic objects in Backbone.js. In Part II we'll use these concepts and others to build out the rest of this application.