1 Introduction

This is version 3.0 of **The Backbone Store**, a brief tutorial on using `backbone.js`. The version you are currently reading has been tested with the latest versions of the supporting software as of April, 2016.

Backbone.js[1] is a popular Model-View-Controller (MVC) library that provides a framework for creating data-rich, single-page web applications. It provides (1) a two-layer scheme for separating data from presentation, (2) a means of automatically synchronizing data with a server in a RESTful manner, and (3) a mechanism for making some views bookmarkable and navigable.

Backbone is dependent upon jQuery[2] and Underscore[3]. Both of those dependencies are encoded into the build process automatically.

The version of this tutorial you are currently reading uses Coffeescript, Stylus, and Ruby's HAML. The purpose of this tutorial is to show how to use Backbone in a modern, constrained programming environment.

CoffeeScript[4] is a lovely little languange that compiles into Javascript. It provides a class-based architecture (that is compatible with Backbone), has an elegant structure for defining functions and methods, and strips out as much extraneous punctuation as possible. Some people find the whitespace-as-semantics a'la Python offputting, but most disciplined developers already indent appropriately anyway.

HAML[5] is a languange that compiles into HTML. Like CoffeeScript, it uses whitespace for semantics: indentation levels correspond to HTML containerizations. It allows you to use rich scripting while preventing heirarchy misplacement mistakes. Its shorthand also makes writing HTML much faster.

Stylus[6] is languange that compiles into CSS. Like CoffeeScript and HAML, it uses whitespace for semantics. It also provides mixins and functions that allow you to define visual styles such as borders and gradients, and mix them into specific selectors in the CSS rather than having to write them into the HTML.

There are a number of other good tutorials for Backbone (See: Meta Cloud[7], &Yet's Tutorial[8], Backbone Mobile[9] (which is written in Coffee[10]), and Backbone and Django[11]. However, a couple of years ago I was attempting to learn Sammy.js, a library very similar to Backbone, and they had a nifty tutorial called The JsonStore[12].

In the spirit of The JSON Store, I present The Backbone Store.

[1]See URL `http://documentcloud.github.com/backbone/`.
[2]See URL `http://jquery.com`.
[3]See URL `http://underscorejs.org`.
[4]See URL `http://jashkenas.github.com/coffee-script/`.
[5]See URL `http://haml-lang.com/`.
[6]See URL `https://github.com/LearnBoost/stylus/`.
[7]See URL `http://www.plexical.com/blog/2010/11/18/backbone-js-tutorial/`.
[8]See URL `http://andyet.net/blog/2010/oct/29/building-a-single-page-app-with-backbonejs-undersc/?utm_source=twit`
[9]See URL `http://bennolan.com/2010/11/24/backbone-jquery-demo.html`.
[10]See URL `http://jashkenas.github.com/coffee-script/`.
[11]See URL `http://joshbohde.com/2010/11/25/backbonejs-and-django/`.
[12]See URL `http://code.quirkey.com/sammy/tutorials/json_store_part1.html`.

1.1 Literate Program

A note: this article was written with the Literate Programming[13] toolkit Noweb[14]. Where you see something that looks like ⟨⟨this⟩⟩, it's a placeholder for code described elsewhere in the document. Placeholders with an equal sign at the end of them indicate the place where that code is defined. The link (U->) indicates that the code you're seeing is used later in the document, and (<-U) indicates it was used earlier but is being defined here.

1.2 Revision

This is version 3.0 of *The Backbone Store*. It includes several significant updates, including the use of both NPM and Bower to build the final application.

1.3 The Store: What We're Going to Build

To demonstrate the basics of Backbone, I'm going to create a simple one-page application, a store for record albums, with two unique views: a list of all products and a product detail view. I will also put a shopping cart widget on the page that shows the user how many products he or she has dropped into the cart. I'll use some simple animations to transition between the catalog and the product detail pages.

1.4 Models, Collections, and Controllers

Backbone's data layer provides two classes, `Collection` and `Model`.

Every web application has data, often tabular data. Full-stack web developers are (or ought to be) familiar with the *triples* of addressing objects on the web: Table URL ☐ Row ☐ Field, or Page URL ☐ HTML Node ☐ Content. The `Collection` object represents just that: a collection of similar items. The `Model` represents exactly one of those items.

To use the Model, you inherit from it using Backbone's own `.extend()` class method, adding or replacing methods in the child object as needed. For our purposes, we have two models: `Product` represents something we wish to sell, and `Item` represents something currently in the customer's shopping cart.

Shopping carts are a little odd; the convention is that `Item` is not a single instance of the product, but instead has a reference to the product, and a count of how many the buyer wants. To that end, I am adding two methods that extend Item: `.update()`, which changes the current quantity, and `.price()`, which calculates the product's price times the quantity:

2 ⟨*models* 2⟩≡ (16)
```
class Product extends Backbone.Model
```

[13]See URL `http://en.wikipedia.org/wiki/Literate_programming`.
[14]See URL `http://www.cs.tufts.edu/~nr/noweb/`.

```
class Item extends Backbone.Model
    update: (amount) ->
        return if amount == @get 'quantity'
        @set {quantity: amount}, {silent: true}
        @collection.trigger 'update', @

    price: () ->
        @get('product').get('price') * @get('quantity')
```

The methods `.get(item)` and `.set(item, value)` are at the heart of Backbone.Model. They're how you set individual attributes on the object being manipulated. Notice how I can 'get' the product, which is a Backbone.Model, and then 'get' its price.

Backbone supplies its own event management toolkit. Changing a model triggers various events, none of which matter here in this context so I silence the event, but then I tell the Item's Backbone.Collection that the Model has changed. Events are the primary way in which Backbone objects interact, so understanding them is key to using Backbone correctly.

Collections, like Models, are just objects you can (and often must) extend to support your application's needs. Just as a Model has `.get()` and `.set()`, a Collection has `.add(item)` and `.remove(id)` as methods. Collections have a lot more than that.

Both Models and Collections also have `.fetch()` and `.save()`. If either has a URL, these methods allow the collection to represent data on the server, and to save that data back to the server. The default method is a simple JSON object representing either a Model's attributes, or a JSON list of the Collection's models' attributes.

The `Product.Collection` will be loading it's list of albums via these methods to (in our case) static JSON back-end.

3 ⟨*product collection* 3⟩≡ (16)

```
class ProductCollection extends Backbone.Collection
    model: Product

    initialize: (models, options) ->
        @url = options.url

    comparator: (item) ->
        item.get 'title'
```

The `.model` attribute tells the `ProductCollection` that if `.add()` or `.fetch()` are called and the contents are plain JSON, a new `Product` Model should be initialized with the JSON data and that will be used as a new object for the Collection.

The `.comparator()` method specifies the per-model value by which the Collection should be sorted. Sorting happens automatically whenever the Collection receives an event indicating its contents have been altered.

The `ItemCollection` doesn't have a URL, but we do have several helper methods to add. We don't want to add Items; instead, we want to add products as needed, then update the count as requested. If the product is already in our system, we don't want to create duplicates.

4a ⟨*cart collection* 4a⟩≡ (16) 4b ▷

```
class ItemCollection extends Backbone.Collection
    model: Item
```

First, we ensure that if we don't receive an amount, we at least provide a valid *numerical* value to our code. The `.detect()` method lets us find an object in our Collection using a function to compare them; it returns the first object that matches.

If we find the object, we update it and return. If we don't, we create a new one, exploiting the fact that, since we specified the Collection's Model above, it will automatically be created as a Model in the Collection at the end of this call. In either case, we return the new Item to be handled further by the calling code.

4b ⟨*cart collection* 4a⟩+≡ (16) ◁ 4a 5 ▷

```
    updateItemForProduct: (product, amount) ->
        amount = if amount? then amount else 0
        pid = product.get 'id'
        i = this.detect (obj) -> (obj.get('product').get('id') == pid)
        if i
            i.update(amount)
            return i

        @add {product: product, quantity: amount}
```

And finally, two methods to add up how many objects are in your cart, and the total price. The first line creates a function to get the number for a single object and add it to a memo. The second line uses the `.reduce()` method, which goes through each object in the collection and runs the function, passing the results of each run to the next as the memo.

5 ⟨*cart collection* 4a⟩+≡ (16) ◁4b

```
getTotalCount: () ->
    addup = (memo, obj) -> memo + obj.get 'quantity'
    @reduce addup, 0

getTotalCost: () ->
    addup = (memo, obj) -> memo + obj.price()
    @reduce addup, 0
```

1.5 Views

Backbone Views are simple policy objects. They have a root DOM element, the contents of which they manipulate and to which they listen for events, and a model or collection they represent within that element. Views are not rigid; it's just Javascript and the DOM, and you can hook external events as needed.

More importantly, if you pass a model or collection to a View, that View becomes sensitive to events *within its model or collection*, and can respond to changes automatically. This way, if you have a rich data ecosystem, when changes to one data item results in a cascade of changes throughout your datasets, the views will receive ``change'' events and can update themselves accordingly.

In a way, a View can be thought of as two separate but important sub-programs, each based on events. The first listens to events from the DOM, and forwards data-changing events to associated models or collections. The second listens to events from data objects and re-draws the View's contents when the data changes. Keeping these separate in your mind will help you design Backbone applications successfully.

I will show how this works with the shopping cart widget.

To achieve the `fadeIn/fadeOut` animations and enforce consistency, I'm going to do some basic object-oriented programming. I'm going to create a base class that contains knowledge about the main area into which all views are rendered, and that manages these transitions.

With this technique, you can do lots of navigation-related tricks: you can highlight where the user is in breadcrumb-style navigation; you can change the class and highlight an entry on a nav bar; you can add and remove tabs from the top of the viewport as needed.

6 ⟨*base view* 6⟩≡ (16) 7a ▷

```
class _BaseView extends Backbone.View
    parent: $('#main')
    className: 'viewport'
```

The above says that I am creating a class called `BaseView` and defining two fields. The first, 'parent', will be used by all child views to identify into which DOM object the View root element will be rendered. The second defines a common class we will use for the purpose of identifying these views to jQuery. Backbone automatically creates a new `DIV` object with the class 'viewport' when a view constructor is called. It will be our job to attach that `DIV` to the DOM. In the HTML, you will see the `DIV#main` object where most of the work will be rendered.

As an alternative, the viewport object may already exist, in which case you just find it with a selector, and the view attaches itself to that DOM object from then on.

7a ⟨*base view* 6⟩+≡ (16) ◁6 7b▷

```
initialize: () ->
    @el = $(@el)
    @el.hide()
    @parent.append(@el)
    @
```

The method above ensures that the element is rendered, but not visible, and contained within the `DIV#main`. Note also that the element is not a sacrosanct object; the Backbone.View is more a collection of standards than a mechanism of enforcement, and so defining it from a raw DOM object to a jQuery object will not break anything.

Next, we will define the hide and show functions.

Note that in Coffeescript, the => operator completely replaces the `.bind()` function provided by modern Javascript.

7b ⟨*base view* 6⟩+≡ (16) ◁7a

```
hide: () ->
    dfd = $.Deferred()
    if not @el.is(':visible')
        return dfd.resolve()
    @el.fadeOut('fast', () -> dfd.resolve())
    dfd.promise()

show: () ->
    dfd = $.Deferred()
    if @el.is(':visible')
        return dfd.resolve()
    @el.fadeIn('fast', () -> dfd.resolve())
    dfd.promise()
```

Deferred is a feature of jQuery called ``promises". It is a different mechanism for invoking callbacks by attaching attributes and behavior to the callback function. By using this, we can say thing like ``*When* everything is hidden (when every deferred returned from **hide** has been resolved), *then* show the appropriate viewport." Deferreds are incredibly powerful, and this is a small taste of what can be done with them.

Before we move on, let's take a look at the HAML we're going to use for our one-page application. The code below compiles beautifully into the same HTML seen in the original Backbone Store.

8 ⟨*index.haml* 8⟩≡

```
!!! 5
%html{:xmlns => "http://www.w3.org/1999/xhtml"}
  %head
    %title The Backbone Store
   %link{:charset => "utf-8", :href => "jsonstore.css", :rel => "stylesheet", :type => "text/c
    ⟨product list template 10a⟩
    ⟨product detail template 12a⟩
    ⟨cart template 13b⟩
    </head>
  %body
    #container
      #header
        %h1
          The Backbone Store
        .cart-info
      #main
    %script{:src => "lib/jquery.js", :type => "text/javascript"}
    %script{:src => "lib/underscore.js", :type => "text/javascript"}
    %script{:src => "lib/backbone.js", :type => "text/javascript"}
    %script{:src => "store.js", :type => "text/javascript"}
```

It's not much to look at, but already you can see where that `DIV\#main` goes, as well as where we are putting our templates. The `DIV\#main` will host a number of viewports, only one of which will be visible at any given time.

Our first view is going to be the product list view, named, well, guess. Or just look down a few lines.

This gives us a chance to discuss one of the big confusions new Backbone users frequently have: *What is `render()` for?*. Render is not there to show or hide the view. `Render()` is there to *change the view when the underlying data changes*. It renders the data into a view. In our functionality, we use the parent class's `show()` and `hide()` methods to actually show the view.

That call to `.prototype` is a Javascript idiom for calling a method on the parent object. It is, as far as anyone knows, the only way to invoke a superclass method if it has been redefined in a subclass. It is rather ugly, but useful.

9 ⟨*product list view* 9⟩≡ (16)

```
class ProductListView extends _BaseView
    id: 'productlistview'
    template: $("#store_index_template").html()

    initialize: (options) ->
        _BaseView.prototype.initialize.apply @, [options]
        @collection.bind 'reset', @render.bind @

    render: () ->
        @el.html(_.template(@template)({'products': @collection.toJSON()}))
        @
```

That `_.template()` method is provided by undescore.js, and is a full-featured, javascript-based templating method. It's not the fastest or the most feature-complete, but it is more than adequate for our purposes and it means we don't have to import another library. It vaguely resembles ERB from Rails, so if you are familiar with that, you should understand this fairly easily.

There are many different ways of providing templates to Backbone. The most common, especially for small templates, is to just include it as an inline string inside the View. The *least* common, I'm afraid, is the one I'm doing here: using the <script> tag with an unusual mime type to include it with the rest of the HTML. I like this method because it means all of my HTML is in one place.

For much larger programs, those that use features such as Require.js[15], a common technique is to keep the HTML template fragment in its own file and to import it using Require's ``text'' plugin.

Here is the HAML for our home page's template:

10a ⟨*product list template* 10a⟩≡ (8)
```
%script#store_index_template(type="text/x-underscore-tmplate")
  %h1 Product Catalog
  %ul
    <% for(i=0,l=products.length;i<l;++i) { p = products[i]; %>
    %li.item
      .item-image
        %a{:href => "#item/<%= p.id %>"}
          %img{:src => "<%= p.image %>", :alt => "<%= p.title %>"}/
      .item-artist <%= p.artist %>
      .item-title <%= p.title %>
      .item-price $<%= p.price %>
    <% } %>
```

One of the most complicated objects in our ecosystem is the product view. It actually does something! The prefix ought to be familiar, but note that we are again using `\#main` as our target; we will be showing and hiding the various `DIV` objects in `\#main` again and again.

The only trickiness here is twofold: the means by which one calls the method of a parent class from a child class via Backbone's class heirarchy, and keeping track of the ItemCollection object, so we can add and change items as needed.

10b ⟨*product detail view* 10b⟩≡ (16) 11a ▷
```
class ProductView extends _BaseView
    className: 'productitemview'
    template: $("#store_item_template").html()
    initialize: (options) ->
        _BaseView.prototype.initialize.apply @, [options]
        @itemcollection = options.itemcollection
```

[15] See URL `http://requirejs.org/`.

There are certain events in which we're interested: keypresses and clicks on the update button and the quantity form. (Okay, ``UQ'' isn't the best for ``update quantity''. I admit that.) Note the peculiar syntax of ``EVENT SELECTOR'': ``methodByName'' for each event.

Backbone tells us that the only events it can track by itself are those that jQuery's ``delegate'' understands. As of 1.5, that seems to be just about all of them.

11a ⟨*product detail view* 10b⟩+≡ (16) ◁10b 11b▷

```
events:
    "keypress .uqf" : "updateOnEnter"
    "click .uq"     : "update"
```

And now we will deal with the update. This code ought to be fairly readable: the only specialness is that it's receiving an event, and we're ``silencing'' the call to `cart.add()`, which means that the cart collection will not publish any events. There are only events when the item has more than zero, and that gets called on `cart_item.update()`.

In the original tutorial, this code had a lot of responsibility for managing the shopping cart, looking into it and seeing if it had an item for this product, and there was lots of accessing the model to get its id and so forth. All of that has been put into the shopping cart model, which is where it belongs: *knowledge about items and each item's relationship to its collection belongs in the collection.*

Look closely at the `update()` method. The reference `@\$` is a special Backbone object that limits selectors to objects inside the element of the view. Without it, jQuery would have found the first input field of class 'uqf' in the DOM, not the one for this specific view. `@\$('.uqf')` is shorthand for `$('uqf', @el)`, and helps clarify what it is you're looking for.

11b ⟨*product detail view* 10b⟩+≡ (16) ◁11a 11c▷

```
update: (e) ->
    e.preventDefault()
    @itemcollection.updateItemForProduct @model, parseInt(@$('.uqf').val())

updateOnEnter: (e) ->
    @update(e) if e.keyCode == 13
```

The render is straightforward:

11c ⟨*product detail view* 10b⟩+≡ (16) ◁11b

```
render: () ->
    @el.html(_.template(@template)(@model.toJSON()));
    @
```

The product detail template is fairly straightforward. There is no `underscore` magic because there are no loops.

12a ⟨*product detail template* 12a⟩≡ (8)
```
%script#store_item_template(type= "text/x-underscore-template")
  .item-detail
    .item-image
      %img(src="<%= large_image %>" alt="<%= title %>")/
    .item-info
      .item-artist <%= artist %>
      .item-title <%= title %>
      .item-price $<%= price %>
      .item-form
      %form(action="#/cart" method="post")
        %p
          %label Quantity:
          %input(type="text" size="2" name="quantity" value="1" class="uqf")/
        %p
          %input(type="submit" value="Add to Cart" class="uq")/

      .item-link
        %a(href="<%= url %>") Buy this item on Amazon
      .back-link
        %a(href="#") &laquo; Back to Items
```

So, let's talk about that shopping cart thing. We've been making the point that when it changes, when you call `item.update` within the product detail view, any corresponding subscribing views sholud automatically update.

12b ⟨*cart widget* 12b⟩≡ (16) 13a ▷
```
class CartWidget extends Backbone.View
    el: $('.cart-info')
    template: $('#store_cart_template').html()

    initialize: () ->
        @collection.bind 'update', @render.bind @
```

And there is the major magic. CartWidget will be initialized with the ItemCollection; when there is any change in the collection, the widget will receive the 'change' event, which will automatically trigger the call to the widget's `render()` method.

The render method will refill that widget's HTML with a re-rendered template with the new count and cost, and then wiggle it a little to show that it did changed:

13a ⟨*cart widget* 12b⟩+≡ (16) ◁12b

```
      render: () ->
          tel = @$el.html _.template(@template)({
              'count': @collection.getTotalCount()
              'cost': @collection.getTotalCost()
          })
          tel.animate({paddingTop: '30px'}).animate({paddingTop: '10px'})
          @
```

And the HTML for the template is dead simple:

13b ⟨*cart template* 13b⟩≡ (8)

```
  %script#store_cart_template(type="text/x-underscore-template")
    %p Items: <%= count %> ($<%= cost %>)
```

Lastly, there is the `Router`. In Backbone, the Router is a specialized View for invoking other views. It listens for one specific event: when the `window.location.hash` object, the part of the URL after the hash symbol, changes. When the hash changes, the Router invokes an event handler. The Router, since its purpose is to control the major components of the one-page display, is also a good place to keep all the major components of the sytem. We'll keep track of the `Views`, the `ProductCollection`, and the `ItemCollection`.

13c ⟨*router* 13c⟩≡ (16) 13d▷

```
  class BackboneStore extends Backbone.Router
      views: {}
      products: null
      cart: null
```

There are two events we care about: view the list, and view a detail. They are routed like this:

13d ⟨*router* 13c⟩+≡ (16) ◁13c 14a▷

```
      routes:
          "": "index"
          "item/:id": "product"
```

Like most Backbone objects, the Router has an initialization feature. I create a new, empty shopping cart and corresponding cart widget, which doesn't render because it's empty. I then create a new `ProductCollection` and and corresponding `ProductListView`. These are all processes that happen immediately.

What does not happen immediately is the `fetch()` of data from the back-end server. For that, I use the jQuery deferred again, because `fetch()` ultimately returns the results of `sync()`, which returns the result of an `ajax()` call, which is a deferred.

14a ⟨*router* 13c⟩+≡ (16) ◁13d 14b▷

```
    initialize: (data) ->
        @cart = new ItemCollection()
        new CartWidget
            collection: @cart

        @products = new ProductCollection [],
            url: 'data/items.json'
        @views =
            '_index': new ProductListView
                collection: @products
        $.when(@products.fetch({reset: true}))
            .then(() -> window.location.hash = '')
        @
```

There are two things to route *to*, but we must also route *from*. Remember that our two major views, the product list and the product detail, inherited from `\_BaseView`, which has the `hide()` and `show()` methods. We want to hide all the views, then show the one invoked. First, let's hide every view we know about. `hide()` returns either a deferred (if the object is being hidden) or null. The `_.select()` call at the end means that this method returns only an array of deferreds.

14b ⟨*router* 13c⟩+≡ (16) ◁14a 15a▷

```
    hideAllViews: () ->
        _.select(_.map(@views, (v) -> return v.hide()),
            (t) -> t != null)
```

Showing the product list view is basically hiding everything, then showing the index. The function `$$.when` takes arguments of what to wait for; to make it take an array of arguments, you use the `.apply()` method.

15a ⟨*router* 13c⟩+≡ (16) ◁14b 15b▷

```
      index: () ->
          view = @views['_index']
          $.when.apply($, @hideAllViews()).then(() -> view.show())
```

On the other hand, showing the product detail page is a bit trickier. In order to avoid re-rendering all the time, I am going to create a view for every product in which the user shows interest, and keep it around, showing it a second time if the user wants to see it a second time.

Not that we pass it the `ItemCollection` instance. It uses this to create a new item, which (if you recall from our discussion of `getOrCreateItemForProduct()`) is automagically put into the collection as needed. Which means all we need to do is update this item and the item collection *changes*, which in turn causes the `CartWidget` to update automagically as well.

15b ⟨*router* 13c⟩+≡ (16) ◁15a

```
      product: (id) ->
          product = @products.detect (p) -> p.get('id') == (id)
          view = (@views['item.' + id] ||= new ProductView(
              model: product,
              itemcollection: @cart
          ).render())
          $.when(@hideAllViews()).then(
              () -> view.show())
```

Finally, we need to start the program

15c ⟨*initialization* 15c⟩≡ (16)

```
  $(document).ready () ->
      new BackboneStore();
      Backbone.history.start();
```

2 The Program

Here's the entirety of the program. Coffeescript provides its own namespace wrapper:

16 ⟨*store.coffee* 16⟩≡
  ⟨*models* 2⟩

  ⟨*product collection* 3⟩

  ⟨*cart collection* 4a⟩

  ⟨*base view* 6⟩

  ⟨*product list view* 9⟩

  ⟨*product detail view* 10b⟩

  ⟨*cart widget* 12b⟩

  ⟨*router* 13c⟩

  ⟨*initialization* 15c⟩

3 A Little Stylus

Stylus is a beautiful little language that compiles down to CSS. The original version of The Backbone Store used the same CSS provided from the original Sammy tutorial, but I wanted to show you this one extra tool because it's an essential part of my kit.

If you want rounded borders, you know that writing all that code, for older browsers as well as modern ones, and providing it to all the different objects you want styled that way, can be time consuming. Stylus allows you to define a function that can be called from within any style, thus allowing you to define the style here, and attach a set style to a semantic value in your HTML:

17 ⟨*jsonstore.styl* 17⟩≡ 18 ▷

```
  rounded(radius)
    -moz-border-radius-topleft: radius
    -moz-border-radius-topright: radius
    -moz-border-radius-bottomleft: radius
    -moz-border-radius-bottomright: radius
    -webkit-border-bottom-right-radius: radius
    -webkit-border-top-left-radius: radius
    -webkit-border-top-right-radius: radius
    -webkit-border-bottom-left-radius: radius
    border-bottom-right-radius: radius
    border-top-left-radius: radius
    border-top-right-radius: radius
    border-bottom-left-radius: radius

  background_gradient(base)
    background: base
   background: -webkit-gradient(linear, left top, left bottom, from(lighten(base, 20%)), to(da
    background: -moz-linear-gradient(top,  lighten(base, 20%), darken(base, 20%))
```

And if you look down below you'll see the `rounded()` function called for the list items, which have borders.

One of the real beauties of Stylus is that you can contains some style definitions within others. You can see below that the header contains an H1, and the H1 definitions will be compiled to only apply within the context of the header. Stylus allows you to write CSS the way you write HTML!

18 ⟨*jsonstore.styl* 17⟩+≡ ◁17

```
body
  font-family: "Lucida Grande", Lucida, Helvetica, Arial, sans-serif
  background: #FFF
  color: #333
  margin: 0px
  padding: 0px


#main
  position: relative

#header
  background_gradient(#999)
  margin: 0px
  padding: 20px
  border-bottom: 1px solid #ccc

  h1
    font-family: Inconsolata, Monaco, Courier, mono
    color: #FFF
    margin: 0px

  .cart-info
    position: absolute
    top: 0px
    right: 0px
    text-align: right
    padding: 10px
    background_gradient(#555)
    color: #FFF
    font-size: 12px
    font-weight: bold

img
  border: 0

.productitemview
  position: absolute
```

```
    top: 0
    left: 0

  #productlistview
    position: absolute
    top: 0
    left: 0

    ul
      list-style: none

  .item
    float:left
    width: 250px
    margin-right: 10px
    margin-bottom: 10px
    padding: 5px
    rounded(5px)
    border: 1px solid #ccc
    text-align:center
    font-size: 12px

  .item-title
    font-weight: bold

  .item-artist
    font-weight: bold
    font-size: 14px

  .item-detail
    margin: 10px 0 0 10px

    .item-image
      float:left
      margin-right: 10px

    .item-info
      padding: 100px 10px 0px 10px
```

And that's it. Put it all together, and you've got yourself a working Backbone Store.

This code is available at my github at The Backbone Store[16].

[16]See URL `https://github.com/elfsternberg/The-Backbone-Store`.