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.

There are a number of other good tutorials for Backbone (See: Meta Cloud[2], &Yet's Tutorial[3], Backbone Mobile[4] (which is written in Coffee[5]), and Backbone and Django[6]. However, a couple of months ago I was attempting to learn Sammy.js, a library very similar to Backbone, and they had a nifty tutorial called The JsonStore[7].

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

1.1 Literate Program

A note: this article was written with the Literate Programming[8] toolkit Noweb[9]. 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 have been has dropped into the

[1] See URL `http://documentcloud.github.com/backbone/`.
[2] See URL `http://www.plexical.com/blog/2010/11/18/backbone-js-tutorial/`.
[3] See URL `http://andyet.net/blog/2010/oct/29/building-a-single-page-app-with-backbonejs-undersc/?utm_source=twit`
[4] See URL `http://bennolan.com/2010/11/24/backbone-jquery-demo.html`.
[5] See URL `http://jashkenas.github.com/coffee-script/`.
[6] See URL `http://joshbohde.com/2010/11/25/backbonejs-and-django/`.
[7] See URL `http://code.quirkey.com/sammy/tutorials/json_store_part1.html`.
[8] See URL `http://en.wikipedia.org/wiki/Literate_programming`.
[9] See URL `http://www.cs.tufts.edu/~nr/noweb/`.

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. Addressing tabular data usually involves three parts: The *table*, *row*, and *column*. In Backbone, these are represented by the `Collection`, the `Model`, and the `attribute`. The `Collection` often has a URL indicating the back-end source of the table; the `Model` may have a URL indicating its specific row in the table, as a way of efficiently saving itserlf back to the table.

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.

The Product literally has nothing to modify. It already provides all the methods we need.

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*⟩≡ (18)

```
var Product = Backbone.Model.extend({});

var Item = Backbone.Model.extend({
    update: function(amount) {
        if (amount === this.get('quantity')) {
            return this;
        }
        this.set({quantity: amount}, {silent: true});
        this.collection.trigger('update', this);
        return this;
    },

    price: function() {
        return this.get('product').get('price') * this.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. This is called a *chain*, and is fairly common in Javascript.

The big secret to Backbone is that it supplies an advanced 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. For this program, it is the collection as a whole whose value matters, because that collection as a whole represents our shopping cart. 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 its list of albums via these methods to (in our case) static JSON back-end. Backbone provides a mechanism for fetching JSON (and you can override the `.parse()` method to handle XML, CSV, or whatever strikes your fancy); to use the default `.fetch()` method, capture and set the Collection's `.url` field:

3 ⟨*product collection* 3⟩≡ (18)

```
var ProductCollection = Backbone.Collection.extend({
    model: Product,
    initialize: function(models, options) {
        this.url = options.url;
    },

    comparator: function(product) {
        return product.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.

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.

4 ⟨*cart collection* 4⟩≡ (18) 5▷

```
var ItemCollection = Backbone.Collection.extend({
    model: Item,

    updateItemForProduct: function(product, amount) {
        amount = amount != null ? amount : 0;
        var pid = product.get('id');
        var item = this.detect(function(obj) {
            return obj.get('product').get('id') === pid;
        });
        if (item) {
            item.update(amount);
            return item;
        }
        return this.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* 4⟩+≡ (18) ◁4

```
    getTotalCount: function() {
        var addup = function(memo, obj) {
            return memo + obj.get('quantity');
        };
        return this.reduce(addup, 0);
    },

    getTotalCost: function() {
        var addup = function(memo, obj) {
            return memo + obj.price();
        };
        return this.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⟩≡ (18) 7▷

```
var BaseView = Backbone.View.extend({
    parent: $('#main'),
    className: 'viewport',

    initialize: function(options) {
        Backbone.View.prototype.initialize.apply(this, arguments);
        this.$el.hide();
        this.parent.append(this.el);
    },
```

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. In older versions of the Backbone Store, we used to assign `this.el` to a jQuery-wrapped version of the element; that's no longer necessary, as Backbone provides you with its own version automatically in `this.$el`.

The 'parent' field is something I created for my own use, since I intend to have multiple child objects share the same piece of real-estate. The 'className' field is something Backbone automatically applies to the generated `DIV` at construction time. If you pass in an existing element at construction time for the View to use (which is not an uncommon use case!), Backbone will *not* apply the 'className' to it; you'll have to do that yourself.

I use the `initialize` method above to ensure 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 or a jQuery object will not break anything.

Next, we will define the hide and show functions.

7 ⟨*base view* 6⟩+≡ (18) ◁6

```
hide: function() {
    var dfd = $.Deferred();
    if (!this.$el.is(':visible')) {
        return dfd.resolve();
    }
    this.$el.fadeOut('fast', function() {
        return dfd.resolve();
    });
    return dfd.promise();
},

show: function() {
    var dfd = $.Deferred();
    if (this.$el.is(':visible')) {
        return dfd.resolve();
    }
    this.$el.fadeIn('fast', function() {
        return dfd.resolve();
    });
```

```
        return 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 HTML we're going to use for our one-page application.

8 ⟨*index.html* 8⟩≡

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>The Backbone Store</title>
    <link charset="utf-8" href="jsonstore.css" rel="stylesheet" type="text/css">
    ⟨product list template 10⟩
    ⟨product detail template 13a⟩
    ⟨cart template 14c⟩
    </head>
  </head>
  <body>
    <div id="container">
      <div id="header">
        <h1>
          The Backbone Store
        </h1>
        <div class="cart-info"></div>
      </div>
      <div id="main"></div>
    </div>
    <script src="lib/jquery.js" type="text/javascript"></script>
    <script src="lib/underscore.js" type="text/javascript"></script>
    <script src="lib/backbone.js" type="text/javascript"></script>
    <script src="store.js" type="text/javascript"></script>
  </body>
</html>
```

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⟩≡ (18)

```
var ProductListView = BaseView.extend({
    id: 'productlistview',
    template: _.template($("#store_index_template").html()),

    initialize: function(options) {
        BaseView.prototype.initialize.apply(this, arguments);
        this.collection.bind('reset', this.render.bind(this));
    },

    render: function() {
        this.$el.html(this.template({
            'products': this.collection.toJSON()
        }));
        return this;
    }
});
```

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. It takes a template and returns a function ready to render the template. What we're saying here is that we want this View to automatically re-render itself every time the given collection changes in a significant way, using the given template, into the given element. That's what this view ``means.''

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[10], 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 HTML for our home page's template:

10 ⟨*product list template* 10⟩≡ (8)

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

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

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.

11a ⟨*product detail view* 11a⟩≡ (18) 11b ▷

```
var ProductView = BaseView.extend({
    className: 'productitemview',
    template: _.template($("#store_item_template").html()),

    initialize: function(options) {
        BaseView.prototype.initialize.apply(this, [options]);
        this.itemcollection = options.itemcollection;
    },
```

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.

11b ⟨*product detail view* 11a⟩+≡ (18) ◁ 11a 12a ▷

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

12a ⟨*product detail view* 11a⟩+≡ (18) ◁11b 12b▷

```
update: function(e) {
    e.preventDefault();
  return this.itemcollection.updateItemForProduct(this.model, parseInt(this.$('.uqf').v
},

updateOnEnter: function(e) {
    if (e.keyCode === 13) {
        this.update(e);
    }
},
```

The render is straightforward:

12b ⟨*product detail view* 11a⟩+≡ (18) ◁12a

```
render: function() {
    this.$el.html(this.template(this.model.toJSON()));
    return this;
}
});
```

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

13a ⟨*product detail template* 13a⟩≡ (8)

```
<script id="store_item_template" type="text/x-underscore-template">
  <div class="item-detail">
    <div class="item-image">
      <img alt="<%= title %>" src="<%= large_image %>">
    </div>
    <div class="item-info">
      <div class="item-artist"><%= artist %></div>
      <div class="item-title"><%= title %></div>
      <div class="item-price">$<%= price %></div>
      <div class="item-form"></div>
      <form action="#/cart" method="post">
        <p>
          <label>Quantity:</label>
          <input class="uqf" name="quantity" size="2" type="text" value="1">
        </p>
        <p>
          <input class="uq" type="submit" value="Add to Cart">
        </p>
      </form>
      <div class="item-link">
        <a href="<%= url %>">Buy this item on Amazon</a>
      </div>
      <div class="back-link">
        <a href="#">&laquo; Back to Items</a>
      </div>
    </div>
  </div>
</script>
```

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.

13b ⟨*cart widget* 13b⟩≡ (18) 14a▷

```
var CartWidget = Backbone.View.extend({
    el: $('.cart-info'),
    template: _.template($('#store_cart_template').html()),

    initialize: function() {
        Backbone.View.prototype.initialize.apply(this, arguments);
        this.collection.bind('update', this.render.bind(this));
    },
```

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:

14a ⟨*cart widget* 13b⟩+≡ (18) ◁13b

```
    render: function() {
        var tel = this.$el.html(this.template({
            'count': this.collection.getTotalCount(),
            'cost': this.collection.getTotalCost()
        }));
        tel.animate({ paddingTop: '30px' }).animate({ paddingTop: '10px' });
        return this;
    }
});
```

You may have noticed that every render ends in `return this`. There's a reason for that. Render is supposed to be pure statement: it's not supposed to calculate anything, nor is it supposed to mutate anything on the Javascript side. It can and frequently does, but that's beside the point. By returning `this`, we can then call something immediately afterward.

For example, let's say you have a pop-up dialog. It starts life hidden. You need to update the dialog, then show it:

14b ⟨*example* 14b⟩≡

```
    myDialog.render().show();
```

Because what render() return is `this`, this code works as expected. That's how you do chaining in HTML/Javascript.

Back to our code. The HTML for the Cart widget template is dead simple:

14c ⟨*cart template* 14c⟩≡ (8)

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

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`.

15a ⟨*router* 15a⟩≡ (18) 15b▷

```
var BackboneStore = Backbone.Router.extend({
    views: {},
    products: null,
    cart: null,
```

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

15b ⟨*router* 15a⟩+≡ (18) ◁15a 15c▷

```
    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.

15c ⟨*router* 15a⟩+≡ (18) ◁15b 16a▷

```
    initialize: function(data) {
        Backbone.Router.prototype.initialize.apply(this, arguments);
        this.cart = new ItemCollection();
        new CartWidget({ collection: this.cart });
        this.products = new ProductCollection([], { url: 'data/items.json' });
        this.views = {
            '_index': new ProductListView({ collection: this.products })
        };
        $.when(this.products.fetch({ reset: true })).then(function() {
            return 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 `_.filter()` call at the end means that this method returns only an array of deferreds.

16a ⟨*router* 15a⟩+≡ (18) ◁15c 16b▷

```
hideAllViews: function() {
    return _.filter(_.map(this.views, function(v) { return v.hide(); }),
                    function(t) { return 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.

16b ⟨*router* 15a⟩+≡ (18) ◁16a 17a▷

```
index: function() {
    var view = this.views['_index'];
    return $.when.apply($, this.hideAllViews()).then(function() {
        return 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. Note that the view only needs to be rendered *once*, after which we can just hide or show it on request.

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.

17a ⟨*router* 15a⟩+≡ (18) ◁16b

```
product: function(id) {
    var view = this.views[id];
    if (!view) {
        var product = this.products.detect(function(p) {
            return p.get('id') === id;
        });
        view = this.views[id] = new ProductView({
            model: product,
            itemcollection: this.cart
        }).render();
    }
    return $.when(this.hideAllViews()).then(function() {
        return view.show();
    });
}
});
```

Finally, we need to start the program

17b ⟨*initialization* 17b⟩≡ (18)

```
$(document).ready(function() {
    new BackboneStore();
    return Backbone.history.start();
});
```

2 The Program

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

18 ⟨*store.js* 18⟩≡
⟨*models* 2⟩

⟨*product collection* 3⟩

⟨*cart collection* 4⟩

⟨*base view* 6⟩

⟨*product list view* 9⟩

⟨*product detail view* 11a⟩

⟨*cart widget* 13b⟩

⟨*router* 15a⟩

⟨*initialization* 17b⟩

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[11].

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