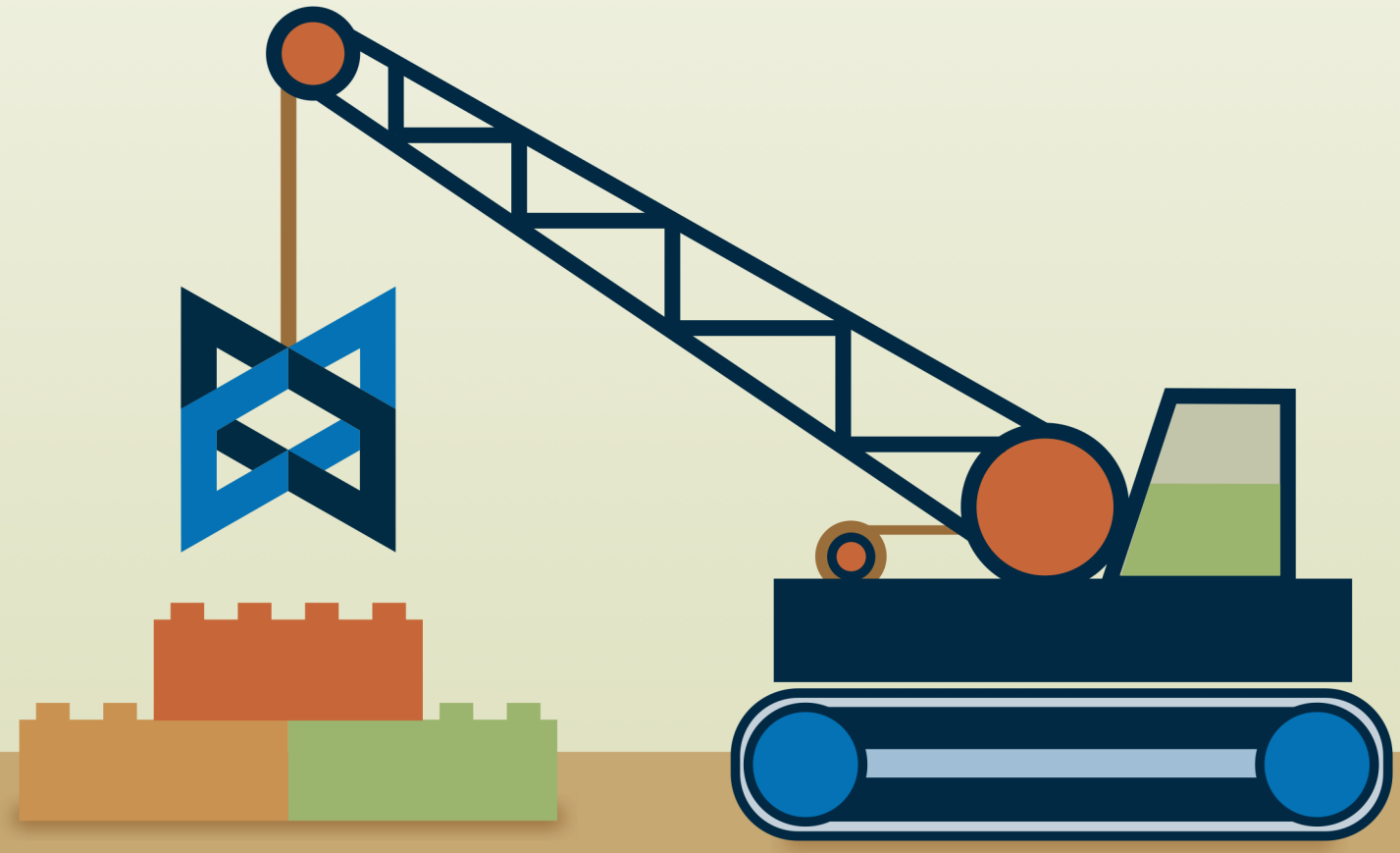
Covers Backbone.js v1.0

BUILDING BACKBONE PLUGINS



By Derick Bailey

Eliminate The Boilerplate In Backbone.js Apps

Building Backbone Plugins

Eliminate The Boilerplate In Backbone.js Apps

Derick Bailey and Jerome Gravel-Niquet

This book is for sale at http://leanpub.com/building-backbone-plugins

This version was published on 2014-05-05

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Derick Bailey and Jerome Gravel-Niquet by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm leveling up my #Backbone skills with the #BackbonePlugins e-book! http://backboneplugins.com

The suggested hashtag for this book is #BackbonePlugins.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#BackbonePlugins

Contents

Quotes And Praise For Building Backbone Plugins

"It was the perfect read given that I was using backbone in a a unstructured manner leaving many design paradigms in the dust. The book is a great guide on how to transition from a mediocre backbone web developer to a full-fledged software engineer with backbone under their tool belt." - Peter Tseng

"im currently sheding tears of joy while reading your book... #AMAZING" - @Lunatikzx

"LOVE IT" - @sheldonj1983

"AWESOME!" - @wbingham

"that [old] cover makes the pins in my knee hurt :D" - @scottpu

Preface

Better, Faster, Cheaper

Building Backbone.js applications without the use of plugins, add-ons and project-specific abstractions is a waste of time and money.

If every feature of every system (in any language or platform) had to be built form the ground up, no one would deliver anything of real value. Instead, we build systems on top of systems - the abstractions, plugins, libraries and frameworks that perform the core functions of the platform and infrastructure we need. But even with the use of existing add-ons and abstractions, the understanding of how to effectively build your own abstractions and re-usable components is a necessary part of software development. JavaScript and Backbone.js applications are no different than any other application or system in this regard. It is our responsibility as developers, technical leads and architects, then, to look for ways to effectively use and create re-usable plugins, add-ons and abstractions.

But while Backbone is no different than any other system in the need for abstractions, it does provide some unique and interesting ways in which we can build our reusable pieces. And having an understanding of how to use Backbone effectively means more than just using the existing components that it comes with, or existing plugins built by other developers. It means understanding how all of the pieces fit together, and how those pieces can be extended and augmented. It means understanding how to pull apart the general needs of a component to create something that can be extended in to a specific scenario. And it means knowing how to recognize and extract the patterns that we create in our own applications, in to re-usable components that may be deliverable as plugins and add-ons in themselves.

This book, then, will show you how to effectively use Backbone.js by building abstractions, add-ons and plugins for your own applications and as open source projects. It will give you the knowledge you need to work with the components that Backbone includes, build complementary objects to be used within JavaScript and Backbone applications, and ultimate save time and money while delivering more features.

What Is Backbone?

Backbone.js is a small library of objects that aim to provide structure to JavaScript applications. It provides a handful of building blocks and tools that help us to build applications with a better architecture, separation of concerns and smaller, focused pieces that all add up to a larger application, including:

- `Backbone.Model`: A set of data, and optionally logic, representing an entity in the application
- `Backbone.Collection`: A group of related models, with standard functions for retrieving models, iterating over the models, etc.
- `Backbone.View`: Views provide a way to encapsulate and organize jQuery or other DOM manipulation library code, and manage, manipulate and respond to the DOM through declarative DOM events, model and collection event bindings, etc.
- `Backbone.Router`: Respond to changes in the application's URL, providing bookmarkable and navigable URL's based on tokenized representations of the application state
- `Backbone.History`: Manage the actual browser history, forward and backward buttons, and respond to configured routes. The `History` object does the actual work that routers define, allowing multiple routers to exist in a single application.
- `Backbone.Events`: Provide an event-driven, or public/subscribe architecture, allowing application components to be decoupled from each other while still allowing communication to happen between them
- `Backbone.sync`: Provides an abstraction for model and collection persistance. The default implementation expects a REST-like API on a server, communicating with JSON over jQuery AJAX calls

For a more complete introduction to Backbone and it's components, see the following resources:

- Addy Osmani's Backbone Fundamentals[1]
- My Intro To Backbone Screencasts[2]
- My Additional Resources[3]

Why Are Plugins And Add-ons Neccessary?

In spite of - or perhaps because of - the flexibility, simplicity and power in the components that Backbone provides, it does not provide everything that a developer needs to create a complete application. There are several concerns that many applications have which Backbone does not directly address, or which Backbone provides some support without a direct implementation.

For example, Backbone's `View` provides a default `render` method that does nothing. It takes no parameters and it produces no observable changes as a result of calling it. However, this method is built in to `Backbone.View` to illustrate the common practice of providing a `render` method on a view, which will generate the HTML structure that the view is to manage.

On the flip side of that coin, Backbone's `Model` and `Collection` provide no method of handling nested or hierarchical models and collections. It's up to the developer and the individual project to set the standard for adding this support, when it's needed.

[1] https://github.com/addyosmani/backbone-fundamentals

[2] http://pragprog.com/screencasts

[3] http://backbonetraining.net/resources

In both of these cases, there are existing plugins and frameworks that provide solutions for Backbone-based projects. Frameworks such as my own Backbone.Marionette[4] or Tim Branyen's Backbone.LayoutManager[5] provide a default, but easily changeable, rendering solution for Backbone views - among many other benefits and capabilities. There are a handful of tools that can provide nested and hierarchical models, such as Backbone-Relational[6], and many other tools, add-ons, plugins and frameworks that provide a tremendous amount of power, flexibility and capabilities on top of what Backbone provides. For a more complete list of the available plugins and frameworks, see the Backbone Wiki[7].

With these existing add-ons, plugins and frameworks available, there is often enough for an application of any size to be completed. But there are also times when the feature set of a specific add-on, or the implementation of an individual framework, may not play well with the opinions and needs of the development team working on an application. There are also scenarios where an individual application needs a very specialized set of functionality to work properly - one that needs to be repeated several times throughout the application, but is also specific enough to the project that it cannot be generalized in to an open source project. In either of these cases, and in many other more subtle scenarios, it is beneficial to understand how to create your own plugins and add-ons for Backbone so that you can take full advantage of encapsulated functionality, customized for your application's needs, while still playing well with existing add-ons. Creating your own add-ons and plugins is both easy, and tremendously frustrating at the same time. Add-ons that are seemingly simple, such as creating the ability to have a "selectable" model and collection via my Backbone.Picky[8], are quick to build and be useful, but become very complex as the more subtle edge cases are discovered.

Throughout this book, I'll walk you through the tools, tricks and lessons learned for building the smallest of add-ons through the largest of application frameworks such as Backbone.Marionette. I'll teach you the tricks that I've learned, show you the pitfalls to avoid, and walk you through the construction of add-ons that are both useful and easy to use, while not creating an undue burden for the developers that are maintaining them or the developers that are using them in their applications.

Who Should Read This Book?

This book is aimed at developers that are already familiar with building Backbone.js applications of any scale, and are looking for ways to reduce their development efforts, clean up their code base, and provide standardized implementations for their teams to use in products and projects. It will show how to take a large code base and reduce duplication and boilerplate code down in to managable, customizable and flexible chunks that an entire team can then apply.

[4] http://github.com/derickbailey/backbone.marionette
[5] https://github.com/tbranyen/backbone.layoutmanager
[6] https://github.com/PaulUithol/Backbone-relational
[7] https://github.com/documentcloud/backbone/wiki/Extensions%2C-Plugins%2C-Resources
[8] http://github.com/derickbailey/backbone.picky

The ideal reader is a person who is building medium to large sized applications in Backbone, and may or may not be part of a team. You may be a team lead, looking to simplify the architecture and implementation of a large project, or a sole-developer or contractor, looking for more effective implementation patterns for your Backbone applications. If you're writing applications and noticing a trend in the amount of boilerplate code, a series of patterns of implementation, or are looking for insight in to how to recognize these patterns, then this book is for you.

This book is not a general introduction to Backbone and its core concepts. There are plenty of books, screencasts, blog posts and other materials that cover a general introduction in great depth. Rather, this book will take your existing knowledge of building Backbone applications to a new level, showing you how to create common abstractions and re-usable implementations that may benefit your project specifically or be more broadly applicable and deliverable as open source.

Additionally, this book is not a general introduction to JavaScript, protoypal inheritance, design patterns, or test-driven development. While this book will touch on these subjects and show how to effectively make use of them in order to produce the highest quality, maintainable and flexible code, it is assumed that are at least familiar with the core JavaScript, it's inheritance system, design patterns and unit testing in general.

Chapter 1: View Rendering

A typical Backbone view will need to render some HTML and have that HTML populated in to the view's `$el`. A simple example, using UnderscoreJS' templates, might look like this:

```
HelloWorldView = Backbone.View.extend({

  render: function(){
    var renderedHtml = _.template("<h1>Hello world!</h1>");
    this.$el.html(renderedHtml);
  }

});

var view = new HelloWorldView();
view.render();

console.log(view.$el); //=> "[<h1>Hello world!</h1>]"
```

If there is any data needed from a Backbone.Model, I can change the `render` method to include it when calling the template method:

```
DataDrivenView = Backbone.View.extend({

  render: function(){
    var renderedHtml = _.template("<h1><%= message %></h1>", this.model.toJSON());
    this.$el.html(renderedHtml);
  }

});

var model = new Backbone.Model({message: "Hello Data-World!"});
var view = new DataDrivenView({
  model: model
});

view.render();

console.log(view.$el); //=> "[<h1>Hello Data-World!</h1>]"
```

And if I need to render a collection of items in to my view's template, I can swap out `this.model` for `this.collection` and iterate through the items in the template:

```
CollectionView = Backbone.View.extend({

  render: function(){

    var data = {
      items: this.collection.toJSON()
    };

    var renderedHtml = _.template("<h1><% _.each(items, function(item){ %><%= mes\
sage %> <% }) %>!</h1>", data);

    this.$el.html(renderedHtml);
  }

});

var collection = new Backbone.Collection([
  {message: "Hello"},
  {message: "Collection"},
  {message: "Driven"},
  {message: "World"}
]);

var view = new DataDrivenView({
  collection: collection
});

view.render();

console.log(view.$el); //=> "[<h1>Hello Collection Driven World!</h1>]"
```

In this case, I had to add a bit more to the render method. Not only did the template have to do the iteration for me, but the data that I passed in to the template needed be wrapped in a parent structure so that I could do the iteration. In the end, though, this view rendered the expected results in to the `$el`.

In looking at these three examples, there are several things that I can say are the same and several things that I can say are different. It can also be said that there is a lot of boilerplate between these three examples. It is boilerplate like this that become the basis for creating plugins and add-ons.

Extract Whats Common, Specify The Differences

Boilerplate code gets very frustrating very quickly. Having to type the same code over and over again is never fun. Copy and paste tends to be the first answer to that problem, but this quickly falls apart. Any time there is a change to the pattern of code being used, all of the places where this copy and paste occurred have to be updated. In any application of any substantial size, this is going to be a nightmare and it's likely that at least one location that needs to be updated won't be. To fix both the boilerplate problem and prevent the problems associated with copy & paste programming, the common parts of the solution can be abstracted away from the specifics of each use.

In the previous view rendering examples, there are some very obvious bits of code duplication or boilerplate. Each of the view's render methods does the following:

- Make a call to the `_.template` method
- Pass an HTML template, as a string, to the `template` method
- Replace the HTML contents of the `$el` property on the view

The differences between these methods can also be seen fairly easily:

- The first view does not use any data
- The second view calls `this.model.toJSON()` to get data, and passes that to the `_.template` function
- The third view calls `this.collection.toJSON()` to get data, wraps that in another object literal, and passes the resulting object to the `_.template` function

The third view shows not only a more complex example, but also a hint at how the common parts of the render functionality can be extracted in to something reusable. The use of the `data` variable in this view shows me that I don't have to supply all of the parameters to each of these function calls as literal values. Instead, I can extract them to variables.

For example, if I extracted all of the parameters in the third example, it might look like this:

```
1  Backbone.View.extend({
2
3    render: function(){
4
5      var template = "<h1><% _.each(items, function(item){ %><%= message %> <% }) %\
6  >!</h1>";
7      var data = {
8        items: this.collection.toJSON()
9      };
10
```

```
    var renderedHtml = _.template(template, data);
    this.$el.html(renderedHtml);
  }

});
```

I can take this a step further as well, with the `template` variable. Since this template is not going to change from one call of the `render` method to another, I can move this out to the view definition:

```
Backbone.View.extend({
  template: "<h1><% _.each(items, function(item){ %><%= message %> <% }) %>!</h1>\
",

  render: function(){
    var data = {
      items: this.collection.toJSON()
    };

    var renderedHtml = _.template(this.template, data);
    this.$el.html(renderedHtml);
  }

});
```

With the template extracted to the view definition, the render function becomes much easier to read and understand. The only code left in this function that is different than the other examples, is the call to create the data. The calls to render the template and populate the data are done with variables now. If I can take this one step further and extract the process of serializing the data that the view needs, then I'll have a way to make this view rendering completely generic.

```
Backbone.View.extend({
  template: "<h1><% _.each(items, function(item){ %><%= message %> <% }) %>!</h1>\
",

  serializeData: function(){
    return data = {
      items: this.collection.toJSON()
    };
  },

  render: function(){
    var data = this.serializeData();
```

```
    var renderedHtml = _.template(this.template, data);
    this.$el.html(renderedHtml);
  }
});
```

This view now has a completely generic `render` method, with two additional attributes that provide the template and the data that the view needs when rendering.

To create a generic base view out of this, I can take advantage of Backbone's `extend` function. Whenever I extend from a type that Backbone provides, the `extend` method comes along with it. This method is like the `extends` keyword in Java, or the `:` inheritance character in C#. The mechanics of how it works are different, as JavaScript and Backbone use prototypal inheritance, but at a high level it is just a form of inheritance.

Given that, I can create a base view that provides the rendering mechanics for an application, like this:

```
BaseView = Backbone.View.extend({
  render: function(){
    var data;
    if (this.serializeData){
      data = this.serializeData();
    };

    var renderedHtml = _.template(this.template, data);
    this.$el.html(renderedHtml);
  }
});
```

Then when I need a specific view to render with a template and data, I only need to extend from that BaseView view instead of Backbone.View. In this case, all three of the previous views that I had created would be able to extend from it. Even the first view that does not need to render any data will work fine, as this `BaseView` has provided a simple check around the existence of the `serializeData` method. If this method does not exist, it won't be called. The `data` variable would be undefined at that point, and it would be ignored by the UnderscoreJS `template` function.

```
HelloWorldView = BaseView.extend({
  template: "<h1>Hello world!</h1>"
});

DataDrivenView = BaseView.extend({
  template: "<h1><%= message %></h1>",

  serializeData: function(){
    return this.model.toJSON();
  }
});

CollectionView = BaseView.extend({
  template: "<h1><% _.each(items, function(item){ %><%= item.message %> <% }) %>!\
</h1>",

  serializeData: function(){
    return {
      items: this.collection.toJSON();
    };
  }
});
```

These three views, extending from the new `BaseView`, no longer have any of the boilerplate rendering code in them. They only specify the differences that each of the views needs.

Lessons Learned

This chapter has provided a quick introduction to creating a simple yet valuable plugin for Backbone. And there are several lessons that can be pulled from this particular abstraction. Some of them are specific to views and rendering of views, but others can be generalized in to something more broadly applicable.

Solve Real Problems

The rendering example isn't just an academic exercise to show how to pull apart several implementations and create something re-usable. It's a real-world solution, based on a real world problem. Developers have written the same rendering code in different view definitions a countless number of times.

Extract Common Code

By examining the differences between the three view definitions and rendering processes, we were able to spot the similarities - this things that were the same. The process of converting the various method parameters in to variables also helped us see what was common and what was different. We ended up with a few lines of code that were repeated through all of the views, and a few lines of code that were specific to each view.

Specify The Differences

Once we had the commonalities in the different view implementations identified, it was easy to see what was different as well. Those differences were then extracted from the core render method in a manner that allowed the render method to be flexible to the different needs of each view. Every view had a template to render, but each view's template was different. Specifying the template as part of the view definition allowed the render method to render the template without having to know what the template's contents were. The data used in rendering followed the same pattern initially, but also added the check to see if we needed to provide any data for the rendering. Specifying the differences for each view, within the view's definition, allowed the commonalities to work properly, and provided more flexibility.

Backbone.View Extension Points

Every `Backbone.View` instance has a `render` method. This method is empty by default, but provides an extension point that we can use to provide rendering for our view. There are a number of other methods and extension points in Views, as well. Some of them are better to use than others, though. Sometimes it's in our best interest to avoid providing an implementation for a specific setting, while other times Backbone expects us or at least encourages us to provide an implementation.

Chapter 12: Building Backbone.localStorage

Lots of real-world apps need to persist data in some form of storage. This is normally achieved by creating a server which acts as an interface between a database and your client. Sadly, this adds a ton of dependencies and complexity. People are less likely to contribute to your project if it's difficult to setup.

When I built Backbone Todos, I didn't want to add any dependencies of the sort. But how would I be able to store data without a dependency on a database system?

For years now, browsers expose a nifty `localStorage` object, which does exactly what the name implies: it stores data locally. More specifically, in a file on your hard drive, a file which your browser knows how to access and parse.

The answer, then, is to use `localStorage` as the data store. While it is a dependency, it is not one that has to be installed. Your browser already has it built in.

Storing data on the client

Choosing to use `localStorage` made sense at the time. Although other solutions like IndexedDB and Web SQL were more powerful, they were less widespread.

`localStorage`'s API can be a pain to use if you're building something complex. You're better off wrapping it with syntactic sugar. For instance, this little `save` method:

```
1  save: function() {
2    this.localStorage().setItem(this.name, this.records.join(","));
3  }
```

In this case, `this.records` holds a simple array of all the documents managed by Backbone.localStorage.

Generating IDs

For some client-only apps, instances might/will not have IDs. Some people use Backbone.localStorage for caching data temporarily (and so, they might have IDs) and others use it as their primary storage strategy (and so, don't have IDs unless they manually generate ones for their models.) But for data storage and retrieval, it is necessary to generate an ID for every object.

Backbone.localStorage needs to support both cases. The former is easy, use whatever ID is supplied. The other case, generating an ID, can be achieved with this snippet of code:

```
// Generate four random hex digits.
function S4() {
  return (((1+Math.random())*0x10000)|0).toString(16).substring(1);
};

// Generate a pseudo-GUID by concatenating random hexadecimal.
function guid() {
  return (S4()+S4()+"-"+S4()+"-"+S4()+"-"+S4()+"-"+S4()+S4()+S4());
};
```

What are `S4()` and `guid()`?

The function `S4` generates a random number, converts it to a string (resembling *"298c"*) and then `guid` assembles them to create a complex string which will almost assuredly be unique (enough for our needs.) The final form looks like *"4df366d6-26d3-dab8-8018-ca6252b252a7"*

Then in the `create` method of the plugin, we use the generated ID:

```
if (!model.id) {
  model.id = guid();
  model.set(model.idAttribute, model.id);
}
```

`localStorage` gotchas

`localStorage` is pretty good when it comes to storing simple key/value data. You come across a few issues when trying to store full data representation of complex models.

Size limit

The default size limit on data you can store in `localStorage` can vary wildly. It may be set by the user, a website may request a user to allow for more storage and, by default, it's set at 5 MB (may vary depending on the browser.)

Once you hit that limit, you need to handle a "QUOTA_EXCEEDED_ERR". The way Backbone.localStorage handles that is by throwing it back at the app through the `options.error` callback.

Performance

`localStorage` is not very fast. It saves data to disk. As much as possible, you want reduce the number and weight of your writes.

In first versions of Backbone.localStorage, everything used to be held in a single, monolithic, key inside `localStorage`. It wasn't a great idea.

Later on, when that performance issue hit, a refactor occurred to store data more or less like a normal database. The plugin now saves an "index" of all the IDs for a collection in a key and then has one key for each record.

Retrieving the data is still pretty fast (even if it has to go through multiple keys.) The real difference, when it comes to writing data to `localStorage`, is about a 66x performance gain.

Overriding `Backbone.sync` properly

Taken from the Backbone documentation (emphasis mine): > `Backbone.sync` is the function that Backbone calls every time it attempts to read or save a model to the server. By default, it uses jQuery.ajax to make a RESTful JSON request and returns a jqXHR. **You can override it in order to use a different persistence strategy, such as WebSockets, XML transport, or Local Storage**.

People often override `Backbone.sync` to accommodate custom headers/params needed to use with their server.

Overriding is too destructive for a proper Backbone plugin. In Backbone.localStorage, we check for a property on the model or collection (quite simply `localStorage`) and then override the `Backbone.sync` function with a small function which checks for that property and calls either the old `Backbone.sync` or a new pimped one. Like so:

```
1  Backbone.ajaxSync = Backbone.sync;
2
3  Backbone.getSyncMethod = function(model) {
4    if(model.localStorage || (model.collection && model.collection.localStorage)) {
5      return Backbone.localSync;
6    }
7
8    return Backbone.ajaxSync;
9  };
10
11 // Override 'Backbone.sync' to default to localSync,
12 // the original 'Backbone.sync' is still available in 'Backbone.ajaxSync'
13 Backbone.sync = function(method, model, options) {
14   return Backbone.getSyncMethod(model).apply(this, [method, model, options]);
15 };
```

Syncing the data

Since we're going through the database directly, instead of going through a server, we don't most of what the current `Backbone.sync` method does.

Let's take a closer look at the replacement function.

Sync methods

```
Backbone.sync = function(method, model, options) {
```

The first argument passed to the function is a method for syncing the data. It can be either one of those: `create`, `read`, `update` or `delete`.

Normally, Backbone translate those to HTTP methods and then passes that to `$.ajax` which calls a endpoint defined by a collection or model's `url` property.

We need none of that here and so the "server logic" for handling data is directly in the plugin

```
switch (method) {
  case "read":
    resp = model.id != undefined ? store.find(model) : store.findAll();
    break;
  case "create":
    resp = store.create(model);
    break;
  case "update":
    resp = store.update(model);
    break;
  case "delete":
    resp = store.destroy(model);
    break;
}
```

Granted, a `switch` isn't the fastest way to do this, but it's not slow enough to sacrifice the legibility it provides.

Faking the response

Once data is gathered, we need to call `success` or `error` callbacks. These are usually "wrapped" by Backbone to handle operating on collections/models **and** responding to manually specified callbacks (for instance: providing the `success` and/or `error` keys in the options for a `fetch`, `create`, etc. method call.)

The `success` Callback

```
if (resp) {
  if (options && options.success) {
    if (Backbone.VERSION === "0.9.10") {
      options.success(model, resp, options);
    } else {
      options.success(resp);
    }
  }
  if (syncDfd) {
    syncDfd.resolve(resp);
  }
}
```

Mostly, this piece of code calls the `success` option with the data returned from storage.

For a while now, Backbone has been using `$.Deferred` and so it also needs to be handle. Those "promises" need to be `resolved` in the case of success and `rejected` in case of failure.

Funky Backbone fact

In Backbone 0.9.10, `Backbone.sync` called the `success` callback with arguments in a different order.

Going from `success(response)` to `success(model, response, options)`.

This was then reverted in the next stable release, Backbone 1.0. Therefore any plugin dealing with `sync` had to have a conditional for 0.9.10 specifically.

Eventually, this little "shim" will be removed from `Backbone.localStorage`.

The `error` Callback

Remember the `localStorage` woes? Due to its not-quite-yet-supported-everywhere-the-same-way nature, error handling is a must.

The crux of the replaced `sync` method is actually wrapped inside a `try {} catch (error) {}` just for that purpose.

The process then goes on doing its thing and if an error occurred earlier, it'll simply call `options.error` and the promise's `reject` method.

Universal Module Definition

UMD[9] is "[...] patterns for JavaScript modules that work everywhere"

[9] https://github.com/umdjs/umd

In a nutshell, it's boilerplate code to support the various module loading standards out there. AMD, CommonJS or plain old globals. In Backbone.localStorage, we've gone the way of supporting all possibilities, with this little piece of code:

```
(function (root, factory) {
   if (typeof exports === 'object' && root.require) {
     module.exports = factory(require("underscore"), require("backbone"));
   } else if (typeof define === "function" && define.amd) {
      // AMD. Register as an anonymous module.
      define(["underscore","backbone"], function(_, Backbone) {
        // Use global variables if the locals are undefined.
        return factory(_ || root._, Backbone || root.Backbone);
      });
   } else {
      // RequireJS isn't being used. Assume underscore and backbone are loaded in\
 <script> tags
      factory(_, Backbone);
   }
}(this, function(_, Backbone) {
```

Let's dig a little deeper in this code.

The function takes the global context (`root`) as a first argument and a `factory` as the second.

The first one, the global context, is either `window` in the browser or `global` in node.js.

The second is our main code. All its dependencies are defined as arguments. In our case, we need Underscore.js _ and Backbone.js `Backbone`.

Within the UMD, we first detect CommonJS support by checking for the existence of an `exports` object and a function `require`.

```
if (typeof exports === 'object' && root.require) {
  module.exports = factory(require("underscore"), require("backbone"));
}
```

If it is present, assign the result of our factory to `module.exports`.

In the case CommonJS is not present, we check for AMD support. A similar approach is used by evaluating if `define` is a function and if it complies to AMD standards.

```
if (typeof define === "function" && define.amd) {
  // AMD. Register as an anonymous module.
  define(["underscore","backbone"], function(_, Backbone) {
    // Use global variables if the locals are undefined.
    return factory(_ || root._, Backbone || root.Backbone);
  });
}
```

A bit more complex since it might need to load libraries asynchronously, we use the `define` method to "define" our module. It takes an array of dependencies and then a function with those dependencies resolved as arguments.

Often, libraries don't support AMD. This is very much the case for Underscore and Backbone (Underscore used to support it way back when.) Therefore, if they're undefined within our function, we use the global context's `Backbone` and `_`.

Finally, we assume Underscore and Backbone are already loaded in the global context if no other module loading strategy is being used.

```
} else {
  // RequireJS isn't being used. Assume underscore and backbone are loaded in <sc\
ript> tags
  factory(_, Backbone);
}
```

This whole module loading thing is fairly important for plugins and libraries in general. It's an easy fix to avoid people having to modify your code for this single purpose. In turn, people are more likely to use a pristine version of your library which reduces the difficulty of upgrading.

Furthermore, it's generally a good practice. Your code will be more contained, less likely to spread in the global context.

Lessons Learned

Backbone's ability to have parts of it replaced or removed speak s to the flexibility of this library. When one or more parts don't work the way you want, you are free to find ways to make it work. Having the ability to modify and/or replace the sync mechanism is a great example of this.

Be Careful When Replacing Things

Just because you can replace something, doesn't mean you should. Sometimes it is better to augment what was there and wrap it in your own code so that you can use the original behavior or the new behavior as needed. This is often referred to as a decorator or proxy pattern, and can help you add features and functionality without altering existing code.

Widespread Adoption Trumps Power

Software development is a series of trade-offs and choices, drawbacks and benefits that have to be weighed against each other. There will be times when you have to make a choice that you don't necessarily like. You may have to support an old browser because a client requires it, or you may have to use `localStorage` instead of IndexDB because you need better support across multiple browsers.

Understanding the context and constraints in which your application will run is important when making decisions. In the case of the Backbone Todos sample application, browser support with fewer external dependencies was more important than using something wide spread or more powerful.

Convenience Comes At A Price

Having something available when you need it, and available across many browsers doesn't always mean it's the best thing out there. The `localStorage` in browsers has limitations including speed and storage size. Be sure you are weighing the pros and cons of convenience and availability against the actual needs of your system.

Reproduce The Complete Behavior, Not Just The API

It's easy to think that reproducing the API of an object or method is good enough. But the API for the object/method is only the surface area of an iceberg under the sea. Before you can replace something, you need to understand the hidden behaviors and expectations of code that uses what you are about to replace.

In the case of `Backbone.sync`, replacing the `sync` method with another method of the same API is not enough. The new implementation needs to ensure the behavior of calling `success` and/or `error` are maintained so that calling code will continue to work as expected.

Chapter 16: Application Workflow

It is unfortunately common to have very poorly defined and constructed workflow in JavaScript and Backbone applications. A significant amount of time is spent creating new and re-usable View and Model types, and plugins and add-ons for them. But, this critical area of application workflow is typically overlooked. Rather than modeling workflow explicitly, applications tend to have the workflow scattered through other objects within an application. When one the application needs to move from one view to the next, the first view will call the second view directly. This path leads toward a mess of tightly coupled concerns, and a brittle and fragile system that is dependent entirely on the implementation details. Worse yet, to understand the higher level workflow and concept, the implementation details must be examined. This makes it very hard to understand the workflow, as the detail of each part tends to hide the high level flow.

A Poorly Constructed Workflow

For example, you might have a human resources application that allows you to add a new employee and select a manager for the employee. After entering a name and email address, we would show the form to select the manager. When the user clicks save, we create the employee. A crude, but all too common implementation of this workflow might look something like this:

```
1  EmployeeInfoForm = Backbone.View.extend({
2    events: {
3      "click .next": "nextClicked"
4    },
5
6    nextClicked: function(e){
7      e.preventDefault();
8
9      var data = {
10       name: this.$(".name").val(),
11       email: this.$(".email").val()
12     };
13
14     var employee = new Employee(data);
15
16     this.selectManager(employee);
17   },
18
```

```
  selectManager: function(employee){
    var view = new SelectManagerForm({
      model: employee
    });
    view.render();
    $(".wizard").show(view.el);
  },

  // ...
  render: function(){ ... }
  // ... etc
});

SelectManagerForm = Backbone.View.extend({
  events: {
    "click .save": "saveClicked"
  },

  saveClicked: function(e){
    e.preventDefault();

    var managerId = this.$(".manager").val();
    this.model.set({managerId: managerId});

    this.model.save();
    // do something to close the wizard and move on
  },

  // ...
  render: function() { ... }
  // ... etc
});
```

Can you quickly and easily describe the workflow in this example? If you can, it's likely because you spent time looking at the implementation details of both views in order to see what's going on and why.

Too Many Concerns

There are at least two different concerns mixed in to each of the objects in the above code. And those concerns have been split apart in some rather un-natural ways at that.

The first concern is the high level workflow:

- Enter employee info
- Select manager
- Create employee

The second concern is the implementation detail of each view, which includes (in aggregate):

- Show the EmployeeInfoForm
- Allow the user to enter a name and email address
- When “next” is clicked, gather the name and email address of the employee.
- Then show the SelectManagerForm with a list of possible managers to select from.
- When “save” is clicked, grab the selected manager
- Then take all of the employee information and create a new employee record on the server

There’s potential for further decision points and branching in this workflow, which have not been accounted for, as well. What happens when the user hits cancel on the first screen? Or on the second? What about invalid email address validation? If you start adding in all of those steps to the list of implementation details, this list of steps to follow is going to get out of hand very quickly.

By implementing both the high level workflow and the implementation detail in the views, thee ability to see the high level workflow at a glance has been destroyed. This will cause problems for developers working with this code.

Imagine coming back to this code after even a few days away. It will be difficult to see what’s going on, to know if the workflow has changed since the last time you worked on it, and to see exactly where the changes were made - in the workflow, or in the implementation details.

Model Explicit Workflow

What we want to do, instead, is get back to that high level workflow with fewer bullet points and very little text in each point. But we don’t want to have to dig through all of the implementation details in order to get to it. We want to see the high level workflow in our code, separated from the implementation details. This makes it easier to change the workflow and to change any specific implementation detail without having to rework the entire workflow.

Wouldn’t it be nice if we could write this code, for example:

```
var orgChart = {

  addNewEmployee: function(){
    var employeeDetail = this.getEmployeeDetail();
    employeeDetail.on("complete", function(employee){

      var managerSelector = this.selectManager(employee);
      managerSelector.on("save", function(employee){
        employee.save();
      });

    });
  },

  // ...
}
```

In this pseudo-code example, we can more clearly see the high level workflow. When we complete the employee info, we move on to the selecting a manager. When that completes, we save the employee with the data that we had entered. It all looks very clean and simple. We could even add in some of the secondary and third level workflow without creating too much mess. And more importantly, we could get rid of some of the nested callbacks with better patterns and function separation.

```
var orgChart = {

  addNewEmployee: function(){
    var that = this;

    var employeeDetail = this.getEmployeeDetail();
    employeeDetail.on("complete", function(employee){

      var managerSelector = that.selectManager(employee);
      managerSelector.on("save", function(employee){
        employee.save();
      });

    });
  },

  getEmployeeDetail: function(){
    var form = new EmployeeDetailForm();
```

```
    form.render();
    $("#wizard").html(form.el);
    return form;
  },

  selectManager: function(employee){
    var form = new SelectManagerForm({
      model: employee
    });
    form.render();
    $("#wizard").html(form.el);
    return form;
  }
}


// implementation details for EmployeeDetailForm go here

// implementation details for SelectManagerForm go here

// implementation details for Employee model go here
```

I've obviously omitted some of the details of the views and model, but you get the idea.

The Challenge Of Workflow Objects

Everything has a price, right? But the price for this is fairly small. You will end up with a few more objects and a few more methods to keep track of. There's a mild overhead associated with this in the world of browser based JavaScript, but that's likely to be so small that you won't notice.

The real cost, though, is that you're going to have to learn new implementation patterns and styles of development in order to get this working, and that takes time. Sure, looking at an example like this is easy. But it's a simple example and a simple implementation. When you get down to actually trying to write this style of code for yourself, in your project, with your 20 variations on the flow through the application, it will get more complicated, quickly. And there's no simple answer for this complication in design, other than to say that you need to learn to break down the larger workflow in to smaller pieces that can look as simple as this one.

In the end, making an effort to explicitly model your workflow in your application is important. It really doesn't matter what language your writing your code in. I've shown these examples in JavaScript and Backbone because that's what I'm using on a daily basis at this point. But I've been applying these same rules to C#/.NET, Ruby and other languages for years. The principles are the same, it's just the implementation specifics that change.

Lessons Learned

There are a number of benefits to writing code like this. It's easy to see the high level workflow. We don't have to worry about all of the implementation details for each of the views or the model when dealing with the workflow. We can change any of the individual view implementations when we need to, without affecting the rest of the workflow (as long as the view conforms to the protocol that the workflow defines). And there's probably a handful of other benefits, as well.

Workflow Should Be Understandable At A Glance

The largest single benefit of writing code like this, is being able to see the workflow at a glance. 6 months from now – or if you're like me, 6 hours from now – you won't remember that you have to trace through 5 different Views and three different custom objects and models, in order to piece together the workflow that you spun together in the sample at the very top of this post. But if you have a workflow as simple as the one that we just saw, where the workflow is more explicit within a higher level method, separated from the implementation details… well, then you're more likely to pick up the code and understand the workflow quickly.

Learn To Recognize Concerns, So They Can Be Separated

Recognizing different types of concerns in code is not always easy. It takes experience to know when two things are part of the same concern, or are actually two separate concerns. Drawing a line in the sand between the high level workflow and implementation detail for that workflow is a good place to start. Separating these concerns allows you to modify how the work flows vs how the details of each step are implemented.

The Dependency Inversion Principle

Be careful not to fall in to the trap of "some of the implementation details changed, so the workflow has to change now". This is a dangerous path that leads to the dark side of code - tightly coupled spaghetti mess. Remember the Dependency Inversion Principle which states that details should depend on policy.

In the case of workflow, the workflow itself is the policy. It is the guidance that tells each step what it must look like from an API perspective. The workflow determines the API of each in the process - how to call a given step to run it, how to get any needed response from that step, what to do with that response, etc. Each individual step - the detail of the workflow - is then only responsible for the detail of that one step, and never any other steps.

Appendix A: Managing Events As Relationships

In my Scaling Backbone Apps With Marionette talk[10], I have some slides that deal with JavaScript zombies in Backbone apps[11]. This isn't a new subject by any means. It is one that I talk about a lot, and spend a lot of time explaining to others. But there is one aspect of this talk and the related material that I have only recently started using: the idea of managing event handlers as relationship and not simply object references. More importantly, though, correctly modeling the relationship between the observer (event handler) and the subject (event broadcaster) can give us insight in to our code and create a more natural representation of how we think about, understand, and observe the real world.

The Observer Pattern

Event systems in the object oriented world are almost always based on the classic Observer Pattern from the "Gang of Four" design patterns book (if you don't own this book, go buy it right now). According to Wikipedia, the observer pattern

> is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

To put that a little more plainly: something triggers an event and other things listen to and respond to the event.

The basic structural set up for this pattern includes three things:

- Event: the thing that happened; the action or outcome that has already occured
- Subject: an object that triggers or broadcasts the occurance of an event
- Observer: an object (or function) that listens for, and responds to an event

Backbone.Events is an implementation of the observer pattern at some level. We could argue about whether or not Backbone implements a "pure" observer pattern, but the important point is that this pattern is the basis for what Backbone and most other event systems implement. It informs us of the design and implementation of Backbone.Events and the object references and relationships involved.

But what is an event? What are the "subjects" and "observers"?

[10] https://speakerdeck.com/derickbailey/scaling-backbone-dot-js-applications-with-marionette-dot-js

[11] http://lostechies.com/derickbailey/2011/09/15/zombies-run-managing-page-transitions-in-backbone-apps/

An Event: Moving A Ball Through A Round Metal Hoop

Think about basketball for a moment. When a player scores, the team's points are increased, everyone adjusts their position to start the next play, fans cheer or complain, and the game goes on.

In this case, the "Subject" is the player – the person that scored the points. When the player is able to move the ball through the round metal hoop at the end of the court in an appropriate manner, a "points scored" event is triggered. The "observers", then, are comprised of a number of people: the other players, the fans, the score keeper, the ref, and many more people that are paying attention to the game. They go about their business of cheering, updating the scoreboard, and doing anything else that they are responsible for doing in response to the points being scored.

Keep this analogy in mind as I'll be coming back around to it in a bit. For now, though, this should give you a good idea of the various players involved in an Observer pattern.

Backbone.Events And References

When we set up an observer in Backbone, we typically use the .on method to tell the Subject about the Observer. In other words, the object that triggers the event will hold a reference to the object or function that handles the event.In the case of an in-line callback function, the reference that we hand to the Subject is a simple function:

```
// a subject triggers an event
subject.on("some:event", function(){
  // the callback function is the observer
  // of the event in this case. do stuff
  // here because the "some:event" event was
  // triggered
});
```

This is a simple reference, and we have no other direct references to the function. When myObject goes out of scope, then, this function reference will be garbage collected.

A common example of this in a Backbone.Collection would be the use of the "reset" event to process a collection being reset

```
myCollection.on("reset", function(){
  // re-render the entire collection, or
  // do something else with the collection
  // knowing that it was reset with new models
});
```

When using a method reference (that is, a function attached to another object) as the callback function, though, we are only handing the function reference to the Subject.

```
subject.on("some:event", anObserver.someHandlerMethod);
```

In this case, the Subject is only directly handed a reference to `anObserver.someHandlerMethod`. This method does not bring along a reference to the `anObserver` object, though. Passing a method like this is passing a method pointer and the method happens to be attached to the `anObserver` object.

There's also a potential problem in this code: the `someHandlerMethod` does not get cleaned up when the `anObserver` goes out of scope. Because `subject` has a reference to `someHandlerMethod`, it can't be cleaned up. This - an event handler that is not cleaned up - is one of the most common causes of zombie objects in JavaScript and Backbone.js apps. There are simple ways to solve this problems, though. We just need to remove the observer reference from the Subject when we're done:

```
subject.off("some:event", anObserver.someHandlerMethod);
```

This removes the Observer method reference and allows the objects to be properly garbage collected. It effectively double-taps the zombie before the zombie has a chance to re-animate.

While this code does handle the zombie problem, it doesn't always do it in the best way. The relationship between the Subject and the Observer is wrong for many of the cases that JavaScript and Backbone apps run in to.

Events And Relationships

When we call the `.on` and `.off` methods of a Backbone object, we are doing more than just setting up object references for an observer pattern implementation. We are setting up relationships and perspectives on those relationships in the mind of the developers reading and writing the code.

It's natural for us to think about event relationships in when/then, cause and effect logic. This is what we are taught in early education: "when this, then that," or in this case, "When this event fires, go do that." Statements like this tells us that the action is the primary actor and the reaction is secondary. This is great when looking at cause and effect in natural language, because it's true. Without the cause the effect would not be there. That's how cause and effect works. But the cause and effect relationship falls apart when looking at events and reactions to events – both in code and in the real world.

How we see objects and their relationships can make or break the maintainability and flexibility of a design and implementation for a software system. It's important to think through the relationships, then, and not just think about raw object references.

Scoring Points And Notifying Everyone

Think back to the basketball analogy and what happens when a basketball player scores. What did the player do? How do they react? If it was an amazing shot they may have celebrated. But, chances are they just moved on to the next play.

Now think about what the player didn't do. They didn't walk around to every other player on the court and tell them that they scored. The didn't walk over to the ref and tell him that points were scored. They didn't walk over to the score keeper and tell that person that points were scored. They didn't go tell … you get the idea.

The basketball player didn't tell anyone about the points being scored, because that is not the responsibility of the player. The ref, the score keeper, the other players, the fans – everyone involved in this sporting event observed the points being scored and reacted appropriately. The player that scored the points was not responsible for telling all of the observers that the event occurred.

Why, then, do we model events in code as if the basketball player, or perhaps the basket itself, is responsible for telling everyone when points are scored? Why do we write code like this?

```
basket.on("points:scored", function(team, player, points){
  team.updateScore(points);
  scoreboard.updateScore(team, points);
  player.updatePointsScored(points);
  // etc
});
```

The answer is object references. The Subject that triggers the event needs references to the Observer objects so that the observers can be notified. Developers typically look at this need for references and design an API around that. The observer pattern pretty much tells us to do that, too. But this relationship is backward and it needs to be fixed if we are going to write maintainable software systems.

The Relationship Problem: Who Owns It?

In the above basketball example, and in Backbone events in general, it's the Subject that owns the reference and therefore, the relationship. By calling basket.on we are telling the basket object to wait for it's own event to be triggered. When the event is triggered, it call the function that was supplied.

It it necessary for the basket object to hold a reference to the callback function because of the way object references work. If the Subject (basket) does not have a reference to the function that needs to be called… well… there's nothing to call when the event is triggered.

The real problem, then, is not object references. The real problem is how we create the references and who controls the relationship facilitated by those references.

Backbone.Events And Relationships

Calling `basket.on("points:scored", function(){…})` clearly sets up a relationship that is owned by the basket object. The basket is responsible for maintaining the reference, so it owns the relationship right? If we're thinking in terms of references then this makes sense. But if we think in

terms of relationships and how a basketball game actually operates, this doesn't make any sense. We don't want the Subject to own the relationships. We want the Observer to own the relationships.

A typical Backbone.View implementation illustrates why we want the Observer to own the relationship:

```
Backbone.View.extend({

  initialize: function(){
    this.model.on("change:foo", this.doStuff, this);
  },

  doStuff: function(foo){
    // do stuff in response to "foo" changing
  },

  remove: function(){
    this.model.off("change:foo", this.doStuff, this);

    // call the base type's method, since we are overriding it
    Backbone.View.prototype.remove.call(this);
  }

});
```

A typical Backbone.Model lives much longer than a view that works with it. The model will be displayed, edited and used for other parts of the application many times while a single view instance that works with it is stood up and torn down relatively quickly.

The zombie problem is also illustrated here. In order to prevent this view from becoming a zombie, we have to unbind the event handler that is set up in the initialize function. This is typically done by overriding the remove method and calling the necessary off event, as shown above.

From a functional perspective, this works perfectly fine. It might be a little annoying to type all of those off method calls, but this will clean up the references just fine. It doesn't model the relationship correctly, though. It still tells us that the model is in control of the references and relationship. The view has to ask the model to create the reference, and ask it to drop the reference. What we want instead, is a way for our view to say "I'm in control of this relationship. When I'm done, I'll sever the relationship. You don't have to worry about it, Model."

Inverting The Observer Relationship For Backbone.View

To invert the code structure and begin thinking about relationships, we need to have the view be in control. We need the view to say "I care about this event, and I will respond to it when it happens". Fortunately, Backbone provides the means to do this: the listenTo and stopListening methods from Backbone.Events.

The view from above can be re-written with .listenTo and .stopListening instead of .on and .off, very easily.

```
Backbone.View.extend({

  initialize: function(){
    this.listenTo(this.model, "change:foo", this.doStuff);
  },

  doStuff: function(foo){
    // do stuff in response to "foo" changing
  }

  // we don't need this. the default `remove` method calls `stopListening` for us
  // remove: function(){
    // this.stopListening();
    // ...
  //}
});
```

There are a few things of note, from a code perspective, here:

- We are calling the listenTo method of the view
- We are passing this.model as an argument to the listenTo method
- We are no longer passing this as the context variable for the last argument of listenTo
- We no longer need to override the remove method, since the default implementation calls stopListening for us

But far more important than the code difference is the shift in perspective. Instead of having code that says "Hey model, I'd like to register a callback method with you," we now have code that says, "I, the view, need to know when the model triggers this event and I, the view, will response appropriately." We also have have the view saying, "I, the view, am ending all relationships that I have previously set up." It's a subtle difference in how the relationship is managed, but it's a very important one as it more correctly models the relationships between the Subject and Observer.

Behind the scenes, the model is still being handed a reference to the view using the on method. There's simply no way around this technical detail. But we can (and should) hide this technical detail behind the veil of abstraction, allowing our code to tell us what relationship matters and who is in control of the references.

Zombie Killing Is My Business

There's another benefit, as well. That one call to `.stopListening()` inside of the remove method will sever all relationships that have been set up with the `.listenTo` method of the view. This cleans up all of the references that the model has to the view, preventing zombie views from having a chance to infect our apps.

This little detail alone is worth it's 100x its weight in the extra few characters that you have to type when using .listenTo.

No Silver Bullet For Modeling Relationships

It's tempting for me to say something like "stop using .on and .off" at this point, but that would be a bad idea. I think it would be safe to say that within the context of a Backbone.View, but there are other scenarios where this doesn't make sense.

In my post about modeling explicit workflow in JavaScript apps before, I talk about using a higher level object to coordinate the workflow of an application. This is a scenario where it might not make sense to use .listenTo and .stopListening.

```
MyApp.someWorkflow = {

  show: function(){
    var layout = new MyApp.LayoutView();

    layout.on("render", this.showDetail, this);

    this.showView(layout);
  },

  // ... showDetail, showView, etc
};
```

In this code, the layout will likely be closed relatively quickly, and the someWorkflow object will likely remain alive forever, as it has been attached to the global app object. This would be a bad place to tie the event reference cleanup to the higher level object. It would never get cleaned up. Instead, you would need to tie the lifecycle of the event handlers to the layout. The above code would be

correct in using .on to set up the event, then The layout being closed will remove the reference to the event handler and things will be cleaned up nicely.

There are likely other scenarios where using .on and/or .off would make more sense for the relationship management and reference management, still. Instead of blindly applying a pattern for handling zombies and relationship management, we need to understand the actual relationship between the objects in question.

Thinking In, And Modeling Relationships vs References

When we start working with relationships instead of just references, our code becomes easier to understand. We can model and code application and system designs that better express the way we think and the way we understand relationships in the real world. Under the hood, we still have to deal with references, but that doesn't mean our API has to expose these raw object references as the relationship we are creating.

More than an API difference, though, the idea of thinking about relationships instead of just references is just that – an idea. It's a shift in our mindset and in how we look at the code we are writing. It's a change in perspective to give us better insight in to how the things we are modeling actually interact and relate to each other. It informs our system design, our API implementation, how we look for and create abstractions, and how we organize our system in to smaller sub-sets that can be composed in to the larger whole.

Appendix C: A Tale Of Two Patterns

Design patterns often differ only in semantics and intent. That is, the language used to describe the pattern is what sets it apart, more than an implementation of that specific pattern. It often comes down to squares vs rectangles vs polygons. You can create the same end result with all three, given the constraints of a square are still met – or you can use polygons to create an infinitely larger and more complex set of things.

When it comes to the Mediator and Event Aggregator patterns, there are some times where it may look like the patterns are interchangeable due to implementation similarities. However, the semantics and intent of these patterns are very different. And even if the implementations both use some of the same core constructs, I believe there is a distinct difference between them. I also believe they should not be interchanged or confused in communication because of the differences.

It's All About Logic

The TL;DR is this: where does the logic live? An event aggregator has no application logic. It is purely infrastructure, forwarding events from a publisher to a subscriber. A mediator, on the other hand, encapsulates the potentially complex logic of application workflow. It coordinates multiple objects and/or services to accomplish a goal within the application. A mediator contains real application / business / workflow / process logic.

Event Aggregator

The core idea of the Event Aggregator, according to Martin Fowler, is to channel multiple event sources through a single object so that other objects needing to subscribe to the events don't need to know about every event source.

Backbone's Event Aggregator

The easiest event aggregator to show is that of Backbone.js – it's built in to the Backbone object directly.

```
var View1 = Backbone.View.extend({
  // ...

  events: {
    "click .foo": "doIt"
  },

  doIt: function(){
    // trigger an event through the event aggregator
    Backbone.trigger("some:event");
  }
});

var View2 = Backbone.View.extend({
  // ...

  initialize: function(){
    // subscribe to the event aggregator's event
    Backbone.on("some:event", this.doStuff, this);
  },

  doStuff: function(){
    // ...
  }
})
```

In this example, the first view is triggering an event when a DOM element is clicked. The event is triggered through Backbone's built-in event aggregator – the Backbone object. Of course, it's trivial to create your own event aggregator in Backbone, and there are some key things that we need to keep in mind when using an event aggregator, to keep our code simple.

jQuery's Event Aggregator

Did you know that jQuery has a built-in event aggregator? They don't call it this, but it's in there and it's scoped to DOM events. It also happens to look like Backbone's event aggregator:

```
$("#mainArticle").on("click", function(e){

  // handle the click event from any element
  // underneath of the #mainArticle

});
```

This code sets up an event handler function that waits for an unknown number of event sources to trigger a "click" event, and it allows any number of listeners to attach to the events of those event publishers. jQuery just happens to scope this event aggregator to the DOM.

Mediator

A Mediator is an object that coordinates interactions (logic and behavior) between multiple objects. It makes decisions on when to call which objects, based on the actions (or in-action) of other objects and input.

A Mediator For Backbone

Backbone doesn't have the idea of a mediator built in to it like a lot of other MV* frameworks do. But that doesn't mean you can't write one in 1 line of code:

```
var mediator = {};
```

Yes, of course this is just an object literal in JavaScript. Once again, we're talking about semantics here. The purpose of the mediator is to control the workflow between objects and we really don't need anything more than an object literal to do this.

```
var orgChart = {

  addNewEmployee: function(){

    // getEmployeeDetail provides a view that users interact with
    var employeeDetail = this.getEmployeeDetail();

    // when the employee detail is complete, the mediator (the 'orgchart' object)
    // decides what should happen next
    employeeDetail.on("complete", function(employee){

      // set up additional objects that have additional events, which are used
      // by the mediator to do additional things
      var managerSelector = this.selectManager(employee);
```

```
15        managerSelector.on("save", function(employee){
16          employee.save();
17        });
18
19      });
20    },
21
22    // ...
23  }
```

This example shows a very basic implementation of a mediator object with Backbone based objects that can trigger and subscribe to events. I've often referred to this type of object as a "workflow" object in the past, but the truth is that it is a mediator. It is an object that handles the workflow between many other objects, aggregating the responsibility of that workflow knowledge in to a single object. The result is workflow that is easier to understand and maintain.

Mediators And Workflow Components

For more information on building mediators that scale better, and offer more features, see chapters 13 and 14.

Similarities And Differences

There are, without a doubt, similarities between the event aggregator and mediator examples that I've shown here. The similarities boil down to two primary items: events and third-party objects. These differences are superficial at best, though. When we dig in to the intent of the pattern and see that the implementations can be dramatically different, the nature of the patterns become more apparent.

Events

Both the event aggregator and mediator use events, in the above examples. An event aggregator obviously deals with events – it's in the name after all. The mediator only uses events because it makes life easy when dealing with Backbone, though. There is nothing that says a mediator must be built with events. You can build a mediator with callback methods, by handing the mediator reference to the child object, or by any of a number of other means.

The difference, then, is why these two patterns are both using events. The event aggregator, as a pattern, is designed to deal with events. The mediator, though, only uses them because it's convenient.

Third-Party Objects

Both the event aggregator and mediator, by design, use a third-party object to facilitate things. The event aggregator itself is a third-party to the event publisher and the event subscriber. It acts as a central hub for events to pass through. The mediator is also a third party to other objects, though. So where is the difference? Why don't we call an event aggregator a mediator? The answer largely comes down to where the application logic and workflow is coded.

In the case of an event aggregator, the third party object is there only to facilitate the pass-through of events from an unknown number of sources to an unknown number of handlers. All workflow and business logic that needs to be kicked off is put directly in to the the object that triggers the events and the objects that handle the events.

In the case of the mediator, though, the business logic and workflow is aggregated in to the mediator itself. The mediator decides when an object should have it's methods called and attributes updated based on factors that the mediator knows about. It encapsulates the workflow and process, coordinating multiple objects to produce the desired system behavior. The individual objects involved in this workflow each know how to perform their own task. But it's the mediator that tells the objects when to perform the tasks by making decisions at a higher level than the individual objects.

An event aggregator facilitates a "fire and forget" model of communication. The object triggering the event doesn't care if there are any subscribers. It just fires the event and moves on. A mediator, though, might use events to make decisions, but it is definitely not "fire and forget". A mediator pays attention to a known set of input or activities so that it can facilitate and coordinate additional behavior with a known set of actors (objects).

Relationships: When To Use Which

Understanding the similarities and differences between an event aggregator and mediator is important for semantic reasons. It's equally as important to understand when to use which pattern, though. The basic semantics and intent of the patterns does inform the question of when, but actual experience in using the patterns will help you understand the more subtle points and nuanced decisions that have to be made.

Event Aggregator Use

In general, an event aggregator is uses when you either have too many objects to listen to directly, or you have objects that are unrelated entirely.

When two objects have a direct relationship already – say, a parent view and child view – then there might be little benefit in using an event aggregator. Have the child view trigger an event and the parent view can handle the event. This is most commonly seen in Backbone's Collection and Model, where all Model events are bubbled up to and through it's parent Collection. A Collection often uses

model events to modify the state of itself or other models. Handling "selected" items in a collection is a good example of this.

jQuery's on method as an event aggregator is a great example of too many objects to listen to. If you have 10, 20 or 200 DOM elements that can trigger a "click" event, it might be a bad idea to set up a listener on all of them individually. This could quickly deteriorate performance of the application and user experience. Instead, using jQuery's on method allows us to aggregate all of the events and reduce the overhead of 10, 20, or 200 event handlers down to 1.

Indirect relationships are also a great time to use event aggregators. In Backbone applications, it is very common to have multiple view objects that need to communicate, but have no direct relationship. For example, a menu system might have a view that handles the menu item clicks. But we don't want the menu to be directly tied to the content views that show all of the details and information when a menu item is clicked. Having the content and menu coupled together would make the code very difficult to maintain, in the long run. Instead, we can use an event aggregator to trigger "menu:click:foo" events, and have a "foo" object handle the click event to show it's content on the screen.

Mediator Use

A mediator is best applied when two or more objects have an indirect working relationship, and business logic or workflow needs to dictate the interactions and coordination of these objects.

A wizard interface is a good example of this, as shown with the "orgChart" example, above. There are multiple views that facilitate the entire workflow of the wizard. Rather than tightly coupling the view together by having them reference each other directly, we can decouple them and more explicitly model the workflow between them by introducing a mediator.

The mediator extracts the workflow from the implementation details and creates a more natural abstraction at a higher level, showing us at a much faster glance what that workflow is. We no longer have to dig in to the details of each view in the workflow, to see what the workflow actually is.

Event Aggregator And Mediator Together

The crux of the difference between an event aggregator and a mediator, and why these pattern names should not be interchanged with each other, is illustrated best by showing how they can be used together. The menu example for an event aggregator is the perfect place to introduce a mediator as well.

Clicking a menu item may trigger a series of changes throughout an application. Some of these changes will be independent of others, and using an event aggregator for this makes sense. Some of these changes may be internally related to each other, though, and may use a mediator to enact those changes. A mediator, then, could be set up to listen to the event aggregator. It could run it's logic and process to facilitate and coordinate many objects that are related to each other, but unrelated to the original event source.

```
var MenuItem = Backbone.View.extend({

  events: {
    "click .thatThing": "clickedIt"
  },

  clickedIt: function(e){
    e.preventDefault();

    // assume this triggers "menu:click:foo"
    Backbone.trigger("menu:click:" + this.model.get("name"));
  }

});

// ... somewhere else in the app

var MyWorkflow = function(){
  Backbone.on("menu:click:foo", this.doStuff, this);
};

MyWorkflow.prototype.doStuff = function(){
  // instantiate multiple objects here.
  // set up event handlers for those objects.
  // coordinate all of the objects in to a meaningful workflow.
};
```

In this example, when the MenuItem with the right model is clicked, the "menu:click:foo" event will be triggered. An instance of the "MyWorkflow" object, assuming one is already instantiated, will handle this specific event and will coordinate all of the objects that it knows about, to create the desired user experience and workflow.

An event aggregator and a mediator have been combined to create a much more meaningful experience in both the code and the application itself. We now have a clean separation between the menu and the workflow through an event aggregator. And we are still keeping the workflow itself clean and maintainable through the use of a mediator.

Pattern Language: Semantics

There is one overriding point to make in all of this discussion: semantics. Communicating intent and semantics through the use of named patterns is only viable and only valid when all parties in a communication medium understand the language in the same way.

If I say “apple”, what am I talking about? Am I talking about a fruit? Or am I talking about a technology and consumer products company? As Sharon Cichelli says: “semantics will continue to be important, until we learn how to communicate in something other than language”.

About The Book Sample

Thank you for taking the time to download and read this small sample of my book! I hope you have enjoyed it, and I hope that this sample will help convince you of the value that the full book provides.

If you have any questions, comments or concerns about purchasing the full book, please feel free to contact me:

- Email: backboneplugins@mutedsolutions.com[12]
- Twitter: @BackbonePlugins[13]
- Website: BackbonePlugins.com[14]

About Me

Hello, my name is Derick Bailey.

I'm a software developer, consultant, screencaster, blogger, speaker, trainer and more. I've been working professionally in software development since the late 90's and have been writing code since the late 80's, and am currently working as a Developer Advocate for Kendo UI[15].

- Employer: Telerik[16].
- Position: Developer advocate for Kendo UI[17]
- Personal Company: MutedSolutions.com[18]
- Technical Blog: DerickBailey.LosTechies.com[19]
- Screencasts (free and paid): WatchMeCode.net[20]
- Open Source Projects: GitHub.com/DerickBailey[21]

[12] mailto:backboneplugins@mutedsolutions.com
[13] http://twitter.com/backboneplugins
[14] http://backboneplugins.com
[15] http://kendoui.com
[16] http://telerik.com
[17] http://kendoui.com
[18] http://mutedsolutions.com
[19] http://derickbailey.lostechies.com
[20] http://watchmecode.net
[21] http://github.com/derickbailey