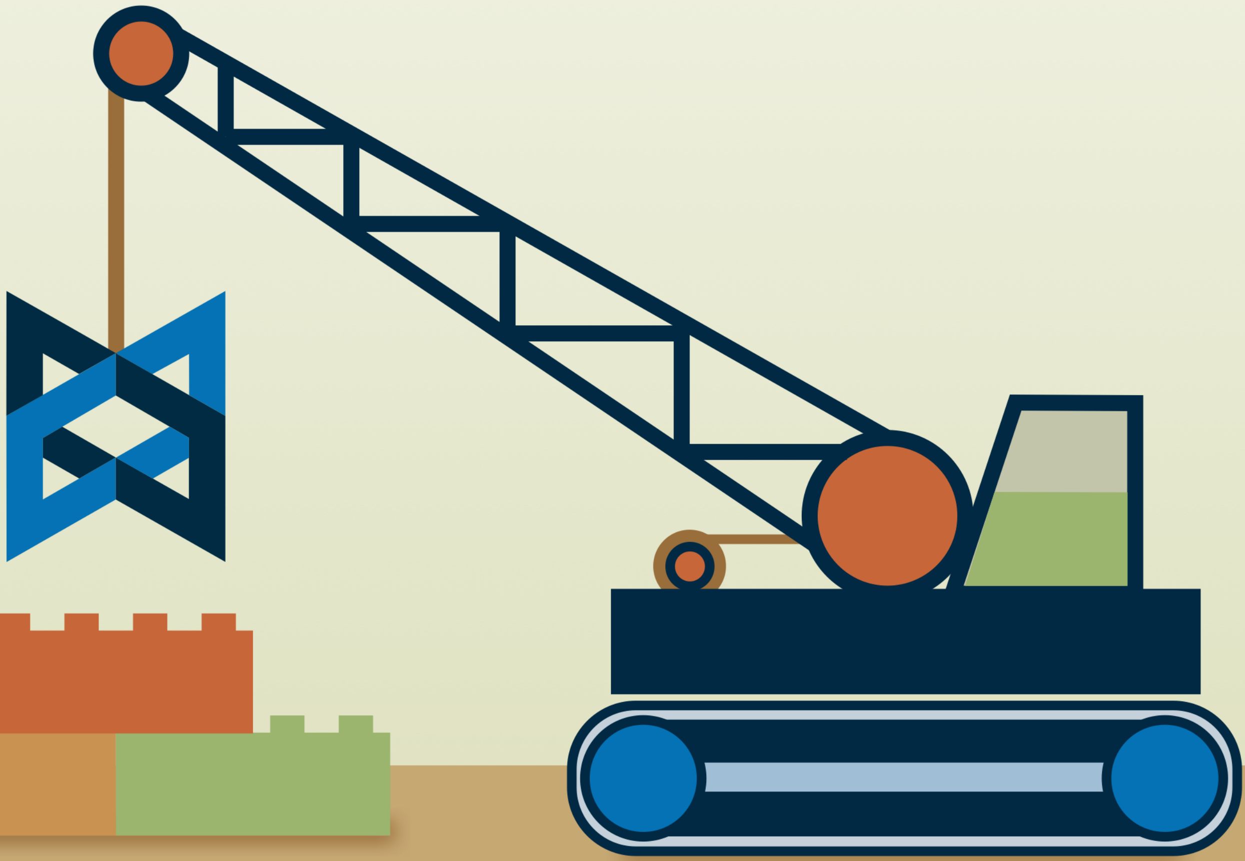
Covers Backbone.js v1.0

# 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

# 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.

# In Relation To MarionetteJS (Backbone.Marionette)

The majority of knowledge contained in this book came directly from my experiences in building MarionetteJS[9] and in building applications with MarionetteJS. Of course there are other examples and plugins that are used in this book. You will find, though, that most of the information found in here is already found in MarionetteJS' implementation. Base views, model views, collection views, regions, application objects and workflow… all of this and more, is available in the MarionetteJS framework for Backbone.

However, the vast majority of features and behaviors found in MarionetteJS are implemented differently than those in this book. This is done with purpose, as Marionette has gone through the rigors of hundreds of applications and thousands of developers using it. It would take an insurmountable effort to explain every last line of code in Marionette, quite frankly. This book aims to simplify the most common and core knowledge for building a framework like Marionette. The value in this book, then, is not that you will build MarionetteJS on your own. It is that you will have first hand knowledge of the decision making process for building a framework like MarionetteJS.

Having this knowledge encoded in Marionette and simplified for this book does not diminish the value of the content of this book or the MarionetteJS framework, though. Rather, I am attempting to provide a clear understanding of why MarionetteJS is built the way it is. I want to extend the knowledge of MarionetteJS users as well as the capabilities of Backbone users that do not wish to use MarionetteJS.

And in the end, I am not trying to sell anyone on using MarionetteJS for their applications. My greatest hope, in fact, is that other developers will find the knowledge in this book so empowering in building their Backbone applications, that they will want to create their own frameworks and plugins. Whether you want to build your own plugins and frameworks or use MarionetteJS, ChaplinJS, Backbone.Giraffe, Backbone.LayoutManager, Thorax or any of the other great application frameworks available, though, I hope that this book will give you a better understanding of how to better use BackboneJS.

[9]http://marionettejs.com

# Copyrights, Code License And Distribution

This book is meant to be a teaching tool, to help you understand how to build better software with Backbone.js. As such, I expect that you will want to copy the code and text from the book and use it in your own projects. Please do! I encourage this, greatly!

## License And Use Of Code

Build your own libraries, add-ons and plugins for Backbone or any other project you are working with. Use the code found in this book as a starting point and customize it to your needs. Or copy and paste it directly, and be done! Have fun, be productive and show the world how awesome your code is. The only thing I ask, is that you respect the copyright and license of the original code that I am writing within this book.

**All code written for this book is freely distributable under the MIT License[10].**

Code from existing and other libraries will remain copyrighted and licensed as outlined in those projects.

## License And Use Of The Text

I hold the same basic principles for ideas as I do for code. You are free to share the ideas and the basic content of this book, with anyone you want, whenever and wherever you want. You are free to copy and paste text, share it, quote the contents of this book and use the ideas in your own projects and writings. I only ask that you keep this within reason and you provide appropriate attribution to the source of your material.

**Please do not re-post, re-publish, copy and paste, or re-distribute any significant portion of any part of the text in this book book, or the book as a whole, without permission** (where "significant portion" is more than 25% of an individual chapter, appendix, or other section of the book).

If you wish to publish or distribute more than a reasonable amount of this book, please contact me. You can find my contact information at the end of the book.

[10] http://mutedsolutions.mit-license.org

# Part 1: Backbone Views

Nearly every developer that works with Backbone quickly realizes that while Backbone is very flexible and powerful, it also leaves a lot of “boilerplate” code and small decisions up to the developer. Boilerplate is the code that has to be written all the time, every time, just to get the project or component off the ground. It’s the code that is necessary because the desired functionality won’t work without it. It’s the repetition and redundancy of copy & paste from object to object. But it is also the code from which patterns and re-usable snippets are often created.

The first place that developers see boilerplate is usually working with views. Views tend to require a number of things to work properly: data serialization, HTML template definition, template compilation and rendering, and more.

In part 1, the boilerplate of views will be eliminated. A generic render method will be built for views. Specialized view types will be extended from a custom a base type. The ability to add new view types that offer flexibility through code and configuration will be there, as well.

# 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:

```
Backbone.View.extend({

  render: function(){

    var template = "<h1><% _.each(items, function(item){ %><%= message %> <% }) %\
>!</h1>";
    var data = {
      items: this.collection.toJSON()
    };

    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 2: Naming And Namespacing Your Plugin

JavaScript has the notion of "global" variables and objects within a browser. Every object, variable or function that is defined in the browser's JavaScript runtime lives in the global scope by default. This is not a big deal with very small applications. But when an application begins to grow, and begins to use plugins such as jQuery, Underscore and Backbone, it becomes very important to not "pollute" the global scope with too many objects.

The risk of too many global objects or variables is that they will have the same name as another part of the project or another library. How many projects have a `config` object or method, for example? If they all used the same `config` object in the global scope, nothing would work together as they would be clobbering each other's global object.

Building namespaced code is one way to avoid polluting the global space, and most popular libraries do this in some way. Backbone provides the `Backbone` namespace, for example. The `View`, `Model`, and `Collection` types all hang off of this namespace. And the `BaseView` that I just created should follow suit. It should be namespaced to either my application or the name of the plugin that I am building.

Unfortunately, JavaScript doesn't have a formal concept of namespaces, like Java, C#, Ruby and other languages do. But that doesn't mean we can't use a namespace pattern to facilitate the same functionality. In fact, there are many options for making this work, including:

- Object literal namespace objects
- The Revealing Module pattern
- … and more variations

Any object can be used as a namespace. All I need to do is assign one object to an attribute of another object, and I effectively have a namespace. This can be a dangerous thing to do, though, as I may accidentally override existing data or functionality on an exsiting object. Instead, it is a better idea to use an empty object literal as the namespace. This way, I am less likely to override something important.

To create an object literal as a namespace, just create an object literal and then assign objects as attributes. For example, the `BaseView` can be attached to a `BBPlug` namespace (short-hand for "Building Backbone Plugins") like this:

```
var BBPLug = {};

BBPlug.BaseView = Backbone.View.extend({

  render: function(){
    var data;

    if (this.serializeData){
      data = this.serializeData();
    };

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

To extend from this object, I must specify the entire path to the object, `BBPlug.BaseView`:

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

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

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

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

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

For a more complete list of options and possibilities for namespacing and providing protection for your code, and why, see Stoyan Stefanov's JavaScript Patterns[11] from O'Reilly press.

[11] http://shop.oreilly.com/product/9780596806767.do

Now the three views that I am working with are all extending from the correct `BBPlug.BaseView`. Any additional objects, functions, data, configuration or other bits that my plugin needs will also hang from the `BBPlug` object. This way, my plugin will not interfere with other plugins and libraries. I only need to be careful to remember to use the `BBPlug` namespace and always declare my variables with the `var` keyword. But I honestly don't worry about this too much. Tools such as JSHint will analyze my code and ensure that I am correctly namespacing and declaring everything.

This simple object literal as namespace will be sufficient for now. Later, the Revealing Module pattern will be used to help create privacy and encapsulation. But that can wait.

## Lessons Learned

Java, C# and other languages provide an explicit construct for namespacing, but JavaScript does not. This does not mean that the concept is not important, though. Namespacing can and should be done in JavaScript. It just has to be done within the constructs that JavaScript provides - as more of an idea than an explicit syntax.

### Don't Pollute The Global Namespace

Namespacing our abstractions is very important. If we don't provide a single, global object for all of our code to hang off of, we run the risk of either breaking someone else's code or library, or having another developer or library break our code. There are several options for implementing a namespace in JavaScript, and none of them are specific to Backbone. But Backbone provides a good example of using a namespace, and we created our own with a simple object literal.

# Chapter 3: Complex And Divergent Views

The `BBPlug.BaseView` will allow an application to have a consistent method of rendering templates and data. There are some rather restricting limitations in it, though. If a view has any custom code that needs to run on the rendered HTML, for example, it can't - not easily at least. Additionally, the requirement of providing a `serializeData` method will create some duplication in all of our views - which is what we were setting out to avoid in the first place.

Fortunately, these limitations on flexibility and capabilities are solvable at the same time that the code duplication can be significantly reduced.

## onRender

In an application of any size beyond a simple ToDo app, a number of views will need to have some very specialized code run against the HTML that is rendered. For example, if a view is rendering a list of comments for a blog post, it may be desirable to show when the individual comments were added using the standard "(time) ago" format: "10 minutes ago", "2 months ago", "over a year ago", etc. This format can be easily obtained using the moment.js[12] plugin, but where would the call to this plugin go in the current `BaseView` implementation?

With the current implementation, the BaseView's `render` method may have to be overridden. This isn't too bad, I guess. I can call in to the BaseView's render method through it's prototype, allowing the standard render to take place and then running my code to format the dates correctly.

```
CommentView = BBPlug.BaseView.extend({
  template: "...",
  serializeData: function(){...},

  render: function(){
    // use the BaseView's prototype to do the default render
    BBPlug.BaseView.prototype.render.call(this);

    // call moment.js' to format our dates correctly
    var commentDate = this.model.get("date");
    var timeAgo = moment(commentDate).fromNow();
```

[12] http://momentjs.com

```

    // update this view's DOM with the correct format
    this.$(".comment-date").text(timeAgo);
  }
});
```

This is functional. It will correctly show the "(time) ago" format that the view needs. But it does so at the cost of an undue burden on the developer that is using this plugin. Every time this code is needed, the developer that is writing it has to remember to call the prototype's `render` method. Now there may be some scenarios where this is necessary, but something as common as manipulating the view's DOM elements after rendering is going to produce a lot of code duplication - boilerplate - and create a host of potential problems, like forgetting to call the prototype method.

To correct this, the `BaseView` can allow an `onRender` function to be specified on views that extend from it. If this method is provided, it will be called after the view has been rendered. This will allow a specific view to modify the DOM as needed, without having to override the render method.

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

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

    // Call the `onRender` method if it exists
    if (this.onRender){
      this.onRender();
    }
  }
});
```

Now I can modify the CommentView to use the `onRender` method instead of overriding the `render` method directly:

```
CommentView = BBPlug.BaseView.extend({
  template: "...",
  serializeData: function(){...},

  onRender: function(){
    // call moment.js' to format our dates correctly
    var commentDate = this.model.get("date");
    var timeAgo = moment(commentDate).fromNow();

    // update this view's DOM with the correct format
    this.$(".comment-date").text(timeAgo);
  }
});
```

This may only save one or two lines of code and one comment in the view, but when that one line is added up 20, 30, or 300 times in a very large application, it can make a significant difference.

## Default serializeData Implementation

There are still some chunks of boilerplate code in the CommentView and previous view implementations from the last chapter. Each of the views, though it has extended from the `BBPlug.BaseView`, must provide a `serializeData` method if it needs data in the template. Most of the time the data that is needed will be one of two things: `this.model.toJSON()` or `{items: this.collection.toJSON()}`. Rather than repeating this code in each view that needs it, a default `serializeData` implementation can be provided.

```
BBPlug.BaseView = Backbone.View.extend({

  serializeData: function(){
    var data;

    if (this.model){
      data = this.model.toJSON();
    }

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

    return data;
  },


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

By including this default implementation of `serializeData`, a view that needs either a model or a collection as its data source will not have to provide its own version of this method.

```
// Model View
// ----------
MyView = BBPlug.BaseView.extend({
  template: "..."
});

myView = new MyView({
  model: myModel
});

myView.render();


// Collection View
// ---------------

MyCollectionView = BBPlug.BaseView.extend({
  template: "..."
});

myCollectionView = new MyCollectionView({
  collection: myCollection
});

myCollectionView.render();
```

Of course any view that needs any custom data can still provide an implementation, but it is not strictly necessary.

## Extracting ModelView And CollectionView

There is one potential problem that this solution creates. If a view happens to have both a `model` and a `collection` set when the `serializeData` function is executed, then the collection will always win. If the view intends to render the model as the data source for the template and only needs the collection as a supporting object for some other purpose, then there's a problem. The view has rendered the collection data instead of the model data. To fix this, the view would have to either override the `serializeData` method to only use the model, or provide the collection to the view in a different manner. While either of these options would work, they are not the most elegant solution.

A better solution - one that creates more flexibility, more capabilities and less ambiguous use - would be to separate the ideas of a `ModelView` and a `CollectionView`. Each of these view types would provide the specific `serializeData` method that it needs, and open up the potential for other features to be added to a specific view type.

The extraction of these two view types does not negate the need for the `BaseView`, though. Instead, it solidifies the usefulness of this base view type. The `ModelView` and `CollectionView` can extend directly from it, keeping the core rendering logic in one place. But the `BaseView` will no longer need a default implementation of `serializeData`, so this method can be removed.

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

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

    // Call the `onRender` method if it exists
    if (this.onRender){
      this.onRender();
    }
  }
});

// Create a `ModelView` view type, to render a model in to the template
BBPlug.ModelView = BBPlug.BaseView.extend({

  serializeData: function(){
    var data;

    if (this.model){
      data = this.model.toJSON();
    }

    return data;
  }

});

// Create a `CollectionView` view type, to render a collection in to the template
BBPlug.CollectionView = BBPlug.BaseView.extend({

  serializeData: function(){
    var data;

    if (this.collection){
      data = this.collection.toJSON();
    }

    return data;
  }

});
```

In both of these new view types, the `render` method from the base view will be used, but the `serializeData` method from the specific view type will be used.

With these new view types in place, the blog post with comments example could be updated so that the blog post itself renders as a `ModelView` while the comments are rendered as a `CollectionView`.

```
BlogPostView = BBPlug.ModelView.extend({
  template: "..."
});

CommentsView = BBPlug.CollectionView.extend({
  template: "..."
});

var blogPostView = new BlogPostView({
  model: blogPost
});

var commentsView = new CommentsView({
  collection: blogPost.comments
});
```

The view definitions for a blog post and the related comments view are now very small. They each only have one line of code, specifying the template that the view needs to use. The remaining code to serialize the data and render the template with that data has been abstracted in to various view types in the `BBPlug` framework. This is a significant reduction in the amount of code that was required even to render a "Hello World" example.

## Lessons Learned

This chapter has provided a much more in-depth exploration of what can be included in a framework of View types that sit on top of Backbone. Starting with the basic `BBPlug.BaseView`, the need to cleanly separate a `ModelView` and `CollectionView` was identified and codified. Additional reasons for providing abstractions and reusable code were also identified, resulting in several new lessons that can be taken and applied to other view types as well as other object types in general.

### Remove Boilerplate. Don't Just Move It.

The initial implementation of `BBPlug.BaseView` required every view to implement a `serializeData` function. This was fine at first, when only a few distinct views were being used. But as new views were identified, the realization that each of these views fell in to the same patterns of what the `serializeData` implementation should look like set in. The BaseView had removed some boilerplate code, but in the process it had also just moved some of it. A simple default implementation of `serializeData` was created and the amount of code that each view was required to implement was reduced.

If the extraction of common functionality requires a specific function to be implemented, or specific data to be provided, ask whether or not the extracted code can have a default implementation. Does the BaseView's `render` method truely require a `template`, for example? In this case yes. Rendering HTML without a template to render wouldn't prove to be of much use. But when the question is asked about the `serializeData` method, the answer was yes, a default implementation could be used.

### Specialization Comes From Generalization

In the process of creating the default `serializeData` method, two separate and specialized use cases became apparent for views: a model view, and a collection view. The initial difference between these two view types was only the way the `serializeData` method was implemented. The `ModelView` uses the view's model for data, while the `CollectionView` used the view's collection.

By creating specialized views from the generalized `BaseView`, though, more flexibility was achieved and more opportunities were opened up. In the future, there may be additional behavioral and implementation changes that are common only to views that render collections. As these specialized needs are discovered, they can be added directly to the `CollectionView` without affecting the `BaseView` or `ModelView` directly.

## Don’t Require Calls To The Prototype’s Method

In many cases a simple implementation of a common function will prove to be insufficient. Specific implementations and use cases will have specific needs. In some cases it may be necessary to override a method that a base type implements, and then call the method on that base prototype before providing the specific implementation needs.

Whenever possible, though, this should be avoided. The burden of having to remember to call a prototype’s method can cause mistakes and bugs in systems. Instead, provide calls to optionally implemented methods such as the `onRender` method to allow custom code in specific view types. This method, when implemented in a view that extends from `BaseView`, allows a specific view to perform any necessary DOM manipulation or other functions just after the view has been rendered. It allows the code to be encapsulated within the view where it is needed, and it also keeps the implementor from having to remember to call a prototype method to facilitate the actual rendering.

## Hide Calls To Prototype Methods When They Are Necessary

There will inevitably be situations where calls to a prototype method are necessary. In these situations, the prototype method calls should be hidden in the implementation of the type that is being extended. The developer that is extending from these types should not be responsible for making the prototype method call, unless they choose to explicitly override a prototype method.

This means that in a view implementation, code that needs to be executed on construction or initialization of the view should not implement an `initialize` method. It’s common practice for a view to have an `initialize` method to set up event handlers, etc. Implementing the `initialize` method on a base class would force all views that need to use this method to call to the prototype’s initialize method. Instead, implement a `constructor` function in base views so that the need to call the prototype’s method can be hidden in the base view. This allows a specific view to implement an `initialize` method as needed, while still allows the base view type to run custom code on construction of a new view instance.

# Chapter 4: Building A Better CollectionView

Splitting apart the concepts of a CollectionView and ModelView opens a number of opportunities for improvements. While there may be some improvements that can be made to the ModelView, the CollectionView stands to gain the most ground by recognizing some of the common use cases and patterns for collections.

## Rendering A Collection Of Views

One of the most common scenarios that a CollectionView needs to handle, is the ability to render a list of models in to individual views. This allows for an individual view per model, with each view instance to reference a single model. From this perspective, each view is a ModelView. This means the view definition for the individual models can act as a ModelView and focus it's code entirely on the single model.

It is not difficult to get this capability in a CollectionView, as is.

```
var MyModelView = BBPlus.ModelView.extend({
  template: "#some-model-template",

  events: {
    "click #foo": "fooClicked"
  },

  fooClicked: function(e){
    // do stuff with this.model, here
  }
});

var CollectionOfModelsView = CollectionView.extend({
  render: function(){
    var html = [];

    // render a model view for each model
    // and push the results
    this.collection.each(function(item){
      var view = new MyModelView({model: item});
      html.push(view.render().$el);
    });

    // populate the collection view
    // with the rendered results
    this.$el.html(html);

    return this;
  }
});
```

This very simple view will loop through every model in it's collection and render a new instance of MyModelView for that model. Each instance of MyModelView only have a reference to the individual model that it renders. The model views never have to worry about dealing with the collection.

As simple as it is to do this, though, there are a number of problems that this creates. The views are rendered, but what happens when you need to close the parent view? The children need to be closed too, or you'll end up with zombies. And what happens when you add a new model to the collection, or remove a model from the collection? These should be handled by the CollectionView automatically, to reduce boilerplate code again. There should also be an easy way to specify the model view to use, instead of having it hard coded in to the render method. And lastly, with the list of models being iterated and rendered, it isn't really necessary to have an implementation of a `serializeData` method anymore.

## API-First Design Of The New CollectionView

Designing the API and usage of the new CollectionView is fairly easy - there isn't much that needs to be configured. You will want to specify a view type to use for each model, and that's about it. Anything else will be customization that can come later, as you need it.

```
var MyModelView = BBPlug.ModelView.extend({
  template: "#some-model-template"
});

var MyCollectionView = BBPlug.CollectionView.extend({
  modelView: MyModelView
});

var myColView = new MyCollectionView({
  collection: myCollectionOfStuff
});
```

This usage is simple and clean. It is obvious what you are intending, and the amount of code that you will reduce in view definitions is significant.

### Collection Views And Templates

Note that there is no need to specify a template for a collection view, since the models each render their own template. Allowing a template on the collection view would add a lot of complexity, too. Where do you render the items, in the template? You have to specify a way to figure that out. And what if the collection view simply doesn't need a template? Are you forced to specify a template and have it be empty? All these questions, and more, led MarionetteJS[13] to separate the CollectionView from what it calls a CompositeView - named after the composite design pattern[14], which is used in creating nested and hierarchical structures.

This minimalist API set for a collection view is not without power, though. Since this view type ultimately inherits from Backbone.View, it will still retain all of the capabilities of this base view, and the base BBPlug.View that it directly extends from. This means you will be able to specify the `tagName` and other attributes of the collection view.

## Declaring The ModelView First

There is one very important point to make in the above example, that has nothing to do with the CollectionView API design. You must declare and define the ModelView type **before** the CollectionView type that uses it. This is due to the way JavaScript performs "variable hoisting" - the process by which JavaScript parsers will move all variable declarations to the top of the function scope, but leave the variable assignment at the location specified in the code.

What this means, in practice, is this code will fail:

```
var MyCollectionView = BBPlug.CollectionView.extend({
  modelView: MyModelView
});

var MyModelView = BBPLug.ModelView.extend({
  // ...
});
```

You won't get a Javascript parsing error with this code, though. You won't know that there is a problem until runtime, in fact. JavaScript parsers see the `MyModelView` and `MyCollectionView` variables on the first pass of parsing, declaring them and making them available from anywhere in

[13]http://marionettejs.com
[14]http://en.wikipedia.org/wiki/Composite_pattern

the containing function. But the assignment does not happen until the line of code that makes the assignment runs. In this case, `MyModelView` will be undefined when it is assigned to the `modelView` attribute. Setting `MyModelView` to a view definition later, will not cause the CollectionView's definition to be updated, and you will have an undefined modelView on your hands.

### Variable Hoisting? Function Scope?

These are some of the most confusing aspects of JavaScript, if you ask me. They are simple rules that are not at all obvious, and can cause a lot of bugs, easily. For more information about these aspects of JavaScript, see my Variable Scope In JavaScript[15] screencast.

## Rendering Models With A `modelView` Setting

Allowing the use of a `modelView` setting in a view configuration is the most simple of the needed changes. Instead of hard coding the view type in the render method, create a method called `getModelView` and have that method retrieve the view type to create, for each model. Call that method from the render method, inside of the loop for rendering each model.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({

  // a method to get the type of view for
  // each model. this method can be overridden
  // to return a different view type based on
  // attributes of the model passed in
  getModelView: function(model){
    return this.modelView;
  },

  render: function(){
    var html = [];

    // render a model view for each model
    // and push the results
    this.collection.each(function(item){

      // get the view type to use
      var ViewType = this.getModelView(item);

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

[15] http://www.watchmecode.net/javascript-scope

```
      view.render();
      html.push(view.$el);
    }, this);

    // populate the collection view
    // with the rendered results
    this.$el.html(html);

    return this;
  }

});
```

The `getModelView` method is a little odd, at this point. It takes in a parameter of `model` but it never uses that parameter. This is done because there will be a day when you want to change the type of view that is used, based on some data found within the model. For example, you may be doing something with social media, and you might want to have models with data that come from Twitter, Facebook, MySpace, or whatever out-dated and irrelevant service was popular at the time this book was written. Having the ability to pick which view type is used for rendering means you can handle each of these networks differently.

The one drawback to allowing the `getModelView` function to change the view type based on the model, is that you have to call this function for every model in the collection. This adds a little overhead in comparison to calling it once and only once, per render. There is little chance that this will have a noticeable impact on your application's performance, though. Unless you're rendering thousands of items, which would be a performance problem in so many other ways, you are likely not going to have any issues with this.

## Render A New View On Model Add

When a model is added to the underlying collection, a new view instance should be rendered and added to the DOM. A simple default implementation of this would just append the new view instance to the end of the existing list.

### Listen To The Collection "add" Event

Add a constructor function in the CollectionView definition, and have it apply the base view's constructor to itself. This ensures the inheritance chain is set up properly. Now add a handler for the "add" event of the view's collection, to the constructor. This handler should call a `modelAdded` method.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({

  constructor: function(options){
    BBPlug.BaseView.call(this, options);

    this.listenTo(this.collection, "add", this.modelAdded);
  },

  // ...

});
```

The `modelAdded` method needs to do a few things: get the view type to use, create an instance of it, render the view, and populate the HTML of the CollectionView with the child view's HTML. If this sounds familiar, it should. This is exactly what the `render` method currently does. Only, the `render` method does it for all models in the collection and you only want it to happen for the new model, in this case.

## Extracting `renderModel` And Calling It From `modelAdded`

There's a redundant code problem at this point. Both the `render` method and `modelAdded` are doing the same basic rendering process. To combat that, extract the code that works on the individual models from the render method and put it in to its own method. Then you can call this new method from both `render` and `modelAdded`.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({

  constructor: function(options){
    BBPlug.BaseView.call(this, options);

    this.listenTo(this.collection, "add", this.modelAdded);
  },

  getModelView: function(model){
    return this.modelView;
  },

  // event handler for model added to collection
  modelAdded: function(model){
    var view = this.renderModel(model);
    this.$el.append(view.$el);
  },

  // render a single model
  renderModel: function(model){
    var ViewType = this.getModelView(item);
    var view = new ViewType({model: item});
    view.render();
    return view;
  },

  // render the entire collection
  render: function(){
    var html = [];

    // render a model view for each model
    // and push the results
    this.collection.each(function(model){
      var view = this.renderModel(model);
      html.push(view.$el);
    }, this);

    // populate the collection view
    // with the rendered results
    this.$el.html(html);

    return this;
  }
});
```

Now when you `.add` a model to the collection that this view is holding, it will be rendered and show up in the DOM.

## Remove A View On Model Remove

A model can be added to the collection, and one can also be removed from the collection. When a model is removed from the collection, the view that holds it and renders it should also be removed. It may be tempting to put a "remove" event handler in a model view and have that view close itself. This creates code duplication, though, as every model view type would have to implement this feature. That means every developer implementing a model view for a collection view would have to know about this requirement. This is boilerplate code that shouldn't have to be written more than once, though. The best place to put it, then, is in the CollectionView.

## Holding View References

In order to remove a view for a given model, you will need to hold a reference to view for each model in the collection. Modify the `renderModel` method to handle this, adding the view instance to an object that acts as storage, using the model's `cid` (client-side id) as the key. Be sure to create a new instance of this storage property in the constructor of the CollectionView, so that each instance has it's own.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({

  constructor: function(options){
    BBPlug.BaseView.call(this, options);

    // set up storage for views
    this.children = {};

    // listen to collection events
    this.listenTo(this.collection, "add", this.modelAdded);
  },

  // ...

  // render a single model
  renderModel: function(model){
    var ViewType = this.getModelView(item);
    var view = new ViewType({model: item});

    // store the child view for this model
    this.children[model.cid] = view;

    view.render();
    return view;
  },

  // ...

});
```

Now you can access the view instance for any given model instance, by looking up the view from the `children` container.

## Removing The View On Model Remove

Add another event handler to the constructor function, this time handling the “remove” event. The method that handles the event should look up the view instance and remove that view from the DOM, ensuring it is closed appropriately.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({

  constructor: function(options){
    BBPlug.BaseView.call(this, options);

    // set up storage for views
    this.children = {};

    // listen to collection events
    this.listenTo(this.collection, "add", this.modelAdded);
    this.listenTo(this.collection, "remove", this.modelRemoved);
  },

  // ...

  // handle removing an individual model
  modelRemoved: function(model){
    // guard clause to make sure we have a model
    if (!model){ return; }

    // guard clause to make sure we have a view
    var view = this.children[model.cid];
    if (!view){ return; }

    // remove the view, if the method is there
    if (_.isFunction(view.remove)){
      view.remove();
    }

    // remove it from the children
    this.children[model.cid] = undefined;
  },

  // ...
});
```

The `modelRemoved` method has two guard clauses in it. The first one checks to see if the method received a model. If it didn't, there is no point in trying to do anything so, so exit. The second guard checks to see if a view was found for the model's `cid`. If a view was not found, there is no point in trying to do anything else, so exit. Finally, the view object that is found must have a `remove` method on it. This method is built in to Backbone.View, so every view instance that is used will have this method available. Objects that are not views, though, or views that have been modified and have this method missing, can't be removed properly.

When a model is removed from the collection, the `modelRemoved` method will fire and the view for that model will shut down and be removed from the DOM.

## Close All The Children With The Parent

Closing a single view is good, but the entire collection of views also needs to be closed when the parent view is closed. Fortunately you've already set up most of the logic to do this.

### Adding A `closeChildren` Method

You'll need a method to iterate and close all existing views. This method will loop over all the views stored in the children reference. This method can either close the child view directly, or remove the model from the collection, allowing the "remove" event on the collection to be handled.

As tempting as it may be to re-use the existing logic of removing a model, this would be a very bad idea. Closing the collection view is not the same as removing a model from a collection. If there are other parts of the application that are using that same collection, removing the model from the collection would potentially break those other areas.

Instead, this method should just close the views, ensuring they are torn down properly. Most of the code you need for this is already found in the `modelRemoved` event handler, though.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({

  // ...

  // handle removing an individual model
  modelRemoved: function(model){
    // guard clause to make sure we have a model
    if (!model){ return; }

    // guard clause to make sure we have a view
    var view = this.children[model.cid];
    this.closeChildView(view);
  },

  // a method to close an individual view
  closeChildView: function(view){
    if (!view){ return; }

    // remove the view, if the method is there
    if (_.isFunction(view.remove)){
      view.remove();
    }

    // remove it from the children
    this.children[model.cid] = undefined;
  },

  // close and remove all children
  closeChildren: function(){
    var children = this.children || {};
    _.each(children, function(child){
      this.closeChildView(child);
    }, this);
  }

});
```

Both the `closeChildren` and `modelRemoved` methods take advantage of the new `closeChildView` method, now. This keeps the code consistent and re-usable, making it easy to see what's going on as well.

## Override `remove` To Close The Children

With the ability to close all of the children, now, the parent CollectionView should do this when it is being closed and removed from the DOM.

Override the `remove` method on the CollectionView, and have it call the `closeChildren` method.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({
  // ...

  // override remove and have it
  // close all the children
  remove: function(){
    BBPlug.BaseView.prototype.remove.call(this);
    this.closeChildren();
  }
});
```

Pay attention to the order in which things are removed from the DOM, in this case. The Collection-View itself is removed first, and then the children are removed. This has a net effect of the entire collection of views being removed from the DOM all at once. When the CollectionView is removed, it will take all of it's children with it, from the DOM. But you still need to call `.remove` on all of the children anyways - not the clean up the DOM, but to clean up all of the event handlers and other code that may have been added to the child views' remove method.

The reason for this order is performance. If you were to remove all of the individual views, first, there's a high likelihood of causing too many changes in the DOM, slowing down the perceived performance of the app. By removing the collection view first, all of the views are gone all at once, and then they can be cleaned up out of memory, which is a much faster thing to do.

## Re-Render The Entire List On Reset

The last feature that the CollectionView needs, is the ability to wipe out the entire list of child views, and re-render the whole thing. And there are two scenarios where this needs to happen:

1. When some code call `.render` directly on the CollectionView instance
2. When the collection is `.reset` with a new set of models

The first scenario doesn't need much to make it work. At the beginning of the `render` method, all of the existing children can be closed using the `.closeChildren` method.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({
  // ...

  render: function(){
    this.closeChildren();

    // ...
  },

  // ...
});
```

The `closeChildren` method already handles for a scenario where there are no views in the `children` list. Calling the `render` method for the first time, then, will simply skip right past most of the code to close child views because there aren't any to close.

The second scenario of covering a reset event is equally as simple a change. You only need to listen for the "reset" event on the collection instance, and call `render` when it happens.

```
BBPlug.CollectionView = BBPlug.BaseView.extend({
  // ...

  constructor: function(options){
    // ...

    this.listenTo(this.collection, "reset", this.render);
  },

  // ...
});
```

Since the `render` function already handles closing any existing children and then re-rendering the list entirely, there isn't any additional code needed for the "reset" event. If you do find some scenarios that need additional code for handling the reset event, though, you can add a separate event handler method for it. Just be sure to call `.render()` when the time comes to re-render the new list.

### Didn’t We Just Avoid Remove All View Individually?!

Yes, unfortunately, this code to close all views at the beginning of the render method or in the reset event handler may run in to the very same performance problem that was avoided in the `remove` method, of having too many DOM changes happening. There may be ways around this, but that would be a performance optimization outside of this scope. For more information on performance optimizations, I recommend Nicholas Zakas’ book on High Performance JavaScript[16].

With that, the core of the new CollectionView type is done. You can now handle rendering a Backbone.Collection with a new view instance for every model. It handles adding, removing, re-rendering and resetting the underlying collection.

### A Better Example Use

For a great use of this style of collection, check out the chapter on rendering a filtered collection at the end of this book. It walks you through the core possibilities for handling what looks like two simple tasks, showing you how these two things combined can create a very unexpected journey.

## View-Per-Model vs Iterating Models In A Template

This new version of the CollectionView is more flexible and more capable than the original version. However, this comes at a cost. There is more code that has to be run and more maintenance for that code. There are also times when the view-per-model pattern that is introduced here will fall apart.

Take the example of a `<select>` list in HTML. If you want to generate a `<select>` from a CollectionView now, you will run in to trouble. Sure, you can set the `tagName` of the CollectionView to `select`, but what about the children? Assuming you want each model to represent an `<option>` tag in the select, you will need to provide a `value` on the option tag for this to be useful. You can set the `tagName` of the model view to `option`, but getting the `value` attribute on the view’s `$el` will prove somewhat challenging. It would be easier, in this scenario, to iterate through the list of models in the template for a single view. The iteration of models could easily use an HTML template to build the correct `option` tag with the right `value` attribute.

But neither this new version of the CollectionView, nor the old version of the CollectionView is the right way to do things all the time. Both of these patterns are important and you should use the one that fits the scenario at hand instead of blindly applying one or the other, trying to force it to fit when it won’t.

[16]http://www.amazon.com/gp/product/059680279X/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=059680279X&linkCode=as2&tag=derickbailey-20

### Get The Model For The Clicked Element

For more information about the two variations of rendering a collection, see my blog post on how to get the model for the clicked DOM element[17].

## Lessons Learned

There are often multiple solutions to the same problem, and understanding which solution fits the current situation is critical. Gaining that understanding is no trivial task, though. It often involves experimenting and becoming familiar with the failures of each solution. This means you will need to set out with the goal of not only failing, but being able to understand why the failure occurred, and under which circumstances these specific failures could be avoided.

### Cleaning Up Children

Handling child views seems like a generally simple task at first. But without proper cleanup of the children, a lot of problems can be introduced. Memory leaks and zombie event handlers can run rampant, and cause all kinds of bugs. Keeping a list of child views and having them closed when the parent view is closed provides a simple way of cleaning up the children and preventing these types of problems.

### Order Of Variable Declaration Matters

JavaScript can be an odd language at times, and the idea of "variable hoisting" is certainly one of those times. Declaring the variables and setting the value of those variables in the right order becomes important in JavaScript, because of this.

You don't want to declare the CollectionView and assign the `modelView` attribute before the model view type is declared. Doing this will cause the `modelView` to be undefined because the variable will have been declared, due to hoisting, but not be assigned a value yet. Always define the model view before the collection view. The same can be said for any JavaScript objects where one needs to reference the other.

### Not All Collections Need A View-Per-Model

There are times when a model needs it's own view and there are times when the entire collection should be rendered in to a single view. Having options is important so that you can make the choice for any given situation.

[17]http://lostechies.com/derickbailey/2011/10/11/backbone-js-getting-the-model-for-a-clicked-element/

# Chapter 5: Cache Pre-Compiled Templates

So far the extracting of common implementations in to reusable objects has been focused on the reduction of code duplication. However, there are other reasons to create common objects and components. At times, performance considerations need to be taken in to account. For example, the `BaseView` view type has a prime candidate for this: pre-compiling the templates.

## A Texas 2-Step

Underscore, like most JavaScript template engines, uses a 2-step process to generate the end result:

1. Compile the specified template in to a function
2. Execute the function, with an option set of data, to generate the HTML result

The BaseView that we've defined happens to compress both of these steps in to a single call - `_-.template(this.template, data)` - but the two step process is still happening.

The problem is that most view definitions will not change the template at runtime. That is, once a template has been defined, the same template will be used over and over again no matter how many instances of that view are created and rendered. This is a performance burden on the view instances, re-doing work that has already been done before, to get the same results. In fact, the performance difference between using a pre-compiled template vs compiling everytime can be quite staggering. Pre-compiled templates can be nearly twice as fast, averaging a little over 1.5 times faster, depending on the browser and version used (see this JSPerf test[18] and this blog post[19] on pre-compiled templates for more information).

If this is the case and a view's template doesn't change at runtime, then it doesn't make sense to re-compile the template every time. The solution, then, is to compile the template once and re-use that pre-compiled and cached template every time a new view instance is created.

## Building A Template Cache

View instances should check for and/or build a template cache when they are instantiated. Template caches should also be re-used between view instances, and not re-compiled per instance. For that to happen, the easiest place to store the cached template function will be the the view type's prototype.

[18]http://jsperf.com/dom-select-vs-cache

[19]http://lostechies.com/derickbailey/2012/04/10/javascript-performance-pre-compiling-and-caching-html-templates/

```
BBPlug.BaseView = Backbone.View.extend({

  constructor: function(){
    Backbone.View.prototype.constructor.apply(this, arguments);

    this.buildTemplateCache();
  },

  buildTemplateCache: function(){
    var proto = Object.getPrototypeOf(this);

    if (proto.templateCache) { return; }
    proto.templateCache = _.template(this.template);
  },

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

    // use the pre-compiled, cached template function
    var renderedHtml = this.templateCache(data);
    this.$el.html(renderedHtml);

    if (this.onRender){
      this.onRender();
    }
  }
});
```

There are several items of interest in this implementation.

First, the use of the `constructor` function. JavaScript itself doesn't typically provide a function like this. Rather, this is a function that Backbone provides as the object's constructor. Its called any time an instance of the object type is created, and it is responsible for the over-all initialization of the type, including the call to the object's `initialize` function that is typically seen in a Backbone object. When overriding the `constructor` function of a Backbone object, it's important to call the prototype's `constructor` function so that the `initialize` method and other necessary code will be called.

So why use the `constructor` function when the `initialize` function could be used instead? Isn't this causing an undue burden on the developer, overriding it? The answer to that isn't quite so simple. Yes, the developer writing this base view has to remember to call the prototype's constructor. However, this is a far better situation than overriding the `initialize` function. Overriding that function would require every developer that extends from this base view to remember to call back to the prototype's `initialize` function if they need to implement their own `initialize`.

If we override the `constructor` instead, only the base view that we are defining has to remember to call back to the prototype's `constructor` function. Views that extend from this base view don't have to call back to the prototype at all - for the `constructor` or for the `initialize` method. This means the views extending from `BaseView` have less to remember, reducing the over-all burden of implementation.

Secondly, the use of `Object.getPrototypeOf` in the `buildTemplateCache` function is a different way of getting the prototype. The `constructor` that we are accessing the prototype of `Backbone.View` directly, but the `buildTemplateCache` uses this function instead. In the constructor, we know that the prototype is `Backbone.View` so we can explicitly state that. Now it can be argued that we should just use `Object.getPrototypeOf` in the constructor function. But the use of it is necessary in the `builtTemplateCache` function because we don't directly know the prototype of the object we are dealing with.

When the `buildTemplateCache` function is called, the context is set to the current instance of whatever view type we have defined. That is, any time you use `this` within the view's methods, it will (most likely) be referencing the current view instance. If you set `this.templateCache = â€¦` then it will always set the `templateCache` attribute on the view instance. This would defeat the purpose of a template cache, as each view instance would build it's own cache. The call to `Object.getPrototypeOf`, then, gives us the prototype of the object instance. By using "proto.templateCache", the cache is set on the prototype directly and all instances will be able to retrieve the precompiled template from the cache.

## A getPrototypeOf Shim

There is one potential drawback to using `Object.getPrototype` - not every browser supports it. According to the Mozilla Developer Network[20], the current support for this method includes FireFix 3.5 and up, Chrome 5 and up, Internet Explorer 9 and up, and Safari 5 and up. Opera is not listed with support at all, and IE 8 and below do not support this. To get support for it, then, a shim has to be put in place. Fortunately, John Resig (the creator of jQuery) has one available via his blog[21]:

[20]https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/GetPrototypeOf

[21]http://ejohn.org/blog/objectgetprototypeof/

```
if ( typeof Object.getPrototypeOf !== "function" ) {
  if ( typeof "test".__proto__ === "object" ) {
    Object.getPrototypeOf = function(object){
      return object.__proto__;
    };
  } else {
    Object.getPrototypeOf = function(object){
     // May break if the constructor has been tampered with
      return object.constructor.prototype;
    };
  }
}
```

This code needs to be included somewhere in the over-all project, and sent down to the browser before any call to `Object.getPrototype` can be called. Otherwise Opera and old versions of IE won't be able to use this method call.

A check is made to see if the object's prototype has a cached version of the template or not. If it is found, the function exits immediately. If it is not found, a compiled version of the template is created and stored. And lastly, the `render` function is updated to use the cached template instead of the raw template string.

With this in place, the view definitions that we have for blog posts, comments, or any other view that extends from `BaseView`, `ModelView`, or `CollectionView`, will receive the performance benefit of pre-compiled and cached templates. The big win, though, is that views extending from `BaseView` do not need to know anything about this optimization or handle it explicitly. They only need to specify a `template` in the view definition, as they have already been doing. The performance benefit was added to the base view and therefore immediately available to all other views.

For more information on prototypal inheritance and the `this` keyword in JavaScript, check out my screencasts at WatchMeCode.net[22]. Episode #4[23] covers JavaScript's context (`this`) keyword while Episode #5[24] covers prototypal inheritance.

## Lessons Learned

Reducing boilerplate code is an important and common reason for creating abstractions and extracting common code. However, performance improvements, encapsulation and other reasons exist to create abstractions and re-usable components. Look for opportunities to simplify the overall framework and the use of it, and opportunities for performance improvements in the framework or plugin.

[22]http://watchmecode.net
[23]http://www.watchmecode.net/javascript-context
[24]http://www.watchmecode.net/javascript-objects

## Performance: Do You Speak It?

Pre-compiled templates are a great example of a place that performance concerns create a need for more abstraction. Rather than having each view implementation be responsible for caching the template that it uses, having a base type implement the caching mechanism makes the cache and pre-compilation transparent.

## Cross-Browser Compatibility And Shims

Of course, this isn't just a plugin and add-on concern. JavaScript application development in general needs to be aware of cross-browser compatibility. Fortunately, there are several frameworks, guidelines and other resources for managing these issues. Using a framework like jQuery, for example, creates a higher likelihood that the functionality you need will be available in all of the browsers that you are supporting. Even still, there are times when additional shims and backward-compatibility implementations must be provided. Tools like Modernizr[25] provide a lot of backward compatibility for modern features, but a quick search for specific methods such as Object.getPrototypeOf[26] or Object.create[27] will often surface individual implementations, reducing the need for something as large as Modernizr.

[25]http://modernizr.com/

[26]https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/GetPrototypeOf

[27]http://javascript.crockford.com/prototypal.html

# Chapter 6: Overriding Backbone.View's 'constructor'

When overriding the base `Backbone.View` to create your own view types from the base views, there are some issues that you may run in to with inheritance. The inheritance of constructor functions can be broken in strange ways, and code that you override in the constructor or other base ModelView or CollectionView functions may not be called when you expect it to be. This can be a tremendously frustrating problem, as the code you write looks correct, but it does not fire like you expect.

Fortunately, the fix is simple. Unfortunately, it's not obvious. Worse, though, is that the simple but not obvious nature of the fix often makes it look like the solution is unnecessary.

Understanding the solution, then, is great - but it is not enough to know the solution. If you don't understand the problem and the cause, it's likely that you'll allow the solution to be removed at a later point. Before seeing the solution, then, you need to understand the problem and the cause. It's also a good idea to mark the solution with comments that explain why it's there, so that others won't accidentally remove the solution in the future.

## Setting An Automation ID On All Views

Say you have an application that always needs to set a specific data-attribute on view instances. You may need to supply an attribute like `data-automation-id` with a unique yet repeatable id for test automation. This id should only be available for test automation, though, and not in the production environment.

To facilitate this, you can add a new JavaScript file to a project and include code such as the following in it.

```
// automationid.js
(function(){

  // store a reference to the original constructor function
  var originalConstructor = Backbone.View.prototype.constructor;

  // create a function that can return a unique, repeatable id
  function getAutomationId(view){

    // do something here to get a unique, yet repeatable id
    // based on the view instance that is passed in. send it
    // back as a return value
    var id = â€¦;

    return id;
  }

  // override the View's constructor
  Backbone.View.prototype.constructor = function(options){

    // call the original constructor function with
    // the options provided to this instance
    originalConstructor.call(this, options);

    // set the automation id on the
    var automationId = getAutomationId(this);
    this.$el.data("automation-id", automationId);

    // log a message, saying the automation-id is set
    console.log("Automation ID set: ", automationId, view);
  };

})();
```

### That's A Bit If-y

The automation ID code is wrapped in an Immediately Invoking Function Expression ("IIFE", pronounced "if-e") so that the reference variable and view definition can be hidden from global scope. You may or may not need this wrapper for private scope. You can learn more about IIFEs from Ben Alman's blog post on the subject[28].

In your project's HTML, include this file immediately after the backbone.js file. Putting it here will ensure that your automation id function is attached to the Backbone objects before anything like BBPlug is loaded.

```
<script src="/js/backbone.js"></script>
<script src="/js/automationid.js"></script>
<script src="/js/bbplug.js"></script>
```

Now to test this and see the automation id.

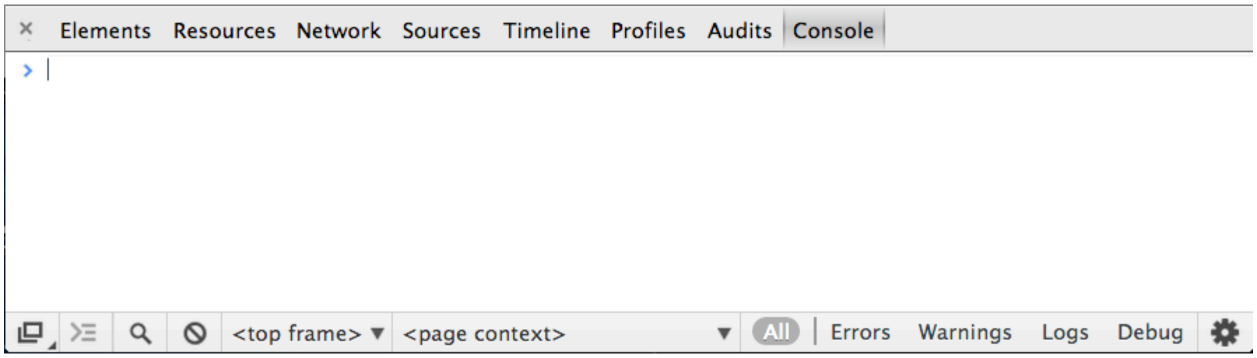
[28]http://benalman.com/news/2010/11/immediately-invoked-function-expression/

## Testing The AutomationID Plugin

To test this, you only need to run an application after the `automationid.js` file has been included.

```
(function(){

  var V = BBPlug.ModelView.extend({
    template: "#model-template"
  });

  $(function(){
    var v = new V();
    v.render();
    $("#main").html(v.$el);
  });

})();
```

Every view instance should log a message stating the automation id with the view instance. Only instead of seeing the message you expect, you will see nothing at all in the console logs.



This is definitely not what you wanted or expected. It should be showing the automation ids for each view, after all, and it isn't.

So why isn't it showing the automation id?

## The Problem Of Taxidermy

The short version of the problem is that replacing the `constructor` method on `Backbone.View.prototype` doesn't work. The `prototype.constructor` attribute is part of the JavaScript prototypal inheritance system, and isn't something you should replace at all. Doing so is essentially trying to replace the type itself, but only replacing a specific reference to the type. Backbone provides the ability to specify a `constructor` in it's type definitions, and it uses that function as the constructor function for the type that is returned. But this is only a convenience if not a coincidental naming of a method.

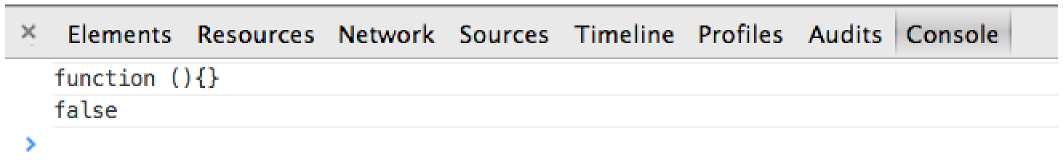
Unfortunately, this creates a larger problem. There are times when you will want to override the constructor of a type, but you shouldn't try to do so the way that the automation id plugin was working.

### Taxidermy On Prototypes

Replacing the `prototype.constructor` is a bit like taxidermy - the end result of "stuffing" a dead bear may still look like a bear on the outside, but it's not really a bear anymore. It just looks like one. In the case of `prototype.constructor` though, this is especially dangerous because you can break code that relies on type checking or prototypal inheritance features that look at the `prototype.constructor`.

Visualize this through code, again:

```
function MyObject(){}

MyObject.prototype.constructor = function(){};

console.log(MyObject.prototype.constructor);
console.log(MyObject.prototype.constructor === MyObject);
```



```
function (){}
false
```

The `prototype.constructor` is no longer pointing to `MyObject`, so the original `MyObject` "constructor function" is not being applied to the new object instance.

## Correctly Overriding The Constructor Function

All of the problems associated with the `prototype.constructor` replacement may be a bit disheartening. But there is hope, and a fairly simple solution.

There are two things you will need, to solve the problem of overriding a Backbone object's constructor function without any other code having to extend from it directly.

1. A new type with the "super-constructor" pattern (where a type calls back to the `prototype.constructor` manually)
2. A complete replacement of the base view who's constructor you want to replace.

By creating a new type that calls back to the original type's constructor, you can ensure the correct chain of inheritance and constructors is handled. Then to get your new type in place without forcing others to extend from your type, you will need to replace the type from which your plugin extends, prior to any other code using it.

## The Super-Method Pattern

The "super-method" pattern is where you call the parent type's method from your own method. Most other languages make it easy to call the "super" function when you override method - to call the original method that you are overriding. JavaScript, unfortunately, doesn't have a "super" or "base" or anything like that. There are ways around this, though, all of which involve getting a reference to the prototype of your type, and calling the method directly.

For example, overriding the `toJSON` method of a model:

```
var MyModel = Backbone.Model.extend({

  // a super-method
  toJSON: function(){
    // call the "super" method -
    // the original, being overriden
    var json = Backbone.Model.prototype.toJSON();
    // manipulate the json here
    return json;
  }
});
```

This will call the `toJSON` method of the type from which the new type was extended. It's a little more code than other languages, but it still gets the job done.

### More Than One Way To Call Prototype

There are a lot of different ways to get to and call the methods of a prototype object. If you'd like to learn more, check out my triple pack of screencasts covering Scope, Context and Objects & Prototypes[29]

## The Super-Constructor Pattern

The constructor function of a Backbone object looks like it would be even easier to replace than a normal method. If you extend from `Backbone.View`, you don't need to access the prototype. You only need to apply `Backbone.View` as a function, to the current object instance.

[29] http://www.watchmecode.net/javascript-fundamentals

Once you have your type set up, you will need to replace the original type with your new type. By doing this, any new type that tries to extend from the original named type, will get yours instead of the original. But you can't just replace `Backbone.View` directly.

```
// define the new type
var MyBaseView = Backbone.View.extend({
  constructor: function(){
    var args = Array.prototype.slice.apply(arguments);
    Backbone.View.apply(this, args);
  }
});

// replace Backbone.View with the new type
Backbone.View = MyBaseView;
```

### Args And Old Browsers

The `args` line is in the super-constructor example to ensure compatibility with older browsers. Some versions of IE, for example, will throw an error if the `arguments` object is null or undefined, and you pass it in to the `apply` method of another function. To work around this, you can slice the `arguments` object in to a proper array. This will return an empty array if the `arguments` is null or undefined, allowing older versions of IE to work properly.

Unfortunately, this setup will fail horribly. When the call to `Backbone.View.apply` is made, it will find your new type's constructor sitting in `Backbone.View`, causing an infinite loop.

To fix this, you need to store a reference to the original `Backbone.View` separately from the new view type. Then you will need to call this reference from your constructor function, and not the `Backbone.View` named function, directly.

```
(function(){
  // store a reference to the original view
  var Original = Backbone.View;

  var MyView = Original.extend({
    // override the constructor, and all the original
    constructor: function(){
      var args = Array.prototype.slice.call(arguments);
      Original.apply(this, args);
    }
  });

  // Replace Backbone.View with the new one
  Backbone.View = MyView;
})();
```

The end result of this is Backbone.View having been replaced with your view type, while still maintaining a reference to the original so that it can be called when needed. Provided the file that includes this code is loaded prior to any other code extending from Backbone.View, all views will receive any functionality defined in `MyView` - which is exactly what you wanted in the Automation ID view.

## Putting The Pieces Together

With the "super-constructor" pattern, the problem of replacing Backbone.View without any other code knowing it has been replaced, can be solved. The Automation ID plugin can be salvaged and it can be used, with relatively minor modifications.

```
// automationid.js
(function(){

  // store a reference to the
  // original constructor function
  var Original = Backbone.View;

  // create a function that can return
  // a unique, repeatable id
  function getAutomationId(view){

    // do something here to get a unique, yet repeatable id
    // based on the view instance that is passed in. send it
    // back as a return value
    var id = â€¦;

    return id;
  }

  // override the View's constructor
  AutomationIDView = Bacbone.View.extend({

    constructor: function(options){
      // call the original constructor function with
      // the options provided to this instance
      Original.call(this, options);

      // set the automation id on the
      var automationId = getAutomationId(this);
      this.$el.data("automation-id", automationId);

      // log a message, saying the automation-id is set
      console.log("Automation ID set: ", automationId, view);
    }
  });

  // replace Backbone.View
  Backbone.View = AutomationIDView;

})();
```

Now you have a plugin that replaces Backbone.View with your AutomationIDView, and no other code in the app knows about it. You only need to ensure this file is loaded before any other file with code that extends from Backbone.View.

## Not Just Backbone.View

This problem and solution applies to any Backbone type, not just View. If you replace the `Backbone.Model.prototype.constructor` or `Backbone.Router.prototype.constructor`, you will run in to the same problem and have the same options for a solution.

```
(function(){

  var origConst = Backbone.Model.prototype.constructor;

  Backbone.Model.prototype.constructor = function(){
    // do custom code here

    var args = Array.prototype.slice.apply(arguments);
    origConst.apply(this, args);

    // do custom code here
  }

})();
```

## Not Every Type You Define

Fortunately, you don't need to apply the super-constructor pattern of calling back to the parent's constructor function, every time you define a new type. You only need to do this when you are defining a base type from which you will be extending.

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

  // apply the super-constructor pattern
  // because this is a base view type from which
  // other views will extend
  constructor: function(){
    var args = Array.prototype.slice.apply(arguments);
    Backbone.View.prototype.constructor.apply(this, args);
  }
});

var CreateUserForm = MyBaseView.extend({
  // no need to apply the super-constructor pattern
  // because this view type is not meant to be extended
  // in to a new view type
});
```

## Lessons Learned

A picture may be worth 1,000 words, but a single line of code to be changed is worth at least 2,0000 if this chapter's length is to be a judgement.

### JavaScript Can Be Complex. Simplify It.

JavaScript's inheritance, prototypes, method invocation patterns, and other aspects all have rather simple rules on the out-set. But the combination of seemingly simple rules can have some very dramatic effects. Understanding each aspect of the rules and how they all combine can be an overwhelming task, at times. Frameworks like Backbone help us to avoid common mistakes and hide a lot of the complexities. But even great frameworks with thousands of users have edge cases.

Don't be surprised when your own applications run in to edge cases and complexities. Do your best to hide the complexity from the developers writing day to day code. Create meaningful abstractions that are easy to modify, so that simple fixes can have a meaningful impact on your application.

## Constructor Functions vs `prototype.constructor` Functions

JavaScript's prototypal inheritance system sets up a default `prototype` for every function. Each prototype has a `constructor` function that points to the original function that defined the type, by default. The `prototype.constructor` should not be touched, in spite of Backbone allowing us to define `constructor` functions. This is akin to taxidermy where something looks like one thing, but is really something else entirely.

## The Super-Constructor Pattern

When creating base types for other types to extend from, you may need to apply the super-constructor pattern. This allows your type to ensure the correct constructor function of the parent type is called.

## Code Injection As A Form Of Plugin

Plugins that are provided as base types can be useful. Specialization of view types, for example, creates an easy way to get functionality in to a system, quickly. But this isn't always the best way to create a plugin. There are times when you need to inject code without any other code knowing.

In the original example for this chapter, there was a need to add an automation id to every view instance. One option for doing this would have been to create an `AutomationIdView` from which every view could extend from. But that would cause the automation id to be in your production environment, as well. Instead, a plugin to replace the `Backbone.View.prototype.constructor` function was created. This allows the automation id to be inserted for a test environment, by including the `automationid.js` file, while leaving the functionality out of the app when in production or other environment that don't need it.

# Part 2: Managing The DOM

With the ability to quickly and easily create new view types, the process of managing where and when the views are displayed now seems quite cumbersome. Having jQuery selectors riddled throughout the code makes it difficult to change the DOM structure without breaking a lot of the app. It also requires a lot of duplication of selectors.

Managing widgets and controls, and figuring out a consistent and simple way to place views in the DOM and remove them from the DOM is as important as the ability to define views. Part 2, then, will show you how to create a few simple objects to manage DOM references, and show and swap out views. Along the way, you will update your views with some new options, as well. You will also combine the ability to define views and manage the DOM, in to a single view type called a Layout.

# Chapter 7: Managing DOM Widgets And Controls

Most Backbone applications either use jQuery or Zepto as their DOM manipulation of choice. I tend to use jQuery as it's supported across more browsers and has more features – though it is a little heavier in terms of download size (and maybe performance). I also use a lot of jQuery plugins for various widgets and controls, to create specific effects, etc. It's generally easy to do, as Backbone's views provide direct access to a jQuery element as `this.$el`. From there, I can call standard jQuery code and plugins.

There are some common patterns I've noticed for doing this, too. Specifically, when to call a particular jQuery function or plugin largely depends on the purpose of that function and sometimes depends on how that function or plugin is built.

## DOM Dependent/Independent

At a very basic level, jQuery manipulations of the DOM can fall in to one of two categories:

1. DOM-dependent
2. DOM-independent

Many of the things I do with jQuery code depend on the HTML that I'm manipulating being in the DOM already. For example, it doesn't make much sense to call ".slideUp" if the HTML we're manipulating isn't currently displayed. The animation that this method causes would still run, but we wouldn't see anything. The end result would not be visible and we would have wasted the browser's execution time on this animation. I generally lump visual changes and animations in to the DOM-dependent category because it doesn't make sense to use these methods if the HTML is not in the DOM yet.

On the other hand, some of the calls I make don't need the HTML to be part of the DOM. In those cases, I can work with document fragments or even raw strings. I often call `.hide()` on HTML fragments before they are attached to the DOM, for example. I do this so that when the HTML fragments are finally added to the DOM, they won't be visible to the user. Then, after I've ttached the HTML to the DOM, I can call `.slideUp()` or any of a number of other animation methods to cause the content to be displayed. I call these methods DOM-independent because they can be called whether we are working with a document fragment that only exists in memory, or working with an element that is already part of the visible DOM.

Of course there are a lot of methods in jQuery that don't deal with HTML elements or the DOM at all. These would clearly fall in to the DOM-independent category, but I'm not going to address those right now. But even if these two categories are a simplified way to view jQuery, it is generally useful. It helps me understand when I should call certain methods vs others, and gives me a better idea of where to integrate jQuery calls with my Backbone applications.

## Simple Manipulations And Events

Most of the simple manipulations that I perform with jQuery are DOM-independent. I can call them and manipulate a document fragment or a DOM element directly, whenever I want to. This includes not only showing / hiding HTML elements, but also adding and removing them, attaching events, and more.

Consider this example:

```
Backbone.View.extend({
  // this is the default for backbone views
  // tagName: "div"

  render: function(){
    var html = $("<ul>");

    html.append("<li>foo</li>");
    html.hide();

    this.$el.html(html);
  }
});
```

With Backbone.View, an HTML element is often created with a view instance. This elements becomes the `this.$el` attribute on the view. In the render method, then, I am using jQuery to create additional document fragments. For performance reasons (since it's possible that the view's `$el` was attached to the DOM without the view knowing it), I'm populating the document fragment with content entirely in memory. This includes the addition of some data and calling `.hide()` on the fragment. I can manipulate the fragment with DOM-independent functions as much as I want at this point.

Once I've completed the rendering of the HTML content, I stuff it all in to the view's `$el`. The `$el` is then attached to the DOM and I can begin using DOM-dependent methods.

I can also attach DOM level events to the fragments that I've generated. Since jQuery is turning our strings in to proper document fragments for me, DOM events are available. That means I can call methods like `.click()` and `.blur()` on the view's `$el`.

```
Backbone.View.extend({
  render: function(){
    // ... build the $el here

    this.$el.click(function(e){
      // handle the click event here
    });

  }
});
```

But I generally consider this to be an anti-pattern. There may be some cases where I need to manually attach events, and it can be done here in the render method. For the most part, though, I should be using Backbone's declarative `events` on my views.

## DOM Events And Simple Animations

Backbone's `View` object abstracts a little bit of jQuery's event system for me through the use of the declarative `events`.

```
Backbone.View.extend({
  events: {
    "click": "showHide"
  },

  showHide: function(){
    this.$("ul").slideToggle();
  },

  // render: function ...
});
```

Surprisingly, DOM events are partially DOM-independent. As I said above, I can add the events to the document fragments before they become part of the DOM. I can even trigger them manually without them being part of the DOM. In general, though, the DOM fires the events for me when the user interacts with the DOM in the manner that I've specified.

Having the user to fire DOM events through DOM interactions (clicks, blurs, changes, etc) is DOM-dependent, of course. The view must attach any HTML structure that it needs to the DOM before the user can interact with them, to fire these events. This is why I said that the events are partially DOM-independent. They can be attached to a document fragment, but they generally need to be in the real DOM for user interactions to occur.

Often when a user clicks or changes a DOM element, I want to respond to this by manipulating the DOM in a visual manner. For example, I might want to call `.hide('slow')` on a portion of the view's `$el` in order to hide some items on the screen with a simple animation. The above example shows this. When I click on the top level `div`, the child `ul` is shown or hidden using the "slideToggle" animation.

## jQuery Plugins And UI Controls

There are a large number of jQuery UI controls available, including the popular jQueryUI[30] suite, commercial products such as Kendo UI[31], and many open source projects that take advantage of jQuery's infrastructure. I've used many different control suites and plugins with Backbone, and I have found that they generally integrate the same way.

Most jQuery based controls are partially DOM-independent. I can call the plugin method to get it started before the document fragment is attached to the DOM. Once the document fragment has been configured with the plugin's code and additional structure, though, the plugin often becomes DOM-dependent. In addition to events, as discussed above, control suites and widgets are often visual in nature and an initialized plugin that is not attached to the DOM won't be visible.

As an example of a control / plugin, I can convert a `ul` list in to a menu structure using Kendo UI. To do this, I can call `.kendoMenu()` during the render method of a view. Or, if I'm using a view layer such as the one developed in the previous chapters, in the `onRender` callback.

Given an HTML template like this:

```
<script type="text/html" id="menu-template">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</script>
```

I can add the `.kendoMenu()` call like this:

[30] http://jqueryui.com
[31] http://kendoui.com

```
var MyView = Backbone.View.extend({
  tagName: "ul",
  template: "#menu-template",

  onRender: function(){
    this.$el.kendoMenu();
  }
});

// ... somewhere else in the app
var view = new MyView();
view.render();
$("#someDiv").html(view.el);
```

Once the Kendo menu is configured, the view's `$el` has to be attached to the DOM (if it isn't already). When that happens, I'll see the menu structure on screen.

## DOM-Dependent UI Controls

There are plenty of jQuery controls and plugins that are entirely DOM-dependent, as well. With these controls, I could not initialize the plugin prior to the view's `$el` being added to the DOM.

The "easy" solution to this is to have the view attach it's "el" directly to the DOM in some fashion. But as I've talked about before[32], this can be a bad idea. Instead, it only takes a few extra lines of code to allow a callback method that be called after the view has been added to the DOM.

```
var AnotherView = Backbone.View.extend({
  onShow: function(){
    this.$el.kendoSplitter({
      // splitter options go here
    });
  }
});
```

In this example, I've added a method to the view called `onShow`. This method contains the call to the Kendo UI Splitter, which is DOM-dependent. It's important to note that the `onShow` method doesn't get called from within the view, directly. This is because it's not the view's responsibility for adding itself to the DOM. This is the responsibility of the code that needs the view. For example:

[32] http://lostechies.com/derickbailey/2011/11/09/backbone-js-object-literals-views-events-jquery-and-el/

```
var view = new AnotherView();
view.render();
$("#someDiv").html(view.$el);

if (_.isFunction(view.onShow)){
  view.onShow();
}
```

The code that needs the view will instantiate it, call render and attach the resulting `view.$el` to the DOM. Next, it checks for the existence of an `onShow` method in the view. If that method does exist, it gets called. Since the code that is using the view knows that it has already added the view to the DOM, it knows when to call the `onShow` method. This allows me to write code that relies on DOM-dependent functionality without the view having to know if it has been added to the DOM or not.

## Extracting The onShow Method Call

I've written this `onShow` method and the code to call it more times than I can count. It has become such a ubiquitous part of my code that I added the check for and call to the function to the `Region` object in MarionetteJS. Adding it here allows me to take advantage of the `onShow` semantics while making it easy for me to know when and where this method will be called.

## Lessons Learned

Working with jQuery plugins can be simple and it can be frustrating at times. There are some tricks that can be used to generally keep it on the simple side, though. A method such as `onShow` and proper timing of this method being called can reduce the amount of complexity in an application by standardizing when and where jQuery plugins are initialized.

### Understand Interactions And Timing

The DOM in an odd beast at times, and working with it at the right or wrong moment can spell the difference between success and failure for an app. It is important to understand when to do certain things in apps, based on the needs of the DOM and interactions with it.

### Context Is King

The patterns and implementations that I've shown here are entirely contextual. There will be scenarios where some DOM-independent code should not be called until after the view's "el" is part of the DOM, for example. Use your judgement, experience and trial-and-error with the various methods and functions that you have to call to get your behavior correct.

The use of a Backbone.View with jQuery code that manipulates the DOM creates a situation where a line of code may work in one spot for one view but need to be moved to another spot for another view. There are no silver bullets in software development. The context of the code being written must be accounted for at all times.

## Use Existing Abstractions When Possible

Backbone has a lot of great abstractions and helpers built in to it. Taking advantage of these will help to reduce the amount of code that is needed without having to write additional abstractions. Take the time to read and understand the Backbone source code, so that the existing abstractions can be used to their full capabilities.

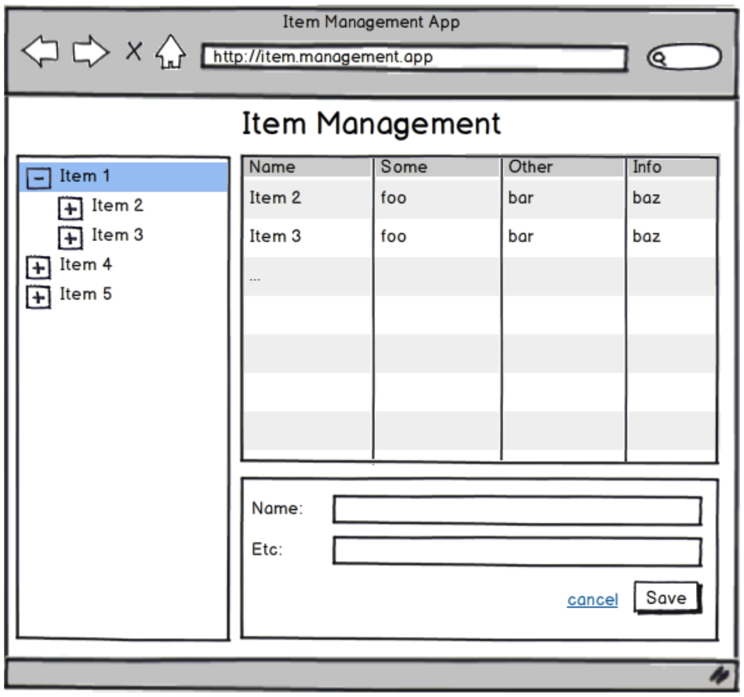
# Chapter 8: Swapping Views In The DOM

Most small Backbone applications can get away with very small, simple methods of managing the DOM. Backbone.View implementations typically contain the majority of what is needed, so why not use them exclusively? As an application grows in size and complexity, though, a more modular application design become necessary. Different areas of functionality within an application should not be directly coupled to other, unrelated areas of functionality. When an application begins to move down this path, it becomes more difficult for Backbone.View to be the sole manager of the DOM content. Having hard coded references to page level DOM elements becomes unmaintainable. Every place in the application that has the DOM element reference is yet another place that has to be changed, if / when the DOM is changed. Fortunately, it is easy to create an object that can manage a DOM element and it's displayed Backbone.View.

## A Need To Change Out A View

I was working on a location management application for a client at one point. It started out with three distinct regions on the screen:

1. Navigation (a tree view)
2. Main content
3. An add/edit form



| Name | Some | Other | Info |
|---|---|---|---|
| Item 2 | foo | bar | baz |
| Item 3 | foo | bar | baz |
| ... | | | |

Once this was in place a new requirement came along: a complex search with search results. To implement this, I needed to modify the application's user interface to swap out the grid and add/edit form out and put in a search results screen instead. The idea being that when the user does a search, the main content area will show the search results. The user can then go back to the location management aspect of the app whenever they need to, as well.

After a bit of searching and experimenting, I found a high level pattern that made this easy. I also realized that I had previously worked with, and implemented the core of this pattern without knowing it.

## Microsoft Prism: Regions

Several years ago, Microsoft released a framework for it's WPF and Silverlight runtimes, called Prism. This was a large scale composite application framework that people used to build well structured and decoupled apps in XAML. I never had a chance to use this framework directly, but I worked with a team of developers that did use it.

One of the things that I liked about what I saw in Prism was the way it used the idea of "regions" to compose the user interface. The gist of it is that you could define a visible area of the screen and build a layout for it without knowing what content was going to be displayed in it at runtime. Then at runtime, the application modules could register themselves to have content displayed in the various regions of the screen.

This pattern fits perfectly with the direction that my Backbone app were heading, so I decided to borrow the names and build my own version in JavaScript.

## A Simple Region For Backbone

In Prism, a region is defined in the XAML markup. In web applications, we have HTML. Similarly, in XAML a region manager is code that you write in C# or other .NET languages. In a web app, though, a region manager is going to be JavaScript. Backbone.js provides a good separation between the markup and the code to run that markup through it's Views, so I initially thought about going down this path for my regions. After a bit of thinking, though, I realized that I didn't necessarily need a Backbone view. What I really need is a JavaScript object that do the following:

- Represent an existing DOM node
- Change out the contents of that DOM node
- Call any required rendering and initialization for content views that will be displayed
- Call any required cleanup for content views when they are removed

What I came up with as an initial pass at handling these needs, was the following (hard coded specifically to use a "#mainregion" element from the DOM):

```
// Define a "Region" object that is hard coded
// to manage the "#mainregion" DOM element
Region = (function (Backbone, $) {
  var el = "#mainregion";
  var currentView;
  var region = {};

  // A method to close the current view
  var closeView = function (view) {
    if (view && view.remove) {
      view.remove();
    }
  };

  // A method to render and show a new view
  var openView = function (view) {
    view.render();
    $(el).html(view.el);
  };

  // Export the API to show a view and
  // close an existing view, if one is
  // already in this DOM element
  region.show = function (view) {
    closeView(currentView);
    currentView = view;
    openView(currentView);
  };

  return region;
})(Backbone, jQuery);
```

Having this `Region` object allowed me to swap out the contents of the `#mainregion` DOM element without any of my views having to know about the DOM element that they were being displayed within. It also prevented the views from having to know about each other. There is now an intermediate object that knows about the view it is currently displaying, and can remove it when a new view needs to be displayed.

## Extracting A Reusable Region Type

From the initial `Region` object, it was easy to extract a re-usable type that can be instantiated and have an `el` assigned to it as the element to manage.

```
// Define a re-usable "Region" object
var Region = (function (Backbone, $) {

  // Define the Region constructor function
  // accept an object parameter with an `el`
  // to define the element to manage
  function R(options){
    this.el = options.el;
    this.currentView = undefined;
  }

  // extend the Region with the correct methods
  _.extend(R.prototype, {

    // A method to close the current view
    closeView: function (view) {
      if (view && view.remove) {
        view.remove();
      }
    },

    // A method to render and show a new view
    openView: function (view) {
      this.ensureEl();
      view.render();
      this.$el.html(view.el);
    },

    // ensure the element is available the
    // first time it is used. cache it after that
    ensureEl: function(){
      if (this.$el){ return; }
      this.$el = $(this.el);
    },

    // show a view and close an existing view,
    // if one is already in this DOM element
    show: function (view) {
      this.closeView(this.currentView);
      this.currentView = view;
      this.openView(view);
    }
  });

  // export the Region type so it can be used
  return R;
})(Backbone, jQuery);
```

The Region can be used by creating instances with the `new` keyword, passing an object literal with an `el` specified:

```
var mainRegion = new Region({
  el: "#mainregion"
});
```

Once I had this in place and started working with it more, I realized that there were a few other things that the Region could do for me.

## Handing onShow Calls

In Chapter 3, I showed how to handle jQuery plugins that rely on the DOM, using an `onShow` method to know that the view is in the DOM. With the Region object in control of placing a View in the DOM, it makes sense for it to also be responsible for calling the `onShow` method.

```
// show a view and close an existing view,
// if one is already in this DOM element
show: function (view) {
  this.closeView(this.currentView);
  this.currentView = view;
  this.openView(view);

  // run the onShow method if it is found
  if (_.isFunction(view.onShow)){
    view.onShow();
  }
}
```

Adding this to the Region was easy and made the region that much more valuable. Now any time I need to work with a DOM-dependent plugin, I can add an `onShow` method and if I'm displaying the view with a Region object, it will be called at the appropriate time.

## Lessons Learned

Building objects that encapsulate common processes can reduce the amount of code needed and provide opportunity to consolidate features in many cases. Looking outside of JavaScript and Backbone can provide insight and opportunity to solve problems in common ways, as well.

### Beg, Borrow and Steal

JavaScript isn't doing anything new with scalable, stateful application development. The vast majority of the patterns and practices that are being used to build Single Page Apps have been around for 20 or 30 years, in fact. It is important to understand the history of stateful application development, not just to avoid repeating the mistakes of the past. Understanding the past and even the current state and future of other languages and platforms allows us to properly beg, borrow and steal patterns and concepts.

In this case, I decided to borrow an idea from Microsoft's Prism framework when it came time to manage swapping view instances in and out of the DOM. Having an existing codebase, documentation, example apps and other resources available made it easy for me to see how this worked and why. I was able to adapt the core concepts from a C#/XAML/Windows world in to the JavaScript world, with great success.

### DRYing Up DOM References

The "Don't Repeat Yourself" principle says that we should reduce duplication in code. This makes life as a developer easier by reducing the number of place that I have to change something. DOM references are no different in this regard - and may even be subject to more frequent change, creating a greater need to reduce the number of references to a given DOM element.

By introducing a Region object to manage the content of a specific DOM element, I was able to reduce the number of references to the "#mainregion" in my application. When (and I truly mean "when" - it happened, multiple times) the DOM element in question changed and I needed to update the selector that my application used, I only had to do it in one location instead throughout every file that needed to display content in that element.

## Consolidating Needs

The `Region` object initially provided a way to manage the contents of a DOM element, but it quickly became apparent that it could do more than that. Since this was the object that knew precisely when a Backbone.View was being inserted in to the DOM, it made sense to have the Region check for and call the `onShow` method of a View instance. Similarly, the need to manage View cleanup became the responsibility of the Region because the Region was already handling showing and removing the views. It was a natural progression for the `close` and/or `remove` method of a Backbone.View to be called from the Region, when it was closing a view.

## Make Assumptions, But Provide Fallbacks

Building a plugin or abstraction that takes advantage of another is usually very beneficial. Assumptions can be made about how certain objects will behave, allowing the combination of existing ideas building new ideas faster. But if the Backbone way of providing a flexible library where pieces can be used independently is to be maintained, fallback mechanisms should be provided when possible.

In the case of a Region, I can facilitate DOM dependent jQuery plugins by having the region call an `onShow` method of a view. Not every view is going to need this method, though, so I check to ensure the method is available before calling it.

Similarly, I can make assumptions about a `close` method being available on my view if I am using Regions and my own View layer. But if I want other developers to be able to take advantage of Regions without forcing them to use my View layer, I need to provide a fallback. Backbone.View has a built-in `remove` method and is a suitable fallback in this case. If a view instance does have a `close` method, I call that. If not, I call the `remove` method instead.

# Chapter 9: Managing Nested Views

Both managing the DOM and reducing boilerplate code through base view types are tremendously important. Each of these will provide a significant boost in your productivity and reduction in the amount of code that you have to write. When you combine your DOM management tools and base view types, though, that's when the real magic begins to happen.

Take the combination of a ModelView and a Region, for example. A region is an object that manages the display of various views within a specified DOM element. A ModelView, on the other hand, allows you to render a template in to some HTML output. If you combine them, you will be able to render a template and have that rendered template display other views.

If this idea of rendering an HTML template and then populating additional HTML in to various parts of that template sounds familiar, you're right! This is basic functionality that you will find in any modern web server. ASP.NET MVC calls this a Master Page. Ruby on Rails and ExpressJS call this a Layout, and other platforms may call it something else, still. The idea is the same, though, and building a Layout for your view framework is going to open a world of possibilities for larger, well-structured applications.

## Nested Views: The Hard Way

It may seem like a good idea to nest views directly inside of each other - especially in very simple use cases.

To start, you only need to add a view's `$el` to the DOM of your current view. This is a simple problem to solve. Using the `onRender` method of the `ModelView`, for example, you can inject the new view after rendering the parent view.

Set up a couple of templates, like this:

```
<script id="child-view" type="text/html">
  <h2>I'm a child view!</h2>
</script>

<script id="parent-view" type="text/html">
  <h1>I'm the parent view!</h1>
  <div class="child-container"></div>
</script>
```

And a ModelView with the onRender method implemented will be able to handle inserting the child view template in to the parent:

```
var ChildView = BBPlug.ModelView.extend({
  template: "#child-template"
});

var ParentView = BBPlug.ModelView.extend({
  template: "#parent-view",

  onRender: function(){
    var child = new ChildView();
    child.render();
    this.$("#child-container").html(child.$el);
  }
});

var parent = new Parent();
parent.render();
$("body").html(parent.$el);
```

This displays the parent view with the child view in the `body` of the HTML document, as expected. Easy enough, right? It certainly looks easy enough, but there are a host of problems that this type of view embedding can cause, including views that are difficult to swap out or change type, and potentially causing unwanted zombies.

If you are bent on solving these problems the hard way, you'll need to do a few things for each of the problems.

## Closing The Child View

The first thing you'll need to do is close the child view. At the very least, you'll need to do this when the parent view is closed. As discussed in the first part of this book, leaving a view unclosed causes a number of problems like zombie view references.

To close the child view, you'll need to hold a reference to the child view and close it when the parent closes.

```
var Parent = BBPlug.ModelView.extend({
  template: "#parent-template",

  onRender: function(){
    // store the child in the parent
    this.child = new Child();
    this.child.render();
    this.$("#child-container").html(this.child.$el);
  },

  onClose: function(){
    // close the child when the parent is closed
    if (this.child){
      this.child.close();
    }
  }
});
```

Now the child view will be closed when the parent closes. Next, you'll need to handle the view type for your child.

## Changing Child View Types

If you want to swap out the child view instance for a new view type, you'll need a way to define which view type to display. You can do this a number of ways, such as `if` statements or `switch` statement, using data from the model or elsewhere.

```
var Parent = BBPlug.ModelView.extend({
  template: "#parent-template",

  onRender: function(){

    // get the child view type
    var ChildViewType;
    switch (this.model.get("foo")) {
      case "bar":
        ChildViewType = BarChild;
        break;
      case "baz":
        ChildViewType = BazChild;
        break;
      default:
        ChildViewType = Child;
        break;
    }

    this.child = new ChildViewType();
    this.child.render();
    this.$("#child-container").html(this.child.$el);
  },

  onClose: function(){
    // close the child when the parent is closed
    if (this.child){
      this.child.close();
    }
  }
});
```

Your parent view can render any of the pre-defined child views, at this point. The logic to get the view type is getting a little jumbled up with the logic to render and display the view, though. And it still hasn't covered the use case of closing the existing child view and re-rendering a new child view type.

### Child Container? Re-rendering? BOILERPLATE!

What would it take to close the current child view and render a new one? What about changing the child view container? Now multiply the lines of code in this solution by the number of children you need to embed in to the parent. This would involve loops, additional logic and multiple storage slots for the child view instances.

Even with good method extraction, the `Parent` view type ends up with a lot of boilerplate code. And what do you do with boilerplate code?

## Defining A Layout, Use-Case-First

Like the other view types at the beginning of this book, the boilerplate for nesting child views within parent views can easily be extracted and encapsulated. The logic and process of managing children can be separated from the logic and process of determining which child view type to use and where in the parent view to display it. Furthermore, most of the logic that you need to store the child view reference, manage swapping views in and out, and closing the child views, is already encapsulated in the `Region` object that you built earlier.

Combining these existing tools with a little more logic can help you to quickly and easily create a new view type to handle view nesting. But before you add regions to handle the nesting, you need to create a clean method of defining which views will go where.

## Requirements For Child View Type And Location

Rather than starting with a Layout view definition, start with the use case of the layout, as if the Layout type already existed. This will let you design how the API and configuration of the Layout looks before implementing it. Having the design in place will make it easier to create the Layout type.

Given the problems that were noted earlier, the configuration of the Layout will need to handle a few things:

1. Easily define DOM locations for child views
2. Allow child view types to be determined programmatically
3. Easily swap view instances in / out of these locations
4. Allow view swapping from within or from outside of the parent view

With the basic requirements list outlines, it's time to think about the configuration of the layout to support these needs. Keeping the configuration consistent with other parts of Backbone and your BBPlug view configuration is important. Using object literals will provide this consistency and provide the needed flexibility in the requirements.

## Named DOM Elements: Regions

There are times when a Layout will be in complete control over all the views that it contains. There are times, though, when a higher level object or workflow handling mechanism will need to be in control over the specific views displayed within the Layout. You already have an object type that can handle these needs: Regions. As long as the Region instance for the specified DOM element is available as part of the Layout's API, you can handle both requirements. To do that, you will want to name a region instance and provide a selector for it.

```
var MyAppLayout = BBPlug.Layout.extend({
  regions: {
    child1: "#child-container",
    child2: "div.something-here",
    child3: ".that-one-there"
  }
});
```

Naming the configuration option `regions` - plural of "region" - implies that multiple regions can be defined. The key in the `regions` configuration items can be used to generate a new attribute on the Layout instance. Each of these new attributes will be an instance of a Region, assigned to the DOM selector.

### Using The Layout's Regions

Using the regions from the Layout will work the same as any other region - you call the `show` method, passing a view instance to it.

```
var layout = new MyAppLayout();
layout.render();

layout.child1.show(new ChildView());
layout.child2.show(new AnotherView());
layout.child3.show(new SubLayout());
```

To ensure the specified DOM element exists for the region, you will need to render the Layout instance first, and then show the children.

### Requirements Met, Unlimited Nesting Enabled

All of the requirements for a Layout are met with this API.

1. You can easily define new regions within a layout by adding a new line to the `regions` configuration item
2. Regions take any valid Backbone.View instance, letting you run any code you need to determine which view instance to display in a region
3. Since regions are used, there is only one method that you need to call to swap a new view in and an old view out
4. The defined regions become part of the public API for the Layout instance, allowing 3rd parties to get involved in determining which view to display

Also note the implication of the third example, above (in the `child3` region). A layout instance is being nested within the parent layout. You can display any valid Backbone.View type in a Region. This means that another Layout can be shown within the region of a parent layout. The result is an infinite amount of nesting for views - a very powerful idea that allows applications to be broken down in to very fine-grain pieces.

## Build The Layout Type

With the requirements met from an API perspective, it's time to create an implementation that will work the way you expect.

Start with a basic view definition, of course.

```
BBPlug.Layout = BBPlug.ModelView.extend({

});
```

Beyond the built in rendering, the `regions` configuration needs to be processed. You might be tempted to use the `onRender` function for this, but that would cause problems when developers want to extend from your Layout. This would force use of the `onRender` method to include a call back to the prototype's onRender. Instead, you'll want to override the `render` function and make the call back to the original render function, within the Layout code. This will make it easier for others to extend from the Layout, later.

Your new render function will call the original to make sure everything is rendered, and then it will call a method to parse the region definitions.

```
BBPlug.Layout = BBPlug.ModelView.extend({
  render: function(){
    // call the original
    var result = BBPlug.ModelView.prototype.render.call(this);

    // call to process the regions
    this.configureRegions();

    return result;
  }
});
```

By taking the hit yourself, and overriding the `render` function, you are freeing up the `onRender` method for those that extend from your Layout.

Now you'll need to add the `configureRegions` method. This one will process the `regions` configuration and create

## Processing The Regions Configuration

The `configureRegions` method will need to loop through all of the named regions. Each of these will turn in to an instance of a Region, as an attribute on the Layout instance.

```
configureRegions: function(){
  // get the definitions
  var regionDefinitions = this.regions || {};
  // loop through them
  _.each(regionDefinitions, function(selector, name){
    this[name] = new Region({el: selector});
  }, this);
}
```

This code is compact, but it does a lot for you.

The first line grabs either the `regions` definition or an empty object literal. If there are no regions defined, the `_.each` method would throw an error. Adding the `|| {}` ensures there is always an object literal to use, preventing the error.

Once inside the loop over each key/value pair, a new Region instance is created. The region instance has it's `el` assigned to the `selector` variable, which is the value half of the key/value pair.

The resulting Region instance is assigned to an attribute on the layout instance, using the `name` (key from the key/value pair) as the attribute.

But there's a problem, here. The selector that the region is handed will run against the entire DOM. It isn't scoped to only the Layout instance.

## Scoping The Region Selector

It might not seem like a bad idea to let the Region's selector be scoped to the full DOM, off-hand. If you're only using CSS IDs for selection, this won't be a problem at all. But when you get in to larger applications with deep and complex structures, there's a good chance that you won't be using IDs for everything. In fact, there's a good chance that you'll repeat HTML and CSS class selectors across views. If that happens, you could easily select DOM elements that are outside of the Layout's `$el`. This is a very bad idea. Your Layout (and Backbone.View instances, in general) should be restricted to the `$el` and only the `$el`. Allowing it to select anything outside of this will cause problems, long run.

The selector for the Regions, then, needs to be scoped to the Layout's `$el`. To do this, look back at how the Region's `el` option works. The `el` option that you pass in is stored for later use. When the Region first shows a view, it calls the `ensureEl` method which turns the `el` in to a proper jQuery selected object.

Given this, there are two options for solving the problem:

1. Pass an already jQuery selected object as the `el`
2. Override the `ensureEl` function on a custom region type

The second option seems fair, off-hand. But if you decide at a later time that you want to allow custom Region types to be used, this gets to be a little more dubious. Instead of having a single Region type that has the over-ridden `ensureEl` method, you now have to override it on every Region type used. This can cause problems with custom Region types that have special implementations of `ensureEl`.

So, you are left with the first option, though it is not without fault, either. Normally, it is not a good idea to take advantage of internal behavior and code from other objects. This type of coupling creates horrible headaches and problems for future developers that don't know the same things as you. Since the Layout object is part of the same framework as the Region, though, an exception to this can be made.

Supply a jQuery selected `el` to the Region instance, using the Layout's `this.$(selector)` method. This method scopes the `selector` to the view's `$el`. jQuery is also smart enough to not re-select an object that is already a jQuery selected object.

```
configureRegions: function(){
  // get the definitions
  var regionDefinitions = this.regions || {};

  // loop through them
  _.each(regionDefinitions, function(selector, name){

    // pre-select the DOM element
    var el = this.$(selector);

    // create the region, assign to the layout
    this[name] = new Region({el: el});
  }, this);
}
```

The end result, then, is that you have pre-selected the DOM element for the Region.

## Closing The Regions With The View

Being able to display a nested view is great. But there's more to it than just showing the view. If you want to avoid additional zombie views and other problems, you will need to close the nested views as some point. The Regions that you're using to show the view can handle this, but they need to be told to close the child view.

Like the `render` method, where you added behavior to set up the child regions, you need to override the `close` method and add behavior to close the regions (and sub-sequently, child views).

```
BBPlug.Layout = BBPlug.ModelView.extend({
  // ... existing methods here

  close: function(){
    // close the Layout
    BBPlug.Layout.prototype.close.call(this);

    // close the regions
    _.each(this.regions, function(selector, name){
      // grab the region by name, and close it
      var region = this[name];
      if (region && _.isFunction(region.close)){
        region.close();
      }
    }, this);

  }
});
```

The new close method will close all of the regions by looping through the `regions` configuration, grabbing a reference to the region by name and calling the `close` method on it. This works because the code to set up the regions uses the name to add attributes of the specified name. The only additional logic is to check and make sure the name returns an object with a close method. Without this check, closing a Layout that had not been rendered would result in errors since the regions would not exist on the Layout instance.

The Layout itself is closed before the child regions in order to remove the entire Layout from the DOM before closing the actual view instances. If you closed the Layout after the regions, you run a change of poor performance and odd visual problems. This would cause each region to remove it's child view from the DOM, one at a time. Each removal from the DOM could trigger a reflow and repaint of the DOM, resulting in artifacts and oddities in the DOM itself. By closing the Layout first, this is avoided because the layout will remove all of the child DOM elements at the same time.

### Repaint, Reflow And Performance

For more information on Repaint and Reflow with the DOM, and performance on the web in general, check out Nicholas Zakas' High Performance JavaScript[33] book.

[33]http://www.amazon.com/gp/product/059680279X/ref=as_li_ss_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=059680279X&linkCode=as2&tag=avocadosoftwa-20

## Re-Rendering The Layout With Regions

The final challenge in a Layout, at this point, is being able to re-render the Layout itself. Re-rendering the Layout and showing new view instances generally works properly. But it can lead to problems of code in the child views not being cleaned up properly. Fortunately, this is a simple problem to solve. You already have the code to close the regions. It just exists inside of the `close` method at the moment.

Extract the code to close the regions in to a `closeRegions` method, and have that called from the `close` method to make sure it still cleans things up.

```
BBPlug.Layout = BBPlug.ModelView.extend({
  // ... existing methods here

  close: function(){
    // close the Layout
    BBPlug.Layout.prototype.close.call(this);
    // close the regions
    this.closeRegions();
  },

  closeRegions: function(){
    _.each(this.regions, function(selector, name){
      // grab the region by name, and close it
      var region = this[name];
      if (region && _.isFunction(region.close)){
        region.close();
      }
    }, this);
  }
});
```

Now you can modify the `render` method to close the regions before doing the real rendering.

```
BBPlug.Layout = BBPlug.ModelView.extend({
   render: function(){
    // close the old regions, if any exist
    this.closeRegions();

    // call the original
    var result = BBPlug.ModelView.prototype.render.call(this);

    // call to process the regions
    this.configureRegions();

    return result;
  },

  // ... existing methods here
});
```

If this is the first time that the view has been rendered, the `closeRegions` method will just loop through the names and not do anything. On subsequent renders, though, the regions will be closed appropriately. This method is already smart enough to handle both situations for you.

## Lessons Learned

Nesting views in Backbone is important, when applications begin to grow. Having the ability replace large swaths of DOM content with feature-specific layouts, and nesting child views in to those layouts, will open a new world of possibilities in application design and modularity. But nesting views doesn't come without it's own challenges.

### Design The API First

Looking at the implementation details first, and slogging through the code to make it work, is a good way to shoot yourself in the foot. When building the Layout, the API for configuring the regions was designed first. This allowed you to see the expected behavior of the Layout, from the API perspective, first. Once you had the API set the way you wanted, implementing the Layout was easier. You didn't have to guess at the configuration or try to figure it out on the fly. You already knew that the configuration would work, and only had to implement something that could read the configuration in question.

### Iterate And Increment

Not every API design is easy, and it is rare to get it right the first time. Even when you design the API first, you will run in to situations where the API doesn't quite match the possibilities for the implementation details. Don't be afraid of this - welcome it. Use it as a learning experience to understand API design. Iterate over the API design, with the implementation details. When one idea works in the API but not implementation, change the API. When a design works in implementation but creates a terrible API, change the API. Work through small, concise and incremental changes until you find a balance between the API and implementation.

## Nesting Views, Infinitely

The implementation of a Layout demonstrates the first bits of view composition in an application framework. Now you can nest views inside of regions that are already nested in a view. Any valid Backbone.View can be used, meaning you can nest the view structure as deep as you need. Given the nature of how views are attached to the regions, you can easily orchestrate an application to compose the user interface on the fly.

## Object Composition, View Composition

The implementation of the Layout demonstrates a key concept to building large scale applications of any kind: object composition. The user interface of an application can be composed from multiple views at runtime, and the Layout object itself is composed of multiple objects. You've combined the ModelView and Region objects with a little bit of glue to hold them together. The end result is a new type of object that opens a world of opportunity in the application's user interface.

## Method Composition

Methods are as easily composed as objects - sometimes easier. In the case of the Layout, you had code that was closing all of the child regions. Then, when the need to close the regions at a different point in the Layout's lifecycle became apparent, this code was extracted in to it's own method. The result was another method that could be used to compose larger behavior sets within the Layout object.

## Self-Documenting Code

In addition to the benefit of behavioral composition within the Layout, extracting the method to close the child regions created a better sense of self-documenting code. That is, code that is expressive of the intent of the code, not the implementation of the code.

Human beings think in hierarchy and abstraction, even if they don't realize it. Having a function named for the intent and behavior of what it does helps people create simple hierarchies that are meaningful. When you look at the `.close` method on the Layout now, you can clearly see that is closes the child regions because of the `this.closeRegions()` call. You can see the same call happening in the `render` method. Both of the close and render methods are easier to understand because of this one line of code replacing 3 or 4 lines of implementation.

The name `closeRegions` describes the behavior of the method without boring you or confusing you with the implementation details. If you need to know how it works, at some point, you can go look at that method without worrying about the rest of the implementation details from the `close` or `render` method.

## Rules Are Meant To Be Broken

I hold the rule of not digging in to an object's internals very dearly. The "tell, don't ask" principle, and the Law of Demeter both give us ample reason not to do this. It creates tightly coupled code and causes problems down the road.

But sometimes it's ok to break the rules.

When you implemented the Layout, and you had to scope the DOM selectors for regions to the Layout's `$el`, you had two choices. Both of the choices required knowledge of how the `Region` object worked. One of the choices would replace a function on the region. The other choice relied on knowing that Regions use jQuery to select their `$el` from the DOM. While neither of these choices were great, having to choose one of them is ok in this case, for two reasons:

1. It let you implement the feature and behavior that you needed, but more importantly,
2. It happened within the internals of two objects within the same over-all framework

Since the Layout, Region and ModelView are all in the same framework, having to dig in to the internals of an object can be overlooked. Just be careful not to do it too many times. And be on the look out for scenarios were this tight coupling does break down and cause problems. You may need to find another solution, in the end.

# Part 3: Data And Meta-data

While views are often the first place that developers see boilerplate code, they are certainly not the last. Nearly every type that Backbone provides will have some form of boilerplate applied to it.

For example, many applications have a model or list of models that can be "selected" from a user interface. The app may need need to show a list of items - often in a table or grid layout - with a checkbox next to each item and have the app user select as many or as few of the items as needed, by checking the boxes. With at least one items selected, they can then perform some action on the selected items. This may be as simple as a bulk-delete, or may be more complex and specific to the application being created.

Part 2 will cover the creation of a plugin that allows models to be select, or "picked". Collections will also be able to track the models that have been picked, as well. Finally, a method of mixing the functionality in to models and collections that doesn't cause issues with method or attribute name collisions will be shown.

### Theme And Variation

Naming things can be difficult and cause problems far beyond the actual name. Names often set a theme or set of expectations around the language used with the thing being named. In the first version of Part 2 of this book, a theme of "select" was chosen. This proved by be a bad idea for reasons described in Appendix E: Theme And Variation. In the end, the theme of "pick" was chosen to correct this set of problems.

# Chapter 10: Pickable Models

A model should know whether or not it has been picked. The model should also provide methods for picking itself, and unpicking. Having this knowledge and these methods directly on the model reduces the amount of code in the views that are displaying the models. It also properly encapsulates the logic of picking / unpicking on to the objects that are being modified. Rather than keeping all of the data and logic surrounding the selection in the view instance, the view only needs to react to changes in the model's state. The view can also call the appropriate method on the model based on user interaction with the checkbox on the view.

## Options For Storing Picked State

There are a few options for storing the state of selection on a model: as part of the model's attributes, as data attached directly to the model (outside of the `attributes`), or in a third party object with no data stores on the model directly.

The first option involves using the model's `get` and `set` methods to store and retrieve the state of selection. The advantage of this is the existing storage mechanism for the selection state. Changing this state will also trigger "change" events that can be used by views or any other objects. The result is a reduction in the amount of code that needs to be written because the model is using existing functionality to facilitate the "picked" state. The downside to this is the data being stored with the model's attributes. This might not be a problem if this state can be ignored by anything reading the data. However, the data is still there in the attributes. It will be sent to a back-end server when the model is saved, and it will be part of the model's attributes for as long as the model is in memory.

The second option involves a little more code: store the state directly on the model and not in the attributes. The benefit is that the events can be customized to a higher degree, and the extra data is not found in the model's attributes or `.toJSON()` call. The selection state is not sent to the server because its not part of the attributes. The model is not restricted to using "change" events, either. It can trigger "picked" and "unpicked" events, or any other event name desired. The downside is that there is more code to write. Methods to manage picking the models, triggering events, and storing the state will have to be added to the model.

The third option is a variation of the second, but requires more code, still. Creating a separate object to house the state of model selection can provide some benefit, though. It keeps the model clean of the selection state. It also allows any model to be selected or "picked", whether or not it was originally designed for this purpose. The major downside is two-fold: the state is not stored in the model, and the methods to manipulate the state are not found on the model.

The choice between less code + attribute pollution, or more code without attribute pollution is not always clear. For this example, the idea of polluting the attributes seems to be the worse of the two, though. Having extra data hanging around the model's attributes, is generally a bad idea. It pollutes the models with information specific to a single user experience or UI feature. The allure of being able to select any model - whether or not it was set up for this purpose - is great, as well. The creation of a plugin to manage the selection state is an option to reduce the overall boilerplate code for managing model selection. The problem of allowing a model to manage that state is solvable as well, and will be examined in chapter 10.

## Backbone.Picky Namespace

To add the selection ability to a model or collection - or, in other words, to allow a user to pick which items they want - a new plugin will be created, and called "Backbone.Picky". Naming the plugin with "Backbone." is quite common. This convention gives any others developers using the plugin a clear understanding that this plugin is meant to be used with Backbone.

To get started, a namespace will be created for the new plugin.

```
Backbone.Picky = {};
```

This will serve as the namespace that all of the selectable model and collection objects and functionality will hang from.

The assumption is `Backbone` being an available object is fairly safe at this point. It does add a requirement of Backbone being loaded prior to this plugin, but that again is a safe assumption.

## Picky.PickableModel

Rather than extending from Backbone.Model directly, a separate object will be added to the Picky namespace. This object will contain all of the functionality and data for the selectable models.

```
Backbone.Picky.PickableModel = function(model){
  this.model = model;
};
```

This constructor function will create instances of the `PickableModel` object, with each instance managing the the model instance that is passed to it.

```
var myModel = new MyModel();
var pickable = new Backbone.Picky.PickableModel(myModel);
```

The PickableModel needs methods to pick and unpick the model, as well. As a simple first implementation, these methods only need to set a `picked` attribute on the model.

```
_.extend(Backbone.Picky.PickableModel.prototype, {

  pick: function(){
    this.model.picked = true;
  },

  unpick: function(){
    this.model.picked = false;
  }

});
```

### _.extend

The use of `_.extend` is a shortcut to add multiple methods and/or attributes to an existing object. It takes the first parameter as the target object and copies all of the methods and attributes from the remaining parameters to it.

When dealing with prototypes, this will make the method available to every instance of the type.

But these method implementations are very naive. There is more behavior that needs to be accounted for when picking or unpicking a model.

Picking a model not only needs to set the correct state, but also trigger an event from the model that says the model has been picked. To prevent infinite loops from multiple events from being triggered, this method should also check to see if the model is currently picked. If it is, don't set the state or trigger the event again.

```
_.extend(Backbone.Picky.PickableModel.prototype, {

  pick: function(){
    if (this.model.picked){ return; }

    this.model.picked = true;
    this.model.trigger("picked", this.model);
  },

  // ...
});
```

Unpicking a model should run the same basic logic, but in reverse. It should trigger an "unpicked" event, but only if the model is currently picked.

```
_.extend(Backbone.Picky.PickableModel.prototype, {
  // ...

  unpick: function(){
    if (!this.model.picked){ return; }

    this.model.picked = false;
    this.model.trigger("unpicked", this.model);
  }
});
```

In both functions, JavaScript's "truthiness" - or coercive nature of evaluation - is being taken advantage of in the guard clauses. If a model has just been created, calling `pick` will check for a `picked` attribute which won't exist. The attribute evaluation will return `undefined`, which gets coerced in to a `false` value for the if statement. Similarly, the guard clause in the `picked` method will negate the coerced value if `unpick` is called on a model that does not have a `picked` attribute. Once the model has been picked, though, the attribute will exist and the guard clauses will no longer coerce the value.

Also note that the `picked` attribute is not being stored directly on the model. Rather, the instance of the SelectableModel is keeping track of this, which will provide more flexibility and options in how the SelectableModel instance is used.

With the basic implementation of a `SelectableModel` in place, it can be used in simple scenarios that do not require any interaction with other selected models or collections. (A more complex scenario covering collections and multiple selections will be covered shortly.)

## Using A SelectableModel

The `SelectableModel` implementation cannot be used as a model directly. It does not extend from `Backbone.Model` and it does not provide the same API as a Model. Instead, it needs to be created as an object instance of its own.

```
var model = new Backbone.Model();
var pickable = new Backbone.Picky.PickableModel(model);
```

The `pickable` object can be managed on its own, with no requirement for the model to know anything about whether or not it is pickable. Passing the `selectable` object to a view allows the view to pick and unpick the model. The events that are triggered would have to be bound from the `selectable.model` or a reference to the original `model`, though. This would look a little odd if the selectable is passed in as the `model` for a view. It would be better to pass it in as a separate option. But it would be best to have the view encapsulate the knowledge of needing the model to be pickable entirely.

```
// Build a PickableView that wraps a model
// in a PickableModel instance
var PickableView = BBPlug.ModelView.extend({

  events: {
    "change .pick": "togglePick",
  },

  initialize: function(options){
    // wrap the view's model in a pickable
    this.pickable = new Backbone.Picky.PickableModel(this.model);
  },

  togglePick: function(e){
    // toggle picking the model, based on
    // the current state of the pickable
    if (this.pickable.picked){
      this.pickable.unpick();
                } else {
      this.pickable.pick();
    }
  }
});
```

With the `PickableView` definition in place, a new instance can be created, passing in the model without having to do anything particularly special to it. The view itself has encapsulated the code that is needed to make the model pickable. The DOM event configuration looks for a '.pick' element - a checkbox, most likely. Changing this checkbox (checking or un-checking it) will toggle the PickableModel's state.

The end result is that any model of any type can be used, and that model will be pickable from the view.

```
// define a new model, but don't do anything to it
var MyModel = Backbone.Model.extend({});

// create a model and view instance
var myModel = new MyModel();
var view = new SelectableView({
  model: myModel
});
```

The model that is passed in to this view is now pickable. This is not because the model had any changes applied to it, but because the view wrapped it in a `PickableModel` instance.

### The “toggle” Boilerplate

You may have noticed that the PickableView provides a method to toggle the model being picked or not. This is exactly the kind of boilerplate code that should be avoided in projects, as it will be copy & pasted from view to view, project to project. I’ve left a `.toggle()` function out of the `PickableModel` implementation, though, to keep this chapter short. But it would be a good idea for you, dear reader, to add this method to your implementation.

## Lessons Learned

There are several things that the `Backbone.Picky` plugin did differently than the previous `BBPlug` add-on. Most notably is the namespace, but also the ability to create a selectable object on its own instead of having to extend from a base model or collection. Several of these lessons may apply to models and collection specifically, but many of them can be more broadly applicable to creating plugins and writing well structure Backbone applications.

### Build Add-Ons As Separate Objects

It’s often tempting to create specialized versions of objects that can be extended from directly. Backbone provides a `.extend` method on every one of it’s types, after all. In the first part of this book, specialized view types were extended from Backbone.View, as well. But this can be problematic. With models, it is generally a bad idea to extend from Backbone.Model to create a specialized type. There are far too many possibilities for extensions and add-ons, and forcing a developer to extend from your model type prevents them from using any other model type as the base.

### Using The Backbone Namespace

Using a custom namespace for an object helps to keep the global object space of the JavaScript runtime clean. However, having a handful of plugins that each create their own global object can still pollute more than is desired. Its common for Backbone plugins to use the `Backbone` namespace to help reduce this pollution and also to give the developer using the plugin more of a direct sense that this plugin is meant to be used exclusively with Backbone.

When using the `Backbone` namespace for a plugin, though, be sure to create a child namespace that hangs off of it instead of placing the plugin’s objects directly on the `Backbone` namespace. `Backbone.Picky` is a good example of this, as it creates a child namespace called `Picky`, which contains all of the objects for the plugin.

## The Principle Of Least Surprise

The principle of least surprise says that the code being used should look and work in a manner that is least surprising. That is, a method's name should be consistent with the behavior of the method. In the case of `Backbone.Picky`, the use of the `SelectableModel` can either enforce or violate this principle. In the case of using the selectable object directly in a view, as either the `model` or a separate object passed in as the options, the use of the functionality provided may be somewhat odd or "surprising". By mixing the functionality directly in to a model instance, though, access to the selectable functionality is more consistent with the expectations of developers writing views.

## Encapsulate Features And Wrappers

It is generally reasonable to ask another developer to send a specific type of object to a method parameter. This is how software development and method calls work, after all. Passing the wrong type to the wrong parameter usually causes bad things to happen. But there are situations where this becomes an undue burden to developers - when a feature's method needs a type that is specific to the feature, asking a developer to wrap their object in this type can be problematic. Encapsulating features and feature-specific wrappers in to method calls allows for flexibility and eases the burden on developers using the feature.

# Chapter 11: Pickable Collections

Adding support for picking multiple models is needed for “select all” and “deselect all” Scenarios. Knowing which models have been picked allows action to be taken on the them, or prevented if some criteria is not met for an action. A user interface can enable and disable various actions based on zero, one or multiple items having been picked.

## Multi-Pick Collections

To get rolling, add a `MultiPickCollection` object to Backbone.Picky. This one is be responsible for tracking models that have been picked and handling pick all / unpick all functionality.

The basic object definition looks similar to that of the `PickableModel` - a constructor function and an extension of the prototype to provide the methods needed.

```
// Constructor function to create MultiPickCollections
Backbone.Picky.MultiPickCollection = function(collection){
  this.collection = collection;
  this.picked = {};
};
```

In addition to the storage of the collection from which models will be selected, the instance has a `picked` attribute to hold the list of picked models. This is an object literal attribute, used like a hash - a key/value pair - to make addition and removal of items easier.

The methods on the MultiPickCollection can be assigned to the prototype. Methods don’t need to be assigned to each instance because they work with the state of the instance they are called from. With prototypal inheritance and the way that JavaScript manages the context in which methods are called (the dreaded `this` operator), calling a prototype method on an instance will in the method working with the instance’ data.

```
_.extend(Backbone.Picky.MultiPickCollection.prototype, {

  pickAll: function(){},

  unpickAll: function(){},

  togglePickAll: function(){}

});
```

Each of these method names and parameter lists should provide insight in to what the method does with the collection. Once again, these method names are following the theme of "picking" models with methods for picking all, none or toggling all models in the collection.

So, why not add `picked` to the list of methods and attributes that are extended on to the prototype? Why add it in the constructor, instead of on the prototype?

Adding `picked` to the prototype would cause unexpected and unwanted behavior due to the way `_.extend` works. When an object is extended, the methods and attributes from the extension are copied on to the target. That is, the name of the attribute is assigned to the value of the same attribute, but the assignment is done on the target.

Since the value of each attribute is assigned to the target, assigning the `picked` attribute to the prototype would result in every instance sharing the same list. In this case, the list shouldn't be shared. Each instance should have its own list of models. Assigning the attribute in the constructor function instead of on the prototype means every instance gets its own list.

## Listening For Model Picks

Picking models can happen one of two ways - through the PickableModel, or through the MultiPickCollection. In either case, the MultiPickCollection needs to be notified so it can keep track of the models that have been picked. The easiest way to handle this is to listen to the "picked" event on the Collection instance. This works because every time a model triggers an event the collection it belongs to also triggers the same event. The collection's version of the event also includes the model, which means the MultiPickCollection will be able to track the picked models, easily.

In the constructor function for the MultiPickCollection, then, you will need to add an event handler for the collection's "picked" event. The MultiPickCollection doesn't know how to listen to events, though. To fix this, use `_.extend` to add Backbone.Events in to the MultiPickCollection, first.

```
_.extend(Backbone.Picky.MultiPickCollection.prototype, Backbone.Events, {
  // ... existing methods here

});
```

Backbone.Events is a simple object literal in the Backbone code base. It is meant to be used as a mix-in in this manner. By extending the prototype of MultiPickCollection, all instances will be able to the event handler and event triggering methods, as needed.

With that in place, set up the collection's "picked" event handler to store a reference to the picked model.

```
Backbone.Picky.MultiPickCollection = function(collection){
  this.collection = collection;
  this.picked = {};

  // When a model is picked, it triggers a "picked" event.
  // The owner collection forwards model events, so listen
  // to the collection's events and capture all picked models
  this.listenTo(this.collection, "picked", function(model){
    this.picked[model.cid] = model;
  });
};
```

The models are stored in the `picked` list by `cid` - the "client id" of the model. This id is guaranteed to be unique amongst all model instances. Unfortunately, though, it will be different between two model instances that have the same data - including the same "id" field. A more robust storage mechanism may want to account for this, but this simple "hash" usage of an object literal is good enough for now.

### An Object Is Not A Hash

There are some limitations to using an object as a hash. Guillermo Rausch has written about why an object is not a hash[a] and makes some compelling arguments. You should take the time to read this article and understand the limitations that you may run in to, and decide whether or not this technique is right for your application.

[a] http://www.devthought.com/2012/01/18/an-object-is-not-a-hash/

Similarly, when a model is "unpicked", it should be removed from the `picked` list. Add this event handler directly underneath of the "picked" handler.

```
  this.listenTo(this.collection, "unpicked", function(model){
    delete this.picked[model.cid];
  });
```

This will remove the model from the `picked` list by deleting the reference. Note that this only deletes the reference, and does not do any kind of "delete" API call or anything like that.

### The Dangers Of "delete"

You should be aware that using the "delete" keyword tends to put browsers in to "slow mode" for objects. That is, the browser will treat the object differently than normal, as it has to re-examine the ever-changing landscape of the object that is having attributes added and deleted.

In all honesty, though, I wouldn't worry about this too much. I have never seen this have a noticeable affect on performance, personally. You should, however, measure the performance difference between deleting an attribute and just setting it to `null` or `undefined`, in your own applications.

## Cleaning Up Event Listeners

In the first part of this book, you wrote View code that used the `listenTo` method to listen to events. The event handlers in the MultiPickCollection constructor also use this method of listening. Unlike views, though, the MultiPickCollection does not automatically clean up the event handlers for you.

Backbone.View has a `remove` method that calls `stopListening` and unbinds the events. To get the same memory management and event cleanup in the MultiPickCollection, then, you'll want to add a similar method. And once again, naming becomes important.

It's common among JavaScript frameworks and objects to use `destroy` or `close` for these end-of-life method names. The choice often comes down to personal preference and past experience. Developers coming from a .NET and C# background, for example, tend to choose `close`. Other developers - especially those grounded in JavaScript patterns and idioms - tend to choose `destroy`. The choice is easier than looking at your own background or preferences, though. Use the semantics and common patterns to pick.

A method name of "destroy" already exists on Backbone.Model. It has the behavioral and semantic meaning of "destroy this model from the data store". Choosing this name would imply the same behavior and semantics for the MultiPickCollection. This is bad since this object and method will not destroy any data in any backing data store. The choice is clear, then: `close`.

Add a `close` method to the MultiPickCollection's prototype, along side the other instance methods. In this method, call the `stopListening` method of the instance. This will clean up all of the event listeners that were set up using the `listenTo` method.

```
_.extend(Backbone.Picky.MultiPickCollection, {
  // … existing methods

  close: function(){
    this.stopListening();
    this.trigger("close");
  }

});
```

In addition to the `stopListening` call, the `close` method is triggering a “close” event. This one extra line of code offers a lot of flexibility for applications that are using a MultiPickCollection. Knowing when an object is being closed can often make life easier for memory management, app shut down, and other needs.

### listenTo vs on

It should be noted that the `listenTo` method is not a panacea of solutions for memory management related to events. There are times when it makes more sense to use the `on` method instead. For more information on when to use which method, see Appendix A: Managing Events As Relationships.

## The “pickAll” and “unpickAll” Methods

The `pickAll` and `unpickAll` methods loop through the list of models in the collection, and trigger a “picked” or “unpicked” event respectively.

```
pickAll: function(){
  this.collection.each(function(model){
    model.trigger("picked");
  });
},

unpickAll: function(){
  this.collection.each(function(model){
    model.trigger("unpicked");
  });
},
```

Each of these methods iterates the collection and picks or unpicked the model as needed. But wait… notice that these methods trigger the "picked" and "unpicked" events for the models - the same as the PickableModel from Chapter 8. Why duplicate this? Why not use a PickableModel here and not have to duplicate code?

## Necessary Duplication… For Now

Let's say you want to have the MultiPickCollection call the "pick" method for each model instead. How are you going to do that? PickableModel is the object that contains this method, and you don't have a PickableModel for these models. Sure, you could create one for each of the models as you're iterating over them. But where would you store them? Or would you just toss them out after they've done their job? Or, you could try to find the existing PickableModel instance for each of the models… but where would this come from? You would need some sort of global registry or locator for them. That would likely be more code and trouble than it's worth.

This is a real problem, honestly. You don't want to duplicate the code of triggering these events in both the PickableModel and MultiPickCollection. But you also don't want to have to write crazy-glue-code to make things work properly.

For now, the best answer is to to duplicate the events, unfortunately. Having this duplication (at least for the time being) allows the PickableModel and MultiPickCollection to work independently, but still allows the MultiPickCollection to respond to PickableModel triggered events.

You can create a MultiPickCollection without worrying about whether or not the models in the collection have a PickableModel associated.

```
var myCollection = new MyCollection([/* ... model data ... */]);

var mpc = new MultiPickCollection(myCollection);
mpc.pickAll();
```

The MultiPickCollection works just fine on its own. If you happen to have a PickableModel for a given model, though, and this PickableModel has its `.pick()` method called, then the MultiPickCollection instance will still receive the "picked" event and still capture the model as being picked.

```
var m1 = new MyModel();
var pm1 = new PickableModel(m1);
// ...
var mN = new MyModel();
var pmN = new PickableModel(mN);

var myCollection = new MyCollection([m1, /* ... */, mN]);
var mpc = new MultiPickCollection(myCollection);

m1.pick();
```

In this example, the PickableModel and MultiPickCollection are interacting through the use of the events.

Still, this isn't the greatest use of code in that it is causing a couple of lines of duplication. In the next chapter, though, methods of integrating the PickableModel and MultiPickCollection will be discussed. This will offer at least one possible solution to this problem - though the cost vs benefit ratio may not be entirely stable.

## The "togglePickAll" Method

The `togglePickAll` method uses a combination of `selectAll` and `deselectAll`, based on the current count of picked models, according to these rules:

1. If no models are picked, pick all of them
2. If at least one or more, and less than all models are picked, pick all of them
3. If all models are picked, unpick all of them

The first two rules could easily be combined in to a single rule, but leaving them separated makes the behavior a little more explicit. The code, however, does not have to be as explicit as long as the documentation and/or behavioral specification (i.e. "unit tests") for this method is explicit.

```
  togglePickAll: function(){
    var pickedLength = _.size(this.picked);
    var collectionLength = this.collection.length;

    if (pickedLength === collectionLength){
      this.unpickAll();
    } else {
      this.pickAll();
    }
  }
```

Underscore's `size` method is used to get the number of picked items stored in the `picked` attribute. This is the object literal used as a key/value store, and object literals not provide a `length` attribute. Underscore compensates for this by counting the number of attributes on the objects.

Once the length of picked models is determined, it is checked against the length of the collection. If the two lengths are the same, then that means all models are currently picked - so, unpick all of them. If the two lengths are not the same, then there are unpicked models and they need to be picked.

## Handling Removed Models

There's a problem in the MultiPickCollection, exposed by the `togglePickAll` method. If a collection has all models picked and one those models is removed from the collection, the list of picked models will still contain the removed model. This will cause the count to be inaccurate, potentially breaking the `togglePickAll` behavior.

To fix this, the model remove must be accounted for. Add an event handler for the collection's "remove" event to do this. The event handler should be set up in the constructor, along with the other events.

```
Backbone.Picky.MultiPickCollection = function(collection){
  // ... existing code

  this.listenTo(this.collection, "remove", function(model){
    delete this.picked[model.cid];
  });
};
```

Yes, this is code duplication from the "unpicked" event - good catch. And, fortunately, this is the kind of duplication that can easily be rectified. Add a "_remove" method to the MultiPickCollection and have it delete the model from the `picked` list. And while you're at it, add an "_add" method to add a model to the list.

```
_.extend(Backbone.Picky.MultiPickCollection, {
  // ... existing methods

  _add: function(model){
    this.picked[model.cid] = model;
  },

  _remove: function(model){
    delete this.picked[model.cid];
  }
});
```

Now update the constructor function to call these methods instead of having the inline callback functions.

```
Backbone.Picky.MultiPickCollection = function(collection){
  this.collection = collection;
  this.picked = {};

  this.listenTo(this.collection, "picked", this._add);
  this.listenTo(this.collection, "unpicked", this._remove);
  this.listenTo(this.collection, "remove", this._remove);
};
```

The use of an "_" in front of the method name is not anything technical or syntax specific to JavaScript or Backbone applications. This is common notation within JavaScript idioms, though, signifying a "private" method or attribute. That is, when you see a method or attribute that starts with an "_" name, you should treat it like it is private within the object. Methods and attributes named like this are subject to change without notice, because it is expected that you will not touch them directly… ever.

## Using A MultiPickCollection

With the MultiPickCollection implemented, an application can easily provide the ability to pick and unpick models in a list. Similar to the PickableView in Chapter 8, a MultiPickView can be created

```
var MyItemView = BBPlug.ItemView.extend({/* … */});

var MultiPickCollection = BBPlug.CollectionView.extend({
  itemView: MyItemView,

  initialize: function(){
    // build the MultiPickCollection
    this.multiPick = new  Backbone.Picky.MultiPickCollection(this.collection);
  },

  // DOM event handlers

  pickNoneClicked: function(e){
    e.preventDefault();
    this.multiPick.unpickAll();
  },

  pickAllClicked: function(e){
    e.preventDefault();
    this.multiPick.pickAll();
  },

  togglePickAllClicked: function(e){
    e.preventDefault();
    this.multiPick.togglePickAll();
  }
});
```

Since the calls to `pickAll`, `unpickAll` and `togglePickAll` each trigger the correct event, the event handlers for these will be fired at the correct time. The view will be updated with the correct buttons being enabled and disabled, or however the view should handle this situation. This makes for a great reduction in the amount of code that an individual view has to write, to facilitate the “select all” and “select none” functionality that it needs.

### Homework: “all”, “none” and “some” Events

You may find yourself wanting to receive events from the MultiPickCollection, telling you when all, some or no models have been picked. As homework for this chapter, work on adding events that will tell you this. Trigger a “picked:all”, “picked:none” or “picked:some” event at the appropriate time, and have the MultiPickView listen to these events and update the UI accordingly.

## Lessons Learned

Creating a pickable collection introduces a number of interesting challenges. The behavior and semantics introduced with the PickableModel need to be maintained, but the API and implementation are very different at the same time. Working with the “pickable” theme and examining existing methods names on the underlying objects provided a guide to building this extension.

### Adding Events To Your Object

Backbone.Events is a simple object literal that is meant to be used as a mixin for other objects. All of Backbone’s objects mix this in, and your objects can as well. Any time you need the ability to trigger or listen to events in an object, just mix the Backbone.Events in to your object.

## Listening To Events

Backbone.Events provides two mechanisms for listening to events: the `.on` and the `.listenTo` methods. Understanding when to use which can help lead to better memory management and fewer zombie problems. There is no rule or absolute for which to use when, though. Each situation must be evaluated for the needs of the objects that are involved and the lifecycle of those objects.

## Extending A Prototype vs Augmenting An Instance

The use of `_.extend` to add methods to prototypes is powerful. It allows types to quickly and easily be augmented. But this isn't always what you want. There are times when each object instance needs to be augmented with data or instance-specific configuration. Adding default and empty values to an instance can be done in a constructor function, ensuring the object instance has the basic configuration that it needs in order to work correctly.

## Sometimes Duplication Is OK

There will be times when the desire to keep your code DRY - "Don't Repeat Yourself" - will cause more problems than it's worth. The DRY principle is important and duplicating code does tend to lead to problems. But this is not a hard and fast rule. Like any other principle, it is more of a guiding light - a direction that needs to be examined in order to determine its appropriateness.

In the case of duplicating the "picked" and "unpicked" events between PickableModel and MultiPickCollection, the DRY principle was out-weighed by the complexity that would have been added to keep it DRY. Adding layers of complexity and obscurity in API or object requirements is a cost that should be paid in this case.

## Iterate For Edge Cases

No matter how hard you try to determine the exact functionality of code before we sit down and write it, you will never get it 100% complete until you start writing it. Business logic and application workflow are often subtle in nature, with edge cases hanging around places that are very unexpected. As code is implemented, these edge cases may present themselves. When this happens, iterate - take the time to evaluate the edge case and determine if it needs to be handled.

In the case of `togglePickAll`, the edge case of a MultiPickModel having a `picked` size that is inconsistent with the number of models in the `collection` came up. This wasn't noticed until the code in the `togglePickAll` method was written, though. Once the potential problem was identified, though, additional code was written to account for this. The end result is a more robust MultiPickCollection, capable of handling additional scenarios.

## Use Callback Functions, Not Inline Functions

While it is easy and tempting to use nested, inline functions for event handlers, this tends to lead to very bad code. It causes duplication and leads toward the dreaded arrow-of-doom code with nesting so deep, you can't see how the code flow anymore. Instead of writing inline and nested event handler functions, then, use function reference. Call out to another named function and keep the code flat and well organized.

In the case of the MultiPickCollection, a `_remove` method handles both "unpicked" and "remove" events of the collection. This reduced both the complexity in the constructor function, and removed code duplication between the event handlers.

## Use _methodNames For "Private" Members

JavaScript does not have access modifiers built in to the language at this point. It is possible to create privacy by using function scope to hide variables, but this isn't always reasonable to do. Often times a method that is attached to an object needs to be considered "private" even though it cannot have an access modifier that enforces this. In this situation, use an "_" as the first character of the method name. This is a common practice in JavaScript to say, "This method should be considered private. Do not call it directly."

# 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:

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

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:

```
Backbone.ajaxSync = Backbone.sync;

Backbone.getSyncMethod = function(model) {
  if(model.localStorage || (model.collection && model.collection.localStorage)) {
    return Backbone.localSync;
  }

  return Backbone.ajaxSync;
};

// Override 'Backbone.sync' to default to localSync,
// the original 'Backbone.sync' is still available in 'Backbone.ajaxSync'
Backbone.sync = function(method, model, options) {
  return Backbone.getSyncMethod(model).apply(this, [method, model, options]);
};
```

## 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[34] is "[…] patterns for JavaScript modules that work everywhere"

[34] 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 13: A Filtered Collection

Displaying a list of items is one of the most common tasks in any application - and JavaScript and Backbone.js apps are no exception.

Backbone generally makes these two tasks easy. It provides the concept of a Collection to manage a list of items, and you can assign a collection to a View instance for display. But what happens when you need to filter that collection, and only display a subset of what it contains? And what if that sub-set changes when a user selects certain options or types something new in to a search box?

Filtering an in-memory collection is simple, sure. The collection has built in methods to do this. But filtering a collection and then rendering the results can create some unexpected challenges. The journey to find simple and clean code can be unexpectedly difficult.

## Basic Filtering

There are a handful of methods built in to Backbone.Collection that allow you to do various forms of filtering and searching. Most of these methods come from Underscore.js, but some of them don't. Regardless of where they come from, the majority of these methods have one thing in common: they return an array of models from the collection.

Look at the where method[35] for example. When you call this method, you provide a set of filters on model attributes. Any model that matches the specified attribute and value combinations will be returned in an array.

```
var myCollection = new Backbone.Collection([
  {a: "first", b: "second"},
  {a: "third", b: "second"},
  {a: "first", b: "fourth"},
  {a: "second", b: "third"}
]);

var results = myCollection.where({
  a: "first",
  b: "fourth"
});
```

The `results` variable will be an array, with only 1 item in it - the item that matches both `a: "first"` and `b: "fourth"`. Having the result list is great, but now you need to display the list in the app.

[35] http://backbonejs.org/docs/backbone.html#section-101

## View-Controlled Filtering

Displaying a filtered list is fairly easy. The only catch is that you don't have a collection coming back from the collection's filtering methods. You have an array of objects, instead. This tends to catch people by surprise. What do you do with this array? How do you get a Backbone.View to display it, without it being a Backbone.Collection? This becomes especially confusing when the view that should be showing the filtered list is already holding a reference to the original collection.

It would be easy, for example, to call the `.where` method in your view. After all, it's the view that has the collection and the user interactions happen through the view, so why not handle the filter in there, as well?

Using a BBPlug.CollectionView, you would override the `serializeData` method to return the filtered object list.

```
var MyItemView = BBPlug.ModelView.extend({
  template: "#some-model-template"
});

var FilteredView = BBPlug.CollectionView.extend({
  modelView: MyItemView,

  events: {
    "click #run": "runFilter"
  },

  // convert the collection in to an array of
  // items that has been filtered to match the
  // specified criteria
  serializeData: function(){
    return this.collection.where(this.filter);
  },

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

    this.filter = {
      // ... build the filter info from
      // the user input and other settings
    };

    this.render();
  }
});
```

Things are looking good at this point. The code is small, easy to understand and it gets the job done.

This code works for simple scenarios as outlined here. If you're dealing with simple scenarios, there might not be anything wrong with doing this. But there are monsters lurking around the corners - and they are only waiting for the next feature or next requirement for filtering, to jump out and eat your code and productivity.

## Spaghetti Monsters

What happens when you need to change how the filtering works? Instead of just calling `.where`, you need to use other methods of filtering? What happens if some other part of the app sets up a list of default or global filters? How do you handle the need to get those other filter settings, with potential for additional method calls to get the right results in place?

Pretty quickly the FilteredView that you've created begins to become a behemoth of far too many responsibilities. You now have logic for setting up filters (including default or global filters), running the filters, converting the filtered list in to HTML, displaying the resulting list, and handling interactions with that list. Now try to re-use this setup in other areas of the application where similar filtering needs to happen. Or try to use the same logic to get a filtered result set that doesn't need to be displayed at all. Maybe it just needs to be available so that some other code can find the item it needs from the filtered list. What do you do, then? How do you get the filtering to work in those scenarios, and in a consistent manner that doesn't create a spaghetti-mess of code duplication?

Filtering collections in the view that displays the results will get out of hand, quickly. You'll end up with a flying spaghetti monster of tangled responsibilities, tearing apart your productivity and ability to create new features without adding new bugs or duplicate code.

# Self-Filtering Collections

Moving the logic of how to apply a filter into the collection itself, is generally a good idea. This gives you purpose-built collections that can encapsulate the logic and process that you need, behind an easy to use API. But truth be told, most collections don't need anything special. The Backbone.Collection type has the majority of use cases for filtering covered already. You just call the existing methods, as shown the filtering example above.

Separating the responsibilities of filtering vs render is important, though. But even with separation, there are additional problems for which you need to watch out. Many projects have seen code go down in the flames of entanglement and unexpected behaviors because the attempt to separate concerns led to other problems. In the case of filtering collections, watch out for the self-filtering collection anti-pattern.

## The Self-Filtering Anti-Pattern

If you tried to have the original collection filter itself and then update itself with the filtered results, you would end up in a worse position than having the view in control of filtering.

Take the collection from the filtering example above as an example:

```
// custom collection type with custom filtering
var MyCollection = Backbone.Collection.extend({});

// collection instance, with data
var myCollection = new MyCollection([
  {a: "first", b: "second"},
  {a: "third", b: "second"},
  {a: "first", b: "fourth"},
  {a: "second", b: "third"}
]);
```

Now add a custom filter method to this collection and update the same collection instance with the results:

```
// custom collection type with custom filtering
var MyCollection = Backbone.Collection.extend({
  customFilter: function(filters){
    var results = this.where(filters);
    this.reset(results);
  }
});

// collection instance, with data
var myCollection = new MyCollection([
  {a: "first", b: "second"},
  {a: "third", b: "second"},
  {a: "first", b: "fourth"},
  {a: "second", b: "third"}
]);

// filter the collection
myCollection.customFiler({
  a: "first"
});
```

Once you have filtered the collection with the custom method, there are only two items left in the collection. This appears to be the desired result, as only the items with an attribute of `a: "first"` are available now. If you try to apply a new filter, though, it will only run against the items that are currently in the collection … the items that were already filtered.

```
myCollection.customFilter({
  a: "first"
});

myCollection.customFilter({
  b: "second"
});
```

This code runs fine, but it won't produce the results that most users would expect. Instead of re-running the filter against the original list, the second run filters the already filtered items, resulting in zero items to show or use. Any additional filter calls will continue to return zero items, even if the user changes the filter back to the original input that returned two items.

This method has made it very difficult to filter the original collection more than once. The only way to fix this and get the expected results is to reset or reload the collection with the original objects. This is lot of unnecessary work.

The solution, then, is not to update the collection that is being filtered. Instead, allow the collection to return the filtered list as it has been doing. But you may not want to deal directly with a JavaScript array, either.

## Pure Filtering And Clean Collections

The name of the game, here is Separation of Concerns[36]. Keeping the filtering process separate from the filtering results is important. It is just as important as keeping the filtering process out of the view that displays the filtered results.

But having the filtering process separated from the filtering results doesn't mean you have to deal with a raw array of models after applying the filter. Having a Backbone.Collection is useful, after all. And there are times when it makes sense to have a collection return another instance of a collection from a filtering method.

### Creating A Filtered Collection

If you want to use the standard Backbone.Collection features for your filtered list, you have to convert the array in to a Backbone.Collection. This results in two collection instance for filtering: the original collection, and a filtered collection. Once you have the filtered collection, you can pass it to a Backbone.View instance and work with it like any other collection.

Creating a filtered collection instance is as easy as calling `new Backbone.Collection` with the filter results.

[36]http://en.wikipedia.org/wiki/Separation_of_concerns

```
var results = myCollection.where({
  a: "first"
});

var filteredCollection = new Backbone.Collection(results);
```

Yes, it's that easy. Go forth and be merry… well… maybe there's a few more things you could add, to make this even more awesome?

## Returning The Same Collection Type

You might want to have the custom collection type return an instance of the same type from a custom filtering method.

```
// custom collection type with custom filtering
var MyCollection = Backbone.Collection.extend({

  customFilter: function(filters){
    var results = this.where(filters);
    return new MyCollection(results);
  }

});

// ... create myCollection instance, here ...

var filteredCollection = myCollection.customFilter({
  a: "first"
});
```

The advantage here, is having an instance of the same collection type with all of it's methods and capabilities, but those behaviors will only be applied to the filtered list. This can be a very powerful tool when you need to process a list of filtered results, ignoring anything that doesn't match the criteria.

There is a disadvantage to this, though. Every time you call the custom filtering method, you end up with a new collection instance as the result. This can impose yet another problem in updating the filtered view with the new list of items. You will have to inject the new collection in to the existing view, or replace the view entirely. Neither of these options is optimal for very many situations.

Hang on to this pattern, though. There are scenarios where it is useful and you'll want to have it in your tool box for the times where it does make sense.

## Rendering The Filtered Collection

With the filtered collection in hand, create a view that knows how to render a collection. Keep in mind that this view does not need to know anything about how the filters are applied to the collection, though. It only knows that it receives a collection and renders it.

Use the existing BBPlug.View and BBPlug.CollectionView to render the new collection.

```
var MyItemView = BBPlug.ModelView.extend({
  template: "#some-model-template"
});

var FilteredView = BBPlug.CollectionView.extend({
  modelView: MyItemView
});

var filteredView = new FilteredView({
  collection: filteredCollection
});

filteredView.render();
$("#some-element").html(filteredView.$el);
```

This will render the list of items in to the `#some-element` DOM element. The result will be the display of the filtered list, as expected.

Note that you no longer have to override the `serializeData` method, either. The collection that is being rendered is already filtered so there is no need to do this.

Displaying the list is a good start, but it is likely that you'll need to update the list as well. To handle this, you'll need a little more code in the `FilteredView`.

### Updating The Filtered List

When the filter options change, you'll need to update the result set and view to account for the changes. The best way to handle this is with a single filtered collection instance - one to hold the filtered results no matter how many times it changes - and update the view when the filtered collection's contents change.

Responding to the collection changing is only a matter of listening for a "reset" event on the collection. When the filter options change, call `.reset` on the collection, passing in the new result set. The "reset" event will trigger after the collection has been updated, and the view that is rendering the collection can re-populate the DOM with the new results.

```
var filteredCollection = new Backbone.Collection();
var filteredView = new FilteredView({
  collection: filteredCollection
});

// ... render and display the filtered view, here ...

// handle the filter changes when the user
// clicks a "run" button
$("#run").click(function(e){

  // retrieve the list of filters to use
  // the details of this are not important,
  // other than they be formatted as shown
  // in the original example, above
  var filter = {
    // ...
  };

  // get the new list of results from
  // the new filter for the collection
  var results = myCollection.where(filter);

  // reset the filtered collection so that
  // it will update and show the new list
  filteredCollection.reset(results);
});
```

Now when the user changes the filter and clicks the “run” button, the filtered collection will be reset and the filtered view will re-render itself with the new results.

The original collection is still in place, as well. You haven’t modified it at all. This means additional filtering will still produce correct results - results that come from the original collection, and not just a sub-set of the previous filter results.

**Using An Array, Not The Collection Type**

Did you notice that the code to update the filtered list skipped the custom filter method - the one that returned a new instance of the same collection type? Instead, this code opted to call the `where` method directly, receiving an array of items as a result.

If you tried to use the `.customFilter` method, you would create a new collection instance for every filter change. To handle this, you would either have to create a new view instance every time or inject the new collection in to the existing view and then tell the view to re-attach any event handlers and other code that deals with the collection. This is overhead that you don't need since you will be updating the filtered collection instance with the results anyways.

## Lessons Learned

With the built in filtering methods, Properly filtering a Backbone.Collection is fairly simple. The journey to get to simple rendering seems to be anything but, however.

### Looks Can Be Deceiving

The simplicity of the typical use case for a collection and view combination presents a number of challenges when dealing with filters. Having too many concerns wrapped up in the view, returning a collection instance from the filter methods, or trying to update the collection that is being filtered with the results, all seem like reasonable approaches at first. But the subtle differences in usage patterns and behaviors for rendering filtered collections creates a need for a distinct implementation.

### Filter Separate From The Collection

Ultimately, the collection from which you are trying to produce filtered results, should be kept separated from the actual result set. Attempting to filter the original collection will result in a situation where you can only apply a single filter, with additional filters or changes producing incorrect results.

### Handling The Click In The Right Place

Having too many concerns wrapped up in the view, returning a collection instance from the filter methods, or trying to update the original collection with the filter results all seem like reasonable approaches at first. But the subtle differences in usage patterns and behaviors for rendering filtered collections creates a need for something a little different.

The filtering mechanism needs to remain separate from the rendering. And the results of the filter should also remain independent of the original collection. Keeping all of these parts separated will keep your code clean and create opportunities for re-using the filtering or rendering logic.

# Part 4: Application Infrastructure

Beyond views, past models and collections, and after the rest of Backbone has been used within the boundaries it creates, there is still a lot of room for growth. The building blocks that Backbone provides are essential in building well structured applications, well organized code, and maintainable feature sets. But these building blocks don't provide everything you need to build scalable JavaScript application.

Scalability in this case is not a discussion of how many users can run the code at any given time. Since this a browser based JavaScript, it is reasonable to expect that an infinite number of users can use the app at the same time. After all, the browser that runs the code is limited to the current user. It doesn't have to support more than that. Scalability in browser based JavaScript and Backbone apps, then, is the discussion of features. How well can an application be extended to add new features? How much effort does it take to re-organize the features in order to correct the workflow and process? Will all users be given all code for all features, when the log in to the app? Or can the features delivered to the browser be limited to what the user should see?

Part 4 will dive in to these discussions and create the basic infrastructure needed for scaling Backbone applications by feature set. New building blocks will be created to facilitate feature separation and communication between them. An application bootstrap will create the needed organization for application startup. It will also provide the means by which features can be separated and loaded as needed. Finally, a method of segmenting and organizing code within larger features will be created. This will allow features and functional areas of the application to be re-organized a modified as needed, in a much easier manner.

# Chapter 14: 3rd Party Events, Commands, And Requests

One of the core tenets of building a Backbone application of any size, is using an event-driven approach. Everything in the Backbone way of writing code and integrating the different object types relies on events. They are used to communicate changes in models and collections, DOM changes and user interaction, and more. But events are not the only communication means that a scalable application needs.

There are times when the code needs to look beyond events - beyond a statement of "something happened" - and start to look at statements like "do this" and "do this, and let me know". To venture in to these other realms of communication, additional patterns are needed - patterns such as commands and a request/response system.

## Event Aggregators

Events are an integral part of Backbone because they provide a simple yet powerful way of connecting application pieces. One object - a listener - can be notified of something happening from another object - a publisher. The listening object can then respond appropriately and get things done.

The vast majority of events in a Backbone application are set up so that the object listening to an event has a direct reference to the object triggering the event. A Model -> View event relationship is a common example:

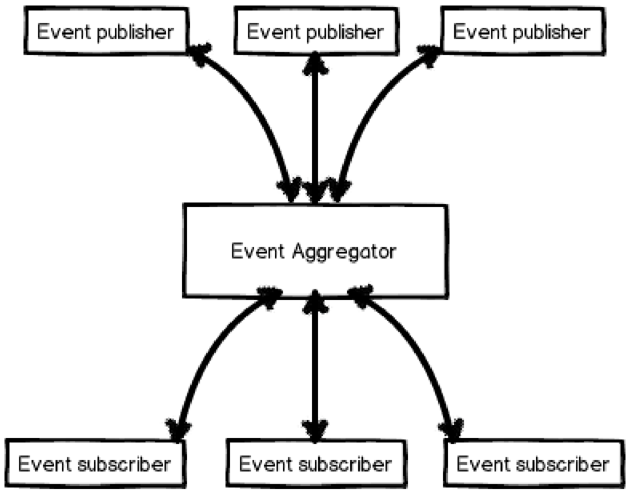
```
var SomeView = Backbone.View.extend({
  initialize: function(){

    // bind to an even from a model, to which
    // I have a direct reference
    this.listenTo(this.model, "change:foo", function(){
      // do stuff
    });

  },
  // ...
})
```

When it comes down to it, building a solid Backbone application requires a fundamental understanding of events and how events can be used to great advantage. But events must be viewed beyond two objects that have a direct reference or relationship. It is not always possible for the listener to have a reference to the publisher.

## 3rd Party Events

When an application begins to grow in size and complexity, it is unreasonable for every object that handles an event to have a reference to the object that publishes an event. Having every object reference each other for events creates a tangled mess of object dependencies. It very tightly couples implementation details of the application's features to each other by crossing boundaries that should not be crossed. This creates a bad situation where changing one part of an application will cause ripple effects that require changes in other parts of the app.

To avoid this, a third party object can be used. An Event Aggregator[37] is an intermediate object that allows other objects to publish events, and objects interested those events to subscribe to them without needing a reference to the publishing object.



Backbone has an event aggregator built in to it: the `Backbone` object itself. This object has `Backbone.Events` mixed in to it, allowing easy event aggregation without having to write any additional code.

A simple use of an event aggregator is view to view communication.

[37]http://martinfowler.com/eaaDev/EventAggregator.html

```
var FooView = Backbone.View.extend({
  events: {
    "click .doIt": "doItClicked"
  },

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

    // trigger an event through the
    // built-in event aggregator
    Backbone.trigger("do:it");

  }
});

var BarView = Backbone.View.extend({
  initialize: function(){

    // subscribe to the "do:it" event from
    // the built in event aggregator
    this.listenTo(Backbone, "do:it", this.doIt);
  },

  doIt: function(){
    // do stuff here
  }
});
```

In this case, the two views need to communicate, but they do not have a direct reference to each other already. When that situation arises, an event aggregator can be used to facilitate the communication. As long as both of the views have a reference to the same event aggregator instance, they can communicate via that 3rd party.

## DIY Event Aggregators

There are times when an event aggregator other than the one built in to Backbone is needed. For example, there may be a sub-application within a larger application. Allowing components with the sub-application to communicate with each other is important. It may also be important to prevent other parts of the over-all application from listening in on this communication. To handle this situation, context-specific event aggregators can be created with only one line of code:

```
var eventAgg = _.extend({}, Backbone.Events);
```

Now I can use the `eventAgg` object anywhere that has a reference to it. If I keep this object's use within a sub-application, then I have effectively created an internal communications mechanism for that sub-app.

## An EventAggregator Type

Having to re-write this one line of code to create new aggregators does get a little tedious, after a while. And although it is simple, it has very low semantic meaning. The only part of this line that tells me the intent is the variable name that is assigned to the result.

To fix the tedium of copy & paste and the nondescript nature of the code, I can build a simple `EventAggregator` type that can be instantiated:

```
// define an event aggregator
var EventAggregator = function(){
  _.extend(this, Backbone.Events);
};

// use the event aggregator type
var eventAgg = new EventAggregator();
```

Having a constructor function (an object type) named `EventAggregator` solves both the copy & paste and ambiguity problem. It allows me to create an instance of an object type that has a clear semantic meaning, and it prevents me from having to remember the slightly odd looking syntax of extending an object literal with `Backbone.Events`.

## No Silver Bullets

Event aggregators are an important concept and are a necessary part of building a scalable Backbone application. However, it doesn't make sense to replace all event bindings with an event aggregator. The primary use case for them is when the object that needs to handle an event does not have a natural relationship or reference to the object that triggers the event. For example, a view that is binding to a model's events does not need to use an event aggregator because the view already has a reference to the model. A menu view, though, may want to use an event aggregator to say a menu item was clicked. This would allow other parts of the application to handle the menu click without needing a reference to the menu.

# Commands

While events are powerful, and 3rd party events through an Event Aggregator even more so, events have a specific semantic that comes with them: "something happened". Unfortunately, I see a lot of semantic violations for events. It's quite common to find code like this, for example:

```
foo.on("save:model", function(model){
  model.save(/* options ... */);
});
```

If I look at the first line of this code, I might think “oh, good. The model was saved. I can go do…” But on looking at the second line, I see that the model has not yet been saved. Instead, this event is being used to tell another object to go save the model. The semantics of “something happened” have been changed and this code actually says “go do this”.

## Building A Command System

Now there isn’t anything wrong with saying “go do this”. The problem with the above code is that the wrong semantics are being used within Backbone’s event system. “Go do this” is the semantic meaning of a command, not an event.

I’m going to build a very simple command system to handle the “go do this” semantic, then.

```
// Create a Commands type to handle commands
var Commands = function(){
  this._handlers = {};
};

_.extend(Commands.prototype, {

  // Handle a specific command by calling
  // the specified handler function
  handle: function(command, handler, context){
    this._handlers[command] = {
      handler: handler,
      context: context
    };
  },

  // Execute the specified command, passing
  // the provided parameters to the handler
  execute: function(command, params){
    params = Array.prototype.slice.call(arguments, 1);
    var config = this._handlers[command];
    if (config){
      config.handler.call(config.context, params);
    }
  }
});
```

With this in place, a command handler can be registered and the command can be properly executed, passing any needed parameters along with the execution call.

```javascript
var cmds = new Commands();

cmds.handle("save:model", function(model){
  model.save(/* options ... */);
});
```

To execute the command, the code that was previously triggering an event can now call the `cmds.execute` method:

```javascript
foo.doSomething = function(){
  cmds.execute("save:model", someModel);
}
```

### ES6 “…params”

Many languages - like Ruby or C#, for example - provide a way to get all parameters passed in to a method, after a certain position. This is often referred to as a “splat” (Ruby) or params list (C#). In the current version of the JavaScript standard - ECMAScript 5 (or ES5) - there is no way of getting an array of all parameters after a current position. ECMAScript 6[38] is introducing a “rest parameters” parameter that will facilitate this: `function(foo, ...bar)`. Until the majority of browsers and other JavaScript runtimes support this feature, though, the Array.slice[39] method can be used to facilitate the same feature.

## Why Have A Command System?

I often get asked why I bother having a command system in my apps. After all, triggering an event works perfectly fine… most of the time… at least, until it doesn’t. But it technically works. I always answer, “semantics”. When it comes down to it, semantics are important. As my friend Sharon Cichelli says,

> Semantics will continue to be important until we find a way to communicate in something other than language.

[38]http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts

[39]https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/slice

Semantics play a role far beyond spoken words. They touch every part of the software that we write, including the code. And there are several semantically important things to note in the code for commands.

First, the `Commands` object has introduced a 3rd party to situation, much like an event aggregator. The introduction of a 3rd party provides an additional level of decoupling that is often needed. It allows the command handler to be registered from one area of the application, and the command execution to happen in another area entirely.

Second, the semantics of commands have been maintained in this code. Instead of having an event that is triggered, forcing a developer to look through the code to find a handler that is actually doing what the event said, this code is saying "go do this". This difference alone can save hours of headache and days of debugging, wondering why an event is causing some such behavior, and why another event handler is getting fired before that behavior.

> "Spent the day figuring out busted tests and trying to decipher other peoplesâ€™ intentions in code." - Chris Hartjes @grmpyprogrammer[40]

Lastly, the same semantics provide a more meaningful and semantically correct API for the `Commands` objects. Rather than re-use the `on` and `trigger` method names from events, the command system provides a `handle` and `execute` method. These method names help to maintain the semantics of commands, and provide distinction between events and commands at an API level.

## Request/Response

Having a command system in place can be a liberating experience once the "AHA!" moment comes around, regarding the semantic difference between an event and a command. But, sometimes a command isn't enough. Sometimes I need to do something more than just say "go do this." Sometimes I need to say, "go do this, and let me know." In this scenario, I'm dealing with a need to send a request and get a response.

### Building A Request/Response System

The basics of a Request/Response system are very similar to that of a Command system. In fact, I've often used Commands as the basis for Request/Response. This does tend to lead to some muddy water, though. In spite of the requirement difference between Commands and Request/Response being so small - just the addition of "and let me know" - the semantic difference can be significant. A method name like `handle` might seem appropriate for requests, but renaming this to `respondTo` provides an much more rich set of semantics in this situation. Other, similar changes can be made in order to provide a very meaningful API, in spite of the similarities in implementation.

[40]https://twitter.com/grmpyprogrammer/status/328984210851364865

```
// Create a RequestResponse type to handle requests
var RequestResponse = function(){
  this._handlers = {};
};

_.extend(RequestResponse.prototype, {

  // Handle a specific request by calling
  // the specified handler function, and
  // returning the result
  respondTo: function(request, handler, context){
    this._handlers[request] = {
      handler: handler,
      context: context
    };
  },

  // Request the specified information, passing
  // the provided parameters to the responder
  request: function(request, params){
    params = Array.prototype.slice.call(arguments, 1);
    var config = this._handlers[request];
    if (config){
      return config.responder.call(config.context, params);
    }
  }
});
```

With the RequestResponse type in place, it can be used to facilitate scenarios where one object needs information from another object, and does not need to know which object can respond to the request.

```
var reqRes = new RequestResponse();

reqRes.respondTo("price:check", function(product){
  var currentPrice = someService.lookupPrice(product.id);
  return currentPrice;
});

var ProductView = BBPlug.ItemView.extend({
  template: "#product-template",

  serializeData: function(){
    var json = this.model.toJSON();
    json.price = reqRes.request("price:check", this);
    return json;
  }
});
```

In this case, I'm using the `BBPlug.ItemView` from earlier chapters to build a view for a product. The data from the product is not quite complete when the view is created, though. I need to reach out and get the latest price for the product and I do so using the request/response system. Once the data is returned, the view can be rendered.

## Why Build A Request/Response System?

The same core reasons for building a Command system also apply to a Request/Response system. The additional requirement of "and let me know", however, adds a new dynamic: return values.

When it comes to the implementation, the only real difference between a Command system and a Request/Response system is that the Request/Response system has a `return` statement for the handler execution. But this one single return statement is part of a larger semantic understanding. When I say `request("price:check")`, there is value in understanding that a request has a response. I expect to receive a response because the semantics of "request" include a response. By comparison, `execute("price:check")` doesn't have the same semantics. It just says "go do a price check" but doesn't say anything about a response.

# Lessons Learned

Scaling Backbone applications introduces a new set of problems, including that of naming things and creating proper semantics in those names. These problems exist within even the smallest of applications, of course, but they become more pronounced as an application grows.

## Create Your Own Types

Backbone provides a foundation of building blocks, but it does not provide every type of object and every piece of functionality needed to build scalable applications. Understanding this, and recognizing the need to build additional object types on top of Backbone is important. Creating additional, specialized object types allows an application's infrastructure to provide much needed functionality, reducing the amount of code the application has, over-all.

The view types that were created in the first two chapters extended directly from Backbone.View. This was the right thing to do because these specialized view types were providing implementations for common functionality within Backbone views. In the case of the `EventAggregator`, the functionality was there but it wasn't available in the needed form for all the desired scenarios. Creating a custom object type and mixing in a little bit of Backbone's provided functionality gives applications new building blocks - more specialized, and more capable.

## Not Every Type Has To Build Off Backbone

The `Commands` and `RequestResponse` types were natural extensions that came from recognizing the semantics of an event vs command, etc. But in building these add-ons, none of Backbone's building blocks were used. This creates a scenario where the code written may be useful outside of Backbone applications. It also shows that we have the freedom to move beyond the core pieces of Backbone and create our own building blocks for our systems' needs.

## Isolate Sub-Systems

As applications begin to grow in size and complexity, it becomes more likely that communication mechanisms will repeat names of events, commands and requests. To avoid potential problems with two independent sub-systems using the same names, create sub-system specific instances of event aggregators, command and request/response systems. This allows each sub-system to have it's own communications bus, freely choosing what names it wants to use.

## Semantics Are Important

With more code and more functionality to keep track of in growing applications, it is important to reduce the amount of overhead in a developer's mind. Proper naming and semantic meaning are important for achieving this goal.

Being able to look at code that was written by someone else - or even written by the person reading it again 3 weeks later - and having an immediate understanding through the semantics of intention revealing names is an easy way to improve developer productivity. By creating an `EventAggregator` type, the intended use of the code becomes clear without having to read the fine details and usage.

## Small Semantic Change, Big API Change

Seemingly unimportant or minuscule differences in requirements or semantics can have very significant effects on a system. It's these subtleties that separate a generalized, high level difference

The requirements differences between a command and a request may be small, but they have created a large semantic and API difference. Instead of just copying the API for Commands in to the Request/Response system, new method names were chosen to reflect the semantics. This allowed the proper semantic meaning to be carried throughout the entire Request/Response system. It would also allow for the Commands and Request/Response systems to vary independently of each other. That is, the Request/Response system would not be forced to take an API change when the command system changes, or vice-versa.

## Similarities Can Be Deceiving

The number of similarities between two sets of requirements can often be deceiving. Our desire, as humans, to see congruence and conformity often leads to a poor understanding of what we are actually looking at. It's important, therefore, to look at the underlying intent and meaning - the semantics of the two things that are similar. It may just be that the similarities are only skin-deep, and that the semantics are pointing the code in two different directions.

# Chapter 15: Boot-strapping A Backbone Application

Languages like C/C++, C#, Java and others typically have a `main` method or similar, as the entry point for an application's code. This is the method that gets called when the application is boot-strapped for execution (however that is done - an executable from a shell, running a GUI application in windows, starting a web service, etc). In JavaScript, there isn't a single defined entry point for code. You write code in a .js file, reference it from the HTML file, and the code gets evaluated and executed in the order that it is included in the page. Evaluation and execution happens in the same order: top-down, first-loaded is first-executed. For small applications this may be fine. But when a Backbone application starts to grow beyond a few simple views and models, something more robust is needed.

## An Application Object

The typical starting point for a JavaScript app is an 'Application' object, which acts as a boot-strapper for the code execution. The first version of this is often an object literal that has an "init" method or "start" method - a single method to call to start the app.

```
// <app.js>

var app = {
  someFeature: {
    views: {},
    models: {},
    controllers: {}
  },
  anotherFeature: {},

  start: function(){
    someFeature.init();
    anotherFeature.init();
    Backbone.history.start();
  }
};

// <someFeature.js>
```

```

app.someFeature.init = function(){
  // do things to start up this feature, here
};
```

The object literal may set up some initial namespaces for various parts of the app by including nested object literals. The “start” method would run initialization for data, collections, and views, and set up a router, for example. This path, a simple object literal with a “start” method, is useful for small application. But for anything substantial, it becomes constraining and problematic, quickly.

Having one “start” method means that it must know about every part of the application that needs to be initialized. This leads to a monolithic method, at best. Worse, though, is that it often leads to large amount of complicated and convoluted logic to try and figure out which parts of the app need to be initialized, when.

To solve the boot-strapping and initialization problem, an application object should allow multiple initialization methods to be registered, and have them run from a single call to a `start` method.

## An Extensible Application

An object literal is a simple way to start, but it would be more flexible if this were an object that could be extended and instantiated as needed. To do that, you’ll define a constructor function to create new object instances. But you don’t want to throw away any investment that you have in your object literal’s structure, either. Fortunately, it’s easy to handle both needs. Allow the object literal to be passed in to the Application’s constructor function. Then extend the Application instance with the object literal.

```
// An Application object constructor function.
function Application(options){
  _.extend(this, options);
}
```

Now you can pass the original object literal in to the constructor of the new Application type.

```
// <app.js>
var app = new Application({
  someFeature: {
    views: {},
    models: {},
    controllers: {}
  },
  anotherFeature: {},

  start: function(){
    someFeature.init();
    anotherFeature.init();
    Backbone.history.start();
  }
});
```

All of the methods and values from the object literal will be copied to the app instance, making them available as if the app object has not been changed. The result is that any existing code that relied on the structure of the object literal will still work. With a new type in place, though, you can create instances that have more behavior than an object literal. The Application type can be extended to include additional features, such as the ability to register initializer methods.

## Adding Initializers

Initializers are functions that get called to initialize - or start up - other areas and feature sets of an application. They allow the Application object to be in control of when the features start, but allow each feature to be in control of how they start.

Before you get to the initializers, extend the Application prototype with Backbone.Events. This will come in handy when you get to the point where the Application is being started, allowing additional flexibility in the use of the Application object.

```
_.extend(Application.prototype, Backbone.Events, {
  // Application instance methods go here
});
```

Next, create an `addInitializer` method and have it accept a callback function as a parameter. You'll need a place to store the callback functions, so add an `_initializers` array to the Application instance, inside of the constructor function.

```
// An Application object constructor function.
function Application(options){
  // extend this instance with all the options provided
  _.extend(this, options);

  // a place to store initializer functions
  this._initializers = [];
}

// Application instance methods go here
_.extend(Application.prototype, Backbone.Events, {

  // Add an initializer, which will be run when the start
  // method is called on the Application instance. The order
  // of execution is not guaranteed.
  addInitializer: function(initializer){
    this._initializers.push(initializer);
  }
});
```

This code isn't much to look at, but it provides a lot of flexibility in use. It also brings up a couple of questions, though.

## Why Add The `_initializers` In The Constructor?

The `_initializers` are added in the constructor function and not extended on to the `Application.prototype` due to the way JavaScript passed objects around by reference.

If the initializers array were defined on the Application prototype, each instance of an Application would reference the same array. In a project where only one Application instance is used, this wouldn't be a big deal. But this would cause serious problems in unit testing or in applications that have multiple Application instances. You don't want the email app running the initializers for the to-do list, for example.

By adding the `_initializers` array in the constructor function, you will have a unique array for each Application instance.

## Why Not Expose `initializers` Directly?

Why should you write a function called `addInitializer` when you could expose the array directly and just call `app.initializers.push(…)` directly?

The short answer is encapsulation.

Exposing the internal data structures of an object can lead to bad situations. Developers using the object and data structure may try to do things that would cause problems with the intended use.

Exposing the array would force developers to know about the API and semantics of arrays. It would also force them to jump outside of their current mode of thinking when working with the array vs the Application API. This breaks concentration and flow every time they have to switch between the semantics of “I’m working on an Application object” to “I’m working with an array”, and back.

Lastly, exposing the internal data structure will cause problems when you need to change that data structure. If someone is using the array in their code and you change it to an object literal with `key: value` pairs, the behavior and semantics will change dramatically. This leads to broken code and frustrated developers.

Keeping the `_initializers` wrapped up in an API method helps to prevent these problems. It encapsulates the internal structure, providing an API that is consistent with the semantics of the object being used. It allows you to change the data structure as needed, and helps to reduce bugs by creating an intention revealing API.

## Executing The Initializers

With the initializers stored in the Application instance, you need a way to execute them. Add a `start` method to the Application and have it loop through the initializers, calling them.

```
_.extend(Application.prototype, Backbone.Events, {
  // … existing methods

  start: function(args){
    var init, i;

    // get the arguments passed to the start method
    // and slice them in to an array so they can
    // be passed to the initializers
    args = Array.prototype.slice.call(arguments);

    // get the # of initializers
    var length = this._initializers.length;

    // loop and execute
    for (i=0, i<length, i++){
      init = this._initializers[i];

      // execute the initializer with the
      // context set to the application,
      // passing args along for the ride
      init.apply(this, args);

    }

  }
});
```

It's useful to provide arguments for the initializer functions that have been added. The easiest way to do this is to pass those arguments to the `start` function directly, forwarding them to each of the initializer functions. The `start` method above does exactly this by slicing the `arguments` object in to a proper array and then passing the args to the `init` function in the loop.

Each of the initializer functions will be executed in the loop, with the context (the `this` argument) of the initializer function set to the Application instance.

### Context And Prototypes

For more information about manage context (`this`) in JavaScript, see my screencast on Context In JavaScript[41].

In addition to context, though, be aware that mixing the tools to manipulate context with prototypal inheritance in JavaScript can result is unexpected bugs. For an introduction to the problem and a description of one or two ways to manage this, see my screencast on When Context And Prototypes Collide[42].

## Using Initializers And Starting The App

Now that the Application object can have initializers added, and they can be started, it's time to finish the conversion from the old application object literal.

### Adding Initializers

The original object literal had it's own start method. When the Application instance is wrapped around that object literal, the `start` method of the object literal overrides the one on the Application prototype. Initializers set up with the `addInitializer` function will never get run if this method is left on the object literal. Fortunately, the new initializer setup is intended to be a replacement for the existing start method. The Application instance, and the other feature files, can be re-written like this:

[41]http://www.watchmecode.net/javascript-context
[42]http://www.watchmecode.net/when-context-and-prototypes-collide

```
// <app.js>
var app = new Application({
  someFeature: {
    views: {},
    models: {},
    controllers: {}
  },
  anotherFeature: {}
};
```

```
// <someFeature.js>
app.addInitializer(function(){
  // do things to start up this feature, here
});
```

```
// <anotherFeature.js>
app.addInitializer(function(){
  // init and run this feature, from here
});
```

Each of the individual feature files now has the initialization code completely contained within itself. Better yet, the "app.js" file is no longer tied directly to each feature's `init` method. This means you can add as many files as you need in your application - each with its own initializer - without touching the Application's start method again.

## Starting The Application

The last thing to do is start the application. This can be done from anywhere in the code. Just call `app.start()` and all of the registered initializers will run. It's common to use a jQuery DOMReady callback function to start the app, for example. And this is often done from within the DOM, at the very bottom of the HTML - that way the rest of the scripts have a chance to be loaded before the app starts.

```
<html>
  <head>
    <!-- … head stuff, here, including script tags as needed -->
    <script src="/js/app.js"></script>
    <script src="/js/someFeature.js"></script>
    <script src="/js/anotherFeature.js"></script>
  </head>
  <body>
    <!-- … the rest of the page … -->

    <script>
    // start the app when the DOM is ready
    $(function(){
      app.start();
    });
    </script>
  </body>
</html>
```

There is one potential problem left in the initializers, though. Initializers that are added after `app.start()` has been called won't be executed. This can happen if you are lazy-loading JavaScript files when a user starts new features in the app (think "GMail" where switching from mail to contacts loads new scripts and data from the server). To make this work, you would have to call the `start` method again, which would be another problem - all of the initializers would be run again, duplicating any work that was done previously. You might try to work around this by `.pop()`ing the initializers from the array in the `start` method. This would work to a point, but you would still have to call the `.start` method multiple times.

A better solution to this problem is to use jQuery's deferred objects[43], which are a variant of JavaScript Promises[44].

## Promising To Run Initializers

According to the CommonJS wiki[45], promises "provide a well-defined interface for interacting with an object that represents the result of an action that is performed asynchronously, and may or may not be finished at any given point in time."

That is, a promise is an object that "promises" to run some functionality at some point in the future. The exact point in time is not known, but as long as the promise is "resolved", then all of the registered functions will be executed.

[43]http://api.jquery.com/category/deferred-object/
[44]http://wiki.commonjs.org/wiki/Promises
[45]http://wiki.commonjs.org/wiki/Promises

You can add a callback function to a promise at any time, and know that the callback will be executed. Even if the callback function is added after the promise has been resolved, the callback will still be executed. This is where the "promise" name comes from - the promise to execute on or after resolution.

The initializers that you set up in the Application object are very close in purpose to the way a promise works. They are a set of callback functions that execute when the application is started. This set up seems to fit the intent of promises perfectly. Adding an initializer will add a the callback to a promise, and starting the Application will resolve the promise.

## Converting Initializers To Promises

Now is the time that having encapsulated the initializers storage behind an `addInitializer` method pays off. Converting the Application's initializer storage from an array to a promise object requires no change in the public API of the Application. It only requires a few changes in the implementation.

Start by replacing the `_initializers` array with a jQuery Deferred instance (jQuery Deferred objects are a variant of Promises, providing the same core features and intent).

```
// An Application object constructor function.
function Application(options){
  // extend this instance with all the options provided
  _.extend(this, options);

  // a place to store initializer functions
  this._initializers = $.Deferred();
}
```

Now change the `addInitializer` method to store the callback function inside of the new `_-initializers` object.

```
addInitializer: function(initializer){
  this._initializers.done(initializer);
}
```

The last change to make is to resolve the deferred object when the application starts.

```
start: function(args){
  // get the complete args list as an array
  args = Array.prototype.slice.call(arguments);

  // resolve the initializers promise
  this._initializers.resolveWith(this, args);

  // trigger the start event
  this.trigger("start", args);
}
```

The promise handles all of the callback execution for you - no more loops, and a lot less code! The `start` function has gone from not quite ten lines of code plus a dozen lines of comments, down to three lines of code and comments each. But most beneficial aspect of this change is that an initializer can now be added after the application is started, and it will still be executed.

```
var app = new Application();

app.addInitializer(function(){
  console.log("I was added before the app was started.");
});

app.start();

app.addInitializer(function(){
  console.log("I was added after the app started!");
});
```

The output of both log statements will be shown, because the promise has been resolved in the start method. With the promise already resolved, any additional callbacks that are added will be executed relatively quickly (if not immediately). This lets you lazy-load JavaScript files as needed - each with their own call to `app.addInitializer` - and have a guarantee that they will be initialized.

## Starting Routers

The Application's initializers provide a powerful and flexible way to organize the initialization code for an app. The ability to load scripts that add initializers at a later time is also very powerful. But this isn't all that a boot-strapper object needs to do.

For example, it's common for large applications to have multiple Backbone.Router instances. But where do you put the call to `Backbone.history.start()`? You don't need, or want, to call this after each router instance is created. But if you put it in an Application initializer, there no guarantee that all (or any) routers will be available when the history is started.

One solution is to trigger a “start” event from the Application’s `start` method. If you trigger this event after all of the initializers have been run, and each of your routers are instantiated in an initializer, you’ll have everything ready to go when Backbone’s history is started.

```
_.extend(Application.prototype, Backbone.Events, {
  // … existing methods

  start: function(args){
    // … existing code

    this.trigger("start", args);
  }
});
```

Pass the same `args` array along to the “start” event. Then you can run code with the same args available, after all of the initializers have been fired.

Now add an even handler to your Application and start the history, there.

```
var app = new Application({
  // … existing code
});

app.on("start", function(args){
  if (Backbone.history){
    Backbone.history.start();
  }
});
```

The `if` checks to see if there is a `Backbone.history` object available. If no router instances have been created, it won’t exist. If at least one router has been defined, though, it will be available. Since you can’t call the `start` method if there isn’t a history object, so it’s a good idea to add this check in place.

If you find yourself writing this same “start” event handler in most of your applications, you might consider adding the history start call directly to the Application’s start method.

```
start: function(args){
  // get the complete args list as an array
  args = Array.prototype.slice.call(arguments);

  // resolve the initializers promise
  this._initializers.resolveWith(this, args);

  // trigger the start event
  this.trigger("start", args);

  // start the routers, if any are defined
  if (Backbone.history){
    Backbone.history.start();
  }
}
```

Now you'll never have to worry about starting routers again. Just start the app and you'll know that any router you created in your initializers (or elsewhere, before the app is started) will be started, too.

## Lessons Learned

Codifying an object from a concept or loosely implemented feature set can introduce a new world of freedom, flexibility and opportunity. Creating an Application object that can be extended, can migrate an existing object literal, and provide the ability to run initializer functions is an important aspects of scaling Backbone applications (or JavaScript in general). It allows you to have a single point of entry for the application, while allowing each area of the application to initialize itself correctly.

### Let The Feature Own Its Initialization

Having a single start method for a simple application works well. Moving beyond small or simple, though, it quickly falls apart. By providing a method of registering initializer functions, the act of starting an application can be decoupled from the functions that run when the application starts. This allows each feature or area of functionality within an application to be in complete control of its initialization. If a feature knows that it should not be available, it does not need to register its initializer with the application.

### Promises, Promises

Promises are powerful objects that can add a lot of flexibility when used appropriately. They provide a mechanism for handling return values when the time of return is completely unknown. Taking advantage of this can lead to many new opportunities and problems solved.

In the case of the Application initializers, switching from a simple array based storage to a jQuery Deferred (promise) provided several benefits. In addition to simplified code, this change allowed for initializers to be executed no matter when they were added. If an initializer is added before the application is started, the initializer is held, waiting for the app to start. If an initializer is added after the application is started, the promise executes the initializer immediately because the promise has already been resolved. This allows additional features and functional areas of an application to be loaded as needed, instead of requiring all features and files to be pre-loaded before starting the application.

## Create Migration Paths

No one wants to adopt yet another all-or-nothing framework when they are looking at fixing or improving an existing application's code base. Providing a migration path that does not require a complete rewrite of the system will increase the adoption rate of a new tool or framework significantly. If developers can simply plug in the new tool and start to use it where it adds immediate value, the will. Over time, the framework should show continued areas of added value that can be reached in incremental and non-intrusive manners.

By providing an Application object that accepts an object literal, and extends all of the methods and attributes from that object literal on to the Application itself, developers are given an opportunity to incrementally improve their own code. An existing investment in an object literal can be leveraged in the migration to the new Application object without massive changes. Once the first step is taken, the next step of migrating from a single `.start` method, out to many initializer methods, becomes apparent. This migration strategy allows applications to grow in to your large-scale framework, instead of requiring a re-write.

## Don't Limit Apps To Backbone's Constructs

Code, libraries and add-ons that support general JavaScript development can often be very useful in Backbone app. And, in fact, tools and techniques from outside of Backbone should be employed. Limiting an application to only the constructs that Backbone provides will directly limit the usefulness of code and hamper a developer's ability to create a clean, well structured system. The application initializers are a good example of how an abstraction can be built to work with any type of JavaScript application. There is no requirement in the code or in the intended purpose of these initializers, that says they can only be used with Backbone application. This same code base could be re-used across any number of JavaScript libraries and frameworks.

And remember: a Backbone application is a JavaScript application. Use all of the power, flexibility and capabilities that JavaScript provides.

## Encapsulation Is More Than Just Theory

It is often easy to expose data structures as part of a public API. This allows other developers to do what they want with that data structure, and abdicates a lot of responsibility from you, the designer of the API. This is not a good thing, by any stretch of the imagination. Providing a clean and clear API for an object or type provides many benefits found in proper encapsulation. It protects internal data structures from unwanted and abusive use. It allows the data structure to change while the API remains in tact. And it allows a developer that is using the API to keep their thought process on that API. They will not have to do mental context switching between your object's API and semantics, and the API and semantics of the data structure that your object happens to use.

## JavaScript And Object References

JavaScript treats primitive types as by-value references. That is, a primitive will be copied between variables that are assigned to the same value. Non-primitive types, on the other hand, are always treated as by-reference. Assign a non-primitive type to multiple variables, and subsequent modification of any one of those variables will be reflected in all of the variables that are assigned.

When designing a JavaScript type, this is an important distinction to understand. If you assigned the `_initializers` to the prototype of the Application, then all instances of an Application would have the same initializers. By assigning the `_initializers` inside of the Application's constructor function, though, this problem is avoided. The use of `this._initializers = ...` as the assignment guarantees a unique initializer set for each instance of an Application.

# 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:

```
EmployeeInfoForm = Backbone.View.extend({
  events: {
    "click .next": "nextClicked"
  },

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

    var data = {
      name: this.$(".name").val(),
      email: this.$(".email").val()
    };

    var employee = new Employee(data);

    this.selectManager(employee);
  },

  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.

# Chapter 17: Building With Components

There are a lot of different definitions of "component" floating around these days. For the purposes of this book, a generalized definition will be borrowed from the up-coming W3C standard for Web Components, and from frameworks such as AngularJS which provide a way to build component based applications.

Components are a combination of visual portions of the application and the logic and processing of that visualization. They encapsulate a small feature or sub-set of a feature in a manner that allows the component to be wired together with other components, creating a larger feature set or functional area of an application. They have at least one view with some amount of logic and process (a.k.a. â€œworkflowâ€�).

**Web Components:**

For more information on Web Components, see this work-in-progress document[46] from the W3C and the Polymer project[47]. AngularJS is also a current framework that builds on the ideas of component based architecture and can be found at AngularJS.org[48].

## Defining A Component Constructor

Defining a Component starts like most other objects - with a constructor function. Components have a visual element to them, though, so you will need a way to manage where the component is displayed. You could pass a jQuery selector in to the constructor function, and possibly duplicate a lot of code that you wrote in Chapter 6. Or, use the Region object from that chapter and allow a region instance to be passed in to the component's constructor.

```
var Component = function(options){
  this.options = options || {};
  this.region = this.options.region;
};
```

[46] http://www.w3.org/TR/2013/WD-components-intro-20130606/
[47] http://www.polymer-project.org/
[48] http://angularjs.org

Now when you create a Component, it will look for a `region` option in the constructor's object literal parameter. But a Component is meant to be heavily customized with the behavior and needs of the functionality encapsulated. To do that, you'll want to allow more than just simple Component instances.

## An `.extend`-able Component

You can borrow several of the techniques that Backbone provides when creating a Component definition. This includes both the use of a `.extend` method, to allow for object extension, and an `initialize` method to allow for a custom constructor function in the Component. It may be useful to allow a Component's region to be specified with the object definition, as well.

```
var Component = function(options){
  this.options = options || {};

  // pull the region from the definition, or the
  // constructor options. precedence goes to the
  // constructor options as an override
  this.region = this.options.region || this.region;

  // execute an `initialize` function if it's there
  if (_.isFunction(this.initialize)){
    this.initialize(this.options);
  }
});

// Borrow Backbone's `extend` method
Component.extend = Backbone.Model.extend;
```

The line that retrieves the `region` checks to see if one was provided in the options, first. If one was, it uses that. If none was provided in the options, a region on an object definition is used (if provided). That way, if you provide a region on a Component definition, you can still override it in the constructor of a specific instance.

The check for the `initialize` function ensures that it is a function, if it exists at all. If it does, it calls the initialize function with the options passed through the constructor. By using `this.options` as the argument, you can ensure the function at least receives an empty object literal. This will help prevent errors and other problems when defining components with an initialize function, later.

Lastly, the `extend` method from Backbone.Model is attached to the Component function. It doesn't matter which Backbone object you pull this function from - it's the same function on all of them. But Backbone does not directly expose this function, so you'll have to pull it from one of the Backbone objects directly.

## Handling Events Within The Component

With Backbone being an event-driven library, and the majority of objects used in Backbone apps able to trigger events, it makes sense for the Component to be able to respond to and handle events. Mixing in Backbone.Events on the Component's prototype will generally take care of this. But it would also be nice to have the Component clean up any event handler it can, when closing the component.

```
_.extend(Component.prototype, Backbone.Events, {

  // allow a Component to be closed
  close: function(){
    // unhook any `listenTo` events
    this.stopListening();

    // close the region
    if (this.region){
      this.region.close();
    }

    // call an `onClose` method for custom cleanup
    if (_.isFunction(this.onClose)){
      this.onClose();
    }
  }
});
```

Providing a `close` method allows any Component instance to be closed. It also gives the Component an opportunity to remove any bound event handlers that were set up with the `listenTo` method of the Component. It also closes the region that it holds on to, if any (this forces the view displayed in the region to close, as well). Lastly, it calls an `onClose` method if one exists - just like the custom View types from the first part of this book.

### Consistent Is Clean

Consistency is an important aspect in API design. Providing the same method names with the same semantics across multiple objects creates a level of consistency and predictability. It also gives developers an easy way to understand the expected uses of the API. All of this leads toward clean use of the API and better code in general.

For more information on good API design, see Brandon Satrom's article on Secrets of Awesome JavaScript API Design[49].

[49]http://webstandardssherpa.com/reviews/secrets-of-awesome-javascript-api-design/

As you can see, there isn't too much behind the Component type. In this case, the idea is more about the semantics and use case of the object type, than the details of the type implementation. Of course there will be additional bits of functionality that can be encapsulated in the core Component type, over time. But other pieces can be added as needed.

With the core of a Component defined, you can begin to build the application specific components.

## Building A FilteredList Component

In chapter 13, you built a collection filtering object that can return a list of items filtered out of a Backbone.Collection instance. But having that object is only half the story. You also need a way for users to provide input for the filter, typically in a search form or other "filter" input box. To do this, then, a filtering component will be built on top of the filtered collection object.

### The Basic Component Options

Define a component that takes three options from the initialize function: a view type (to display the collection) and the collection instance to be displayed will come form the `.options`, and a region in which to display the view is already handled by the base Component.

```
var FilteredList = Component.extend({

  initialize: function(options){
    this.viewType = options.viewType;
    this.collection = options.collection;
  }

});
```

Allowing these three parameters to be passed in to the component means you can change out the view that is used for display, the collection being displayed, and where in the app it is displayed, without having to modify the component implementation.

### Render The Collection, Display The View

Add a `show` method to the component. This method will set up the filtered collection, pass it to an instance of the view type, render the view and finally, show the view in the region.

```
var FilteredList = Component.extend({

  // ... initialize method is here

  show: function(){
    // set up the filtered collection
    this.filteredCollection = new Backbone.Collection();

    // create an instance of the view,
    // with the filtered collection to display
    this.view = new this.viewType({
      collection: this.filteredCollection
    });

    // show the view in the region
    this.region.show(this.view);
  }

});
```

## Filtering From User Input

The view needs to pass the filter information from the DOM out to the component that is controlling the view. There are a number of options for doing this, but the easiest is through the use of an event. Have the view trigger a “filter:updated” event from the DOM click handler for the `#run` button. In this event, include the filter information from the view.

```
var FilteredView = BBPlug.CollectionView.extend({
  // ... existing stuff...

  events: {
    "click #run": "runClicked"
  },

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

    var filterInfo = {
      // get the info from the view inputs
    };

    this.trigger("filter:updated", filterInfo);
  }
});
```

Now the component needs to respond to this event, and run the filtering process against the original collection, passing the results to the filtered collection. The best place to set up the handler is in the `show` method, but you will want to create a separate method for doing the actual filtering - one that can be called from any part of the app that has a reference to the component.

```
var FilteredList = Component.extend({
  // ...

  show: function(){
    // ...

    this.listenTo(this.view, "filter:updated, this.filter);
  },

  filter: function(filterOptions){
    // handle filtering in whatever manner is appropriate
    var results = this.collection.where(filterOptions);
    // populate the filtered list
    this.filteredCollection.reset(results);
  }
});
```

The `filter` method can be called any time the data needs to be filtered and displayed in the view. This can be from the view instance triggering a “filter:updated” event, or from any object that has a reference to the component instance. With this flexibility, the view is not required to trigger this event, either. You could create a view that does not have the input for creating the filter options. In that case, another object outside of the component would be responsible for setting up the filter options and calling the `filter` method.

## Using The FilteredList Component

With everything in place for both the FilteredList component, and the updated FilteredView, you can add it to your application.

```
var myCollection = new Backbone.Collection([
  {a: "first", b: "second"},
  {a: "third", b: "second"},
  {a: "first", b: "fourth"},
  {a: "second", b: "third"}
]);

var myFilterList = new FilteredList({
  viewType: FilteredView,
  collection: myCollection,
  region: someRegion
});

myFilterList.show();
```

You now have a complete component that is displayed on the screen in the region specified. It can filter the collection provided and update the displayed items based on the filters. You can also call the `.filter` method of the component from any code that has a reference to the component instance. And all of this is encapsulated in only a few lines of code where it is being used.

## Lessons Learned

Components are the way of the future in web development. The more we move toward a component based architecture, the more composable our systems become.

### Reduced Cost

Composition provides significantly lower costs when putting systems together and modifying systems. It allows us to plug in and remove pieces as needed.

### Build UI On Top Of Process

Building a component to place in to an application does not require 100% custom / unique code. It is common and recommended that you look at existing processes and visual aspects of your system, and build reusable components on top of those processes.

### Purpose Trumps Implementation

Like many design patterns, we are often faced with code that looks very similar on the surface. The proxy pattern and decorator pattern, for example, are often interchangeable in implementations. The primary differentiator being intention and semantics. The core implementation of a Component is another case where the code shares many details with other pieces that you have been working with. This is not a bad thing. In fact, this is favorable as it creates familiar patterns of use in your applications.

It is important, however, to understand the semantics and differences between what we are calling a Components and what we are calling a View. Just because they look the same in implementation, doesn't mean they serve the same purpose.

# Appendices

# Appendix A: Managing Events As Relationships

In my Scaling Backbone Apps With Marionette talk[50], I have some slides that deal with JavaScript zombies in Backbone apps[51]. 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"?

[50]https://speakerdeck.com/derickbailey/scaling-backbone-dot-js-applications-with-marionette-dot-js
[51]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 B: 3 Stages of App Initialization

There are three different stages in the initialization of a Backbone application. Each of these stages has it's own responsibilities that should be kept separate if an application is to scale properly. Smaller applications may be able to ignore this. Moving beyond anything trivial without accounting for these stages will lead toward more complexity than is necessary in an application.

The three stages of an application's initialization include:

1. Download And Parse
2. Application Initialization
3. State Restoration (optional)

This could break this down in to a significantly larger number of stages if a much more fine-grained and detailed analysis is done. But as a general place to start, identifying 3 stages works well.

## Stage 1: Download And Parse

Consider this the stage where the browser is simply grabbing all of the specified resources from the server and running through the first pass of parsing the HTML, laying out what the static HTML has provided, styling it with the basic CSS definitions, displaying any images and text from the static HTML, and parsing the JavaScript files so it can get ready to run them. Every app does this because without it, your browser doesn't have anything to do or show.

## Stage 2: Application Initialization

Every application has an initialization stage. Some are more elaborate than others. At times you can get away with the default top-down, parse what is found first method of ad-hoc code. Anything more than a few functions, though, and you'll need application object of some sort - whether it's an object literal with an "init" method on it, or a complete Application type with initializers as described in chapter 12. Whatever form this application object takes, your app will need a series of initializers - functions that start up a given area of an application.

### What goes in the initializers?

Initializers are the bits of code, functionality, data and display that are absolutely required for your app to do it's job, no matter what part of the application the user is trying to load up and use. They are the bits that must be initialized before the user can do anything meaningful with your application.

If your app has a menu structure generated by Backbone code, and it must always be present in the application, this should be in an initializer. If you're building a multi-room chat application and you need to list the rooms that the user has favorited in a small block on the screen that is always visible to the user, this should be an initializer. If you're building an image gallery and you need to load a thumb-nail list of images to show, no matter which image the user is trying to view, this is an initializer.

Other parts of your application code that the user doesn't directly see may also be contenders for initializers, too. For example, if you have a router that needs to be up and running, the router probably needs to be instantiated inside of an initializer. But – and this is an important but – the routes on that router should not be executed during initialization. Instantiate the router and wait until after initialization has completed to call the "Backbone.history.start()" method, kicking off your route handlers.

## Stage 3: State Restoration (optional)

This stage is optional because not every application has a context that needs to be restored on start up. Sometimes an application only needs initializers. The classic "todo" application is a great example of this - there is no contextual start up for this app. When the application is loaded, it initializes itself with the list of to do items and then it waits for the user to interact with the list.

When a Backbone application uses a router to respond to url pushState or hash fragment changes, though, it does have a contextual starting point. Each route that a user is allowed to bookmark or copy as a direct link will provide the context that our application needs to use, to get the user back to where they want to be. Even if a Backbone app has a router and contextual start, though, it may not be used. If there user hits the root of the application, there may not be any additional code to run for the empty ("") route. When the user hits a route, though, that route must server up the context and application state that the user expects to see.

The problem in most routed Backbone applications, though, is that they bundle the application initialization with the contextual start. That is, developers tend to use the router and its callback methods as the sole place to get the application initialized and get the user back to the context of the route that they requested. For trivial applications, this may be fine. The initialization code may be so small that it doesn't really matter if it's crammed in to the router. But for any real application with any amount of complexity, this is a bad idea. It couples two very distinct parts of the application startup very tightly, and it can lead to bloated and unmaintainable routers with limited entry points in to the application.

So… what goes in the contextual start, other than just saying route callbacks?

Your route callbacks should be as simple as possible. They shouldn't initialize your system as a whole. Rather, they should be used to determine the state of the application that the user wishes to see. Therefore, the code that goes in to a route callback should be the smallest amount of code that you can write to get your application from it's initial state (the state that it was in when the initializers completed) to the desired state.

## Contextual Examples

In an image gallery application, a user may hit a bookmark that points to a specific image. For example, "#images/4". The route callback that executes should load the requested image and display it on the screen:

```
Backbone.Router.extend({
  routes: {
    "image/:id": "imageById"
  },

  imageById: function(id){
    var image = imageCollection.get(id);
    App.showImage(image);
  }
});
```

In a multi-room chat application, a user may hit a bookmark that should take them directly in to a specific chat room. For example, "#backbone". The route callback that executes should take this room name and call the code that is necessary to enter the chat room.

Chances are this is a fairly involved set of code. You'll need to load the list of users in the chat room. You'll need to clear the current chat windows and possibly pre-load recent messages to be displayed. You may also need to change a browser's websocket event listeners to pick up events for this specific room instead, so that messages for this room can be displayed as they come in.

This is a lot of code to run, and it's fare more code than should be allowed in a route callback method. It should, then, be encapsulated in an object that is responsible for registering the user as having entered that chat room. This object is then called from the router:

```
Backbone.Router.extend({
  routes: {
    ":roomname": "chatroom"
  },

  chatroom: function(roomname){
    ChatApp.enterRoom(roomname);
  }
});
```

Furthermore, there's a high likelihood that the user will be able to enter chat rooms using some interaction on the web page. They may click on a chat room name in their favorite's list, or they may type in a command like "/join #backbone" in to the chat application's command area. For any of the the multiple ways to enter a chatroom, the code that is executed should be the same. Having the route callback be as simple and stupid as possible will promote code re-use and allow these three options to be easily implemented. If you only need to call `ChatApp.enterRoom("someRoom")` to enter any room that the user specifies, providing options for how the user enters a room becomes trivial.

## Lessons Learned

Building a layer of abstraction can often help to identify patterns in applications, as shown with the application initializers. The idea of having two or three distinct phases of an application's start up can help to identify the boundaries between objects and methods, as well. And there are still more valuable lessons that can be pulled from this code, such as not limiting an application's codebase to pure Backbone or Backbone-augmenting objects.

### Making The Implicit Explicit

One of the largest challenges that developers face is understanding and recognizing the implicit or implied parts of an application or process. And a JavaScript application's initialization is no different. Developers often write code that ignores the implicit phases of an application's lifecycle. The code will simply start itself up when it has been loaded, or it will litter a handful of jQuery DOMReady callback methods throughout the application. This can cause maintainability and performance problems in an application.

By making the application start up process more explicit, though, many of these problems can be avoided and new opportunities for organizing code and optimizing the application can be exposed. Having an explicit application definition phase, separated from an explicit initialization phase allows code to know when it will be executed and know what will be available. The third and optional contextual rehydration - using a route or other preserved state to spin up a specific part of the application - allows for even more flexibility, giving the application a solid foundation to run from before moving in to the specific context that the user requested.

## Divide And Conquer

In the previous chapters, the principles of Separation of Concerns and Single Responsibility have been applied in the object and methods within the plugins and add-ons. These principles can and should be applied at higher level abstractions, though. The use of modules in JavaScript in a necessary part of creating large-scale, well functioning, maintainable systems. As Justin Meyer says, "The secret to building large apps is never build large apps. Break your applications into small pieces. Then, assemble those testable, bite-sized pieces into your big application."

Application initializers are not a complete solution for this separation and modularization, of course. Using a module system of some sort - whether it's a simple Immediately Invoking Function Expression, or a framework like RequireJS - helps to keep code separated and organized correctly. The use of application initializers, then, allows a module to be initialized at the appropriate time.

# 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);
      managerSelector.on("save", function(employee){
        employee.save();
      });

    });
  },

  // ...
}
```

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”.

# Appendix D: Better JavaScript Mixins

**Unfinished Chapter**

Note: This chapter is incomplete or contains known errors and inconsistencies. Further work is being done to finish this chapter. All feedback is welcome, still.

Mixins are generally easy in JavaScript, though they are semantically different than what Ruby calls a mixin, which are facilitated through inheritance behind the scenes. The most basic of mixins in JavaScript is done with an "extend" method from a library like underscore.js or jQuery. While this method of copying methods and attributes is powerful, it is also limited.

## Object Extension Is A Poor Man's Mixin

I love the "extend" methods of jQuery and Underscore. I use them a lot. They're powerful and simple and make it easy to transfer data and behavior from one object to another – the essence of a mixin. But in spite of my love of underscore.js and jQuery's "extend" methods, there's a problem with them in that they apply every attribute from one or more source objects to a target objects.

You can see the effect of this in this example from Chris Missal[52]:

```
var start = {
    id: 123,
    count: 41,
    desc: 'this is information',
    title: 'Base Object',
    tag: 'uncategorized',
    values: [1,1,2,3,5,8,13]
};
var more = {
    name: 'Los Techies',
    tag: 'javascript'
};
var extra = {
    count: 42,
    title: null,
```

[52] http://lostechies.com/chrismissal/2012/09/27/extending-objects-with-javascript/

```
    desc: undefined,
    values: [1,3,6,10]
};

var extended = _.extend(start, more, extra);
console.log(JSON.stringify(extended));
```

This produces the following output:

```
{
    "id": 123,
    "count": 42,
    "title": null,
    "tag": "javascript",
    "values": [1,3,6,10],
    "name": "Los Techies"
}
```

In this example, the first object contains a key named “values” and the third object also contains a key named “values”. When the extend method is called, the last one in wins. This means that the “values” from the third object end up being the values on the final target object.

So, what happens if we want to mix two behavioral sets in to one target object, but both of those behavior sets use the same underlying “values”, or “config”, or “bindings” (as reported in this Marionette issue)? One or both of them will break, and that’s definitely not a good thing.

## A Problem Of Context

The good news is there’s an easy way to solve the mixin problem with JavaScript: only copy the methods you need from the behavior set in to the target.

That is, if the object you want to mix in to another has a method called “doSomething”, you shouldn’t be forced to copy the “config” and “_whatever” and “foo” and all the other methods and attributes off this object just to get access to the “doSomething” method. Instead, you should only have to copy the “doSomething” method from the behavior source to your target.

```
var foo = {
  doSomething: function(){
    // ...
  }
}

var bar = {};
bar.doSomething = foo.doSomething;
```

But this poses it's own challenge: the source of the behavior likely calls "this.config" and "this._whatever()" and other context-based methods and attributes to do it's work. Simply copying the "doSomething" function from one object to another won't work because the context of the function will change and the support methods / data won't be found.

For more detail on context and how it changes, check out my JavaScript Context screencast[53].

## Solving The Mixin Problem

To fix the mixin problem then, we need to do two things:

- Only copy the methods we need to the target
- Ensure the copied methods retain their original context when executing

This is easier than it sounds. We've already seen how requirement #1 can be solved, and requirement #2 can be handled in a number of ways, including the raw ECMAScript 5 "bind" method, the use of Underscore's "bind" method, writing our own, or by using one of a number of other shims and plugins.

The way to facilitate a mixin from one object to another, then, looks something like this:

[53]http://www.watchmecode.net/javascript-context

```
var foo = {
  baz: function(){
    // ...
  },

  config: [ ... ]
}

var bar = {
  config: { ... }
};

// ECMAScript "bind"
bar.baz = foo.baz.bind(foo);

// Undescore "bind"
bar.baz = _.bind(foo.baz, foo);

// ... many more options
```

In this example, both the source object and the target object have a “config” attribute. Each object needs to use that config attribute to store and retrieve certain bits of data without clobbering each the other one, but I still want the “baz” function to be available directly on the “foo” object. To make this all work, I assign a bound version of the “baz” function to foo where the binding is set to to the bar object. That way whenever the baz function is called from foo, it will always run in the context of the original source – bar.

## A Real Example

In this scenario, the model instance is extended with the functionality of the SelectableModel. This provides the `select` and `deselect` methods on the model directly, and still triggers the events from the model. Now the model instance can be used within the view with no oddities in accessing the selectable functionality.

```
SelectableView = BBPlug.ModelView.extend({

  initialize: function(){
    this.model.on("selected", this.modelSelected, this);
    this.model.on("deselected", this.modelDeselected, this);
  },

  selectModel: function(){
    this.model.select();
  },

  deselectModel: function(){
    this.model.deselect();
  }

});

var model = new MySelectableModel();
new SelectableView({
  model: model
});
```

The downside to this approach is that the model is now being directly modified by the `SelectabelModel` in both the extension of the model, and when the `select` and `deselect` methods are called. This could have some potential side effects if care is not taken to ensure these methods and attributes are not overridden by other code. The benefit, though, is a more clear use of the selectable methods on the model. With a little planning and possibly some documentation or code comments around the model, the potential side effects can be avoided.

The end result, though, is that the `SelectableModel` is a very flexible object in terms of it's use. It can be created on it's own, with only a reference to a model. It can be mixed in to a model directly. Or it can be used in any of a number of other ways that JavaScript objects can be created, used and destroyed.

Ok, enough "foo, bar, baz" nonsense. Let's look at a real example of where I'm doing this: Marionette's use of Backbone.EventBinder. I want to bring the "bindTo", "unbindFrom" and "unbindAll" methods from the EventBinder in to Marionette's Application object, as one example. To do this while allowing the EventBinder to manage it's own internal state and implementation details, I use the above technique of assigning the methods as bound functions:

```
Marionette.addEventBinder = function(target){

  // create the "source" of the functionality i need
  var eventBinder = new Marionette.EventBinder();

  // add the methods i need to the target object, binding them correctly
  target.bindTo = _.bind(eventBinder.bindTo, eventBinder);
  target.unbindFrom = _.bind(eventBinder.unbindFrom, eventBinder);
  target.unbindAll = _.bind(eventBinder.unbindAll, eventBinder);
};

// use the mixin method
var myApp = new Marionette.Application();
Marionette.addEventBinder(myApp);
```

Now when I call any of those three methods from my application instance, they still run in the context of my eventBinder object instance and they can access all of their internal state, configuration and behavior. But at the same time, I can worry less about whether or not the implementation details of the EventBinder are going to clobber the implementation details of the Application object. Since I'm being very explicit about which methods and attributes are brought over from the EventBinder, I can spend the small amount of cognitive energy that I need to determine whether or not the method I'm creating on the Application instance already exists. I don't have to worry about the internal details like "_eventBindings" and other bits because they are not going to be copied over.

## Simplifying Mixins

Given the repetition of creating mixins like this, it should be pretty easy to create a function that can handle the grunt work for you. All you need to supply is a target object, a source object and a list of methods to copy. The mixin function can handle the rest:

```
// build a mixin function to take a target that receives the mixin,
// a source that is the mixin, and a list of methods / attributes to
// copy over to the target

function mixInto(target, source, methodNames){

  // ignore the actual args list and build from arguments so we can
  // be sure to get all of the method names
  var args = Array.prototype.slice.apply(arguments);
  target = args.shift();
```

```
    source = args.shift();
    methodNames = args;

    var method;
    var length = methodNames.length;
    for(var i = 0; i < length; i++){
      method = methodNames[i];

      // bind the function from the source and assign the
      // bound function to the target
      target[method] = _.bind(source[method], source);
    }

  }

  // make use of the mixin function
  var myApp = new Marionette.Application();
  mixInto(myApp, Marionette.EventBinder, "bindTo", "unbindFrom", "unbindAll");
```

This should be functionally equivalent to the previous code that was manually binding and assigning the methods. But keep in mind that this code is not robust at all. Bad things will happen if you get the source's method names wrong, for example. These little details should be handled in a more complete mixin function.

## An Alternate Implementation: Closures

An alternate implementation for this can be facilitated without the use of a "bind" function. Instead, a simple closure can be set up around the source object, with a wrapper function that simply forwards calls to the source:

```
  // build a mixin function to take a target that receives the mixin,
  // a source that is the mixin, and a list of methods / attributes to
  // copy over to the target

  function mixInto(target, source, methodNames){

    // ignore the actual args list and build from arguments so we can
    // be sure to get all of the method names
    var args = Array.prototype.slice.apply(arguments);
    target = args.shift();
    source = args.shift();
```

```
      methodNames = args;

      var method;
      var length = methodNames.length;
      for(var i = 0; i < length; i++){
        method = methodNames[i];

        // build a function with a closure around the source
        // and forward the method call to the source, passing
        // along the method parameters and setting the context
        target[method] = function(){
          var args = Array.prototype.slice(arguments);
          source[method].apply(source, args);
        }

      }

    }

    // make use of the mixin function
    var myApp = new Marionette.Application();
    mixInto(myApp, Marionette.EventBinder, "bindTo", "unbindFrom", "unbindAll");
```

I'm not sure if this version is really "better" or not, but it would at least provide more backward compatibility support and fewer requirements. You wouldn't need to patch 'Function.prototype.bind' or use a third party shim or library for older browsers. It should work with any browser that supports JavaScript, though I'm not sure how far back it would go. Chances are, though, that any browser people are still using would be able to handle this, including – dare I say it? – IE6 (maybe… I think… I'm not going to test that, though :P )

## Potential Drawbacks

As great as all this looks and sounds, there are some limitations and drawbacks – and probably more than I'm even aware of right now.

We're directly manipulating the context of the functions with this solution. While this has certainly provided a measured benefit, it can be dangerous. There may be (are likely) times that you just don't want to mess with context – namely when you aren't in control of the function context in the first place. Think about a jQuery function callback, for example. Typically, jQuery sets the context of a callback to the DOM element that was being manipulated and you might not want to mess with that.

In the case of my EventBinder with Marionette, I did run in to a small problem with the context binding. The original version of the code would default the context of callback functions to the object that “bindTo” was called from. This meant the callback for “myView.bindTo(…)” would be run with “myView” as the context. When I switched over to the above code that creates the bound functions, the default context changed. Instead of being the view, the context was set to the EventBinder instance itself, just like our code told it to. This had an effect on how my Marionette views were behaving and I had to work around that problem in another way.

There certainly some potential drawbacks to this, as noted. But if you understand that danger and you don’t try to abuse this for absolutely everything, I think this idea could work out pretty well as a way to produce a mixin system that really does favor composition over inheritance, and avoids the pitfalls of simple object extension.

## Backbone.Include

There’s a library from Anthony Short that aims to solve the same problem for Backbone, specifically. Check out Backbone.Include[54] for Anthony’s implementation.

[54]https://github.com/anthonyshort/backbone.include

# Appendix E: Theme And Variation

There are two hard problems in software development:

1. Naming things
2. Cache invalidation
3. Off by one errors

> The "off by one" addition is a variation of the a quote by Phil Karlton, which says the two hard things in computer science are cache invalidation and naming things. The variation was popularized by Martin Fowler in a short blog post[a].
>
> [a] http://martinfowler.com/bliki/TwoHardThings.html

Naming things truly is difficult. Names provide meaning and expectations. Things that are named randomly or by different convention will appear unrelated. It is important to name things properly, then. And naming things with a theme can often help.

The theme of the project often leads to the names of the individual items. Naming the items by this theme makes them easier to remember and understand, if the theme is chosen properly. If a theme is chosen poorly, though, it can cause as many problems as were solved - or more.

## Selecting vs Picking

Part 2 of the book is devoted to building a plugin named "Backbone.Picky". The name comes from the idea of "picking" the models and that are needed. However, the original method names, attributes and other aspects of the code did not follow the theme of being "picked". Instead, the theme changed to "selection" in the API. Models could be "selected" and "deselected" with a `.select()` and `.deselect()` method. This works, technically. Computers don't care what names are given to things. But from a human perspective - which is the only perspective that matters, honestly - naming is very important. By switching from a "picked" theme to a "selected" theme in the method and attribute names, several problems were created.

## A Variation On Theme

The idea of a "theme" isn't anything new in software development. This practice has been around for a long, long time. It has also undergone many different name changes, including the use of "metaphor" at the height of the eXtreme Programming (XP) days.

Whatever it is called, the theme of a project is something that often needs to be chosen carefully. I've heard many a story, for example, of XP teams picking a strong metaphor for a code base and having it bite them in the butt. For example, one person I talked to recounted a metaphor of a "warehouse" for a system. They had objects, services and code centered around equipment and actions that would be found in a warehouse. The problem was that this system had nothing to do with warehousing. Having a `forklift.liftAndShelve()` method for a system that dealt with customer relations (or whatever it was) may have been fun at first, but it quickly fell apart and became a source of problems.

In the end, themes or metaphors are important because they create continuity. But they can be taken too far if applied incorrectly. Be sure to pick a theme that makes sense for the system at hand.

The first of many problems that can be created by not sticking with the theme is confusion in the API and use. The name of the project creates a sense of expectation in the language used around the project. If a developer is talking about picking items in a collection, then it would make sense the `.pick()` them. Saying `.pick()` implies an opposite method of `.unpick()`. It sounds a little odd to say "pick" and "unpick", though, so the names were changed to "select" and "deselect". This lead to a number of questions and issues in the project, as developers that were using it were wondering why the theme for naming was different. It was confusing having to think about the project name and then use method and attribute names that didn't follow the theme.

So, then change the name of the plugin to "Backbone.Selectable" or something like that, right? Then the names of the methods and attributes can revolve around the more natural language of "selection": a model can be "selected" and "deselected." This creates a much more subtle problem, though - one of semantics, expected behavior, and overriding methods.

When it comes time to give a Backbone.Collection the methods that allow it to select models or deselect models, the "select" theme breaks. Backbone.Collection already has a method called `.select()` - and it has nothing to do with marking models as selected. Rather, it is used to return an array of models that match a certain criteria.

It's arguable as to whether or not the semantics of "select" are being broken in this case. Is the act of "selecting" a model and merely marking it as selected vs getting a reference to it really any different? The end result is an identification of which models are selected, after all. The real problem is in the behavioral change, then. If this plugin created a method named "select" and added it to the collection instance, it would be replacing the existing select method and it would change the behavior of that method. It would be possible to write this method in a way that preserves the existing behavior of the collection's "select" method, of course. This may or may not be acceptable, and may or may not be difficult. It does add yet another requirement to the code, though. It also creates an unexpected modification to the behavior in the "select" method of collections. A developer that has become accustomed to the "select" method as defined by Backbone.Collection directly, may find the new behavior to be problematic. At the very least, it would be surprising to see that a method they have been using for however long, suddenly has additional behavior and semantics.

## Knowing When To Move On

The solution to all of the problems that were created around the theme of "selection" was not to continue hacking on the code and documenting the changes in bold letters as was originally done. This only served to exacerbate the problems - like putting a band-aid made of sand paper and rubbing alcohol on a large wound. Rather than fixing the actual problem, this creates a worse situation that tends to lead toward other bad things.

The real solution is to change the method names and attributes to match the theme of the plugin. And sometimes, changing the theme of the plugin to start with. Instead having "select" and "deselect" methods, they will be "pick" and "unpick". The language used around these methods will also be changed. The behavior of the methods can then be focused to the needs and not have to worry about breaking semantics or behavior in existing methods.

## Lessons Learned

"Picking" a theme (see what I did there? :P) and sticking with it can spell the difference between success and failure in a project. A well chosen theme can create a sense of cohesiveness and unity. But a poorly chosen theme can have the opposite effect. Even if a theme seems to be well chosen at first - as was the case with Backbone.Picky and the "selection" theme - it can prove to be terrible in the end.

### Don't Mix Themes

One of the easiest mistakes to make in choosing a theme is creating a set of expectations in a project name, and then missing those expectations in the project API. If a theme is chosen for a project, then name should reflect the same thing as the API.

In the case of Backbone.Picky, the project name did not match the API theme. This caused a lot of confusion and questions. It would have been better to name the project something entirely unrelated to "selection" or "picking" than to have a name that came close to the theme, but with different words and names.

## Don’t Let A Good Theme Go Bad

Picking a theme for a project is often easy at the outset. It may seem obvious what a project theme should be. But even the best themes can go bad over time. Always be on the look out for the limitations of a theme and be ready to fix the problems when they occur, not just work around them. This may mean a deeper examination of the names used from the theme, or choosing another theme entirely.

## Be Wary Of Re-using Existing Function Names

When a plugin or add-on uses the same name for a function that is used by the types that are being augmented, problems can ensue. If the plugin is mixed in to the targeted type or instance of that type, the results can be detrimental causing a number of unknown and difficult to track down bugs. Whenever possible, use method and attribute names that are not typically part of the base types.

# About Derick

Hello, my name is Derick Bailey.

I'm a software developer and entreprenuer, a 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.

My career has spanned many different tools, technologies, platforms and languages. I began working with JavaScript in Netscape 2.0, and found it to be both fun and powerful at the time. For the next 10 years, I had a love/hate relationship with the language, until I was introduced to Backbone.js in mid 2011. I almost immediately fell in love with Backbone, as I saw the potential for large scale, event-driven, composite applications being built in JavaScript with it. I've spent my time since then working with Backbone, blogging about it, training other developers to work on it, and building many different add-ons and plugins.

You can find those writings, plus much more, at my blog. I also produce screencasts, provide information about my consulting services, and more, through the following websites:

- My Company: MutedSolutions.com[55]
- My Blog: DerickBailey.com[56]
- Screencasts (free and paid): WatchMeCode.net[57]
- Open Source Projects: GitHub.com/DerickBailey[58]

If you have any questions, comments or concerns, you can contact me at the following:

[55]http://mutedsolutions.com
[56]http://derickbailey.com
[57]http://watchmecode.net
[58]http://github.com/derickbailey

- Email: derick@mutedsolutions.com[59]
- Twitter: @derickbailey[60]

Or contact me about the book, specifically:

- Email: backboneplugins@mutedsolutions.com[61]
- Twitter: @BackbonePlugins[62]

[59]mailto:derick@mutedsolutions.com
[60]http://twitter.com/derickbailey
[61]mailto:backboneplugins@mutedsolutions.com
[62]http://twitter.com/backboneplugins

# About Jerome Gravel-Niquet

Jerome is the author of the Backbone.LocalStorage[63] plugin, and of chapter 12 of this book.

[63]https://github.com/jeromegn/Backbone.localStorage

# A Special Offer From Derick

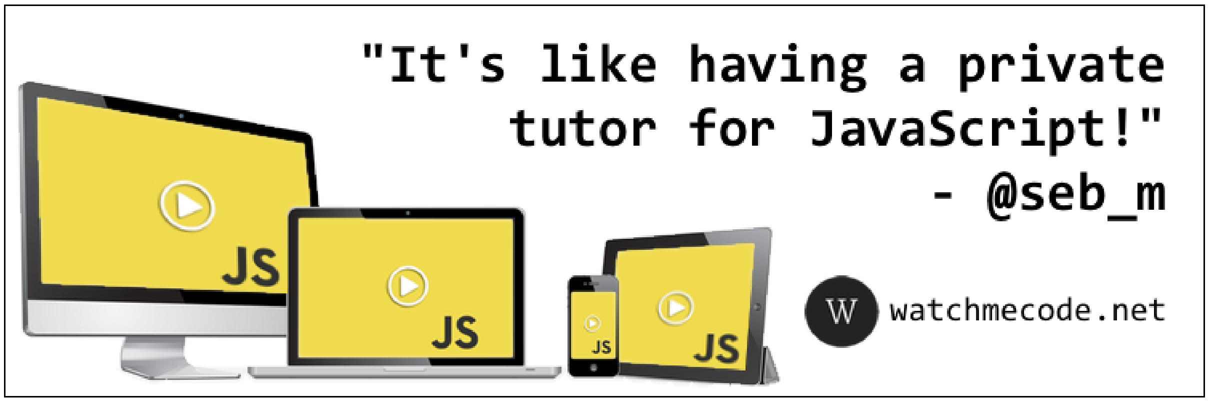
Thanks again for taking the time to read this book! I sincerely hope that it has helped you to improve your Backbone.js projects, and possibly helped you move forward with JavaScript in general. To that end, though, I'd like to offer you something special to help you continue to move forward on your JavaScript journey.

As I mentioned in my profile a few pages back, I run a JavaScript screencast subscription service at WatchMeCode.net[64]. I've mentioned a few of these screencasts in the book as a way to get additional information about the language that runs the web as we know it.

There are far more screencasts available than what I've mentioned in this book, though, and I would like to make them all available to you at a discounted price! Head over to WatchMeCode and sign up with the following discount code to get 35% off the monthly subscription:

- URL: **watchmecode.net/backbonepluginsbook**[65]
- Discount Code: **backbonepluginsbook**
- Discount Amount: **35% off!**

At WatchMeCode.net, you'll find everything from the language fundamentals to the best of today's tools and techniques to help you master the art of JavaScript.



[64]http://watchmecode.net
[65]https://sub.watchmecode.net/backbonepluginsbook