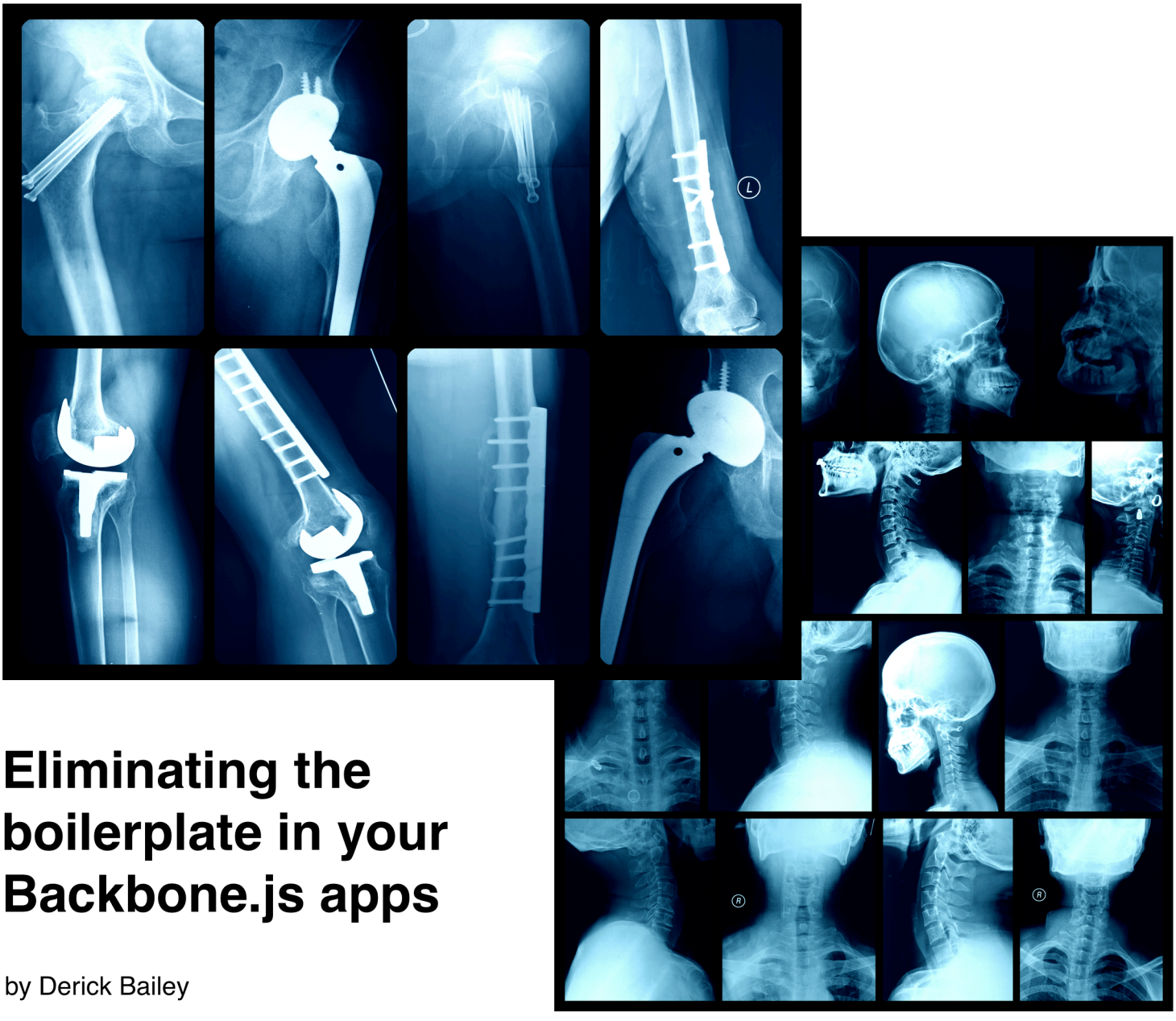
Prosthetics And Orthotics



Eliminating the boilerplate in your Backbone.js apps

by Derick Bailey

Prosthetics And Orthotics

Eliminating the boilerplate in your Backbone.js apps

Derick Bailey

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

This version was published on 2013-05-26

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

Tweet This Book!

Please help Derick Bailey by spreading the word about this book on Twitter!

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/#BackbonePlugins

Contents

A Work In Progress

This book is a work in progress, with many of the chapters complete, additional chapters in rough draft, and one or two chapters planned beyond that. Expect this book to change over time, with edits and additional chapters being added. The amount of demand I am seeing from early adopters will largely determine the schedule that I keep for additional releases, and the eventual price that is charged for the finished product.

Please Provide Feedback

Any and all feedback that I receive on the book is greatly appreciated. Whether it's positive, negative, grammer and speling (see what I did there? :D), content suggestions or otherwise.

Please feel free to contact me with any feedback that you have. The more feedback I get, the better the book will be.

- twitter: @BackbonePlugins[1]
- email: backboneplugins@mutedsolutions.com[2]

Please feel free to leave comments on the Leanpub page for this book as well.

[1] http://twitter.com/backboneplugins
[2] mailto:backboneplugins@mutedsolutions.com

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 peices. 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 peices fit together, and how those peices 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 peices 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[3]
- My Intro To Backbone Screencasts[4]
- My Additional Resources[5]

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.

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

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

[5] 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[6] or Tim Branyen's Backbone.LayoutManager[7] 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[8], 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[9].

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[10], 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.

[6] http://github.com/derickbailey/backbone.marionette

[7] https://github.com/tbranyen/backbone.layoutmanager

[8] https://github.com/PaulUithol/Backbone-relational

[9] https://github.com/documentcloud/backbone/wiki/Extensions%2C-Plugins%2C-Resources

[10] 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.

For a general introduction to JavaScript, Backbone, protoypal inheritance, design patterns or test driven development, please see Appendix B for a list of resources.

Part 1: Managing The DOM

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, populating the rendered HTML in to the view, and more. Beyond rendering, though, there are additional problems that lend themselves to boilerplate. Widgets and control frameworks often need special treatment, for example. And deciding where and when to show a view instance in the DOM often creates repeated code, as well.

In part 1, the boilerplate of views and managing the DOM 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. Managing widgets and controls, and figuring out a consistent and simple way to place views in the DOM and remove them from the DOM will also be tackled.

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.toJ\
SON());
    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){ %><\
%= message %> <% }) %>!</h1>", data);

    this.$el.html(renderedHtml);
  }

});

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

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

view.render();

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

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

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

Extract Whats Common, Specify The Differences

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

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

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

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

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

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

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

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

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

});
```

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

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

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

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

});
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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.

```
1  CommentView = BBPlug.BaseView.extend({
2    template: "...",
3    serializeData: function(){...},
4
5    render: function(){
6      // use the BaseView's prototype to do the default render
7      BBPlug.BaseView.prototype.render.call(this);
8
9      // call moment.js' to format our dates correctly
10     var commentDate = this.model.get("date");
11     var timeAgo = moment(commentDate).fromNow();
12
```

[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;
  },
```

```
16
17
18    render: function(){
19      // ...
20    }
21  });
```

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.

```
1   // Model View
2   // ----------
3   MyView = BBPlug.BaseView.extend({
4     template: "..."
5   });
6
7   myView = new MyView({
8     model: myModel
9   });
10
11  myView.render();
12
13
14  // Collection View
15  // ---------------
16
17  MyCollectionView = BBPlug.BaseView.extend({
18    template: "..."
19  });
20
21  myCollectionView = new MyCollectionView({
22    collection: myCollection
23  });
24
25  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 te\
mplate
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
});

```

```
13 var commentsView = new CommentsView({
14   collection: blogPost.comments
15 });
```

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: 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[13] and this blog post[14] 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.

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

[14] 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[15], 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[16]:

[15] https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/GetPrototypeOf
[16] 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[17]. Episode #4[18] covers JavaScript's context (`this`) keyword while Episode #5[19] 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.

[17] http://watchmecode.net

[18] http://www.watchmecode.net/javascript-context

[19] 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[20] provide a lot of backward compatibility for modern features, but a quick search for specific methods such as Object.getPrototypeOf[21] or Object.create[22] will often surface individual implementations, reducing the need for something as large as Modernizr.

[20] http://modernizr.com/

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

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

Chapter 5: 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[23] suite, commercial products such as Kendo UI[24], 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:

[23] http://jqueryui.com
[24] 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[25], 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:

[25] 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.

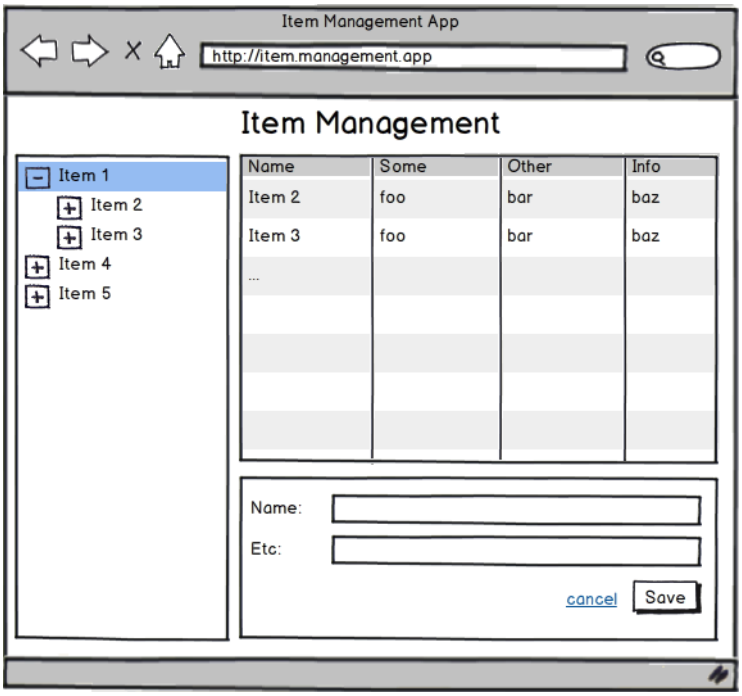
Chapter 6: Swapping Views In The DOM

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



region illustration

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):

```
1  // Define a "Region" object that is hard coded
2  // to manage the "#mainregion" DOM element
3  Region = (function (Backbone, $) {
4    var el = "#mainregion";
5    var currentView;
6    var region = {};
7
8    // A method to close the current view
9    var closeView = function (view) {
10     if (view && view.remove) {
11       view.remove();
12     }
13   };
14
15   // A method to render and show a new view
16   var openView = function (view) {
17     view.render();
18     $(el).html(view.el);
19   };
20
21   // 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.

Part 2: 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.

But first, a brief diversion in to themes, naming projects and the potential problems that naming something "select" vs "pick" can cause.

Chapter 7: 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

This part 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.

Chapter 8: 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.

```
1 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.

```
1 Backbone.Picky.PickableModel = function(model){
2   this.model = model;
3 };
```

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

```
1 var myModel = new MyModel();
2 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 9: 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.

```
1  // Constructor function to create MultiPickCollections
2  Backbone.Picky.MultiPickCollection = function(collection){
3    this.collection = collection;
4    this.picked = {};
5  };
```

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.

```
1  var myCollection = new MyCollection([/* ... model data ... */]);
2
3  var mpc = new MultiPickCollection(myCollection);
4  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.collectio\
n);
  },

  // DOM event handlers

```

```
14    pickNoneClicked: function(e){
15      e.preventDefault();
16      this.multiPick.unpickAll();
17    },
18
19    pickAllClicked: function(e){
20      e.preventDefault();
21      this.multiPick.pickAll();
22    },
23
24    togglePickAllClicked: function(e){
25      e.preventDefault();
26      this.multiPick.togglePickAll();
27    }
28  });
```

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 Placeholder

Chapter coming soon ...

Chapter Placeholder

Chapter coming soon ...

Part 3: 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 3 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 11: 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
    });

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

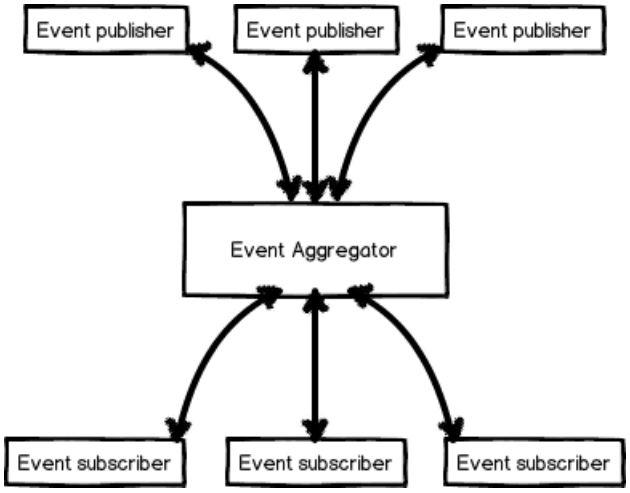
When it comes down to it, building a solid Backbone application requires a fundamental understanding of events and how events can be used to great advantage. But events must be viewed beyond

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 reasonable 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[26] 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.

[26] 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:

```
1  // define an event aggregator
2  var EventAggregator = function(){
3    _.extend(this, Backbone.Events);
4  };
5
6  // use the event aggregator type
7  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.

```
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:

```
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[27] 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[28] 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.

[27] http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts

[28] 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[29]

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.

[29] 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.

Appendices

Appendix A: Managing Events As Relationships

In my Scaling Backbone Apps With Marionette talk[30], I have some slides that deal with JavaScript zombies in Backbone apps[31]. 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

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

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

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"?

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:

```
1 // a subject triggers an event
2 subject.on("some:event", function(){
3   // the callback function is the observer
4   // of the event in this case. do stuff
5   // here because the "some:event" event was
6   // triggered
7 });
```

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.

About Me

Hello, my name is Derick Bailey.

I'm a software developer, consultant, screencaster, blogger, speaker, trainer and more. I've been working professionally in software development since the late 90's and have been writing code since the late 80's, and am currently working as a Developer Advocate for Kendo UI[32].

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:

- Employer: Telerik[33].
- Position: Developer advocate for Kendo UI[34]
- Personal Company: MutedSolutions.com[35]

[32] http://kendoui.com
[33] http://telerik.com
[34] http://kendoui.com
[35] http://mutedsolutions.com

- Technical Blog: DerickBailey.LosTechies.com[36]
- Screencasts (free and paid): WatchMeCode.net[37]
- Open Source Projects: GitHub.com/DerickBailey[38]

If you have any questions, comments or concerns, you can contact me at the following:

- Email: derick@mutedsolutions.com[39]
- Twitter: @derickbailey[40]

Or contact me about the book, specifically:

- Email: backboneplugins@mutedsolutions.com[41]
- Twitter: @BackbonePlugins[42]

[36] http://derickbailey.lostechies.com
[37] http://watchmecode.net
[38] http://github.com/derickbailey
[39] mailto:derick@mutedsolutions.com
[40] http://twitter.com/derickbailey
[41] mailto:backboneplugins@mutedsolutions.com
[42] http://twitter.com/backboneplugins