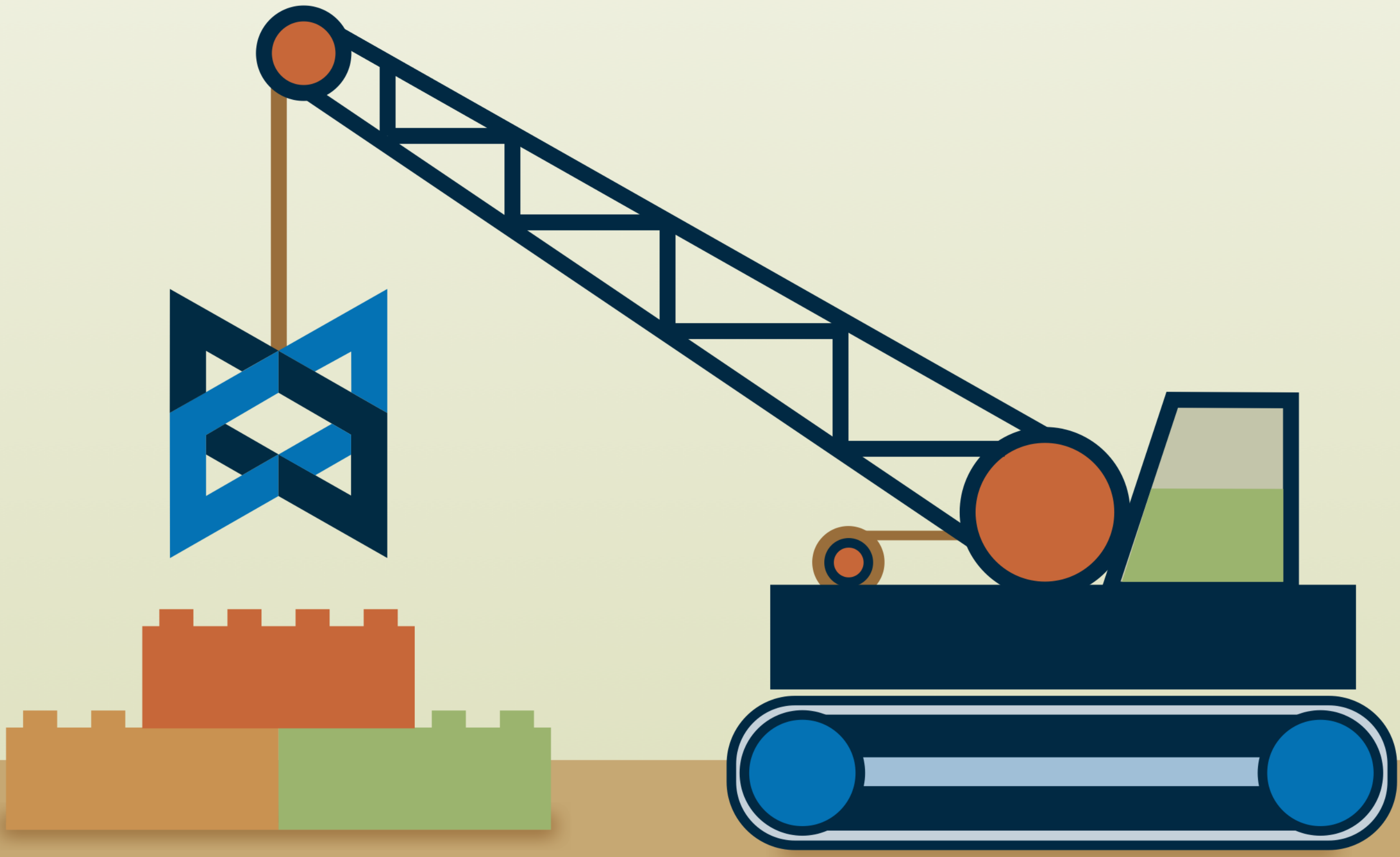
Covers Backbone.js v1.0

BUILDING BACKBONE PLUGINS



By Derick Bailey

Eliminate The Boilerplate In Backbone.js Apps

Building Backbone Plugins

Eliminate The Boilerplate In Backbone.js Apps

Derick Bailey and Jerome Gravel-Niquet

Tweet This Book!

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

The suggested tweet for this book is:

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

The suggested hashtag for this book is #BackbonePlugins.

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

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

Contents

Chapter 1: View Rendering

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

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

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

});

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

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

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

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

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

});

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

view.render();

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

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

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

  render: function(){

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

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

    this.$el.html(renderedHtml);
  }

});

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

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

view.render();

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

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

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

Extract Whats Common, Specify The Differences

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

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

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

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

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

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

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

```
Backbone.View.extend({

  render: function(){

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

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

});
```

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

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

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

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

});
```

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

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

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

  render: function(){
    var data = this.serializeData();
    var renderedHtml = _.template(this.template, data);
    this.$el.html(renderedHtml);
  }
});
```

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

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

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

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

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

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

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

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

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

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

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

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

Lessons Learned

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

Solve Real Problems

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

Extract Common Code

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

Specify The Differences

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

Backbone.View Extension Points

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