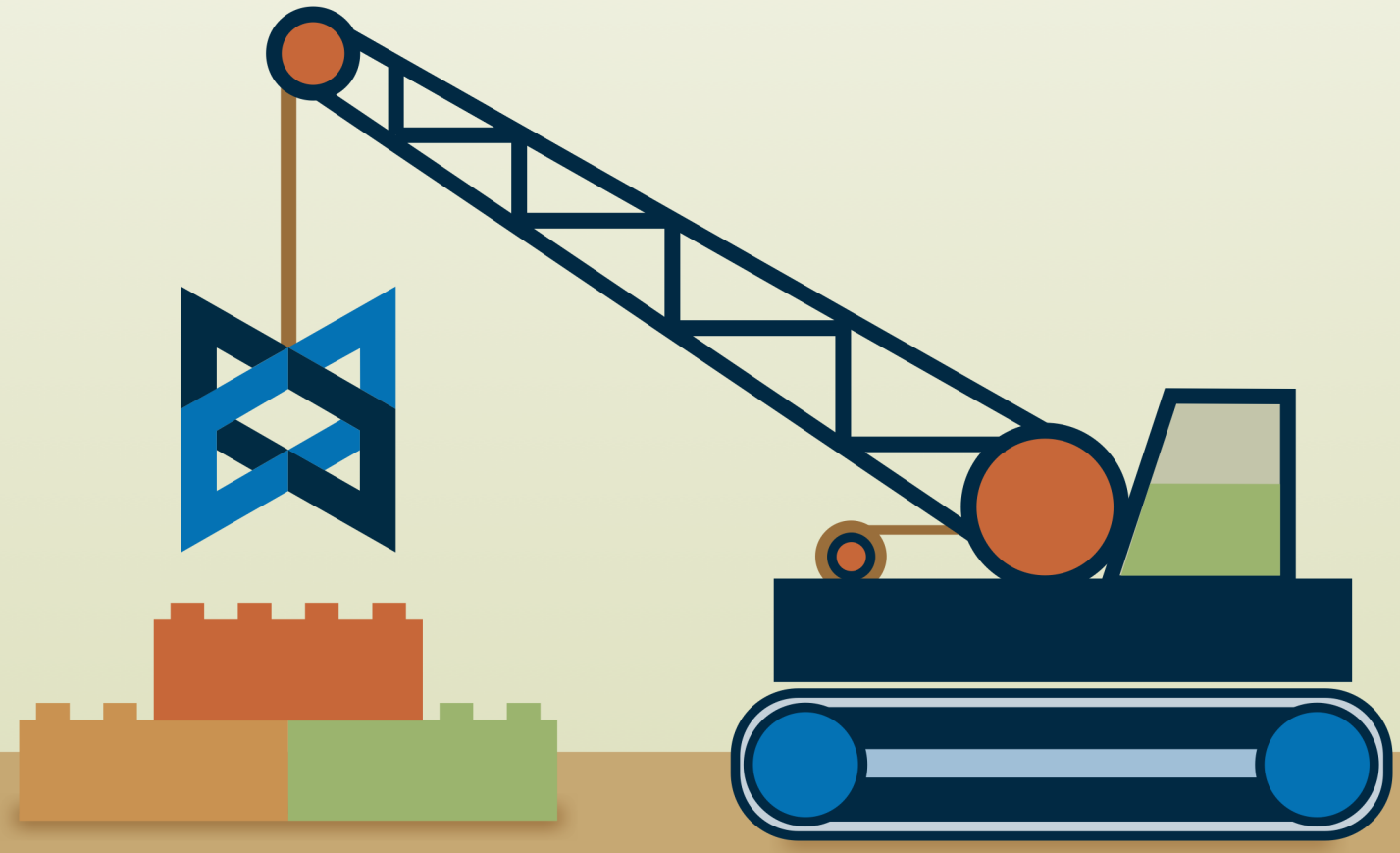
Covers Backbone.js v1.0

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

©2013 - 2014 Muted Solutions, LLC. All Rights Reserved. Backbone.js and the Backbone.js logo are Copyright 2010-2013 Jeremy Ashkenas and DocumentCloud Inc.

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 16: Application Workflow

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

A Poorly Constructed Workflow

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

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

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

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

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

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

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

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

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

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

Too Many Concerns

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

The first concern is the high level workflow:

- Enter employee info
- Select manager
- Create employee

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

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

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

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

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

Model Explicit Workflow

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

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

```
var orgChart = {

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

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

    });
  },

  // ...
}
```

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

```
var orgChart = {

  addNewEmployee: function(){
    var that = this;

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

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

    });
  },

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

```
19      form.render();
20      $("#wizard").html(form.el);
21      return form;
22    },
23
24    selectManager: function(employee){
25      var form = new SelectManagerForm({
26        model: employee
27      });
28      form.render();
29      $("#wizard").html(form.el);
30      return form;
31    }
32  }
33
34
35  // implementation details for EmployeeDetailForm go here
36
37  // implementation details for SelectManagerForm go here
38
39  // implementation details for Employee model go here
```

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

The Challenge Of Workflow Objects

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

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

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

Lessons Learned

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

Workflow Should Be Understandable At A Glance

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

Learn To Recognize Concerns, So They Can Be Separated

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

The Dependency Inversion Principle

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

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