Chapter 6. Collection View

6.1. Introduction

The Collection View is a pattern that describes how to render views within views, specifically with a Collection holding many Models which is a common occurrence in a Backbone application. However, this pattern can easily be applied to any situation where you have a view that needs to render a dynamic number of sub-views.

6.2. The Problem

In server-side applications, it is common to see routes that represent many items of the same type. For example, the `/appointments` route might display HTML for all of the appointments in the system.

Typically, this server-side code gathers up all `appointments` via a database query. It then iterates over each record, rendering them as HTML in a template. This is all well and good when the following conditions are true:

1. The server-side template library can handle iteration (or arbitrary code)

2. The generated page is not interactive

Unfortunately, neither of these conditions hold for a modern client-side web application. Additionally, we encounter other obstacles:

1. Maintaining client-side templates quickly grows disorganized and confusing when they are filled with logic and iteration

2. A lot of interactive code is concentrated into a few "master" views instead of spread throughout the models

The first point is immediately apparent for anyone who has worked on a large client-side application (otherwise, take our word for it!). The second point is more subtle and will creep into your application over time.

Consider our `appointments` application, which might consist of:

- `Models.Appointments`
- `Collections.Appointments`
- `Views.Appointments`
- `Templates.Appointments`

Think about what `Templates.Appointments` would look like. One of the first lines will be the beginning of an iteration over individual `appointments`. **The majority of this view will be concerned with rendering an individual `appointment`**. This should immediately be an indicator that `Templates.Appointments` is not doing what it was designed to do. A template for rendering multiple appointments should only be concerned with concepts like lists and ordering, not with the process of rendering individual items.

Additionally, if we have a master `AppointmentsView`, its event bindings will be on the *list of appointments* not on the *individual appointments*. This will be much harder to implement naturally using Backbone's event binding.

Furthermore, if an event is triggered signaling that an individual `appointment` has changed and must be re-rendered, we need to re-render the entire list of `appointments`. This is not only expensive, but can jar the user's view by breaking their scrolling position (if the list is long and they are in the middle). It also means that a single view is listening to events triggered by many models, which is another code smell.

In well designed server-side applications, `Templates.Appointments` will simply loop over the `appointments` and immediately render a `Templates.Appointment` template for each one, thus delegating that work onto another class. This is what we want to do in Backbone. The difference is that, in Backbone, it is much simpler and more natural to have views call subviews, instead of having templates call subtemplates.

6.3. The Solution

First, we need a new application structure:

- `Models.Appointments`

- `Collections.Appointments`
- `Views.Appointments`
- `Templates.Appointments`
- **`Views.Appointment`**
- **`Templates.Appointment`**

We have added a second view and second template to handle individual appointments. Let's take a look at what the top level `Templates.Appointments` and `Views.Appointments` might look like:

```
Templates.Appointments = _.template(
  "<h2>Here is a list of Appointments</h2>"
);
```

```
Views.Appointments = Backbone.View.extend({
  template: Templates.Appointments,

  initialize: function(options) {
    _.bindAll(this, 'render', 'addAll', 'addOne');
    this.collection.bind('add', this.addOne);
  },

  render: function() {
    $(this.el).html(this.template());
    this.addAll();
    return this;
  },

  addAll: function() {
    this.collection.each(this.addOne);
  },

  addOne: function(model) {
    view = new Views.Appointment({model: model});
    view.render();
    $(this.el).append(view.el);
    model.bind('remove', view.remove);
  }
});
```

```
$(function() {
  // Create a collection
  var appointments = new Collections.Appointments([
    {title: 'Doctor Appointment', date: '2011-01-04'},
    {title: 'Birthday Party', date: '2011-01-07'},
    {title: 'Book Club', date: '2011-01-14'}
  ]);

  // Create our top level view attached to the dom
  new Views.Appointments({
    collection: appointments, el: $('#appointments')
  }).render();
});
```

Here is what `Views.Appointments` is responsible for:

1. Rendering its own template (the template data not relevant to individual appointments)
2. Iterating over `Collections.Appointments`
3. Creating new `Views.Appointment` when a new `appointment` is added to the collection and appending that view's DOM element to its own
4. Asking the view to remove itself when the model is removed from the collection

More importantly, note what `Views.Appointments` is *not* responsible for:

1. Rendering individual `appointments`
2. Listening to events on individual `appointments`
3. Updating the individual `appointment` view
4. Removing the view when a model is destroyed

Now that we have that sorted out, let's look at `Templates.Appointment` and `Views.Appointment`:

```
Templates.Appointment = _.template(
  "<div class='title'>{{ title }}</div>" +
  "<div class='date'>{{ date }}</div>"
```

```
);
```

Warning

Try to keep javascript code out of templates. It is a good habit to pass a JSON-style object to a template, not pass a full model to a template. The key point here is to pass key value pairs of JSON primitives like integers and strings, and not expect functions to be available. Avoid iteration by doing the iteration in the view and creating subviews. Conditionals are subjective, if they are short it is OK, but as they grow, consider subtemplates or subviews.

```
Views.Appointment = Backbone.View.extend({
  template: Templates.Appointment,

  initialize: function(options) {
    _.bindAll(this, 'render', 'remove');
    this.model.bind('change', this.render);
    this.model.bind('destroy', this.remove);
  },

  render: function() {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  },

  remove: function() {
    $(this.el).remove();
  }
})
```

Here is what `Views.Appointment` is responsible for:

1. Rendering an individual `appointment`
2. Updating the view when the `appointment` changes
3. Removing the view when the `appointment` is destroyed

An interesting distinction between the `destroy` and `remove` events can be observed here. Both are causing the same effect in the view and in the DOM, but they are very different events!

The `destroy` event occurs when the model is deleted from the persistence system (the server, or client storage, *etc.*). For example, we could have a button on our view called "Delete". Or, **more importantly**, there could be a button on an entirely different part of our application that deletes models.

Consider a side-panel that has a button called "Remove all read appointments" that only removes `appointment` models if they have the `read` attribute set. We could easily say, "when the delete button is clicked, remove this element". If we do that, we would have to do that for every instance that a model is deleted in some way *and* would need to hook it up to every view that displays that model. The power of events in Backbone is that, by binding to relevant events, we can avoid this duplication.

We also need to be aware of `remove` actions, because we may be maintaining multiple collections with the same set of models in them. Consider if we had all our appointments in a global `MyApplication.Appointments`, but then we created two sub-collections: `MyApplication.ReadAppointments` and `MyApplication.UnreadAppointments`. Any time a `Model.Appointment` was marked as `read` or `unread`, we move it from one collection to the other. In memory, those are the same `Model.Appointment` in the top-level `MyApplication.Appointments` and in the sub-collections.

If we had a `Views.Appointments` for each of the sub-collections, we need to remove the view elements on a `remove` event, but the model is not deleted, just removed from the collection. Since this is a Collection View we are representing the state of the collection, and must modify the view to mirror the collection's state.

Tip

Always try to use the most appropriate event when binding to an action. Don't see the right event? We will cover firing and listening to custom events in a later chapter.

6.4. Conclusion

The Single Responsibility Principle is just as important in client-side Javascript as it is in server-side code. It is an indication of good, object-oriented design that each entity is responsible for a single task. The Collection View divides up the tasks of iteration, interactivity, and output into separate objects, each with their own simple goals.