Composite View

Previously, we've seen a lightweight way of managing subviews for a collection. For a simple index, this approach is adequate; in a more complex view, however, we will need something more sophisticated. Behold: the App Academy CompositeView! Take a look at the code; what follows is simply an explanation of it.

The Idea

The key underlying component of our CompositeView class will be an instance variable, `this._subviews`, which is a plain old JavaScript object. The keys will be CSS selectors (strings) corresponding to where we want to render the subviews onto the page, and the values will be arrays of instances of subviews. Something like this:

```
{
  '.friends': [aUserIndexItemView, anotherUserIndexItemView, ...],
  '.fav-photos': [aPhotoItemView, anotherPhotoItemView, ...],
  '.new-photo-form': [aPhotoFormView]
}
```

CompositeView Inheritance

We're going to write a `Backbone.CompositeView` class that inherits from `Backbone.View` to house our general logic. Then, when we want to use our CompositeView class, we can just `extend` this class in the same way we normally `extend` the base View class. Backbone `extend` is a marvelous method: by writing

```
// app/assets/javascripts/utils/composite_view.js
Backbone.CompositeView = Backbone.View.extend({
  // ... CompositeView methods here ...
});
```

we can call

```
// app/assets/javascripts/views/awesome_things/awesome_show_view.js
MyApp.Views.AwesomeShowView = Backbone.CompositeView.extend({
  // ... AwesomeShowView methods here ...
});
```

to set up a composite view just like we have been doing for simple views.

Adding Subviews

To add a subview, we'll need two things: the view instance itself and something to tell us where on the page that subview belongs. Since we're using jQuery for DOM manipulation, we can use simple CSS selector strings to keep track of where we want the views to go.

Lazy-initializing `this._subviews`

We could create the `_subviews` object in an initialize function, but then we'd have to call that initialize function whenever we use the class. It will be more convenient to simply have a reader method that lazy-initializes the `_subviews` object. This method will create a `_subviews` object if one doesn't already exist and store it as an instance variable.

```
// composite_view.js
subviews: function () {
  this._subviews = this._subviews || {};

  // ... more to come
}
```

Getting subviews

Sometimes we will want to get all of the selectors and subviews. Other times, we only want to get the subviews for a specific selector/part of the page. Let's have our getter method take an optional argument and return only the subviews for the given selector if an argument is present, otherwise return the whole `_subviews` object.

```
subviews: function (selector) {
  this._subviews = this._subviews || {};

  if (selector) {
    this._subviews[selector] = this._subviews[selector] || [];
    return this._subviews[selector];
  } else {
    return this._subviews;
  }
}
```

Note that in the selector-specific branch we are lazy-initializing an array to store the subviews at that selector, if we don't already have one. This lets us be fearless about asking for subviews in our other code.

Adding a subview

Great! Now we have a data structure to store our subviews and a method to access them. We also need to be able to add subviews and render them onto the page. This will involve three steps:

1. Store the subview instance in the composite view's `_subviews` object
2. call the subview's `render` method, so that its `$el` actually contains some content
3. Attach the subview's `$el` to the page, i.e. insert its `$el` into the DOM

This is not difficult. The steps above correspond, in order, to the three lines of the following method:

```
addSubview: function (selector, view) {
  this.subviews(selector).push(view);
  view.render();
  this.attachSubview(selector, view); // we'll look at #attachSubview next
}
```

Or, because `#render` should return the view upon which it is called, we can combine the last two lines:

```
addSubview: function (selector, view) {
  this.subviews(selector).push(view);
  this.attachSubview(selector, view.render());
}
```

Putting an individual subview on the page is simple: append the subview's `$el` to the DOM element indicated by the corresponding CSS selector string. We'll also use `delegateEvents` to ensure that event listeners are bound, in case the subview's `$el` has previously been removed from the DOM (which destroys the associated DOM event listeners). Finally, we'll check to see whether the child being attached is also a composite view (by checking if `attachSubviews` is defined on it) and recursively attach the child's subviews, if necessary.

```
attachSubview: function (selector, subview) {
  this.$(selector).append(subview.$el);
  subview.delegateEvents();

  if (subview.attachSubviews) {
    subview.attachSubviews();
  }
}
```

We're using Backbone's View#$ method here. `view.$(selector)` is just shorthand for `view.$el.find(selector)`.

Attaching all subviews to the page

To attach all the subviews (i.e. insert each of their `$el`s into the appropriate place in the DOM), we'll need to iterate over the CSS selectors in our subviews object, then over the array of subviews at each selector, using jQuery to put the views onto the page. Vanilla Javascript can do this, but we'll use the Underscore library for convenience. (Note that Underscore's `each` method, when called on an object, passes `value, key` pairs to the callback, rather than `key, value` pairs as you might expect.)

```
attachSubviews: function () {
  var view = this;
  _(this.subviews()).each(function (selectorSubviews, selector) {
    view.$(selector).empty();
    _(selectorSubviews).each(function (subview) {
      view.attachSubview(selector, subview);
    });
  });
}
```

All that we are doing here is inserting the subviews' `$el`s into the DOM. Unlike when we initially add a subview to the composite view, we are not calling `render` on the subviews. After the initial `render`ing of the subview when it is added to the composite view, we will trust that it's `$el` contains appropriate content from there on out. We will delegate the responsibility for updating the content of each `$el`, as needed, to the individual views (via their `listenTo` and DOM event handlers).

Removing Subviews

Now we can add subviews to the view and the page, but we also need to be able to remove them. There are two cases to consider:

1. Removing a specific subview, such as would be useful when the user destroys a particular model that is being displayed on the page. In this case, we need to find the subview in the `_subviews` object and remove it from both the page and the object (so that it won't reappear on the next call to `attachSubviews`).

2. Removing the whole composite view, which we will want to do when navigating to an entirely different view via the Backbone router, for example. Right now, when we call `remove` on our parent view, the children can remain in memory as zombie views. We can fix this by overriding the `remove` method. Note that, by overriding a piece of

the standard API, we can be indifferent to whether a particular view is simple or composite: the same method will take both views off the page, recursively removing subviews in the case of a composite view.

First, removing a particular subview:

```
removeSubview: function (selector, subview) {
  subview.remove();

  var subviews = this.subviews(selector);
  subviews.splice(subviews.indexOf(subview), 1);
}
```

Then, removing the parent with all its children:

```
remove: function () {
  Backbone.View.prototype.remove.call(this);
  _(this.subviews()).each(function (subviews) {
    _(subviews).each(function (subview) {
      subview.remove();
    });
  });
}
```

We use `Backbone.View.prototype.remove.call(this)` to invoke the original Backbone View#remove on the composite view. (If we had simply said `this.remove()`, then our new version of the `remove` function would call itself in an infinite recursive loop.) We then call `remove` on each subview as well.

Note that, when we remove the parent view, we don't need to remove views from the `_subviews` object, as the parent will be discarded when we're done.

Using CompositeView

When we `extend` our `CompositeView` class, we will have a nice API for working with numerous subviews associated with multiple areas on the page. The main thing we will need to do is remember to attach our subviews whenever we render; my boilerplate render function looks like this:

```
// a_cool_composite_view.js
render: function () {
  var content = this.template();
  this.$el.html(content);
  this.attachSubviews();
  return this;
}
```

This `render` method relies upon our subviews having already been "registered" with the composite view via the `addSubview` method. Commonly, this will be done in the composite view's `initialize` method and/or a method that is triggered when a new model is added to a collection.