



Seperating the pattern into a base.js file

We will refer to nested views inside a layout as **regions**. Each layout may have one or more dynamically rendered regions. Let's extract this pattern into a separate class inside the *base.js* and call it Layout:

```
Organizer.Layout = Backbone.View.extend({
  render: function() {
    var template = Handlebars.compile($(this.template).html());

    this.$el.html(template());

    return this;
  }
});
```

Every layout has its own template to render.

We'll provide regions as an object and assign it to the regions property. Each region will have a name and a selector. We'll store all this info inside the layout:

```
render: function() {
  var that = this;

  var template = Handlebars.compile($(this.template).html());

  this.$el.html(template());

  _.each(this.regions, function(selector, name) {
    that[name] = that.$(selector);
  });

  return this;
}
```

Having this in place we can refactor our EventsLayoutView:

```
Organizer.EventsLayoutView = Organizer.Layout.extend({
```

Declare template and regions property for it:

```
template: '#index-template',
regions: {
  eventsList: '#events-list',
  newEvent: '#new-event'
},
```

Now we need some place to instantiate and render our views. Let's introduce the ready callback that will be fired when the layout is actually ready:

```
ready: function() {
  var eventsListView = new Organizer.EventsListView({
    collection: this.collection
  });
  var newEventView = new Organizer.NewEventView();
},
```

As you see I've copied some code from the render function that you may now remove completely.

Testing the callback for the layout

Make sure ready callback is being fired:

```
render: function() {
  var that = this;

  var template = Handlebars.compile($(this.template).html());

  this.$el.html(template());

  _.each(this.regions, function(selector, name) {
    that[name] = that.$(selector);
  });

  if (this.ready) this.ready();

  return this;
}
```

And now finish this callback up by rendering views into the desired regions:

```

ready: function() {
  var eventsListView = new Organizer.EventsListView({
    collection: this.collection
  });
  var newEventView = new Organizer.NewEventView();

  this.eventList.append(eventsListView.render().el);
  this.newEvent.append(newEventView.render().el);
}

```

Rename the template for the layout:

```
<script id="events-layout-template" type="text/x-handlebars-template">
```

```
template: '#events-layout-template',
```

Creating a ShowLayout Template

We can now apply the same pattern to the show view. Here is the layout:

```

Organizer.ShowEventLayoutView = Organizer.Layout.extend({
  initialize: function() {
    this.render();
  },
  template: '#show-event-layout-template',
  regions: {
    event: '#event'
  },
  ready: function() {
    var eventView = new Organizer.ShowEventView({
      model: this.model
    });

    this.event.append(eventView.render().el);
  }
});

```

The template:

```

<script id="show-event-layout-template" type="text/x-handlebars-template">
  <div id="event"></div>
</script>

```

Re-write the handler function for the route:

```
showEvent: function(id) {  
  Organizer.events.fetch();  
  
  new Organizer.ShowEventLayoutView({  
    model: Organizer.events.localStorage.find({id: id}),  
    el: '#show'  
  });  
}
```

I've renamed the show-event to just show:

```
<div id="app" class="container">  
  <div id="index"></div>  
  
  <div id="show"></div>  
</div>
```

Remove the render function from the ShowEventView.

Summary of Layouts

The idea of layouts and regions is pretty common and as you see really useful. However, we still have one more step to do: remove all hard-coded elements from the app block and render the appropriate layouts dynamically. To achieve this we will have to introduce yet another layout, that will control all other layouts.