How I've Improved My Angular Apps by Banning ng-controller

Posted on Friday Oct 24, 2014 by Tero Parviainen; original

The "Angular 2.0 Core" talk by Igor Minar and Tobias Bosch was an ng-europe highlight for me. What they basically did was announce a mass slaughter of Angular 1.x features and API cruft, worthy of a Game of Thrones wedding.

One of my favorite moments was when they announced the removal of controllers. This is because I've recently come to a realization that using standalone controllers, as with ng-controller, rarely leads to an optimal design.

Now, this is not a new idea. I've talked with several people who have said they don't use ng-controller at all in their apps, and just use directives instead. However, until recently, it didn't really click for me, and the idea never came up when I was first learning Angular. I hope this article will help communicate the approach.

This Isn't MVC

When I first started with Angular, the use case for controllers seemed straightforward: Angular is an MVC framework, and in such frameworks the role of controllers is well established. It is the arbitrator between models and views, and the first receiver of user actions.

But the more I got into it, the less clear this picture became.

For instance, Angular has *scopes*. Every controller is paired with a *separate* object - a scope - which is used to communicate with views. When you include scopes in the picture, controllers are no longer "the first receivers of user actions". Scopes are. When you think about it like this, the pattern becomes more like Model-View-ViewModel, with the Scope being the ViewModel, as Igor Minar has described.

Also, Angular has *directives*, and directives can in many cases be used interchangeably with controllers. Thus, the decision between writing a directive and just using ng-controller is not always obvious.

Early on, I preferred to use directives just to "extend the DOM" with generic features like tab bars and data grids, and leave all the application-specific behavior to plain controllers and ng-controller. Over time my thinking on this has shifted significantly, to the point where I now find little reason to ever use ng-controller.

So, a part of the Angular learning curve is coming to understand that controllers aren't quite what they seem, especially if you're coming from a straight-up MVC framework. In fact, using plain controllers in Angular may actually not be a good idea at all.

Problems With ng-controller

There are a few reasons why I think using ng-controller is problematic.

Inheritance

The ng-controller directive creates a controller, and gives it a scope that prototypally inherits from the parent scope. This prototypal inheritance tends to trip a lot of people up.

It is tricky to understand the effects of scope inheritance on your data. Every Angular newcomer runs into this at some point, and the "dot rule" is something everyone needs to learn.

Granted, this is just JavaScript's prototype inheritance in action, which is something every JavaScript developer should learn anyway. But the thing is, I don't think we really need it. There are better ways, like isolate scopes, to "pass data" along to your children. There's no need to add the cognitive overhead of inheritance chains.

It think it is a good idea to try to minimize scope inheritance in applications. Given that many of the built-in directives, like ng-repeat, use inheritance, you can't really remove inheritance *compleletely*. But you can remove the inheritance done by ng-controller, by not using ng-controller.

Semi-Global Data

While the mechanics of prototypal inheritance are something you can learn to work with, there's also a trickier problem that rampant scope inheritance causes. It has to do with the sharing of data and functions.

Deep scope hierarchies encourage you to share data and behaviour by just making it directly available on scope objects. This seems like a very convenient thing to do - it even seems what you're *supposed* to do - because it's so easy: Just put an attribute on a parent scope and all the nested ng-controllers and included templates gain access to it.

Unfortunately, while this is easy, it also makes your code more difficult to understand and maintain.

When you access some piece of data on a scope, how do you know where that data comes from? When you have even a couple of nested controllers, this isn't at all clear. The data might come from any controller that has access to some scope in the inheritance chain. You can easily lose track of where things are defined. (I should mention that the Angular 1.2+ "controller as" syntax does a lot to help with this particular problem, if you use it consistently.)

On the other hand, when you put some data on a scope, how do you know where that data is accessed? You don't, if there's inheritance involved. When you want to change the structure of that data, or even just rename an attribute, you'll have to find all the locations in nested controllers and templates that *might* access that data. And there are not many clues in the code for where they might be. You just have to remember them, or rely on a thorough test suite to find them, or go over all the code every time. There have been several instances when I've failed to do that, resulting in some hilarious bug hunting sessions.

Poor View Organization

Since you can annotate any DOM element with an ng-controller, you may have several ng-controller instances in one HTML template. This causes a common problem, which is that there's no one-to-one correspondence between HTML files and controller JavaScript files. You'll often end up asking yourself

where a particular controller might have been used in the HTML, and also where the markup that goes with a particular controller might be found.

This problem, like all of the problems I've described, can be overcome with a strict organizational discipline. It's just that the framework doesn't really guide you in the right direction. If you're like me, that means you'll often just get it wrong.

Solution: Component Pattern

To deal with these problems, I set out to do an experiment: Ban the use of ng-controller completely and see what happens. It turns out, this is probably the best thing that ever happened to the code in my current project. The code is now clearer, easier to change, and more organized.

So, what did I replace ng-controller *with*? I replaced it with directives. To be more precise, I replaced it with *isolate scope directives* that have their own *templates* and *controllers*.

This is a step in the direction of how Components work in AngularDart and how they are likely to work in Angular 2.0. It also results in an organization close to what you get when you use Web Components, so I've been calling this the Component Pattern.

Each component consists of a few key pieces:

1. An isolate scope definition

The scope definition *explicitly* defines what incoming data the component accepts. A component does *not* inherit its parent scope, but everything it needs should come in through the isolate.

2. A controller

Since the component is defined as a directive, its behavior could just be defined in its link function. But since I'd rather have the behavior be unit-testable without a dependency to the DOM, I'm pairing each component directive with a controller.

3. An HTML template

Components have a one-to-one correspondence with HTML templates. When you apply a component, what goes inside it is defined by the component's own template. Transclusion may be used to augment the component's contents, but I've yet to come across a use case for that.

Example

Here's a simple application with a list of people, and a little form for adding more people to the list:

index.html

```
<div ng-app="contestantApp">                                                ?
  <h1>Contestants</h1>
  <section ng-controller="ContestantsCtrl as ctrl">
    <ul>
```

```
      <li ng-repeat="contestant in ctrl.contestants">
        {{contestant.firstName}} {{contestant.lastName}}
      </li>
    </ul>
    <form ng-controller="ContestantEditorCtrl as editorCtrl">
      <h2>New Contestant</h2>
      <fieldset>
        <label>
          First name
          <input ng-model="editorCtrl.contestant.firstName">
        </label>
        <label>
          Last name
          <input ng-model="editorCtrl.contestant.lastName">
        </label>
        <button ng-click="editorCtrl.save()">Save</button>
      </fieldset>
    </form>
  </section>
</div>
```

The app has two controllers: One, called `ContestantsCtrl`, whose job is to provide the list of contestants to display, and one, called `ContestantEditorCtrl`, whose job is to deal with the form for adding new contestants.

The list controller does nothing but initialize the collection of people to show:

app.js

```
app.controller('ContestantsCtrl', function() {
  this.contestants = [
    {firstName: 'Rachel', lastName: 'Washington'},
    {firstName: 'Joshua', lastName: 'Foster'},
    {firstName: 'Samuel', lastName: 'Walker'},
    {firstName: 'Phyllis', lastName: 'Reynolds'}
  ];
});
```

The form controller adds a new contestant to the list when the save button is clicked, and then reinitializes the form:

app.js

```
app.controller('ContestantEditorCtrl', function($scope) {
  this.contestant = {};
  this.save = function() {
    $scope.ctrl.contestants.push(this.contestant);
    this.contestant = {};
  };
```

```
});
```

While this app is tiny, it already exemplifies some of the problems that the use of ng-controller may cause:

- Both of the controllers are used in the same HTML template. Looking at the controller JavaScript code, it isn't straightforward to deduce where in the HTML it may have been used. In a large application this can be a real problem.
- The inner `ContestantEditorCtrl` mutates the `contestants` array introduced by the parent controller. It relies on this global state that "just comes from somewhere." It isn't at all clear where.

How might one "componentize" this application? Well, first of all, let's think of the editor form. Instead of just defining it in our index template, we could refer to a specialized component. Any data it needs will be *explicitly given to it*:

index.html

```
<div ng-app="contestantApp">
  <h1>Contestants</h1>
  <section ng-controller="ContestantsCtrl as ctrl">
    <ul>
      <li ng-repeat="contestant in ctrl.contestants">
        {{contestant.firstName}} {{contestant.lastName}}
      </li>
    </ul>
    <myapp-contestant-editor-form contestants="ctrl.contestants">
    </myapp-contestant-editor-form>
  </section>
</div>
```

This component is defined by a directive, which we can define in its own module. The module also contains the controller for the component:

my_app_contestant_editor.js

```
angular.module('myAppContestantEditor', [])
  .directive('myAppContestantEditorForm', function() {
    return {
      scope: {
        contestants: '='
      },
      templateUrl: 'my_app_contestant_editor.html',
      replace: true,
      controller: 'ContestantEditorFormCtrl',
      controllerAs: 'ctrl'
    };
  })
  .controller('ContestantEditorFormCtrl', function($scope) {
    this.contestant = {};
```

```
    this.save = function() {
      $scope.contestants.push(this.contestant);
      this.contestant = {};
    };
  });
```

The form markup that used to be in index.html is now in the component's own template:

my_app_contestant_editor.html

```
<form>
  <h2>New Contestant</h2>
  <fieldset>
    <label>
      First name
      <input ng-model="ctrl.contestant.firstName">
    </label>
    <label>
      Last name
      <input ng-model="ctrl.contestant.lastName">
    </label>
    <button ng-click="ctrl.save()">Save</button>
  </fieldset>
</form>
```

The logic is pretty much the same as before, with two major differences in how things are organized: The array to which the contestant is pushed no longer comes from an implicit parent scope. It is *explicitly* passed in to the component and there is no confusion about where it comes from. Also, the component has its own view template now, removing the confusion about where to find the markup: Just find the .html file matching the .js file.

We can go one step further and also make the outer "contestant list" a component of its own, leaving almost nothing in the main template:

index.html

```
<div ng-app="contestantApp">
  <h1>Contestants</h1>
  <my-app-contestant-list></my-app-contestant-list>
</div>
```

The structure of the list component is the same as for the form component. This component does not have any inputs, as is made clear in the empty isolate scope definition:

my_app_contestant_list.js

```
angular.module('myAppContestantList', [])
  .directive('myAppContestantList', function() {
    return {
      scope: {},
```

```
      templateUrl: 'my_app_contestant_list.html',
      replace: true,
      controller: 'ContestantListCtrl',
      controllerAs: 'ctrl'
    };
  })
  .controller('ContestantListCtrl', function() {
    this.contestants = [
      {firstName: 'Rachel', lastName: 'Washington'},
      {firstName: 'Joshua', lastName: 'Foster'},
      {firstName: 'Samuel', lastName: 'Walker'},
      {firstName: 'Phyllis', lastName: 'Reynolds'}
    ];
  });
```

This component's own view template now has the contents of the list, as well as the inclusion of the nested form component:

my_app_contestant_list.html

```
<section>
  <ul>
    <li ng-repeat="contestant in ctrl.contestants">
      {{contestant.firstName}} {{contestant.lastName}}
    </li>
  </ul>
  <c-contestant-editor-form contestants="ctrl.contestants">
  </c-contestant-editor-form>
</section>
```

Now, while I like the feel of this pattern, what I don't like so much is the fact that it results in more code. In particular, the directive definition object is quite verbose, with no fewer than five required attributes. Also, the controller, defined just a couple of rows beneath the directive, needs to be referred to by name, causing some unfortunate duplication. You *could* define the controller function right in the directive, but this would make it harder to unit test, which is not a trade-off I'm willing to make.

These issues are caused mostly by Angular's directive API, which Miško Hevery himself described as "convoluted" at ng-europe. There are probably ways to make the pattern a lot less verbose by introducing an Angular extension module that is streamlined for it. I might explore this in the near future. In any case, defining actual, official Angular Components will be a lot more streamlined in Angular 2.0.

There are Plunkers for the original version of the app, as well as the componentized version, if you want to explore the code further.

Conclusion

The component pattern, as described here, has clarified the design of my Angular apps a great deal. It is nice to hear Angular 2.0 moving in a direction where this style is preferred – just with a much better API.

Angular 2.0 Components will also be based on Web Components standards like the Shadow DOM, which opens up possibilities for further modularity. For example, each component may bundle its own CSS.

The pattern solves the problems of *prototypal inheritance* by just not doing it. The attribute shadowing issues that so often trip up beginners just aren't there in the same sense. With isolate scopes you use two-way data binding where, instead of *shadowing* the parent scope's property, you actually *reassign* it, matching most people's intuition of what should happen.

The pattern solves the problem of *semi-global state* by explicitly encoding what attributes a component takes from the parent scope, and how they are bound. Thanks to the isolation, there is no magical semi-global context on which to do ad hoc data access.

The pattern solves the problem of *poor view organization* by, as a rule, pairing each component with its own HTML template. It becomes obvious where the markup of a specific component is, and vice versa. A further development of the pattern might also bundle component-specific CSS, as Angular 2.0 Components are likely to do.

Contact: tero (at) teropa.info