PubSub Controller Communication

AngularJS - Communicating Between Controllers - Part 2

Richard Clayton

Unrepentant Thoughts on Software and Management.

In the last post, we discussed communicating between controllers using a controller's scope. This mechanism, however, is limited only to parent-child relationships between controllers (i.e. one controller is nested within another).

A more general way of communicating between controllers is to use Angular's `PubSub` mechanism.

PubSub in AngularJS

PubSub, or publish-subscribe, is a programming pattern commonly used to decouple components by allowing an *interested* component to subscribe to events published by another. Implementation of the PubSub pattern varies, but in many cases, it's common for a third party component to act as a

"broker" of communication between components sending messages and components receiving them. Angular's PubSub mechanism works in this fashion, where Angular itself serves as the broker.

To listen for an event in Angular, you simply register an event listener on a scope variable:

```
$scope.$on("event-name", function(eventContext, arg1, arg2, arg3){
    // React to the event.
});
```

The `eventContext`, which I typically abbreviate it to `e`, contains contextual information (scope emitted from and current scope handling event), as well as a method to stop the propagation of the event.

PubSub in Angular is bound very tightly to scopes. In fact, **the scope hierarchy and event name are the two factors used to determine how messages are delivered**. As we've already seen, event listeners are bound to a scope variable. Angular provides us two methods in which to send messages to those listeners: `$emit` and `$broadcast`.

The difference between `$emit` and `$broadcast` is how the event is propagated. Both methods will notify listeners registered on the same scope in which they are called. `$emit`, however, will also notify listeners on all parent, grandparent, and other direct ancestor scopes all the way back to the `$rootScope`. `$broadcast` does the exact opposite; the method notifies all child scopes of the event (these would be controllers nested within the controller broadcasting the event).

PubSub Example

Here's a really simple example of using PubSub. We will have two controllers talk to each other, but neither will exist in the other's scope hierarchy (I find this to be a safe assumption, and is quite often the case in Angular applications).

```
<div ng-controller="PingCtrl">
    <h3>Ping Controller</h3>
    <p>Received last pong at {{ time }}.</p>
    <button type="button" ng-click="ping()">Ping!</button>
</div>
<br />
<div ng-controller="PongCtrl">
    <h3>Pong Controller</h3>
    <p>Received last ping at {{ time }}.</p>
    <button type="button" ng-click="pong()">Pong!</button>
</div>
```

Now, with the `PingCtrl`, we will produce a message called `ping` and listen for messages called `pong`. In this case, we are going to listen for and `$broadcast` events on the `$rootScope`:

```
app.controller("PingCtrl", [ '$scope', '$rootScope',
function($scope, $rootScope){

    $scope.time = "(never)";

    $rootScope.$on("pong", function(e, time){
        $scope.time = time;
    });

    $scope.ping = function(){
        $rootScope.$broadcast("ping", new Date());
    };
}]);
```

We do the opposite with the `PongCtrl`, listening on its `$scope` and `$emit`ing events up the hierarchy (towards `$rootScope`):

```
app.controller("PongCtrl", [ '$scope', function($scope){

    $scope.time = "(never)";
```

```
    $scope.$on("ping", function(e, time){
        $scope.time = time;
    });

    $scope.pong = function(){
        $scope.$emit("pong", new Date());
    };
}]);
```

This is what it should look like when you run the example and click the **Ping!** and **Pong!** buttons.

PubSub Gotchas

While the PubSub mechanism looks pretty clean, there are a couple of things you should be aware of.

To `$emit` or to `$broadcast`, that is the question.

We've seen how easy it is to grab an instance of the `$rootScope` and broadcast a message to all controllers in the hierarchy. In general, this shouldn't be done. Not only is it a sign of poor conceptualization of your components, but it's also going to be a big performance hit if you are doing it for every message.

The answer is to think about the event you are trying to send and whom you are trying to communicate it to. If the event is supposed to be global (representing some major state transition or user action), then you may want to broadcast it from the `$rootScope`. However, if your intent is to have the parent controller notify it's children of some event, target your broadcast to the parent controller's `$scope`.

On the other hand, if you are designing a component where children are expected to help compose some complex piece of state (say a multi-section form or wizard), have the children `$emit` pertinent events to the parent from their own scopes. Remember, the parent can also, optionally, stop the propagation of the event from reaching higher scopes. This is a very nice solution if you are developing reusable form components since the child is unaware of the parent and the parent's binding to the child is through an intermediary (the PubSub mechanism). I use the same pattern when creating directives in Angular.

Where My Event?

A common error I see in Angular is when one controller attempts to communicate an event to a peer controller (i.e. not in same hierarchy) and the recipient (peer) never receives the event. Typically this happens because the sender uses `$scope.$emit` and the recipient is using `$scope.$on`. Note that the scopes are not the same object and they are not related to each other, thus `$emit` sends the event down to the `$rootScope` without ever firing the handler. The easiest way to fix this is to have the peer controller listen on `$rootScope` for the event. Alternatively, the producer could also `$broadcast` from the `$rootScope`.

Event Pollution.

PubSub is great, but a common consequence of adopting this pattern is that you will start to use it everywhere (which means you will have a ton of events and handlers). This isn't a performance problem so much as a maintenance and coherency issue. If you don't feel like you have this problem, don't worry about it and move on. If you start to hate the PubSub system

because you have hundreds of events (and even more handlers), consider consolidating events. For instance, if you have a movie player and you are publishing an event every time the video transitions to `playing`, `paused` or `stopped`, consider collapsing those events into a single `playerStateChanged` event and provide a `state` variable that indicates the new state. Your handlers become a tiny bit more complex since they have to determine what to do given the provided state, but you're now only writing one (vice three).

Beware of Consuming Your Own Events.

Two common issues in a PubSub architecture occur when a component consumes its own event. The first issue is the "reapplication of handler behavior". This is best shown rather than described:

```
var isPlaying = false;

function toggle(){
    isPlaying = !isPlaying;
}

$scope.$on("playerStateChanged", toggle);

$scope.playPauseButtonPressed = function(){
    toggle();
        $scope.$emit("playerStateChanged", { state:
(isPlaying)? "playing" : "paused" });
}
```

Since we are both watching and emitting `playerStateChanged`, we will receive our own events. If we immediately execute `toggle` but then also produce and consume the event, we will erroneously call toggle again.

This issue seems trivial, but it's quite common. The fix in this example is simple: remove the call to `toggle` in `$scope.playPauseButtonPressed`. The problem is, JavaScript is

single threaded and we don't know necessarily how long it will take to have our listener executed (assuming listeners are executed sequentially and we may have to wait for one or more to complete). A simple way to fix this is to add a property like `caller` and perform a check to see if that component had sent the message (ignoring if it did).

The other issue in *consuming your own events* is the potential for creating an infinite loop that locks up the browser. This occurs when you are being clever and you produce the event you are consuming from your event handler. It's not that this practice is discouraged necessarily (I think it's akin to recursion). You simply need to be careful how often the event you are consuming is republished. Since this is JavaScript, you only have one thread, which means there's going to be no opportunity to do something else until the thread of execution ceases.

Here's how the infinite loop is produced:

```
$scope.$on("tick", function(e){
    $scope.$emit("tick");
});
// Kick off the process.
$scope.$emit("tick");
```

Infinite loops can also be produced between two or more event handlers, even on separate controllers. A classic example is the `Ping-Pong` scenario:

```
$scope.$on("ping", function(e){
    $scope.$emit("pong");
});

$scope.$on("pong", function(e){
    $scope.$emit("ping");
});
```

```
// Kick off the process.
$scope.$emit("ping");
```

Benefits and Pitfalls of PubSub Communication

Benefits

1. Better decoupling of components (Angular is the intermediary).
2. One-to-many, many-to-many, and many-to-one notifications.

Pitfalls

1. More boilerplate code than other mechanisms (but not much more).
2. No guaranteed order in which handlers will receive an event.
3. Potential for abuse (loops, event pollution, etc.) which may lead to performance issues and a lack of clarity in the code base.

Code

You can find all the examples from this blog post here: https://github.com/rclayton/NG-Communicate-Ctrls/tree/master/pubsub

Richard Clayton

Unrepentant Thoughts on Software and Management.

July 01, 2014

Read Next: Parent/Child Controller Communication