Animation in Angular 2.0

matsko@google.com

Animation in Angular 2.0 consists of three levels that can be used to take control of what is happening:

- CSS Triggers (Keyframes, Transitions, Stagger Animations)

- JavaScript Animations (Custom Classes with Annotations)

- Timeline Animations (The full package)

Each stage builds on top of the former.

# Application States

Unlike Angular 1.x, 2.x will not bother with using CSS classes as the source of truth regarding state. Instead we will make use of component-level states which can then be collected for animation use. Take this component for example:

<spread-sheet-component>

...

</spread-sheet-component>

This spreadsheet takes control of the entire page and renders out to a 5x5 grid of input fields.

<spread-sheet-component>

<!-- row 1 -->

<input />

<input />

<input />

<input />

<input />

<!-- row 2 -->

<input />

<input />

<input />

<input />

<input />

</spread-sheet-component>

Now let’s imagine that a modal appears on screen and **each of the input fields become disabled**. Upon being disabled an animation could potentially run for each input field which would be tricky to hook into using CSS classes. But what about the spread-sheet container? It too could use a CSS class to trigger an animation.

This approach of using CSS classes to drive animations is problematic for the following reasons:

* Hooking into transitions and keyframes and trying to control them is a game of cat and mouse
* Too much code is required and it’s not worth the effort

Instead we should make use of an internal state system that **clearly defines state points** for a component:

@States({

disabled => disabled

enabled => enabled

});

class SpreadSheetComponent {

@State(‘disabled’)

get disabled => \_disabled;

get enabled => !\_disabled;

setAsDisabled() {

\_disabled = true;

}

}

(Also keep in mind that if the element contains a view container on the same element then it will expose the **enter, leave** and **swap** states.)

Now our component contains two states: enabled and disabled. What about if we wanted to have an animation latch itself onto it? There are three ways we can hook into it.

# CSS Triggers

CSS-based animations work close to the same was as in AngularJS. The main difference is that we will not bother with hooking into class-based animations (something for example that is powered by ngClass). Instead we will only hook into the states exposed on the component level.

When the states change on the component, it will automatically decorate the element with **ng-STATE** and **ng-STATE-ACTIVE**. This is the same behaviour say for **ng-enter** and **ng-enter-active** back from Angular 1.x.

So when the view container inserts the element we get:

<spread-sheet-component class=”ng-enter ng-enter-active”>

</spread-sheet-component>

Then we can hook into the animation much the same way as with 1.x.

Since the state values drive everything, we’re left with each element always having a state value (with the exception of enter/leave).

<!-- the animation is over so the CSS class is present -->  
<spread-sheet-component class=”ng-some-state”>

</spread-sheet-component>

With a change of state it is very easy to have a transition take over and make the state change happen.

<!-- .ng-some-state changed to .ng-another-state -->  
 <spread-sheet-component class=”ng-another-state”>

</spread-sheet-component>

Now that another state has changed, it is up to the user to provide a transition to kick things off. ngAnimate will take the coordinate values and place them on the element ready to be animated. This way if a transition is placed then we have automatic morphing between element states for free.

Since everything is state based, a parent animation can use a CSS selector to match child elements just as normal and apply a transition value.

ngAnimate in Angular 2.0 will be able to detect when multiple CSS-based animations are fired on multiple elements and capture groups of animation as a whole.

# JS Animations

JS Animations in Angular 1.x are very limited since they work off of callbacks and they’re limited to the language features that ES5 offers. They also allow for no criss-cross between events (say if you want to enter and then leave) and they also **do not allow for multiple elements to be animated in parallel or sequence with each other**.

If for example you wanted to catch a page change in Angular 1.x and animate the interior structural changes one by one then it would be close to impossible in 1.x unless you hacked the JS animation code.

Instead we should change our API to allow for the following:

* Introduce the ability to select one or more elements to animate in a single animation event
* Introduce the ability to funnel multiple child animations into a parent animation and make those inner animations run in parallel or sequence based on a promise-oriented system.
* Introduce the ability to animate elements that have nothing to do with animation state and are just apart of the DOM.
* Override or hijack other animations from happening.

So this is a lot of features to support. What will the API look like.

First things first we need to create an Animator class and reference that to our template via a new annotation system.

**@Animator({**

**selector: ‘.page-animator’**

**})**

class PageAnimator {

//...

}

Now let’s attach **page-animator** to our body tag on our app to drive the animations:

<body class=”**page-animator**”>

...

</body>

Nothing happens so far. We haven’t defined any animations. How will animations be captured? We will make use of the state-management system that we defined earlier in our code as a gateway to detect animations in our code:

Let’s say that our spreadsheet will now have a modal that appears on top of it. The **modal** and the **spreadsheet are unrelated components**, however the **parent container can be used as point of reference to what’s going on**.

So if for example we have a view that contains both then we have the component identify that the state is **modal-active.**

<body class=”page-animator”>

<some-view-component class=”**ng-modal-active**”>

<modal-component class=”**ng-active**”>

</modal-component>

<spread-sheet-component class=”**ng-disabled**”>

</spread-sheet-component>

</some-view-component>

</body>

There are three components with a state for each one. When the states change, this triggers **three animations** (one for each component). Let’s say if we wanted to hook into the state change that is triggered then we can do so by making an animation within our **PageAnimator** class and referencing the **ng-modal-active** state within an annotation.

@Animator({

selector: ‘.page-animator’

})

class PageAnimator {

@Animation({

**selector: ‘some-view-component.ng-modal-active’**

})

showModalAnimation(element) {

var animation = animateAndReturnPromise(element);

}

}

This animation will run on the parent component first and it **will disable animations from being looked up on the children** since the parent animation takes priority. If it was not defined and matched then the child states (**modal-component.ng-active** and **spread-sheet-component.ng-disabled**) would attempt to animate through the JS animation code or via CSS lookup.

But what about if we wanted to figure out what children would also animate? But we still want to run the **showModalAnimation?** We can request more info within the animation annotation:

@Animator({

selector: ‘.page-animator’

})

class PageAnimator {

@Animation({

selector: ‘some-view-component.ng-modal-active’,

collect: **‘.ng-active, ng-disabled’**

})

showModalAnimation(element, matches) {

var animation = animateAndReturnPromise(element);

}

}

We can also just find all of the active state changes:

@Animation({

selector: ‘some-view-component.ng-modal-active’,

collect: **‘\*’**

})

Or just select a combination of all the elements mixed in with the state-changed elements with their animations:

@Animation({

selector: ‘some-view-component.ng-modal-active’,

find: **‘\*’**

})

Now we get all this data in as a second param within the animation:

showModalAnimation(element, matches) {

console.log(matches);

}

Matches contains a list of the elements with their associated animation and coordinate details:

matches = [

// a child element that contains a state change

[ element, coordinates, state , originalAnimationFn ],

// a child element that does not have a state change but was picked up by the find selector (if that was used)

[ element, coordinates, null, null ]

]

Now if we wanted to we could fully control what animations happen and when by making a promise chain within the animation function.

showModalAnimation(element, matches) {

someJQueryAnimation(element).then(function() {

var a = someJQueryAnimation(matches[0][0]);

var b = someJQueryAnimation(matches[1][0]);

return Promise.all([a,b]);

});

}

The magic happens within the annotations.

# Timeline Animations

Timeline animations work on a level higher up from that of the JS animations and are triggered to run based off a timeline script described from HTML. They way that they’re triggered is based off of state.

<ng-timeline selector=”.page-animator”>

<ng-timeline selector=”.some-view-component” state=”ng-modal-open”>

....

</ng-timeline>

</ng-timeline>

Timelines allow for a more complex mix of data, styling and sequencing.