Component States in Angular 2

What if components in Angular 2 were wired always have an internal state which allowed for other parts of the framework to do preparatory UI-related work so that when that state changes Angular can attempt to take on as much of the animation as possible.

# Setting the foundation for animations

Animations in Angular 2.0 are not here yet, however, state changes between a component or multiple state changes among multiple components would be a nice platform for Animations to be triggered on.

## Basic State Change

Let’s imagine that we have a component called <zippy> which is designed to open and close. By definition there are two states and these two states are states that the component manages itself within its definition:

@Component({

selector: ‘zippy’

})

class Zippy {

open: Boolean

}

Triggering animations, however, is non-existent since Angular doesn’t know that the open/close boolean value dictates the state of the component. Therefore we will place a value within the annotation to reflect that:

@Component({

selector: ‘zippy’

state: ‘open’ // the `open` boolean member value is watched

})

Now when the open state changes it will impose that change on to its component container. Now it’s when the state change occurs that the animation will take off. Let’s imagine that our CSS transition code below will match the state change for the zippy component:

/\* remember that you can also place a class on the element and

target that one specifically \*/

zippy.from-closed.to-open {

transition: 0.5s linear all;

}

Notice how there is no actual CSS styling code in here to tell the transition what to do? All that we have here is a transition value which will then activate the state transition for us. Now what about the actual style changes that occur for the zippy for its open and closed states? Since those states have nothing to do with animation, and are a required piece to exist for when the zippy is open or closed, the CSS styles for the zippy are defined within the component container.

zippy.ng-open {

max-height:500px;

}

zippy.ng-closed {

max-height:500px;

}

Since we provided a transition for the state change then that’s all required for the component to do its thing.

## Implicit States - Enter / Leave / Move

When a component is inserted into the DOM then that is a state change from **”detached”** to **whatever state the component will be or a default state called “ng-default”**. When a component is removed then it will be moved from **its current state to “detached”.**

So if we set our zippy to be open by default when it is placed in the DOM then the state should go from **detached** to **open**.

<zippy \*ng-if=”yes” [open]=”yes”></zippy>

zippy.ng-detached {

opacity:0;

}

zippy.ng-open {

max-height:500px;

}

zippy.from-detached.to-open {

transition:0.5s linear all;

}

Now when the item is inserted then it will perform this animation automatically.

# Group-based State-Change Sequencing

State changes become powerful when there are a multitude of state changes that occur between a collection of elements. Let’s say for example we had the following HTML code for our view on our page:

<!-- home.html →

<h2>Home Page</h2>

<div \*ng-if=”showUsers” class=”users-container”>

<div \*ng-for=”#user of users” class=”user”>

{{ user.name }}

</div>

</div>

Now what happens if **showUsers** changes to true? The outer container is evaluated and inserted into the page and the inner container is also evaluated and its list of users is also inserted into the page. If no animations are detected then nothing happens, however, if animations are detected on both the **.users-container** and **.user** element containers then the parent container (in this case **.users-container**) will win and the inner elements will not animate. However if we use some special CSS code then we can perform an animation sequence state-change orchestration.

To get started with this let’s reiterate the presence of the state changes for both sets of elements. The outer container is being inserted and the inner containers as well. So we can start by defining an animation on the outer container:

.users-container.from-detached {

opacity:0;

}

.users-container.from-detached.to-default {

transition:0.5s linear all; /\* this element will fade in \*/

}

Now the inner **.user** elements are also being inserted from detached to any, but the **parent container may not know the exact details about the inner state changes.** So in order for it to pick up **any state change** among its children then we could issue a CSS selector to pick that up.

<!-- this will go from “detached” to it’s default state →

<!-- but any state change will also have a CSS class of “ng-state-change” -->

<div \*ng-for=”#user of users” class=”user”>

{{ user.name }}

</div>

Whenever any component performs a state change a temporary class of **“ng-state-change”** will be applied to the elements that are changing state. This allows for a CSS selector to query the inner, changing elements and issue some kind of delay or stagger orchestration.

But how do we do this in CSS?

## CSS Custom Styles/Properties

Custom CSS Styles/Properties are possible with a hack which makes use of the **CSS content property** by preparing a list of key/value properties, creating a JSON object, and pasting the contents of that JSON into that content property on the element.

body {

content: “{“custom”:”value”};

}

This works in all browsers other than IE, but if we duplicate the content tag and prepend it with a hyphen then it works in IE when accessed via the **element.currentStyle** hash object.

body {

content: “{“custom”:”value”};

-content: “{“custom”:”value”};

}

To make this less ugly we will use a build step in our Angular 2.0 build pipeline that will make custom properties convert themselves to this.

So this

body {

--custom: “value”;

}

Becomes this:

body {

--custom: “value”;

content: “{“custom”:”value”};

-content: “{“custom”:”value”};

}

Then ngAnimate can read these custom values directly via getComputedStyle and element.currentStyle.

## Custom Properties to Orchestrate Animations

Since we can use custom properties we can put together a unique property that tells ngAnimate to hold off from animations that occur on child elements.

We can have our parent container called **.users-container** tell the child animations to wait:

.users-container.from-detached.to-default {

/\* run all child animations for 0.5s and stagger by 0.1s \*/

--ng-animate-children: \*.ng-state-change 0.5s 0.1s;

}

Or issue a specific transition or keyframe to run

.users-container.ng-enter {

--ng-animate-children:

“\*.ng-state-change (.some-class 0.5s) 0.1s”;

--ng-animate-children:

“\*.ng-state-change (@keyframe 0.5s) 0.1s”;

--ng-animate-children:

“\*.ng-state-change (@keyframe 0.5s, .some-class 0.5s) 0.1s”;

}

The nice thing here is that the child animations still animate their own way, but the parent container/component can override and reorganize their behaviour.

The JS-driver equivalent can do the same thing.

class PageAnimator {

@Animate({

target: ‘.users-container’,

children: ‘\*.ng-state-change’

})

usersAnimation(target, children) {

// do whatever you want on the child elements

var promises;

children.forEach(function(child) {

promises.push(new Promise(resolve) {

setTimeout(function() {

child.animate(resolve);

}, i \* 100);

}));

});

return Promise.all(promises);

}

}

# Animating Non-Angular Elements

Since both CSS and JS sequencing use CSS-style selectors this means that we can sequence animations on elements that are not associated with Angular or ngAnimate.

Therefore something like this could be used to animate the parent element and the child animations with a single selector.

.if-container.from-detached.to-default {

--ng-animate: \* (.fade 0.5s) serial;

}

<div \*ng-if=”exp”>

<p>...</p>

<p>...</p>

<div [css.hide]=”exp”>...</div>

<p>...</p>

<p>...</p>

</div>

Everything in here now animates **one by one**.

# Integration with the Router

Since each container contains a state, the router will automatically issue states based on the **alias value** defined in each route.

@RouteConfig({

‘/’ : { as: ‘home’, component: HomeCmp },

‘/about’ : { as: ‘about’, component: AboutCmp },

})

Now when the route changes, the component container will automatically include the state change:

app-component.from-home-route.to-about-route {

--ng-animate: “\*.to-detached (.fade-out 0.5s) serial,

\*.from-detached (.fade-in 0.5s) serial”;

}

# Custom States from the template

Since state changes are issued from within the component, it gets difficult to apply your own state values from within the template without creating a component of your own. Say for example we had \*ng-if and we wanted to have our own state work with it:

<!-- the only states here are enter/leave -->

<div \*ng-if=”exp”> </div>

But if we use the ng-state attribute then it will work:

<!-- the only states here are enter/leave -->

<div \*ng-if=”exp” [ng-state]=”myCustomState”> </div>