ngAnimate in Angular 2.0

[matsko@google.com](mailto:matsko@google.com)

[rmesserle@google.com](mailto:rmesserle@google.com)

# Basic Idea of ngAnimate

The ngAnimate module makes CSS, JS and Timeline-based animations work with pre-existing and custom components/directives present in Angular.

# What this Document Proposes

* How to link animations between elements and CSS, JS, timelines
* Changes in detection and naming
* Improvements to animation reusability
* Animation plugins and default component animations
* Pre-detection and tooling support

# How Angular 1.x worked

Directives trigger events (like enter and leave) and ngAnimate will attempt to capture a CSS or JS animation based on what it detects. Detection for CSS usually involves using getComputedStyle which causes a reflow. Repeated reflows slow things down and getComputedStyle in general is very unreliable.

The major problems with ngAnimate in 1.x are:

* ngClass is overused by ngAnimate and causes some buggy behaviour with detection and multiple class computations
* Too many reflows due to requestAnimationFarme, getComputedStyle and document.body.clientWidth results in performance issues
* Too little pre-calculated information (reflows are used to direct ngAnimate to behave in a certain way) cause things to slow down
* Too much DOM traversal within $animate due to CSS class behaviour
* No real way to extend sequencing behaviour and to include child animations
* CSS transitions and keyframes are subject to browser inconsistencies and require too much code to create a levelled-out platform
* Tooling is non-existent
* No support for metadata
* No way to mix CSS, JS and timelines together
* The code is difficult to maintain and is too big overall

# The Angular 2.0 way

First things first, Angular 2.0 will not use addClass/removeClass (ngClass) to trigger animations. So something like this won’t work:

<style>

.my-component.fade-add { … }

.my-component.fade { … }

.my-component.fade-add-active { … }

</style>

<div class=”my-component” ng-class=”{fade:true}”>Fade In</div>

Dealing with CSS is complicated for when classes are added and removed. Should the follow-up animation cancel out the previous one? When do we merge things together? These questions are pointment and **there is no reason for the framework to have to make these decisions**. Instead we should just treat standard transitions, keyframe and class-based interactions outside of the realm of ngAnimate (let the browser handle it).

When it actually comes down to ngAnimate-powered animations, instead of relying on the global landscape of CSS (where everything is defined as floating CSS classes and selectors), Angular 2.0 will follow a mechanism of **attaching animations to components via an app-level manifest file.**

The file looks something like this

// animations.json

{

“\*.ng-enter” : “.fade”

“.some-page-page.ng-leave” : “{fadeJSAnimation}”

“.some-page-page” : “<fadeTimeline>”

“.another-class” : {

}

}

The biggest issue that this solves is that **all of the animations defined in the manifest file are handled by ngAnimate** and all other normal animations are **handled outside of ngAnimate.**

Now the animations.json file is attached to the AppCmp component via an annotation:

@Animations(“animations.json”)

class AppCmp {}

# CSS-Based Animations (transitions and keyframes)

The **.fade** CSS class is a regular CSS class that floats around in some stylesheet. In this case, when **any element enters (via \*.ng-enter),** ngAnimate will find the matching selector and perform an animation via the fade class. So if **.fade** contains a transition or keyframe then it will execute that. We can also fine the duration value directly in the JSON definition:

// animations.json

{

“\*.ng-enter” : “.fade(5s 1s)”

}

The timing for the animation is now defined within the JSON file. The fade animation will run for 5s and have a starting delay of 1s. **When ngAnimate starts the animation it will build out the custom duration/delay transition by itself with the functionality of $animateCss**. This means that a developer can create a stylesheet **with only CSS classes containing styles and zero animation code.** This cleans up some of the mess of vendor prefixing and browser nuances.

## Sequenced and Parallel CSS-based animations

Since everything is defined in a central JSON file, we can define a language for sequenced animations. Let’s say for example we wanted to run **fade then slide then rotate then explode:**

// animations.json

{

“my-component.ng-enter” : [

“.fade-prepare!”,

“.fade(5s 1s)”,

“.slide(1s)”,

“.rotate(10s)”,

“.explode(1000ms)”]

}

The array syntax does the trick. But what about if we wanted to run everything in parallel on that element?

// animations.json

{

“my-component.ng-enter” : “.fade(5s 1s) .slide(1s) .rotate(10s) .explode(1000ms)”

}

Then mixing both together is trivial

// animations.json

{

“my-component.ng-enter” :

[“.fade(5s 1s) .slide(1s)”,

“.rotate(10s) .explode(1000ms)”]

}

You can also just pass a bunch of CSS classes and have $animateCss do the detection like it did in 1.x:

// animations.json

{

“my-component.ng-enter” : {

“combine-properties()”: true,

“animation”:“.flip .rotate .spin .explode .fade-out”

}

}

This makes using other CSS libraries like animate.css trivial:

// animations.json

{

“my-component.ng-enter” : “.animated.rotateInOut”

}

We can also use keyframes directly

// animations.json

{

“my-component.ng-enter” : “@bounce(2s)”

}

# JavaScript-based Animations

The JSON manifest file also be able to trigger JavaScript animations. That we wanted to run a special pageChangeAnimation then we can do using this code:

// animations.json

{

“my-component.ng-enter, my-component.ng-leave” : “**{pageAnimations.change(10s 2s)}**

}

The JavaScript animation will then be executed as:

class PageAnimations {

static change(elements, metaData): Promise {

return somePromiseForTheAnimation;

}

}

The function will be executed with the params:

// for when both enter and leave are detected

change( { enter: enterElm, leave: leaveElm }, { duration : 10, delay : 2 })

// for when only enter is present

change( { enter: enterElm }, { duration : 10, delay : 2 })

We can also collect children that are also animating

// animations.json

{

“my-container”: {

“collect”: “.ng-enter, .ng-leave”,

“animation”: “PageAnimations.change(10s 2s)”

}

}

Since PageAnimations is a class associated with Angular’s DI, we need to attach that class to the injector of the app:

@Injectable()

class PageAnimations { … }

# Timeline Based Animations

Timeline animations are important for complex sequencing (which Material needs) and for platform specific data. Timelines are **defined in XML** and are used to choreograph animations which occur on multiple elements and on multiple levels:

// animations.json

{

“my-container”: “<super-timeline>”

}

The timeline XML file is referenced in the JSON file as well:

// animations.json

{

“super-timeline”: “super-timeline.xml”,

“.my-container”: “<super-timeline>”

}

The timeline will now take over the entire animation.

# Performance Enhancements

Using this JSON manifest approach, developers can fine-tune animations to provide hard coded duration values directly into the animation:

// animations.json

{

**// there is no need to call getComputedStyle or any reflow code here since we know the duration**

“.ng-enter” : “.slide(10s)”

**// this will call getComputedStyle behind the scenes**

“.ng-enter” : “.slide”

}

NgAnimate in 1.x performs various steps to “prepare” the element so that it can be animated using -add and -remove styling. This styling isn’t always necessary and slows things down. However if you want to have styles be applied beforehand then that is possible via changing classes:

// animations.json

{

**// still no need to run getComputedStyle**

“.ng-enter” : [“.ng-hide-add!”, “.ng-hide(10s)”]

}

Since we are using simple CSS classes, the Angular 2.0 building system can do extra work to attempt to fill in the duration values for us. So if we define **.fade** like so then:

.ng-hide-add { opacity:1; transition:1.5s; }  
.ng-hide { opacity:0; transition:1.5s; }

Then ngAnimate within the compilation step can go in and fill that into our compiled manifest file to speed things up. We can also perform more aggressive caching.

## CSS? Greensock? WebAnimations? Mobile?

Since the JSON manifest communicates with CSS directly without the need for transitions or keyframe animations this means that changing the underlying rendering mechanism can be done without any consequence to the user.

Libraries like Greensock and the Web Animations API allow for frame by frame control of the animation. Therefore if this rendering mechanism is used then the library can leverage those features and the user can dictate that those animations need those features. For example:

// animations.json

{

“.page-change”: {

“driver”: “web-animations”,

“animate”: [{ “.old-page” : “.fadeOut” },

{ “.new-page” : “.fadeIn” }]

}

}

Now the developer can expose a player API which can hook into the page change animation which allows to change the frame at an instantaneous point. We can also provide extra meta data to handle snap points:

// animations.json

{

“.page-change”: {

“driver”: “web-animations”,

“snapToEnd”: “60%”

}

}

Once again, if the driver supports this stuff then we should allow the developers to use it.

# Tooling

Since the JSON file exists and **all the Angular-specific animations are explicitly declared** this means that IDEs and tools can inspect the file and provide hints within CSS code to tell us that the animations belong to Angular.

Upon compilation, we can also inspect all the CSS files against the JSON manifest file to see if any classes, JS animations and/or timelines are not defined or are configured incorrectly.

# Plugins and Libraries

Much like the example of animate.css, animation styling can be defined within the realm of CSS code, but the animation and sequencing is defined within the JSON manifest. However what about pre-existing plugins that have their own animations? How could something like Material safely define their own animations and allow the user to change things around if they want to:

We can allow a plugin library like Material to create their own JSON file and attach that to their own library.

@Animations(“md-progress.json”)

class MdProgressLinear { }

Then the animations are parsed at normal and behave as normal within the application. If the user choses to disable all animations then they can do so within their own AppCmp

@Animations(false)

class AppCmp {}

But if the user choses to allow Material animations and to extend their behaviour then they can do so via their own top-level animations.json file:

// change the animation entirely

{ “md-progress.ng-enter” : “.fade .flip .rotate” }

// change the duration or delay

{ “md-progress.ng-enter” : { duration: 0.5, delay: 0.5 } }

// prepend or append the animation with other animations

{ “md-progress.ng-enter” : [“.flip”, “inherit” ] }

{ “md-progress.ng-enter” : [“inherit”, “.flip” ] }

# Testing

Since animation definitions and sequencing/registration are separated, we can have protractor run through **each defined animation within the JSON manifest and attempt to run the animation.** Upon running each animation it can detect to see if the animation actually triggers a style change or not. Since we know exactly what CSS classes, JS animations and timelines we are dealing with it makes things easier to detect when there are collisions and no animations that are triggered. Take this for example:

<style>

.blue { background: blue ! important; }

.red { background: red; transition:1s; }

</style>

<div class=”colourful-component blue”>I am blue</div>

//animations.json

{ “.colourful-component.ng-enter”: “.red” }

Within this circumstance the color will never change from blue to red since blue always wins. Normally this would be impossible to detect within ngAnimate 1.x, but in 2.x we can have a series of unit tests be build upon compile to find these bugs.

“.page-change”: {

“animation”: {

“50%” : “{jsFn}”,

“90%”: “.class”

}

}

<animation>

<animation-step …>