AngularDart: Animation

*Authors:* [*codelogic@google.com*](mailto:codelogic@google.com) *(Paul Rohde)*

# Contents

[Contents](#h.6ie3h9r2ss7d)

[Objective](#h.fxpk50cps4zs)

[Background](#h.wwgnimruvhlu)

[Prior Art](#h.v93ljdiqj2co)

[Detailed Design](#h.gijrsdjw51q5)

[Tentative Animate interface:](#h.hvdrfd6tb56c)

[Considerations](#h.huzy8ro8b0cp)

[Render Performance](#h.98a4g0r44kzu)

[AnimationRunner](#h.kc3m9r5149fd)

[Child Animation Canceling](#h.mlj6q5wazz8y)

[Detail: Animation](#h.iqcf5lperxcu)

[Caveats](#h.ydr456a07ivp)

[Security Considerations](#h.qjnbvlr7uej1)

[Performance Considerations / Test Strategy](#h.s30znup8hx9j)

[Work Breakdown](#h.59osdxvm5jwp)

# Objective

Build an animation framework into AngularDart similar to the system in AngularJS that takes advantage of Dart language features. Reconsider the framework and approaches taken in AngularJS and improvements that have been made to the web and animation approaches since ng-animate was initially conceived.

# Background

Animation significantly alters not only the visual transitions of elements, but significantly alters the lifecycle of dom elements. Its no longer possible to simply state “value is now false. Remove element ‘x’ from the dom.”. A naive approach will quickly cause the UI to become inconsistent as elements are created, destroyed, and have their state changed by the backing state model. Because of this, UI developers routinely avoid small visual effects for performance and because of the overhead of managing the additional state that has to be tracked and managed between an element and an object model while animations execute.

Any animation framework should do several things:

* Make it simple to define the intent of an action so that it can be animated (or not animated if animations are turned off for performance reasons).
* Optimal performance using best practices by default.
* Flexible to easily cover the most common use cases, and extensible to enable others to hook into the animation system.

# Prior Art

List of things which we think can be improved upon from AngularJS animations which we would like to fix in AngularDart:

* Future based completion
* Distinct dom mutate and dom read phases that do not occur at more than 60hz. (window.requestAnimationFrame)
* Opinionated Animation Loop with hooks that makes it easy to do efficient animations that read and mutate dom.
* Instant response < 2 browser paints.
* Better non-CSS

Things we like and want to keep from AngularJS.

* CSS backed animations
* Integration with core angular directives
* Throttling / limiting concurrent animations by parent
* Extensibility to hook in 3rd part animation frameworks or define custom animations.

# Detailed Design

There are several distinct patterns that are especially common in any form of document animation, most of which are already covered by angular and deal with the lifecycle of dom elements:

* ‘insert’ physically inserting a dom element (or set of dom elements) into the page.
* ‘remove’ physically detaching a dom element (or set of dom elements) from the page.
* ‘addClass’ apply an additional style or mode to an existing element.
* ‘removeClass’ remove a style or mode from an existing element.
* ‘setClass’ apply a CSS class directly
* ‘move’ in the case of lists of element shuffling or reordering constitutes a move operation.
* ‘play’ there will be times when the above behaviors are not able to fit into and encapsulate the desired effect since it may fall outside the bounds of a discrete time bounded operation, or may be gratuitous and have no lasting effect on the elements they are run on.

## Tentative Animate interface:

abstract class **NgAnimate** {

**AnimationHandle** addClass(Iterable<Element> elements, String cssClass);

**AnimationHandle** removeClass(Iterable<Element> elements, String cssClass);

**AnimationHandle** add(Iterable<Element> elements) ;

**AnimationHandle** remove(Iterable<Element> elements);

**AnimationHandle** move(Iterable<Element> elements);

**AnimationHandle** play(Iterable<**Animation>** animations);

}

abstract class **AnimationHandle** {

Future<AnimationResult> get onCompleted;

complete();

cancel();

}

class **AnimationResult** {

static const COMPLETED = const AnimationResult.\_('COMPLETED');

static const COMPLETED\_IGNORED = const AnimationResult.\_('COMPLETED\_IGNORED');

static const CANCELED = const AnimationResult.\_('CANCELED');

final String value;

const AnimationResult.\_(this.value);

}

abstract class **Animation** {

final Element element;

Future<**AnimationResult**> get onCompleted;

Animation(this.element);

attach() { }

start(DateTime time, num offsetMs) { }

bool update(DateTime time, num offsetMs) { }

read(DateTime time, num offsetMs) { }

detach(DateTime time, num offsetMs) { }

interruptAndCancel() { }

interruptAndComplete() { }

}

There’s a couple of things worth defining and pointing out:

**NgAnimate**

Animation service that defines the primary lifecycle for animating elements. Should be used by all dom manipulation that might be animated(able).

**Animation**

An animation instance is the state machine for an animation on a particular element. One animation per element, one element per animation.

Every method with (DateTime time, num offsetMs) will occur in an animation frame.

interruptAndCancel() and interruptAndComplete() can be called at any time to cancel or instantly complete the animation. If the animation is already completed or canceled the methods should do nothing. If the animation has not been completed, the onCompleted future should still be executed.

onCompleted should ALWAYS be executed.

**AnimationHandle**

This provides a completed handler and methods for forcibly completing or canceling an animation. handle.complete() is synonymous with calling interruptAndComplete() on the animation this handle represents, same with interruptAndCancel()

**AnimationResult**

This is an enumerated set of values that can be returned from the onCompleted future of an Animation.

There are several differences from the AngularJS version:

* ~~DOM addition and removal is~~ **~~no longer handled by the animation framework~~**~~. It should be handled in the normal digest loop in whatever class makes use of element construction. This is made possible by using window.animationFrameCallback instead of timer delay callbacks to group animation reads / writes. It also means that very little of the dom manipulation code in angular needs to be rewritten (This ensures the animation framework doesn’t need to know about blocks, block factories, scopes and lifecycles).~~
* add and remove are the primary verbs for altering the dom such that something is added or removed. For dom this is “add” and “remove”, for css classes this is “addClass” and “removeClass”. Each should map to an appropriate set of css styles:
  + addClass(items, ‘foo’); => ‘foo-add’ + ‘foo-add-active’
  + removeClass(items, ‘foo’); => ‘foo-remove’, ‘foo-remove-active’
  + insert(items); => ‘ng-insert’, ‘ng-insert-active’
  + remove(items); => ‘ng-remove’, ng-remove-active’
  + move(items); => ‘ng-move’, ‘ng-move-active’
* play(Animation animation) This would allow non-css animations to be created and run by implementing the animation interface.
* playClass(elements, ‘foo’); => ‘foo-play’, ‘foo-play-active’  
  This may be an additional method for effects that ‘occur’ but don’t wish to maintain or change state after execution. A gratuitous no-op.

## Considerations

AngularJS also attaches an ng-animate class onto elements that are currently running an animation. Unless there’s a good reason crops up, we will not be continuing this practice.

## Render Performance

[jankfree.org](http://jankfree.org/)

First, the browser is ultimately a rendering engine that has to run on a physical device that has a physical screen refresh and redraw rate that usually runs around 60hz. When running a frame by frame animation ‘update an element as fast as possible’ is a waste of processing time if the animation runs faster than the browsers refresh rate.

Second, performing operations on the dom invalidates sections of the dom, when rendered styles and layout have to be recomputed. In modern browsers this step is deferred as long as possible in order to prevent unneeded computations. Operations that need to know the size, position, and computed properties of dom nodes will cause these modifications to invalidate and recompute chunks of dom, and if the operation is significant enough, can cause frame skips and delays in code execution known as “jank”.

## AnimationRunner

[window.requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window.requestAnimationFrame) gives the browser control over the animation cycle through callbacks and can pause them when the page is not in view.

There needs to be distinct phases to any animation runner:

* DOM Manipulation
* DOM Read

This will cause the animation loop to be ordered somewhat oddly, so it’s important to understand why:

* **update(...)**Perform frame by frame updates on an element, for frame by frame animations this will consist of dom alterations, but should not consist of physical insertion, removal, or reparenting of elements.
* **detach(...)**  
  Attach and detach for CSS animations occur when a dom is altered, added, or removed.
* **read(...) / start(...)**Compute properties from dom mutations. start() should happen in the same frame as attach, but these are separate so that the mutate and read operations are grouped. read would only be used in an animation that requires frame by frame recomputation of properties.

Looking at the lifecycle of an animation:

* **attach();** // Mutate, preframe in the main digest loop.
* **start();** // Read, Frame n+0
* **update();** // Possibly Mutate, Frame n+1..n+m;
* **read();** // Possibly Read, Frame n+1..n+m;
* **update();** // Animation returns false n+m+1;
* **detach();** // Final mutate n+m+1

In the case that animations are disabled or canceled, the animation framework should execute the desired behavior immediately and return.

## Child Animation Canceling

Child animations are canceled by doing a parent walk and checking to see if any parent elements are currently active in the animation system. If one is active, the result of the animation is immediately completed.

**TODO(codelogic):** There needs to be a way we can turn animations on / off for a whole section of dom. For example, dumping a huge table into the page, we absolutely do not want to animate each element or set of elements in “ever”. How granular can we make this? Can we make it easy to turn off for a ‘particular’ operation? (eg. “this callback is going to eat the dom alive, turn off animations, execute, then re-enable after 1-2 frames for only this section of dom)

**TODO(codelogic): Rate limiting** - It would be awesome, and difficult, to rate limit the number of concurrent animations on the page. If there was some way to measure the fps of the page and turn the number of total concurrent animations up or down, it would be awesome. Downside would be mobile since you would like (if possible) for as clean an initial experience as possible.

## Detail: Animation

To further define the interface and interactions with an actual running animation, the following animation interface will be defined, and represents the internal state of a single animation running on a given element.

abstract class Animation {

final Element element;

Future<AnimationResult> get onCompleted;

Animation(this.element);

attach() { }

start(DateTime time, num offsetMs) { }

bool update(DateTime time, num offsetMs) { }

read(DateTime time, num offsetMs) { }

detach(DateTime time, num offsetMs) { }

interruptAndCancel() { }

interruptAndComplete() { }

}

This has the following effects:

* 1 Animation per Element
  + A second animation requested to run on an element with an already running animation will call the interruptAndCancel() method to forcibly cleanup and release the element.
* 1 Element per Animation  
  The animation represents the internal state machine for the animation of an element. In the case of CSS, even animating the same class on two or more elements could easily have different properties and durations making a case for multiple elements represented by the same animation difficult.

# Caveats

AngularJS has ‘staggered animations’ it may be difficult to implement this without adding a frame of latency for the initial animation since adding a class, calculating a computed value for delay becomes more and more expensive for each successive class.

Shadow Dom may make it difficult or nearly impossible to cancel child animations based on running parent animations depending on the ability to traverse the parent relationships of the dom.

Please add additional comments.

# Security Considerations

For adding and removing css classes, there should be no security concerns, and for custom animations it will be up to the author of those animations to address the security considerations of their implementation.

# Performance Considerations / Test Strategy

For performance considerations I intend to run multiple versions in the browser and evaluate the frame breakdown through chrome:tracing. (I have no idea beyond this how to test the exact rendered performance characteristics of the solution)

Tests will validate the order and lifecycle of a running animation, behavior in different scenarios, interruption of running animations, ignoring animations that are children of other animations, and throttling the total number of running animations.

# Work Breakdown

Paul Rohde ([codelogic@google.com](mailto:codelogic@google.com)) is currently building an initial implementation.