# **Design decisions in templating**

**Last updated: <when>**

**Owner: <who>**

**Status: <design|implemented|abandoned>**

Templating in AngularJS 2.x is based on the same concepts as AngularJS 1.x: dependency injection, directives and data binding.

## Create directive categories, no general purpose directive

The directive API in AngularJS 1.x was very complicated as it needed to cover all cases. In AngularJS 2.x we separated the use cases for directives into individual directive categories:

* ComponentDirective: provides a template and an isolated execution context for that template. In AngularJS 1.x this corresponds to directives with isolated scope and a template.
* TemplateDirective: instantiates templates and adds/removes/moves them to the DOM via the ViewPort. In AngularJS 1.x this corresponds to directives with transcludes.
* DecoratorDirectives: enhances existing elements with additional functionality. In AngularJS 1.x this corresponds to directives with no transclude, no own scope and no template.

TODO: reword this... Controllers from AngularJS 1.x are no more supported. They allowed to add functionality to the execution context (aka "scope" in AngularJS 1.x) without defining a clear interface (no isolation)

## Use the <template> tag

* For custom elements, a <template> tag is required to prevent their immediate instantiation.
* When a template has multiple root elements, grouping them into a <template> tag is a simpler way compared to the -start and -end suffixes in AngularJS 1.x
* When template directives are nested, their order is defined by nesting multiple <template> tags with a single template directive on each one. This is also simpler compared to the directive priority in AngularJS 1.x

Note: If a template directive should contain a single root element, the template tag can be omitted. E.g. <div ng-if="...">

## Make it easy to consume any custom element

TODO add details

* bind-, {{}}, on-

### Why not use {{...}} syntax everywhere and remove bind-\*?

Double curly braces should have the same semantic at every place. E.g.

<input foo="{{model}}" value="{{model}}">foo: {{model}}

* foo: {{model}}: one way data binding with interpolation
* value="{{model}}": bidirectional binding
* foo="{{model}}": possibly unidirectional binding, depending on what the component chose to use as binding type.

I.e. by just looking at the template the binding type cannot be determined. Knowledge of directive specifics is required to understand the template.

## Make templates analyzable by tools

* know which parts in a template are expressions and which are not
* know the directives in a template to be able to verify a template (i.e. which attributes are allowed and which are not)
  + ng-config in templates

## Bind to properties and not to attributes by default

* all attributes of native elements have a corresponding property (TODO: To be proven!)
  + the binding will detect attributes that don't have a corresponding property and bind to the attribute instead.
  + to simplify the readability of templates, there will be a list of attribute/property mappings, e.g. attribute class maps to property className.
  + element properties always contain the current value of the property, attributes sometimes only specify the initial value (e.g. the attribute value for an <input> element)
* some element properties don't have an attribute but it would be nice to use them in a data binding (e.g. indeterminateproperty of checkboxes)
* element properties for boolean attributes (i.e. attributes that are either set or not set, e.g. the selected attribute on<option>) always have a value. Using attributes we would need to distinguish bindings for boolean attributes (adding/removing them) from bindings to non boolean attributes (setting the value).
* attributes are always strings, properties can have any value
* for native attributes, browsers will automatically update the attribute when the corresponding property changes. This is called "reflecting" the property in the attribute. E.g. changing the src property on an <img> will automatically change the src attribute as well.x

## Make execution context an isolated plain old JS object

In AngularJS 1.x, the "scope" was the execution context for expressions as well as the manager for watches. In early versions, the this of a controller was the scope and the controller methods were mixins into the scope. However, this added "magic" methods to every controller instance provided by the scope, and also allowed to access the parent execution context implicitly.

In Angular 2.x, the execution context for the expressions of a template can be any object. For components, this is the component instance.

## Use DI to get hold of parent directives without using data binding

Getting hold of parent directive instances was already possible in AngularJS 1.x using a controller on that directive. In AngularJS 2.x a directive can ask for any parent directive via DI and the type of the parent directive.

Why not always use data binding? With multiple nestings, the parent needs to be passed through every component with a binding.

## Provide query mechanism to get hold of child directives in their DOM order

In AngularJS 1.x broadcast events were used to communicate with child directives. However, broadcast events did not provide the directives in their current DOM order and also did not return the directive instances directly.

Use cases for accessing child directives:

* tab-container wants its child tab-panes in the current DOM order
* ng-model wants to know about all validators on the same element
* video-player wants to get hold of the <video> tag in its template.

## Don't support scope events from AngularJS 1.x

TODO add details / reword

Querying the required directive instances and working with them directly is much easier to debug than firing an event and look who receives it

## Mark directives that should be queriable with a role

TODO add details / reword

Why not allow to query child directive using css selectors?

* an element can have multiple directives associated, need to know which one to return
* performance: if the elements that are queryable are known in advance, queries can be implemented in a simpler and more performant way

Why use strings as roles for queryables?

* for ngModel and validators: To not need a common base class for all validators
* to prevent cyclic type references: In src/examples/tab the component TabContainer queries for all child TabPanes, butTabPane uses the type TabContainer as well.

## Don't use promises in the templating

When using promises a request to get the promise value is always executed at the end of the turn, even if the promise is already resolved. However, instantiating Views synchronously is e.g. important for a custom element when a user calls document.createElement('some-ng-custom-element').

## Not every angular component is a custom element but can be exported as one

TODO add details / reword

* Lightweight components
* Support older browsers without using a polyfill

Angular components: Higher abstraction level, they do not touch the DOM directly.

## Use HTML imports for templates

AngularJS 2.x uses HTML imports for loading the templates of Angular components. This helps when the template references other resources like a CSS stylesheet to resolve relative paths.

## Bidirectional naming strategy between component classes and template URLs

In AngularJS 1.x directives, the template URL was defined as part of the directive registration in JavaScript.

AngularJS 2.x uses a bidirectional naming strategy to connect a component class with its template URL and vice versa. I.e. Angular can load the component class given a template URL but also load the template given a component class. This is needed as AngularJS 2.x supports defining Angular components as well as custom elements:

E.g. given the following Angular template, Angular first knows about the component module some-component and then needs to load the corresponding template:

<template ng-config="some-component">  
 <some-component></some-component>  
</template>

E.g. given the following non Angular template that uses an Angular custom element some-component, Angular first knows about the template and then needs to load the corresponding component module:

<link type="import" href="some-component.html">  
<some-component></some-component>

## Separate Angular components from the underlying DOM element

TODO

## Don't separate the component from the underlying DOM element for angular custom elements

For custom elements, a user should be able to create the custom element, set some properties and then call a method on the element instance synchronously. E.g.:

var el = document.createElement('my-input');  
el.value = 'someValue';  
if (el.valid()) { ... }

Separating the component instance from the DOM element would require synchronous digests on every method call.

## Don't modify DOM structure directly, but use View / ViewPort / ViewFactory abstraction

TODO add details / reword

* allow implementing transclusion without ShadowDOM in a performant way
  + the granularity of View/ViewPort is much higher than allowing users to modify the DOM directly.

## Delay DOM modification until the end of the turn (flush phase)

Reasoning:

* modifying detached DOM is faster than modifying attached DOM (see below). To optimize this, we:
  + flush in depth first reverse order (e.g. nested repeater can add rows before the parent is attached)
  + don't touch stuff that is already removed (e.g. nested repeater where parent item has been removed)
* coalesce changes from XHRs or events that arrive quickly after each other

Benchmarks:

* Chrome (and maybe other browsers as well) calculate style information for getComputedStyle() for parent and sibling elements in one turn. This benchmark adds [children](http://jsperf.com/style-recalc) resp. [siblings](http://jsperf.com/style-recalc-sibling) and compares the following cases: Add a new element and do getComputedStyle in every loop vs. add all elements and then do getComputedStyle() on all. It shows that batching up is faster, which can also be seen in the timeline (only one recalculate style event in the click listener).
* Writing to not attached DOM is faster than working with attached DOM
  + for input.value there is no difference
  + at least on Firefox (not in Chrome) there is a significant difference for adding/removing css classes: [benchmark](http://jsperf.com/attached-detached-nonstructural-dom-manip)
  + there is a difference when adding/removing nodes: [benchmark](http://jsperf.com/attached-detached-dom-manip)
* Chrome (and maybe other browsers as well) coalesce layouts and paints that happen quickly after each other (e.g. in parallel setTimeout(0)). See [this](http://plnkr.co/edit/MzE5M4Ocz8LjSij5LsvA?p=preview) benchmark that updates 4 regions on the screen and look at the layout and paint events in the DevTools timeline. So using requestAnimationFrame to prevent paints / layouts is not always required!

Note: in flush phase we only buffer structural changes, but not property changes to individual elements:

* gets very complicated to decide for custom elements and native elements when a binding can be moved to flush phase and when not. E.g. <input bind-value="value"> can be moved to flush phase, but <input bind-validation-message="error"> cannot / need to cause another digest.
* using the directive.bind notation Angular already buffers reads to native properties
* not buffering property writes on elements helps for some usecases where synchronous reaction is needed. E.g. prevent double submit of a form with a click handler that disables the button / form.

TODO add details / reword / investigate more

The following cases require the flush phase to be after the digest phase in the same turn:

* a directive that has a domAttached callback and needs to trigger a digest afterwards.
* a bidirectional binding to a custom element that requires the attached callback to initialize. E.g.

<polymer-tab-container bind-selected="selectedTabIndex">  
 <polymer-tab-pane ng-repeat="tab in tabs" ng-no-flush="true">  
 </polymer-tab-pane>  
</polymer-tab-container>

Delaying the flush phase in those cases would lead to missed frames. If those cases are not present, the flush phase can also be delayed until the next requestAnimationFrame.

TODO This detection is not implemented yet, i.e. flush is always directly after digest.

## Order calls to document.register Element by order in root document

TODO: add details / reword

* so that parent elements are created and attached before child elements.
* needed for DI to work properly
* see loadBarrier