Templating

***Status: Draft***

*Authors:* [*misko@google.com*](mailto:misko@google.com)*,* [*tbosch@google.com*](mailto:tbosch@google.com)

Date: March 2014

[Objective](#heading=h.fxpk50cps4zs)

[Background](#heading=h.wwgnimruvhlu)

[Why simplify the directive API?](#heading=h.wc2jzxci433y)

[Why integrate with other component frameworks?](#heading=h.te1bv0y2uvfp)

[Why improve performance?](#heading=h.e16syui33lbe)

[Why make templates analyzable by tools?](#heading=h.836814yvweti)

[Prior Art](#heading=h.v93ljdiqj2co)

[Polymer](#heading=h.85vkosi0q2ay)

[TODO Evaluate HTML Imports and Node.bind](#heading=h.4cj02ormkxxp)

[Strong points](#heading=h.vq0eq72cuj7k)

[Weak points](#heading=h.n2xnh3nv4ssi)

[X-Tags](#heading=h.g6oihq651rl)

[Strong points](#heading=h.laja0s32w4k)

[Weak points](#heading=h.clzuwls7slqy)

[React.js](#heading=h.gsj1b5cyemrl)

[MontageJS](#heading=h.9emn7hal386f)

[Ember.js](#heading=h.960qc2vipnr)

[Detailed Design](#heading=h.waulgszaejty)

[Conceptual overview](#heading=h.w4ivl8awy0d5)

[Directive Types](#heading=h.m4vrrf92763l)

[General restriction for directives](#heading=h.1c5d6wr90qwt)

[Directive type: Decorator Directive](#heading=h.kseaj6kx9adb)

[Directive type: Template Directive](#heading=h.1nbvu5qn2h6)

[Directive type: Component Directive](#heading=h.s63raxrnoohb)

[Instantiation Order of directives](#heading=h.m3ie4exf3mdj)

[Data binding configuration in templates](#heading=h.nstq9396uu5k)

[Binding to events of elements or directives](#heading=h.k972nqcof3e)

[Binding to properties of elements](#heading=h.5i8lai5znheb)

[Interpolating properties of elements](#heading=h.744fcyl05j9u)

[Interpolating text](#heading=h.93flvf5s9i34)

[Expressions](#heading=h.t4sg86p3zlaf)

[Deeply integrated with dirty checking](#heading=h.xsxqhtz0tnmr)

[One time binding](#heading=h.6rzfundz70u6)

[Errors from expressions](#heading=h.8bkyc34d3eu)

[Directive-API](#heading=h.5p54f6o4103h)

[Class annotation](#heading=h.pn1ptb72lsup)

[Constructor, DI and directive communication](#heading=h.bpw1rcnj3zi8)

[Directive communication](#heading=h.w3hux8ry2l3m)

[Example](#heading=h.8d7t4zbp1g57)

[Bindable properties](#heading=h.zhi3cqvcrjd8)

[Microsyntax for properties](#heading=h.d777tozfbsta)

[Lifecycle hooks](#heading=h.4fydw9fdaryb)

[Events](#heading=h.16drnw98apy9)

[Compiler-API](#heading=h.3t0ip9vd8okx)

[Example implementations of directives](#heading=h.mg2asle4eqcv)

[Decorator: ng-show](#heading=h.scn7f5iypwys)

[Template Directive: ng-if](#heading=h.t1ct7bne4s8y)

[Template directive: ng-include](#heading=h.39qdxwqnpnfr)

[Template directive: ng-repeat](#heading=h.fv4052m60wg8)

[Component Directive: pane](#heading=h.eemoytw63f3n)

[Consuming and publishing components from/to other web component frameworks](#heading=h.r9g2q5f6mrso)

[Frameworks that use Custom elements (X-Tags, Polymer)](#heading=h.yz80vb2d1zoj)

[Frameworks that use Node.bind (Polymer)](#heading=h.2hefdebrqak1)

[Frameworks that use html imports (Polymer)](#heading=h.ko1c0xdtulbr)

[Conclusions](#heading=h.sm2w3k47enh)

[Templates are analyzable and validatable by IDEs](#heading=h.4pwli5780eiu)

[Performance improvement](#heading=h.y08xmcvkuz0n)

[Event delegation](#heading=h.usl8gifd6pna)

[Lightweight directives](#heading=h.c93hj8tmxfe3)

[New dirty checking mechanism for expressions](#heading=h.g49auixmv9b)

[Simplified Directive API](#heading=h.sshzjz2murav)

[Integrate with other web frameworks using web standards](#heading=h.k8neepigd7z6)

[FAQ](#heading=h.8ld4akvsf9fd)

[We want the ng-controller of Angular 1.x back](#heading=h.ew884z7lmsne)

[Why not use Polymer as HTML Compiler?](#heading=h.w9nzbq3s8t0b)

[Why not use {{...}} syntax everywhere and remove bind-\*?](#heading=h.ccqi85bjsc6x)

[Caveats](#heading=h.pz20okjzphgm)

[Security Considerations](#heading=h.qjnbvlr7uej1)

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

[Work Breakdown](#heading=h.59osdxvm5jwp)

# Objective

Redesign of the Angular HTML compiler that:

* simplifies the directive API
* integrates with other component frameworks using web standards
* improves performance
* allows tools like IDEs to analyze and validate templates

# Background

## Why simplify the directive API?

* The concepts in Angular 1.2.x are very intermingled and because of this they are hard to explain and learn (e.g. Scope). This design doc tries to simplify and reduce the amount of used concepts.
* In Angular 1.2.x there is only one API to define all kinds of directives. However, the API is very complicated. This design doc defines special cases for component directives, template directives and decorator directives, so that the API can be optimized for those cases (e.g. only template directives need to care about compile/link separation)

## Why integrate with other component frameworks?

With the evolving web component standards it is expected that many components will be built on top of them. Angular should play nicely with all of those frameworks to allow Angular apps to leverage those components and also allow components built with Angular to be used with other frameworks.

## Why improve performance?

Being fast by default enables people to focus on solving their business problems rather than tweaking their apps to get better performance. Better performance always enables more complex or even new breeds of apps to be built.

## Why make templates analyzable by tools?

In Angular 1.2.x tools that validated templates didn't work very well, as they had to have special cases for every directive. By this, many errors were not detected until someone clicked through the app. Validating templates can help detecting errors in a very early stage and enables faster development cycles.

# Prior Art

List of existing solutions and their corresponding strong/weak points.

## Polymer

### TODO Evaluate HTML Imports and Node.bind

### Strong points

Test bed for web components standards. Uses polyfills/prolyfills and adds sugaring on top.

### Weak points

Polyfills

* Custom Element
  + granularity: attribute level components are not doable with custom elements
  + testability: elements are registered at the document, no way to unregister them without reloading the document
  + instantiation: does not work well with DI that uses constructor injection
    - no constructor
    - no hook to intercept the creation of the element
  + polyfill is not performant, as it uses DOM Mutation events: TODO measure this
    - we could work around this by just deactivating the change listening part
* Shadow DOM
  + polyfill is not performant: TODO measure this
  + ?? provide a lightweight implementation based on transclusion and the <content> tag, that we only use when the real Shadow DOM is not present ?? → bad: different semantics (e.g. regarding css, css selectors, …)
* Object.observe / observe-js
  + does not support watching function calls and getters
  + Angular always uses a hybrid model
    - use Object.observe for properties. If not present Angular has a fallback
    - use own dirty checking for things that are not supported, e.g. function calls
    - provides higher abstraction level (compared to Object.observe), just like observe-js

Polyfills

* TemplateBinding
  + fixed set of template operations (e.g. if, repeat), i.e. not extensible (e.g. ng-switch)
  + not clear how animations are handled (e.g. fadein, fadeout)

Opinionated parts (how Polymer defines elements)

* polymer-element and Polymer constructor
* Opinion of Polymer: everything is a component, and everything is done in HTML
* Opinion of Angular: business logic and ui logic should be separated. Business logic should be written in JS, so it's easier to test and express.
* DI is hard to combine with Polymer (see above for custom elements)

See <https://docs.google.com/a/google.com/document/d/16O2Im1ekfdJ4FU8FBbVRYGjqsXjmcV3tYFg1vyfhYC8/edit#heading=h.j75blxt2a4j7>

## [X-Tags](http://www.x-tags.org/docs)

* Does not implement data binding, but allows components to listen for attribute changes and implement setters for object properties.
* Uses Custom Elements and Mutation Observers from Polymer as Polyfills, but does not use Shadow DOM.
* The default components under <https://github.com/x-tag> don't seem to listen for attribute changes but only use property setters and DOM events.

### Strong points

* Plain and simple component model using custom elements
* Component library [Brick](https://www.google.com/url?q=https%3A%2F%2Fmozbrick.github.io%2F&sa=D&sntz=1&usg=AFQjCNHyWX4g1fivqLGvyMt1Wuk8ljPDaQ)

### Weak points

* No data binding

## 

## React.js

Not a full framework like AngularJS

Strong

* Virtual DOM to optimize performance
* Support server side rendering

Weak

* API is complex
* Components access (virtual) DOM directly -> ?? testability??
  + Components don’t “access” it, they produce it. A component is a function from input state to a chunk of virtual dom
    - https://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html#components-are-just-state-machines
  + Example of a test
    - https://facebook.github.io/jest/docs/tutorial-react.html
* Jsx: JavaScript with inline html
  + it is not really inline html, it desugars to a single JavaScript expression
    - <https://facebook.github.io/react/tips/if-else-in-JSX.html> is a consequence of it
  + [+] allows keeping whole component definition in a single file
  + [+] why build a special templating language when it is possible to use javascript.
    - instead of “understanding” the template language React re-renders everything to Virtual DOM and diffs to see what should be changed on the page.
  + React.js catchphrase: “separate concerns, not technologies”

## 

## MontageJS

Strong

* simple, very few concepts to learn: everything is a component that get composed
* good story for packaging and reusing components
* good documentation, examples that look nice (e.g. mobile widgets)
* separate render phase to improve performance when writing to the DOM
* functional reactive binding (frb)
  + only way how components communicate

Weak

* a lot of code needed to define a component
* ?? injecting services via modules are always singletons -> testability ??

## Ember.js

# Detailed Design

## Conceptual overview

Instead of creating custom elements for behavior, Angular associates behavior to DOM elements. This behavior is called a *directive*. A directive is a JavaScript class that is instantiated for every DOM element that matches a directive specific CSS selector. Angular allows multiple directives to be associated with the same element.

DOM elements use properties and events to communicate with an application, very simple but powerful. Directives are built using the same interface, i.e. properties and events.

The user interface of an Angular application is built using so called *component directives*. A component directive stores application data and logic, as well as a *html template* that defines the user interface of the component. The component instantiates the template,Angular also supports directives that add additional behavior to an element or component, without being a component itself, the so called *decorator directives*. Those directives are usually triggered by an attribute on an element, and there can be multiple decorator directives per element. E.g. adding the attribute ng-show="false" to an element hides that element.

To connect the data/logic of the component with the directives in its template, Angular uses *bidirectional data binding*. I.e. changes in the data of the component are automatically propagated to the directives and vice versa. The data binding is configured in the template and uses special html attributes. A binding is defined using a so called *expression*, which allow access to data or call functions. Every expression is evaluated in an *execution context*, that is the object which is used to dereference the fields in the expression. Usually, this is the component instance.

Angular also provides a *dependency injection* framework, which takes care of instantiating and wiring all parts of the application. Using DI, functionality can easily be split into parts and combined together.

Finally, Angular allows to export its components as web components as well as import web components as Angular components.



## Directive Types

Why:

* Cluster the usage patterns for directives and optimize/simplify them based on the experience from Angular 1.2.x. I.e. no general directive that can do everything but with a very complex api (e.g. remove the compile/link phase from Angular 1.2.x)

Overview

|  |  |  |  |
| --- | --- | --- | --- |
|  | Provides execution context for expressions | Provides a hole in the DOM to insert template instances | Isolates the DOM/expressions/css for reuse |
| Decorator | no | no | no |
| Template | can create child execution context, don't have to. | yes | no |
| Component | isolated execution context, always. | no | yes |

### General restriction for directives

* directives should not remove / change the structure of the DOM that they did not create

Why:

* The directive that did create that part of the DOM would not know about this and by this make wrong assumptions about elements and their place in the DOM.

### 

### Directive type: Decorator Directive

Directives to decorate elements, e.g. add a tooltip, show/hide elements, …

Restrictions:

* Create new elements and do anything with them
* Only change attributes on other elements that they did not create

Examples: ng-show

Why:

* Be able to decorate existing elements / directives, and have multiple directives for the same element.

|  |
| --- |
| <div ng-show="true">  hello world!  </div> |

### Directive type: Template Directive

Template directives turn the element that they are on into a template. The original location of the element/template gets replaced with a hole that the directive can fill with instances of the template.

A template directive may create a child execution context for a template instance, but it does not have to. E.g. ng-repeat will create separate execution contexts for every row, but ng-if will reuse the parent execution context.

Examples: ng-if, ng-repeat, ng-view, ng-switch, ng-include.

General syntax

|  |
| --- |
| <template ng-repeat>  <div>...</div>  </template> |

The template element itself gets removed and is not part of the template itself. Only one template directive is allowed per <template> element. To apply multiple template directives to some html nested <template> elements have to be used.

Example: Applying multiple template directives to an element

|  |
| --- |
| <template ng-repeat>  <template ng-if>  <div>...</div>  </template>  </template> |

As most of the times only one template directive is used for every element, template directives can also be used on normal elements. Behind the scenes the element that they are on will then get converted into a template.

E.g.

|  |
| --- |
| <ul>  <li ng-repeat>  <span ng-if>...</span>  </li>  </ul> |

is equivalent to

|  |
| --- |
| <ul>  <template ng-repeat>  <li>  <template ng-if>  <span>...</span>  </template>  </li>  </template>  </ul> |

Why:

* nesting template elements makes the instantiation order of the templates explicit (e.g. when ng-if and ng-repeat were on the same element). In Angular 1.2.x this was solved using an ambiguous priority field for every directive.
* a template tag allows multiple children as root nodes of the template instance (in Angular 1.2.x. this was solved using <directive>-start and <directive>-end markers)
* template directives were the reason for the compile/link phase for directives in Angular 1.2.x. Angular 2.0 removes the general compile/link construct by introducing the special template directive.

### Directive type: Component Directive

Encapsulates logic in javascript, a html template and an optional css stylesheet into a component that can be easily reused. For this it isolates expressions in the template, the dom tree and the css from the rest of the application and provides a well defined interface.

Isolation of the execution context: The component directive instance becomes the new execution context for all expressions in its template. I.e. expressions in the template can only access properties and functions that are on the component directive instance.

Isolates the DOM and CSS: Component directives use Shadow DOM to prevent other directives from modifying their content. Also, events from elements in the Shadow DOM don't bubble out of it.

Interface for a component:

* Properties and events: Templates that use a component can bind to properties of the component directive and by this read/write data from/to it. A component can also fire events that a parent component can listen to.
* CSS properties: Templates that use a component can change the styling in the component using the default means of Shadow DOM
* Child fragments: Templates that use a component can provide children elements that the component can insert into it's template using the <content> tag of Shadow DOM. Expressions in those children elements keep connected to the outer template's execution context and not to the component's execution context.

Notes

* most often, the initial html page will contain a component on the <body> tag to provide an execution context for the expressions used in the page.

Why:

* components allow to easily reuse a html/css/code block and build bigger blocks out of it
* the execution context for an expression is always an instance of a class (the component directive). By this we can do type checks, … for expressions.

### 

### Instantiation Order of directives

1. Directives on parent element are instantiated before directives on child elements
2. If existing: Instantiate the template directive (there can be only 1 template directive per element)
3. Instantiate decorator directives for the element. Order is determined by dependencies in the directive constructor
4. If existing: Instantiate the component directive

## Data binding configuration in templates

### Binding to events of elements or directives

Syntax: @on-[event name]="[expression]"

Example: <button on-click="doSomething()">

Semantic: Executes the expression (doSomething()) whenever the event (click) is fired on the element.

Why:

* It allows us to bind to any third party event system, which is based on DOM events.
* If a directive declares a new event, we don't need a new directive to listen to it.
* Could be used to bridge to jQuery custom events (would require jQuery event bridge).
* Can be implemented using event delegation (a single top level handler for all event types, rather than each element having its own handler). This is faster, uses less memory.

### 

### Binding to properties of elements

Syntax: @bind-[property name]="expression"

Example: <input bind-value="user.name">

Semantics: Watches the value of the expression (user.name) for changes and updates the property on the element (value). This will also update the property on all directives that support this property and are applied to this element. Also listens for change events on the element or matching directives, and if they fire, updates the expression with the new value.

Note: This will not do a data conversion (e.g. to a string), in contrast to interpolating properties of elements (see below).

Note: This will throw an error if

* The property on the element changes but the expression is not writable
* The value of the expression changes but the property is not writable

Why:

* Bidirectional data binding to any property of every element and directive, not just through some special directive (e.g. ng-model for inputs)

Why bind to element properties and not attributes:

* almost all attribute have a corresponding property
  + the binding will detect attributes that don't have a corresponding property and bind to the attribute instead.
  + 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)
* 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 have 3 cases: Attribute is not present, attribute is present, attribute is present with a defined value.
* some element properties don't have an attribute but it would be nice to use them in a data binding (e.g. indeterminate property of checkboxes)
* for custom elements / directives it's easier to access properties on their instance, compared to attribute values (no need to call element.getAttribute() and no need for serializing / deserializing into a string)
* for native attributes, browsers will automatically update the attribute when the corresponding property changes (e.g. changing the src property on an <img> will automatically change the src attribute).

### Interpolating properties of elements

Syntax: @[property name]="a {{ [expression] }} b"

Example: <input title="some Text {{someValue}}">

Semantics: <input bind-title="'a '+ someValue+' b'">

This will create a data binding for the property where {{}} is used. The value will always be converted into a string, and the binding will always be unidirectional.

I.e. <input title="{{someValue}}"> corresponds to <input bind-title="'' + someValue">

Why:

* A way for the template designer to easily combine content in the html with content from JavaScript.

### Interpolating text

Syntax: {{ [expression] }}

Example: hello {{user}}

Semantics: hello <span bind-text-content="''+user">

Creates a data-binding for the content of a text node. The value will always be converted into a string, and the binding will always be unidirectional.

Why:

* A way for the template designer to easily combine content in the html with content from JavaScript.

## 

## 

## Expressions

Expressions in Angular 2 are very similar to expressions in Angular 1.2.x (<http://docs.angularjs.org/guide/expression>). In Angular 2, the following features are new:

### 

### Deeply integrated with dirty checking

Expressions are deeply integrated in the the dirty checking process. By this, the dirty checking knows about common prefixes in expressions or constant parts and can do optimizations. For details see the [dirty checking design doc](https://docs.google.com/a/google.com/document/d/10W46qDNO8Dl0Uye3QX0oUDPYAwaPl0qNy73TVLjd1WI/edit#heading=h.xgjl2srtytjt).

### One time binding

Syntax: add a double colon before the expression, e.g. {{::foo}}

Example: Interpolate labels that won’t change afterwards.

Why?

* To leverage implicit knowledge about the data to improve performance

Why part of the expression language?

* Only the template author knows which things don’t change; a directive might be used at one place with one time bindings and at another place with normal bindings
* Should work inside of {{}} as well as inside of bind-...
* May also work for parts of an expression in the future

Semantics:

* Wait until the value is no more undefined. Then apply the value to the binding and remove the binding.

### Errors from expressions

In Angular 1.2.x expressions were very forgiving. However, this made it hard to find mistakes in variable names, ...

In Angular 2, the semantics for a.b.c and a[b][c] change in the following way:

* If b is not in the object a, i.e. !('b' in a), throw an error

## Directive-API

Directives are defined using ES6 classes and use annotations to define meta data.

### Class annotation

Every directive is an ES6 class that has a special annotation, i.e. @DecoratorDirective, @TemplateDirective or @ComponentDirective. Those annotations have the following properties in common:

* selector (string) (e.g. input[type="text"]): Defines via a css selector to which elements this directive should be applied to.
  + no child selector or pseudo classes
* events (list of string): A list of event names that this directive fires. In templates, those events can then be bound using on-... attributes. See [Events](#h.16drnw98apy9) for details.
* visibility (['local' | 'direct-children' | 'any-children']): defines which directives in the element's subtree can access the directive instance. See below for details.
* microsyntax (string): Defines the syntax for special properties and how parts of those properties map to other properties. See [Microsyntax for properties](#h.d777tozfbsta) for details.

Example:

|  |
| --- |
| import {ng} from '...';  @ng.DecoratorDirective(selector: '[ng-show]')  class NgShowDirective {} |

This will create a new instance of the NgShowDirective for every element that has a ng-show attribute.

Why:

* Annotation allows tools to find the directives for an element without instantiating the directive / running js code.

### Constructor, DI and directive communication

The constructor of every directive gets the values for its parameters via dependency injection. By this, directives can get a hold of other object instances that are created via DI.

Example:

###### 

|  |
| --- |
| import {ng} from '...';    @ng.DecoratorDirective(selector: '[ng-show]')  class NgShowDirective {  @Inject(window.HTMLElement, ng.Http, ng.SomeOtherDirective)  constructor(element, http, someOtherDirective) { … }  } |

###### 

Whenever the NgShowDirective is created it will get the element to which it's applied as an argument.

#### Directive communication

Besides non-ui objects like Http, a directive can also access other directives on the same element or on parent elements. For this, a directive simply requests the other directive type to be injected in it's constructor. Using the visibility flag of the class level directive annotation, a directive can also restrict which directives on the element's subtree have access to the directive instance.

#### Example

A form that contains an input element should bind its value to the expression user.value and also validate the text to be not empty. Listening to input changes is done through the NgInputDirective, validation is done in NgModelDirective and state keeping is done in the NgFormDirective (see Angular 1.2.x for the separation of concerns). For this the NgModelDirective needs access to the NgFormDirective instance on the parent <form> element, and the NgInputDirective needs access to the NgModelDirective on the same element.

###### 

|  |
| --- |
| <form>  <input type="text" ng-model="user.value" required>  </form>    import {ng} from '...';    @ng.DecoratorDirective(  selector: 'form',  visibility: 'subtree'  )  class NgFormDirective {  constructor() { … }  }    @ng.DecoratorDirective(  selector: 'input[type=text]',  visibility: 'local'  )  class NgModelDirective {  @Inject(NgFormDirective)  constructor(ngForm) { … }  }    @ng.DecoratorDirective(selector: 'input[type=text]',)  class NgInputDirective {  @Inject(NgModelDirective)  constructor(ngModel) { … }  } |

Why:

* allow directives that know about each other to communicate without separate bindings in the template
* easy access to other directives and services
* helps in determining the instantiation order for directives via the constructor dependencies. See [Instantiation Order of directives](#h.m3ie4exf3mdj) for details

### 

### Bindable properties

All properties of directives can be used for data binding. To configure the data binding, setters can have an optional @ng.PropertySet annotation.

@ng.PropertySet: Configures the way data binding writes to a property

* Syntax: @ng.PropertySet({trigger: ['reference'|'collection'|'deferred'], domOnly: [false|true] })
* trigger: controls when and with what the setter should be called
  + 'reference': if the reference changes, pass in the reference (default)
  + 'collection': if an entry in the collection changed, pass in the collection
  + 'deferred': pass in the compiled expression so that the directive can execute it later on (a function will be passed in that takes an execution context as argument)
* domOnly: the setter only changes the DOM but no other property that is watched
  + implication: Will be called only once during DOM write phase
  + default: false

Why:

* Preserve the property names after minification
* Provide an easy way to databind to properties of a directive

Example:

|  |
| --- |
| import {ng} from '...';  @ng.DecoratorDirective {  selector: 'dialog'  }  class Dialog {  constructor() {  this.\_content = null;  }  get content(content) {  return this.\_content;  }  @ng.PropertySet({domOnly: true})  set content(content) {...}  }  <dialog content=”{{1+2}}”> |

### Microsyntax for properties

Sometimes directives need multiple values from the template designer. E.g. ng-repeat needs a collection, a property to put the collection value in while iterating and an optional track-by expression. Angular 2.0 provides a general construct called microsyntax that allows any directive to easily parse multiple parts out of an expression.

A microsyntax for a property is defined using the microsyntax property in one of the directive class level annotations.

Syntax for defining a microsyntax:

* MICROSYNTAX=(VARIABLE|FIXED|OPTIONAL)+
* VARIABLE=$\w+
* FIXED=[^$\[\]]+
* OPTIONAL=\[MICROSYNTAX\]

Example of defining the microsyntax for ng-repeat:

|  |
| --- |
| @TemplateDirective(  selector: '[ng-repeat]'  microsyntax: {  'ng-repeat': '$item-name in $collection [track by $track-by]'  }) class NgRepeat {  @ng.PropertySet({trigger: 'reference'})  set ngRepeatItemName() { … }   @ng.PropertySet({trigger: 'collection'})  set ngRepeatCollection() { … }  @ng.PropertySet({trigger: 'deferred'})  set ngRepeatTrackByFn; } |

This can then be used in the following ways:

|  |
| --- |
| <div ng-repeat="item in items track-by item.id">  <div ng-repeat-item-name="item" ng-repeat-collection="items" ng-repeat-track-by="item.id"> |

Note that instead of using a property with microsyntax directly the parts defined in the microsyntax can also be filled using separate properties that correspond to the parts of the microsyntax.

Why:

* make it less verbose to use a directive with multiple properties like ng-repeat
* the microsyntax can still be validatable by tools

### Lifecycle hooks

Any directive can implement the following special functions

* attach: is called once, after all setters have been called by their corresponding binding in templates
* detach: is called when the element of the directive gets destroyed

Component directives can implement the following functions

* templateLoaded: called when template of the component has been loaded and added to the component element.

Why

* Allow templates to know about their lifecycle and do initialization and finalization logic.

### Events

Directives can fire custom events using HTMLElement.dispatchEvent and listen for events using HTMLElement.addEventListener (see <https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events>). The events bubble up the tree until they reach a component (they isolate events too).

Directives have to declare all events that they fire in the class annotation. Angular uses this information for:

* Trigger dirty checking: Whenever one of those events fire, a dirty check run will be scheduled.
* Allow tools to validate the on-... attributes in templates.

Note that Angular might provide a utility service to make this easier.

Example:

|  |
| --- |
| @ng.DecoratorDirective(  selector: 'dialog',  events: ['close']  )  class Dialog {  @Inject(window.HTMLElement)  constructor(element) { ... }  close() {  var evt = new Event('close');  this.element.dispatchEvent(evt);  if (!evt.defaultPrevented()) {  // really close the dialog...  }  }  } |

### Compiler-API

The compiler API is mostly used by template directives.

|  |
| --- |
| class Injector {  // see <https://github.com/angular/di.js/blob/master/src/injector.js>  }  class Directives {  // contains all directives that are available on  // an injector.  }  class Compiler {  call(elements:NodeList, directives:Directives):ViewFactory  }  class ViewFactory {  call(injector:Injector, childContext:Object)  }  class View {  }  class ViewHole{  append(block:View)  /\* inserts or moves, just like DOMElement \*/  insertBefore(previousView:View, view:View)  /\* inserts or moves, just like DOMElement \*/  insertAfter(previousView:View, view:View)  remove(view)  } |

## Example implementations of directives

### Decorator: ng-show

Example implementation for ng-show:

|  |
| --- |
| import {ng} from '...';  @ng.DecoratorDirective(  selector: '[ng-show]'  )  class NgShow {  @Inject(window.HTMLElement)  constructor(element) {  this.element = element;  }  set ngShow(value) {  this.element.style.display = value ? 'block' : 'hidden';  }  } |

### Template Directive: ng-if

A basic template directive that uses the content of the template tag:

|  |
| --- |
| import {ng} from '...';  @ng.TemplateDirective(  selector: '[ng-if]'  )  class NgIf {  @Inject(ng.ViewHole, ng.ViewFactory, Injector)  constructor(viewHole, viewFactory, injector) {  this.viewFactory = viewFactory;  this.viewHole = viewHole;  this.injector = injector;  this.view = null;  }  set ngIf(value) {  if (this.view) {  this.viewHole.remove(this.view);  }  if (value) {  this.view = this.viewFactory(injector);  this.viewHole.add(this.view);  }  }  } |

### Template directive: ng-include

A template directive can also choose not to use the template that was initially provided:

|  |
| --- |
| import {ng} from '...';  @ng.TemplateDirective(  selector: '[ng-include]'  )  class NgInclude {  @Inject(ng.ViewHole, ng.Compiler, ng.Http, ng.Directives,  Injector)  constructor(viewHole, compiler, http, directives, injector) {  this.viewHole = viewHole;  this.compiler = compiler;  this.directives = directives;  this.injector = injector;  this.view = null;  this.$http = http;  }  set ngInclude(value) {  var self = this;  this.$http(value).always(removeView).then(addView);  function removeView() {  if (self.view) {  self.viewHole.remove(self.view);  self.view = null;  }  }  function addView(templateString) {  var viewFactory =  this.compiler(templateString, this.directives);  this.view = viewFactory(injector);  this.viewHole.add(this.view);  }  }  } |

### Template directive: ng-repeat

A template directive can also provide an new execution context. All expressions will check this execution context first for variables. If the variable is not present on it the variable will be resolved on the current execution context (which is stored in the Injector). A simple implementation (and not very accurate implementation regarding change detection) of ng-repeat could look like this:

|  |
| --- |
| import {ng} from '...';  @ng.TemplateDirective(  selector: '[ng-repeat]'  microsyntax: {  'ng-repeat': '$item-name in $collection [track by $track-by]'  })  class NgRepeat {  @Inject(ng.ViewHole, ng.ViewFactory, Injector injector)  constructor(viewHole, viewFactory, injector) {  this.viewHole = viewHole;  this.viewFactory = viewFactory;  this.injector = injector;  this.views = [];  }  @ng.PropertySet({trigger: 'reference'})  set ngRepeatItemName(value) {  this.itemName = value;  this.update();  }  @ng.PropertySet({trigger: 'collection'})  set ngRepeatCollection(arrayChangeRecord:ArrayChangeRecord) {  this.arrayChangeRecord = arrayChangeRecord;  this.update();  }  @ng.PropertySet({trigger: 'deferred'})  set ngRepeatTrackByFn(value) {  this.trackByFn = value;  this.update();  }  update() {  …  // deleting a view  this.viewHole.remove(deletedView);  …  // creating a new view  var childContext = {  $index: index,  $odd: !!index%2,  $even: !index%2  };  childContext[this.itemName] = this.collection[index];  var view = this.viewFactory(injector, childContext);  viewHole.append(view);  }  ...  }  } |

### Component Directive: pane

Example implementation of a pane that decorates content with a title

|  |
| --- |
| import {ng} from '...';  @ng.ComponentDirective(  select: 'pane',  template: ng.inline('<div>{{title}}</div><content/>'),  css: ng.url('pane.css'),  )  class sample.Pane {  constructor() {}  set title() { … }  } |

## Consuming and publishing components from/to other web component frameworks

### Frameworks that use Custom elements (X-Tags, Polymer)

When Angular detects that the custom elements API is available, it will automatically register it's components as custom elements. The properties of the component directive will be made accessible on the custom element instance.

Export an Angular component as a custom element:

* When the custom element API is available, any JavaScript application can create Angular components, access their properties and listen for their events in the same way as accessing a DOM element.

Use a custom element inside of an Angular app:

* Angular is able to use any custom elements directly, as it is able to use native DOM elements like <input> directly.
* To listen for changes on custom elements, they have to fire events, just like native DOM elements do (e.g. <input> fires a change event, …)

### 

### Frameworks that use Node.bind (Polymer)

Some frameworks (e.g. Polymer) use Node.bind to notify a consumer about changed properties and also be notified about changes to their properties. Therefore, if the Node.bind API is available, Angular's data binding will integrate with Node.bind for reading of properties, writing to properties as well as listening to changes of properties of a component.

### Frameworks that use html imports (Polymer)

Some frameworks (e.g. Polymer) use html imports to load components. To load an Angular component using an html import, Angular is able to create an html snippet in a separate build step that can be used with an html import.

## 

# Conclusions

### Templates are analyzable and validatable by IDEs

In Angular 1.x directives were allowed to interpret attributes of elements in arbitrary ways. Because of this, tools like IDEs needed to manually collect meta data about those directives to analyze and validate them in a template. Angular 2.x changes this with special syntax in the html templates and annotations on directives.

Allowed elements and attributes:

* all valid HTML elements and attributes
* all elements and attributes for which there is a directive that matches that element/attribute combination in it's css selector.
* all bind-\* and on-\* attributes

Value of html attributes:

* If the attribute starts with bind-\* or on-\*: must be an expression
* parts in the value enclosed in {{}} must be expressions
* attributes for which there is a property in a directive that specifies that the attribute should be handed to the directive as a compiled expression must be expressions

### Performance improvement

#### Event delegation

Due to the new syntax rule for event attributes, it’s easy to use event delegation, ie. only install one event listener at the root of the document for every event type: Whenever the event reaches the document root Angular only needs to check all attributes that start with “on-” and execute the appropriate expression. This saves time and resources by not having to install listeners on every element.

#### Lightweight directives

Angular directives are no web components by default. Only the components that should be reusable in other applications will be exported as web components. By this we can do performance improvements by e.g. not using shadow dom for not exported components on platforms that don't support shadow dom natively, …

#### New dirty checking mechanism for expressions

Angular 2 deeply integrates dirty checking with the expressions and also uses a new dirty checking implementation. Because of this, dirty checking in Angular 2 will be faster compared to Angular 1.2.x.

### Simplified Directive API

Clustering the usage patterns into different directive types allows to simplify the API for those use cases.

E.g.

* no compile/link phase anymore
* simpler way to communicate with other directives using DI
* order for instantiation of directives is well defined, no need for a priority definition
* execution context can only be introduced in components and is an instance of a class. There is no need to put data on a separate object (in Angular 1.x $scope).

### Integrate with other web frameworks using web standards

Angular is able to import components from/export components to web component frameworks like Polymer or X-Tags.

# FAQ

## We want the ng-controller of Angular 1.x back

* please use a component instead. This will also make your app more testable.

## Why not use Polymer as HTML Compiler?

* collect thoughts / reasoning about this
* really need to study Polymer!

## 

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

|  |
| --- |
| <input value="{{model}}">  <input bind-value="model"> |

Pro

* Simplification: There is only one way to write the binding syntax.
  + For string properties that are not changed by the element itself (or by the user) the following is equivalent:

<input title="{{someExpr}}">

<input bind-title="someExpr">

Con

* Double curly braces should have the same semantic at every place (Miško)  
  <input foo="{{model}}" value="{{model}}">foo: {{model}}  
  Issue: semantics of {{}} change due to context.
  + foo: {{model}} in a text node provides one way data binding with interpolation
  + value="{{model}}": Interpolation not allowed, because value chose to interpret the binding in bidirectional way.
  + foo="{{model}}": Foo has different semantics from value because the component chose not to treat it in bidirectional way. There is no way to tell this from looking at the template. Knowledge of directive specifics is required.
* When using {{}} together with other text it’s not assignable and would lead to runtime errors when using it with bidi binding.
  + <input value=”asdf{{someExpr}}asdf”>

Approach for Angular 2.0:

* bind-... is the general case, and {{}} is just one special case of it.

|  |
| --- |
| <dialog title="{{model}}">  ===  <dialog bind-title="''+model"> |

# Caveats

Don't bind to attributes but to properties of elements

* see [Binding to properties of elements](#h.5i8lai5znheb)

Don't make every directive a custom element

* be more flexible in defining additional markup that should be used in html (e.g. use new elements, new attributes and/or css classes)
* allow light weight directives for performance reasons

# Security Considerations

* Binding to every property (e.g. like innerHTML) can introduce security problems. We will use Strict Contextual Escaping as already implemented in Angular 1.2.x using a whitelist of properties that should be checked.
* The templating will also filter the src attribute for images and anchors for a white list regex, just like Angular 1.2.x does.
* In addition to the existing security strategies from Angular 1.x, script injection attacks should be mitigated by allowing interpolation symbols to be escaped (discussed at <https://github.com/angular/angular.js/issues/5601> and <https://github.com/angular/angular.js/pull/5628>).

# 

# Performance Considerations / Test Strategy

Angular 2.0 will not create a web component for every directive. However, Angular uses a separate export build step to export Angular components as web components. By this, Angular is able to have great performance also on browsers that don't implement the web components standards, especially as the corresponding polyfills (e.g. Shadow DOM) are affecting performance.

Angular 2.0 will use a very optimized version of dirty checking, see the [dirty checking design do](https://docs.google.com/a/google.com/document/d/10W46qDNO8Dl0Uye3QX0oUDPYAwaPl0qNy73TVLjd1WI/edit#heading=h.xgjl2srtytjt)c.

# Work Breakdown

Description of development phases and approximate time estimates.

1. Port the dirty checking and expression parser from AngularDart to JavaScript.
2. Port the current implementation of the templating in AngularDart to JavaScript based on this document (which contains changes/simplifications to the templating in AngularDart).
3. Build the flagship app with it (<https://docs.google.com/a/google.com/document/d/1ofxNVwEpY2xDqpSSmJIyzuaPdWKW2XTYeTPLmkeGG44/edit>)
4. Do performance measurements with the Web Components Polyfill and maybe make the use of Shadow DOM optional for Angular components.
5. Explore and refine the interoperability with Polymer/X-Tags