Design: Template Syntax Constraints and Reasoning

*Status: (Draft)*

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

*Last updated: March, 2015*

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

# Objective

This document attempts to describe the reasoning behind the template syntax and how we got there.

# Background

The syntax of Angular 1.x falls short in several categories:

* Not compatible with Web Components
* Not compatible with custom events
* Not possible to determine if the string in the attribute is a literal or expression (without understanding the semantics of active directives)
* Micro syntax (such as item in items) prevents template expression precompilation

As an explicit goal we want to be able to

* Reason about the template without knowing the semantics of the directives which are active
* Be able to clearly identify all expressions
* Be able to clearly identify location of sub-templates
* Be able to generate expression errors at compile time.
* Be able to pregenerate the expressions to allow full minification/obfuscation.
* Remain expressive in the resulting HTML
* Provide a syntax that is approachable and understandable to both professional developers and those new to Web development

# Prior Art

* [Databinding with Web Components](https://docs.google.com/document/d/1kpuR512G1b0D8egl9245OHaG0cFh0ST0ekhD_g8sxtI/edit#heading=h.xgjl2srtytjt)

# Detailed Design

Let's start by assuming that we have the following component/model representing a music artist. The artist contains a collection of Albums which in turn contain a collection of Tracks.

|  |
| --- |
| @Component({  selector: '...',  })  class Artist {  var name:string = 'World';  var albums:Array<Album> = [];  }  class Album {  var name:string;  var tracks:Array<Track> = [];  }  class Track {  var name:string;  var url:string;  } |

Here is a simple template for the above component.

|  |
| --- |
| Artist: {{name}} |

Notice that at this point we can validate the {{name}} binding at compile time because we can reflect on the Artist type and know that it has a name property. More generally we can say that the only things we can have in an expression are expressions which evaluate against the component.We know this would be an error because foo is not a field of Artist.

|  |
| --- |
| Artist: {{foo}} |

Being able to catch such errors early is an important property of the template.

## Repeater in Angular 1.x

|  |
| --- |
| Artist: {{foo}}  <ul>  <li ng-repeat="album in albums">  {{$index}}. Album: {{album.name}}  <ul>  <li ng-repeat="track in album.tracks">  {{$parent.$index}}-{{$index}}: {{track.name}}  </li>  <ul>  </li>  </ul> |

The above template may look natural, but in reality it has many issues which prevent Angular 1.x from running fast. Let's enumerate them:

1. **Microsyntax**: item in items seems very natural. The issue is that without knowing the semantics of the ng-repeat there is no way to know what the microsyntax consists of. In this case the item is the declaration of a new variable, in is a keyword and items is an expression. Because we don't know what part is an expression and which is not, it is not possible for the compiler to precompile the expression ahead of time. This has several negative implications.
   1. **speed**: we have to parse and set up watching for each instance of the repeater. It is also up to the repeater to parse the microsyntax and extract the parts. There is also a cost of unused variables slowdown. These are variables which ng-repeat creates but the template does not need. These variables are costly to compute and set up, and if they are unused they negatively impact the performance of the system. See $index, $first, $middle, $last, $even, $odd, and so on.
   2. **local variables**: ng-repeat declares local variables, but without understanding the semantics of the repeater and the microsyntax it is not possible to know what these variables are. The issue is that because any directive could potentially create any variable it is no longer possible to verify the expression for typos ahead of time. (That is because your foo symbol which is not found in controller could be injected by some directive, and hence we can not rule out that it is invalid)
2. **$parent**: because the variable created can shadow each other there needs to be a mechanism to escape the current context and read the parent variable. This hack is done through $parent.
3. **Silent Template**: ng-repeat actually creates a template of itself. This means that in production there can be zero, one or many lis. And the li can even be wrapped by the ng-repeat in other divs (i.e. transclusion). There is no way to determine by looking at a template where these templates are without understanding the semantics of all active templates.

In short without knowing which directives are active and what their semantics are it is difficult to reason about the runtime characteristics of the template. This difficulty manifests itself in understanding by developers, building tools, and optimizing for speed.

## Solving the repeat/template problem

The above section describe the issues with the current repeat, let's look at a simpler example before we revisit the Artists example above.

|  |
| --- |
| <li template def-person  bind-ng-repeat="people"  bind-ng-repeat-track-by="someFn"  bind-other-binding="someValue">  {{person}}  </li> |

In the above example there are few changes.

1. **template** attribute clearly designates this to be a template. This lets the compiler know that this element needs to be turned into a template for fast instantiation. Because it is declarative, tools and developers know that this DOM fragment can be removed, cloned, or wrapped without knowing anything about which directives are active. The template keyword is actually a shorthand for the template tag.:

|  |
| --- |
| <template def-person  bind-ng-repeat="people"  bind-ng-repeat-track-by="someFn">  <li bind-other-binding=”someValue”>  {{person}}  </li>  </template> |

1. **Binding hoisting:** Notice that we need to know which bindings need to be hoisted to the <template> tag and which stay behind on the <li> tag. This again creates an issue since one must understand the binding semantics of the template directive before one can reason about which bindings get hoisted.
2. **def-person**: The local variable person is explicitly defined using def-person. Explicit creation of variables solves the issue of not being able to verify whether an expression is valid, since only symbols which are either in the component or explicitly defined are valid symbols. Explicit variables also solve the issue with$parent and unused variables slow down.
3. Better alignment with the template tag in upcoming Web Components standard.

But this is not an ideal syntax. The reason is that the bind- prefix implies that there is a ngRepeat property on the li element. But this is not the case. When the application runs the reality is a bit different.

|  |
| --- |
| <!-- anchor: ng-repeat="people" ng-repeat-track-by="someFn">  <li def-person="...">person1</li>  <li def-person="...">person2</li> |

Notice that the ng-repeat and ng-repeat-track-by get hoisted to an anchor, while the `li tags get stripped off the repeater and the local variable declaration gets cloned along with the template but different assignments get made to it for each instance of the `li tag. Looking at a set of attributes, it is not clear which attributes will get hoisted and which will stay behind. Template-based directives are not a good fit for the bind- syntax, which binds an expression to an element property. The property on the element may be real in the case of Web Component or implicit in the case of Angular Component, but it is a property. In the case of template (such as ng-repeat and ng-if) the property is not on the element, but on the anchor. There is one property (one anchor) but many clones of the template. The bind- syntax is not a good fit here.

Let's rewrite the above in the long WebComponent syntax

|  |
| --- |
| <template def-ng-repeat="person"  bind-ng-repeat-in="people"  bind-ng-repeat-track-by="someFn">  <li>{{person}}</li>  </template>  <!-- instances -->  <li>adam</li>  <li>eric</li> |

Here all of the ambiguity disappears from the template:

1. Template is easy to identify using the <template> tag
2. The bindings are already hoisted from the <li> to the <template> and so it is clear which bindings go with the template controller and which go with the template instance.
3. Template becomes the anchor for the template controller, ie, the instances get added as siblings of the anchor.

The downside is that the above syntax is extremely verbose, (also not compatible with browsers which do not support <template> element yet.) What we need is to keep the above as the mental model and the long version (for supported browsers) but have a shorthand way of expressing the above syntax.

Instead let's propose a new hybrid syntax which solves the above issues. Let's allow the declaration of the template on any element using the template attribute. Then propose a syntax for the attribute value which would expand to the set of attributes on the <template> tag. A secondary goal would be to allow the attribute keys to be composable into DSL. An example of such syntax may be as shown below, but let's not discuss the exact syntax just yet.

|  |
| --- |
| <li template="ng-repeat #person in people; trackBy someFn">  {{person}}  </li> |

Notice that:

1. A template directive must be declared inside of a template attribute. This makes it clear that it is a template.
2. Pseudo micro syntax based on keywords, symbols and expressions. Where the expressions, keywords and symbols are clearly identifiable.

The above gets expanded to the long syntax as shown here.

|  |
| --- |
| <template def-ng-repeat="person"  bind-ng-repeat-in="people"  bind-ng-repeat-track-by="someFn">  <li>{{person}}</li>  </template> |

## Template Syntax

The goal of template syntax should be:

* Easy/predictable mapping from the syntax to the attribute mappings on the <template> (hoisting)
* Allow building simple DSLs in the form of `foreach item in items`

The templates are made up of three parts:

* Keywords: which will map back to template attribute names.
* Expressions: which will map back to template attribute value.
* Symbols: which will allow creation of new symbols.
* Delimiters: things such as `,`, `;`, `:` which will ease in reading the syntax, but which have no semantic meaning.

Let's look at simple definitions:

|  |  |  |
| --- | --- | --- |
|  | **template=""** | **<template>** |
| symbol mapping | ng-repeat #item | def-ng-repeat="item" |
| expression mapping | in expression | bind-ng-repeat-in="expression" |

Here are some examples:

* Basic example:  
  <div template="ng-repeat #person in people">  
  <template def-ng-repeat="person" bind-ng-repeat-in="people">
* Using Track by:  
  <div template="ng-repeat #o in objs; trackBy: fn">  
  <template def-ng-repeat="o" bind-ng-repeat-in="objs"  
   bind-ng-repeat-track-by="fn">
* Declaring additional variables  
  <div template="ng-repeat #o in objs; index: #i, first: #f">  
  <template def-ng-repeat="o" bind-ng-repeat-in="objs"  
   def-ng-repeat-index="i" def-ng-repeat-first="f">

The above syntax consists:

1. Alternating keywords and (symbol or expression)
2. Keywords can optionally be followed by `:` which has no semantic meaning
3. Keyword-symbol-expression pairs can be separated with empty space ` `, comma `,` or semicolon `;`.
4. Symbols must start with `#` and their semantic meaning that a new variable is being declared by the template.
5. Expressions have no prefix.
6. Proper use of template tag: ([HTML new template tag](http://www.html5rocks.com/en/tutorials/webcomponents/template/))

|  |
| --- |
| template\_syntax := [keyword(:)? [#symbol|expression](,|;)? \*]+ |

## Rewriting the example

The example above can now be rewritten in this syntax:

|  |
| --- |
| Artist: {{foo}}  <ul>  <li template="ng-repeat: #album in albums; index: #i">  {{i}}. Album: {{album.name}}  <ul>  <li template="ng-repeat: #track in album.tracks; index: #j">  {{i}}-{{j}}: {{track.name}}  </li>  <ul>  </li>  </ul> |

Benefits:

* Clear demarcation of all templates
* Explicit declaration of new template variables
  + Solves nested repeater `$parent` problem
* Pseudo microsyntax to keep complex such as ng-repeat compact and easy to read.

## Referring to other components

There are times when it is necessary for an event handler to refer to other portions of the page, as in this example when we want to enable playing the current track and showing the current location in the track.

|  |
| --- |
| Artist: {{foo}}  <ul>  <li template="ng-repeat #album in albums; index: #i, even: #e">  {{i}}. Album: {{album.name}}  <ul>  <li template="ng-repeat #track in album.tracks index: #j">  {{i}}-{{j}}: {{track.name}}  <button on-click="**player.src=track.src;**  **player.play()**">play</button>  </li>  <ul>  </li>  </ul>  <media-player **var-player**></media-player> {{**player**.head | time}} |

Things to notice

* Explicit declaration of new variable using `var-player`. Notice that the variable is visible to the whole template, (not just the element children)
* An `on-click` event can write to properties on the reference as well as invoke methods on the `player`.
* The `player` can become a source of data binding as in the case of showing the current location of the head in the playback {{player.head | time}}
* The first keyword is special in the sense that it will be prefixed to all subsequent keywords.

## Other Examples

|  |
| --- |
| <div template="ng-if: expression">show</div>  <div template="ng-switch: expression">  <div template="ng-case: 'foo'">foo</div>  <div template="ng-default">foo</div>  </div>  <tabs>  <pane name="overview">  Some Text  </pane>  <pane template="ng-repeat: #pane in panes" name="{{}}">  {{pane.text}}  </pane>  <template template="ng-repeat: #pane in panes" name="{{}}">  <pane>  <template>  {{pane.text}}  </template>  </pane>  </template>  </tabs> |

## Short Hand

There is some debate whether we should also support a shorthand syntax of the above. TBD

# Summary

|  |  |  |
| --- | --- | --- |
| **Description** | **Canonical** | **Short** |
| Text binding | <div bind-text-*index*=exp>  <div  bind-text-0="a|str"  bind-text-2="c|str">  \_<b>x</b>\_  </div> | <div>{{exp}}</div>  <div>  {{a}}<b>x</b>{{c}}  </div> |
| Property binding | <div bind-foo="exp|str">  <div bind-class="selected|str"> | <div foo={{exp}}>  <div class="{{selected}}"> |
| Property binding | <div bind-*prop*=*exp>*  <div bind-hidden="true"> | <div [*prop*]=*exp*>  <div [hidden]="true"> |
| Event binding | <div on-*event*=*exp*>  <div on-click="doX()"> | <div (*event*)=*exp*>  <div (click)="doX()"> |
| Event bubble binding | <div on-bubble-*event*=*exp*>  <div on-bubble-mouseover="hlite()"> | <div (^*event*)=*exp*>  <div (^mouseover)="hlite()"> |
| Declare reference | <div def=*symbol*>  <player def="fooBar"> | <div #*symbol*>  <player #foo-bar> |
| Declare template reference | <template def-*directive*=*symbol*>  <template  def-ng-repeat="item"  bind-ng-repeat-in="items"> | <template #*directive*=*symbol*>  <template  #ng-repeat="item"  [ng-repeat-in]="items"> |
| Designating a template | <template><div></div></template>  <template bind-ng-if="visible">  <li bind-class="classes">  Some content  </li>  </template>  <!-- instance: may be here -->  <li bind-class="classes">  Some content  </li> | <div template>  <li  template="ng-if: visible"  bind-class="classes">  Some content  </li> |
| Template syntax | *[keyword:?*  *[#symbol|exp](,;)?*  *]+* |  |

## Summary: Prefix of HTML attributes

This table summarizes the kinds of prefixes that the developer has to be aware of.

|  |  |
| --- | --- |
| **Description** | **Canonical Prefix** |
| Bind to property | **bind-***property\_name* |
| Bind to event | **on-**event\_*name* |
| Declare template variable (scoped to sub-template) | **def-***directive*="*symbol*" |

This table summarizes special attributes which developer has to be aware of.

|  |  |
| --- | --- |
| **Description** | **Canonical Prefix** |
| Declare variable (scoped to whole template) | **def**="*symbol*" |
| Declare sub-template | **template**="hoist content" |

The meaning of each of these attributes is well defined and there are easy to explain, and universally true regardless of the presence or semantics of the directives.

# Caveats

You may need to describe what you did not do or why simpler approaches don't work. Mention other things to watch out for (if any).

# Security Considerations

How you’ll be secure

# Performance Considerations / Test Strategy

How you’ll be fast.

# Work Breakdown

Description of development phases and approximate time estimates.

# Counterproposals

Here is a section where you can add a counterproposal. Please take the template, make a copy of it below and then fill it out.

Syntax Proposal Template

### Example of syntax

|  |
| --- |
| <div some syntax here>  </div> |

### Explanation of Reasoning

Explain why you think this is better

### Scoring

Explain how the syntax correctly passes all of these tests:

|  |  |  |
| --- | --- | --- |
|  | **Test** | **Explanation** |
| A | Directive Semantics Independent | Needs to be able to reason about the behavior of the templates without knowing anything about which directives are active and what their semantics are. |
| B | Expressions | Clearly identify all expressions. |
| C | Variables | Clearly identify all of the exported variables |
| D | Template | Clearly identify all of the template boundaries where the DOM structure can mutate. |
| E | Hoisting | If template, need a clear way to know which bindings go with:   1. clone of template and 2. which go with the cloning directive |
| F | WebComponents | Must not allow web-component to be able to read the binding as literal, only as resolved value. |
| G | Expressiveness | Remain expressive and approachable in the resulting HTML |

Angular's Team Proposal

### Example of syntax

|  |
| --- |
| <div template="ngRepeat: #item in items; trackBy: someFn"  bind-foo="expr">  </div>  <!-- expands to -->  <template def-ng-repeat="item" bind-ng-repeat-in="items"  bind-ng-repeat-track-by="someFn">  <div bind-foo="expr"></div>  </template>  <!-- cloned instances -->  <div bind-foo="expr"></div>  <div bind-foo="expr"></div> |

### Explanation of Reasoning

Explain why you think this is better

### Scoring

Explain how the syntax correctly passes all of these tests:

|  |  |  |
| --- | --- | --- |
|  | **Test** | **Explanation** |
| A | Directive Semantics Independent | yes, you can reason without knowing anything about which directives are in the template or how they work. |
| B | Expressions | yes, all expressions are in {{}} or in attributes which are prefixed by bind-. |
| C | Variables | yes, in attributes which are prefixed by def-. |
| D | Template | yes, either explicitly using <template> element or implicitly with template attribute. |
| E | Hoisting | yes, only the contents of the template attribute needs to be hoisted. |
| F | WebComponents | yes, literal are in attributes, expressions are in attributes prefixed by bind-. |
| G | Expressiveness | yes, retains the existing ngRepeat concept and “in” syntax. Readily readable by any level of developer.  no, requires understanding of “template” and when “template” needs to be used instead of a simple HTML element. |

### [*Yet Another Syntax Proposal (@sheerun)*](https://docs.google.com/document/d/1-a--8tg9inFwiTgGHGONjmkYMFil6TPmAXJSTH6BKJc/edit?usp=sharing)[misko: REJECTED, it fails most of the tests.]

[Angular 2 Binding -- Attribute Sections (@jonrimmer)](https://docs.google.com/document/d/14WIvcxA-KwM372a_wKGJK0jX6ltSxY9gfwCze-Q66Cc/edit?usp=sharing)

[An angular 2 template syntax proposal inspired by ES6](https://docs.google.com/document/d/1f0gRhFKSzcT20VwG34RNpvwJznXVFS9n7w_-v_2_QhM/edit#heading=h.lyxn49pndah4)