Databinding with Web Components

*Status: (Draft, Final)*

*Authors:* [*misko@hevery.com*](mailto:misko@hevery.com)*;* [*rob@bluespire.com*](mailto:rob@bluespire.com)

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

Objective: We will look at the primary considerations involved in designing a databinding system that works with Web Components and Angular coexist. This includes a discussion of both technical issues and stylistic opinions.

# Background

The term *Web Components* usually refers to a collection of four related specifications:

* [Custom Elements](http://w3c.github.io/webcomponents/spec/custom/) - Enables the extension of HTML through custom tags.
* [HTML Imports](http://w3c.github.io/webcomponents/spec/imports/) - Enables packaging of various resources (HTML, CSS, JS, etc.).
* [Template Element](http://www.w3.org/TR/html5/scripting-1.html#the-template-element) - Enables the inclusion of *inert* HTML in a document.
* [Shadow DOM](http://w3c.github.io/webcomponents/spec/shadow/) - Enables encapsulation of DOM and CSS.

By combining these four capabilities web developers can create declarative components (Custom Elements) which are fully encapsulated (Shadow DOM). These components can describe their own views (Template Element) and can be easily packaged for distribution to other developers (HTML Imports).

When these specifications become available in all major browsers, we are likely to see developer creativity explode as many endeavor to create reusable components to solve common problems or address deficiencies in the standard HTML toolkit.

## Prior Art

There are a number of libraries that have offered similar technology to that of Web Components. Here are a few of the noteworthy entries in this space:

* AngularJS v1 https://angularjs.org/
* AngularDart https://angulardart.org/
* KnockoutJS http://knockoutjs.com/
* Polymer https://www.polymer-project.org/

The first three in this list are a bit different from the last. Polymer differs in that it actually uses native Web Components technology if present or falls back to polyfills. The other libraries use “home grown” solutions to emulate components.

It is not clear at this time whether native Web Components actually perform better than custom solutions. In all cases above, the custom solutions provide greater capabilities than that of Web Components. Even Polymer adds additional functionality on top of its core Web Components-based design to make up for features it deems are missing. It may be that by the time Web Components are made “workable” by adding features like databinding, that they offer no significant advantage over custom solutions. At this time, we just don’t know for sure.

Another difference, due to Polymer’s Web Component-first design, is that its data binding system can work with any Web Component. There are some drawbacks based on how it’s implemented though. (Those can be determined by reading further. For additional information, see [this document](https://docs.google.com/a/bluespireconsulting.com/document/d/16O2Im1ekfdJ4FU8FBbVRYGjqsXjmcV3tYFg1vyfhYC8/edit#).) The data binding systems in the other three libraries are hard-coded to known HTML elements. These libraries aren’t currently prepared to deal with Web Components. The reasons for this are part of the subject of the rest of this document.

# Design

As mentioned above, Angular does not know about Web Components. Furthermore, Angular should not distinguish between native HTML elements and Web Components. If this were not the case, it would imply that all Web Components would need to publish themselves to Angular. This is obviously backwards. When a Web Component is registered on the page, from the point of view of Angular, it should be like any other element. As far as Angular is concerned there should be no difference between <button> and <fancy-button>. The fact that the former is a built-in element and the latter is a Web Component should in no way influence the way Angular treats or binds to the element.

In order to build a databinding system which can support the above requirements, we’ve concluded that the following three constraints must hold true:

1. HTML attributes are used only to initialize components.
2. All bindings manipulate element properties, not their attributes.
3. The HTML attribute name of a databound attribute must be *escaped* in some way.

*Read on to hear how we came to these three conclusions...*

## Problem: Databinding to Attributes

Suppose we want to bind to button in order to dynamically set whether it is disabled or not. We might write something like this:

|  |
| --- |
| <button disabled="{{isDisabled}}" title="{{myTitle}}">OK</button> |

Notice that we have two bindings: disabled and title. It turns out that this will not behave the way you might expect. The reason for this is that disabled is what is known as a *boolean attribute*. For this type of attribute, its text value is not what matters, but rather whether or not the attribute itself is present. To understand this better, have a look at the following possibilities:

|  |
| --- |
| <button disabled="true" title="Accept">OK</button> <- disabled  <button disabled="false" title="Accept">OK</button> <-- disabled  <button disabled="" title="Accept">OK</button> <-- disabled  <button title="Accept">OK</button> <-- enabled |

Notice that only the last button is enabled. Unfortunately, we cannot get to this state by using the attribute-based databinding described above. The trouble is that there is no way for Angular to know that the disabled attribute is a *boolean attribute* or that it follows a different set of rules from other attributes. Even if we special case the built-in boolean attributes, a Web Component can still declare its own set, which the framework would not know about.

This is not the only issue with binding to attributes. There is a second problem: attributes can only accept string values. Something like this will not work at all:

|  |
| --- |
| <map tile-source="{{tileDataSource}}">OK</map> |

In this example, the map Web Component needs a reference to a tile-source object which has some API and cannot be represented as a string.

*How are we to make databinding work correctly in these scenarios?*

## Solution: Databinding to Properties

Instead of databinding to element attributes, we can databind to element properties. If we revisit the button example from above, using the same syntax, but with new semantics, we will get the desired runtime behavior. Here’s the markup again, for convenience:

|  |
| --- |
| <button disabled="{{isDisabled}}" title="{{myTitle}}">OK</button> |

This time we won’t rely on setting attributes. Instead we will write a boolean true/false value to the button’s disabled *property*. This will do the right thing, regardless of whether the attribute is boolean or not. It also allows the use case of passing objects by reference, such as in the second example from above. So, by manipulating properties rather than attributes, we solve both problems.

### Bonus: Fewer Angular Directives Required

It turns out that many Angular directives are just shallow front ends to underlying properties on elements. Here is a list of directives which would no longer be needed if we bound to properties.

|  |  |  |
| --- | --- | --- |
| ng-bind | [text-content] |  |
| ng-bind-html | [inner-html] |  |
| ng-bind-template | text-content=  "{{foo}}" | which will get translated to  [text-content]="foo|stringify" |
| ng-class | [class]  [class-list] |  |
| ng-hide  ng-show | [hidden] |  |
| ng-href | [href]="expr" |  |
| ng-disabled | [disabled] |  |

## The Role of Attributes

Here’s how we see the relationship between attributes and properties: an element uses attributes to initialize itself at the time of instantiation, but after that it uses properties for all runtime interaction. To see an example of this, let's look at an input element.

|  |
| --- |
| <input type=text value="Hello"> |

In this case, the input's value *attribute* will be used to initialize its value *property* to "Hello". Afterwards, all changes are only visible in the value *property*, not in the attribute. Some properties may choose to [reflect back to attributes](http://dev.w3.org/html5/spec-preview/common-dom-interfaces.html#reflecting-content-attributes-in-idl-attributes), but that is up to the component and does not affect the data binding layer.

Existing Web Components ([Polymer](http://www.polymer-project.org/docs/elements/), [Brick](http://mozbrick.github.io/)) already expose their API through properties and initialize from attributes. When writing Vanilla JS, it is natural to work with properties, rather than with attributes. The expectation is that Web Component implementations will follow native components in their behavior. After all, the point of Web Components is to extend the browser’s native capabilities in a way that looks and feels the same. ([Polymer](http://www.polymer-project.org/docs/elements/) and [Brick](http://mozbrick.github.io/) already follow this guideline.)

So, now you can see how we arrived at our first two conclusions above:

1. HTML attributes are used only to initialize components.
2. All bindings manipulate element properties, not their attributes.

*But what about our third conclusion? Why do we need to escape attribute names?*

## Attribute Escaping

Because Angular does not know which elements are native and which are Web Components, the syntax for databinding must take care not to interfere in any way with normal Web Component behavior. Let’s see an example to help clarify this.

|  |
| --- |
| <pane title="{{exp}}"></pane> |

In this example, the pane, upon instantiation, would read the title attribute and render its title as "{{exp}}". This is because the data binding layer is *above* the Web Component and *not* *part of* the Web Component. Once the Web Component is instantiated the Angular databinding layer would evaluate exp and assign the result to the title property. The effect of this process is that the Web Component would first flash the expression binding literal “{{exp}}” and then be replaced by the actual expression value. This is clearly not the desired behavior.

*NOTE: The above description of WebComponent behavior may not be accurate. If the WebComponent is contained within a <template> element, it appears that the upgrade process does not happen until after the template is instantiated. This would make sense if templates are supposed to represent truly dormant html. We need to test this further as it has implications on design.*

There are several ways to solve the issue of Web Components getting ahold of the raw binding. Let's discuss the possibilities and their implications:

1. Web Components could be aware of the databinding syntax.
   1. **DOWNSIDE:** This breaks encapsulation. Web Components should be framework agnostic, and frameworks should be able to choose any syntax without telling the Web Component.
2. Angular could remove the bindings from the template during compilation.
   1. The resulting DOM in the debugger would look like this: <pane></pane>.
   2. Even though no attributes are present in the DOM, the underlying data binding system knows that it needs to update the title property when the expression changes.
   3. **DOWNSIDE:** It is difficult to debug the application using the web-tools, since the bindings are all hidden from the developer.
3. Angular could improve on the previous strategy by rewriting the template at runtime to show the bindings:
   1. The resulting DOM in the debugger would look like this: <pane X-title-X="{{exp}}"></pane>. (Let's discuss syntax later.) It is important to realize that the only way to add debugging information to the Web Component element and not have the Web Component *read* the attribute is to munge the attribute *name*. Any form of syntax based on escaping the attribute *value*, will cause the issue which we are trying to avoid. In this example we rename title to X-title-X.
   2. **DOWNSIDE:**
      1. The syntax of the HTML in the inspector does not match the syntax available to the developer during application development.
      2. The template rewriting will have an unnecessary performance impact during template processing as bindable attributes are rewritten. (This may be mitigated by doing it only in debug mode.)

Given that we cannot have the property name naked in the DOM, and that we don't want to have different syntax during debugging and HTML authoring, it seems that having the developer write the template in escaped attribute mode is probably most consistent.

|  |
| --- |
| <pane X-title-X="{{exp}}"></pane> |

Note: Syntax intentionally ugly. See the syntax discussion below.

This is what the author of the HTML template writes and what would be present in the DOM via the inspector. Hopefully it is clear how we arrived at our third conclusion from above: *The HTML attribute name of a databound attribute must be escaped in some way.*

### Events

Web Components, like native components, can fire events in response to user input. Native components solve this with on*event* attributes. For example:

|  |
| --- |
| <div onclick="..."> |

The on*event* attribute has many issues (security and global execution context for starters), which is why most frameworks implement their own event handling mechanism. In Angular 1.x this was handled by a dedicated set of directives such as ng-click. Unfortunately, this is not scalable for Web Components since they can invent any new event name. Can you imagine having to create a custom directive for every event of every Web Component? In addition, the registration of event handlers is different from that of assigning to a Web Component property. Property binding and event listening are two fundamentally different aspects of databinding. For this reason we need a way to execute an expression based on an event. Following our previous conclusion, we should encode the event name in the attribute somehow, but in a way that distinguishes it as an event rather than an attribute. Here’s an example:

|  |
| --- |
| <div X-ON-click-X="..."> |

## Note: Syntax intentionally ugly. See the syntax discussion below.

## Syntax

Here’s a quick summary of what we must achieve with our syntax:

1. Databinding: The property name must be *encoded* in the attribute. The binding expression is in the attribute’s value.
2. Event Binding: The event name must be *encoded* in the attribute. We must also be able to distinguish it from an attribute. The event expression is in the value.

Given the above rules, here are couple of proposed syntaxes for databinding.

|  |  |
| --- | --- |
| <web-component  bind-title="exp"  on-close="exp()"> | The simplest syntax proposal is to prefix the property name and event name with *bind-* and *on-* respectively. There are two downsides. First, not all properties are nouns/adjectives. Some, such as 'if' or 'switch', are conjunctions or verbs, making 'bind-ng-if' and 'bind-ng-switch' awkward. That’s also fairly verbose. The second issue is that this may make it harder to see the attribute/directive name. For ltr reading, your eye first comes across the prefix. Some developers may prefer to see the attribute more clearly before the metadata describing whether or not it’s bound. |
| <web-component  title@="exp"  close+="exp()"> | The next proposal is to use a non-pronounceable character such as @ and + for properties and events. The upside is that such characters are silent. They convey meaning, without forcing specific pronunciations. They also allow the attribute/directive name to remain most prominent. The downside is that, while special characters are allowed [according the HTML spec](http://www.w3.org/TR/html-markup/syntax.html#syntax-attributes), editors, syntax highlighters and HTML validators may not treat them as valid. |
| <web-component  [title]="exp"  (close)="exp()"> | Here’s another variation based on symbols, inspired from JS object notation. The [] are used to dereference properties and () are used to invoke functions. The downside is that the attribute now has both a prefix and suffix. Additionally, turning the static literal <div title="foo"> into the dynamic binding <div [title]="foo"> requires more editing keystrokes. |
| <web-component  title[]="exp"  close()="exp()"> | Here’s an alternate version of previous syntax which is more keystroke friendly. |

Note: There is still an issue not addressed by any of these syntax options. None of them convey the *directionality* of the binding. If we have a single bind operator for properties, does that represent one-way binding? or two-way binding? You cannot infer directionality since it can vary not only per property, per component but also per use. You need a specific way to indicate that a binding is intended to be one-way (from model to view only) or two-way (from model to view and view back to model). This is an area of active debate amongst team members at present. Some are arguing for the removal of two-way binding in favor of an (not yet specified) alternate mechanism. Others favor syntax to uniquely represent one-way and two-way binding.

### What about the {{}} Syntax?

So many libraries are using the {{}} syntax, but can we keep it? Yes. However, there are two issues that need to be addressed:

1. As described above, any escaping which relies solely on attribute *value* escaping will allow the Web Component to read the raw binding string. This means that the {{}} will have to be converted to attribute name escaping or removed during compilation.
2. The {{}} can only bind strings. To understand this better, let’s assume we have a piece of template such as this:  
    <map tile-source="{{tileDataSource}}">  
   Assuming that the binding has nothing before or after {{}} we could pass by reference instead of by string. So, the above should work fine. But the moment a character is added it would fail (notice the extra space before {{}} below).  
    <map tile-source=" {{tileDataSource}}">  
   This may not seem too bad, but the corollary would also be true. Assume that you have a type called User which has a toString() method. In that case, this would work as expected:  
    <dialog title="Hello {{user}}">  
   However, this would break:  
    <dialog title="{{user}}">  
   The issue is that, since there is nothing before or after the {{}}, we would silently drop the call to the toString() method, resulting in a pass by reference. Here’s the important point: for the system to be consistent, we cannot change the semantics of the binding based on whether or not there is anything around the binding. The semantics must either always have a toString() call, or never.

In conclusion we can make {{}} work in most cases, if we rewrite the templates or remove the attribute and always convert to string.

Note: While this syntax is used by other binding engines, it should be clear that it is not usually interpreted by these engines as a stringification of the data in the way that we want to enforce. So, while the syntax is the same, for us, the semantics are different. In order to make the string-nature of our syntax more apparent, we should consider using the ES6 string interpolation notation instead. It would provide a couple of benefits:

1. It will be obvious that this type of binding includes a toString operation. This will help clarify the behavior for developers coming from other libraries that use the same syntax, but with different or inconsistent behavior.
2. It will align us with ES6 syntax and enable the developer to have less cognitive dissonance between their programming language and markup.

The resulting syntax would look like this:

|  |
| --- |
| <dialog title="Hello ${user}"> |

The above is really just a shorthand to:

|  |
| --- |
| <dialog [title]="'Hello ' + (user|stringify)"> |

### The Oddity of Template Directives

There are a special set of directives which are known as *template directives*. Examples of these are ng-if, ng-repeat and ng-switch. These directives are unique in two ways. First, there can only be one template directive per element. The following is not allowed:

|  |
| --- |
| <div ng-repeat ng-if> |

This is confusing as it is not clear if the ng-if should get applied before ng-repeat or if ng-if only works after ng-repeat is unrolled. Second, the directive itself exists outside the element on which it is declared. Consider this:

|  |
| --- |
| <div ng-if> |

It would be incorrect to think of ng-if as part of the div. If ng-if evaluates to false the div is removed entirely. It is better to think of ng-if as something which manages the div and is above/around it rather than part of it.

Because these directives represent structural differences unlike normal attributes or decorator directives, the question arises “Should we have a special syntax?” To discuss this further, we’ll make some assumptions about the general binding syntax. Let’s assume the following is used for standard property bindings:

|  |
| --- |
| <div [property]=exp> |

We could then use a variation like one of the following to represent template bindings:

|  |
| --- |
| <div directive[]=exp> <div directive#="condition"> |

Here’s some examples of how we might represent the identifier needed by ng-repeat:

|  |
| --- |
| <div ng-repeat[person]="people">{{person}}</div> <div ng-repeat#person="people">{{person}}</div> |

If we don’t want to have different binding syntax for template directives, we still need a way to handle the ng-repeat scenario. Here are a few ideas:

|  |
| --- |
| <div [ng-repeat]person="people">{{person}}</div> <div [ng-repeat|person]="people">{{person}}</div> |

Notice that each of these ideas involves encoding more data into the attribute name. But that is not the only option available to us. We could also use a *microsyntax* such as what is found in Angular 1.x. For ng-repeat, we could then have:

|  |
| --- |
| <div ng-repeat="person in people">{{person}}</div> |

The *microsyntax* is much more natural for those familiar with Angular 1.x or any language with list comprehensions. It’s also more flexible, allowing each directive the opportunity to define it’s own syntax if necessary. Of course, this sort of power can be easily misused or abused. If used in a limited way though, it seems simpler to visually parse and understand than the alternative of embedding this information in the attribute. The issue with microsyntax is that it will make it hard for IDE's to correctly interpret what is in the value of the binding.

*This aspect of the binding syntax is still being heavily debated internally.*

# Conclusion

We have the following basic constraints, their reasoning having been explained above:

1. HTML attributes are used only to initialize components.
2. All bindings manipulate element properties, not their attributes.
3. The HTML attribute name of a databound attribute must be *escaped* in some way.

We’ve been over a variety of different styles or opinions on how to escape the attribute and/or embed data in it. The current majority vote on binding syntax amongst team members is as follows:

### Standard Bindings

|  |
| --- |
| <date-picker [value]=”*expression*” (change)=”*expression*”> |

### Template Bindings

|  |
| --- |
| <div [ng-repeat|person]="people">{{person}}</div> |

Note: One of the authors of this document (Rob) holds two dissenting opinions. First, he’s not willing to let two-way databinding go without seeing a robust alternative presented, which has not been done at this point. If no alternative can be found, the general binding syntax may need to be revised to include directionality. Second, he does not believe that details of the templating system should be conflated with the binding system. Thus, he is not in favor of embedding additional data into the attribute. He would rather support a microsyntax that is transparent to the binding system, but handled by the template compiler. Such a system would convert the microsyntax into standard bound properties. This would open up a lot of possibilities for developers. However, its potential danger for misuse/abuse should be stressed to the community in order to prevent a proliferation of custom syntax. As an alternative to microsyntax, the expression language could be extended to include the idea of “binding options” which could allow for the specification of a local variable, directionality, triggers, etc.

…….