Well Behaved Web Components

*Status: (Draft)*

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

*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 will argue that WebComponent should:

* Behave like a native DOM element
  + Be usable from vanila JS.
  + Use Attributes for initialization
  + Use Properties for communication
  + Only fire events on user interaction
* Web components should be idempotent and asynchronous to forward(model -> component) bindings.
* Web components should deliver reverse (component -> model) updates synchronously.

# Background

There are a lot of technical specification documents describing what the WebComponent API is, but there seems to be lack of a document which describes what a "Well behaved WebComponent is". This document tries to answer that question.

# Prior Art

* AngularJS / AngularDART - we have going a lot of experience building data binding systems while building Angular and supporting large scale applications with it.
* Polymer / Brick - we have studied existing web-components and tried to integrate them with Angular, the learnings of these integrations are included here.

# Detailed Design

This document will be broken down to several Axioms. Each axiom will have supporting arguments.

## WebComponent must be usable in VanillaJS

A WebComponent allows one to extend the vocabulary of HTML elements which are available to the developer. If the HTML's <input type=text> is not working for you should be able to create <fancy-input>. From this it follows that the way one interacts with native components or web-components should be identical. By identical we mean that the web-component can be replaced by the native component and nothing should change in the way one interacts with it.

To interact with native component the element provides these four API surfaces:

* Attributes: the attributes are used to deserialize the element from HTML to DOM. They are the only serialization mechanism available to us.
* Properties: At runtime the JS can read/write to the DOM properties
* Methods: At runtime the JS can invoke methods on the element
* Events: At runtime the JS can listen to events on the element.

Here is a typical way of using the native elements.

<input id='myInput' type="text" value="World">

var input = document.getElementById('myInput');  
input.focus();  
input.addEventListener('change', function() {  
 alert(input.value);  
 input.value="";  
});

In this example the initial HTML contains an attribute value which is set to World string. Attributes only support strings. Once the HTML is deserialized into DOM, JS can get a reference to the Element and invoke methods on it such as focus(). In order to get notified of changes the JS can register an event handler. Events can fire due to external changes (in this case user entering new data. )

The JS can also assign values to the properties of an element. The properties can accept any value, not just strings. Notice that events only fire due to external changes. Assigning a blank value to input.value does not cause the change event to be fired again.

Notice also that writing a value to input.value does not reflect the value back to the attributes. But writing input.title does. Which properties get [reflected back to the attributes](http://dev.w3.org/html5/spec-preview/common-dom-interfaces.html#reflecting-content-attributes-in-idl-attributes) is the choice of the component. The usual rule is that only attributes which could be used for CSS selector styling get reflected.

### WebComponent

* Should provide behaviors as methods on element (ie input.focus())
* Should use attributes to initialize its properties upon instantiation.
* Should assume that at runtime the properties are the primary mechanism of data communication.
* Should fire events upon interactions with outside world, but not when property is set programmatically or method is called. (ie The source of the event must be external in nature)
* May chose to reflect some properties back to attributes.
* Should NOT expect that it will be instantiated in any particular framework, so that it can be used from vanila.js.

**Web Components and forward data-binding**

A web-component (or native component) may be instantiated in a framework. A web component can not play favorites which framework it should be instantiated in. This implies that a web-component can not expect/create external API other than the ones described above. (Internally a web-component can use whatever it wants.)

Data-binding is an asynchronous operation. Data-binding is a projection of application state to the UI. As such the projection needs to be idempotent. This has several implications. Think how <input type=text> would behave when we would set the value property on it.

* A UI framework may bind the same value multiple times.
* A UI framework may coalesce intermediary values and only do a single update on the end, or even skip updates and only do an update on requestAnimationFrame.

Idempotence of bindings has an implication that a property write can not be changed by the Web-Component, (only external event can change a property).

Imagine that you would have a <select> with several <option>s in it. Now imagine this sequence of operations.

1. select.selectedValue = 'foo'
2. XHR returns with option list which are written as select.addOption(...);

Because of requestAnimationFrame coalesces bindings it is possible that the options get added either in a single VM turn or across two separate VM turns. If the bindings are not coalesced than selecting a not yet written option results in an inconsistent state for the <select> component. It would be wrong for the component to try to correct the selectedValue by resetting selectedValue to null for example, and firing an event (or two way data-binding mechanism) back to the application. The issue is that application would behave differently depending on whether or not the two writes got coalesced or not. The proper behavior is for the <select> component to honor the selectedValue regardless of whether it is valid or not and render appropriately. The <select> component can change the selectedValue only due to an interaction from the user. The other way to think about it is that the web-components are just a projection of the state of the application, and as such need to be inert, they can not impose their opinions on the application.

A web-component which breaks this rule will have unexpected behavior in different data-binding frameworks.

**Web Components and reverse data-binding**

A reverse data-binding is a process of collecting changes from the external events (such as user interaction with UI or XHR) and delivering them to the application. As described above forward data-binding is the way the data is projected to the user. Forward data-binding must be idempotent and asynchronous. Reverse databinding is synchronous and non-idempotent. This asymmetry is very surprising, so let's look at why.

An application may behave differently if the user enters characters 'a' followed by 'b' vs 'a', 'x', backspace, 'b'. For example typing application may want to keep track of errors. For this reason coalescences is not possible in reverse data-binding. Every action user takes may matter in some application. Furthermore the reverse data-binding events need to be delivered synchronously. It may be that entering 'a' followed by 'x' puts the UI in a mode, where the interpretation of backspace and 'b' will be treated as totally different input.

### Empirical Data

In AngularJS and AngularDart the forward and reverse databinding are idempotent, non-coalesced and asynchronous, ie symmetrical. This seems cute for making hello world applications, but in large scale system it creates problems. The biggest of which is that we never know when we are done rendering. We can not answer a simple question as when is it safe to do a DOM read, since any operation can potentially change model which will get propagated forward/reverse to the rest of the system. Breaking the symmetry solves this as it allows us to clearly define the synchronous reverse binding phase and asynchronous forward phase.

I believe that Polymer team came to the same conclusion, since node.bind delivers forward bindings asynchronously but delivers reverse bindings through a synchronous callback.

### Changing the Model

A web component can get a hold of a complex model as an assignment to one of its properties. Since reverse bindings must be synchronous, the web-component is not allowed to write to the model properties, but can write to the model through a method call, since that would give the model a chance to intercept the method call and perform synchronous operations on it.

### WebComponent

* Should only change property due to external event.
* Should honor invalid input, as it may become valid later
* Can not change the property due to non external causes.
* Should not write to any properties on the model object.
* Can call methods on the model object
* Should deliver the changes to the property in a synchronous fashion (ie through an event)

# Putting it all together

### WebComponent

* Should provide behaviors as methods on element (ie input.focus())
* Should use attributes to initialize its properties upon instantiation.
* Should assume that at runtime the properties are the primary mechanism of data communication.
* May choose to reflect some properties back to attributes.
* Should NOT expect that it will be instantiated in any particular framework, so that it can be used from vanilla.js.
* Should only change property due to external event.
* Should honor invalid input, as it may become valid later
* Can not change the property due to non external causes.
* Should deliver the changes to the property in a synchronous fashion (ie through an event)

The implication of the above is that we don't need two way data-binding, because forward and reverse bindings are asymmetrical. Instead frameworks may provide forward data-binding and reverse data-bindings are done through DOM events only. Since reverse data-binding only happened due to external events, the slowness of the DOM events is not an issue.

# 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.