# WebComponents and Angular

*Status: (Draft)*

*Authors: misko@google.com*

## Objective

Describe how Angular will work with Web Components, both consuming (interacting with) as well as producing (defining) custom elements.

More specifically the goal of this proposal is to:

1. allow Angular to consume any Web Components without resorting to some Angular unique APIs and or conventions. (Proposal for other frameworks require that component be aware of framework specific API )
2. allow Angular components to be packaged as Web Components and be used in other frameworks or in vanilla JavaScript.

# Background

WebComponents is not a single standard but rather an umbrella standard for:

- Shadow DOM
- HTML templates
- HTML Imports
- Custom Elements
- DOM Mutation Observers

Together these standards aim to allow a component designer to extend the browser's built in vocabulary of Elements in a way which is indistinguishable from the built in set of elements. *These elements are known as custom elements.*

# References

- Angular v2.0 templating design document.
- Polyfils: Shadow DOM, Custom Elements, Mutation Observers,
- X-Tags: A mini-framework for building custom components on top of web-components polyfils.
- Polymer

# Detailed Design

## Custom Element Interface

The core premise of Web Components is that everything is a DOM element. Custom elements should be indistinguishable from browser provided elements. Today DOM elements' APIs consist of:

1. DOM events
2. DOM attributes
3. Element properties
4. Element methods

For custom element to behave as browser built-in elements, the custom element API must be limited to what the web already has, which is events, attributes, properties and methods. *Let's refer to the above API surface as the Custom Element Interface.*

There are two sides to the interface. The consumer and the implementor.

### Interface Consumer

To explain what we mean by interacting with custom elements, here is an example of code which uses the interface as a consumer.

```
var element = ...; // Get an element from someplace.

// Example of using DOM attributes;
element.setAttribute('title', 'hello world');

// Example of DOM event
element.addEventListener('change', ...);

// Example of using Element property.
element.value = 'today';

// Example of Element API
element.focus();
```

Notice that in the above example there is no difference between the `element` being browser native element or a custom element. The API is what you'd expect of natively implemented components like divs, spans, or inputs. The consumer of the element interacts with the element in the same way.

### Interface Implementer

When creating a custom element, the goal is to simulate the behavior of browser native components in terms of syntax as well as semantics. Here we need to turn to WebComponents technology.

1. Use Custom Elements to register new Element types. This is both using JavaScript type as well as DOM element name. The goal is that when browser parser encounters `<my-element>` and the developers gets hold of the `element`, then that `element` is an instance of `MyElement` type.
2. Use Shadow DOM to create a private encapsulated rendering tree of the component. This also helps with component composition and CSS encapsulation.

```
var Greeter = Object.create(HTMLElement.prototype, {
```

```
  // define a property
  name: {
    get: function() { return this.getAttribute('name'); },
    set: function(value) { this.setAttribute('name', value); }
  },
  // define methods
  greet: function() { ... }
});
document.registerElement('x-greeter', { prototype: Greeter });


var div = document.createElement('div');
div.innerHTML = '<greeter>'; // trigger internal DOM parser.
var greeter = div.firstChild; // retrieve the greeter

// Notice that we are now instance of Greeter
expect(greeter instanceOf Greeter).toBe(true);

// Notice that it can be used just like any other DOM Element
greeter.name = 'world';
greeter.greet();
```

## DOM Elements and Data-binding Frameworks

Web Components are designed to make custom elements indistinguishable from built in Elements for the interface consumer.

A custom element should be usable in plain JavaScript as well as in any frameworks of today. The implication of this is that a custom element implementation should not depend on or favor any existing framework and instead be universally usable. The flip side is that the frameworks should not impose any new rules on custom elements other than what is already expected of native browser elements.

A specific issue which modern frameworks need to solve is how to perform data-binding on elements. The data-binding comes in two directions.

1. Writing data into element. Given the existing API for native elements this is either property or attribute.
2. Reading data from element. This is always a property read. This is because native elements never change the DOM Attributes when the user interacts with them. The native elements always change property. For example the input element `value` attribute is used for initial input state but user changes are always reflected in the

`value` property, whereas the `value` attribute is unchanged.

The writing of data is pretty straight forward since Angular has internal mechanisms to determine when something changes and how that change needs to be reflected into an element or a property. The element does not need to concern itself with change detection, or how the change detection is set up. The element only needs to be concerned with processing the change once the change is written into it using the element's property or attribute. This is demonstrated with `<input>` as an example.

```
// framework locates the element
var input = ...;

// framework updates the element using the element's property
// interface when a change is detected.
input.value = newValue;
```

Reading of the data from element is more complicated since the framework needs to be notified of the change by the underlying element. Currently the browser's built in elements do this using DOM events. So a framework needs to do something like this:

```
// framework locates the element
var input = ...;

// framework registers input event.
input.addEventListener('input', () {
  // The framework needs to read the new value.
  var newValue = input.value;

  // Framework needs to notify the rest of the binding system
  // of the change. This is framework specific.
  framework.notify(input.value);
});
```

While the Web Components specification doesn't specify how one should be notified of changes to internal state, for Angular the most natural extension is to assume that the custom elements will publish internal state changes in the form of events. This is what the web-developers are used to, it is what the browsers do with built-in elements, and it is what we would expect from custom elements which extend the semantics of the built in elements.

This raises a question of what the event should be called. Since WebComponents don't specify this, Angular can assume a default event, but needs to be flexible in allowing for custom element specification during WebComponents import.

## Angular and Web Components

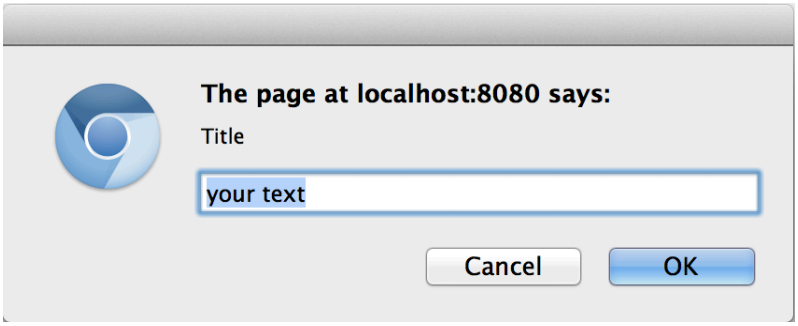
Premise: Because custom elements are just DOM elements they should already work inside Angular.

This section explains how Angular can use Web Components for seamless integration

1. How angular can consume custom elements.
2. How angular can become a custom element and be used by third party, both in plain JavaScript as well as in other non-angular frameworks.

### Angular Consuming Custom Elements

For the purposes of the explanation, lets propose a hypothetical Prompt WebComponent. It will have a title, editable content, and accept / cancel button as shown below.




We have chosen this example because it is a single example which we can use to demonstrate the different kinds of APIs which component may use to render itself.

The Prompt API is:

- Element properties
  - `title`: Dialog title text.
  - `content`: This property changes when the user edits the title. The component also fires `edited` event to notify that the content has changed.
  - `visible`: True if the dialog is currently visible. Changes to visible are accompanied by `open` and `closed` event.

- Element methods
  - `open()`: opens the dialog.
  - `close()`: closes the dialog.
- DOM Attributes
  - `accept-text`: text to show on accept button.
  - `cancel-text`: text to show on cancel button.
- DOM Events
  - `accepted`: fired when accept button is clicked.
  - `canceled`: fired when cancel button is clicked.
  - `open`: fired when dialog box opens.
  - `close`: fired when dialog box closes.
  - `edited`: fired when user edits the title of the dialog box.

```
// EXAMPLE OF WHAT PROMPT COMPONENT MAY LOOK LIKE

var Prompt = Object.create(HTMLElement.prototype, {
  // define a property
  title: { get: function() {...}, set: function(value) {...}},
  content: { get: function() {...}, set: function(value) {...}},
  visible: { get: function() {...}, set: function(value) {...}},

  // define attribute notifications
  attributeChangedCallback(name, old, new) {...},

  // define methods
  open: function() { ... },
  close: function() { ... }
});

document.registerElement('x-prompt', { prototype: Prompt });
```

Instantiating Web Components

Once WebComponent is registered as a custom element using `document.registerElement()` it gets instantiated every time a DOM Element with that name is created.

```
<div ng-repeat="item in items">
  <x-prompt></x-prompt>
</div>
```

There is nothing additional which needs to be done by Angular to support custom

elements. As angular inserts and removes the DOM nodes which contain `<x-prompt>` the browser automatically instantiates the corresponding custom element type as well as notifies the instance of destruction.

### Listening to events

A custom element may fire a custom event such as a `close` event as in this example. Angular needs to be able to listen to this events and perform behavior. The old Angular only supports a known set of events through predefined directives such as `ng-click`. A new event fired by custom element such as `close` would require a new directive, and creating new directive for each event would very quickly become cumbersome. For this reason in Angular v2.0 we are proposing a general `on-*` event syntax so that any event can be listened to.

```
<x-prompt on-close="doSomething()">
```

### Binding from expression to Attributes/Properties

Angular's data binding is a way to automatically update destination when the corresponding source expression changes. Data-binding is at the source of Angular. These questions need to be considered when performing data binding.

1. Angular needs to know if the changed value should be written into DOM attribute or element property. Here Angular can use a heuristic in combination with specific overrides to correctly chose.

   ```
   function updateValue(element, name, value) {
     // lookup behavior from registry
     var useProperty = checkOverrides(element, name);
     if (useProperty == null) {
       // use heuristic
       useProperty = name in element;
     }
     if (useProperty) {
       element[name] = value;
     } else {
       element.setAttribute(name, value);
     }
   }
   ```

2. Angular needs to know if a string `foo` should be interpreted as literal `foo` or as expression which is dereferencing `foo`. It also needs to know if the binding is bi or

uni directional.

```
<x-pane accept-text="I {{name}} accept"
        cancel-text="Cancel"
        bind-title="contract.name">
```

- `cancel-text="Cancel"`:
  This is the simplest kind of binding. It simply sets string literal `Cancel` to DOM attribute `accept-text`. It is an attribute because no corresponding property is defined on element .
  `updateValue(element, 'cancel-text', 'Cancel');`

- `accept-text="I {{name}} accept"`:
  This is a string interpolating form of the binding which always produces a string. The resulting string will be written to DOM attribute `accept-text`. It is attribute because no corresponding property is defined on element .
  `updateValue(element, 'I ' + name + ' accept', 'Cancel');`

- `bind-title="contract.name"`:
  This is a bidirectional binding between the expression `contract.name` and element property `title`. We know that it is an element property `title` because it is defined on element. Any changes in expression will update the property and any changes on property will update the expression. There are two caveats:
  - The expression must be assignable, otherwise this is unidirectional binding.
  - Angular needs to be able to detecting changes on custom element properties / attributes. This is discussed later.

## Getting hold of WebComponents

Expressions in Angular are evaluated against a context which usually is the controller for view. There are times when the expression needs to be evaluated against a different element which may be a custom element.

Consider this example which allows the user to click `Greet` button, that brings up a prompt which queries for user name. The prompt is prefilled with the `World` as the name, but which user can change. Once the dialog is accepted `Hello name!` is alerted.

```
class MyApp {
  constructor() {
    this.name = 'World';
  }
```

```
  greet() {
    alert('Hello ' + this.name + '!');
  }
}
```

---

```
<div>
  <x-prompt ng-id="prompt"
            title="Your name?"
            bind-content="name"
            on-accepted="greet()"></x-prompt>
  <button on-click="prompt.open()">Greet</button>
</div>
```

- `on-accepted="greet()"`:
  This sets up an `accept` listener on the `x-prompt`. When the button is clicked it calls the `greet()` on the controller. We know that the `greet()` is on the controller because the controller is the default execution context for the view.

- `on-click="prompt.open()"`:
  This sets up a `click` listener on the button. However when what we need to do is to execute a method `open()` on the `x-prompt` not on the default controller. What we need is a way to get a hold of the `x-prompt` as a reference. This is achieved with thi `ng-id="prompt"`. This sets up a local reference `prompt` which can then be used to invoke `prompt.open()`.

The above example demonstrates how it is possible to get a hold of components and execute methods on them.

## Detecting property/attribute changes on Elements

An important part which we have glanzed over in previous section is "How does Angular detect a change in custom element?".

A custom element can change DOM attributes or Element property. Changes to DOM attributes can be detected using DOM Mutation Observers. Angular will set up listeners and then react to changes appropriately.

Detecting Element property changes is more complicated as there is no easy way of detecting changes to Element properties in JavaScript or DOM APIs. Our normal

dirty-checking approach will not work here because DOM Elements are really proxies to native objects. Reading properties requires crossing from the VM to native code which is slow and would negatively impact the performance. Object.observe is also unlikely to support observing of element properties for the same reason. This leaves listening to events as the only option to get notified of changes. Event listening is consistent with how native elements work.

Let's look at `input` element as an example of detecting changes and generalize to custom components. In vanila JavaScript one would set up an `input` event listener on `input` element to be notified of changes. Once `input` event is fired the listener can read the `value` property. The issue is that there is no way for Angular to know which events are associated which which properties. This can be solved by specifying event-property map to angular as additional configuration information along with the list of directives. Angular could then use this information to map properties to their corresponding events and setup detect property changes.

This approach works just as well with native elements as it does with WebComponents. WebComponents should not favor any one framework. It should be usable without a framework as well as in traditional jQuery apps. Using events is how this is achieved today in JavaScript by ui components whether or not they are WebComponents compliant.

One thing which Angular can do is to create a heuristic that it will automatically set up a `change` event listener on all components. Any component which emits the event can then have its properties dirty checked for changes. Since Angular only has to dirty check the element which fired the event and only the properties which angular is binding to, the resulting set of property reads is very limited and hence will not have a negative impact on performance. Using a `change` event is already consistent with many of the browser's built in elements as well as third-party components such as JQuery UI and x-tags.

## Angular Component as a WebComponents

In previous section we have discussed how Angular can consume custom-elements. Now lets look at how Angular will create components so that it (the element) behave as a custom element.

The basic idea here is to have angular create a DOM element where all of the Events, properties and directive APIs are published so that it can be consumed by vanila JavaScript, other frameworks and be compliant with WebComponent standards.

### Publishing events, properties and methods

Lets assume that you create a component in Angular like so.

```
@ComponentDirective({
  selector: 'x-zippy'
  template: ...
})
class ZippyComponent {
  constructor(events) {
    this.events;
  }

  @Property('title')
  get title() ...;
  set title(value) ...;

  @Property('isOpen')
  get isOpen() ...;
  set isOpen(value) {
    if (value != this.isOpen) this.toggle();
  }


  @Publish('open')
  open() {
    this.isOpen = true;
    this.events.fire('open');
  }

  @Publish('close')
  close() {
    this.isOpen = false;
    this.events.fire('close');
  }

  @Publish('toggle')
  toggle() {
    value ? close() : open();
  }
}
```

Notice the presence of `@Property` and `@Publish` annotations. These annotations tell angular to expose these properties/methods onto the element so that they will become accessible from the element itself without any knowledge of Angular framework.

The component could be used like so:

```
<x-zippy title="Some clever title.">
  some content which is shown and hidden goes here.
</x-tab>
```

In order for the component to be indistinguishable from the native element it must behave as such:

```
var tab = _lookup_custom_element_;
var log = null;
tab.addEventListener('open', function() { log = 'open'; })
tab.addEventListener('close', function() { log = 'close'; })

tab.close();
expect(tab.isOpen).toEqual(false);
expect(log).toEqual('close');

tab.toggle();
expect(tab.isOpen).toEqual(true);
expect(log).toEqual('open');
```

From the point of view of vanilla JavaScript this is indistinguishable from a native Element. It has properties, events, and methods just as native browsers or custom elements do. The important thing to realize is that Angular framework has set up the getter/setters/methods on the element. Angular will do this on any directive and it will even merge exports from multiple directives into a single Element. By doing this Angular components (and directives) can be used by `vanila.js` as well as any third party framework because Angular elements act just like any other element in the browser.

### Registering as Custom Element

The last piece of the puzzle is that we need to export Angular components as custom elements so that they can be instantiated outside the scope of Angular. (NOTE: this API is speculative but it shows our line of thinking / intention)

```
var injector = angular.createInjector(setOfModules);
var customElementFactory = injector.get(CustomElementFactory);
var Greeter = customElementFactory(GreeterDirective);
document.registerElement('x-greeter', { prototype: Greeter });
```

```
var div = document.createElement('div');
div.innerHTML = '<greeter>'; // trigger internal DOM parser.
var greeter = div.firstChild; // retrieve the greeter

// Notice that we are now instance of Greeter
expect(greeter instanceOf Greeter).toBe(true);

// Notice that it can be used just like any other DOM Element
greeter.name = 'world';
greeter.greet();
```

Angular will provide a `CustomElementFactory` which will create a facade between the `registerElement` API and Angular internals. The resulting facade can be registered with custom elements and become part of browser vocabulary.

## Caveats

...

## Security Considerations

...

## Performance Considerations / Test Strategy

...

## Work Breakdown

...