Polymer Notes

*Author: rob@bluespire.com*

The Polymer project can be found at <http://www.polymer-project.org/> It produces three main categories of functionality: polyfills, prollyfills and elements. It also contains some additional code which stitches these pieces together forming an opinionated framework. The purpose of this document is to examine the polyfills and prollyfills from the perspective of an Angular developer and to determine if there is an opportunity to leverage some or all of it in the construction of Angular 2.0.

## Polyfills

Implement a W3C spec to fill in a “hole” in the browser’s functionality.

**Includes**: [Custom Elements](http://www.polymer-project.org/platform/custom-elements.html), [Shadow DOM](http://www.polymer-project.org/platform/shadow-dom.html), [HTML Imports](http://www.polymer-project.org/platform/html-imports.html), [Pointer Events](http://www.polymer-project.org/platform/pointer-events.html), [Web Animations](http://www.polymer-project.org/platform/web-animations.html), [Mutation Observers](https://github.com/Polymer/MutationObservers), [Weak Map](https://github.com/Polymer/WeakMap), [Object.observe](https://github.com/Polymer/observe-js)

*Note*: *Object.observe isn’t actually a polyfill in the traditional sense. The observe-js library which is used implements its own API to abstract away the details of observation. All its observers match the same interface regardless of what they are observing or how. Under the covers it will use Object.observe if present or fall back to dirty checking. See below for more details.*

## Prollyfills

Implement an API that is not available as a W3C spec, but is intended to be submitted for standardization in the future. Prollyfills are associated with the “Extensible Web Community Group” More info is here: <http://prollyfill.org/>

**Includes**: [Template Binding](http://www.polymer-project.org/platform/template.html), [Node.bind](http://www.polymer-project.org/platform/node_bind.html)

## Elements

On top of the polyfills and prollyfills, Polymer provides its own abstraction of Web Components that augments it with new features and imbibes a particular “style” of declarative development. They also provide a host of their own custom elements built with this.

# Pertinence to Angular 2.0

Angular 2.0 will have a new compiler. Development should be able to immediately take advantage of several of Polymer’s polyfills such as *Mutation Observers* and *Shadow DOM*. But is it possible to leverage some other pieces? For example, **could** the compiler be built on top of Polymer’s *Node.bind* and/or *Template Binding* prollyfills? Some positive side-effects of doing this would be reduction of code maintained by the Angular team, shorter pathway to Angular 2.0 and interoperability with Web Components (interoperability with PolymerElement and XTags as a result). This would also create a stronger internal bond between the Polymer and Angular team. Finally, if these prollyfills can be proven out via an Angular implementation, then that supplies compelling evidence for their inclusion as a W3C standard. If that were to happen, Angular 2.0 would be in a fantastic position long term.

Note that even if these prollyfills aren’t used, there’s still a lot to be learned from their implementation, especially studying how they leverage the lower level polyfills. Angular won’t have any use for Polymer’s element abstraction (*PolymerElement*) or custom elements. This document will basically ignore those features.

# How *Template Binding* Works

The *Template Binding* system’s main function is to instantiate DOM fragments from inert templates. The first time this happens an internal map of all the bindings in the template is generated and cached. The *Template Binding* prollyfill is implemented so that it uses {{ }} and [[ ]] to identify attributes and text content that have bindings. Once the bindings in the template are located, the *Template Binding* passes the binding expression to a *Binding Delegate*. This is an extensibility point that allows a 3rd party to create it's own syntax within the *mustache*. The *Binding Delegate* is expected to return a function which serves as a factory for an *Observable*. Recall, this is all done once per template and stored in a map on the original template. Each time the template is instantiated a document fragment is created and then the *Template Binding* prollyfill walks the template content along with the binding map, using the factories in the binding map to create *Observables*. Finally, it calls *Node.bind* passing it the attribute name, *Observable* and whether or not it is a one-time binding. Internally, the *Node.bind* implementation of each element is responsible for connecting the requested attribute to the *Observable* that it is handed.

# Problems with *Template Binding*

## Binding Discovery

The prollyfill internalizes its method of discovering bindings. It is hard-coded to select only attributes and text content that contain {{ }} or [[ ]]. The Angular team is considering alternative forms of identifying bound attributes, such as using a *bind-* prefix and would like to express “one time” bindings inside the expression language. The prollyfill implementation being as it is today would prevent those types of customizations. It might be worth engaging with the Polymer team to see if they can open up the API in the same way they opened up the expression syntax itself via the *Binding Delegate* hook. A couple of simple delegate hooks would be all that is needed. Perhaps something like this:

function hasBinding(attributeName, value) : bool

function parseBindings(value) : Array

## 

**Fixed Template Directives**

The *TemplateBinding* prollyfill only supports bind, repeat, if and ref directives. It does not appear to be possible to add new template directives. Something similar **could** be accomplished through creating a custom element, but would likely lose benefits of the native directives and need to re-implement behavior.

## Template Directive Callbacks

The TemplateElement can have an if or a repeat attribute defined. These attributes serve the same purpose as ngIf and ngRepeat in Angular. The problem here is that there doesn’t seem to be any hooks into the events of adding or removing the DOM fragments. So, if the underlying array for the repeat has items removed, there’s no way to be notified that the DOM is about to change. The main scenario this makes difficult is animating in/out as the underlying data changes. Similar to the binding discovery issue, I think this can be solved by adding some [strategic hooks into the template binding process](https://www.google.com/url?q=https%3A%2F%2Fgithub.com%2FPolymer%2FTemplateBinding%2Fblob%2F279934127f6777bdfb9f752e98eb0da172fbca1b%2Fsrc%2FTemplateBinding.js%23L886&sa=D&sntz=1&usg=AFQjCNFiak6u7Muw_spVZ6VLxfFfWCjBTw). Though it’s probably a bit more complicated in practice.

# Problems *with Node.bind*

## Syntax Leakage

There is a leak of the expression syntax into *Node.bind* and an inconsistency in how binding expressions and attributes are handled. The *Node.bind* prollyfill uses an attributeName? syntax to determine if a binding should result in adding/removing an attribute rather than setting its value. The check for the ? in the attribute's name is built into [Element.prototype.bind](https://www.google.com/url?q=https%3A%2F%2Fgithub.com%2FPolymer%2FNodeBind%2Fblob%2Feb5ee7941f712cbee755da24ab7553e2d90fb99d%2Fsrc%2FNodeBind.js%23L136&sa=D&sntz=1&usg=AFQjCNHZlkr9-vdrfFX_q-6Z8W9Py3Ls7Q). This means that syntax is spread across three locations.

|  |  |
| --- | --- |
|  |  |

The *PolymerExpression* houses the code that evaluates the binding expression itself. It also has custom logic for handling on-\* event syntax. *Node.bind* contains logic to identify conditional attribute binding. *TemplateBinding* contains logic to discover binding syntax. You can see the inconsistency when you compare the two scenarios represented in the diagram above. Additionally, the implementation of the on-\* syntax essentially hijacks the normal *TemplateBinding* and *Node.bind* behavior. The need to do this shows where the Polymer team themselves are “banging heads” with their own design in an attempt to handle real-world feature needs.

## One Time Binding and Eager Fetching

Since one-time binding is handled directly by *Node.bind* rather than the expression/observation mechanism issues can arise from one-time binding of lazily fetched data.

## Attribute Binding

At present *Node.bind* supports basic HTML attributes well. Cases like select[selected] and input[type=radio] are not supported in a clean fashion though. Additionally, custom elements must override *bind* in order to support non-primitive attribute values. It’s also difficult to handle the “decorator” concept represented by Angular’s ngShow and ngModel directives. Sometimes these types of directives need access to their associated element and they need to register for callbacks notifying them when that element is attached to or removed from the DOM. (Imagine an element that encapsulates a rich textbox, for example.) This could **possibly** be achieved by creating a custom *BindingDelegate*. [PolymerExpressions already has logic](https://www.google.com/url?q=https%3A%2F%2Fgithub.com%2FPolymer%2Fpolymer-expressions%2Fblob%2F470cced7cf167bd164f0b924ceb088dd7a8240b9%2Fsrc%2Fpolymer-expressions.js%23L619&sa=D&sntz=1&usg=AFQjCNFygcFGHG4Qq24Kyb-NQV8dP47ZNQ) to handle the on-\* event syntax. A similar approach may be possible for decorators by using [the open/close interface of Observers](https://www.google.com/url?q=https%3A%2F%2Fgithub.com%2FPolymer%2Fpolymer-expressions%2Fblob%2F470cced7cf167bd164f0b924ceb088dd7a8240b9%2Fsrc%2Fpolymer-expressions.js%23L577&sa=D&sntz=1&usg=AFQjCNH0J1qYkzBorkDNoRNuFj3e4LNBdw). The open/close mechanism would enable resource initialization and disposal. However, it is unclear if this handles the scenario of DOM attach callbacks. In order to implement a full solution Mutation Observers may need to be incorporated as well. Ultimately, it’s not entirely clear if this is possible or not and even if it were, it would be a hack on top of a hack.

# Problems with *PolymerExpression*

PolymerExpression is an implementation of a custom BindingDelegate. It supports an expression syntax that is very similar to Angular 1.x, including support for filters. As mentioned above, it also supports the on-\* event syntax. However, its implementation of event bindings has a couple of problems. First, it [invokes addEventListener](https://www.google.com/url?q=https%3A%2F%2Fgithub.com%2FPolymer%2Fpolymer-expressions%2Fblob%2F470cced7cf167bd164f0b924ceb088dd7a8240b9%2Fsrc%2Fpolymer-expressions.js%23L568&sa=D&sntz=1&usg=AFQjCNEcmcU-ebVM-vcxNxhC8K3vxI2RFg) for every occurrence of an on-\* attribute. If the event delegation strategy previously discussed in [the template document](https://docs.google.com/document/d/1f5VWROeTI2kJwVKbNsrHuEz5IqtZe14OpoxM9fEYJNU/edit#) were to be followed, we would need to write our own BindingDelegate. (This is an expected extensibility point and a large amount of the existing code could be used.) We would also have to address a second issue: the event binding syntax can only be simple path expressions. Method invocations within the expression are not allowed. This simplifies the implementation quite a bit at the cost of restricting expressiveness of templates and reducing testability of components.

# Observations on *observe-js*

## Inconsistent Native vs. Polyfill Behavior

The primary problem with observe-js is that it can cause different behavior in apps running in browsers that have native Object.observe support vs. those that don’t. The problem can be seen when using ES5 getter/setters. When dirty checking is being used to observe changes, it doesn’t matter whether we are observing a simple attribute or a complex computed property. However, if Object.observe is used, this is not true. Object.observe can only watch simple attribute changes. If a getter/setter is involved it can no longer function. To work around this, you must request a custom *notifier* and write additional code. Unfortunately, this code is rather ugly and verbose.

## Dirty Check Initiation

An interesting characteristic of observe-js is that it doesn’t automatically perform dirty checks. You must call Platform.performMicrotaskCheckpoint() to initiate the process. The “polymer platform” handles this with two techniques:

1. Strategically invoking this method at known times. ie. After template bootstrapping is complete.
2. Polling every 125 milliseconds for changes.

*Note: If the browser supports native Object.observe, calls to Platform.performMicrotaskCheckpoint() are essentially a noop.*

# Custom Builds

Using the *polyfills* is straightforward. You just drop the script library into your project and go. One exception to that is *observe-js* which requires some additional work to ensure that dirty checking occurs. However, when you begin to involve the prollyfills and higher level components of the Polymer platform, things get a bit more difficult to mix and match. If a custom build is necessary, there are usually other small scripts that need to be included in order for everything to work properly.

# Summary

There are a number of design problems with the prollyfills that make them difficult to leverage for the purposes of building Angular 2.0. The angular team has expressed some additional concerns as well:

* *Node.bind* doesn't fit well into the existing DOM concepts and APIs (binding becomes a 4th API surface for elements, preceded by attributes, properties and events)
* The semantics of the binding (one-way, two-way, interpolation, etc) and of the expression syntax are handled by the wrong framework components. This mixes up the responsibilities of the component author, the user and the underlying framework.
* One-time binding is implemented at the *Node.bind* level which is unnecessarily low. It would be better to let higher templating layers call a deregistration function that would close the binding when needed.
* It’s very difficult or impossible to create “decorators” or custom “template” directives.
* The compiler is less deterministic (Currently the order that directives are instantiated would be fixed in a not totally desirable way)
* There doesn’t appear to be a way to integrate DI with *TemplateBinding.*
* The closest thing to "asynchronous templates" is templates loaded via HTML imports, which is not very nice. (or is it?)

This could be an excellent opportunity to work with the Polymer team to help address these issues, in particular if they are planning to submit *Node.bind* or *TemplateBinding* for standardization. It’s also important to recognize Polymer’s positive qualities:

* An excellent set of polyfills for various standards.
* Elegantly designed compiler; easy to follow.
* Excluding *Node.bind()/TemplatingBnding* (for the time being), relatively in line with emerging web standards.

While it is unlikely that Angular 2.0 can simply be built as a layer on top of Polymer’s prollyfills, it is likely that use can be made out of the polyfills and that much can be learned from some of the positives and negatives of their prollyfill implementations.