Templating implementation ideas

Status: Draft

Authors: username@angularjs.org

Objective

Collect implementation ideas for the Templating Engine of Angular 2.0

Ideas

- https://github.com/angular/angular.dart/pull/316 - Some experimental work has been discussed on AngularDart for providing interop with Polymer (which may not necessarily support 2-way binding of data)

- AngularDart is already using some of the Web Components pieces to support things directive replace templates.

“Upgrading” elements:

During bootstrap, and on request, a tree of DOM nodes should be walked, and elements which are determined to be directives should be upgraded, recursively, similar to how it is done in Angular 1.x.

Exceptions: Class names should not be inspected and processed, class name directives are cumbersome and seldom used in most applications, and leads to confusion about behaviour not changing when the classes change.

Comments should most likely be ignored as well, except that they may have a use in enabling some server-side pre-rendering as suggested in https://github.com/angular/angular.js/issues/5406.

When upgrading an element, we may be replacing or extending its markup with a template, which may be available synchronously or asynchronously. In order to simplify the compile path, it may be desirable (***) to treat all cases as asynchronous. AngularDart acts as a precedent for making use of the ShadowDOM (or a polyfill) for inserting content, and I agree with this decision. It should enable the use of templates with multiple root nodes.

A native implementation of Web Components should automatically upgrade elements, which unfortunately seems not to be an option for much of AngularJS’s work. Therefore, it’s important to be aware that DOM nodes may already have an existing ShadowDOM before compilation, and an attempt to replace their shadow DOM should probably fail (or fail to

insert/replace the template)

Not every attribute needs to be interpolated. Only text nodes should be interpolated by default, if they contain interpolation markers. Attribute interpolation should occur when asked for by a directive, exclusively.

Attributes used by a directive should be known in advance, similar to the isolate scope definition in Angular 1.x. An Attributes object should exclusively be concerned with known attributes.

Transcluded content:

Rather than using a custom directive (ngTransclude) or a custom transclusion function, Angular 2.0 will come into line with Web Components standards and make use of the <*content*> tag to control the insertion point of child elements.

In Angular 1.x, there is an issue of some annoyance with transclusion scopes, where a new scope is created when it is meant to share the parent scope. While creation of a new scope is probably necessary, ES6 proxy objects could be used to make the new scope behave almost identically to parent scope, and avoid some of the caveats of the existing approach.

Web Components / Custom Elements:

Leveraging web components / custom elements standards, which will likely ship in the relatively near future, is a way to make the framework relatively future-proof. The shadow DOM should enable much more flexibility in what sorts of content can be used, and also should (hopefully) enable some significant performance benefits as user agents implementations mature.

The primary concern with component interop in the Angular environment is data-binding, without necessarily relying on Polymer's own Node.bind() strategy. It is not entirely clear which strategies will the most appropriate. Object.observe or mutation observers may provide a solution which improves on the existing dirty-checking implementation.

Things that are known for certain is that the Shadow DOM portion of Web Components should enable a much more flexible way to presentational directives.

It has been noted that there is no real definition decided on as of yet, for what "interop with Web Components" really means. I have a couple of ideas about this:

1) Perform compilation on Web Components <content> tags, in order to perform some work on their transcluded content. In a Component-heavy application, it is likely that the contents of a component is just additional components, AND SO... This probably means

walking through the entire component tree.

Problems with this: 1) in native implementations, no knowledge of when component elements are upgraded. 2) walking entire component tree can be quite expensive.

2) Provide an API for external components to talk to Angular's scopes. This is iffy, because it creates a limit on which components can really "interop" with Angular, and is probably a messy thing to implement.

3) Strategy similar to https://github.com/angular/angular.dart/pull/316, which could be implemented as a separate class of directive. I think this would be a simpler approach, but I don't necessary like this one.

Things we want to do with external Components:

a) Binding angular scoped data to text nodes and attributes

b) Being notified of user interactions by Components (This seems like it is the author's responsibility, not Angular's)

- Reducing the amount of things which are inspected during compilation (CSS class directives are cumbersome and not used frequently in directives). Comments would also be a good candidate for this, except that they have a potential use in helping to optimize server-side rendering for an initial pageload (to prevent FOUC)

Metadata Providers (Rob)

Many aspects of Angular 2.0 rely on metadata in order to function properly. Some examples include the Dependency Injector's list of injectibles for a class and the Compiler's information about directives. Throughout the design documents this metadata is represented by *annotations* on classes. However, services that use metadata should not be strongly coupled to the particular *location* or *discovery mechanism* for metadata. Each service that uses metadata should have a default behavior for locating metadata, but it should expose a hook which developers can use to provide their own locator function. A few reasons why this is useful:

- Currently annotations are only enabled in traceur. If they find their way into ES6 we may need to update Angular to accommodate for any changes in the spec. We want our changes to be isolated. (Classic SRP)
- Other languages, like TypeScript in particular, may end up implementing their own version of annotations. If this happens, users of the language will want to use its

native features with Angular. They need a way to "teach" Angular about their language's metadata format and location.
- Modern frameworks have tended to use more "Convention over Configuration". Metadata Providers would allow developers a way to plug in their own conventions and "teach" Angular how they want to build apps.
- Some developers will want or need to do things in an imperative style. Consider metaprogramming of directives, for example.

Each service may need to define their own extensibility mechanism for this purpose, but it would be nice to keep things as simple as possible. I think the average service would only need to expose a single function for customization. It might looks something like this:

```
getMetadata(type:constructor, instance:object):metadata
```

The obvious exception to this rule would be the dependency injector, which must gather all its information from the constructor function.

The structure of the metadata object can be fleshed out as we implement the various services. It should include the same types of information seen throughout the various design documents: class metadata, property metadata and constructor metadata. There's no problem with simply using the actual annotation instances in a certain pre-defined structure. The real issue here is not so much the data structure as *how the data is acquired.*

Directive Lifecycle (Rob)

Some directives need to be more aware of their associated element's addition to or removal from the DOM. Also, various directives may need to perform some operation when the directive is fully "configured" (instantiated and all properties set from values and bindings). Here are a couple of examples:

- A rich text editor element may need to measure its available space in order to determine the editor surface size. For this to work, the associated element must be in the DOM and visible (or previously been visible...essentially it must have been laid out before it has a size).
- A directive that wraps a jQuery plugin which needs to be cleaned up when removed from the DOM.
- A directive that performs network, local db or other data access that wants to initiate the async request as soon as property values are available.

There are a couple of examples of "lifecycles" that may be useful in deriving Angular's final implementation. The first comes from Web Components, the second from Durandal.

Templating prototype (Rob)

https://github.com/EisenbergEffect/ng2proto

Experience for the design doc

- hard (complex): find directives via css selectors

Compiler is similar to TemplateBinding (Polymer)

- always work with document fragment -> can have multiple root nodes
- recursively walk the fragment
- extract information from the DOM tree, i.e. what directives and binding expressions match to which nodes
- when instantiating the template the information is evaluated / directives instantiated
- TemplateBinding does not have directives, only bindings

Injector

- sparse, only for nodes that have binding expressions or directives

Nasty:

- bidi binding to inputs: conversion to JS data and validation

Nice:

- Injector works nice
- Rob loves DI in the constructor
- Rob loves defining directives using ES6 classes (was sceptical at first)

Property Binding (deboer)

Design doc reference

How would ng-class be implemented with the new binding syntax.

Today:

With property binding:

using JS properties:

using Dart properties:

using a custom directive ("ng-class")

or call in "bind-ng-class"

Through property binding:

or

Property Binding Implementation.

TODO for selector:

- capture attributes, events and interpolations for each element

ViewFactory (formerly the BlockFactory)

responsible for creating directive instances.

The ViewFactory sees only populated attributes e.g. everything is done through NodeAttrs--e.g. bind-attr, attr and attr={{expr}} are the same.

Classes in the compilation process:

- Compiler
    - The job of the compiler is to build up the directive positional information for the View Factory.
    - recursively iterates over the DOM elements.
    - for each DOM element it calls Selector to collect directives
- Selector
    - For a given Element the selector iterates over the attributes and generates:
        - Hash of the attributes
        - List of Directive Types
        - List of interpolations for a given element / or text node

        - List of on-* Pseudo Selectors
        - List of bind-* Pseudo Selectors
    - Returns: ElementBinder
- ElementBinder
    - Given an injector the element binder creates the directives and bind the attributes / listeners.
    - Fields:
        - Hash of the attributes
        - List of Directive Types
        - List of interpolations for a given element / or text node
        - List of on-* Pseudo Selectors
        - List of bind-* Pseudo Selectors
- ViewFactory
    - Clone the template
    - Use the cloned DOM and feed it to ElementBinder for binding
    - Create a View
- View
    - Container for bound DOM
- ViewHole
    - Collection which can add/remove views.