DOM as a rendering engine VS Being the DOM

*Status: Draft*

*Author: igor@angularjs.org*

*tldr; Angular v1, Knockout, Ember, …. is the former, solutions that utilize Web Components at their core, like Polymer and X-Tags, are the latter. While it’s common to see Web Components as a standard for just defining visual components (widgets), it is possible to use this paradigm for construction of the whole web app rather than only its individual components.*

*We need to decide which paradigm Angular v2 should use because it's a major design choice with many pros and cons to consider.*

# Being DOM

Being DOM means that the core building block of the framework is a custom element that is registered with the browser. It's the browser that instantiates and wires up all of the components. Nothing can happen without a custom element ([Polymer’s](http://www.polymer-project.org/docs/elements/polymer-elements.html) polymer-ajax, polymer-json and polymer-datastorage custom elements are good examples of this approach). There are typically many small elements that are composed to build the final app. Every component is isolated from others and they can communicate only via DOM APIs.

## Pros/Cons:

* [+] for purposes of styling and DOM manipulation, there’s no need to treat custom elements as special because they behave and can be treated like native element
* [+] instantiating a component is easy but with hidden cost (see disadvantages below) and usually done by the browser automatically which can potentially be faster but since components still need to wait for their dependencies to get instantiated the difference might not be big or visible at all
* [+] can leverage existing and upcoming tooling for DOM and web components
* [-] the presentation and business logic is hard not to mix because a model for maintaining a separation of concerns is missing which makes it difficult to build maintainable and testable applications
* [-] instantiation of non-trivial elements has to rely on global state to get wire up non-DOM dependencies, which breaks encapsulation of components, resulting in hard reuse and testing
* [?] instead of "fixing up" native elements via decoration, it is preferable to create a custom component that replaces them. For small tweaks this is a cumbersome approach because it is a ton of work and looks weird for one off tweaks. However it is a better option if done as part of a well align suite of components – e.g. custom replacement for all form elements.

# Using DOM as rendering engine

Solutions in this category typically have a model layer that is projected through a template to obtain the final DOM view to be rendered by the browser. The browser is typically unaware of any kind of component structure in the document. It just merely renders a soup of tags prepared based on the templates by the framework.

Components created in this solution are usually not custom elements registered via `document.registerElement`but are instead valid native elements or instances of HTMLUnknownElement. The contents of such components can be implemented within light DOM or shadow DOM that belongs to these elements.

## Pros/Cons:

* [+] full control over element instantiation means great testing and dependency story. Preserves encapsulation between components by removing dependency on global state.
* [+] much easier to create separation of concerns between the model layer and presentation layer by overlaying the model layer on top of the DOM/presentation layer (since components use DOM for rendering but are not DOM)
* [+] if implemented correctly can have full interoperability with custom elements implemented with any solution in the "Being DOM" category.
* [+] full control over timing and batching of operations can result in performance improvements at least until the DOM apis are significantly faster.
* [-] exporting custom elements as web components will require extra work because internally a component is not a custom element and any dependencies not provided to the element with global context need to be delivered to the element somehow.
* [-] since the ren
* dered DOM is just a side-effect of non-standard component architecture, reasoning about how the resulting DOM was created is harder and requires aid from the framework[-] can only partially benefit from existing and standards based tooling and will require custom tools

# Wrap up

We love being DOM as the way for describing visual elements.

We feel this paradigm breaks down when describing full applications. The main reasons are lack of separation of concerns and lack of ability to manage global state. Both of these result in application code that is hard to maintain and reuse at scale.

On the Angular project, we will pursue the option of using DOM for rendering and being fully capable of [integrating](https://docs.google.com/a/google.com/document/d/1L-9bBL-smMrAmxC_pVBdKWKcB-ZlOLTj-jmzAJ2jKeE/edit#heading=h.xgjl2srtytjt) with components built using Web Components that "are the DOM". This approach enables us to support developers in following best practices while also integrating other solutions.

We are hopeful that the issues that are preventing us from building a solution that is based on "being the DOM" paradigm can be resolved in the web platform and will keep on investigating the possible solutions.