Angular 2 Rendering Architecture

**Warning: This document is publicly readable by anyone**

Yegor Jbanov (yjbanov@google.com)

Tobias Bosch (tbosch@google.com)

[Objective](#h.kl49np3rcxwy)

[Goals](#h.owsdtw3w8t06)

[Non-goals](#h.ecs37rw6d14z)

[Background](#h.hjg6pm7bve6y)

[Design overview](#h.1mxz0l46srf9)

[Principles](#h.ltvowz1franf)

[Detailed Design](#h.vstlgrec0sr)

[Custom render services](#h.793hvzy7itzx)

[Use cases](#h.yokaqs4lyhd3)

[DomRenderer](#h.nyoe821fgsfo)

[NativeScript renderer](#h.9qbsmq9ykh4l)

[A note on events and reading the DOM](#h.bn040jgp700q)

[Interaction with Animations](#h.3jf4lx6chh8s)

[Prior Art](#h.camep0pntypv)

# Objective

## Goals

Separate Angular 2 runtime into two layers *application layer* and *render layer*. The application layer contains APIs and runtime that application code interacts with directly. The rendering layer provides a common protocol for performing UI updates. The split will allow Angular 2 applications to run in different environments/platforms while providing the same set of abstractions to application developers (or as Reactjs team likes to say: “learn once, write anywhere”). We want to support the following environments:

* web apps running in browser’s main process (this is the only environment supported by Angular 1 and we will continue to support it)
* web apps split across browser’s main process and web workers
* web apps running on the server side (initially for testing purposes)
* native apps built with Angular 2 (currently evaluating [NativeScript](https://www.nativescript.org/) and [Fletch](https://github.com/dart-lang/fletch))
* testing web app UI outside the web browser

The second benefit we are trying to realize with this split is a clearer separation of application logic from the graphical aspects of the application. This would allow developers to optimize the experience based on device capabilities. For example, on low performance devices or devices in battery saving mode an application could run with animations disabled while preserving full functionality.

In this document we define a specification of the rendering layer’s API and protocols.

## Non-goals

It is outside the scope of this document to provide exact implementations of the various rendering layers. Specific implementations will be described in separate design docs.

# Background

In the past several years platforms diversified significantly from isolated single-process browser pages. Mobile took off. For application developers this presents opportunities in performance and exposure to new markets. We want to make sure that Angular 2 takes advantage of these shifts in the platforms.

New areas available to us:

* Web workers
  + Running application code in a web worker allows the UI to stay responsive even when the application performs computation-heavy actions and take advantage of multiple CPU cores in users’ devices.
* Mobile native UIs:
  + Native UIs on mobile offer excellent user experience and great performance. React Native, NativeScript and Dart Fletch all demonstrate that this can be done from scripting environments such as JS and Dart.
* Testing: mock out location changes, ...

# Design overview

The following illustrates the layout of the rendering architecture:



One key aspect of the design is that the elements of the application do not directly depend or access the elements of the render code, and vice versa. They can only communicate via a *renderer transport*. This leads to the following properties:

* Application and render code can be supplied via separate files (compilation units)
* Application code can run in a separate process from the process where the renderer code runs (e.g. web worker, server)

From the developer’s perspective the architecture splits application elements into two sections we refer to as *application layer* and *render layer*. A given feature exists in one of the two, but not both. Here are the examples of what “lives” where:



## Principles

Where possible, the abstraction layer should not add more abstractions/indirections to the implementation for the browser main process so that we don't get slowed down.

The abstraction layer should be as minimal as possible to make it easy to add more implementations. Concrete implementations could provide more functionality (e.g. reading out an element's position), but an app using these extras would not run in other environments.

# Detailed Design

Angular is split into the following parts:

* **core**: contains classes and behavior that execute in the application layer. These classes do not depend on platform APIs, such as HTML DOM.
* **render/api**: provides the vocabulary and the protocol for the application layer to talk to the render layer. The main interfaces are the RenderCompiler and the Renderer. This API also has no dependencies on platform APIs.
* **renderer/proxy:** Contains proxies that allow to use RenderCompilers and Renderers in other locations (e.g. WebWorkers, …).
* **render/dom/dom\_renderer**: is a renderer implementation for the web browser. Each platform will provide its own renderer implementation.

The initial refactoring of the architecture won't include a proxy for the web-worker scenario.



View hierarchy management (simplified):

1. app layer: method on ViewManager is called (e.g. createViewInContainer). Passes the RenderProtoViewRef to the renderer.
2. render layer: instantiate a RenderView based on the given RenderProtoViewRef and return a RenderViewRef (via ViewManager).
3. app layer: instantiate AppView and associate the RenderViewRef to it

View update:

1. app layer: detect change via change detection and call render layer with RenderViewRef that is stored in the AppView
2. render layer: apply the update to the RenderView

A note on recursion:

When a component is created, contained child components also need to be created. Only the ViewManager on the application side implements this recursion and calls the renderer with basic operations. Because of this, the methods on the renderer stay simple and are easy to implement.

Renderer api

|  |
| --- |
| createView(protoView:RenderProtoViewRef,  eventDispatcher:EventDispatcher):RenderViewRef  // the app layer needs to know  // when animations are done...  destroyView(view:RenderViewRef):Promise  // called on renderer of child component view!  attachComponentView(hostElementRef:RenderElementRef,  componentView:RenderViewRef)  // called on renderer of child component view!  detachComponentView(hostElementRef:RenderElementRef,  componentView:RenderView)  attachViewInContainer(hostElementRef:RenderElementRef,  vcElementRef:RenderElementRef,  atIndex:number,  viewRef:RenderViewRef)  detachViewInContainer(parentView:RenderViewRef,  boundElementIndex:number, atIndex:number,  view:RenderViewRef)  // called on renderer of view that contains the renderLocation  // -> renderer knows how to interpret the renderLocation!  becomeHostView(parentHostElementRef:RenderElementRef,  renderLocation: any, protoView:RenderProtoViewRef):  RenderViewRef  unbecomeHostView(parentHostElementRef:RenderElementRef,  hostView:RenderViewRef)  hydrateView(hostElementRef:RenderElementRef,  childView:RenderViewRef)  dehydrateView(view:RenderView)  setElementProperty(elementRef:RenderElementRef,  propertyName:string, propertyValue:any)  setText(view:RenderViewRef, textNodeIndex:number, text:string) |

RenderView

|  |
| --- |
| class AbstractRenderView {  hostElements:Array<any>;  lightDoms:Array<any>;  } |

## Custom render services

Custom render services are services that live on the render layer and are accessible via an interface on the app layer. They are used to abstract things like the document title, cookies, animations, … away from the application layer. Angular will provide a general RPC mechanism that allows to create client and services given an interface with custom serialization logic.

The Renderer interface itself is a special case of a render service.

## Use cases

### DomRenderer

Every component can define its own renderer. By default, every component is using the DomRenderer. This renderer is made up of the DomCompiler, which compiles templates written in html. To instantiate views, it uses the DomViewManager, which clones the DOM elements of the compiler and inserts them the main browser document.

Some more use cases for other renderer implementations:

### NativeScript renderer

The NativeScript renderer will probably use a compiler that is based on XML but shares most of the functionality with the DomRenderer. For creating views, the NativeScript renderer will use the default mechanism of NativeScript to create native elements out of XML. I.e. the RenderViews won't contain DOM nor XML but custom JS classes.

## A note on events and reading the DOM

The DOM is only read when events are fired. As a DOM event is not serializable we need to know which properties that application logic is interested in. This is done via transforming event expressions:

E.g. to read out the value of an input element when the user changes it:

|  |
| --- |
| <input (change)="updateState($target.value)"> |

The compiler in the render layer already parses all expressions. Because of this, it knows which properties should be read (in the example: property value). So when the event happens the render view will read out the needed properties and send them over to the application side as well.

How about preventDefault?

* we preventDefault always
* allow WebWorker to decide whether we actually need to do it
* if we should not prevent default, then replay the event with a synchronous Event

## Interaction with Animations

Via the render interface there is a single place where all changes to the UI go through. Animations will hook into this place and maybe delay some of them.

# Prior Art

[Ember environment](http://emberjs.com/blog/2014/12/22/inside-fastboot-the-road-to-server-side-rendering.html)

* Ember provides [abstractions](https://github.com/emberjs/ember.js/blob/master/packages/ember-metal/lib/environment.js) around the DOM and the browser to be able to prerender on the server side for a fast bootstrap.

[Browserify](http://browserify.org/)

* Normalize APIs across server and client

[Barista (Google internal)](https://docs.google.com/document/d/1GDfU11LXKH9B42fEuRzOMGnRPSRQ6iSA1JgmbOyEC60/edit)

* General purpose virtual DOM implementation to access most features of the browser DOM from a worker or server process

[virtual-dom](https://github.com/Matt-Esch/virtual-dom)

* General purpose library inspired by Facebook's React DOM diffing
* General purpose virtual DOM implementation to access most feature of the browser DOM from a worker process

[Facebook React virtual DOM](http://facebook.github.io/react/docs/glossary.html)

* Contains a virtual DOM implementation that is bound to React, i.e. not general purpose

[Facebook React Native](http://www.reactnative.com/)

* Allows syncing virtual DOM onto native UI on mobile platforms.