Angular 2 Render Directives

STATUS: DRAFT

yjbanov@google.com

A *render directive* is a new kind of directive introduced in Angular 2. It addresses the need to implement platform-specific features that Angular core does not provide out-of-the-box. Render directives provide API to the developer to interact *directly* with the Render API and platform APIs, such as browser HTML DOM API.

This document describes the design for render directives *for the browser*. It is likely that we will need a similar concept for non-browser environments, but that is out of scope for this document. Code samples in this document are written in Dart but apply to TypeScript as well.

Background

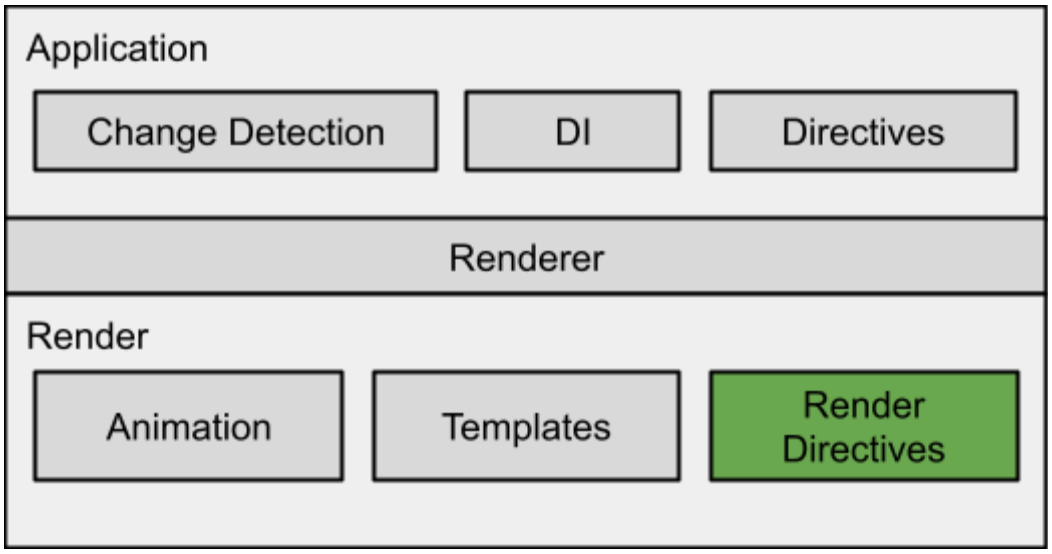
Angular's application layer is designed to be platform-agnostic. This includes DI, directives, change detection and the template language. It does not mean that you write an application once and run it on all platforms. It means you *learn* the core Angular concepts and APIs once, and reuse that knowledge on many platforms. However, a given Angular application targets a specific platform, and therefore needs access to that platform's features. Use-cases include:

- Binding to native properties, such as "classList" (DOMTokenList), "style" (CSSStyleDeclaration) and "attributes" (NamedNodeMap).
- Animations, overlays and other styling and theming aspects
- Custom data models-to-DOM syncing, for example:
  - Game scene graph synced to canvas
  - VirtualDOM synced to real HTML DOM
  - Chart data synced to SVG

Design

Definition: *Render directives is a developer-facing API for implementing browser-specific features in Angular.*

Render directives live in the render layer and operate on native HTML DOM nodes and View objects as defined in Angular's Render API. They do not directly participate it change detection and dependency injection. The application is connected to render directives through the Renderer, and can pass data through the property binding mechanism:



A developer of a render directive needs to be able to:

- select which nodes to apply a render directive to
- select which views to apply a render directive to

To do anything useful in the directive the developer needs to be able to:

- get a reference to the HTML node
- get a reference to the View
- get data from the application layer

RenderDirective annotation

The design continues the philosophy of using annotations as a way for developers to supply declarative information to Angular. This satisfies two design goals:

- **Familiarity:** we are pushing hard with annotations, so it is expected that developers will become very comfortable with the concept
- **Static analysis:**

    - code navigation in the IDE
    - type warnings
    - static linking of render directives via transformers

Definition:

```
class RenderDirective {
  /// Used to indicate which elements the directive applies to.
  final String selector;

  /// Specifies a pipe to apply to application-side value prior to
  /// passing the value to the render directive.
  final String autoPipe;

  /// Indicates whether this render directive is in charge of
  /// scheduling the lifecycle events of a view.
  final bool controlsLifecycle;

  const RenderDirective({
    this.selector = null,
    this.autoPipe = null,
    this.controlsLifecycle = false
  });
}
```

Selecting Nodes

Render directives support three types of selectors:

- **Property:** this selector applies the render directive to elements on the template that bind to properties. Selectors of this type begin with @, for example “@style”.
- **Element and component:** this selector applies the render directive to an HTML element or an Angular component. Selector of this type uses normal CSS selector, e.g. “my-button”, “#blinking-title”, “.slide-in”.
- **Event:** this selector applies the render directive to elements on the template that bind to an event of given type. Selectors of this type wrap the event type name in parentheses, e.g. “(map-location-change)”.

Examples:

```
// Matches any of:
//   <div [style]=”myStyleMap”></div>
//   <div [style.width]=”dialogWidth”></div>
@RenderDirective(
```

```
  selector: "@style"
)

// Matches: <my-button>Save</my-button>
@RenderDirective(
  selector: "my-button"
)

// Matches: <google-map (map-location-change)="update($event)">
@RenderDirective(
  selector: "(map-location-change)"
)
```

Multiple render directives may be applied to one element. However, only one of them can control the lifecycle. It is up to the developer to make sure the directives are compatible with each other.

Defining render directives

Render directives can be implemented either via a class or via a function. Example:

```
@RenderDirective(
  selector: "@style"
)
class StyleRenderDirective {
  ...
}

@RenderDirective(
  selector: "@style"
)
void styleRenderDirective(...) {
  ...
}
```

Which one to use depends on the use-case, as each has its tradeoffs:

Class-based:

- its lifespan is tied to the owning view
- it has access to view lifecycle
- it can have local state
- it is instantiated for every match and therefore can use more memory

Function-based:

- easy on memory as the same function is reused for all matches
- cannot have local state
- does not have access to lifecycle

Context objects

A render directive is free to access all of browser APIs. However, for many use-cases it is important that it knows the objects that are in the context where the directive is applied. These objects are supplied inside a RenderContext object:

```
class RenderContext {
  /// Element `this` render directive is applied to.
  Element get element;

  /// View that owns [element].
  View get view;

  /// View under [element], or `null` if none.
  View get childView;

  /// Property in the LHS expression. This field is only
  /// available if the render directive selects a property
  /// and only if a property exists. Otherwise, it is `null`.
  ///
  /// Example 1: selector == "@style", property == "width"
  ///
  ///    <div [style.width]="w"></div>
  ///
  /// Example 2: selector == "@style", property == null
  ///
  ///    <div [style]="styleMap"></div>
  ///
  /// Example 3: selector == "my-button", property == null
  ///
  ///    <my-button>Save</my-button>
  String property;

  /// A render directive calls this method to emit an event
  /// to the application layer.
  void emitEvent(event);
}
```

Class-based render directives receive RenderContext via constructor. For example:

```
@RenderDirective(
  selector: "@style"
)
class StyleRenderDirective {
  final RenderContext _renderContext;
  StyleRenderDirective(this._renderContext);
}
```

Function-based render directives receive it as a parameter.

```
@RenderDirective(
  selector: "@style"
)
void styleRenderDirective(RenderContext ctx, ...) {
  ...
}
```

Value binding

Finally, we need to be able to pass bound values to render directives. A render directive receives an initial value, followed by updates as dictated by the change detection system.

Class-based directives receive values via a reserved method `syncValue`. Here's a complete implementation of the style directive using a class:

```
@RenderDirective(
  selector: "@style"
)
class StyleRenderDirective {
  final RenderContext _ctx;
  StyleRenderDirective(this._ctx);

  void syncValue(value) {
    if (value is Map) {
      value.forEach((prop, val) {
        _ctx.element.style.setProperty(prop, val);
      });
    } else if (_ctx.property != null) {
      _ctx.element.style.setProperty(_ctx.property, val);
    }
  }
}
```

The above directive is stateless, so it could be a function. Function-based directives receive their context object and values as normal parameters:

```
@RenderDirective(
  selector: "@style"
)
void styleRenderDirective(RenderContext ctx, value) {
  // code similar to the above
}
```

Auto-pipes

Imagine we are binding a `Map` to `[style]`. If the map contains a large number of properties it is inefficient to set all of them when only one of them changes. Instead, when a structural change occurs we should apply a delta. Pipes allow us to transform the value of a RHS expression in the template prior to passing it through the binding. So in the style case we could do:

```
<div [style]="styleMap | stylePatcher"></div>
```

Here "stylePatcher" pipe computes efficient deltas to apply to the DOM. However, it would be very inconvenient for the user to have to specify the pipe every time they bind to [style] if its always the same pipe, which is extremely likely for render directives such as [style].

`autoPipe` field of the `@RenderDirective` annotation indicates that a specific pipe needs to be applied whenever the render directive is used. Example:

```
@RenderDirective(
  selector: '@style',
  autoPipe: 'stylePatcher'
)
class StyleRenderDirective {
  …
}
```

`autoPipe` is optional. Not every render directive may require transformation of bound values.

Lifecycle

Class-based directives may also control the scheduling of View add/remove operations. An animated transition may wish to retain both views temporarily, when one of the views is replacing another (i.e. one being removed while another one being added). The directive would indicate that using the `controlsLifecycle` field of the `@RenderDirective` annotation and extend the `ViewLifecycleControl` class:

```
/// The life-cycle of a RenderDirective follows the pattern
///
/// instantiate => (dehydrate => hydrate)* => destroy (GC)
abstract class ViewLifecycleControl {
  /// Schedules the add operation via a Future that triggers
  /// the addition. Return `null` to add immediately.
  Future scheduleAdd() => null;

  /// Schedules the remove operation via a Future that triggers
  /// the removal. Return `null` to remove immediately.
  Future scheduleRemove() => null;

  /// Called prior to reusing this directive with a new value
  hydrate(value) {};

  /// Allows the directive to cleanup before the view is returned
  /// to the pool. This method is called immediately after removal.
  dehydrate() {};
}
```

Example:

```
@RenderDirective(
  selector: '@slide-in-out',
  controlsLifecycle: true
)
class StyleRenderDirective extends ViewLifecycleControl {
  final RenderContext _ctx;

  StyleRenderDirective(this._ctx);

  void syncValue() { … }

  Future scheduleAdd() {
    // implements slide-in animation on _ctx.element
    return new Future.delayed(...);
  }

  Future scheduleRemove() {
    // implements slide-out animation on _tx.element
    return new Future.delayed(...);
  }
```

```
  dehydrate() {
    // restore _ctx.element.style
  }
}
```

Events

A render directive can be a source of custom events. To emit an event a render directive calls `emitEvent` on the `RenderContext` object. The runtime will dispatch the event to the correct handling statement.

Example: a naive quadruple-click custom event

```
@RenderDirective(
  selector: '(quadruple-click)',
  controlsLifecycle: true
)
class StyleRenderDirective extends ViewLifecycleControl {
  final RenderContext _ctx;
  StreamSubscription _sub;

  StyleRenderDirective(this._ctx);

  hydrate(_) {
    int count = 0;
    _sub = _ctx.element.onclick.listen((_) {
      count++;
      if (count == 4) {
        count = 0;
        _ctx.emitEvent(new QuadrupleClick());
      }
    });
  }

  dehydrate() {
    _sub.cancel();
  }
}
```

Usage:

```
<button (quadruple-click)="processEvent($event)">
  Click me 4 times
```

```
</button>
```