Angular 2 Static Content Projection

**Attention: This is a public document!**

Tobias Bosch ([tbosch@google.com](mailto:tbosch@google.com))

Date: September 2015

[Purpose](#h.jmx8s3venarq)

[Proposal](#h.eiobuva63odc)

[Requirement 1: the association of a light dom node does not change](#h.2de4hqou64en)

[Requirement 2: project embedded templates all or nothing](#h.t4tshqntbocm)

[Requirement 3: keep non projected light dom if combined with native shadow dom](#h.fw9nfkwm00g5)

[Notes for the implementation](#h.6dipmu9akf1a)

[Merge the ProtoViews of nested components when a View is created](#h.qeijunz1lv1x)

[Special case for <ng-content> inside of an ng-if / ng-for / …](#h.pjpkygfv5elz)

[Results](#h.6s263bje7a7m)

[Appendix](#h.mltofks9hlj6)

[Integration into the new compile pipeline](#h.8vhwhpirvn74)

[Changes to also support the new slot proposal of ShadowDom](#h.b1yyhsqo1epb)

# Purpose

The [original implementation of content projection emulation](https://docs.google.com/document/d/1NFmp2ptjFfzTEYf0OPcpkacRV2_7LgTIrP5nWfJQL0o/edit#heading=h.z2blzd2pdtwt) in Angular 2 had the following downsides:

* it is complicated to implement and tied to the browser DOM. I.e. other renderers (e.g. for native mobile apps via NativeScript) have a hard time reimplementing it, making content projection unusable for them.
* it requires O(number of DOM nodes to be projected X number of content tags in a component) checks to find the association between DOM nodes and content elements, with every check being a CSS selector match check.
* it needs to be re executed whenever Angular creates, moves or destroys views. This slows down view creation when components use content projection.
* it can't be used together with manual content projection. I.e. move projected content somewhere else programmatically.
* there are tricky corner cases when Angular components are used together with components of other frameworks that also emulated content projection (e.g. Polymer), as both are modifying the DOM at the same time

# Proposal

Main idea:

* don't change the association of nodes to a <ng-content> element when a previous <ng-content> is added / removed.
* project embedded templates (ng-for, ng-if, …) all or nothing
* project <ng-content> all or nothing

This allows to precompute the association of DOM nodes to content elements in a build step and does not require to re execute content projection when views are created / destroyed.

To make this different behavior explicit to users, use <ng-content> instead of <content> as tag name.

## Restriction 1: the association of a light dom node to an <ng-content> does not change when other <ng-content> elements appear / disappear

Example:

|  |
| --- |
| <my-comp>  <b></b>  hello  </my-comp>  @View({  template: `  <template [ng-if]="showB">  B: <ng-content select="b"/>  <template>  Rest: <ng-content select="\*"/>  `  })  class MyComp {  showB: boolean;  } |

When "showB" is true, both, dynamic and static content projection will display the following:

|  |
| --- |
| <my-comp>  B: <b></b>  Rest: hello  </my-comp> |

However, when "showB" is false:

|  |
| --- |
| Dynamic content projection:  <my-comp>  Rest: <b><b>hello  </my-comp>  Static content projection:  <my-comp>  Rest: hello  </my-comp> |

## 

## Restriction 2: project embedded templates all or nothing

If an embedded template should be projected, either project the whole embedded template or none of it, so that we don't have to do reproject when an ng-if / ng-for fires. I.e. match the <template> element itself against the selector of <ng-content>, and not the children of the <template> element.

E.g. the following example will always project nothing with static content projection, as the selector of <ng-content> does not look inside of the embedded template.

|  |
| --- |
| <my-comp>  <template ng-if="test">  <a></a>  <b></b>  </template>  </my-comp>  @View({  template: `<ng-content select="a">`  })  class MyComp {} |

## Restriction 3: project <ng-content> all or nothing

If <ng-content> should be projected itself (so called reprojection), either reproject all nodes that will match the <ng-content> or none of them.

E.g. the following example will project nothing with static content projection, as the selector of <ng-content> inside of BComp does not match the <ng-content> inside of AComp. The <ng-content> inside of BComp will not look at the nodes that are projected into the <ng-content> of AComp.

|  |
| --- |
| Usage:  <a>  <div class="b">hello</div>  </a>  @Component({  selector: 'a'  })  @View({  template: `<b><ng-content select=".b"></b>`,  directives: [BComp]  })  class AComp {}  @Component({  selector: 'b'  })  @View({  template: `<ng-content select=".b">`  })  class BComp {} |

# Notes for the implementation

## Merge the ProtoViews of nested components when a View is created

I.e. we don't first create the component views and then move the light dom into them, but create the Views with already projected elements.

Data structure implications for AppView:

We count bound element indices over all merged ProtoViews, grouped by ProtoView and share the arrays for ElementInjectors, ViewContainers, … in Views between all Views of merged components.

E.g. a host ProtoView with one component:

AppProtoView0:

* elementBinders: [binder0] // contains reference to component AppProtoView1

AppProotView1:

* elementBinders: [binder1]

Created AppViews:

View0:

* elementInjectors: [injector0, injector1]
* elementOffset: 0

View1:

* elementInjectors: [injector0, injector1]
* elementOffset: 1

## Special case for <ng-content> inside of an ng-if / ng-for / …

E.g.

|  |
| --- |
| <my-comp>  <a></a>  </my-comp>  @View({  template: `  <template [ng-for]="items">  <ng-content select="a"/>  <template>  `  })  class MyComp {} |

In this case we need to treat the first instance of the embedded template in special way, as it also contains the projected light dom. All the other instances of the embedded template must not include the light dom, as light dom nodes can be projected only once.

For this situation, we create one View of the embedded template right when the component template is instantiated and project the light dom into it. We don't hydrate this embedded view when the component view is hydrated, but wait until the ng-for fires the first time. We also keep this embedded View instance in a special place in the parent view, so we can just dehydrate it when the ng-for clears its views and don't loose the reference to it. These special views are called fragments in the implementation.

For the subsequent views that should be created from the embedded template we create fresh instances without doing any content projection.

As we store this view in a special place, manual content projection can also use this. I.e. a template directive that creates the view at another place.

## Keep non projected light dom if combined with native shadow dom

To be able to mix and match native shadow dom with <ng-content>, we keep all nodes that have not been projected by ng-content in the light dom:

E.g.

|  |
| --- |
| <my-comp>  <a></a>  <b></b>  </my-comp>  @View({  template: `<ng-content select="a"><content select="b">`  })  class MyComp {} |

This will display as:

|  |
| --- |
| <my-comp>  <b></b>  #shadow-root  <a></a>  <content select="b">  </my-comp> |

# Results

This new content projection lead to a 2x faster speed at creating static components (the time for creating 1024 components dropped from 49.59+-14%ms to 26.84+-8%, [see this commit message for details](https://github.com/angular/angular/commit/854b5b7da85ae25d22979b47e63a54eade45f8a7)). I.e. components that are unconditionally included in their parents (measured with the [static tree benchmark](https://github.com/angular/angular/tree/master/modules/benchmarks/src/static_tree)). On a sample application that mixes static components with components inside of ng-if, this lead to 1.4x faster view creation speed.

Furthermore, using <ng-content> has no impact on the view creation time any more!

Content projection now works across all renderers, which includes the regular DOM renderer as well as a renderer for react-native and NativeScript.

# Appendix

## Integration into the [new compile pipeline](https://docs.google.com/document/d/1iVcmAaONB2ztA5GElL9pOkqcU4Gi5LNJDA0Yyp19AQA/edit)

Initially, content projection was implemented by merging ProtoViews in the Compiler. Here is how it will integrate into the new compile pipeline:

Given the following template and component definition:

|  |
| --- |
| <my-comp>  hello  <a></a>  <b></b>  </my-comp>  @View({  template: `  <ng-content select="a"/>  <template [ng-if]="showB">  <ng-content select="b"/>  <template>  <ng-content select="\*"/>  `  })  class MyComp {} |

During compiling of components, we create the following representation of the template:

|  |
| --- |
| beginComponent('my-comp');  text('hello', ngContentIndex:2);  beginElement('a', ngContentIndex:0);  endElement('a');  beginElement('b', ngContentIndex:1);  endElement('b');  endComponent(); |

I.e. we precalculate the index of the <ng-content> element into which the given element should be projected. This can be implemented in a very fast way.

## Changes to also support the new slot proposal of ShadowDom

[The ShadowDom spec](http://w3c.github.io/webcomponents/spec/shadow/#distributions) is about to simplify content projection: It introduces a slot attribute on elements that is used in the light dom and a <slot> element that is used instead of <ng-content> to select the elements that should be projected.

E.g.

|  |
| --- |
| <my-comp>  hello  <a slot="a"></a>  <b slot="b"></b>  </my-comp>  @View({  template: `  <slot name="a"/>  <slot name="b"/>  <slot name="\*"/>  `  })  class MyComp {} |

For Angular 2, we will probably introduce a <ng-slot> element that will be equivalent to <ng-content select="[slot=...]">. As we index light dom nodes already per ng-content element in our compile pipeline, this won't have any performance effects on Angular.