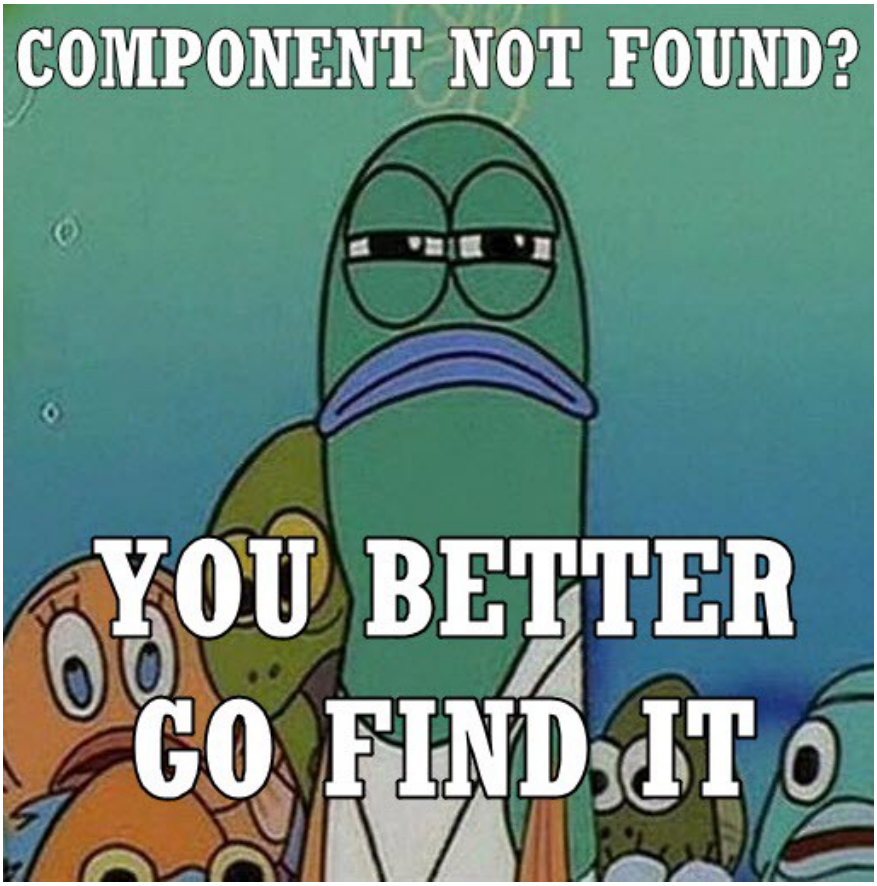
Here is why you will not find components inside Angular

Max Koretskyi aka Wizard Follow
Jun 27, 2017 · 9 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

Component is just a directive with a template? Or is it?

From the moment I started using Angular I was intrigued by the question what's the difference between components and directives. It's a nagging question especially for those who come from the AngularJS world since we only had directives that we often used as components. If you search the web for the explanation you will see many phrases like these:

> *Components are just directives with a content defined in a template…*

> *Angular components are a subset of directives. Unlike directives, components always have…*

> *Components are high-order directives with templates and serve as …*

The claims seem to be true since when I looked at the factories generated for the components I **didn't find component** definitions there! And you will not find either. Only directives…

And I haven't found an *why* explanation because to provide it one must have a good understanding of how Angular works inside. If this question bugged you for a while then this article is for you. It is intended to divulge the mystery. But get ready for some hardcore stuff 😎 .

In essence, this article explains how Angular represents components and directives under the hood and introduces new view node definition—directive definition.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

• • •

Good old-fashioned view

If you've read some of my previous articles, particularly how Angular updates DOM, you probably already know that under the hood

Angular application is a tree of views. Each **view is generated from the factory** and is comprised of view nodes of different types each with a specific functionality. In the mentioned article (it will greatly help to understand this article) I've shown two most simple node types—element definition and text definition. The former is created for all element DOM nodes and the latter is generated for all text nodes.

So if you have a template like this:

```
<div><h1>Hello {{name}}</h1></div>
```

the compiler will generate the view definition with two element nodes for `div` and `h1` DOM elements and one text node for the `Hello {{name}}` part. These are very important nodes as without them we wouldn't be able to see anything on the screen. But since the component composition pattern dictates that we should be able to nest components there must be another type of a view node for embedded components. To figure out what those special nodes are let's first see what the component is comprised of. A component is essentially a DOM element with an attached behavior implemented in the component class. Let's start with the DOM element.

Custom DOM elements

You probably know that you can create a new HTML tag and use it in your html. For example, if you don't use any framework and you insert the following into your html :

```
<a-comp></a-comp>
```

and then query the DOM node and check its type you will see that it's a perfectly valid DOM element:

```
const element = document.querySelector('a-comp');
element.nodeType === Node.ELEMENT_NODE; // true
```

This `a-comp` element will be created by a browser using the HTMLUnknownElement interface that inherits from HTMLElement interface, but without implementing any additional properties or methods. You will be able to style it using CSS and also attach event listeners to common events like `click`. As I said, perfectly fine html element.

You can create a much upgraded version of this element if you turn it into custom element. You would need to create a class for it and register it using provided API:

```
class AComponent extends HTMLElement {...}
window.customElements.define('a-comp', AComponent);
```

Does it seem similar to something you've been doing for a while?

Right, this is very similar to what we're doing in Angular when defining a component. Actually, Angular follows the web components spec pretty closely but simplifies many things for us so we don't have to create shadow root yourself and attach it to the host element. However, the components that we create in Angular are not registered as custom elements and are processed by the framework in a very specific way. If you're curios how you can create a component without any framework read Custom Elements v1: Reusable Web Components.

Okay, so we've seen that we can create any HTML tag and use it in a template. It should come as no surprise then that if we use it in Angular's component template the framework will create an element definition for this tag:

```
function View_AppComponent_0(_l) {
    return jit_viewDef2(0, [
        jit_elementDef3(0, null, null, 1, 'a-comp', [], ...)
    ])
}
```

However you would have to indicate to Angular that you're using custom element by adding `schemas: [CUSTOM_ELEMENTS_SCHEMA]` to the module or component decorator properties or otherwise Angular compiler will generate an error:

```
'a-comp' is not a known element:
1. If 'a-comp' is an Angular component, then ...
2. If 'a-comp' is a Web Component then add...
```

So, we have an element but we're missing the class. Is there anything in Angular that has class besides a component? Sure there is—a directive! Let's add a directive and see what we end up with.

Directive definition

You probably know that each directive has a selector which can be used to target a specific DOM element. Most of the directives use attribute selectors but element selectors are also perfectly fine. In fact, Angular form directive uses element selector `form` to implicitly attach specific behavior to html forms.

So we can create a directive that does nothing and apply it to our custom element. Let's do that and see what the view definition will look like:

```
@Directive({selector: 'a-comp'})
export class ADirective {}
```

And now let's check the factory:

```
function View_AppComponent_0(_l) {
    return jit_viewDef2(0, [
        jit_elementDef3(0, null, null, 1, 'a-comp', [],
...),
        jit_directiveDef4(16384, null, 0, jit_ADirective5,
[],...)
    ], null, null);
}
```

All right, so now the compiler added the new `jit_directiveDef4` node to the view definition alongside element definition. It also set `childCount` parameter for the element definition to `1` because all directives applied to an element are considered children of that element.

The newly added directive definition is a pretty simple node definition that is generated by the directiveDef function. It takes the following parameters:

```
+----------------+-----------------------------------------
-+
|      Name      |                Description
|
+----------------+-----------------------------------------
-+
| matchedQueries | used when querying child nodes
|
| childCount     | specifies how many children
|
|                | the current element have
|
| ctor           | reference to the component or
|
|                | directive constructor
|
| deps           | an array of constructor dependencies
|
| props          | an array of input property bindings
|
| outputs        | an array of output property bindings
|
+----------------+-----------------------------------------
-+
```

For the purposes of this article we're only interested in the `ctor` parameter. This is just a reference to the class `ADirective` that we defined for the directive. When Angular will be creating directive instances (I'll be writing about that soon, so make sure to follow me 😃) it will instantiate a directive class here. It will then store it as provider data on the view node.

Okay, so our experiments show that a component is just an element and directive definition. Is it just it? As you probably know it's always not that simple with Angular.

• • •

Representing a component

I've shown above how we can emulate a component by creating a custom HTML element and a directive that targets this element. Let's now define a real component and compare the generated factory with the one we got in our experiments:

```
@Component({
  selector: 'a-comp',
  template: '<span>I am A component</span>'
})
export class AComponent {}
```

Ready to compare now? Here is the generated factory:

```
function View_AppComponent_0() {
    return jit_viewDef2(0, [
        jit_elementDef3(0, null, null, 1, 'a-comp', [], ...
                    jit_View_AComponent_04,
jit__object_Object_5),

        jit_directiveDef6(49152, null, 0, jit_AComponent7,
[], ...)
```

Okay, so we just confirmed what followed from the previous chapters. Indeed, Angular represents a component as two view nodes—an element and a directive definition. But when using a real component there are some differences in the parameters list to the element and the directive definition nodes. Let's explore them.

Node flags

Node flags is the first parameter to all node definitions. It is actually a bitmask of node flags that contain specific node information used by the framework mostly during change detection cycle. And this number is different in both cases: `16384` —for the simple directive and `49152` for the component directive. To understand what flags have been set by the compiler let's simply convert the numbers into binary form:

```
16384 =  100000000000000 // 15th bit set
49152 = 1100000000000000 // 15th and 16th bit set
```

If you're curios about how the conversion takes place read The simple math behind decimal-binary conversion algorithms. So, for the simple directive the compiler sets only `15-th` bit which is done like this in Angular sources:

```
TypeDirective = 1 << 14
```

and for the component node both `15-th` and `16-th` bits are set, which is

```
TypeDirective = 1 << 14
Component = 1 << 15
```

Now it should be clear why the numbers differ. The node generated for a directive is marked as `TypeDirective` node and the node generated for the component directive is additionally marked as `Component`.

View definition resolver

Since `a-comp` is a component now with the following simple template:

```
<span>I am A component</span>
```

the compiler generates a factory for it with the its own view definition and view nodes:

```
function View_AComponent_0(_l) {
    return jit_viewDef1(0, [
        jit_elementDef2(0, null, null, 1, 'span', [], ...),
        jit_textDef3(null, ['I am A component'])
```

Angular is a tree of views so parent view definition need to have a reference to the child views definitions. The child views definitions are stored on the element nodes generated for components. In our case, the element definition node generated for the `a-comp` will hold the view for `a-comp`. And the `jit_View_AComponent_04` parameter that `a-comp` element node receives is a reference to the proxy class that will resolve the factory that will create a view definition. Each view definition is created only once and then stored on the

DEFINITION_CACHE. This view definition is then used when Angular creates view instance.

Component renderer type

Angular uses several DOM renders depending on the ViewEncapsulation mode specified in the component decorator:

- Emulated encapsulation renderer
- Shadow renderer
- Default renderer

The renderer for a component is created by the DomRendererFactory2 class. The parameter `componentRendererType` that is passed inside the definition—in our case it's `jit__object_Object_5` —is basically a descriptor of the renderer that needs to be created for the component. The most important information it holds is the view encapsulation mode and styles that need to be applied to the component view:

```
{
  styles:[["h1[_ngcontent-%COMP%] {color: green}"]],
  encapsulation:0
}
```

If you define any styles for your component the compiler automatically sets encapsulation mode for your component to `ViewEncapsulation.Emulated` . Or you can specify the mode explicitly with the `encapsulation` component decorator property. If you don't set any styles and don't specify encapsulation mode for your component the descriptor is defined as `ViewEncapsulation.Emulated` and is in effect ignored. The component with such a descriptor will be using parent component renderer.

• • •

Child directives

Now, the last piece of information remains is what will be generated if we apply a directive to a component in the template like this:

```
<a-comp adir></a-comp>
```

We already know that when generating factory for the `AComponent` the compiler will create an element definition for the `a-comp` HTML element and a directive definition for the `AComponent` class. But since the compiler generates directive definition node for each directive the factory for the above template will look like this:

```
function View_AppComponent_0() {
    return jit_viewDef2(0, [
        jit_elementDef3(0, null, null, 2, 'a-comp', [], ...
    jit_View_AComponent_04, jit__object_Object_5),

    jit_directiveDef6(49152, null, 0, jit_AComponent7, [],
...)
    jit_directiveDef6(16384, null, 0, jit_ADirective8, [],
...)
```

And it contains nothing we haven't seen before. Just one more directive definition was added and the child count for the element was increased to 2.

That's it. Whew 😓 !

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story. For more insights follow me on Twitter and on Medium.