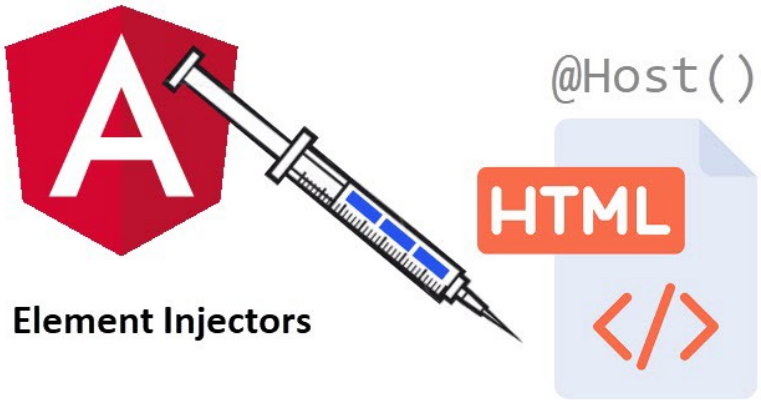
A curious case of the @Host decorator and Element Injectors in Angular

Max Koretskyi aka Wizard Follow
Jun 7, 2018 · 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.

As you know, Angular's dependency injection mechanism includes a bunch of decorators like `@Optional` and `@Self` which impact the way dependencies are resolved. And while most of them are pretty straightforward and self-explanatory, the `@Host` decorator has puzzled me for a long time. I haven't found the docs that explain it, except for a comment in the sources:

> *Specifies that an injector should retrieve a dependency from any injector until reaching the host element of the current component.*

Since most tutorials on the web mention module and component injectors in Angular, I figured it's related to a component's injectors

hierarchy. My assumption was that this decorator can be applied in a child component to restrict resolution only to itself and a parent's component injector. So I put together a small example to test this hypothesis:

```
@Component({
    selector: 'my-app',
    template: `<a-comp></a-comp>`,
    providers: [MyAppService]
})
export class AppComponent {}

@Component({selector: 'a-comp', ...})
```

Alas, it produced the `No provider for MyAppServic` error. Interestingly, if I remove the `@Host` decorator the `MyAppService` is resolved from the parent component as expected. So what's going on here? To find out, I rolled up my sleeves and started investigating 🕵. Let me share with you what I've found.

At the risk of jumping ahead, I'll tell you now that the word `until` in the definition I mentioned above is very important:

> *...retrieve a dependency from any injector **until** reaching the host element*

It means that the `@Host` decorator only applies to the resolution process inside a component's template and doesn't even get to the host element. That's why I got the error in my demo—Angular wouldn't resolve a dependency from a host parent component.

So now we know that the `@Host` decorator can't be used in child components to resolve providers from parent components. It means that this decorator's resolution mechanism is not using a hierarchy of component injectors.

So what kind of injectors hierarchy does it use?

Well, as it turns Angular has a third type of injectors besides modules and components. It's the **element injectors** hierarchy that is created by HTML elements and directives.

> 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!**

. . .

Element Injectors

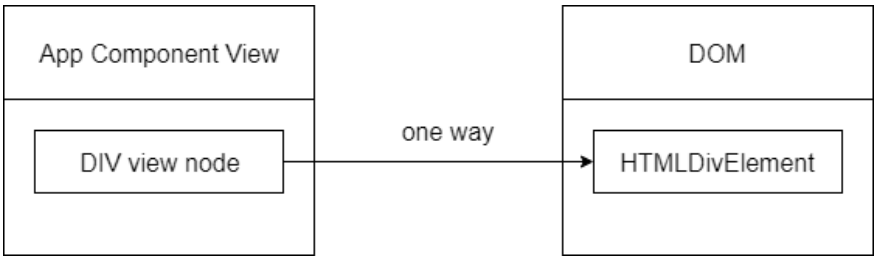
Angular resolves dependencies in 3 stages starting with the hierarchy of element injectors and moving up to component injectors and then module injectors. If you're interested to learn about the details of an entire resolution process I highly recommend checking out this in-depth article by Alexey Zuev.

The last two stages of the resolution process that goes upwards through module and component injectors should be familiar to you. Angular creates a hierarchy of module injectors when you lazy load a module. I've explained this process in details in my talk at NgConf and have written an article. The hierarchy of component injectors is created by nesting components in templates. Internally, component injectors are also referred to as View Injectors and we'll see shortly why.

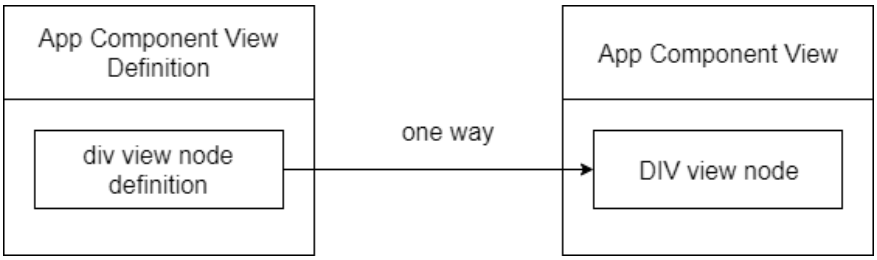
A hierarchy of **element injectors**, on the other hand, is a lesser-known feature of Angular's DI system mostly because it's not really documented anywhere. But exactly this kind of injectors is the **first stage of the DI resolution process**. And these injectors make up a hierarchy that is used to resolve dependencies decorated with the `@Host` decorator. So let's take a look at this kind of injectors.

An element injector

As you probably know from my previous articles, Angular internally represents a component using a data structure commonly referred to as a **View** or a **Component View**. Actually, that is where the name View Injector that refers to Component Injector comes from. The main purpose of a view is to hold references to DOM nodes created for HTML elements specified in a component's template. So, internally, each view consists of different kinds of view nodes. The most common type is `element node` that holds a reference to a corresponding DOM element. Here is a diagram that represents a relationship between a view and DOM:



Each view node is created using a node definition that holds metadata describing the node. For example, a type of a node, like `element` type used to hold DOM element references. This metadata is generated by a compiler based on the component's template and directives applied to each element. Here is a diagram that represents a relationship between a view node definition and its instance:



A node definition describing a node of `element` type has one interesting peculiarity.

> *In Angular, a node definition that describes an HTML element defines its own injector. In other words, an HTML element in a component's template defines its own element injector. And this injector can populated with providers by applying one or more directives on the corresponding HTML element.*

Let's see an example.

Suppose you have a component's template with one `div` element and two directives `A` and `B` applied to it:

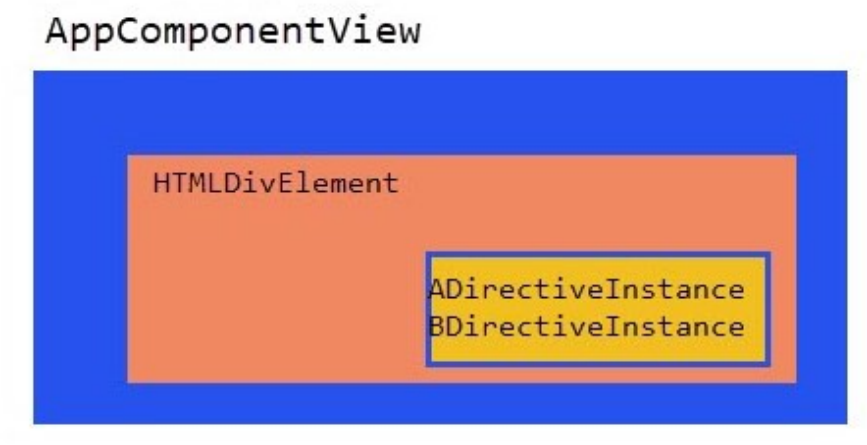
```
1  @Component({
2      selector: 'my-app',
3      template: `<div a b></div>`
4  })
5  export class AppComponent {}
6
7  @Directive({ selector: '[a]' })
8  export class ADirective {}
```

The definition that Angular creates for this template includes the following metadata for the `div` element:

```
const DivElementNodeDefinition = {
    element: {
        name: 'div',
        publicProviders: {
            ADirective: referenceToADirectiveProviderDe
            BDirective: referenceToBDirectiveProviderDe
        }
```

As you can see, this node definition defines `element.publicProviders` **property that acts as an injector** with two providers `ADirective` and `BDirective`. These are actually directive class instances applied to the `div` element. And since they are provided by the same element injector, you can inject one directive instance into the other. Of course, they can't cross inject each other because it would be impossible to instantiating one without instantiating the other.

So here is diagram that illustrates what we have now:



Notice that a host `app-comp` element is outside the `AppComponentView` because it belongs to a parent view.

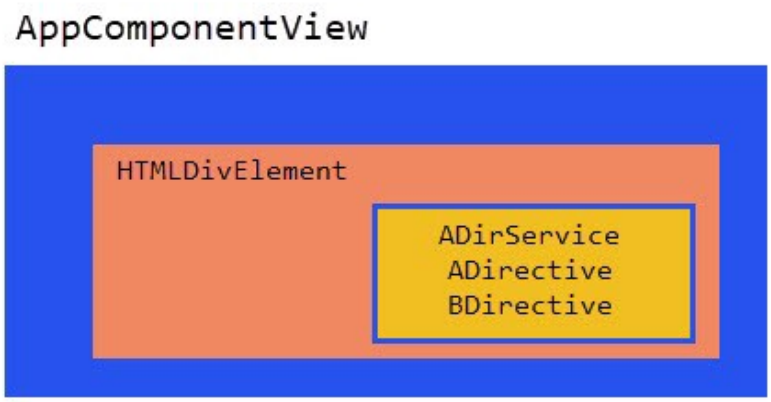
Now what do you think will happen if the directive `A` declares a provider?

```
@Directive({
    selector: '[a]',
    providers: [ADirService]
})
```

Well, as expected, that provider will be added to the element injector created by the `div` element:

```
const divElementNodeDefinition = {
    element: {
        name: `div`,
        publicProviders: {
            ADirService: referenceToADirServiceProvider
            ADirective: referenceToADirectiveProviderDe
        }
```

Again, if we put it on a diagram, here is what we have now:



Hierarchy of element injectors

In the section above we had only one HTML element. Nested HTML elements make up a hierarchy of DOM elements and **in Angular's DI system these elements constitute a hierarchy of element injectors** within a component's view.

Let's see an example.

Suppose you have a component's template with one parent and one child `div` elements. Also, we have two directives `A` and `B`. The directive `A` is applied to the parent `div` and declares `ADirService`

provider. The directive `B` is applied to the child `div` and doesn't declare any providers.

Here is the code that demonstrates the setup:

```
@Component({
    selector: 'my-app',
    template: `
        <div a>
            <div b></div>
        </div>
    `
})
export class AppComponent {}
@Directive({
    selector: '[a]',
```

If we now explore the definition that Angular creates for this template, we will find two nodes of type `element` that describe metadata for both `div` elements:

```
const viewDefinitionNodes = [
    {
        // element definition for the parent div
        element: {
            name: `div`,
            publicProviders: {
                ADirective: referenceToADirectiveProvi
                ADirService: referenceToADirServicePro
            }
        }
    },
    {
        // element definition for the child div
        element: {
```

As we discovered in the previous section, each `div` element definition has `publicProviders` property that acts as a DI container. And since `A` directive applied to a parent `div` also defines `ADirService` provider, it's added to the element injector of a parent `div`.

This nested HTML structure creates a hierarchy of element injectors

Interestingly, a child component also creates an element injector that is part of element injectors hierarchy. For example, the following template:

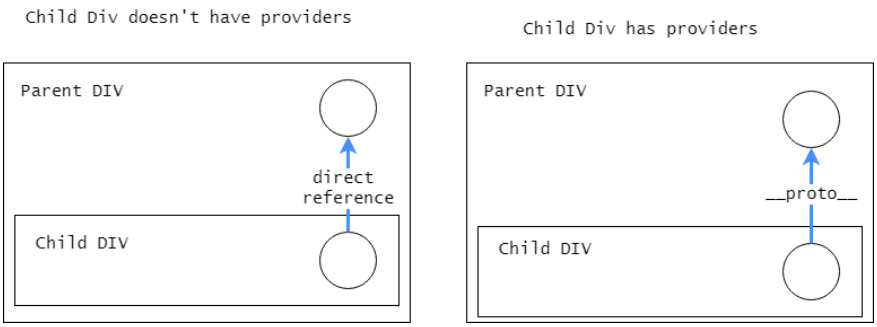
```
<div adir>
    <a-comp></a-comp>
</div>
```

where `adir` declares a provider creates a hierarchy of two element injectors—parent injector created on the `div` element and the child injector created on the `a-comp` element. And it's not surprising because a component is mostly an HTML element with a component directive applied to it.

Creating element injectors

When Angular creates an element injector for a nested HTML element, it either inherits it from a parent's element injector or directly assigns a parent's element injector to the child node definition. A prototype based inheritance between element injectors is only created if directives applied to a child element declare providers. In other words, if an element injector on the child HTML element has providers, the injector should be inherited. Otherwise, there's no need to create a separate injector for a child component and if needed the dependencies can be resolved directly from a parent's injector.

Here is a diagram that demonstrates this behavior:



Resolution process

Setting up a hierarchy between element injectors inside a component's view simplifies the resolution process in element injectors. Instead of coming up with its own implementation of injector's traversal, Angular relies on the JavaScript's mechanism of a property lookup in the prototype chain to resolve a dependency in one step:

```
elDef.element.publicProviders[tokenKey]
```

And because of the way JavaScript works, a key in the `publicProviders` object will be resolved either directly from a parent's element injector or through the prototype chain.

@Host decorator

So why are we talking about element injectors and not the `@Host` decorator? It's because what this decorator does is to simply **restrict a lookup to element injectors within one view**. During the regular DI resolution process, if a token can't be resolved using element injectors inside a view, Angular traverses parent views and checks view/component injectors. If not found, then module injectors are traversed and checked. But when the `@Host` decorator is used, the process stops at the first stage of resolving a dependency in element injectors within one component view.

Examples

The `@Host` decorator is heavily used inside built-in form directives. For example, to inject a hosting form into the `ngModel` directive and register a form created by the directive with the form. This is typical markup for a template driven form:

```
<form>
    <input ngModel>
</form>
```

Under the hood, the `form` element is matched by a selector of the `NgForm` directive that registers itself as a `ControlContainer` provider:

```
@Directive({
    selector: 'form',
    providers: [
        {
            provide: ControlContainer,
            useExisting: NgForm
        }
    ]
```

The `ngModel` directive, in turn, injects a parent form using the same `ControlContainer` token and uses it to register a control with the form:

```
@Directive({
    selector: '[ngModel]',
})
export class NgModel {
    constructor(@Optional() @Host() parent: ControlCon
    private _setUpControl(): void {
        ...
```

As you can see, it uses the `@Host` decorator to restrict the resolution process only to the current component's template. In most cases it's exactly the desired behavior, but sometimes in nested forms you need to inject a hosting form from a parent component. Our curios friend Alexey Zuev has found a way to do that and has written an article. Check it out.

The article I referenced above also mentions another interesting behavior. If I tweak a little bit the example I started this article with by registering `MyAppService` in the `viewProviders` instead of `providers`:

```
@Component({
    selector: 'my-app',
    template: `<a-comp></a-comp>`,
    viewProviders: [MyAppService]
})
export class AppComponent {}

@Component({selector: 'a-comp', ...})
```

it gets resolved and is successfully injected into the child component.

It works because Angular has an additional check for `viewProviders` on the parent component when resolving a dependency decorated with `@Host` :

```
// check @Host restriction
if (!result) {
    if (!dep.isHost || this.viewContext.component.isHo
        this.viewContext.component.type.reference ===
        // this line
        this.viewContext.viewProviders.get(tokenRefere
        result = dep;
    } else {
```

. . .

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.