Hierarchical injectors

Injectors in Angular have rules that you can leverage to achieve the desired visibility of injectables in your applications. By understanding these rules, you can determine in which NgModule, Component or Directive you should declare a provider.

Two injector hierarchies

There are two injector hierarchies in Angular:

1. `ModuleInjector` hierarchy—configure a `ModuleInjector` in this hierarchy using an `@NgModule()` or `@Injectable()` annotation.
2. `ElementInjector` hierarchy—created implicitly at each DOM element. An `ElementInjector` is empty by default unless you configure it in the `providers` property on `@Directive()` or `@Component()`.

{@a register-providers-injectable}

`ModuleInjector`

The `ModuleInjector` can be configured in one of two ways:

- Using the `@Injectable()` `providedIn` property to refer to `@NgModule()`, or `root`.
- Using the `@NgModule()` `providers` array.

Tree-shaking and `@Injectable()`

Using the `@Injectable()` `providedIn` property is preferable to the `@NgModule()` `providers` array because with `@Injectable()` `providedIn`, optimization tools can perform tree-shaking, which removes services that your application isn't using and results in smaller bundle sizes.

Tree-shaking is especially useful for a library because the application which uses the library may not have a need to inject it. Read more about tree-shakable providers in Introduction to services and dependency injection.

`ModuleInjector` is configured by the `@NgModule.providers` and `NgModule.imports` property. `ModuleInjector` is a flattening of all of the providers arrays which can be reached by following the `NgModule.imports` recursively.

Child `ModuleInjector`s are created when lazy loading other `@NgModules`.

Provide services with the `providedIn` property of `@Injectable()` as follows:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'  // <--provides this service in the root ModuleInjector
})
export class ItemService {
  name = 'telephone';
}
```

The `@Injectable()` decorator identifies a service class. The `providedIn` property configures a specific `ModuleInjector`, here `root`, which makes the service available in the `root` `ModuleInjector`.

Platform injector

There are two more injectors above `root`, an additional `ModuleInjector` and `NullInjector()`.

Consider how Angular bootstraps the application with the following in `main.ts`:

```
platformBrowserDynamic().bootstrapModule(AppModule).then(ref => {...})
```

The `bootstrapModule()` method creates a child injector of the platform injector which is configured by the `AppModule`. This is the `root` `ModuleInjector`.

The `platformBrowserDynamic()` method creates an injector configured by a `PlatformModule`, which contains platform-specific dependencies. This allows multiple applications to share a platform configuration. For example, a browser has only one URL bar, no matter how many applications you have running. You can configure additional platform-specific providers at the platform level by supplying `extraProviders` using the `platformBrowser()` function.

The next parent injector in the hierarchy is the `NullInjector()`, which is the top of the tree. If you've gone so far up the tree that you are looking for a service in the `NullInjector()`, you'll get an error unless you've used `@Optional()` because ultimately, everything ends at the `NullInjector()` and it returns an error or, in the case of `@Optional()`, `null`. For more information on `@Optional()`, see the `@Optional()` section of this guide.

The following diagram represents the relationship between the `root` `ModuleInjector` and its parent injectors as the previous paragraphs describe.


NullInjector, ModuleInjector, root injector

While the name `root` is a special alias, other `ModuleInjector`s don't have aliases. You have the option to create `ModuleInjector`s whenever a dynamically loaded component is created, such as with the Router, which will create child `ModuleInjector`s.

All requests forward up to the root injector, whether you configured it with the `bootstrapModule()` method, or registered all providers with `root` in their own services.

`@Injectable()` vs. `@NgModule()`

If you configure an app-wide provider in the `@NgModule()` of `AppModule`, it overrides one configured for `root` in the `@Injectable()` metadata. You can do this to configure a non-default provider of a service that is shared with multiple applications.

Here is an example of the case where the component router configuration includes a non-default location strategy by listing its provider in the `providers` list of the `AppModule`.

`ElementInjector`

Angular creates `ElementInjector`s implicitly for each DOM element.

Providing a service in the `@Component()` decorator using its `providers` or `viewProviders` property configures an `ElementInjector`. For example, the following `TestComponent` configures the `ElementInjector` by providing the service as follows:

```
@Component({
  ...
  providers: [{ provide: ItemService, useValue: { name: 'lamp' } }]
})
export class TestComponent
```

Note: See the resolution rules section to understand the relationship between the `ModuleInjector` tree and the `ElementInjector` tree.

When you provide services in a component, that service is available by way of the `ElementInjector` at that component instance. It may also be visible at child component/directives based on visibility rules described in the resolution rules section.

When the component instance is destroyed, so is that service instance.

`@Directive()` and `@Component()`

A component is a special type of directive, which means that just as `@Directive()` has a `providers` property, `@Component()` does too. This means that directives as well as components can configure providers, using the `providers` property. When you configure a provider for a component or directive using the `providers` property, that provider belongs to the `ElementInjector` of that component or directive. Components and directives on the same element share an injector.

{@a resolution-rules}

Resolution rules

When resolving a token for a component/directive, Angular resolves it in two phases:

1. Against the `ElementInjector` hierarchy (its parents)
2. Against the `ModuleInjector` hierarchy (its parents)

When a component declares a dependency, Angular tries to satisfy that dependency with its own `ElementInjector`. If the component's injector lacks the provider, it passes the request up to its parent component's `ElementInjector`.

The requests keep forwarding up until Angular finds an injector that can handle the request or runs out of ancestor `ElementInjector`s.

If Angular doesn't find the provider in any `ElementInjector`s, it goes back to the element where the request originated and looks in the `ModuleInjector` hierarchy. If Angular still doesn't find the provider, it throws an error.

If you have registered a provider for the same DI token at different levels, the first one Angular encounters is the one it uses to resolve the dependency. If, for example, a provider is registered locally in the component that needs a service, Angular doesn't look for another provider of the same service.

Resolution modifiers

Angular's resolution behavior can be modified with `@Optional()`, `@Self()`, `@SkipSelf()` and `@Host()`. Import each of them from `@angular/core` and use each in the component class constructor when you inject your service.

For a working application showcasing the resolution modifiers that this section covers, see the resolution modifiers example.

Types of modifiers

Resolution modifiers fall into three categories:

1. What to do if Angular doesn't find what you're looking for, that is `@Optional()`
2. Where to start looking, that is `@SkipSelf()`
3. Where to stop looking, `@Host()` and `@Self()`

By default, Angular always starts at the current `Injector` and keeps searching all the way up. Modifiers allow you to change the starting (self) or ending location.

Additionally, you can combine all of the modifiers except `@Host()` and `@Self()` and of course `@SkipSelf()` and `@Self()`.

{@a optional}

`@Optional()`

`@Optional()` allows Angular to consider a service you inject to be optional. This way, if it can't be resolved at runtime, Angular resolves the service as `null`, rather than throwing an error. In the following example, the service, `OptionalService`, isn't provided in the service, `@NgModule()`, or component class, so it isn't available anywhere in the app.

`@Self()`

Use `@Self()` so that Angular will only look at the `ElementInjector` for the current component or directive.

A good use case for `@Self()` is to inject a service but only if it is available on the current host element. To avoid errors in this situation, combine `@Self()` with `@Optional()`.

For example, in the following `SelfComponent`, notice the injected `LeafService` in the constructor.

In this example, there is a parent provider and injecting the service will return the value, however, injecting the service with `@Self()` and `@Optional()` will return `null` because `@Self()` tells the injector to stop searching in the current host element.

Another example shows the component class with a provider for `FlowerService`. In this case, the injector looks no further than the current `ElementInjector` because it finds the `FlowerService` and returns the yellow flower 🌼.

`@SkipSelf()`

`@SkipSelf()` is the opposite of `@Self()`. With `@SkipSelf()`, Angular starts its search for a service in the parent `ElementInjector`, rather than in the current one. So if the parent `ElementInjector` were using the value 🌿 (fern) for `emoji`, but you had 🍁 (maple leaf) in the component's `providers` array, Angular would ignore 🍁 (maple leaf) and use 🌿 (fern).

To see this in code, assume that the following value for `emoji` is what the parent component were using, as in this service:

Imagine that in the child component, you had a different value, 🍁 (maple leaf) but you wanted to use the parent's value instead. This is when you'd use `@SkipSelf()`:

In this case, the value you'd get for `emoji` would be 🌿 (fern), not 🍁 (maple leaf).

`@SkipSelf()` with `@Optional()`

Use `@SkipSelf()` with `@Optional()` to prevent an error if the value is `null`. In the following example, the `Person` service is injected in the constructor. `@SkipSelf()` tells Angular to skip the current injector and `@Optional()` will prevent an error should the `Person` service be `null`.

```
class Person {
  constructor(@Optional() @SkipSelf() parent?: Person) {}
}
```

`@Host()`

`@Host()` lets you designate a component as the last stop in the injector tree when searching for providers. Even if there is a service instance further up the tree, Angular won't continue looking. Use `@Host()` as follows:

Since `HostComponent` has `@Host()` in its constructor, no matter what the parent of `HostComponent` might have as a `flower.emoji` value, the `HostComponent` will use 🌼 (yellow flower).

Logical structure of the template

When you provide services in the component class, services are visible within the `ElementInjector` tree relative to where and how you provide those services.

Understanding the underlying logical structure of the Angular template will give you a foundation for configuring services and in turn control their visibility.

Components are used in your templates, as in the following example:

```
<app-root>
    <app-child></app-child>
</app-root>
```

Note: Usually, you declare the components and their templates in separate files. For the purposes of understanding how the injection system works, it is useful to look at them from the point of view of a combined logical tree. The term logical distinguishes it from the render tree (your application DOM tree). To mark the locations of where the component templates are located, this guide uses the `<#VIEW>` pseudo element, which doesn't actually exist in the render tree and is present for mental model purposes only.

The following is an example of how the `<app-root>` and `<app-child>` view trees are combined into a single logical tree:

```
<app-root>
  <#VIEW>
    <app-child>
```

```
      <#VIEW>
        ...content goes here...
      </#VIEW>
    </app-child>
  </#VIEW>
</app-root>
```

Understanding the idea of the `<#VIEW>` demarcation is especially significant when you configure services in the component class.

Providing services in `@Component()`

How you provide services using a `@Component()` (or `@Directive()`) decorator determines their visibility. The following sections demonstrate `providers` and `viewProviders` along with ways to modify service visibility with `@SkipSelf()` and `@Host()` .

A component class can provide services in two ways:

1. with a `providers` array

```
@Component({
  ...
  providers: [
    {provide: FlowerService, useValue: {emoji: '🌺'}}
  ]
})
```

2. with a `viewProviders` array

```
@Component({
  ...
  viewProviders: [
    {provide: AnimalService, useValue: {emoji: '🐶'}}
  ]
})
```

To understand how the `providers` and `viewProviders` influence service visibility differently, the following sections build a step-by-step and compare the use of `providers` and `viewProviders` in code and a logical tree.

NOTE: In the logical tree, you'll see `@Provide` , `@Inject` , and `@NgModule` , which are not real HTML attributes but are here to demonstrate what is going on under the hood.

- `@Inject(Token)=>Value` demonstrates that if `Token` is injected at this location in the logical tree its value would be `Value` .
- `@Provide(Token=Value)` demonstrates that there is a declaration of `Token` provider with value `Value` at this location in the logical tree.
- `@NgModule(Token)` demonstrates that a fallback `NgModule` injector should be used at this location.

Example app structure

The example application has a `FlowerService` provided in `root` with an `emoji` value of 🌺 (red hibiscus).

Consider an application with only an `AppComponent` and a `ChildComponent`. The most basic rendered view would look like nested HTML elements such as the following:

```
<app-root> <!-- AppComponent selector -->
    <app-child> <!-- ChildComponent selector -->
    </app-child>
</app-root>
```

However, behind the scenes, Angular uses a logical view representation as follows when resolving injection requests:

```
<app-root> <!-- AppComponent selector -->
    <#VIEW>
        <app-child> <!-- ChildComponent selector -->
            <#VIEW>
            </#VIEW>
        </app-child>
    </#VIEW>
</app-root>
```

The `<#VIEW>` here represents an instance of a template. Notice that each component has its own `<#VIEW>`.

Knowledge of this structure can inform how you provide and inject your services, and give you complete control of service visibility.

Now, consider that `<app-root>` injects the `FlowerService`:

Add a binding to the `<app-root>` template to visualize the result:

The output in the view would be:

```
Emoji from FlowerService: 🌺
```

In the logical tree, this would be represented as follows:

```
<app-root @NgModule(AppModule)
        @Inject(FlowerService) flower=>"🌺">
  <#VIEW>
    <p>Emoji from FlowerService: {{flower.emoji}} (🌺)</p>
    <app-child>
      <#VIEW>
      </#VIEW>
     </app-child>
  </#VIEW>
</app-root>
```

When `<app-root>` requests the `FlowerService`, it is the injector's job to resolve the `FlowerService` token. The resolution of the token happens in two phases:

1. The injector determines the starting location in the logical tree and an ending location of the search. The injector begins with the starting location and looks for the token at each level in the logical tree. If the token is found it is returned.

2. If the token is not found, the injector looks for the closest parent `@NgModule()` to delegate the request to.

In the example case, the constraints are:

1. Start with `<#VIEW>` belonging to `<app-root>` and end with `<app-root>`.

- Normally the starting point for search is at the point of injection. However, in this case `<app-root>` `@Component` s are special in that they also include their own `viewProviders`, which is why the search starts at `<#VIEW>` belonging to `<app-root>`. (This would not be the case for a directive matched at the same location).
- The ending location happens to be the same as the component itself, because it is the topmost component in this application.

2. The `AppModule` acts as the fallback injector when the injection token can't be found in the `ElementInjector` s.

Using the `providers` array

Now, in the `ChildComponent` class, add a provider for `FlowerService` to demonstrate more complex resolution rules in the upcoming sections:

Now that the `FlowerService` is provided in the `@Component()` decorator, when the `<app-child>` requests the service, the injector has only to look as far as the `<app-child>` 's own `ElementInjector`. It won't have to continue the search any further through the injector tree.

The next step is to add a binding to the `ChildComponent` template.

To render the new values, add `<app-child>` to the bottom of the `AppComponent` template so the view also displays the sunflower:

```
Child Component
Emoji from FlowerService: 🌻
```

In the logical tree, this would be represented as follows:

```
<app-root @NgModule(AppModule)
        @Inject(FlowerService) flower=>"🌺">
  <#VIEW>
    <p>Emoji from FlowerService: {{flower.emoji}} (🌺)</p>
    <app-child @Provide(FlowerService="🌻")
               @Inject(FlowerService)=>"🌻"> <!-- search ends here -->
      <#VIEW> <!-- search starts here -->
        <h2>Parent Component</h2>
        <p>Emoji from FlowerService: {{flower.emoji}} (🌻)</p>
      </#VIEW>
     </app-child>
  </#VIEW>
</app-root>
```

When `<app-child>` requests the `FlowerService`, the injector begins its search at the `<#VIEW>` belonging to `<app-child>` (`<#VIEW>` is included because it is injected from `@Component()`) and ends with `<app-child>`. In this case, the `FlowerService` is resolved in the `<app-child>` 's `providers` array with

sunflower 🌻. The injector doesn't have to look any further in the injector tree. It stops as soon as it finds the `FlowerService` and never sees the 🌺 (red hibiscus).

{@a use-view-providers}

Using the `viewProviders` array

Use the `viewProviders` array as another way to provide services in the `@Component()` decorator. Using `viewProviders` makes services visible in the `<#VIEW>`.

The steps are the same as using the `providers` array, with the exception of using the `viewProviders` array instead.

For step-by-step instructions, continue with this section. If you can set it up on your own, skip ahead to Modifying service availability.

The example application features a second service, the `AnimalService` to demonstrate `viewProviders`.

First, create an `AnimalService` with an `emoji` property of 🐳 (whale):

Following the same pattern as with the `FlowerService`, inject the `AnimalService` in the `AppComponent` class:

Note: You can leave all the `FlowerService` related code in place as it will allow a comparison with the `AnimalService`.

Add a `viewProviders` array and inject the `AnimalService` in the `<app-child>` class, too, but give `emoji` a different value. Here, it has a value of 🐶 (puppy).

Add bindings to the `ChildComponent` and the `AppComponent` templates. In the `ChildComponent` template, add the following binding:

Additionally, add the same to the `AppComponent` template:

Now you should see both values in the browser:

```
AppComponent
Emoji from AnimalService: 🐳

Child Component
Emoji from AnimalService: 🐶
```

The logic tree for this example of `viewProviders` is as follows:

```
<app-root @NgModule(AppModule)
        @Inject(AnimalService) animal=>"🐳">
  <#VIEW>
    <app-child>
      <#VIEW
       @Provide(AnimalService="🐶")
       @Inject(AnimalService=>"🐶")>
       <!-- ^^using viewProviders means AnimalService is available in <#VIEW>-->
       <p>Emoji from AnimalService: {{animal.emoji}} (🐶)</p>
      </#VIEW>
```

```
      </app-child>
  </#VIEW>
</app-root>
```

Just as with the `FlowerService` example, the `AnimalService` is provided in the `<app-child>` `@Component()` decorator. This means that since the injector first looks in the `ElementInjector` of the component, it finds the `AnimalService` value of 🐶 (puppy). It doesn't need to continue searching the `ElementInjector` tree, nor does it need to search the `ModuleInjector`.

`providers` vs. `viewProviders`

To see the difference between using `providers` and `viewProviders`, add another component to the example and call it `InspectorComponent`. `InspectorComponent` will be a child of the `ChildComponent`. In `inspector.component.ts`, inject the `FlowerService` and `AnimalService` in the constructor:

You do not need a `providers` or `viewProviders` array. Next, in `inspector.component.html`, add the same markup from previous components:

Remember to add the `InspectorComponent` to the `AppModule` `declarations` array.

Next, make sure your `child.component.html` contains the following:

The first two lines, with the bindings, are there from previous steps. The new parts are `<ng-content>` and `<app-inspector>`. `<ng-content>` allows you to project content, and `<app-inspector>` inside the `ChildComponent` template makes the `InspectorComponent` a child component of `ChildComponent`.

Next, add the following to `app.component.html` to take advantage of content projection.

The browser now renders the following, omitting the previous examples for brevity:

```
//...Omitting previous examples. The following applies to this section.

Content projection: This is coming from content. Doesn't get to see
puppy because the puppy is declared inside the view only.

Emoji from FlowerService: 🌻
Emoji from AnimalService: 🐳

Emoji from FlowerService: 🌻
Emoji from AnimalService: 🐶
```

These four bindings demonstrate the difference between `providers` and `viewProviders`. Since the 🐶 (puppy) is declared inside the <#VIEW>, it isn't visible to the projected content. Instead, the projected content sees the 🐳 (whale).

The next section though, where `InspectorComponent` is a child component of `ChildComponent`, `InspectorComponent` is inside the `<#VIEW>`, so when it asks for the `AnimalService`, it sees the 🐶 (puppy).

The `AnimalService` in the logical tree would look like this:

```
<app-root @NgModule(AppModule)
        @Inject(AnimalService) animal=>"🐳">
  <#VIEW>
    <app-child>
      <#VIEW
       @Provide(AnimalService="🐶")
       @Inject(AnimalService=>"🐶")>
       <!-- ^^using viewProviders means AnimalService is available in <#VIEW>-->
       <p>Emoji from AnimalService: {{animal.emoji}} (🐶)</p>
       <app-inspector>
        <p>Emoji from AnimalService: {{animal.emoji}} (🐶)</p>
       </app-inspector>
      </#VIEW>
      <app-inspector>
        <#VIEW>
          <p>Emoji from AnimalService: {{animal.emoji}} (🐳)</p>
        </#VIEW>
      </app-inspector>
     </app-child>
  </#VIEW>
</app-root>
```

The projected content of `<app-inspector>` sees the 🐳 (whale), not the 🐶 (puppy), because the 🐶 (puppy) is inside the `<app-child>` `<#VIEW>`. The `<app-inspector>` can only see the 🐶 (puppy) if it is also within the `<#VIEW>`.

{@a modify-visibility}

Modifying service visibility

This section describes how to limit the scope of the beginning and ending `ElementInjector` using the visibility decorators `@Host()`, `@Self()`, and `@SkipSelf()`.

Visibility of provided tokens

Visibility decorators influence where the search for the injection token begins and ends in the logic tree. To do this, place visibility decorators at the point of injection, that is, the `constructor()`, rather than at a point of declaration.

To alter where the injector starts looking for `FlowerService`, add `@SkipSelf()` to the `<app-child>` `@Inject` declaration for the `FlowerService`. This declaration is in the `<app-child>` constructor as shown in `child.component.ts`:

```
  constructor(@SkipSelf() public flower : FlowerService) { }
```

With `@SkipSelf()`, the `<app-child>` injector doesn't look to itself for the `FlowerService`. Instead, the injector starts looking for the `FlowerService` at the `<app-root>`'s `ElementInjector`, where it finds nothing. Then, it goes back to the `<app-child>` `ModuleInjector` and finds the 🌺 (red hibiscus) value, which is available because the `<app-child>` `ModuleInjector` and the `<app-root>` `ModuleInjector` are flattened into one `ModuleInjector`. Thus, the UI renders the following:

```
Emoji from FlowerService: 🌺
```

In a logical tree, this same idea might look like this:

```
<app-root @NgModule(AppModule)
        @Inject(FlowerService) flower=>"🌺">
  <#VIEW>
    <app-child @Provide(FlowerService="🌻")>
      <#VIEW @Inject(FlowerService, SkipSelf)=>"🌺">
      <!-- With SkipSelf, the injector looks to the next injector up the tree -->
      </#VIEW>
      </app-child>
  </#VIEW>
</app-root>
```

Though `<app-child>` provides the 🌻 (sunflower), the application renders the 🌺 (red hibiscus) because `@SkipSelf()` causes the current injector to skip itself and look to its parent.

If you now add `@Host()` (in addition to the `@SkipSelf()`) to the `@Inject` of the `FlowerService`, the result will be `null`. This is because `@Host()` limits the upper bound of the search to the `<#VIEW>`. Here's the idea in the logical tree:

```
<app-root @NgModule(AppModule)
        @Inject(FlowerService) flower=>"🌺">
  <#VIEW> <!-- end search here with null-->
    <app-child @Provide(FlowerService="🌻")> <!-- start search here -->
      <#VIEW @Inject(FlowerService, @SkipSelf, @Host, @Optional)=>null>
      </#VIEW>
      </app-parent>
  </#VIEW>
</app-root>
```

Here, the services and their values are the same, but `@Host()` stops the injector from looking any further than the `<#VIEW>` for `FlowerService`, so it doesn't find it and returns `null`.

Note: The example application uses `@Optional()` so the application does not throw an error, but the principles are the same.

`@SkipSelf()` and `viewProviders`

The `<app-child>` currently provides the `AnimalService` in the `viewProviders` array with the value of 🐶 (puppy). Because the injector has only to look at the `<app-child>`'s `ElementInjector` for the `AnimalService`, it never sees the 🐳 (whale).

As in the `FlowerService` example, if you add `@SkipSelf()` to the constructor for the `AnimalService`, the injector won't look in the current `<app-child>`'s `ElementInjector` for the `AnimalService`.

```
export class ChildComponent {

// add @SkipSelf()
  constructor(@SkipSelf() public animal : AnimalService) { }
```

```
}
```

Instead, the injector will begin at the `<app-root>` `ElementInjector`. Remember that the `<app-child>` class provides the `AnimalService` in the `viewProviders` array with a value of 🐶 (puppy):

```
@Component({
  selector: 'app-child',
  ...
  viewProviders:
  [{ provide: AnimalService, useValue: { emoji: '🐶' } }]
})
```

The logical tree looks like this with `@SkipSelf()` in `<app-child>`:

```
<app-root @NgModule(AppModule)
        @Inject(AnimalService=>"💪")>
  <#VIEW><!-- search begins here -->
    <app-child>
      <#VIEW
       @Provide(AnimalService="🐶")
       @Inject(AnimalService, SkipSelf=>"💪")>
       <!--Add @SkipSelf -->
      </#VIEW>
      </app-child>
  </#VIEW>
</app-root>
```

With `@SkipSelf()` in the `<app-child>`, the injector begins its search for the `AnimalService` in the `<app-root>` `ElementInjector` and finds 💪 (whale).

`@Host()` and `viewProviders`

If you add `@Host()` to the constructor for `AnimalService`, the result is 🐶 (puppy) because the injector finds the `AnimalService` in the `<app-child>` `<#VIEW>`. Here is the `viewProviders` array in the `<app-child>` class and `@Host()` in the constructor:

```
@Component({
  selector: 'app-child',
  ...
  viewProviders:
  [{ provide: AnimalService, useValue: { emoji: '🐶' } }]

})
export class ChildComponent {
  constructor(@Host() public animal : AnimalService) { }
}
```

`@Host()` causes the injector to look until it encounters the edge of the `<#VIEW>`.

```
<app-root @NgModule(AppModule)
        @Inject(AnimalService=>"🐳")>
  <#VIEW>
    <app-child>
      <#VIEW
       @Provide(AnimalService="🐶")
       @Inject(AnimalService, @Host=>"🐱")> <!-- @Host stops search here -->
      </#VIEW>
      </app-child>
  </#VIEW>
</app-root>
```

Add a `viewProviders` array with a third animal, 🦔 (hedgehog), to the `app.component.ts` `@Component()` metadata:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ],
  viewProviders: [{ provide: AnimalService, useValue: { emoji: '🦔' } }]
})
```

Next, add `@SkipSelf()` along with `@Host()` to the constructor for the `Animal Service` in `child.component.ts`. Here are `@Host()` and `@SkipSelf()` in the `<app-child>` constructor:

```
export class ChildComponent {

  constructor(
  @Host() @SkipSelf() public animal : AnimalService) { }

}
```

When `@Host()` and `SkipSelf()` were applied to the `FlowerService`, which is in the `providers` array, the result was `null` because `@SkipSelf()` starts its search in the `<app-child>` injector, but `@Host()` stops searching at `<#VIEW>` —where there is no `FlowerService`. In the logical tree, you can see that the `FlowerService` is visible in `<app-child>`, not its `<#VIEW>`.

However, the `AnimalService`, which is provided in the `AppComponent` `viewProviders` array, is visible.

The logical tree representation shows why this is:

```
<app-root @NgModule(AppModule)
        @Inject(AnimalService=>"🐳")>
  <#VIEW @Provide(AnimalService="🦔")
         @Inject(AnimalService, @Optional)=>"🦔">
    <!-- ^^@SkipSelf() starts here,  @Host() stops here^^ -->
    <app-child>
      <#VIEW @Provide(AnimalService="🐱")
             @Inject(AnimalService, @SkipSelf, @Host, @Optional)=>"🦔">
               <!-- Add @SkipSelf ^^-->
```

```
      </#VIEW>
      </app-child>
  </#VIEW>
</app-root>
```

`@SkipSelf()` , causes the injector to start its search for the `AnimalService` at the `<app-root>` , not the `<app-child>` , where the request originates, and `@Host()` stops the search at the `<app-root>` `<#VIEW>` . Since `AnimalService` is provided by way of the `viewProviders` array, the injector finds 🦔 (hedgehog) in the `<#VIEW>` .

{@a component-injectors}

`ElementInjector` use case examples

The ability to configure one or more providers at different levels opens up useful possibilities. For a look at the following scenarios in a working app, see the heroes use case examples.

Scenario: service isolation

Architectural reasons may lead you to restrict access to a service to the application domain where it belongs. For example, the guide sample includes a `VillainsListComponent` that displays a list of villains. It gets those villains from a `VillainsService` .

If you provided `VillainsService` in the root `AppModule` (where you registered the `HeroesService`), that would make the `VillainsService` visible everywhere in the application, including the *Hero* workflows. If you later modified the `VillainsService` , you could break something in a hero component somewhere.

Instead, you can provide the `VillainsService` in the `providers` metadata of the `VillainsListComponent` like this:

By providing `VillainsService` in the `VillainsListComponent` metadata and nowhere else, the service becomes available only in the `VillainsListComponent` and its sub-component tree.

`VillainService` is a singleton with respect to `VillainsListComponent` because that is where it is declared. As long as `VillainsListComponent` does not get destroyed it will be the same instance of `VillainService` but if there are multiple instances of `VillainsListComponent` , then each instance of `VillainsListComponent` will have its own instance of `VillainService` .

Scenario: multiple edit sessions

Many applications allow users to work on several open tasks at the same time. For example, in a tax preparation application, the preparer could be working on several tax returns, switching from one to the other throughout the day.

This guide demonstrates that scenario with an example in the Tour of Heroes theme. Imagine an outer `HeroListComponent` that displays a list of super heroes.

To open a hero's tax return, the preparer clicks on a hero name, which opens a component for editing that return. Each selected hero tax return opens in its own component and multiple returns can be open at the same time.

Each tax return component has the following characteristics:

- Is its own tax return editing session.

- Can change a tax return without affecting a return in another component.
- Has the ability to save the changes to its tax return or cancel them.

 Heroes in action

Suppose that the `HeroTaxReturnComponent` had logic to manage and restore changes. That would be a straightforward task for a hero tax return. In the real world, with a rich tax return data model, the change management would be tricky. You could delegate that management to a helper service, as this example does.

The `HeroTaxReturnService` caches a single `HeroTaxReturn`, tracks changes to that return, and can save or restore it. It also delegates to the application-wide singleton `HeroService`, which it gets by injection.

Here is the `HeroTaxReturnComponent` that makes use of `HeroTaxReturnService`.

The *tax-return-to-edit* arrives by way of the `@Input()` property, which is implemented with getters and setters. The setter initializes the component's own instance of the `HeroTaxReturnService` with the incoming return. The getter always returns what that service says is the current state of the hero. The component also asks the service to save and restore this tax return.

This won't work if the service is an application-wide singleton. Every component would share the same service instance, and each component would overwrite the tax return that belonged to another hero.

To prevent this, configure the component-level injector of `HeroTaxReturnComponent` to provide the service, using the `providers` property in the component metadata.

The `HeroTaxReturnComponent` has its own provider of the `HeroTaxReturnService`. Recall that every component *instance* has its own injector. Providing the service at the component level ensures that *every* instance of the component gets its own, private instance of the service, and no tax return gets overwritten.

The rest of the scenario code relies on other Angular features and techniques that you can learn about elsewhere in the documentation. You can review it and download it from the .

Scenario: specialized providers

Another reason to re-provide a service at another level is to substitute a *more specialized* implementation of that service, deeper in the component tree.

Consider a Car component that depends on several services. Suppose you configured the root injector (marked as A) with *generic* providers for `CarService`, `EngineService` and `TiresService`.

You create a car component (A) that displays a car constructed from these three generic services.

Then you create a child component (B) that defines its own, *specialized* providers for `CarService` and `EngineService` that have special capabilities suitable for whatever is going on in component (B).

Component (B) is the parent of another component (C) that defines its own, even *more specialized* provider for `CarService`.

 car components

Behind the scenes, each component sets up its own injector with zero, one, or more providers defined for that component itself.

When you resolve an instance of `Car` at the deepest component (C), its injector produces an instance of `Car` resolved by injector (C) with an `Engine` resolved by injector (B) and `Tires` resolved by the root injector (A).


car injector tree

More on dependency injection

For more information on Angular dependency injection, see the DI Providers and DI in Action guides.