Navigate the component tree with DI

Marked for archiving

To ensure that you have the best experience possible, this topic is marked for archiving until we determine that it clearly conveys the most accurate information possible.

In the meantime, this topic might be helpful: Hierarchical injectors.

If you think this content should not be archived, please file a GitHub issue.

Application components often need to share information. You can often use loosely coupled techniques for sharing information, such as data binding and service sharing, but sometimes it makes sense for one component to have a direct reference to another component. You need a direct reference, for instance, to access values or call methods on that component.

Obtaining a component reference is a bit tricky in Angular. Angular components themselves do not have a tree that you can inspect or navigate programmatically. The parent-child relationship is indirect, established through the components' view objects.

Each component has a *host view*, and can have additional *embedded views*. An embedded view in component A is the host view of component B, which can in turn have embedded view. This means that there is a view hierarchy for each component, of which that component's host view is the root.

There is an API for navigating *down* the view hierarchy. Check out `Query`, `QueryList`, `ViewChildren`, and `ContentChildren` in the API Reference.

There is no public API for acquiring a parent reference. However, because every component instance is added to an injector's container, you can use Angular dependency injection to reach a parent component.

This section describes some techniques for doing that.

{@a find-parent} {@a known-parent}

Find a parent component of known type

You use standard class injection to acquire a parent component whose type you know.

In the following example, the parent `AlexComponent` has several children including a `CathyComponent`:

{@a alex}

Cathy reports whether or not she has access to *Alex* after injecting an `AlexComponent` into her constructor:

Notice that even though the @Optional qualifier is there for safety, the confirms that the `alex` parameter is set.

{@a base-parent}

Unable to find a parent by its base class

What if you *don't* know the concrete parent component class?

A re-usable component might be a child of multiple components. Imagine a component for rendering breaking news about a financial instrument. For business reasons, this news component makes frequent calls directly into its parent instrument as changing market data streams by.

The app probably defines more than a dozen financial instrument components. If you're lucky, they all implement the same base class whose API your `NewsComponent` understands.

Looking for components that implement an interface would be better. That's not possible because TypeScript interfaces disappear from the transpiled JavaScript, which doesn't support interfaces. There's no artifact to look for.

This isn't necessarily good design. This example is examining *whether a component can inject its parent via the parent's base class.*

The sample's `CraigComponent` explores this question. Looking back, you see that the `Alex` component *extends* (*inherits*) from a class named `Base`.

The `CraigComponent` tries to inject `Base` into its `alex` constructor parameter and reports if it succeeded.

Unfortunately, this doesn't work. The confirms that the `alex` parameter is null. *You cannot inject a parent by its base class.*

{@a class-interface-parent}

Find a parent by its class interface

You can find a parent component with a class interface.

The parent must cooperate by providing an *alias* to itself in the name of a class interface token.

Recall that Angular always adds a component instance to its own injector; that's why you could inject *Alex* into *Cathy* earlier.

Write an *alias provider*—a `provide` object literal with a `useExisting` definition—that creates an *alternative* way to inject the same component instance and add that provider to the `providers` array of the `@Component()` metadata for the `AlexComponent`.

{@a alex-providers}

Parent is the provider's class interface token. The *forwardRef* breaks the circular reference you just created by having the `AlexComponent` refer to itself.

Carol, the third of *Alex*'s child components, injects the parent into its `parent` parameter, the same way you've done it before.

Here's *Alex* and family in action.

{@a parent-tree}

Find a parent in a tree with *@SkipSelf()*

Imagine one branch of a component hierarchy: *Alice* -> *Barry* -> *Carol*. Both *Alice* and *Barry* implement the `Parent` class interface.

Barry is the problem. He needs to reach his parent, *Alice*, and also be a parent to *Carol*. That means he must both *inject* the `Parent` class interface to get *Alice* and *provide* a `Parent` to satisfy *Carol*.

Here's *Barry*.

Barry's `providers` array looks just like *Alex*'s. If you're going to keep writing *alias providers* like this you should create a helper function.

For now, focus on *Barry*'s constructor.

It's identical to *Carol*'s constructor except for the additional `@SkipSelf` decorator.

`@SkipSelf` is essential for two reasons:

1. It tells the injector to start its search for a `Parent` dependency in a component *above* itself, which *is* what parent means.
2. Angular throws a cyclic dependency error if you omit the `@SkipSelf` decorator.

```
NG0200: Circular dependency in DI detected for BethComponent.
Dependency path: BethComponent -> Parent -> BethComponent
```

Here's *Alice*, *Barry*, and family in action.

{@a parent-token}

Parent class interface

You learned earlier that a class interface is an abstract class used as an interface rather than as a base class.

The example defines a `Parent` class interface.

The `Parent` class interface defines a `name` property with a type declaration but *no implementation*. The `name` property is the only member of a parent component

that a child component can call. Such a narrow interface helps decouple the child component class from its parent components.

A component that could serve as a parent *should* implement the class interface as the `AliceComponent` does.

Doing so adds clarity to the code. But it's not technically necessary. Although `AlexComponent` has a `name` property, as required by its `Base` class, its class signature doesn't mention `Parent`.

`AlexComponent` *should* implement `Parent` as a matter of proper style. It doesn't in this example *only* to demonstrate that the code will compile and run without the interface.