

Hierarchical Dependency Injectors

You learned the basics of Angular Dependency injection in the [Dependency Injection](#) guide.

Angular has a *Hierarchical Dependency Injection* system. There is actually a tree of injectors that parallel an application's component tree. You can reconfigure the injectors at any level of that component tree.

This guide explores this system and how to use it to your advantage.

Try the .

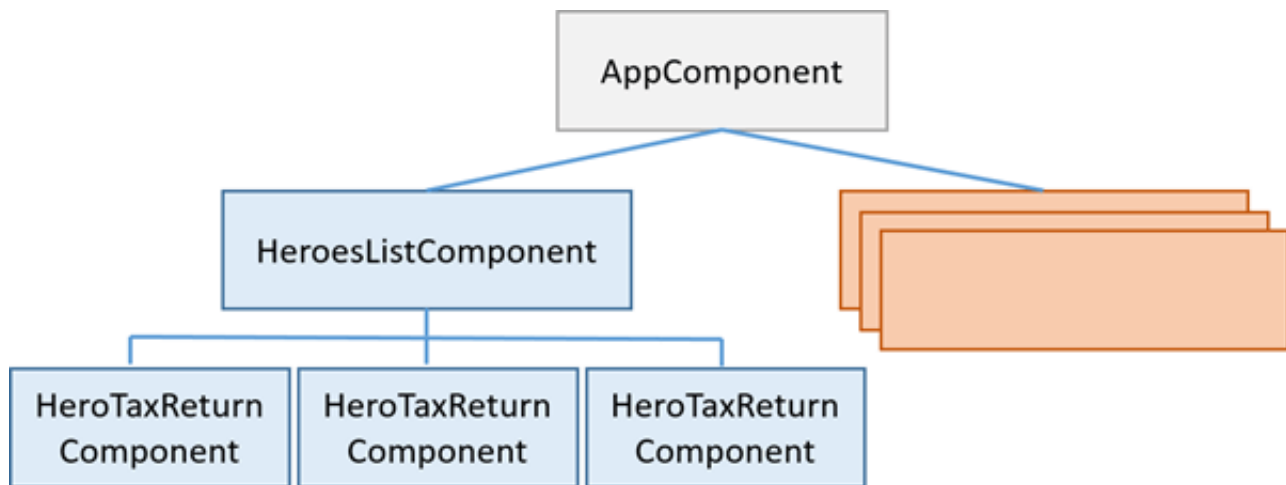
The injector tree

In the [Dependency Injection](#) guide, you learned how to configure a dependency injector and how to retrieve dependencies where you need them.

In fact, there is no such thing as ***the*** injector. An application may have multiple injectors. An Angular application is a tree of components. Each component instance has its own injector. The tree of components parallels the tree of injectors.

The component's injector may be a `_proxy_` for an ancestor injector higher in the component tree. That's an implementation detail that improves efficiency. You won't notice the difference and your mental model should be that every component has its own injector.

Consider this guide's variation on the Tour of Heroes application. At the top is the `AppComponent` which has some sub-components. One of them is the `HeroesListComponent`. The `HeroesListComponent` holds and manages multiple instances of the `HeroTaxReturnComponent`. The following diagram represents the state of the this guide's three-level component tree when there are three instances of `HeroTaxReturnComponent` open simultaneously.



Injector bubbling

When a component requests a dependency, Angular tries to satisfy that dependency with a provider registered in that component's own injector. If the component's injector lacks the provider, it passes the request up to its parent component's injector. If that injector can't satisfy the request, it passes it along to *its* parent injector. The requests keep bubbling up until Angular finds an injector that can handle the request or runs out of ancestor injectors. If it runs out of ancestors, Angular throws an error.

You can cap the bubbling. An intermediate component can declare that it is the "host" component. The hunt for providers will climb no higher than the injector for that host component. This is a topic for another day.

Re-providing a service at different levels

You can re-register a provider for a particular dependency token at multiple levels of the injector tree. You don't *have* to re-register providers. You shouldn't do so unless you have a good reason. But you *can*.

As the resolution logic works upwards, the first provider encountered wins. Thus, a provider in an intermediate injector intercepts a request for a service from something lower in the tree. It effectively "reconfigures" and "shadows" a provider at a higher level in the tree.

If you only specify providers at the top level (typically the root `AppModule`), the tree of injectors appears to be flat. All requests bubble up to the root `NgModule` injector that you configured with the `bootstrapModule` method.

Component injectors

The ability to configure one or more providers at different levels opens up interesting and useful possibilities.

Scenario: service isolation

Architectural reasons may lead you to restrict access to a service to the application domain where it belongs.

The guide sample includes a `VillainsListComponent` that displays a list of villains. It gets those villains from a `VillainsService`.

While you *could* provide `VillainsService` in the root `AppModule` (that's where you'll find the `HeroesService`), that would make the `VillainsService` available everywhere in the application, including the *Hero* workflows.

If you later modified the `VillainsService`, you could break something in a hero component somewhere. That's not supposed to happen but providing the service in the root `AppModule` creates that risk.

Instead, 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. It's still a singleton, but it's a singleton that exist solely in the *villain* domain.

Now you know that a hero component can't access it. You've reduced your exposure to error.

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.

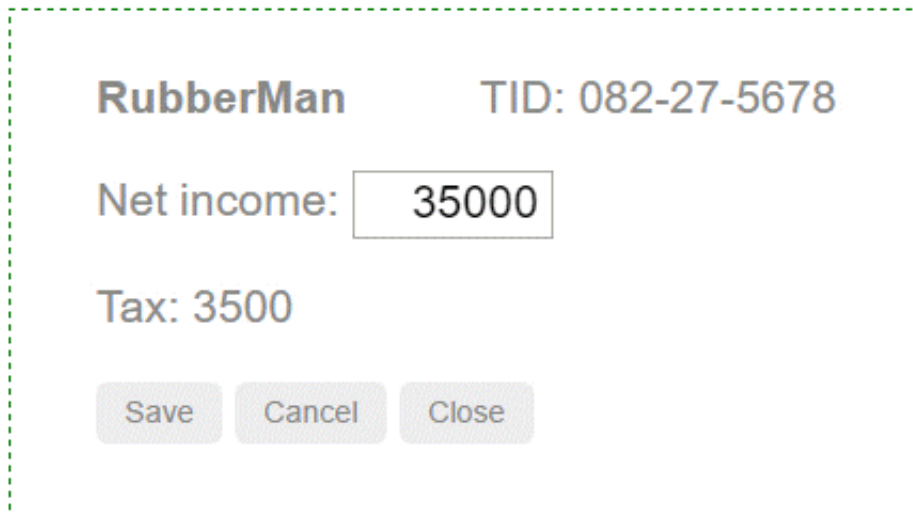
To open a hero's tax return, the preparer clicks on a hero name, which opens a component for editing that return. Each selected hero tax return opens in its own component and multiple returns can be open at the same time.

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.

Hero Tax Returns

- RubberMan
- Tornado



RubberMan TID: 082-27-5678

Net income:

Tax: 3500

One might suppose that the `HeroTaxReturnComponent` has logic to manage and restore changes. That would be a pretty easy task for a simple hero tax return. In the real world, with a rich tax return data model, the change management would be tricky. You might delegate that management to a helper service, as this example does.

Here is the `HeroTaxReturnService`. It 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 it.

The *tax-return-to-edit* arrives via 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.

There'd be big trouble if *this* service were an application-wide singleton. Every component would share the same service instance. Each component would overwrite the tax return that belonged to another hero. What a mess!

Look closely at the metadata for the `HeroTaxReturnComponent`. Notice the `providers` property.

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. No tax return overwriting. No mess.

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

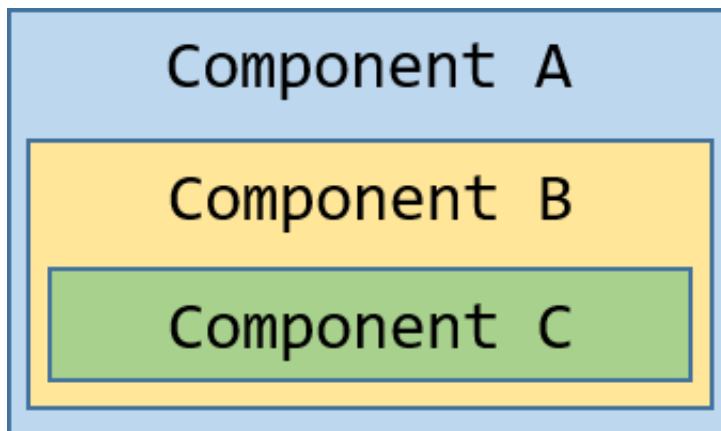
Another reason to re-provide a service is to substitute a *more specialized* implementation of that service, deeper in the component tree.

Consider again the Car example from the [Dependency Injection](#) guide. 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.

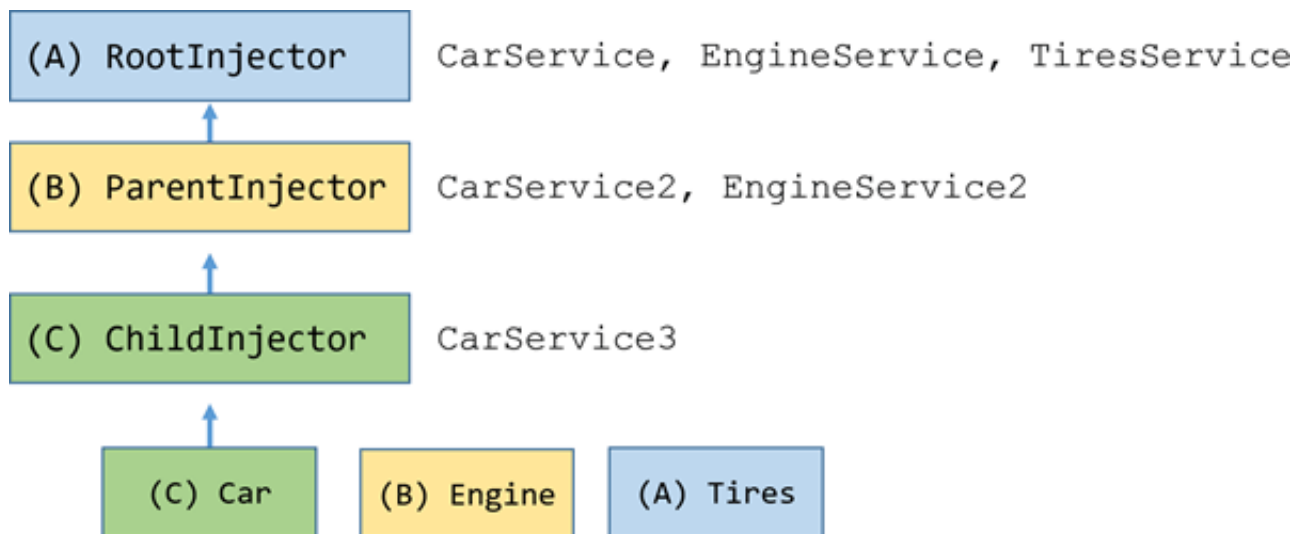
Then you create a child component (B) that defines its own, *specialized* providers for `CarService` and `EngineService` that have special capabilities suitable for whatever is going on in component (B).

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



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).



The code for this `_cars_` scenario is in the ``car.components.ts`` and ``car.services.ts`` files of the sample which you can review and download from the .