HIERARCHICAL INJECTORS

Developer Preview Only - some details may change

Angular's hierarchical dependency injection system supports nested injectors in parallel with the component tree.

We learned the basics of Angular Dependency injection in the Dependency Injection chapter.

Angular has an Hierarchical Dependency Injection system. There is actually a tree of injectors that parallel an application's component tree. We can re-configure the injectors at any level of that component tree with interesting and useful results.

In this chapter we explore these points and write some code.

Live Example.

The Injector Tree

In the Dependency Injection chapter we learned how to configure a dependency injector and how to retrieve dependencies where we need them.

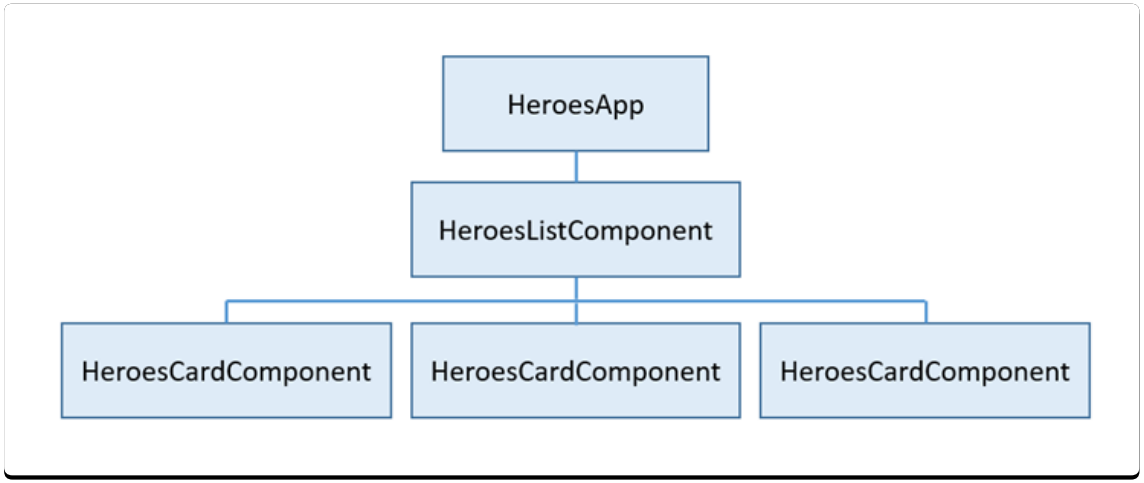
We oversimplified. In fact, there is no such thing as ***the*** injector! An application may have multiple injectors!

An Angular application is a tree of components. Each component instance has its own injector! The tree of components parallels the tree of injectors.

> Angular doesn't *literally* create a separate injector for each component. Every component doesn't need its own injector and it would be horribly inefficient to create masses of injectors for no good purpose.
>
> But it is true that every component ***has an injector*** (even if it shares that injector with another component) and there may be many different injector instances operating at different levels of the component tree.
>
> It is useful to pretend that every component has its own injector.

Consider a simple variation on the Tour of Heroes application consisting of three different components: `HeroesApp`, `HeroesListComponent` and `HeroesCardComponent`. The `HeroesApp` holds a single instance of `HeroesListComponent`. The new twist is that the `HeroesListComponent` may hold and manage multiple instances of the `HeroesCardComponent`.

The following diagram represents the state of the component tree when there are three instances of `HeroesCardComponent` open simultaneously.



Each component instance gets its own injector and an injector at one level is a child injector of the injector above it in the tree.

When a component at the bottom 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 component's injector. The requests keep bubbling up until we find an injector that can handle the request or run out of component ancestors. If we run out of ancestors, Angular throws an error.

> There's a third possibililty. An intermediate component can declare that it is the "host" component. The hunt for providers will climb no higher than the injector for this host component. We'll reserve discussion of this option for another day.

Such a proliferation of injectors makes little sense until we consider the possiblity that injectors at different levels can be configured with different providers. We don't *have* to re-configure providers at every level. But we *can*.

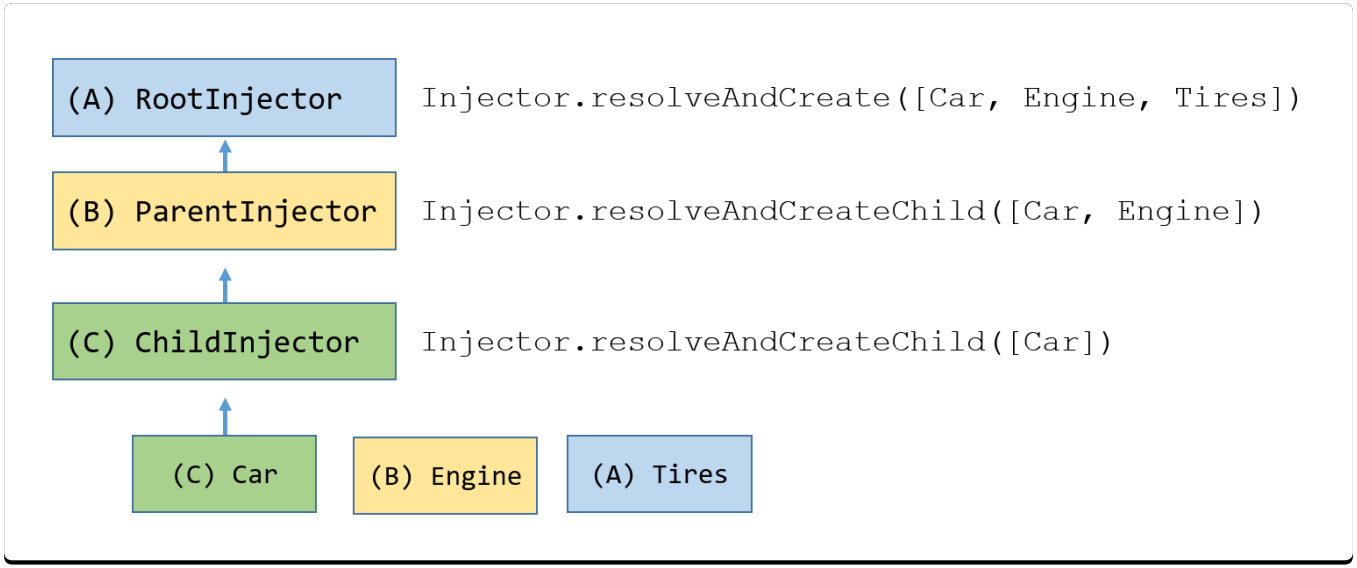
If we don't re-configure, the tree of injectors appears to be flat. All requests bubble up to the root injector that we configured with the `bootstrap` method.

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

Let's return to our Car example. Suppose configured the root injector (marked as A) with providers for `Car`, `Engine` and `Tires`. We create a child component (B) that defines its own providers for `Car` and `Engine` This child is the parent of another component (C) that defines its own provider for `Car`.

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

When we resolve an instance of `Car` at the deepest component (C), it's 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).



Component Injectors

In the previous section, we talked about injectors and how they are organized like a tree. Lookups follow the injector tree upwards until they found the requested thing to inject. But when do we actually want to provide providers on the root injector and when do we want to provide them on a child injector?

Consider you are building a component to show a list of super heroes that displays each super hero in a card with it's name and superpower. There should also be an edit button that opens up an editor to change the name and superpower of our hero.

One important aspect of the editing functionality is that we want to allow multiple heroes to be in edit mode at the same time and that one can always either commit or cancel the proposed changes.

Let's take a look at the `HeroesListComponent` which is the root component for this example.

app/heroes-list.component.ts

```
1. import {Component} from 'angular2/core';
2. import {EditItem} from './edit-item';
3. import {HeroesService} from './heroes.service';
4. import {HeroCardComponent} from './hero-card.component';
5. import {HeroEditorComponent} from './hero-editor.component';
6. import {Hero} from './hero';
7.
8. @Component({
9.   selector: 'heroes-list',
10.  template: `
11.    <div>
```

```
      <ul>
        <li *ngFor="#editItem of heroes">
          <hero-card
            [hidden]="editItem.editing"
            [hero]="editItem.item">
          </hero-card>
          <button
            [hidden]="editItem.editing"
            (click)="editItem.editing = true">
              edit
          </button>
          <hero-editor
            (saved)="onSaved(editItem, $event)"
            (canceled)="onCanceled(editItem)"
            [hidden]="!editItem.editing"
            [hero]="editItem.item">
          </hero-editor>
        </li>
      </ul>
    </div>`,
  directives: [HeroCardComponent, HeroEditorComponent]
})
export class HeroesListComponent {
  heroes: Array<EditItem<Hero>>;
  constructor(heroesService: HeroesService) {
    this.heroes = heroesService.getHeroes()
                               .map(item => new EditItem(item));
  }

  onSaved (editItem: EditItem<Hero>, updatedHero: Hero) {
```

```
    editItem.item = updatedHero;
    editItem.editing = false;
  }

  onCanceled (editItem: EditItem<Hero>) {
    editItem.editing = false;
  }
}
```

Notice that it imports the `HeroService` that we've used before so we can skip its declaration. The only difference is that we've used a more formal approach for our `Hero` model and defined it upfront as such.

app/hero.ts

```
export class Hero {
  name: string;
  power: string;
}
```

Our `HeroesListComponent` defines a template that creates a list of `HeroCardComponents` and `HeroEditorComponents`, each bound to an instance of hero that is returned from the `HeroService`. Ok, that's not entirely true. It actually binds to an `EditItem<Hero>` which is a simple generic datatype that can wrap any type and indicate if the item being wrapped is currently being edited or not.

app/edit-item.ts

```
export class EditItem<T> {
```

```
  editing: boolean
  constructor (public item: T) {}
}
```

But how is `HeroCardComponent` implemented? Let's take a look.

app/hero-card.component.ts

```
import {Component, Input} from 'angular2/core';
import {Hero} from './hero';

@Component({
  selector: 'hero-card',
  template: `
    <div>
      <span>Name:</span>
      <span>{{hero.name}}</span>
    </div>`
})
export class HeroCardComponent {
  @Input() hero: Hero;
}
```

The `HeroCardComponent` is basically a component that defines a template to render a hero. Nothing more.

Let's get to the interesting part and take a look at the `HeroEditorComponent`

app/hero-editor.component.ts

```ts
import {Component, Input, Output, EventEmitter} from 'angular2/core';
import {RestoreService} from './restore.service';
import {Hero} from './hero';

@Component({
  selector: 'hero-editor',
  providers: [RestoreService],
  template: `
    <div>
      <span>Name:</span>
      <input [(ngModel)]="hero.name"/>
      <div>
        <button (click)="onSaved()">save</button>
        <button (click)="onCanceled()">cancel</button>
      </div>
    </div>`
})

export class HeroEditorComponent {
  @Output() canceled = new EventEmitter();
  @Output() saved = new EventEmitter();

  constructor(private restoreService: RestoreService<Hero>) {}

  @Input()
  set hero (hero: Hero) {
    this.restoreService.setItem(hero);
  }

  get hero () {
```

```
31.     return this.restoreService.getItem();
32.   }
33.
34.   onSaved () {
35.     this.saved.next(this.restoreService.getItem());
36.   }
37.
38.   onCanceled () {
39.     this.hero = this.restoreService.restoreItem();
40.     this.canceled.next(this.hero);
41.   }
42. }
```

Now here it's getting interesting. The `HeroEditorComponent` defines a template with an input to change the name of the hero and a `cancel` and a `save` button. Remember that we said we want to have the flexibility to cancel our editing and restore the old value? This means we need to maintain two copies of our `Hero` that we want to edit. Thinking ahead this is a perfect use case to abstract it into it's own generic service since we have probably more cases like this in our app.

And this is where the `RestoreService` enters the stage.

app/estore.service.ts

```
1. export class RestoreService<T> {
2.   originalItem: T;
3.   currentItem: T;
4.
5.   setItem (item: T) {
6.     this.originalItem = item;
7.     this.currentItem = this.clone(item);
```

```
  }

  getItem () :T {
    return this.currentItem;
  }

  restoreItem () :T {
    this.currentItem = this.originalItem;
    return this.getItem();
  }

  clone (item: T) :T {
    // super poor clone implementation
    return JSON.parse(JSON.stringify(item));
  }
}
```

All this tiny service does is define an API to set a value of any type which can be altered, retrieved or set back to it's initial value. That's exactly what we need to implement the desired functionality.

Our `HeroEditComponent` uses this services under the hood for it's `hero` property. It intercepts the `get` and `set` method to delegate the actual work to our `RestoreService` which in turn makes sure that we won't work on the original item but on a copy instead.

At this point we may be scratching our heads asking what this has to do with component injectors? Look closely at the metadata for our `HeroEditComponent`. Notice the `providers` property.

```
  providers: [RestoreService],
```

This adds a `RestoreService` provider to the injector of the `HeroEditComponent`. Couldn't we simply alter our bootstrap call to this?

```
// Don't do this!
bootstrap(HeroesListComponent, [HeroesService, RestoreService])
```

Technically we could, but our component wouldn't quite behave the way it is supposed to. Remember that each injector treats the services that it provides as singletons. However, in order to be able to have multiple instances of `HeroEditComponent` edit multiple heroes at the same time we need to have multiple instances of the `RestoreService`. More specifically each instance of `HeroEditComponent` needs to be bound to it's own instance of the `RestoreService`.

By configuring a provider for the `RestoreService` on the `HeroEditComponent`, we get exactly one new instance of the `RestoreService` per `HeroEditComponent`.

Does that mean that services aren't singletons anymore in Angular 2? Yes and no. There can be only one instance of a service type in a particular injector. But we've learned that we can have multiple injectors operating at different levels of the application's component tree. Any of those injectors could have its own instance of the service.

If we defined a `RestoreService` provider only on the root component, we would have exactly one instance of that service and it would be shared across the entire application.

That's clearly not what we want in this scenario. We want each component to have its own instance of the `RestoreService`. Defining (or re-defining) a provider at the component level creates a new instance of the service for each new instance of that component. We've made the `RestoreService` a kind of "private" singleton for each

`HeroEditComponent`, scoped to that component instance and its child components.

Next Step

Upgrading from 1.x