ANGULAR 2020 EDITION

DEPENDENCY INJECTION

Recommended way

Register

Creates a 🌴 tree shakable, singleton service in the root injector.

```
@Injectable({
  providedIn: "root"
})
export class MyService {}
```

Use

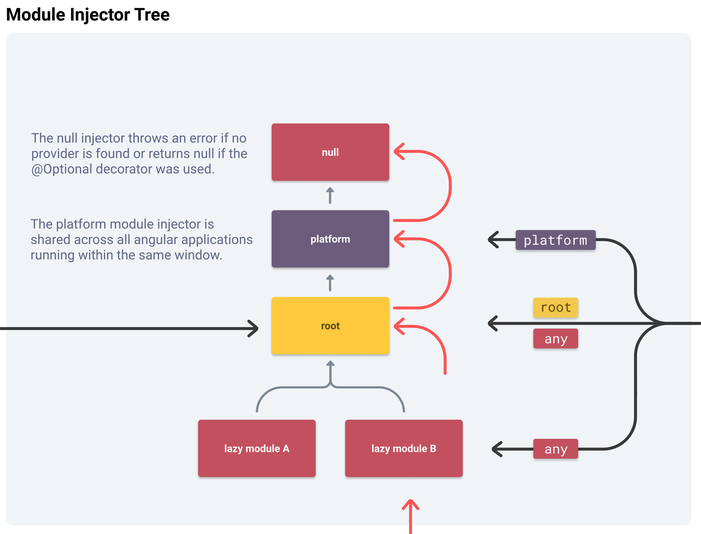
```
@Component({ ... })
export class MyComponent {
    constructor(private myService: MyService)
}
```

Resolution

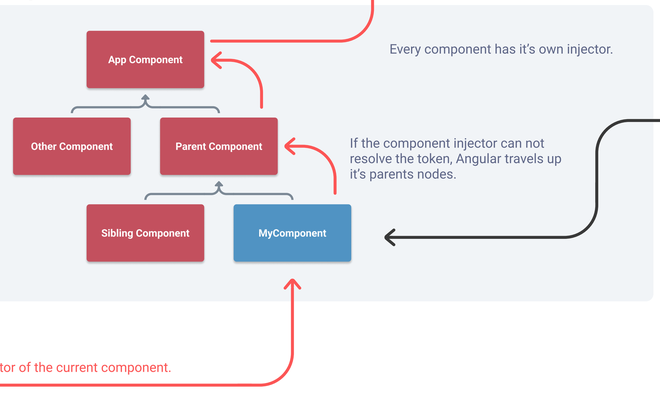
MyService class is used a a token. Like calling Injector.get(MyService) directly.

Hierarchical injectors

Module Injector Tree



Node Injector Tree



Other ways to register

🌴 Tree Shakable Injection Tokens

```
const MyServiceToken = new InjectionToken<MyService>("My service", {
    providedIn: "root" | "any" | "platform"
    factory: () => new MyService(inject(MyDep))
});
```

🌴 Tree Shakable Services with a factory

```
@Injectable({
  providedIn: "root" | "any" | "platform"
  useFactory: () => new MyService()
})
export class MyService {}
```

NgModule Providers

```
@NgModule({
    ...
    providers: [MyService]
})
export class MyModule {}
```

Component Level Providers

Use this to create a singleton service for a component and its child components.

```
@Component({
    providers: [MyService],
    viewProviders: [MyService],
})
export class MyComponent {}
```

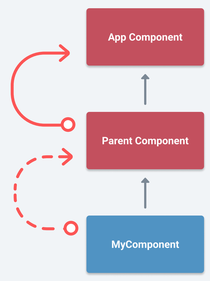
Providers makes the service available to it's component, all child components including projected components through ng-content.

ViewProviders limits the provider to it's component and child components. All child components within ng-content don't see the provider.

Resolution Modifiers

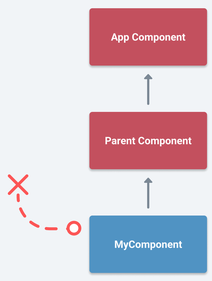
@SkipSelf()

Skips itself and starts with the parent component injector



@Self()

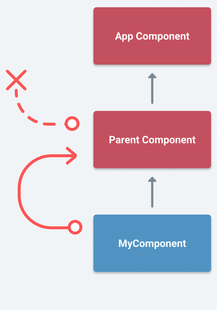
Only looks on the current component injector



@Host()

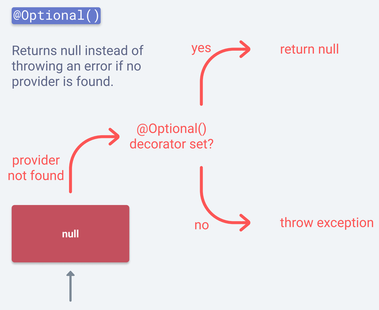
Lets you designate a component as the last stop in the injector tree when searching for providers.

Even if there is a service instance further up the tree, Angular won't continue looking



@Optional()

Returns null instead of throwing an error if no provider is found.




```
// Usage
export class MyComponent {
    constructor(@Host() myService: MyService)
}
```

```
// Modifiers can be combined
export class MyComponent {
    constructor(@Self() @Optional() myService: MyService)
}
```

Providers Syntax

```
providers: [MyService]
```

is a shorthand for

```
providers: [{ provide: MyService, useClass: MyService }]
```

Instead of a class token, we could also use:

String Token

Use string token for simple values like dates, numbers and strings, or shapeless objects like arrays and functions.

```
[{ provide: "Base_Url", ... }]
```

Injection Token The better "String Token"

The use case of an injection token is the same as for string tokens. The main advantage of injection tokens is that the prevent name clashes. Use injection tokens over string token whenever possible.

```
const BaseUrl = new InjectionToken<string>("Base_Url");
[{ provide: BaseUrl, ... }]
```

Use @Inject to retrieve **String Token** **or** **Injection Token**

```
constructor(@Inject("Base_URL") private baseUrl: string)
```

```
constructor(@Inject(BaseURL) private baseUrl: string)
```

Instead of usClass, we could also use:

```
[{ ..., useValue: "http://localhost"}]
```

```
[{ ..., useFactory: () => "http://localhost"}]
```

```
[{ ..., useExisting: ExistingService }]
```

UseExisting is like an alias to an existing service.

multi: true

Used to register multiple services or values with one token. Returns an array with all services or values. Without the multi: true, the last provider would just override the existing one. In this example "colors" would return "black".

```
[
  { provide: "colors", useValue: "white", multi: true },
  { provide: "colors", useValue: "black", multi: true }
]

constructor(@Inject("colors") colors: string[]) {
  console.log(colors); // Logs: ["white", "black"]
}
```


Chris Kohler
christiankohler.net
@kohlerchristian

Version 1.0 © 2020 Chris Kohler