ANGULAR

2020 EDITION INCLUDING THE NEW ANGULAR 9 FEATURES

DEPENDENCY INJECTION

What is an injector?

An injector is basically a key/value map. Here is a code example showing how an injector is created and used under the hood:

```
// Register
const injector = Injector.create([
  { provide: “color”, useValue: “blue” }
]);

// Retrieve
injector.get(“color”); // returns “blue”
```

Recommended way

This is what you need to know to get started with Angular DI. How to register a service and then how to inject it into a component. This is the most basic and most common use case.

Register

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

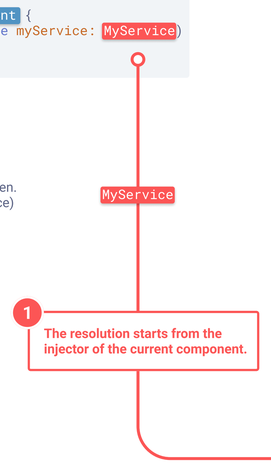
```
@Injectable({
  providedIn: “root”
})
export class MyService {}
```

Inject into a Component

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

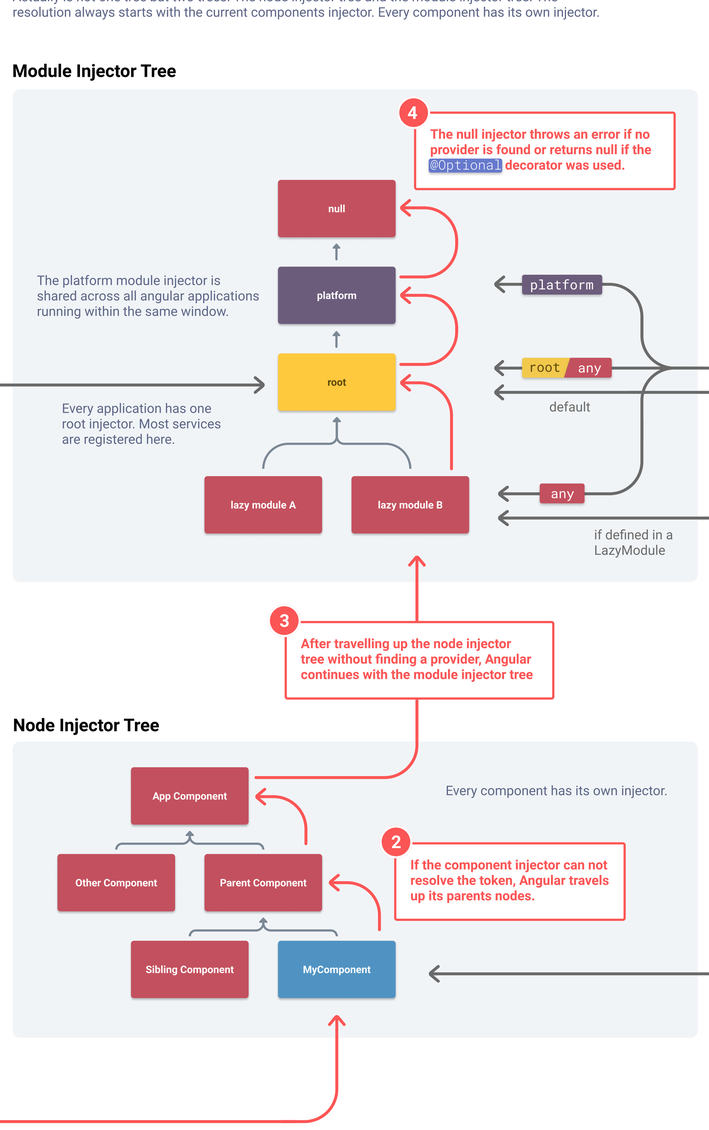
Resolution

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



Hierarchical injectors (under the hood)

This is how the injector tree looks under the hood. It’s a hierachical tree of injectors. Every injector has a parent with the null injector as last injector.
Actually is not one tree but two trees. The node injector tree and the module injector tree. The resolution always starts with the current components injector. Every component has its own injector.



Other ways to register

There are some other ways how to register a service. The most common one is the provider array in NgModule which can be mostly replaced by tree-shakable alternatives.

🌴 Tree-shakable Injection Tokens

Use them to create tree-shakable tokens for non class types like strings. Read more about InjectionTokens in the “Provider Syntax” section.

```
const BaseURL = new InjectionToken<string>(“BaseUrl”, {
    providedIn: “root” | “any” | “platform”
    factory: () => “localhost:4200”
});
```

🌴 Tree-shakable Services with a Factory

Add a factory to customize how the service is created.

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

NgModule Providers

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

NgModule providers should be avoided since they are not 🌴 tree-shakable.

NgModule providers are usually registered in the root injector. Only exception are LazyModules. In this case the providers are registered in the lazy module injector and is not available to components outside of the lazy module.

🌴 What is Tree-shaking?

Tree-shaking eliminates dead code by removing unused code. Angular removes tree-shakable providers from the final bundle when the application doesn’t use those services. This can reduce the bundle size.

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 {}
```

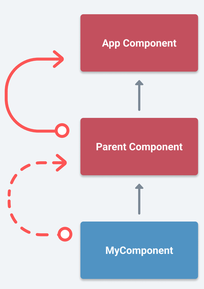
Providers makes the service available to its component, all child components including projected components through ng-content.

ViewProviders limits the provider to its component and child components. All child components within ng-content don’t see the provider.

Resolution Modifiers

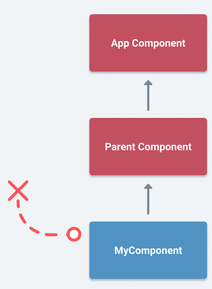
@SkipSelf()

Skips itself and starts with the parent component injector



@Self()

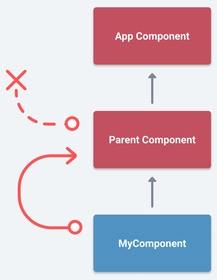
Only looks on the current component injector



@Host()

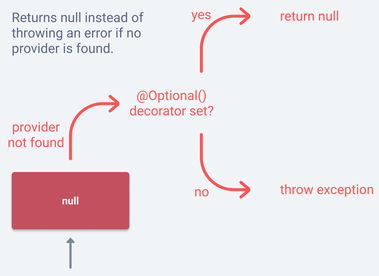
Lets you designate a component as the last stop in the injector tree when searching for providers.

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 InjectionToken is the same as for string tokens. The main advantage of InjectionTokens is that the prevent name clashes. Use InjectionTokens over string tokens 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 useClass, 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

Subscribe: @kohlerchristian