DEPENDENCY INJECTION

Ori Calvo, 2017

oric@trainologic.com

https://trainologic.com

The Pattern

- The client delegates the responsibility of providing its dependencies to external code (the injector)
- The use of the new keyword or specific factory function is prohibited
- Creates a more testable & “composable” code
- Usually harder to debug since there is a lot of “magic” behind the scene

Angular POV

- Application consists of components and services
- A component should ask a reference to a service (A.K.A dependency)
- Angular's injector is responsible for resolving all dependencies upon creation of the component
- Unlike Angular1 there are many injectors at runtime
  - A.K.A hierarchical injector

Basic Sample


```
@Injectable()
export class ContactService {
  getAll(): Promise<Contact[]> {
    return Promise.resolve([
      {id: 1, name: "Ori"},
      {id: 2, name: "Roni"},
    ]);
  }
}
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [...],
  providers: [ContactService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
@Component({...})
export class AppComponent implements OnInit {
  contacts: Contact[];

  constructor(private contactService: ContactService) {
  }

  async ngOnInit() {
    this.contacts = await this.contactService.getAll();
  }
}
```

Dependency Token

Dependency Token (again)

How does it work ?

- @angular/core offers a class named ReflectiveInjector
- It is a factory class which knows how to create an injector instance from a list of providers
- The injector knows how to instantiate a “service” based on its dependencies
- Services are singletons in the context of a single injector

Ingredients

- Token – A unique value that can be resolved into a service
  - Must be of type InjectionToken or Type
  - The usage of a string is now deprecated
- Provider – Maps a token to a list of dependencies
- Injector – Holds a set of providers and is responsible for resolving dependencies
- Dependency – The “thing” that is being injected

ReflectiveInjector


```
class MyClass1 {
    dump() {
        console.log("xxx");
    }
}
```

Very simple scenario. There are no dependencies

```
const injector = ReflectiveInjector.resolveAndCreate([
    {provide: MyClass1, useClass: MyClass1},
])
```

Provider

Token

Resolving a token

```
const obj = injector.get(MyClass1);
```

Don't get panic

- We usually don't create injectors manually
- Angular creates several injectors during application bootstrapping
- The two most important
  - PlatformRef – All providers related to the platform
  - NgModuleRef – All providers defined by the application and sub modules

Class Provider

- Instead of writing

```
const injector = ReflectiveInjector.resolveAndCreate([
    {provide: MyClass1, useClass: MyClass1},
])
```

- We can just use the class name

```
const injector = ReflectiveInjector.resolveAndCreate([
    MyClass1,
])
```

Value Provider


```
class A {
   dump() {
      console.log("A");
   }
}

export const a = new A();

const injector = ReflectiveInjector.resolveAndCreate([
   {provide: A, useValue: a},
]);

const obj = injector.get(A);
obj.dump();
```

Any 2rd party global object can be made injectable

Factory Provider


```
function createA(version: number) {
   console.log(version);

   return new A();
}

const VERSION = "VERSION";

const injector = ReflectiveInjector.resolveAndCreate([
   {provide: VERSION, useValue: 123},
   {provide: A, useFactory: createA, deps: [VERSION]},
]);

const obj = injector.get(A);
```

Must specify dependencies manually

useClass Dependencies

- JavaScript has no real Reflection capabilities and therefore below code fails to run

```
class MyClass1 {
}

class MyClass2 {
    constructor(obj1: MyClass1) {
    }
}

const injector =
ReflectiveInjector.resolveAndCreate([
    MyClass1,
    MyClass2
]);

const obj = injector.get(MyClass2);
```

Angular knows that the ctor has 1 parameter but it cant tell the parameter type

Error

Dependencies Metadata


```
class MyClass1 {
   dump() {
      console.log("xxx");
   }
}

class MyClass2 {
   static parameters = [MyClass1];

   constructor(obj1: MyClass1) {
   }
}

const injector = ReflectiveInjector.resolveAndCreate([
   MyClass1,
   MyClass2
]);

const obj = injector.get(MyClass2);
```

Dependencies are specified manually

Typescript Metadata

- Typescript is capable of generating “parameters” metadata automatically
- The metadata is generated only if decorating the class with a decorator
- The metadata is defined using the ECMA6 Reflection API
- Use the reflect-metadata shim
- Once Angular detects Reflect API it will use the metadata created by Typescript

Generated Metadata

The generated JavaScript

```
var MyClass2 = (function () {
    function MyClass2(obj1) {
    }
    return MyClass2;
}());

MyClass2 = __decorate([
    Blabla(),
    __metadata("design:paramtypes", [MyClass1])
], MyClass2);
```

Metadata

```
function Blabla() {
    return function(ctor: any) {
        return ctor;
    }
}

class MyClass1 {
    dump() {
        console.log("xxx");
    }
}

@Blabla()
class MyClass2 {
    constructor(obj1: MyClass1) {
    }
}
```

Class decorator definition

Use the decorator

@Injectable Decorator

- A convenient decorator offered by Angular
- Like any other decorator it enforces Typescript to emit constructor metadata
- The name might be confusing
  - Implies the class's dependencies can be resolved automatically
  - Does not implies that you can inject the class into another class

Duplicates Token

- You may specify the same token twice
- Last definition wins !!!
- It means you can override built-in Angular services

```
const injector = ReflectiveInjector.resolveAndCreate([
   {provide: "myService", useClass: MyClass1},
   {provide: "myService", useClass: MyClass2},
]);

const obj = injector.get("myService");

console.log(obj instanceof MyClass2);
```

Child Injector

- Once creating an injector you cannot add new providers to it
  - By design → Allows for better optimization
- However, you can create a new child injector which "extends" it

```
const injector = ReflectiveInjector.resolveAndCreate([A]);
const childInjector = injector.resolveAndCreateChild([B]);

const a1 = injector.get(A);
const a2 = childInjector.get(A);
console.log(a1 == a2);

const b = childInjector.get(B);
```

True

Components & Injectors

- Each component has its own injector
- Each component can define new providers using
  - providers
  - viewProviders
- A component "enjoy" all providers defined by itself and its parent (up until the root component)
- So why do we need viewProviders ?

ng-content

- Consider below dialog component

Dialog template

```
<div class="title">
  Dialog
</div>

<div class="content">
  <ng-content></ng-content>
</div>
```

```
<button (click)="about()">About</button>

<my-dialog #dialog>
  <my-clock></my-clock>

  <button>Close</button>
</my-dialog>
```

Using dialog inside other component

- Should clock get access to providers defined by dialog ?
- Dialog may use viewProviders to publish services to its children only but not to clock

Overriding

- A child injector may specify the same provider again
- In that case it will create a new instance object for the "redefined" provider

```
const injector = ReflectiveInjector.resolveAndCreate([MyClass1]);
const childInjector = injector.resolveAndCreateChild([MyClass1]);

const obj1 = injector.get(MyClass1);
const obj2 = childInjector.get(MyClass1);

console.log(obj1 == obj2);
```

Aliasing

- An existing provider may be “reused” and be configured as a provider for another token

```
const CORE_PROVIDERS = [
   A
];

const injector = ReflectiveInjector.resolveAndCreate([
   CORE_PROVIDERS,
   B,
   {provide: A, useExisting: B},
]);

const a = injector.get(A);
const b = injector.get(B);
console.log(a == b);
```

We assume CORE_PROVIDERS is a 3rd party definition that cannot be modified

True

InjectionToken

- Below code fails to compile

```
interface IMyService {
   doSomething(): any;
}

class MyService {
   doSomething(): any {
      console.log("MyService");
   }
}

const injector = ReflectiveInjector.resolveAndCreate([
   {provide: IMyService, useClass: MyService},
]);
```

TS2693 error: IMyService only refers to a type, but is being used as a value here

InjectionToken

- Creates an object wrapper around the interface

```
interface IMyService {
   doSomething(): any;
}

const IMY_SERVICE = new InjectionToken<IMyService>("xxx");

class MyService {
   doSomething(): any {
      console.log("MyService");
   }
}

const injector = ReflectiveInjector.resolveAndCreate([
   {provide: IMY_SERVICE, useClass: MyService},
]);

const obj = injector.get(IMY_SERVICE);
obj.doSomething();
```

You may use any interface. However, is should be the one that resembles the service API

Optional Dependency


```
class Config {
}

@Injectable()
class MyService {
   constructor(@Optional() config: Config) {
      console.log(!!config);
   }
}

const injector = ReflectiveInjector.resolveAndCreate([
   //Config,
   MyService,
]);

const obj = injector.get(MyService);
```

Dependency marked as Optional may be unresolved and remain empty

False

Order does matter !

- Below code compiles successfully but fails to run ☹

Generated JavaScript

```
var MyService = (function () {
    function MyService(config) {
        console.log(!!config);
    }
    return MyService;
}());
MyService = __decorate([
    core_1.Injectable(),
    __metadata("design:paramtypes", [Config])
], MyService);
var Config = (function () {
    function Config() {
    }
    return Config;
}());
```

Config is undefined

```
@Injectable()
class MyService {
    constructor(config: Config) {
    }
}

class Config {
}

const injector = ReflectiveInjector.resolveAndCreate([
    Config,
    MyService,
]);

const obj = injector.get(MyService);
```

The error is "cannot resolve all parameters for MyService"

forwardRef

- Allows us to use a dependency token that was not initialized yet
- Must be initialized before resolving providers

```
@Injectable()
class MyService {
   constructor(@Inject(forwardRef(() => Config)) config: Config) {
      console.log(!!config);
   }
}

class Config {
}

const injector = ReflectiveInjector.resolveAndCreate([
   Config,
   MyService,
]);

const obj = injector.get(MyService);
```

Multi Provider

- Register multiple providers with the same token
- When resolved, an array of services is returned

```
const injector = ReflectiveInjector.resolveAndCreate([
    {provide: MyService, useClass: MyService, multi: true},
    {provide: MyService, useClass: MyService, multi: true},
]);

const arr = injector.get(MyService);
console.log(arr);
```

Array of MyService objects

Multi Provider - Why

- Extendibility mechanism
- Angular defines that token + basic implementation
- You may extend with your own providers
- Angular uses them all. For example,
  - APP_INITIALIZER
- You cannot mix regular and multi providers

Cyclic Dependency

- Two different providers might be dependent on each other

```
@Injectable()
class MyService1 {
    constructor(@Inject(forwardRef(()=>MyService2)) private service2: MyService2) {
    }
}

@Injectable()
class MyService2 {
    constructor(private service1: MyService1) {
    }
}

const injector = ReflectiveInjector.resolveAndCreate([
    MyService1,
    MyService2,
]);

const obj = injector.get(MyService1);
```

Cyclic dependency Angular does not support that !!!

Without forwarfRef MyService2 token is undefined

Resolving cyclic dependencies

- The injector instance is itself an injectable service
- You can use it as a service locator
  - Some consider this pattern a bad practice
  - You may explore the injector's parent directly
- Break the cycle by deleting a dependency from the constructor and move it to a property/field

Resolving cyclic dependencies


```
@Injectable()
class MyService1 {
   _service2: MyService2;

   constructor(private injector: Injector) {
   }

   get service2() {
      if(!this._service2) {
         this._service2 = this.injector.get(MyService2);
      }

      return this._service2;
   }
}

@Injectable()
class MyService2 {
   constructor(private service1: MyService1) {
   }
}
```

Caching the dependency

Resolve it on demand

@Self

- Prohibit using the parent injector when resolving dependencies

```
@Injectable()
class MyService1 {
}

@Injectable()
class MyService2 {
   constructor(@Self() private service1: MyService1) {
   }
}

const injector = ReflectiveInjector.resolveAndCreate([
   MyService1,
]);

const child = injector.resolveAndCreateChild([MyService2]);

const obj = child.get(MyService2);
```

Error is thrown

Can add @Optional to get null dependency instead of error

@SkipSelf

- Always resolve dependency using parent injector

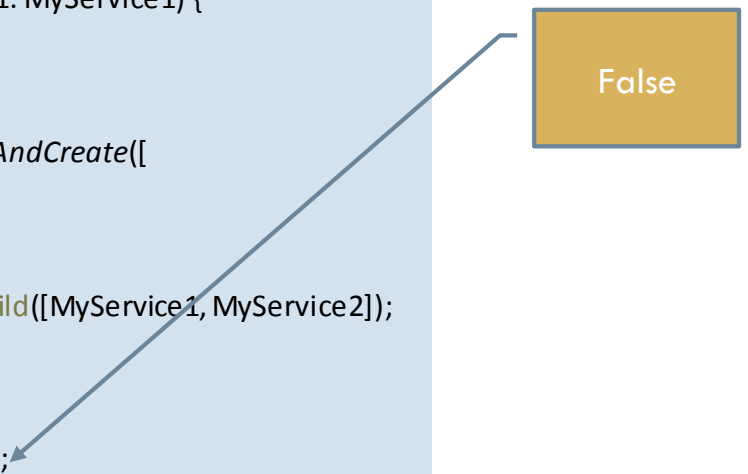
```
@Injectable()
class MyService1 {
}

@Injectable()
class MyService2 {
   constructor(@SkipSelf() private service1: MyService1) {
   }
}

const injector = ReflectiveInjector.resolveAndCreate([
   MyService1,
]);

const child = injector.resolveAndCreateChild([MyService1, MyService2]);

const service2 = child.get(MyService2);
const service1 = child.get(MyService1);
console.log(service1 == service2.service1);
```



Modules & Injectors

- For each module Angular generates an injector
- The injector contains a flat list of all providers from "sub" modules and the current module

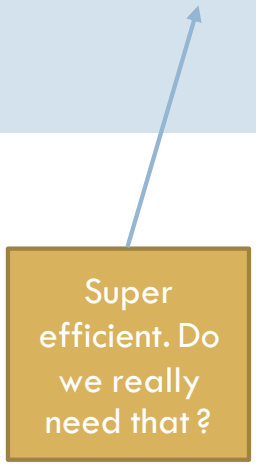
```
@NgModule({
    imports: [
        BrowserModule,
        CommonModule,
        Module1Module,
    ],
    providers: [
        {provide: CommonService, useClass: MyService},
    ],
    bootstrap: [
        AppComponent,
    ],
    declarations: [
        AppComponent,
    ],
})
export class AppModule {}
```

providers win over imports

Module Injector

- The module injector is generated by Angular at runtime/AOT according to @NgModule metadata

```
AppModuleInjector.prototype.getInternal = function(token,notFoundResult) {
   var self = this;
   if ((token === jit_CommonService41)) { return self._CommonService_9; }
   if ((token === jit_MainService35)) { return self._MainService_10; }
   …
   return notFoundResult;
};
```



Duplicated Service Instances

- Angular flattens the providers list
- Last provider wins
- Therefore, no duplicated service instances at run time
- But what about lazy loading a module
  - In that case a new injector is created
  - If a provider is redefined → new service instance might be created ☹

Lazy load a Module

Download the module from the server

Compile the module and get a factory

```
const {LazyModule} = await SystemJS.import("app/lazy/lazy.module.js");

const {ngModuleFactory, componentFactories} =
this.compiler.compileModuleAndAllComponentsSync(LazyModule);

const moduleInjector = ngModuleFactory.create(this.injector);

const componentFactory = componentFactories[0];

this.marker.createComponent(componentFactory, 0, <any>moduleInjector);
```

Create the component with the new injector

We must create a new injector, else, the component will be service-less

The duplication is implicit

Importing again CommonModule means a redefinition of all its providers

```
@NgModule({
  imports: [
    CommonModule,
  ],
  providers: [
    LazyService
  ],
  declarations: [
    LazyComponent
  ]
})
export class LazyModule {
  constructor() {
    console.log("LazyModule");
  }
}
```

This is a new provider downloaded with the module

forRoot & forChild

Specify components for both forRoot & forChild

forRoot specify providers

forChild does not specify providers

```
@NgModule({
  declarations: [CommonComponent],
  exports: [CommonComponent]
})
export class CommonModule {
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: CommonModule,
      providers: [
        CommonService,
      ]
    };
  }

  static forChild(): ModuleWithProviders {
    return {
      ngModule: CommonModule,
    };
  }
}
```

Summary

- Angular offers its own DI mechanism
- Quite “standard”
- However, support the notion of child injector
- Be prepared to handle cyclic dependency errors