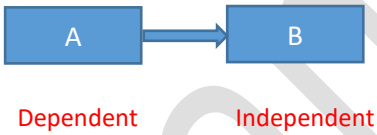
Agenda

- Understanding Dependency Injection
- Understanding DI in Angular Framework
- ReflectiveInjector
- Exploring Provider
- Types of Tokens
- Types of Dependencies
- Configuring DI using Providers
- Implementing DI in Angular

Understanding Dependency Injection

Dependency Injection (DI) is the software design pattern in which a class receives its dependencies from external sources rather than creating them itself.

DI is used in an application when a **module A** needs to access the data from **module B** then **module B** is named as **Independent** Module and **module A is dependent.**



To make this **tightly coupled access to loosely coupled** we use **DI pattern**.

DI in angular has mainly three things

1. **Injector:** Injector object is used to expose API's and create instances of dependencies.
2. **Provider:** Provider tells the injector how to create instance for a dependency. A provider takes **token** and **maps** to a factory to create instances.
3. **Dependency:** It's a type of which an object should be created.

Let's look at a simple example: **Student class without DI**

File: Student.ts

```
export class Address
{ }
export class Subjects
{ }
class Student {
   constructor() {
      this.address = new Address();
      this.subjects = new Subjects();
      this.subjects.push("Sub1");
      this.subjects.push("Sub2");
```

```
  }
  learn(subject) {
    this.subjects.push(subject);
  }
}
```

Drawbacks of this approach:

1. **Inflexible**: Student is directly dependent on Address and Subjects classes and is thus inflexible difficult to test.
   Eg: What if the **Address** class evolves and its constructor requires a parameter? Our **Student** is broken and stays broken until we rewrite it along the lines of **this.address** = new Address(**theNewParameter**).
2. **Hard to test**: The original code and original data should never get impact while testing, this approach will impact the original data while testing. Student creates Real Address and Real Subjects instead of Mock version of these.
3. Main class **cannot share objects** with other classes as it is creating dependent objects itself in the constructor.
4. The dependency is hidden. We have no control over the stud's hidden dependencies. When we can't control the dependencies, a class becomes difficult to test.

So, to make this code easy to re-use easy to test and easy to maintain let's change the code as follows.
We would like to follow the approach by injecting dependencies in the **constructor.**
In this approach the **constructor** will expect all the dependencies needed.

Student class with Address and Subjects injected:

```
export class Address
{ }
export class Subjects
{ }
export class Student {
  constructor(public address: Address, private subjects: Subjects) {
  }
  learn(subject) {
    this.subjects.push(subject);
  }
}
```

Now here we have decoupled all the dependencies from our **Student** class. To create instance for Student we need to create as follows.

app.component.ts

```
import { Component, ChangeDetectorRef  } from '@angular/core';
import { Address, Subjects, Student } from "./Student"
@Component({
   selector: 'my-app',
   template: `
      <div>{{stud.drive(10)}}</div>
`})
export class AppComponent {
   stud = new Student(new Address(), new Subjects());
}
```

OR

```
class MockAddress extends Address { }
class MockSubjects  extends Subjects  { name = 'TestSubject'; }
let stud = new Student(new MockAddress(), new MockStudents());
```

Note: The definition of the address and tire dependencies are decoupled from the Student class itself. We can pass in any kind of address or subjects we like, as long as they conform to the general API requirements of an address or subjects.

Drawbacks of this approach:

The *consumer* of Student has the problem, **they need to explicitly create dependent objects.** The Student class shed its problems at the consumer's expense.

Another Option: Using Factory Approach:

We need something that takes care of assembling all parts of Student without needing to explicitly create each dependent object.

carfactory.ts

```
import { Address, Subjects, Student } from './stud';
export class StudentFactory {
   createStudent() {
      let stud = new Student(this.createAddress(), this.createSubjects());
      stud.description = 'Factory';
      return stud;
   }
   createAddress() {
      return new Address();
   }
   createSubjects() {
      return new Subjects();
   }
```

```
}
```

Maintaining such a class will become complicated as the application grows. This factory is going to become a huge spider web of interdependent factory methods!.

It would be nice if we could simply list the things we want to build without having to define which dependency gets injected into what?

Anyways, **we just learned what dependency injection is**.

This is where the dependency injection framework comes into play. The framework has something called **an *injector***. We register some classes with this injector, and it figures out how to create them.

Understanding DI in Angular Framework

Using **Injector**, we are going to achieve angular's **Dependency Injection** by using **ReflectiveInjector** from **'@angular/core'** library.

Injector resolve a token into a dependency.

- Injector takes **input token** and returns **dependency or list of dependencies**.
- Injector does not returns the class but an instance using **new keyword.**

Let's see how angular can use DI

```
import { Component, ReflectiveInjector } from '@angular/core';
import { Student, Address, Subjects } from './Student';

export class AppComponent {
  constructor() {
    var injector = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]); //Student, Address, Subjects are Tokens
    var stud = injector.get(Student);
  }
}
```

resolveAndCreate: A factory function that creates an injector and takes a list of providers.

Now the problem is how the injector knows which dependencies needs to be in order for instantiating the stud. To do this we import **Inject** from the angular framework and apply it as decorator **@Inject(Token)** to constructor parameters.

```
import {Inject,Component} from '@angular/core';
export class AppComponent {
constructor(@Inject(Address) address, @Inject(Subjects) subjects ) {
//your code
}
```

```
}
```

Dependency Caching

- Multiple calls to the **same injector** for same token **will return the same instance.**
- Multiple calls to **different injector** for same token **returns different instances**.

```
var injector = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]);
var stud = injector.get(Student);
var stud1 = injector.get(Student);
//Here stud === stud1   → true
```

```
var injector = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]);
var injector1 = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]);
var stud = injector.get(Student);
var stud1 = injector1.get(Student);
//Here stud === stud1   → false
```

Child Injector

Injectors can have one or more child injectors, for creating child injector we need to use **resolveAndCreateChild( ).** This works similar to parent injector but with few additions.

- Parent injector and child injector instances are **different** if both are resolving for the **same provider**.
- Parent injector and child injector instances are **same** if child injector is **not configured** for any **provider**.

```
var injector = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]);
var childInjector = injector.resolveAndCreateChild([Student, Address, Subjects]);
var stud = injector.get(Student)
var stud1 = childInjector.get(Student);
// stud === stud1 ----→ false
```

```
var injector = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]);
var childInjector = injector.resolveAndCreateChild([]);
var stud = injector.get(Student)
var stud1 = childInjector.get(Student);
// stud === stud1 → true
```

Exploring Provider

Provider **is an object** which describes a token and configuration for how to create the associated dependency

```
var injector = ReflectiveInjector.resolveAndCreate([
```

```
    Student,
    Address,
    Subjects
]);
```

This injector syntax is actually a shorthand syntax of

```
var injector = ReflectiveInjector.resolveAndCreate([
    { provide:Student, useClass:Student },
    { provide:Address, useClass:Address },
    { provide:Subjects, useClass:Subjects }
]);
```

Properties of Provider:

- **Provide:** This property is the **token**, it can be either a **string** or **type** or **instance.**
- **useClass:** This property is the **dependency** used for **configuring classes** in the instance.

Switching Dependencies

- **Token** and **dependencies** can be of different types, in such case when token of one service is requested it returns instance of other type.

```
var injector = ReflectiveInjector.resolveAndCreate([
    { provide:Student, useClass:SeniorStudent }
]);
```

Here when we request with token **"Student"** it returns instance of **SeniorStudent.**

Types of Tokens

There are there different types of tokens in angular framework they are

1. String Tokens
2. Type Tokens
3. Injection Tokens

1. **String tokens:** We can use **strings** as tokens as follows

```
var injector = ReflectiveInjector.resolveAndCreate([
    {provide:"Student", useClass:Student}
]);
```

2. **Type tokens:** Token with any of the type specifically we will use **Class name** as type.

```
var injector = ReflectiveInjector.resolveAndCreate([
    { provide:Student useClass:Student }
]);
```

3. **Injection tokens:** In this type of token we can inject the token via an instance of **InjectionToken**.

```
var c = new InjectionToken<string>("Student");
var injector = ReflectiveInjector.resolveAndCreate([
    { provide:c useClass:Student }
]);
```

Types of Dependencies

There are 4 types of dependencies providers in angular they are:

1. **useClass:** Provider mapping token to a class
2. **useExisting:** Two tokens mapping to same provider
3. **useValue:** Provider mapping token to a value
4. **useFactory:** Provider token mapping to call back function

1. **UseClass:** When our dependency is of type class they we choose **useClass,** when the user request the token injector resolves and returns the instance of type **class.**

```
var injector = ReflectiveInjector.resolveAndCreate([ Student ]);
```

The above syntax is a shorthand syntax of provider configuration using **useClass**. Here **Student** is a **class.**

```
var injector = ReflectiveInjector.resolveAndCreate([ { provide:Student, useClass:Student } ]);
```

2. **useExisting:** We can map multiple tokens to the same dependency via **aliases.**

```
class Student {}
class Address {}
class Subjects {}
var injector = ReflectiveInjector.resolveAndCreate([
                    { provide:Student, useClass:Student },
                    { provide:Address, useExisting:Student },
                    { provide:Subjects, useExisting:Student }
                ]);
```

Here in this example all the tokens **Student, Address, Subjects** returns the same instance of type **Student** because all the tokens are mapping to the existing dependency.

```
var stud = injector.get(Student);
```

```
var stud1 = injector.get(Address);
var stud2 = injector.get(Subjects);
// stud === stud1 === stud3 → true
```

3. **useValue:** When the user wants to configure the application with some constants then we can choose this. We can return either a **single value or list of keys** with single token.

```
var injector = ReflectiveInjector.resolveAndCreate([
                    { provide:"ApiKey", useValue:"APIXYZ"}
                ]);
```

To return list of keys pass an object

```
var injector = ReflectiveInjector.resolveAndCreate([
                    { provide:"ApiKey",  useValue: { "key1":"APIXYZ",  "key2":"APIABC" } }
        ]);
```

When we want to make the object to be **immutable** then use **Object.freeze ()**

```
var injector = ReflectiveInjector.resolveAndCreate([
{
        provide:"ApiKey",
        useValue: Object.freeze( {
                        "key1":"APIXYZ",
                        "key2":"APIABC"
                    } )
        }
]);
```

4. **useFactory:** When the user want to call a function when the token is requested we can choose this.

```
var injector = ReflectiveInjector.resolveAndCreate([
   {
        provide:"ApiKey",
        useFactory: () => {
                                if (true) return new Student();
                                else return new Address();
                        }
   }
]);
```

Configuring Dependency Injection

To configure the injectors we have two ways

- Configuring using **Top-level Parent Injector** (**NgModule**).
- Configuring using **Components and Directives.**

Using Top-level (NgModule) injector: NgModule decorator has property called providers which can accept list of providers, which is same as the **injector** which is created using **ReflectiveInjector.**

```
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
  providers:[Student,Address,Subjects]
})
export class AppModule { }
```

Using Components / Directives: We can configure the components and directives as the same way we do for **NgModule** using **providers.**

```
@Component({
  selector: 'my-app',
  templateUrl: './template.html',
  providers:[Student,Address,Subjects]
})
```

Implementing Dependency Injection In Angular

In Angular we can achieve DI using two ways

- Using **@Inject** decorator.
- Using **@Injectable** decorator.

Step1: Registering classes with injector

Option1: Decorating constructor parameters with @Inject

Student.ts

```
import { Inject } from '@angular/core';
export class Address
{ }
export class Subjects
{ }
export class Student {
  address: Address
```

```
  subjects: Subjects
  constructor(@Inject(Address) address: Address, @Inject(Subjects) subjects: Subjects) {
    this.address = address;
    this.subjects = subjects;
  }
  drive(miles: number) {
    return "Student is being driven at " + miles + " miles";
  }
}
```

Note: For Class to be Injectable, it should either have parameter less constructor or all parameters must be decorated with Inject.

Option 2: Decorating constructor parameters with @Injector

```
import { Injectable } from '@angular/core';

@Injectable()
export class Address
{ }
@Injectable()
export class Subjects
{ }
@Injectable()
export class Student {
  address: Address
  subjects: Subjects
  constructor(address: Address, subjects: Subjects) {
    this.address = address;
    this.subjects = subjects;
  }
  drive(miles: number) {
    return "Student is being driven at " + miles + " miles";
  }
}
```

Step2: Injecting the object into the component.

Option1: Implicit Injector Creation

App.component.ts

```
import { Component} from '@angular/core';
```

```
import { ReflectiveInjector } from '@angular/core';
import { Address, Subjects, Student } from "./stud"
@Component({
   selector: 'my-app',
   template: `
      <div>{{stud.drive(10)}}</div>
`})

export class AppComponent {
   stud: Student;
   constructor()
   {
      var injector = ReflectiveInjector.resolveAndCreate([Student, Address, Subjects]);
      this.stud = injector.get(Student);
   }
}
```

Option2: Using Providers Metadata and Component Constructor (recommended approach)

app.component.ts

```
import { Component} from '@angular/core';
import { Address, Subjects, Student } from "./stud"
@Component({
   selector: 'my-app',
   providers: [Student],
   template: `
      <div>{{stud.drive(10)}}</div>
`})

export class AppComponent {
   constructor(private stud: Student)
   { }
}
```

Optional Dependencies

As in our example below, Our object *requires* a Logger, but what if it could get by without a logger? We can tell Angular that the dependency is optional by annotating the constructor argument with **@Optional()**

```
import { Optional } from '@angular/core';
```

```
constructor(@Optional() private logger: Logger) {
  if (this.logger) {
    this.logger.log(some_message);
  }
}
```

Note: When using @Optional(), our code must be prepared for a null value. If we **don't register** a logger somewhere up the line, the injector will set the **value of logger to null**.