# Dependency Injection Framework

## CS569 – Web Application Development II

Maharishi University of Management

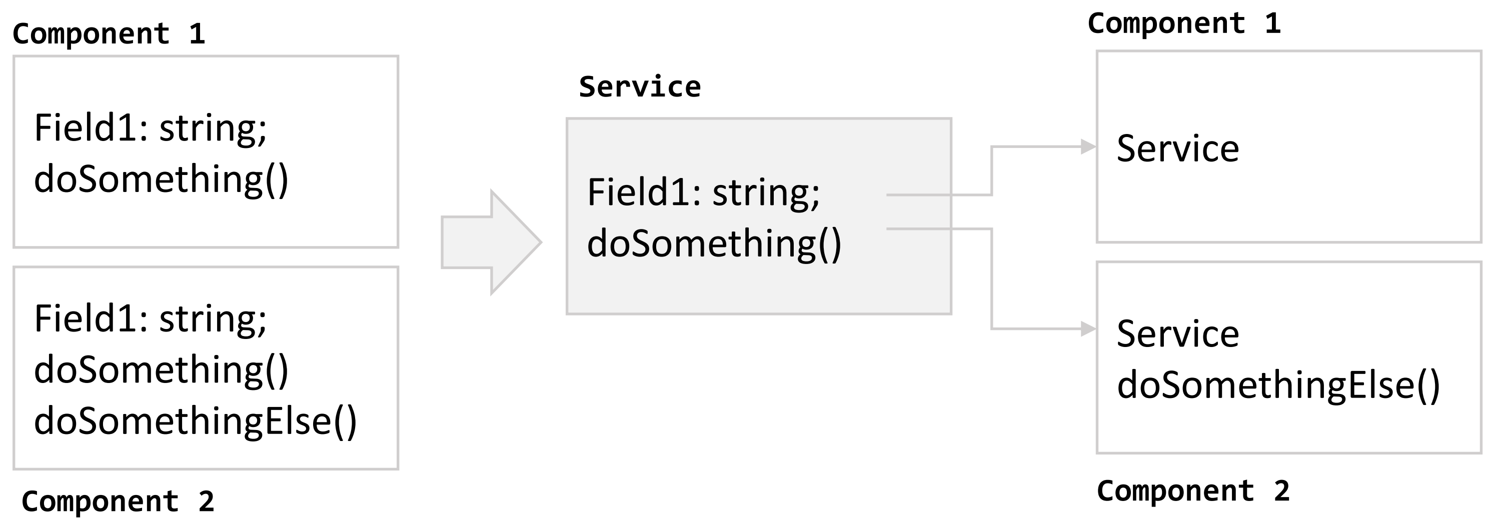
Department of Computer Science

Assistant Professor Asaad Saad

## Maharishi University of Management - Fairfield, Iowa

## Services

- Common layer to store and interact with Data (Database on server)
- Communication channel between components
- Services provide access to certain functionality from various places in your application.



Services can be used to centralize certain tasks, remove redundant code duplication or outsource heavy tasks.

## Dependency Injection

The idea behind dependency injection is very simple. If you have a component that depends on a service. You do not create that service yourself.

Instead, you request one in the constructor, and the framework will create an instance and provide it to you.

This leads to more **decoupled code** (separation of concerns), which enables testability, easy code refactoring and modularity.

## Example

Let’s imagine we have a `Product` class. Each product has a base price. In order to calculate the full price for this product, we rely on a service that takes the base price of the product when it's been initialized and calculate the price based the state we’re selling it to.

```
class Product {
        constructor(basePrice: number) {
                this.service = new PriceService();
                this.basePrice = basePrice;
        }
        price(state: string) {
                return this.service.calculate(this.basePrice, state);
        }
}
```

## A Better Way

```
class Product {
        constructor(service: PriceService, basePrice: number) {
                this.service = service;
                this.basePrice = basePrice;
        }
        price(state: string) {
                return this.service.calculate(this.basePrice, state);
        }
}
```

This technique of injecting the dependencies relies on a principle called the **Inversion of Control (IoC)** principle is also called informally the Hollywood principle (don’t call us, we’ll call you).

When we use DI we are moving towards a more **loosely coupled** architecture where changing bits and pieces of a single component affects the other areas of the application less. And, as long as the interface between those components don’t change, we can even swap them altogether, without any other components even realizing.

## Dependency Injection Parts

The dependency injection framework in Angular has these parts:

- **The Provider** maps a token (type) to a list of dependencies. It tells Angular **how to create/instantiate** an object.
- **The Injector** that holds a set of bindings and is responsible for **resolving dependencies** and injecting them when creating objects.
- **The Injectable** that is what’s being injected.

In Angular when you want to access an injectable you add the dependency into the constructor, Angular’s dependency injection framework will locate it, create an instance and provide it to you.

## Dependency Injection in Apps

When writing our apps there are three steps we need to take in order to perform an injection:

1. Create the service class (the injectable)
2. Declare the dependencies on the receiving component (in constructor)
3. Configure the injection (register the injection with Angular in our NgModule – providers[])

When we declare the injection in our component constructor, Angular will see that we are looking for an object in the constructor and check the DI system for an appropriate injection.

## Registering Providers

Angular dependency injection framework takes care of creating the required instance, but it needs to know how to create such an instance.

There are two ways to provide services

| Global | Locally (component level) |
|---|---|
| `// in the Service`<br>`@Injectable({ providedIn: 'root' })` | `// In the Component:`<br>`providers = [ MyService ]`<br>`    OR`<br>`providers = [`<br>`   {provide: MyService, useClass: MyService}`<br>`]` |

## Tree-Shakable Providers

For general services in your application, it is advised that you provide them globally using the property **`providedIn:'root'`**, and not to declare them using the property **`providers`**, this will allow the services to be tree-shakable, meaning if they are not being used, they will not be part of the production bundle.

```
@Injectable({ providedIn: 'root' })
export class MyGeneralService { }
```

## Angular Native Services

All Angular native services are provided at root by simply importing any Angular official module, you may ask for them in your constructor, Angular knows where to find them and provide an instance for you. *(There is no need to provide those)*.

## Create Services from CLI

Injecting a singleton instance of a class is probably the most common type of injection.

When you provide the service at the root level, Angular creates a single, shared instance of service and injects into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

To create a new service from Angular CLI:

```
ng g s serviceName
```

# Global Service

@Injectable() tells Angular that if this Service asked for anything in the constructor() then Angular will be able to inject it.

```
@Injectable({ providedIn: 'root' })
export class LogService {
        constructor() {}
        logText(input: string) {
                console.log(input);
        }
}
```

This tells Angular to provide the Service at the Root level and the whole application can use the same instance (singleton)

```
@Component({ selector: 'comp', template: ` `, })
export class MyComponent {
        constructor (private logService: LogService) {}
        onLog(value: string) {
                this.logService.logText(value);
        }
}
```

## Local Service

This service is not provided at root. It should be provided at the Component level

```
@Injectable()
export class LogService {
        constructor() {}
        logText(input: string) {
                console.log(input);
        }
}
```

```
@Component({ selector: 'comp', template: ` `, providers: [LogService]})
export class MyComponent {
        constructor (private logService: LogService) {}
        onLog(value: string) {
                this.logService.logText(value);
        }
}
```

## Dependency Injection Algorithm

When resolving the backend dependency of a component, Angular will start with the injector of the component itself. Then, if it is unsuccessful, it will climb up to the injector of the parent component, and, finally, will move up to until it reaches the root.

```
let inj = this;
while (inj) {
   if (inj.has(requestedDependency)) {
      return inj.get(requestedDependency);
   } else {
       inj = inj.parent;
   }
}
throw new NoProviderError(requestedDependency);
```

## Hierarchical Injector

Angular Dependency Injector is **hierarchical**. This basically means, that providers may be set up on different levels of the application, leading to different outcomes. **Normally you get an instance per provider.**

The most common pattern is that you provide all your services at root so your application can share the **same instance**.

If providers are specified on each individual component, **different instances** will be created.

## `providedIn`

Determines which injectors will provide the injectable:

**`'root'`** The application-level injector in most apps.

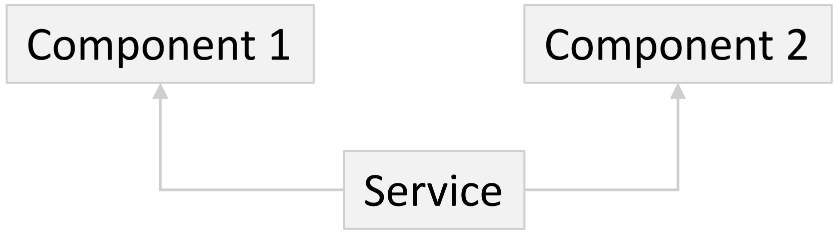
**`'platform'`** A special singleton platform injector shared by all applications on the page.

**`'any'`** Provides a unique instance in every module (including lazy modules).

Marking a class with `@Injectable` ensures that the compiler will generate the necessary metadata to create the class's dependencies when the class is injected.

# Cross-Component Communication

We can implement **Push-Notifications'** like feature using Services:



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

@Injectable({ providedIn: 'root' })
export class MyService {
        // A channel so component1 and component2 can exchange data
        obs$ = new EventEmitter<string>();
}
```

# Cross-Component Communication

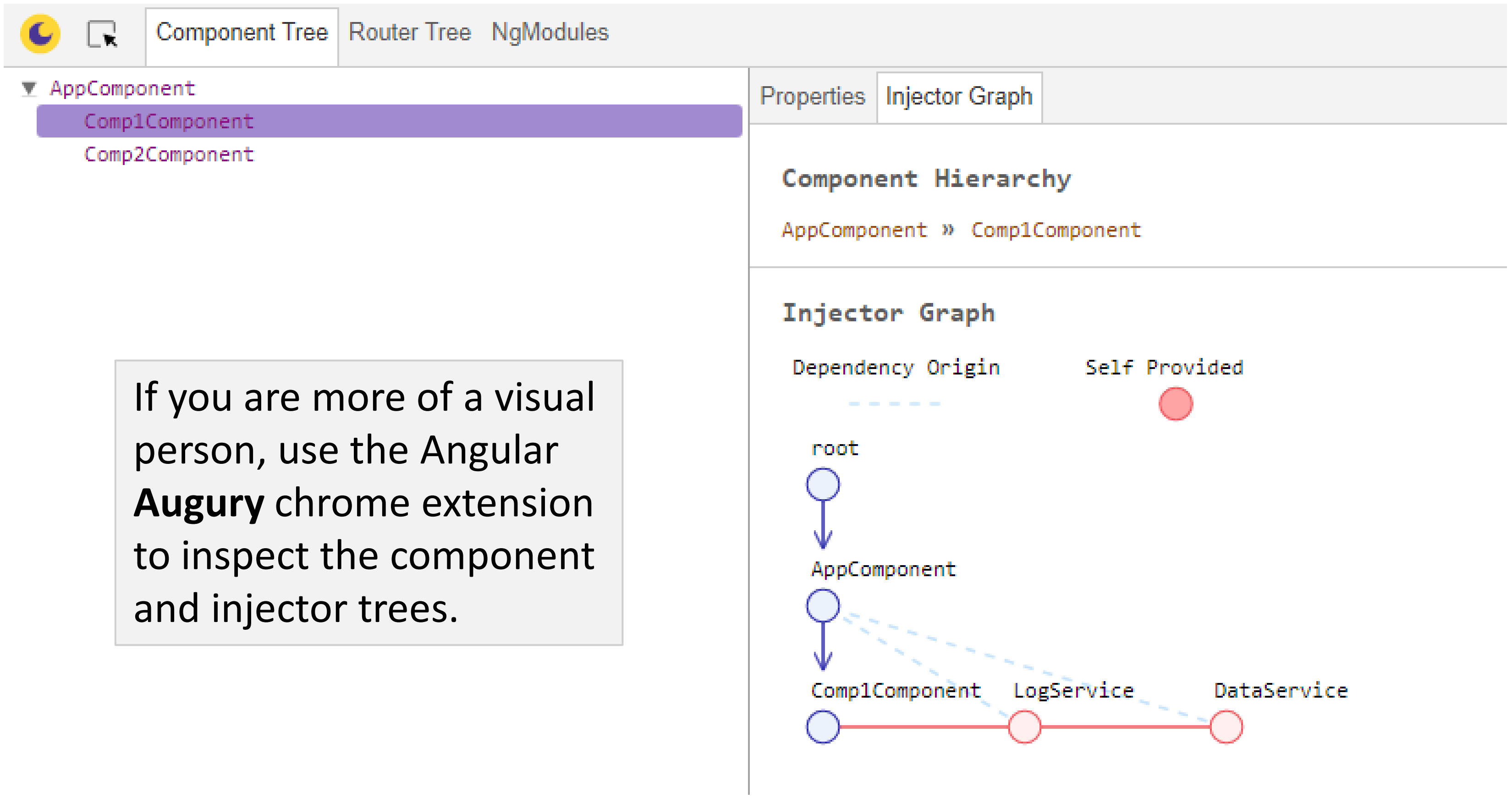
next()

```
export class Comp1Component {
        constructor (private myService: MyService) {}
        onSend(value: string) {
                this.myService.obs$.next(value);
        }
}
```

subscribe()

```
export class Comp2Component implements OnInit {
        private value = '';
        constructor (private myService: MyService) {}
        ngOnInit() {
             this.myService.obs$.subscribe(data => this.value = data);
        }
}
```

# Visualizing Injector Tree



If you are more of a visual person, use the Angular **Augury** chrome extension to inspect the component and injector trees.

## Define an Aliased Class Provider

In case we want to replace a service with a new version without breaking the code, we can define an alias service provider.

```
// this will create two difference instances of MyNewService
providers: [
        MyNewService,
        {provide: MyService, useClass: MyNewService}
]

// this will create one instance
providers: [
        MyNewService,
        {provide: MyService, useExisting: MyNewService}
]
```

## Using a JavaScript Object as a Value Provider

You can simply create a provider out of a simple JavaScript object rather than a full class.

```
providers: [
        {provide: MyService, useValue: myObj}
]

const myObj = {
        log(msg: string){
                console.log(msg);
        }
}
```

## Instantiate a Service using Factory Provider

Factory providers in Angular allow to define a function which will then be called by the dependency injector to construct the given service instance.

```
export class MyService{
       constructor(private isEnabled: boolean){}
       ...
}

providers: [
       {provide: MyService, useFactory: myFactory}
]

const myFactory = ()=>{
       return new MyService(true);
}
```

## @Self()

If we decorate the dependency in the constructor with **`@Self()`**, the only place allowed to find the injector is the component itself.

```
@component({
      providers = [myService]}
)
... // look only to local provider, skip the higher levels
constructor(@Self() s: myService) {}
```

## @Optional()

if your component doesn’t absolutely need that service, you can decorate a dependency with the **`@Optional()`** decorator and in such case of no provider found, no error will occur. Instead Angular will set the value for our service to **`null`**

```
// could be provided at higher level, if not: no error.
constructor(@Optional() s: myService) {}
```

# @SkipSelf()

**`@SkipSelf()`** decorator will be skipping the step of looking for a possible injector in the requesting component.

```
// it is definitely provided at higher level
constructor(@SkipSelf() s: myService) {}
```

# @Host()

**`@Host()`** decorator makes Angular look for the injector on the component **itself**, so in that regard it may look similar to the **`@Self()`**. But if the injector is not found there, it looks for the injector **one level up** on its host component.

```
// service is provided either here or on it's host
constructor(@Host() s: myService) {}
```

**We can have:** `constructor (@Host() @SkipSelf() @Optional() s: myService) {...}`

## Main Points

Dependency injection is a key component of Angular.

You can configure dependency injection at the component or root level.

Dependency injection allows us to depend on interfaces rather than concrete types. This results in more decoupled code and improves testability.