**Understanding Dependency Injection (DI) in Angular**

Dependency Injection (DI) is a fundamental concept in Angular. It's a design pattern where a class receives its dependencies from an external source rather than creating them itself. This promotes loose coupling, testability, and maintainability.

**Constructor-Based Hierarchical Injector**

Angular uses a hierarchical injector system. Each component has its own injector, and injectors form a tree structure mirroring the component tree. When a dependency is requested, the injector searches for it in its own registry. If not found, it delegates the search to its parent injector.

TypeScript

import { Component, Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class DataService {

getData() {

return 'Some data';

}

}

@Component({

selector: 'app-component',

template: `

<p>Data: {{ dataService.getData() }}</p>

`,

})

export class AppComponent {

constructor(private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example, DataService is provided in the root injector using providedIn: 'root'. When AppComponent is created, Angular injects an instance of DataService into its constructor.

**Singleton and Non-Singleton Objects**

* **Singleton:** A single instance of a service is shared across the entire application. This is achieved by providing the service in the root injector using providedIn: 'root'.
* **Non-Singleton:** A new instance of a service is created for each component or injector. This is achieved by providing the service at the component level or using providedIn: 'any'.

TypeScript

import { Component, Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class SingletonService {

count = 0;

increment() {

this.count++;

}

}

@Injectable()

export class NonSingletonService {

value: number;

constructor(public id: number) {

this.value = id;

}

}

@Component({

selector: 'app-component',

template: `

<p>Singleton count: {{ singletonService.count }}</p>

<p>Non-singleton value: {{ nonSingletonService.value }}</p>

`,

})

export class AppComponent {

constructor(

public singletonService: SingletonService,

private nonSingletonService: NonSingletonService

) {

singletonService.increment();

}

}

Use code [with caution.](/faq#coding)

In this example, SingletonService is a singleton, while NonSingletonService is non-singleton.

**Using Service Classes**

Service classes are commonly used for sharing data and logic across components. They are injected into components using the constructor.

TypeScript

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

@Injectable({ providedIn: 'root' })

export class UserService {

getUserData() {

// ...

}

}

Use code [with caution.](/faq#coding)

**Working with Decorators in DI**

* **@Injectable():** Marks a class as injectable, allowing it to be injected into other classes.
* **@Injector():** Provides access to the injector instance.

TypeScript

import { Component, Injectable, Injector } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class DataService {

getData() {

return 'Some data';

}

}

@Component({

selector: 'app-component',

template: `

<p>Data: {{ dataService.getData() }}</p>

`,

})

export class AppComponent {

constructor(private injector: Injector) {

const dataService = injector.get(DataService);

console.log(dataService);

}

}

Use code [with caution.](/faq#coding)

**Advanced Topics**

* **Tree-shakable injectors:** Angular can optimize the injector tree by removing unused injectors at build time.
* **Lazy loading:** Injectors can be lazily loaded when needed, improving performance.
* **Custom injectors:** You can create custom injectors for specific use cases.
* **Provider factories:** You can customize the creation of dependencies using provider factories.
* **Dependency resolution:** Understanding how Angular resolves dependencies, including cyclic dependencies and optional dependencies.

**Additional Considerations**

* **Best practices:** Follow Angular's style guide and best practices for DI.
* **Testing:** Write unit tests to ensure the corr

**Constructor-Based Hierarchical Injector in Angular**

Angular employs a sophisticated dependency injection (DI) system built on a hierarchical injector structure. This means each component within an Angular application possesses its own injector, and these injectors are organized in a tree-like fashion mirroring the component hierarchy.

**How it works:**

1. **Component Creation:** When a component is instantiated, Angular creates an injector for it.
2. **Dependency Request:** When a component requires a dependency (like a service), it declares it in its constructor.
3. **Injector Search:** The component's injector attempts to locate the requested dependency within its own registry.
4. **Hierarchical Search:** If the dependency is not found in the local injector, the search propagates up the injector hierarchy towards the root injector.
5. **Dependency Resolution:** Once the dependency is found, the injector creates or retrieves an instance and injects it into the component.

**Key Points:**

* **Hierarchical Structure:** Injectors form a tree-like structure, with the root injector at the top and child injectors for each component.
* **Dependency Scope:** The scope of a dependency is determined by where it's provided. A dependency provided at the root level is accessible to all components, while a dependency provided at the component level is only accessible within that component and its children.
* **Provider Registration:** You can register providers for dependencies at different levels of the injector hierarchy using the providers array in the component's @Component decorator or the providers array in the module's @NgModule decorator.
* **Constructor Injection:** The most common way to inject dependencies into a component is by declaring them as constructor parameters.

**Example:**

TypeScript

import { Component, Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class DataService {

getData(): string {

return 'Some data from root';

}

}

@Component({

selector: 'app-component',

template: `

<p>Data: {{ dataService.getData() }}</p>

`,

})

export class AppComponent {

constructor(private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example:

* DataService is provided at the root level using providedIn: 'root'.
* AppComponent injects DataService into its constructor.
* When AppComponent is created, its injector searches for DataService.
* Since DataService is provided at the root level, it's found and injected into AppComponent.

**Advanced Considerations:**

* **Lazy Loading:** Angular can optimize the injector tree by lazy loading injectors when needed, improving performance.
* **Custom Injectors:** You can create custom injectors for specific use cases, providing more granular control over dependency resolution.
* **Provider Factories:** You can customize the creation of dependencies using provider factories.
* **Dependency Resolution:** Understanding how Angular resolves dependencies, including cyclic dependencies and optional dependencies, is crucial for advanced usage.

By comprehending the hierarchical injector system, you can effectively manage dependencies in your Angular applications, promoting loose coupling, testability, and maintainability.

**Understanding Injectors in Angular**

**Imagine a restaurant kitchen.**

* **Ingredients:** These are your services in Angular. They provide the necessary elements for your application to function.
* **Chef:** This is your component. It needs ingredients (services) to create the final dish (component output).
* **Kitchen:** This is the injector. It manages the ingredients and provides them to the chef when needed.

**What is an Injector?**

In Angular, an injector is a service locator that creates and manages instances of objects (like services, components, and directives). It's responsible for resolving dependencies between these objects. Think of it as the 'middleman' that connects the dots.

[1. Introduction to services and dependency injection - Angular](https://angular.io/guide/architecture-services" \l ":~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service" \t "_blank)

[[Source icon](https://angular.io/guide/architecture-services#:~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service)](https://angular.io/guide/architecture-services" \l ":~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service" \t "_blank)

[angular.io](https://angular.io/guide/architecture-services" \l ":~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service" \t "_blank)

[2. Angular Dependency Injection and the function of Injectors & Providers - mobileLIVE](https://www.mobilelive.ca/blog/angular-dependency-injection" \l ":~:text=This%20Angular%20injector%20looks%20for,referred%20to%20as%20DI%20Container." \t "_blank)

[[Source icon](https://www.mobilelive.ca/blog/angular-dependency-injection#:~:text=This%20Angular%20injector%20looks%20for,referred%20to%20as%20DI%20Container.)](https://www.mobilelive.ca/blog/angular-dependency-injection" \l ":~:text=This%20Angular%20injector%20looks%20for,referred%20to%20as%20DI%20Container." \t "_blank)

[www.mobilelive.ca](https://www.mobilelive.ca/blog/angular-dependency-injection" \l ":~:text=This%20Angular%20injector%20looks%20for,referred%20to%20as%20DI%20Container." \t "_blank)

**How it works:**

1. **Registration:** You register services (or other injectable objects) with an injector. This is usually done using the providers array in a component, directive, or module.

[1. Introduction to services and dependency injection - Angular](https://angular.io/guide/architecture-services" \t "_blank)

[[Source icon](https://angular.io/guide/architecture-services)](https://angular.io/guide/architecture-services" \t "_blank)

[angular.io](https://angular.io/guide/architecture-services" \t "_blank)

1. **Dependency Injection:** When a component or service needs another object, it declares it in its constructor. The injector then finds and provides the required instance.

[1. Understanding dependency injection - Angular](https://angular.io/guide/dependency-injection" \t "_blank)

[[Source icon](https://angular.io/guide/dependency-injection)](https://angular.io/guide/dependency-injection" \t "_blank)

[angular.io](https://angular.io/guide/dependency-injection" \t "_blank)

[2. Introduction to services and dependency injection - Angular](https://angular.io/guide/architecture-services" \l ":~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service" \t "_blank)

[[Source icon](https://angular.io/guide/architecture-services#:~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service)](https://angular.io/guide/architecture-services" \l ":~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service" \t "_blank)

[angular.io](https://angular.io/guide/architecture-services" \l ":~:text=If%20a%20requested%20service%20instance,injector%20before%20returning%20the%20service" \t "_blank)

**Simple Example:**

TypeScript

import { Component, Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class DataService {

getData(): string {

return 'Some data';

}

}

@Component({

selector: 'app-my-component',

template: `

<p>Data: {{ dataService.getData() }}</p>

`

})

export class MyComponent {

constructor(private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example:

* DataService is the ingredient (service).
* MyComponent is the chef (component).
* The injector is the implicit mechanism that provides DataService to MyComponent when it's created.

**Key Points:**

* **Hierarchical Injectors:** Angular has a hierarchical injector system. Each component has its own injector, and they form a tree-like structure. This allows for different scopes of dependencies.

[1. Hierarchical Dependency Injectors - ts - GUIDE - Angular](https://v2.angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html" \l ":~:text=Angular%20has%20a%20Hierarchical%20Dependency,parallel%20an%20application's%20component%20tree." \t "_blank)

[[Source icon](https://v2.angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html#:~:text=Angular%20has%20a%20Hierarchical%20Dependency,parallel%20an%20application's%20component%20tree.)](https://v2.angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html" \l ":~:text=Angular%20has%20a%20Hierarchical%20Dependency,parallel%20an%20application's%20component%20tree." \t "_blank)

[v2.angular.io](https://v2.angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html" \l ":~:text=Angular%20has%20a%20Hierarchical%20Dependency,parallel%20an%20application's%20component%20tree." \t "_blank)

* **Provider:** A provider is a set of instructions telling the injector how to create an instance of a dependency.

[1. Angular Dependency Injection Explained with Examples - freeCodeCamp](https://www.freecodecamp.org/news/angular-dependency-injection/" \l ":~:text=Injectors%20are%20data%20structures%20that,class%20as%20its%20data%20type." \t "_blank)

[[Source icon](https://www.freecodecamp.org/news/angular-dependency-injection/#:~:text=Injectors%20are%20data%20structures%20that,class%20as%20its%20data%20type.)](https://www.freecodecamp.org/news/angular-dependency-injection/" \l ":~:text=Injectors%20are%20data%20structures%20that,class%20as%20its%20data%20type." \t "_blank)

[www.freecodecamp.org](https://www.freecodecamp.org/news/angular-dependency-injection/" \l ":~:text=Injectors%20are%20data%20structures%20that,class%20as%20its%20data%20type." \t "_blank)

* **Dependency Injection:** The process of providing dependencies to a class without it having to create them itself.

[1. Angular Dependency Injection: A Complete Guide - Simplilearn.com](https://www.simplilearn.com/tutorials/angular-tutorial/angular-dependency-injection" \l ":~:text=DI%20is%20a%20coding%20pattern,rather%20than%20creating%20them%20itself." \t "_blank)

[[Source icon](https://www.simplilearn.com/tutorials/angular-tutorial/angular-dependency-injection#:~:text=DI%20is%20a%20coding%20pattern,rather%20than%20creating%20them%20itself.)](https://www.simplilearn.com/tutorials/angular-tutorial/angular-dependency-injection" \l ":~:text=DI%20is%20a%20coding%20pattern,rather%20than%20creating%20them%20itself." \t "_blank)

[www.simplilearn.com](https://www.simplilearn.com/tutorials/angular-tutorial/angular-dependency-injection" \l ":~:text=DI%20is%20a%20coding%20pattern,rather%20than%20creating%20them%20itself." \t "_blank)

**Additional Notes:**

* Injectors are essential for testability. You can easily mock dependencies in tests.

[1. Injectors in Angular: A Comprehensive Guide with Example | by Gurunadh Pukkalla](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408" \l ":~:text=1.,resolving%20dependencies%20between%20these%20objects." \t "_blank)

[[Source icon](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408#:~:text=1.,resolving%20dependencies%20between%20these%20objects.)](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408" \l ":~:text=1.,resolving%20dependencies%20between%20these%20objects." \t "_blank)

[medium.com](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408" \l ":~:text=1.,resolving%20dependencies%20between%20these%20objects." \t "_blank)

* Understanding injectors helps you optimize your application's performance by controlling the creation of objects.
* Angular provides several built-in injectors, but you can also create custom injectors for specific needs.

[1. Injectors in Angular: A Comprehensive Guide with Example | by Gurunadh Pukkalla](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408" \l ":~:text=Creating%20a%20custom%20injector%20in,an%20injector%20with%20specific%20providers." \t "_blank)

[[Source icon](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408#:~:text=Creating%20a%20custom%20injector%20in,an%20injector%20with%20specific%20providers.)](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408" \l ":~:text=Creating%20a%20custom%20injector%20in,an%20injector%20with%20specific%20providers." \t "_blank)

[medium.com](https://medium.com/@gurunadhpukkalla/injectors-in-angular-a-comprehensive-guide-with-example-d9b7f57a1408" \l ":~:text=Creating%20a%20custom%20injector%20in,an%20injector%20with%20specific%20providers." \t "_blank)

By grasping the concept of injectors, you'll have a better understanding of how Angular manages dependencies and builds efficient applications.

**Would you like to explore specific use cases or advanced topics related to injectors?**

**Understanding the Process**

When a component requests a service, Angular's injector follows a specific search path to locate and provide the necessary instance.

1. **Component Constructor:** The component declares the service as a dependency in its constructor.
2. **Injector Search:**
   * The injector associated with the component is consulted first.
   * If the service is found in the component's injector, an instance is provided.
   * If not found, the search continues up the injector hierarchy to the parent injector.
   * This process continues until the service is found or the root injector is reached.
3. **Instance Creation:** If no existing instance is found, the injector creates a new instance of the service based on the registered provider.
4. **Dependency Injection:** The created or retrieved service instance is injected into the component.

**Simple Example**

TypeScript

import { Component, Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class DataService {

getData(): string {

return 'Some data';

}

}

@Component({

selector: 'app-my-component',

template: `

<p>Data: {{ dataService.getData() }}</p>

`

})

export class MyComponent {

constructor(private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example:

* DataService is registered at the root level using providedIn: 'root'.
* When MyComponent is created, its injector looks for DataService.
* Since DataService is not found in the component's injector, the search moves to the parent injector (which is the root injector in this case).
* The root injector finds DataService and creates an instance if it doesn't exist.
* The created instance is injected into MyComponent.

**Key Points**

* **Hierarchical Structure:** Injectors are organized in a hierarchical manner, allowing for different scopes of services.
* **Provider Registration:** Services are registered with injectors using providers.
* **Instance Creation:** If a service instance doesn't exist, the injector creates one based on the provider.
* **Dependency Resolution:** The injector resolves dependencies by searching through the injector hierarchy.

By understanding this process, you can effectively manage dependencies in your Angular applications and optimize performance by controlling the creation and management of service instances.

**Understanding Dependency Resolution Decorators**

**@Optional()**

This decorator indicates that a dependency is optional. If the dependency is not found, the injected value will be null instead of throwing an error.

**Example:**

TypeScript

import { Injectable, Optional } from '@angular/core';

@Injectable({ providedIn: 'root' })

export class ConfigService {

config: any;

constructor(@Optional() config: any) {

this.config = config || { defaultConfig: true };

}

}

Use code [with caution.](/faq#coding)

In this example, ConfigService has an optional dependency on a config object. If the config object is not provided, the config property in the service will be initialized with a default value.

**@Self()**

This decorator instructs the DI framework to look for the dependency only in the local injector (injector of the current component or directive). It prevents the DI framework from searching for the dependency in parent injectors.

**Example:**

TypeScript

import { Component, Self } from '@angular/core';

import { DataService } from './data.service';

@Component({

selector: 'app-child',

template: `

<p>Data from child: {{ dataService.getData() }}</p>

`

})

export class ChildComponent {

constructor(@Self() private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example, the ChildComponent explicitly requests the DataService from its own injector, ensuring that any DataService provided by a parent component is ignored.

**@SkipSelf()**

This decorator instructs the DI framework to look for the dependency in parent injectors, skipping the local injector. It's useful when you want to access a service from a parent component without creating a new instance.

**Example:**

TypeScript

import { Component, SkipSelf } from '@angular/core';

import { DataService } from './data.service';

@Component({

selector: 'app-child',

template: `

<p>Data from parent: {{ dataService.getData() }}</p>

`

})

export class ChildComponent {

constructor(@SkipSelf() private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example, the ChildComponent explicitly requests the DataService from the parent injector, ensuring that any DataService provided by the child component itself is ignored.

**@Host()**

This decorator resolves the dependency in the view by checking injectors of child elements, stopping at the host element of the current component. It's primarily used with directives.

**Example:**

TypeScript

import { Directive, Host } from '@angular/core';

import { DataService } from './data.service';

@Directive({

selector: '[appMyDirective]'

})

export class MyDirective {

constructor(@Host() private dataService: DataService) {}

}

Use code [with caution.](/faq#coding)

In this example, the MyDirective will inject the DataService from the component that hosts the directive.

**Note:** These decorators provide fine-grained control over dependency resolution, but use them judiciously as they can make code harder to understand and maintain.

**Would you like to explore a specific use case or learn more about a particular decorator?**

**@Self, @SkipSelf & @Optional Decorators Angular**

[3 Comments](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comments) / 6 minutes of reading / March 9, 2023

[**Providedin**](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/)

[**Angular Tutorial**](https://www.tektutorialshub.com/angular-tutorial/)

[**Host Decorator**](https://www.tektutorialshub.com/angular/host-decorator-in-angular/)

@Self, @SkipSelf, @Optional & [@Host](https://www.tektutorialshub.com/angular/host-decorator-in-angular/) are Angular Decorators that configure how the DI Framework should resolve the dependencies. These decorators are called *Resolution Modifiers* because they modify the behavior of injectors. In this tutorial, we will learn @Self, @SkipSelf, & @Optional. We will look at the [@Host](https://www.tektutorialshub.com/angular/host-decorator-in-angular/) in the next tutorial

**Table of Contents**

* [How Angular DI Framework Resolves Dependencies](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#how-angular-di-framework-resolves-dependencies)
* [@Self, @SkipSelf & @Optional Example](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#self-skipself-optional-example)
* [@Self](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#self)
* [@SkipSelf](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#skipself)
* [@Optional](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#optional)
* [Source Code](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#source-code)
* [Reference](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#reference)

**How Angular DI Framework Resolves Dependencies**

When a component asks for Dependency, the DI Framework resolves it in two phases.

In the first phase, it starts to look for the Dependency in the current component’s ElementInjector. If it does not provide the Dependency, it will look in the Parent Components ElementInjector. The Request bubbles up until it finds an injector that provides the service or reaches the root ElementInjector.

If ElementInjector does not satisfy the request, Angular looks for the Dependency in the ModuleInjector hierarchy. If Angular still doesn’t find the provider, it throws an error.

The older versions of the Angular created only one Injector tree. But in the later versions, the tree was split into two trees. One is ElementInjector for elements (components, directives & pipes etc) and the other one is ModuleInjector for [Angular Modules](https://www.tektutorialshub.com/angular/angular-modules/).

**@Self, @SkipSelf & @Optional Example**

We have created an example project in Angular to explain the @Self, @SkipSelf, & @Optional. You can find the Source Code in [StackBlitz](https://stackblitz.com/edit/angular-self-optional-skipself-example?file=src/app/app.component.ts).

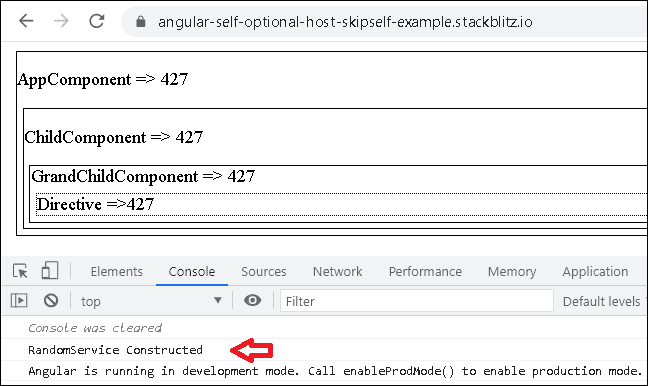
The Code contains a RandomService, which generates a Random Number when initialized. The [Angular Service](https://www.tektutorialshub.com/angular/angular-services/) is added to the Providers array of the AppModule. We can inject this service anywhere in our Application.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Injectable({    providedIn: "root"  })  export **class** RandomService { |

The project contains three [Angular Components](https://www.tektutorialshub.com/angular/angular-component/) (AppComponent, ChildComponent & GrandChildComponent) all inject the RandomService and displays the Random Number from the Service.

We also have testDirective, which we include in the template of GrandChildComponent. It also displays the Random Number from the Service.

Ensure that the Providers array is empty in all components & directives. Run the App. Angular creates only one instance of the RandomService. That is why all the components and directives show the same number.



Now, let us check how we can modify the above behavior with @Self, @SkipSelf, & @Optional.

First let us start with @Self

**@Self**

The @Self decorator instructs Angular to look for the dependency only in the local injector. The local injector is the injector that is part of the current component or directive.

Open the GrandChildComponent and add the @Self() on randomService as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Component({    selector: "my-grandChild",    template: `      <div **class**="box">        GrandChildComponent => {{ randomNo }}        <div **class**="dirbox" testDirective>fdf</div>      </div>    `,    providers: []  })  export **class** GrandChildComponent {    randomNo;    constructor(@Self() private randomService: RandomService) {  **this**.randomNo = randomService.RandomNo;    }  } |

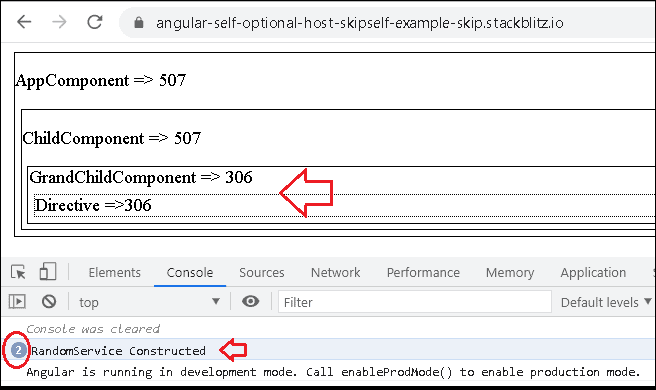
This forces the Angular DI Framework to look for the Dependency attached to the current Component. Since it does find one it will throw the error

**Error: NG0201: No provider for RandomService found in NodeInjector**

Add the RandomService to the providers array of the GrandChildComponent and the error goes away.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Component({    selector: "my-grandChild",    template: `      <div **class**="box">        GrandChildComponent => {{ randomNo }}        <div **class**="dirbox" testDirective>fdf</div>      </div>    `,    providers: [RandomService]  })  export **class** GrandChildComponent {    randomNo;    constructor(@Self() private randomService: RandomService) {  **this**.randomNo = randomService.RandomNo;    }  } |

As you can see from the image Angular creates two instances of RandomService. One from the AppModule and another from the GrandChildComponent. Also, note that testDirective picks up the RandomService provided from the GrandChildComponent and not from the AppModule



**@SkipSelf**

The @SkipSelf decorator instructs Angular to look for the dependency in the Parent Injector and upwards.

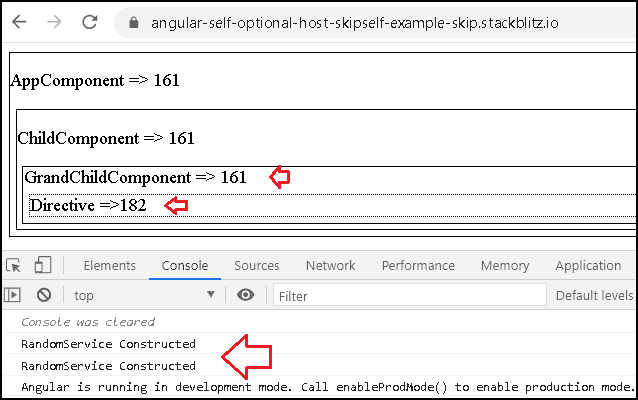
It tells [Angular](https://www.tektutorialshub.com/angular-tutorial/) not to look for the injector in the local injector, but start from the Parent. You can think of this decorator as the opposite of the @Self

Open the GrandChildComponent again. Add the SkipSelf instead of Self decorator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div **class**="box">        GrandChildComponent => {{ randomNo }}        <div **class**="dirbox" testDirective>fdf</div>      </div>    `,    providers: [RandomService]  })  export **class** GrandChildComponent {    randomNo;    constructor(@SkipSelf() private randomService: RandomService) {  **this**.randomNo = randomService.RandomNo;    }  } |

As you can see from the image, the GrandChildComponent, picks up RandomService instance provided by the Module and not the one provided by itself.

But, the testDirective still picks up the RandomService provided by the GrandChildComponent.



**@Optional**

Optional marks the dependency as Optional. If the dependency is not found, then it returns null instead of throwing an error

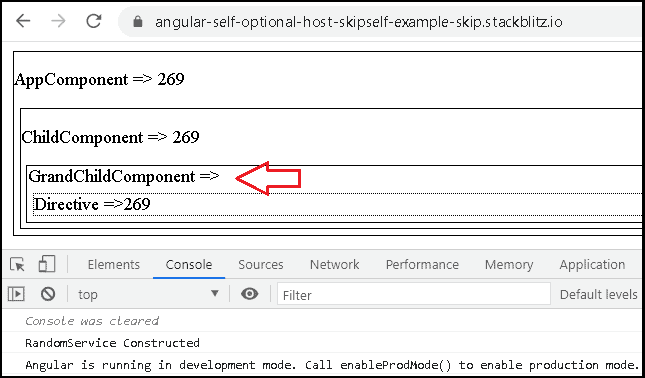
In the GrandChildComponent remove the RandomService from the Providers Array and add the @Self decorator. You will instantly receive the error “*No provider for RandomService found in NodeInjector*“.

Add the @Optional decorator along with the @Self. Now, the dependency injection will return null instead of an error.

Also, remember to add the ? in randomService?, else you will get the “*Cannot read property ‘RandomNo’ of null*” error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div **class**="box">        GrandChildComponent => {{ randomNo }}        <div **class**="dirbox" testDirective>fdf</div>      </div>    `,    providers: []  })  export **class** GrandChildComponent {    randomNo;    constructor(@Optional() @Self() private randomService: RandomService) {  **this**.randomNo = randomService?.RandomNo;    }  } |

As you can see in the image, GrandChildComponent does not receive any values, while testDirective picks up the RandomService provided by the AppModule



**Source Code**

***app.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component, VERSION } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-app",    providers: [],    viewProviders: [],    template: `      <div **class**="box">        <p>AppComponent => {{ randomNo }}</p>        <my-child></my-child>      </div>    `  })  export **class** AppComponent {    randomNo;    constructor(private randomService: RandomService) {  **this**.randomNo = randomService.RandomNo;    }  } |

***child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-child",    providers: [],    viewProviders: [],    template: `      <div **class**="box">        <p>ChildComponent => {{ randomNo }}</p>          <my-grandChild></my-grandChild>      </div>    `  })  export **class** ChildComponent {    randomNo;    constructor(private randomService: RandomService) {  **this**.randomNo = randomService.RandomNo;    }  } |

***grand-child.component.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import { Component, SkipSelf, Self, Optional, Host } from "@angular/core";  import { RandomService } from "./random-service";    @Component({    selector: "my-grandChild",    template: `      <div **class**="box">        GrandChildComponent => {{ randomNo }}        <div **class**="dirbox" testDirective>fdf</div>      </div>    `,    providers: []  })  export **class** GrandChildComponent {    randomNo;    constructor(private randomService: RandomService) {  **this**.randomNo = randomService.RandomNo;    }  } |

***test-directive.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | import {    Directive,    ElementRef,    Input,    OnInit,    SkipSelf,    Self,    Optional,    Host  } from "@angular/core";  import { RandomService } from "./random-service";    @Directive({    selector: "[testDirective]",    providers: []  })  export **class** testDirective implements OnInit {    @Input() ttClass: **string**;      constructor(private el: ElementRef, private randomService: RandomService) {}      ngOnInit() {  **this**.el.nativeElement.innerHTML =        "Directive =>" + **this**.randomService.RandomNo;    }  } |

***random-service.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import { Injectable } from "@angular/core";    @Injectable({    providedIn: "root"  })  export **class** RandomService {    private \_randomNo = 0;      constructor() {      console.log("RandomService Constructed");  **this**.\_randomNo = Math.floor(Math.random() \* 1000 + 1);    }    **get** RandomNo() {  **return** **this**.\_randomNo;    }  } |

***styles.css***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | .box {    margin: 5px;    border: 1px;    border-style: solid;  }    .dirbox {    margin: 5px;    border: 1px;    border-style: dotted;  } |

***app.module.ts***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { NgModule } from "@angular/core";  import { BrowserModule } from "@angular/platform-browser";  import { FormsModule } from "@angular/forms";    import { AppComponent } from "./app.component";  import { ChildComponent } from "./child.component";  import { GrandChildComponent } from "./grand-child.component";  import { testDirective } from "./test-directive";    @NgModule({    imports: [BrowserModule, FormsModule],    declarations: [      AppComponent,      ChildComponent,      GrandChildComponent,      testDirective    ],    bootstrap: [AppComponent]  })  export **class** AppModule {} |

**Reference**

* [Source Code](https://stackblitz.com/edit/angular-self-optional-skipself-example?file=src/app/app.component.ts)
* [Resolution Modifiers](https://angular.io/guide/hierarchical-dependency-injection#resolution-modifiers)
* [@Self API](https://angular.io/api/core/Self)
* [@SkipSelf API](https://angular.io/api/core/SkipSelf)
* [@Optional API](https://angular.io/api/core/Optional)
* [@Host API](https://angular.io/api/core/Host)

**Read More**

1. [**Angular Tutorial**](https://www.tektutorialshub.com/angular-tutorial/)
2. [**Services**](https://www.tektutorialshub.com/angular/angular-services/)
3. [**Dependency injection**](https://www.tektutorialshub.com/angular/angular-dependency-injection/)
4. [**Injector, @Injectable & @Inject**](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/)
5. [**Providers**](https://www.tektutorialshub.com/angular/angular-providers/)
6. [**Injection Token**](https://www.tektutorialshub.com/angular/injection-token-in-angular/)
7. [**Hierarchical Dependency Injection**](https://www.tektutorialshub.com/angular/how-dependency-injection-resolution-works-in-angular/)
8. [**Angular Singleton Service**](https://www.tektutorialshub.com/angular/angular-singleton-service/)
9. [**ProvidedIn root, any & platform**](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/)
10. [**@Self, @SkipSelf & @Optional Decorators**](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/)
11. [**@Host Decorator in Angular**](https://www.tektutorialshub.com/angular/host-decorator-in-angular/)

[**Providedin**](https://www.tektutorialshub.com/angular/providedin-root-any-platform-in-angular/)

[**Angular Tutorial**](https://www.tektutorialshub.com/angular-tutorial/)

[**Host Decorator**](https://www.tektutorialshub.com/angular/host-decorator-in-angular/)

**1shares**

* [**Facebook**1](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fwww.tektutorialshub.com%2Fangular%2Fself-skipself-optional-decorators-angular%2F&t=%40Self%2C%20%40SkipSelf%20%26%20%40Optional%20Decorators%20Angular)
* [**X**](https://x.com/intent/tweet?text=%40Self%2C%20%40SkipSelf%20%26%20%40Optional%20Decorators%20Angular&url=https%3A%2F%2Fwww.tektutorialshub.com%2Fangular%2Fself-skipself-optional-decorators-angular%2F)
* Pinterest
* [**LinkedIn**](https://www.linkedin.com/shareArticle?url=https%3A%2F%2Fwww.tektutorialshub.com%2Fangular%2Fself-skipself-optional-decorators-angular%2F&title=%40Self%2C%20%40SkipSelf%20%26%20%40Optional%20Decorators%20Angular&summary=%40Self%2C%20%40SkipSelf%2C%20%40Optional%20are%20Angular%20Decorators%20configure%20how%20the%20DI%20resolve%20the%20dependencies.%20They%20modify%20the%20behavior%20of%20injectors.&mini=true)

**Related Posts**

[**Best Resources to Learn Angular**](https://www.tektutorialshub.com/angular/angular-list-of-learning-resources/)

[1 Comment](https://www.tektutorialshub.com/angular/angular-list-of-learning-resources/#comments) / [Angular](https://www.tektutorialshub.com/category/angular/) / By [TekTutorialsHub](https://www.tektutorialshub.com/author/tektutorialshub/" \o "View all posts by TekTutorialsHub)

[**Introduction to Angular | What is Angular?**](https://www.tektutorialshub.com/angular/angular-introduction/)

[1 Comment](https://www.tektutorialshub.com/angular/angular-introduction/#comments) / [Angular](https://www.tektutorialshub.com/category/angular/) / By [TekTutorialsHub](https://www.tektutorialshub.com/author/tektutorialshub/" \o "View all posts by TekTutorialsHub)

3 thoughts on “@Self, @SkipSelf & @Optional Decorators Angular”

1. **Anonymous**

[April 12, 2023 at 9:02 pm](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comment-67090)

It war really cool.  
TY!

[Reply](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comment-67090)

1. **Roohi**

[August 26, 2022 at 3:35 am](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comment-48167)

It was easily understood.  
Thank you very much.

[Reply](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comment-48167)

1. **Naresh**

[July 30, 2021 at 10:34 pm](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comment-10841)

Awesome

[Reply](https://www.tektutorialshub.com/angular/self-skipself-optional-decorators-angular/#comment-10841)

Leave a Comment

Top of Form

Your email address will not be published. Required fields are marked \*

Type here..

Name

Email

Website

Bottom of Form

This site uses Akismet to reduce spam. [Learn how your comment data is processed](https://akismet.com/privacy/).