Pipes, Services & Dependency Injection

# **Parametrized Pipes**

# **Chaining Multiple Pipes**

# **Creating a Custom Pipe**

# **Creating a Filter Pipe**

# **Pure and Impure Pipes (or: How to "fix" the Filter Pipe)**

# **Understanding the "async" Pipe**

# **Services**

# **Dependency Injections**

# **Creating Data Service**

# **Understanding Hierarchical Injector**

# **Services for Cross Component Communication**

# **Injection Tokens**

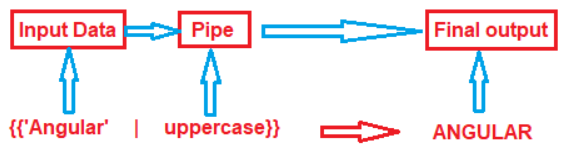
# Pipes

<https://angular.io/guide/pipes>

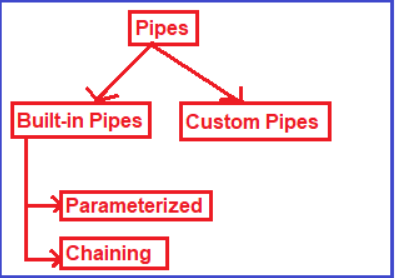
**Why to use Pipe?**

First get the data, then transform the data into some format, and finally, show the formatted data to the users.

**Syntax**



**Types of Pipes**



Use Case Apply Pipes on Employee Details

|  |  |  |
| --- | --- | --- |
|  | <h4 class="card-title">{{employee.name|titlecase}}</h4> |  |
|  | <p class="card-text">{{employee.dateOfJoining| date:'dd/MM/yy'}} |
|  | <p [ngClass]="{'text-success':employee.salary >10000,  'text-danger':employee.salary <= 10000 }" > {{employee.salary| currency:'USD':true}} |

Others

|  |  |
| --- | --- |
| **<b>**{{00.54565 | percent}}**</b>** |  |
| jsonval = {name: 'Alex', age: '25', address:{a1: 'Paris', a2: 'France'}};  **<b>**{{ jsonval | json }}**</b>** |  |

# Parametrized Pipes

|  |  |
| --- | --- |
| **<p>**Date Pipe : {{today | date}}**</p>**  **<p>**Full Date : {{today | date:'fullDate'}}**</p>**  **<p>**Mediate Date : {{today | date:'medium'}}</**p>**  **<p>**Short Date : {{today | date:'short'}}**</p>**  **<p>**Date (dd/MM/yyyy) : {{today | date:'dd/MM/yyyy'}}<**/p>**  **<p>**Time : {{today | date:'h:mm a z'}}**</p>**  **<p>**Medium Time : {{today | date:'mediumTime'}}</**p>** |  |
| <p>Currency USD in Symbol : {{salary | currency:'USD':true}}</p>  <p>Currency INR in Symbol : {{salary | currency:'INR':true}}</p>  <p>Currency USD in Code : {{salary | currency:'USD':false:'4.2-2'}}</p>  <p>Currency INR in Code : {{salary | currency:'INR':false:'1.3-3'}}</p> |  |

# Chaining Multiple Pipes

|  |  |
| --- | --- |
| import {Component } from '@angular/core';  @Component({  template: `  <h5>Chain Pipes</h5>  <p>Month is {{today | date | uppercase | slice:0:4}}  `,  })  export class ChainPipeComponent {  today = new Date();  } |  |
| The chained hero's birthday is {{ birthday | [date](https://angular.io/api/common/DatePipe) | [uppercase](https://angular.io/api/common/UpperCasePipe)}}  The chained hero's birthday is {{ birthday | [date](https://angular.io/api/common/DatePipe):'fullDate' | uppercase}} |  |

# Creating a Custom Pipe

Use Case: Create a pipe to reverse the String

|  |  |  |
| --- | --- | --- |
| 1. | Ng g p ReverseStringPipePipe | import { Pipe, PipeTransform } from '@angular/core';  @Pipe({  name: 'reverseStringPipe'  })  export class ReverseStringPipePipe implements PipeTransform {  transform(value: unknown, ...args: unknown[]): unknown {  return null;  }  } |
| 2 | Write logic to reverse the string in ts file | import { Pipe, PipeTransform } from '@angular/core';  @Pipe({  name: 'reverseStringPipe'  })  export class ReverseStringPipePipe implements PipeTransform {  transform(str: string ): string {  var splitString = str.split("");  var reverseArray = splitString.reverse();  var joinArray = reverseArray.join("");  return joinArray;  }  } |
| 3. | <td [ngClass]="{'one':true, 'two':false, 'four':true}">{{ dept.dname | reverseStringPipe}}</td> |  |

Use Case: Add Mr/Ms as prefix

export class MyTitlePipe implements PipeTransform {

transform(name: string, gender: string): string {

if (gender.toLowerCase() == "male")

return "Mr. " + name;

else

return "Miss. " + name;}

# Creating a Filter Pipe

Use Case: Filter Employees by name using Pipes

**Prerequisite:**

1. **Employee list**
2. **Employee class definition in model folder**

**Step 0: model/employee.model.ts**

export class Employee{

id : number;

name: string;

phoneNumber:number;

dateOfJoining:Date;

did:number;

photo?:string;

isActive:boolean;

salary:number;

}

**Step 1: Create Search Pipe**

D:\Angular\Angular2020\code\BookProj2020>ng g p searchByEmpNameFilter

**Step 2: Create Search div in list-employee.html**

<div class="container-fluid">

<h1>EmployeeDetails</h1>

<div class="form-group">

Search By Name: <input type="text" class="form-control" placeholder="Search by Name" [(ngModel)]="searchTerm" />

</div>

<table class="table table-dark">

<td border="2" \*ngFor="let employee of employees | empNameFilter:searchTerm" >

<div clasol-sms="card" style="width:200px">

<img class="card-img-top" src="{{employee.photo}}" width="50" alt="Card image" style="width:100%">

<div class="card-body">

<h4 class="card-title">{{employee.name|titlecase}}</h4>

<p class="card-text">{{employee.did}}</p>

<p class="card-text">{{employee.dateOfJoining| date:'dd/MM/yy'}}

<p class="card-text">{{employee.phoneNumber}}</p>

<p [ngClass]="{'text-success':employee.salary >10000,

'text-danger':employee.salary <= 10000

}" > {{employee.salary| currency:'USD':true}}

</p>

<p [ngStyle] ="{'background-color':employee.isActive === false ? 'red' : 'green'}" class="card-text">{{employee.isActive}}</p>

<button (click)="callGreetParent()"> Greet</button>

</div>

</div>

</td>

</table>

</div>

**//Define variable searchTerm=” ”**

**Step 3: Write the Search pipe.**

import { Pipe, PipeTransform } from '@angular/core';

import { Employee } from './model/employee.model';

@Pipe({

name: 'empNameFilter'

})

export class SearchByEmpNameFilterPipe implements PipeTransform {

transform(employees: Employee[],searchTerm: string): unknown {

if(!employees || !searchTerm){ //Handles Truthy condition

return employees;

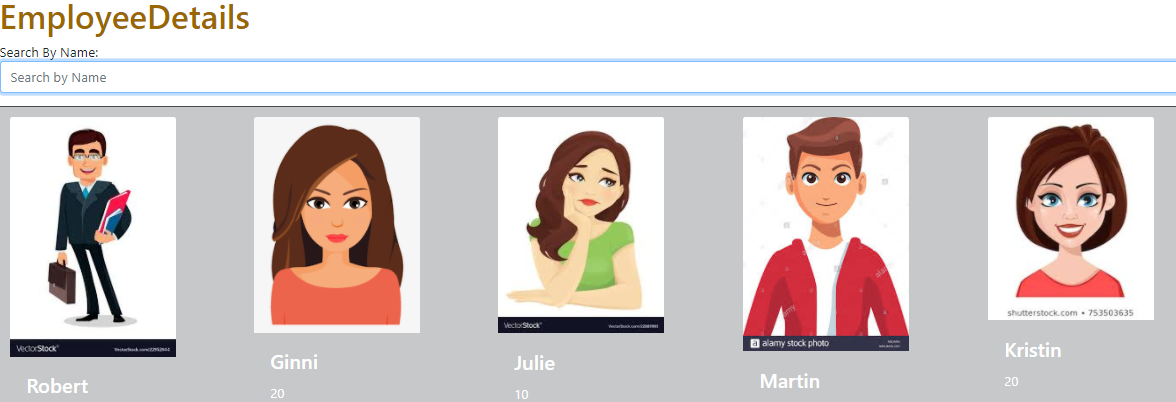
}

return employees.filter(employee=>employee.name.toLowerCase().indexOf(searchTerm.toLowerCase()) !== -1);

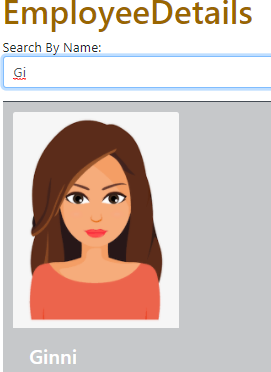
}

**Step 4: Test**

**Before filter**

****

**After Filter**

****

# Pure and Impure Pipes (or: How to "fix" the Filter Pipe)

|  |  |
| --- | --- |
| **Pure Pipe** <https://www.youtube.com/watch?v=G7WaP-yoiQg> | **Impure Pipe** |
| Executed only when a pure change is detected. Pure Change is either a change to a primitive input value(**number, string, Boolean**) or changed object reference (**Array, Date, Object**) | Impure Pipe is process at every change (irrespective of only value or reference change) even when source data does not change. Eg: Mouse Over is not pure change.  You make the filterpipe as impure by setting pure:false in filter pipe. |
| By default all pipes are pure.  **Change in Value: (**Change in firstname)  ***List.component.html***  <div class="form-group">  <button (click)="changeName()" class='btn btn-warning' >Update Name </button> </div>  <h4 class="card-title">{{employee.name|uppercase}}</h4>  ***List-component.ts***  //- ------------Pure/Impure Pipe----------changeName(){  return this.employees[0].name='Jim';}  Note: Here the on click of update button will change the name of first employee change and pipe will detect as value of change is primitive type here.  **Change in reference:**  Create a new array copy the existing array to it. Change the first name and assign the newarray value to old array.  changeName(){  //return this.employees[0].name='Raunak';  const newEmployeeArray: Employee []= Object.assign([], this.employees) ;  newEmployeeArray[0].name='Jimmy';  this.employees=newEmployeeArray;  }  Note: Here the change in Array detected by pipe and reflected to pipe. Now Reference point of array is different than original array. | Scenario: The currently names in array are Robert, Ginni, Julie, Richard and Kristin.  **With Pure:**  With reference to **Change in reference, w**hen you type j , it will display Julie and when you press “Update name” it will update name to ‘Jimmy’ and list will automatically populate as result of execution of Filter pipe which is pure in nature as of now.  But this not work .  With reference to **Change in Value: (** this.employees[0].name='Jim';).  When you j in text box, it wills show Julie . click on update name which will update ‘Ronak to Jim but will not reflect in list in context to Filter Pipe.    Solution: To make it happen, change pure=false.    So, to entertain all types of changes we use impure change. |

# Understanding the "async" Pipe

The AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes.

@Component({

selector: 'async-observable-pipe',

template: `<div><code>observable|async</code>:

Time: {{ time | async }}</div>`

})

export class AsyncObservablePipeComponent {

time = new Observable(observer =>

setInterval(() => observer.next(new Date().toString()), 2000)

);

}

# Services

##### What are Angular Services?

A service is used when a **common functionality needs** to be provided to various modules. Services allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components.

* It follows
  + DRY – Do not repeat itself.
  + Single responsibility Principles

##### Why Service?

* Shared Data
* Implement application logic
* External Interaction

##### Advantages:

* Services are easier to test.
* Services are easier to Debug.
* You can reuse the service.

Naming Conventions –

.service.ts

# Dependency Injections

Dependency injection (DI), is an important application design pattern in which a class asks for dependencies from external sources rather than creating them itself. Angular comes with its own dependency injection framework for resolving dependencies( services or objects that a class needs to perform its function).So you can have your services depend on other services throughout your application.

<https://codecraft.tv/courses/angular/dependency-injection-and-providers/overview/>

The DI framework in Angular consists of 4 concepts working together:

**Token**

This uniquely identifies something that we want injected. A dependency of our code.

**Dependency**

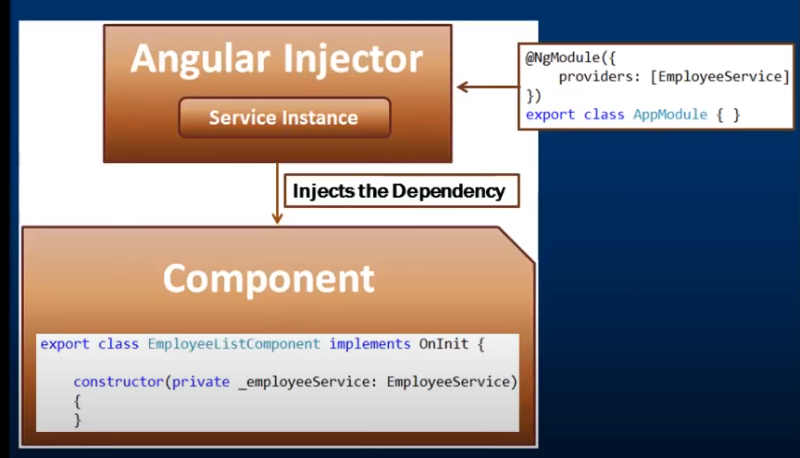
The actual code we want injected.

**Provider**

This is a map between a token and a list of dependencies.

**Injector**

This is a function which when passed a token returns a dependency (or list of dependencies).



This providers can be specified at module level or component level.

# Creating Data Service

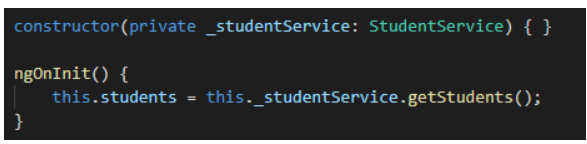
Use Case : Create user component and display details using UserService

|  |  |  |
| --- | --- | --- |
| 1 | Create Service – User  D:\Angular\Angular2020\code\BookProj2020>ng g s User | import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root'  })  export class UserService {  getUsers(): any[] {  return [  { uId: 'user1',FirstName:'Preety’,LastName: 'Tiwari',Password: 'pass1' }, { uId: 'user2',FirstName:'Sushmita',LastName: 'Sen',Password: 'pass2'},  { uId: 'user3',FirstName:'Priya',LastName:'Sharma',Password: 'pass3'},  ]; } |
| **Note:** The @Injectable() decorator in angular is used to inject other dependencies into the service. At the moment our service does not have any other dependencies, so, youcan remove the @Injectable() decorator and the service should works. However, the Angular Team recommends to always use @Injectable() decorator to ensures consistency. | | |
| 2 | Register userService with angular.module.ts | import {UserService} from './user.service';  imports: [  BrowserModule,  FormsModule ,  AppRoutingModule  ],  providers: [UserService],  bootstrap: [AppComponent] |
| 3 | Use the UserService in User component | import { Component, OnInit } from '@angular/core';  import {UserService} from '../user.service';  @Component({  selector: 'app-user',  templateUrl: './user.component.html',  styleUrls: ['./user.component.css'],  })export class UserComponent implements OnInit {  users: any[];  constructor(private \_userService: UserService) {  this.users = this.\_userService.getUsers();  } ngOnInit(): void { } } |
| 4 | Update User html | <p>Service Demonstration</p>  <div class="container">  <table class="table table-bordered">  <thead>  <tr>  <th>ID</th>  <th>First Name</th>  <th>Last Name</th>  <th>Password</th>    </tr>  </thead>  <tbody>  <tr \*ngFor='let user of users'>  <td>{{user.uId}}</td>  <td>{{user.FirstName}}</td>  <td>{{user.LastName}}</td>  <td>{{user.Password}}</td>    </tr>  </tbody>  </table>  </div> |
| 5 | Test |  |

##### Difference between constructor and ngOnInit

Whenever you create an instance of a class, the class constructor is automatically called. Like other programming languages, the class constructor in angular is also used to initialize the members of the class and it’s sub classes.

The ngOnInit is a life cycle hook method provided by Angular which is called after the constructor and is generally used to perform tasks related to Angular bindings. For example, ngOnInit is the right place to call a service method to fetch data from a remote server. We can also do the same using a class constructor, but the general rule of thumb is, tasks that are time consuming should use ngOnInit instead of the constructor. As fetching data from a remote server is time consuming, the better place for calling the service method is ngOnInit.



# Understanding Hierarchical Injector

<https://www.tektutorialshub.com/angular/angular-hierarchical-dependency-injection/>

## **What is Angular Injector**

The Angular Injector is responsible instantiating the dependency and injecting into the component or service. The Injector looks for the dependency in the [Angular Providers](https://www.tektutorialshub.com/angular/angular-providers/) using the token. The Angular Providers array returns the Provider, which contains the information about how to create the instance of the dependency. The Injector creates the instance and injects it into [Component](https://www.tektutorialshub.com/angular/angular-components/) or [service](https://www.tektutorialshub.com/angular/angular-services/).

## **When Injector is created?**

The Angular creates an injector when the application root module (named as AppModule) is bootstrapped. This injector is called as root injector. The root injector has the application wide scope. It acts as parent to all injectors.

Angular root module loads the AppComponent. We call this as root component. The AppComponent gets its own injector. This injector created as the child of the root injector.

The Root Component contains all other components. Angular App will create child components under the Root Component. All these child component can have their own child components creating a tree of components. The Angular Injector is also created for all those components creating a Injector tree closely mimicking the component tree.

## **Injector Tree**

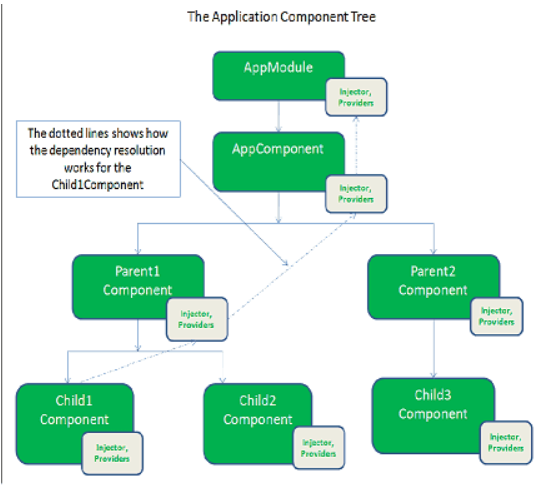
The Angular creates the [Injector,](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) when the application root module (named as AppModule) is bootstrapped. This injector is called as root injector and acts as a parent to all other injectors. The root injector also gets its own copy of [Providers](https://www.tektutorialshub.com/angular/angular-providers/). It gets it from the Providers is metadata of @NgModule of AppModule.

The AppModule loads the AppComponent, which is the root component of our application. The AppComponent gets its own injector with a copy of [Providers](https://www.tektutorialshub.com/angular/angular-providers/) defined in Providers metadata of the AppComponent.

The Root Component acts as a parent to every component we create. Each of those components can contain child components creating a tree of components. The Injector is created for each of those component creating a tree of injector, which closely resembles the component tree. This is called a hierarchical pattern. The injectors also get their own copy of providers from the @component metadata.

The injector is destroyed when the associated component is destroyed.

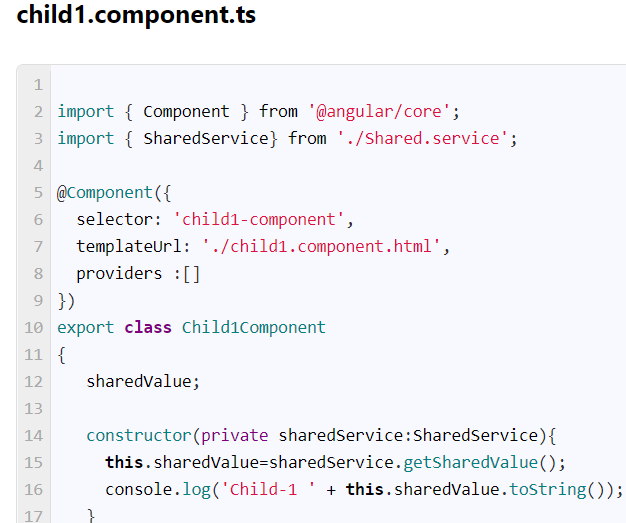
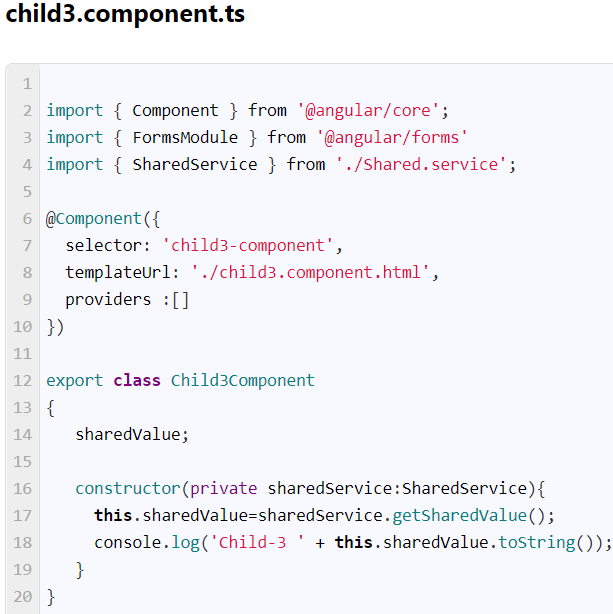
Consider following :



**Shared Service**



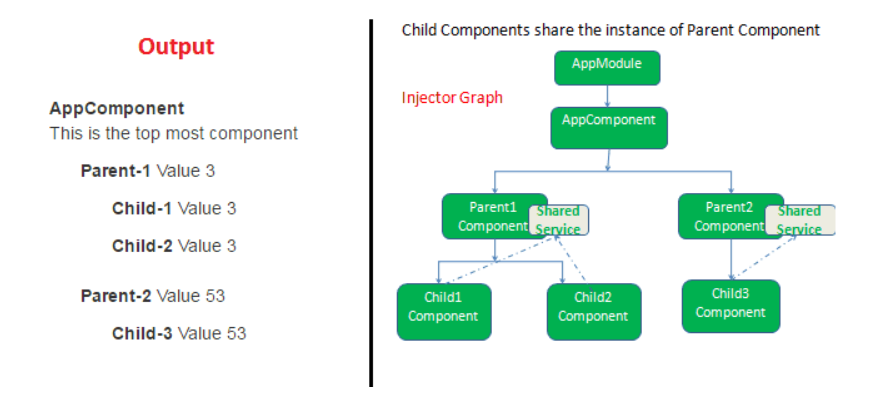
  

**How Dependency gets resolved?**

What happens if the child1Component requests a **Service**. The injector attached to Child1Component looks for the provider in the Providers collection registered with the Child1Component.

If it does not find the provider, it then passes the request to the injector instance of the parent1Component as shown by the dotted arrow in the image above. If the provider is found, the request is returned with the instance of the dependency else the request is passed on the injector of the AppComponent. This process continues until it reaches the top-level injector.

The output and the injection graph

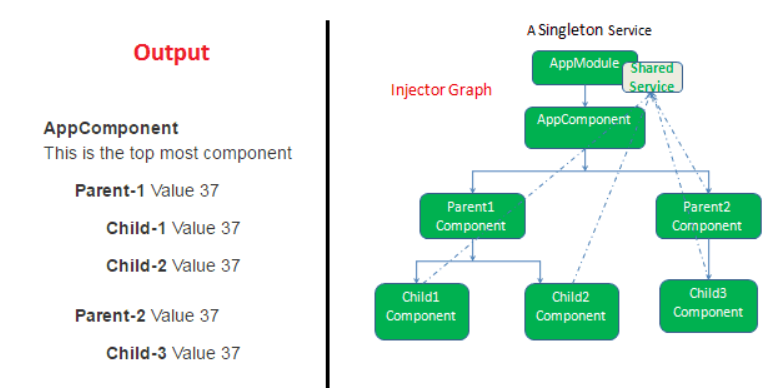


## Angular Singleton Service

You can create a Singleton Service by moving the Provider registration to the root injector i.e the root module (AppModule in the example).

Remove the provider registration from all the components and add it to the Providers array in the AppModule

Now the entire application will share the same instance of SharedService as shown below making the Service as Singleton.



# Services Cross Component Communication

# Injection Tokens

Injection tokens are a feature of Angular that allows the injection of values that don't have a runtime representation.