PIPES

When we want to develop any application, we always start the application with a simple task: retrieve data, transform data, and then display the data in front of the user through the user interface. Retrieval of data from any type of data source totally depends on data service providers like web services, Web API, etc. So, once data arrives, we can push those raw data values directly to our user interface for viewing by the user. But sometimes, this is not exactly what happens. For example, in most use cases, users prefer to see a date in a simple format like 15/02/2017 rather than the raw string format Wed Feb 15 2017 00:00:00 GMT-0700 (Pacific Daylight Time). So, it is clear from the above example that some values require editing before being viewed in the user interface. Also, that same type of transformation might be required by us in many different user interfaces. So, in this scenario, we think about some style-type properties that we can create centrally and apply whenever we require it. So, for this purpose, the Angular framework introduced Angular pipes, a definite way to write display–value transformations that we can declare in our HTML.

In Angular 8.0, pipes are typescript classes that implement a single function interface, accept an input value with an optional parameter array, and return a transformed value. To perform the value transformation, we can implement any type of business logic as per our requirements. That pipe can be used in any UI to transform that particular type of data as per the desired result.

Basic Concept of Pipes

Basically, pipes provide a sophisticated and handsome way to perform the tasks within the templates. Pipes make our code clean and structured. In Angular, Pipes accept values from the DOM elements and then return a value according to the business logic implemented within the pipes. So, pipes are one of the great features through which can transform our data into the UI and display. But we need to understand one thing very clearly, pipes do not automatically update our model value. It basically performs the transformation of data and returns to the component. If we need to update the model data after transformation through pipes, then we need to update our model data manually. Expect these, we can use pipes for any of the below reasons.

* If we want to retrieve the position of the elements
* If we want to track the user inputs in input-type elements
* If we want to restrict the user to input some value in the input controls

Why is Pipe Required?

In any application, Pipe can be used for the following reasons.

* We can display only some filtered elements from an array.
* We can modify or format the value.
* We can use them as a function.
* We can do all of the above combined.

Types of Pipe

In Angular 8, we can categorize the pipes into two types i.e. Pure Pipes and Impure Pipes.

Pure Pipes

Pure pipes in angular are those pipes that always accept some arguments as input values and return some value as the output according to the input values. Some examples of pure pipes are – decimal pipes, date pipes, etc. When we use these types of pipes in Angular, we provide input value with related configuration value to the pipes and pipes return us the formatted value as an output.

Impure Pipes

Impure pipes in angular are those pipes that also accept the input values, but return the different types of the value set according to the state of the input value. An example of the impure pipes is async pipes. These pipes always store the internal state and return different types of values as the output according to the internal state and logic.

Filter vs Pipe

The concept of the filter is mainly used in the Angular 1. x version. From Angular 2 onwards, Google deprecated the Filter concept and introduced a new concept called Pipe. Now, as per the functionality, filter, and pipes, both are working the same. But still, there are some differences as below.

1. Filters act just like helpers similar to functions where we can pass input and other parameters and it will return us a formatted output value. In the case of a pipe, it works as an operator. It also accepts input value and modifies that value to return the desired output.
2. Filters can not handle directly any async type operations. We need to set those values manually. But Pipe can handle async operations on its own. For these types of operations, we need to use async-type pipes.

Basic Pipes

Most of the pipes provided by Angular 8 will be familiar to us if we already worked in the previous Angular version. Actually, pipes do not provide any new features in Angular 8. In Angular 8, we can use logic in the template. We can define any function within the pipe class to implement any special type of data conversion or business login and then execute that particular function from the HTML template to obtain the desired result. The syntax of the Pipe in the HTML template begins with the input value and then follows the pipe symbol (|) and then needs to provide the pipe name. The parameters of that pipe can be sent separately by a colon (:). The order of execution of a pipe is right to left. In General, Pipe is working within the HTML only. The most commonly used built-in pipes are.

* Currency
* Date
* Uppercase
* Lowercase
* JSON
* Decimal
* Percent
* Async

**Syntax of Pipes**

{{ myValue | myPipe:param1:param2 | mySecondPipe:param1 }}

Markup

Copy

Custom Pipes

Now we can define custom pipes in Angular 8 as per our custom business logic. To configure custom pipes, we need to use a pipe object. For this, we need to define a custom pipe with the @Pipe decorator and use it by adding a pipe property to the @View decorator with the pipe class name. We use the transform method to do any logic necessary to convert the value that is being passed in as an input value. We can get a hold of the arguments array as the second parameter and pass in as many as we like from the template. The @Pipe decorator contains the below two properties.

1. **name:**It contains the name of the pipe which needs to be used in the DOM elements.
2. **pure:**It accepts the Boolean value. It identifies the pipe as a pure or impure pipe. The default value of the pure property is true i.e. it always considers the custom pipe as a pure-type pipe. It means Angular Framework will execute a pure pipe only when it detects a pure change in the input value. Pure change data can be primitive (means data contains only single values) or non-primitive (means data contains such data type which accepts a group of values). If we need to make the pipe as impure then we need to pass false as the value against this property. In the case of the Impure pipe, the Angular framework will execute the pipes on every component change detection cycle. In this case, the pipe is often called as often as every keystroke or mouse-move.

When we define any pipe class using the @Pipe decorator, we need to implement the PipeTransforms interface which is mainly used to accept the input values (optional values) and return the transform values to the DOM.

What is Viewchild?

Basically, Viewchild is one of the new features introduced in the Angular framework. Angular is basically dependent on component-based architecture. So when we try to develop any web page or UI, it is most obvious that that page or UI must be a component that basically contains multiple numbers of different functional components as a child component. So in simple words, it is basically a parent component – child component-based architecture. In this scenario, there are some situations that occur when a parent component needs to interact with the child component. We can establish connections between parent and child components in many ways. One of the ways is ViewChild decorator. ViewChild decorator can be used if we need to access the instance of a child component or a directive from the parent component class. So when the need to invoke any method of the child component from the parent component, it can inject the child component as a Viewchild within the parent component. In cases where you’d want to access multiple children, you’d use ViewChildren instead.

To implement ViewChild, we need to use the @ViewChild decorator in the parent component. The @ViewChild decorator provides access to the class of child components from the parent component. The @ViewChild decorator is basically a function or method that accepts the name of a component class as its input and finds its selector in the template of the related component to bind to. The basic syntax of the ViewChild decorator in Angular 8 is as below.

@ViewChild(ChildComponent, { static: true })

\_calculator: ChildComponent;

TypeScript

Copy

So, as per the above example, the ViewChild decorator contains the following metadata.

1. **selector:** It contains the directive or component name that needs to be used as a view child.
2. **static:**It accepts the Boolean value. We need to pass true as a value if we want to resolve the query result before change detection occurs. So, when we pass the value as true, then the Angular framework tries to locate that element at the time of component initialization i.e. in the ngOnInit method. If we pass the value as false, then Angular will try to find the element after the initializing of the view i.e. in the ngAfterViewInit method.

Demo 1. Basic Pipes Demo

In this example, we will demonstrate how to use the inbuilt pipes in Angular 8.

**app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls : ['./custom.css']

})

export class AppComponent implements OnInit {

public todayDate: Date;

public amount: number;

public message: string;

constructor() { }

ngOnInit(): void {

this.todayDate = new Date();

this.amount = 100;

this.message = "Angular 8.0 is a Component Based Framework";

}

}

TypeScript

Copy

**app.component.html**

<p>

<h1>Demonstrate of Pipe in Angular 8</h1>

<h2>Date Pipes</h2>

Full Date: {{todayDate}}<br />

Short Date: {{todayDate | date:'shortDate'}}<br />

Medium Date: {{todayDate | date:'mediumDate'}}<br />

Full Date: {{todayDate | date:'fullDate'}}<br />

Time: {{todayDate | date:'HH:MM'}}<br />

Time: {{todayDate | date:'hh:mm:ss a'}}<br />

Time: {{todayDate | date:'hh:mm:ss p'}}<br />

<h2>Number Pipes</h2>

No Formatting: {{amount}}<br />

2 Decimal Place: {{amount | number:'2.2-2'}}

<h2>Currency Pipes</h2>

No Formatting: {{amount}}<br />

USD Dollar($): {{amount | currency:'USD':true}}<br />

USD Dollar: {{amount | currency:'USD':false}}<br />

INR(): {{amount | currency:'INR':true}}<br />

INR: {{amount | currency:'INR':false}}<br />

<h2>String Related Pipes</h2>

Actual Message: {{message}}<br />

Lower Case: {{message | lowercase}}<br />

Upper Case: {{message | uppercase}}<br />

<h2> Percentage Pipes</h2>

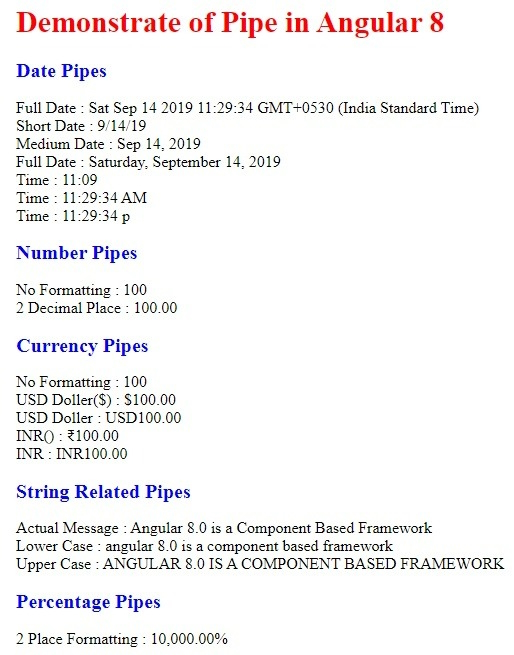
2 Place Formatting: {{amount | percent :'.2'}}<br /><br />

</p>

Markup

Copy

Now check the output into the browse.



Demo 2. Custom pipe proper case

Now, it's time to define some custom pipes as per our requirements. In the above example, we see that in general, the Angular 8 framework provides us with UpperCase and LowerCase pipes against any string type value as in-build pipes. But, it does not provide any proper case-type pipes. So, let's define a pipe that will transfer any string value as a proper case and return the result. For that purpose, add the below typescript class files.

**propercase.pipe.ts**

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

@Pipe({

name: 'propercase'

})

export class ProperCasePipe implements PipeTransform {

transform(value: string, reverse: boolean): string {

if (typeof (value) == 'string') {

let intermediate = reverse == false ? value.toUpperCase() : value.toLowerCase();

return (reverse == false ? intermediate[0].toLowerCase() :

intermediate[0].toUpperCase()) + intermediate.substr(1);

}

else {

return value;

}

}

}

TypeScript

Copy

Now, include the proper case pipe in the AppModule as below.

import { BrowserModule } from '@angular/platform-browser';

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

import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { ColorChangeDirective } from './app.directive';

import { ProperCasePipe } from './propercase.pipe';

@NgModule({

declarations: [

AppComponent,

ColorChangeDirective,

ProperCasePipe

],

imports: [

BrowserModule,

FormsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

TypeScript

Copy

**app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls : ['./custom.css']

})

export class AppComponent implements OnInit {

public message: string;

constructor() { }

ngOnInit(): void {

this.message = "This is a Custom Pipe";

}

}

TypeScript

Copy

**app.component.html**

<p>

<p class="form-horizontal">

<h2 class="aligncenter">Custom Pipes - Proper Case</h2><br />

<p class="row">

<p class="col-xs-12 col-sm-2 col-md-2">

<span>Enter Text</span>

</p>

<p class="col-xs-12 col-sm-4 col-md-4">

<input type="text" id="txtFName" placeholder="Enter Text" [(ngModel)]="message" />

</p>

</p>

<p class="row">

<p class="col-xs-12 col-sm-2 col-md-2">

<span>Result in Proper Case</span>

</p>

<p class="col-xs-12 col-sm-4 col-md-4">

<span>{{ message | propercase }}</span>

</p>

</p>

</p>

</p>

Markup

Copy

Now run the output into the browser.



Demo 3. View child

Now, in this demo, we will discuss how to use ViewChild decorator within a component. For that purpose, first, need to create a child component that will be used as a ViewChild in the parent or root component.

**child.component.ts**

import { Component, OnInit, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'child',

templateUrl: 'child.component.html'

})

export class ChildComponent implements OnInit {

private firstNumber: number = 0;

private secondNumber: number = 0;

private result: number = 0;

@Output() private addNumber: EventEmitter<number> = new EventEmitter<number>();

@Output() private subtractNumber: EventEmitter<number> = new EventEmitter<number>();

@Output() private multiplyNumber: EventEmitter<number> = new EventEmitter<number>();

@Output() private pideNumber: EventEmitter<number> = new EventEmitter<number>();

constructor() { }

ngOnInit(): void { }

private add(): void {

this.result = this.firstNumber + this.secondNumber;

this.addNumber.emit(this.result);

}

private subtract(): void {

this.result = this.firstNumber - this.secondNumber;

this.subtractNumber.emit(this.result);

}

private multiply(): void {

this.result = this.firstNumber \* this.secondNumber;

this.multiplyNumber.emit(this.result);

}

private pide(): void {

this.result = this.firstNumber / this.secondNumber;

this.pideNumber.emit(this.result);

}

public clear(): void {

this.firstNumber = 0;

this.secondNumber = 0;

this.result = 0;

}

}

TypeScript

Copy

**child.component.html**

<p class="ibox-content">

<p class="row">

<p class="col-md-4">

Enter First Number

</p>

<p class="col-md-8">

<input type="number" [(ngModel)]="firstNumber" />

</p>

</p>

<p class="row">

<p class="col-md-4">

Enter Second Number

</p>

<p class="col-md-8">

<input type="number" [(ngModel)]="secondNumber" />

</p>

</p>

<p class="row">

<p class="col-md-4">

</p>

<p class="col-md-8">

<input type="button" value="+" (click)="add()" />

<input type="button" value="-" (click)="subtract()" />

<input type="button" value="X" (click)="multiply()" />

<input type="button" value="/" (click)="pide()" />

</p>

</p>

</p>

Markup

Copy

Now, include this child component in the app.module.ts file.

import { BrowserModule } from '@angular/platform-browser';

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

import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { ChildComponent } from './child.component';

@NgModule({

declarations: [

AppComponent,

ChildComponent

],

imports: [

BrowserModule,

FormsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

TypeScript

Copy

Now, use the above child component in the root component as below.

**app.component.ts**

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

import { ChildComponent } from './child.component';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls : ['./custom.css']

})

export class AppComponent {

private result: string = '';

@ViewChild(ChildComponent, { static:true }) private \_calculator: ChildComponent;

constructor() {

}

private add(value: number): void {

this.result = 'Result of Addition ' + value;

}

private subtract(value: number): void {

this.result = 'Result of Subtraction ' + value;

}

private multiply(value: number): void {

this.result = 'Result of Multiply ' + value;

}

private pide(value: number): void {

this.result = 'Result of pision ' + value;

}

private reset(): void {

this.result = '';

this.\_calculator.clear();

}

}

TypeScript

Copy

**app.component.html**

<h2>Demo of ViewChild</h2>

<p>

<child (addNumber)="add($event)" (subtractNumber)="subtract($event)" (multiplyNumber)="multiply($event)"

(pideNumber)="pide($event)" #calculator></child>

</p>

<h3>Result</h3>

<span>{{result}}</span>

<br />

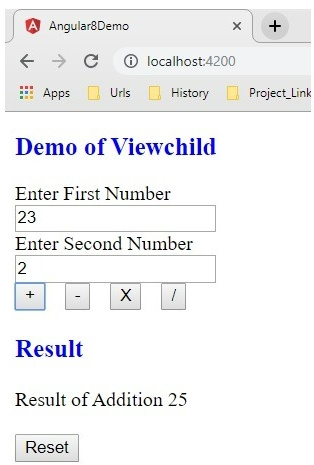
<br />

<input type="button" value="Reset" (click)="reset()" />

Markup

Copy

Now, check the output in the browser.



Conclusion