Practical No.5

Aim: Design webpage using *ngIf,*ngFor,*ngswitch and ngStyle Directives.

A. Objective:

Manipulate the structure of the DOM (Document Object Model) based on conditions and other logical statements.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering.*

C. Expected Skills to be developed based on competency:

Learner can get various skills such as understanding of basic Angular concepts, Familiarity with Type Script, Knowledge of Angular directives, Experience with Angular CLI.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(PRo)

Design a web page to display student grading system in tabular format with alternate color style using ngSwitch, ngStyle Directives.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Structural directives in Angular are a type of directive that allow you to modify the structure of the DOM based on certain conditions or states. They are used to add, remove, or update elements in the DOM based on the result of an expression or the state of the application.

There are three common structural directives in Angular:

1. ngIf: The ngIf directive is used to conditionally add or remove an element from the DOM based on a given expression. If the expression is true, the element is added to the DOM, and if it is false, the element is removed from the DOM.

2. ngFor: The ngFor directive is used to repeat a section of HTML code for each item in an array or collection. It can be used to loop through an array of objects, an array of strings, or any other iterable object.

3. ngSwitch: The ngSwitch directive is used to conditionally display content based on a given expression. It allows you to define a set of possible values and associate each value with a template to render.

Structural directives are identified by the prefix "ng" followed by the directive name. They are enclosed in square brackets and are typically used in combination with other directives and HTML tags to create dynamic and responsive user interfaces.

<u>*ngIf Directory Structure</u>

*ngIf is a structural directive in Angular that allows you to conditionally render or remove HTML elements based on an expression. The directive evaluates the expression passed to it and renders the element if the expression is truthy, otherwise it removes the element from the DOM.

In addition to *ngIf, there is also an *ngIf-else directive that allows you to specify an alternative template to render when the expression evaluates to false. The syntax for *ngIf-else is as follows:

```
<div *ngIf="condition; else elseBlock">
<!-- content to show when condition is true -->
</div>
<ng-template #elseBlock>
<!-- content to show when condition is false -->
</ng-template>
```

In this example, the **div** element will be rendered if the **condition** expression is truthy. If **condition** is falsy, the **div** element will be removed from the DOM and the **elseBlock** template will be rendered instead.

The **elseBlock** template is defined using the **ng-template** element with a template reference variable of **#elseBlock**. This allows us to refer to the template later in the *ngIf-else directive.

You can also use multiple *ngIf-else directives to conditionally render different templates based on different expressions. Here's an example:

```
<div *ngIf="condition1; else elseBlock1">
<!-- content to show when condition1 is true -->
</div>
<ng-template #elseBlock1>
<div *ngIf="condition2; else elseBlock2">
<!-- content to show when condition1 is false and condition2 is true -->
</div>
<ng-template #elseBlock2>
<!-- content to show when both condition1 and condition2 are false -->
</ng-template>
</ng-template>
```

In this example, we first check **condition1**. If it is true, we render the first **div** element. If it is false, we check **condition2**. If **condition2** is true, we render the second **div** element. If both **condition1** and **condition2** are false, we render the content in the second **ng-template**.

ngFor Directory

***ngFor** is a structural directive in Angular that iterates over a collection and creates a template for each item in the collection. It is commonly used to render a list of items or to repeat a section of a template.

The basic syntax of ***ngFor** is:

```
*ngFor="let item of collection"
```

where**item** is a variable that represents each item in the collection, and **collection** is an array or any iterable object.
Here are some examples of how ***ngFor** can be used in Angular:

1. Rendering a list of items

```
<ul>
<li *ngFor="let item of items">{{ item }}</li>
</ul>
```

items=['apple', 'banana', 'orange']

In this example, **items** is an array of stringsthat is declared in ts file, and ***ngFor** creates an**li** element for each item in the array.

2. Looping over an array of objects

```
<table>
<tr *ngFor="let user of users">
<td>{{ user.name }}</td>
<td>{{ user.age }}</td>
</tr>
</table>

users = [
   {
name: 'chintan',
age: 36
   },
   {
name: ‘mohit’,
age: 45
    },
    {
name: ‘hiral’,
age: 25
    },
  ]
```

In this example, **users** is an array of objects that is declared in ts file, and ***ngFor** creates a **tr** element for each object in the array. The **td** elements contain the **name** and **age** properties of each object.

3: Using index for conditional rendering

```
<div *ngFor="let item of items; index as i">
<div *ngIf="i % 2 == 0" class="even-item">{{item}}</div>
<div *ngIf="i % 2 != 0" class="odd-item">{{item}}</div>
</div>
```

In this example, we are using the "index as i" syntax to access the current index of the loop iteration and use it for conditional rendering. We are displaying each item on a new line, but alternating the background color based on whether the index is even or odd.

Overall, *ngFor with index is a powerful tool in Angular that allows developers to loop through collections and access the current index of the loop iteration for various use cases.

***ngSwitch**

The *ngSwitch directive is used in Angular to conditionally display content based on a specified expression. It's similar to a switch statement in programming languages. Here's an example of how to use *ngSwitch in an Angular template file:

```
<div [ngSwitch]="dayOfWeek">
<div *ngSwitchCase="'Monday'">It's Monday!</div>
<div *ngSwitchCase="'Tuesday'">It's Tuesday!</div>
<div *ngSwitchCase="'Wednesday'">It's Wednesday!</div>
<div *ngSwitchCase="'Thursday'">It's Thursday!</div>
<div *ngSwitchCase="'Friday'">It's Friday!</div>
<div *ngSwitchCase="'Saturday'">It's Saturday!</div>
<div *ngSwitchCase="'Sunday'">It's Sunday!</div>
<div *ngSwitchDefault>Invalid day of the week!</div>
</div>
```

In the above example, we have a div with the attribute **[ngSwitch]="dayOfWeek"**, where **dayOfWeek** is a property in the component class. Inside this div, we have several divs that use the ***ngSwitchCase** directive to display different content based on the value of **dayOfWeek**. The ***ngSwitchDefault** directive is used to specify the default case if none of the ***ngSwitchCase** expressions match.

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

@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
dayOfWeek = 'Monday';
}
```

In the component class, we have a property called **dayOfWeek** that is set to **'Monday'**. Based on this value, the template will display the div with the ***ngSwitchCase="'Monday'"** directive.

Practical related Exercises.

1. Design webpage to displaying different images based on a selected option.

app.component.html

```
<app-ng-switch></app-ng-switch>
```

ng-switch.component.html

```
<h1 align="center">ngSwitch Example</h1>
<div (change)="chooseProduct($event)">
   <select>
      <option value="">Choose Tool</option>
      <option value="vscode">Visual Studio Code</option>
      <option value="virtualbox">Virtual Box</option>
      <option value="androidstudio">Android Studio</option>
   </select>
</div>

<div [ngSwitch]="selectedtool" class="main">
   <div *ngSwitchCase = "'vscode'">
      <p style="margin-top: 20px;">this is VS Code logo</p> <br>
      <img src="assets/3.png" ></div>
   <div *ngSwitchCase = "'virtualbox'">
      <p style="margin-top: 20px;">this is virtualbox logo </p><br>
```

```
<img src="assets/1.jpg"></div>
    <div *ngSwitchCase = "'androidstudio'">
       <p style="margin-top: 20px;"> this is androidstudio logo</p> <br>
        <img src="assets/2.png"></div>

</div>
<div style="margin-top: 150px;margin-bottom: 150px;">
    <hr>
  </div>
```

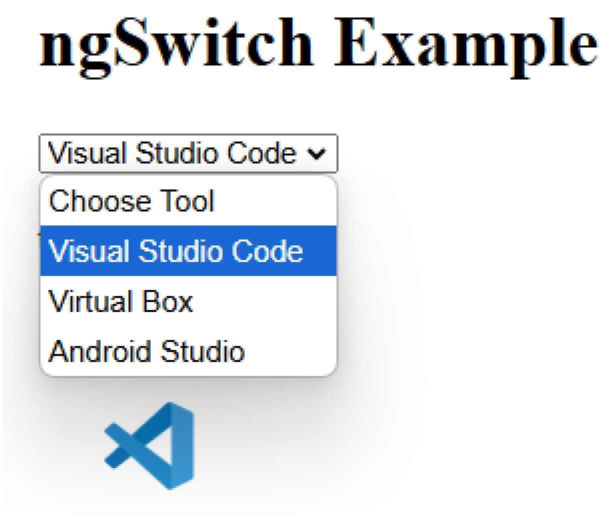
ng-switch.component.ts

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

@Component({
  selector: 'app-ng-switch',
  templateUrl: './ng-switch.component.html',
  styleUrls: ['./ng-switch.component.css']
})
export class NgSwitchComponent {
  selectedtool ='';

  chooseProduct($event : any){
    this.selectedtool=$event.target.value;
  }
}
```

Output :


ngSwitch Example

Practical No.6

Aim: Design component to perform following tasks

[1] To Add or Remove number of students using textbox and button controls and display it in tabular structure format.

[2] Give row level remove button option to student table and record should be deleted when click on it.

A. Objective:

The learner will help to manipulate data on web page based on event , conditions and generate desired outputs.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering.*

C. Expected Skills to be developed based on competency:

Learner should have knowledge of how to Dynamic rendering of lists, efficiently adding and removing elements on web page and Familiarity with template syntax.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(PRo)

Learner will be able to Add or Remove number of students using textbox and button controls and display it in tabular structure format along with give row level remove button option to student table and record should be deleted when click on it.

.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Template reference variable

Template reference variable is a variable that you can assign to a template element or component using the **#** symbol. This allows you to reference the element or component in your template code and in your component code.

Here's an example of using a template reference variable to reference an input element in a template:

```
<input type="text" #nameInput>

<button (click)="greet(nameInput.value)">Greet</button>
```

In this template, we're using the **#nameInput** syntax to create a template reference variable that we can use to reference the **input** element. We're also using an event binding to call a **greet** method on our component when a button is clicked.

In our component, we can access the value of the input element using the template reference variable:

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


@Component({

selector: 'app-greeter',

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

styleUrls: ['./greeter.component.css']
})
export class GreeterComponent {
greet(name: string) {
alert(`Hello, ${name}!`);
  }
}
```

In this component, we've defined a **greet** method that takes a **name** argument and displays an alert message with the name. When the button in the template is clicked, the **greet** method is called with the value of the **nameInput** template reference variable as the argument.

PUSH method

The **push** method in Angular is used to add elements to an array. It's a built-in method of the JavaScript **Array** object, and it can be used in Angular components to manipulate arrays and update the view.

Here's an example of using the **push** method in an Angular component:

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

@Component({
selector: 'app-product-list',
templateUrl: './product-list.component.html',
styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
products: string[] = [];

addProduct(productName: string) {
this.products.push(productName);
  }
}
```

In this example, we've defined a **products** array in our component and initialized it to an empty array. We've also defined an **addProduct** method that takes a **productName** argument and adds it to the **products** array using the **push** method.

Now, in our component's template, we can use **ngFor** to iterate over the **products** array and display each product in a list:

```
<ul>
<li *ngFor="let product of products">{{ product }}</li>
</ul>

<input type="text" #productName>
<button (click)="addProduct(productName.value)">Add Product</button>
```

In this template, we're using **ngFor** to iterate over the **products** array and display each product as a list item. We're also using an input and a button to capture new product names and call the **addProduct** method when the button is clicked.

When the **addProduct** method is called, it uses the **push** method to add the new product to the **products** array. Because the view is bound to the **products** array using **ngFor**, the view is automatically updated to show the new product in the list.

Splice method

The splice method in Angular is used to add or remove elements from an array at a specified index. It's a built-in method of the JavaScript Array object, and it can be used in Angular components to manipulate arrays and update the view.

Here's an example of using the splice method in an Angular component:

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

@Component({
selector: 'app-product-list',
templateUrl: './product-list.component.html',
styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
products: string[] = ['apple', 'banana', 'orange'];

removeProduct(index: number) {
this.products.splice(index, 1);
  }
}
```

In this example, we've defined a **products** array in our component and initialized it with three products. We've also defined a **removeProduct** method that takes an **index** argument and removes the product at that index from the **products** array using the **splice** method.

Now, in our component's template, we can use **ngFor** to iterate over the **products** array and display each product in a list. We can also use a button to call the **removeProduct** method with the index of the product we want to remove:

```
<ul>
<li *ngFor="let product of products; let i = index">{{ product }}
<button (click)="removeProduct(i)">Remove</button>
</li>
</ul>
```

In this template, we're using **ngFor** to iterate over the **products** array and display each product as a list item. We're also using a button to call the **removeProduct** method with the index of the product we want to remove.

When the **removeProduct** method is called, it uses the **splice** method to remove the product at the specified index from the **products** array. Because the view is bound to the **products** array using **ngFor**, the view is automatically updated to remove the product from the list.

app.component.html

```
<app-trv></app-trv>
```

trv.component.html

```
<div class="container">
    <h2>Practical 6</h2>
        <div class="add-section">
      <input
        type="text"
        #studentInput
        placeholder="Enter student name"
        (click)="addStudent(studentInput)">
      <button (click)="addStudent(studentInput)">Add Student</button>
    </div>
    <table *ngIf="students.length > 0" class="student-table">
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Action</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let student of students">
          <td>{{student.id}}</td>
          <td>{{student.name}}</td>
          <td>
            <button class="remove-btn" (click)="removeStudent(student.id)">
              Remove
            </button>
          </td>
        </tr>
      </tbody>
    </table>
    <p *ngIf="students.length === 0">No students added yet.</p>
  </div>
```

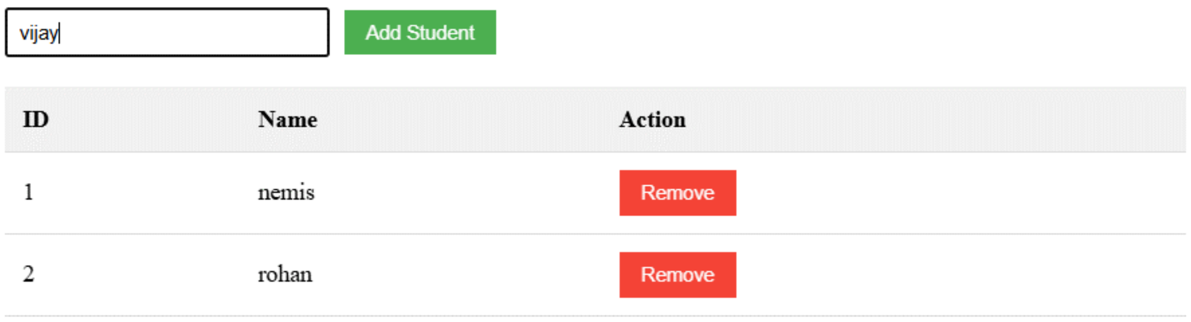
trv.component.css

```
.container {
   max-width: 800px;
   margin: 20px auto;
   padding: 20px;
 }
 .add-section {
   margin-bottom: 20px;
 }
 input {
   padding: 8px;
   margin-right: 10px;
   width: 200px;
 }
 button {
   padding: 8px 15px;
   background-color: #4CAF50;
   color: white;
   border: none;
   cursor: pointer;
 }
 button:hover {
   background-color: #45a049;
 }
 .student-table {
   width: 100%;
   border-collapse: collapse;
   margin-top: 20px;
 }
 .student-table th,
 .student-table td {
   padding: 12px;
   text-align: left;
   border-bottom: 1px solid #ddd;
 }
 .student-table th {
   background-color: #f2f2f2;
 }
 .remove-btn {
   background-color: #f44336;
 }
 .remove-btn:hover {
   background-color: #da190b;
 }
```

trv.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-trv',
  templateUrl: './trv.component.html',
  styleUrls: ['./trv.component.css']
})
export class TrvComponent {
  students: { id: number; name: string }[] = [];
  nextId: number = 1;
  addStudent(studentInput: HTMLInputElement) {
    const name = studentInput.value;
    if (name) {
      this.students.push({
        id: this.nextId++,
        name: name
      });
      studentInput.value = '';
    }
  }
  removeStudent(id: number) {
    const index = this.students.findIndex(student => student.id === id);
    if (index !== -1) {
      this.students.splice(index, 1);
    }
  }
}
```

Output :



ID	Name	Action
1	nemis	Remove
2	rohan	Remove

Practical No 7

Aim: Create a component to display a products list from array. the product component should display a product Id, name, purchase date, price, and image for the product and search using various pipes.

A. Objective:

Simplify the process of displaying data in a format that is appropriate for the user by using custom pipe and you can reduce the amount of code needed in your template , improve the performance and usability of your application by using filter pipe.

.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis**: Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering.*

C. Expected Skills to be developed based on competency:

Learner can develop skills in data filtering, working with arrays and objects, applying filter conditions.These skills are valuable for building robust and efficient Angular applications.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(PRo)

Learner will be able todisplay a products list from array and search required product using filter, custom pipes

.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Pipes are used to format and transform data in a template. Pipes take in an input value, transform it, and then return the transformed value. There are two types of pipes available in Angular:

1. Built-in pipes - These are provided by Angular.
   Here is a table listing the built-in pipes in Angular along with a brief description of each pipe:

Pipe	Description
DatePipe	Used to format a date according to a specified format string and locale.
DecimalPipe	Used to format a number with a fixed number of digits before and after the decimal point.
LowerCasePipe	Used to convert a string to lowercase.
PercentPipe	Used to format a number as a percentage.
SlicePipe	Used to create a new array or string containing a subset of the elements of the input array or string.
TitleCasePipe	Used to convert a string to title case, where the first letter of each word is capitalized.
UpperCasePipe	Used to convert a string to uppercase.

2. Custom pipes - These are created by developers and can be used to transform data in a specific way.

Custom pipes are useful when you need to transform data in a specific way that is not provided by the built-in pipes. In addition to custom pipes, Angular also provides filter pipes. Filter pipes are used to filter data based on a specific criteria.

Here is an example of using the built-in UpperCasePipe in Angular:

```
<!-- app.component.html -->
<p>{{ 'hello world' | uppercase }}</p>
```

In this example, the string 'hello world' is passed to the uppercase pipe, which transforms it to 'HELLO WORLD'. The transformed string is then displayed in the template using string interpolation.

Here is an example of creating a custom pipe that appends an exclamation mark to a string:

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { AddExclamationPipe } from './add-exclamation.pipe';

@NgModule({
declarations: [
    AppComponent,
    AddExclamationPipe
  ],
imports: [
    BrowserModule
  ],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

// add-exclamation.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
```

```
name: 'addExclamation'
})
export class AddExclamationPipe implements PipeTransform {
transform(value: string): string {
return value + '!';
  }
}

<!-- app.component.html -->
<p>{{ 'hello world' | addExclamation }}</p>
```

In this example, a custom pipe called AddExclamationPipe is created using the Pipe decorator. The pipe takes in a string value, appends an exclamation mark to it, and then returns the transformed value. The custom pipe is then used in the template by passing the string 'hello world' to it.

here's an example of how to use the **filter** pipe in Angular to filter an array of objects, including the relevant HTML, TypeScript, and pipe files, as well as the app module file:

```
<input type="text" [(ngModel)]="searchQuery">
<ul>
<li *ngFor="let item of items | filter: searchQuery">
   {{ item.name }}
</li>
</ul>
```

In this example, we have an input field where the user can type in a search query. We also have a list of items that we want to filter based on the search query. We use the **filter** pipe to filter the items based on the search query.
TypeScript:

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

@Component({
selector: 'app-item-list',
templateUrl: './item-list.component.html',
styleUrls: ['./item-list.component.css']
})
export class ItemListComponent {
items = [
{ id: 1, name: 'Item 1' },
{ id: 2, name: 'Item 2' },
{ id: 3, name: 'Item 3' },
{ id: 4, name: 'Item 4' },
{ id: 5, name: 'Item 5' }
  ];
searchQuery: string = '';
}
```

In the TypeScript file, we define an array of items that we want to filter. We also define a **searchQuery** variable to hold the search query entered by the user.

Filter Pipe:

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

@Pipe({
name: 'filter'
})
export class FilterPipe implements PipeTransform {
transform(items: any[], searchText: string): any[] {
if (!items) {
return [];
    }
if (!searchText) {
return items;
    }
searchText = searchText.toLowerCase();
return items.filter(item => {
return item.name.toLowerCase().includes(searchText);
    });
  }
}
```

In the filter pipe, we define a custom **filter** pipe to filter the items based on the search query. The pipe takes in an array of items and a search query, and returns a filtered array based on the search query. The filter function checks if each item's name includes the search query and returns only those items that match.

App Module:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { FilterPipe } from './filter.pipe';
```

```
@NgModule({
 declarations: [
    FilterPipe,
    ItemListComponent
  ],
imports: [
    BrowserModule,
    FormsModule
  ],
providers: [],
bootstrap: []
})
export class AppModule { }
```

In the app module, we import the **FilterPipe** and declare them in the declarations array.

app.component.html

```
<app-productfilter></app-productfilter>
```

productfilter.component.html

```
<h1 align="center">Search for  : <input type="text" placeholder="search item" [(ngModel)]="searchQuery"
style="height: 30px;"></h1>
<p>{{searchQuery}}</p>
<table align="center" style="border: 1px solid black;"width="500px;">
    <tr>
        <th>sr. no</th>
        <th>name</th>
        <th>price</th>
        <th>image</th>
        <th>date</th>
    </tr>
    <tr *ngFor="let item of products|filter:searchQuery; index as i">
        <td>{{i+1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.price}}</td>
        <td><img src="{{item.image}}" style="height: 150px;width: 150px;"></td>
        <td>{{todaydate|date}}</td>
    </tr>
</table>
```

productfilter.component.ts

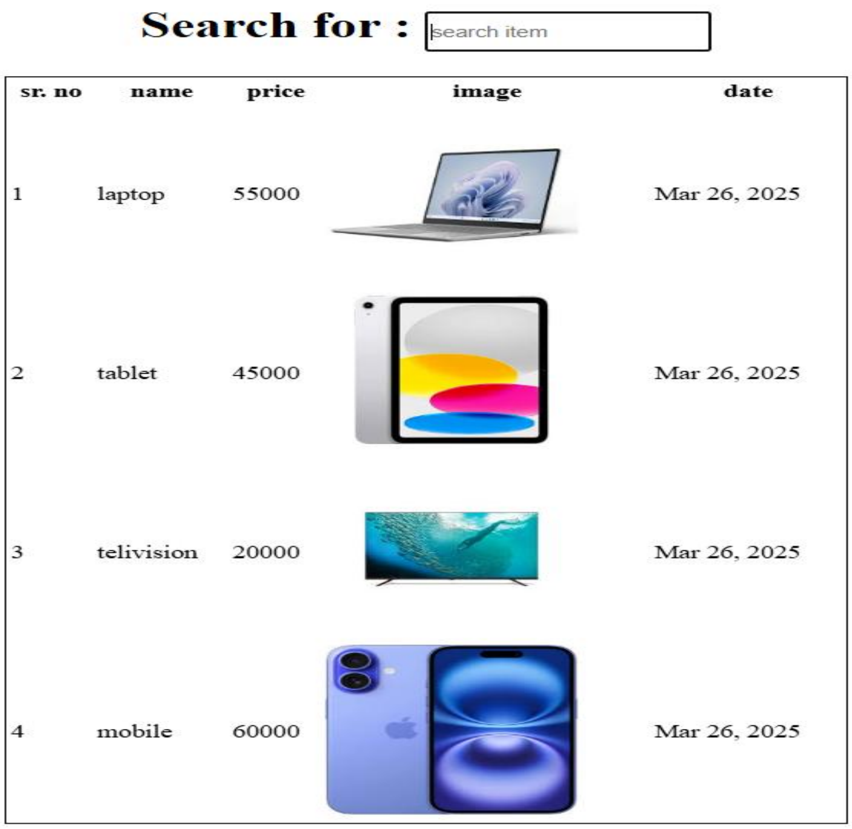
```
import { Component } from '@angular/core';
@Component({

  selector: 'app-productfilter',
  templateUrl: './productfilter.component.html',
  styleUrls: ['./productfilter.component.css']
})
export class ProductfilterComponent {
  todaydate=new Date();
  searchQuery:any='';
  products=[
  {name:'laptop',price:55000,image:'assets/laptop.jpg'},
  {name:'tablet',price:45000,image:'assets/tablet.jpg'},
  {name:'telivision',price:20000,image:'assets/tv.jpg'},
  {name:'mobile',price:60000,image:'assets/mobile.jpg'}
]
}
```

filter.pipe.ts

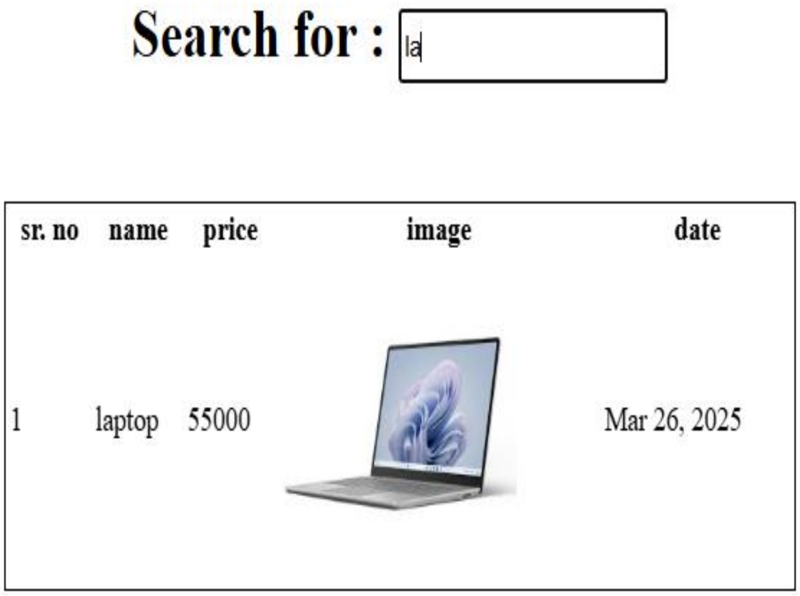
```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'filter'
})
export class FilterPipe implements PipeTransform {
  transform(products: any[], searchQuery: any): any[] {
    return products.filter(items=> {
      return items.name.toLowerCase().includes(searchQuery.toLowerCase())
    })
  }
}
```

Output :



Search for : search item

sr. no	name	price	image	date
1	laptop	55000		Mar 26, 2025
2	tablet	45000		Mar 26, 2025
3	telivision	20000		Mar 26, 2025
4	mobile	60000		Mar 26, 2025



Search for : la

sr. no	name	price	image	date
1	laptop	55000		Mar 26, 2025