Why Angular4 and Not Angular3?

The Angular team faced some versioning issues internally with their modules and due to the conflict they had to move on and release the next version of Angular – the Angular4.

Let us now see the new features added to Angular 4 −

ngIf

Angular2 supported only the **if** condition. However, Angular 4 supports the **if else** condition as well. Let us see how it works using the ng-template.

<span \*ngIf="isavailable; else condition1">Condition is valid.</span>

<ng-template #condition1>Condition is invalid</ng-template>

as keyword in for loop

With the help of **as** keyword you can store the value as shown below −

<div \*ngFor="let i of months | slice:0:5 as total">

Months: {{i}} Total: {{total.length}}

</div>

The variable total stores the output of the slice using the **as** keyword.

Animation Package

Animation in Angular 4 is available as a separate package and needs to be imported from @angular/animations. In Angular2, it was available with @**angular/core**. It is still kept the same for its backward compatibility aspect.

Template

**Angular 4** uses **<ng-template>** as the tag instead of **<template>;** the latter was used in Angular2. The reason Angular 4 changed **<template>** to **<ng-template>** is because of the name conflict of the **<template>** tag with the html **<template>** standard tag. It will deprecate completely going ahead. This is one of the major changes in Angular 4.

TypeScript 2.2

Angular 4 is updated to a recent version of TypeScript, which is 2.2. This helps improve the speed and gives better type checking in the project.

Pipe Title Case

Angular 4 has added a new pipe title case, which changes the first letter of each word into uppercase.

<div>

<h2>{{ 'Angular 4 titlecase' | titlecase }}</h2>

</div>

The above line of code generates the following output – **Angular 4 Titlecase**.

Http Search Parameters

Search parameters to the http get api is simplified. We do not need to call **URLSearchParams** for the same as was being done in Angular2.

Smaller and Faster Apps

Angular 4 applications are smaller and faster when compared to Angular2. It uses the TypeScript version 2.2, the latest version which makes the final compilation small in size.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Major part of the development with Angular 4 is done in the components. Components are basically classes that interact with the .html file of the component, which gets displayed on the browser. We have seen the file structure in one of our previous chapters. The file structure has the app component and it consists of the following files −

* **app.component.css**
* **app.component.html**
* **app.component.spec.ts**
* **app.component.ts**
* **app.module.ts**

The above files were created by default when we created new project using the angular-cli command.

If you open up the **app.module.ts** file, it has some libraries which are imported and also a declarative which is assigned the appcomponent as follows −

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

The declarations include the AppComponent variable, which we have already imported. This becomes the parent component.

Now, angular-cli has a command to create your own component. However, the app component which is created by default will always remain the parent and the next components created will form the child components.

Let us now run the command to create the component.

ng g component new-cmp

When you run the above command in the command line, you will receive the following output −

C:\projectA4\Angular 4-app>ng g component new-cmp

installing component

create src\app\new-cmp\new-cmp.component.css

create src\app\new-cmp\new-cmp.component.html

create src\app\new-cmp\new-cmp.component.spec.ts

create src\app\new-cmp\new-cmp.component.ts

update src\app\app.module.ts

Now, if we go and check the file structure, we will get the new-cmp new folder created under the src/app folder.

The following files are created in the new-cmp folder −

* new-cmp.component.css − css file for the new component is created.
* new-cmp.component.html − html file is created.
* new-cmp.component.spec.ts − this can be used for unit testing.
* new-cmp.component.ts − here, we can define the module, properties, etc.

Changes are added to the app.module.ts file as follows −

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

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

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

import { NewCmpComponent } from './new-cmp/new-cmp.component';

// includes the new-cmp component we created

@NgModule({

declarations: [

AppComponent,

NewCmpComponent // here it is added in declarations and will behave as a child component

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent] //for bootstrap the AppComponent the main app component is given.

})

export class AppModule { }

The **new-cmp.component.ts** file is generated as follows −

import { Component, OnInit } from '@angular/core'; // here angular/core is imported .

@Component({

// this is a declarator which starts with @ sign. The component word marked in bold needs to be the same.

selector: 'app-new-cmp', //

templateUrl: './new-cmp.component.html',

// reference to the html file created in the new component.

styleUrls: ['./new-cmp.component.css'] // reference to the style file.

})

export class NewCmpComponent implements OnInit {

constructor() { }

ngOnInit() {}

}

If you see the above new-cmp.component.ts file, it creates a new class called NewCmpComponent, which implements OnInit.In, which has a constructor and a method called ngOnInit(). ngOnInit is called by default when the class is executed.

Let us check how the flow works. Now, the app component, which is created by default becomes the parent component. Any component added later becomes the child component.

When we hit the url in the **http://localhost:4200/** browser, it first executes the index.html file which is shown below −

<!doctype html>

<html lang = "en">

<head>

<meta charset = "utf-8">

<title>Angular 4App</title>

<base href = "/">

<meta name="viewport" content="width = device-width, initial-scale = 1">

<link rel = "icon" type = "image/x-icon" href = "favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

The above is the normal html file and we do not see anything that is printed in the browser. Take a look at the tag in the body section.

<app-root></app-root>

This is the root tag created by the Angular by default. This tag has the reference in the **main.ts** file.

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

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

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule);

AppModule is imported from the app of the main parent module, and the same is given to the bootstrap Module, which makes the appmodule load.

Let us now see the **app.module.ts** file −

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

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

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

import { NewCmpComponent } from './new-cmp/new-cmp.component';

@NgModule({

declarations: [

AppComponent,

NewCmpComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Here, the AppComponent is the name given, i.e., the variable to store the reference of the **app. Component.ts** and the same is given to the bootstrap. Let us now see the **app.component.ts** file.

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

}

Angular core is imported and referred as the Component and the same is used in the Declarator as −

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

In the declarator reference to the selector, **templateUrl** and **styleUrl** are given. The selector here is nothing but the tag which is placed in the index.html file that we saw above.

The class AppComponent has a variable called title, which is displayed in the browser.

The **@Component** uses the templateUrl called app.component.html which is as follows −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

It has just the html code and the variable title in curly brackets. It gets replaced with the value, which is present in the **app.component.ts** file. This is called binding. We will discuss the concept of binding in a subsequent chapter.

Now that we have created a new component called **new-cmp**. The same gets included in the **app.module.ts** file, when the command is run for creating a new component.

**app.module.ts** has a reference to the new component created.

Let us now check the new files created in new-cmp.

new-cmp.component.ts

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

@Component({

selector: 'app-new-cmp',

templateUrl: './new-cmp.component.html',

styleUrls: ['./new-cmp.component.css']

})

export class NewCmpComponent implements OnInit {

constructor() { }

ngOnInit() {}

}

Here, we have to import the core too. The reference of the component is used in the declarator.

The declarator has the selector called **app-new-cmp** and the **templateUrl**and **styleUrl**.

The .html called **new-cmp.component.html** is as follows −

<p>

new-cmp works!

</p>

As seen above, we have the html code, i.e., the p tag. The style file is empty as we do not need any styling at present. But when we run the project, we do not see anything related to the new component getting displayed in the browser. Let us now add something and the same can be seen in the browser later.

The selector, i.e., **app-new-cmp** needs to be added in the **app.component .html** file as follows −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<app-new-cmp></app-new-cmp>

When the **<app-new-cmp></app-new-cmp>** tag is added, all that is present in the .html file of the new component created will get displayed on the browser along with the parent component data.

Let us see the **new component .html** file and the **new-cmp.component.ts**file.

new-cmp.component.ts

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

@Component({

selector: 'app-new-cmp',

templateUrl: './new-cmp.component.html',

styleUrls: ['./new-cmp.component.css']

})

export class NewCmpComponent implements OnInit {

newcomponent = "Entered in new component created";

constructor() {}

ngOnInit() { }

}

In the class, we have added one variable called new component and the value is “**Entered in new component created**”.

The above variable is bound in the **.new-cmp.component.html** file as follows −

<p>

{{newcomponent}}

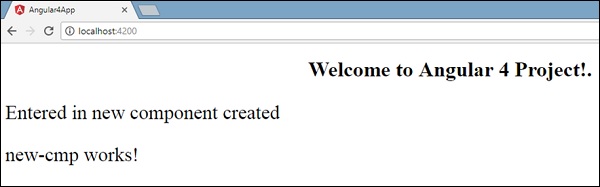
</p>

<p>

new-cmp works!

</p>

Now since we have included the **<app-new-cmp></app-new-cmp>**selector in the **app. component .html** which is the .html of the parent component, the content present in the new component .html file (new-cmp.component.html) gets displayed on the browser as follows −



Similarly, we can create components and link the same using the selector in the **app.component.html** file as per our requirements.

**Module** in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.

In case you are developing a website, the header, footer, left, center and the right section become part of a module.

To define module, we can use the **NgModule**. When you create a new project using the Angular –cli command, the ngmodule is created in the app.module.ts file by default and it looks as follows −

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

The NgModule needs to be imported as follows −

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

The structure for the ngmodule is as shown below −

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

It starts with **@NgModule** and contains an object which has declarations, import s, providers and bootstrap.

### Declaration

It is an array of components created. If any new component gets created, it will be imported first and the reference will be included in declarations as shown below −

declarations: [

AppComponent,

NewCmpComponent

]

### Import

It is an array of modules required to be used in the application. It can also be used by the components in the Declaration array. For example, right now in the @NgModule we see the Browser Module imported. In case your application needs forms, you can include the module as follows −

import { FormsModule } from @angular/forms;

The import in the **@NgModule** will be like the following −

imports: [

BrowserModule,

FormsModule

]

### Providers

This will include the services created.

### Bootstrap

This includes the main app component for starting the execution.

# Angular 4 - Data Binding

Advertisements

[Previous Page](https://www.tutorialspoint.com/angular4/angular4_module.htm)

[Next Page](https://www.tutorialspoint.com/angular4/angular4_event_binding.htm)

Data Binding is available right from AngularJS, Angular 2 and is now available in Angular 4 as well. We use curly braces for data binding - {{}}; this process is called interpolation. We have already seen in our previous examples how we declared the value to the variable title and the same is printed in the browser.

The variable in the **app.component.html** file is referred as {{title}} and the value of title is initialized in the **app.component.ts** file and in **app.component.html**, the value is displayed.

Let us now create a dropdown of months in the browser. To do that , we have created an array of months in **app.component.ts** as follows −

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

// declared array of months.

months = ["January", "Feburary", "March", "April", "May",

"June", "July", "August", "September",

"October", "November", "December"];

}

The month’s array that is shown above is to be displayed in a dropdown in the browser. For this, we will use the following line of code −

<!--The content below is only a placeholder and can be replaced. -->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

We have created the normal select tag with option. In option, we have used the **for loop**. The **for loop** is used to iterate over the months’ array, which in turn will create the option tag with the value present in the months.

The syntax **for** in Angular is **\*ngFor = “let I of months”** and to get the value of months we are displaying it in {{i}}.

The two curly brackets help with data binding. You declare the variables in your **app.component.ts** file and the same will be replaced using the curly brackets.

Let us see the output of the above month’s array in the browser



The variable that is set in the **app.component.ts** can be bound with the **app.component.html** using the curly brackets; for example, **{{}}**.

Let us now display the data in the browser based on condition. Here, we have added a variable and assigned the value as true. Using the if statement, we can hide/show the content to be displayed.

### Example

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true; //variable is set to true

}

<!--The content below is only a placeholder and can be replaced.-->

<div style = "text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor = "let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

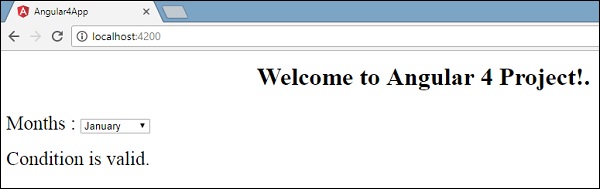
<span \*ngIf = "isavailable">Condition is valid.</span>

//over here based on if condition the text condition is valid is displayed.

If the value of isavailable is set to false it will not display the text.

</div>

### Output



Let us try the above example using the **IF THEN ELSE** condition.

### Example

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = false;

}

In this case, we have made the **isavailable** variable as false. To print the **else** condition, we will have to create the **ng-template** as follows −

<ng-template #condition1>Condition is invalid</ng-template>

The full code looks like this −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

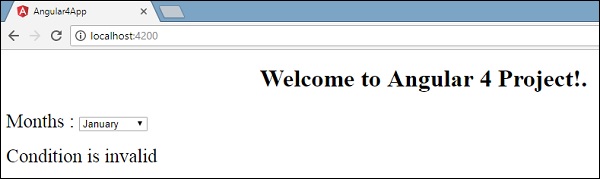
<span \*ngIf="isavailable; else condition1">Condition is valid.</span>

<ng-template #condition1>Condition is invalid</ng-template>

</div>

**If** is used with the else condition and the variable used is **condition1**. The same is assigned as an **id** to the **ng-template**, and when the available variable is set to false the text **Condition is invalid** is displayed.

The following screenshot shows the display in the browser.



Let us now use the **if then else** condition.

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

}

Now, we will make the variable **isavailable** as true. In the html, the condition is written in the following way −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf="isavailable; then condition1 else condition2">Condition is valid.</span>

<ng-template #condition1>Condition is valid</ng-template>

<ng-template #condition2>Condition is invalid</ng-template>

</div>

If the variable is true, then **condition1**, else **condition2**. Now, two templates are created with id **#condition1** and **#condition2**.

The display in the browser is as follows −

# Angular 4 - Event Binding

Advertisements

[Previous Page](https://www.tutorialspoint.com/angular4/angular4_data_binding.htm)

[Next Page](https://www.tutorialspoint.com/angular4/angular4_templates.htm)

In this chapter, we will discuss how Event Binding works in Angular 4. When a user interacts with an application in the form of a keyboard movement, a mouse click, or a mouseover, it generates an event. These events need to be handled to perform some kind of action. This is where event binding comes into picture.

Let us consider an example to understand this better.

### app.component.html

<!--The content below is only a placeholder and can be replaced.-->

<div style = "text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor = "let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf = "isavailable; then condition1 else condition2">

Condition is valid.

</span>

<ng-template #condition1>Condition is valid</ng-template>

<ng-template #condition2>Condition is invalid</ng-template>

</div>

<button (click)="myClickFunction($event)">

Click Me

</button>

In the **app.component.html** file, we have defined a button and added a function to it using the click event.

Following is the syntax to define a button and add a function to it.

(click)="myClickFunction($event)"

The function is defined in the **.ts** file: **app.component.ts**

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "Feburary", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

myClickFunction(event) {

//just added console.log which will display the event details in browser on click of the button.

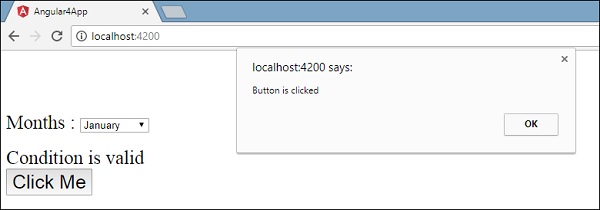
alert("Button is clicked");

console.log(event);

}

}

Upon clicking the button, the control will come to the function **myClickFunction** and a dialog box will appear, which displays **the Button is clicked** as shown in the following screenshot −



Let us now add the change event to the dropdown.

The following line of code will help you add the change event to the dropdown −

<!--The content below is only a placeholder and can be replaced.-->

<div style = "text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select (change) = "changemonths($event)">

<option \*ngFor = "let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf = "isavailable; then condition1 else condition2">

Condition is valid.

</span>

<ng-template #condition1>Condition is valid</ng-template>

<ng-template #condition2>Condition is invalid</ng-template>

</div>

<button (click) = "myClickFunction($event)">Click Me</button>

The function is declared in the **app.component.ts** file −

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "Feburary", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

myClickFunction(event) {

alert("Button is clicked");

console.log(event);

}

changemonths(event) {

console.log("Changed month from the Dropdown");

console.log(event);

}

}

The console message “**Changed month from the Dropdown**” is displayed in the console along with the event.



Let us add an alert message in **app.component.ts** when the value from the dropdown is changed as shown below −

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

myClickFunction(event) {

//just added console.log which will display the event details in browser

on click of the button.

alert("Button is clicked");

console.log(event);

}

changemonths(event) {

alert("Changed month from the Dropdown");

}

}

When the value in dropdown is changed, a dialog box will appear and the following message will be displayed - “**Changed month from the Dropdown**”.

# Angular 4 - Pipes

Advertisements

[Previous Page](https://www.tutorialspoint.com/angular4/angular4_directives.htm)

[Next Page](https://www.tutorialspoint.com/angular4/angular4_routing.htm)

In this chapter, we will discuss what are Pipes in Angular 4. Pipes were earlier called filters in Angular1 and called pipes in Angular 2 and 4.

The | character is used to transform data. Following is the syntax for the same

{{ Welcome to Angular 4 | lowercase}}

It takes integers, strings, arrays, and date as input separated with **|** to be converted in the format as required and display the same in the browser.

Let us consider a few examples using pipes.

Here, we want to display the text given to uppercase. This can be done using pipes as follows −

In the **app.component.ts** file, we have defined the title variable −

### app.component.ts

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

}

The following line of code goes into the **app.component.html** file.

<b>{{title | uppercase}}</b><br/>

<b>{{title | lowercase}}</b>

The browser appears as shown in the following screenshot −



Angular 4 provides some built-in pipes. The pipes are listed below −

* Lowercasepipe
* Uppercasepipe
* Datepipe
* Currencypipe
* Jsonpipe
* Percentpipe
* Decimalpipe
* Slicepipe

We have already seen the lowercase and uppercase pipes. Let us now see how the other pipes work.

The following line of code will help us define the required variables in **app.component.ts** file −

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

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

todaydate = new Date();

jsonval = {name:'Rox', age:'25', address:{a1:'Mumbai', a2:'Karnataka'}};

months = ["Jan", "Feb", "Mar", "April", "May", "Jun",

"July", "Aug", "Sept", "Oct", "Nov", "Dec"];

}

We will use the pipes in the **app.component.html** file.

<!--The content below is only a placeholder and can be replaced.-->

<div style = "width:100%;">

<div style = "width:40%;float:left;border:solid 1px black;">

<h1>Uppercase Pipe</h1>

<b>{{title | uppercase}}</b><br/>

<h1>Lowercase Pipe</h1>

<b>{{title | lowercase}}</b>

<h1>Currency Pipe</h1>

<b>{{6589.23 | currency:"USD"}}</b><br/>

<b>{{6589.23 | currency:"USD":true}}</b> //Boolean true is used to get the sign of the currency.

<h1>Date pipe</h1>

<b>{{todaydate | date:'d/M/y'}}</b><br/>

<b>{{todaydate | date:'shortTime'}}</b>

<h1>Decimal Pipe</h1>

<b>{{ 454.78787814 | number: '3.4-4' }}</b> // 3 is for main integer, 4 -4 are for integers to be displayed.

</div>

<div style = "width:40%;float:left;border:solid 1px black;">

<h1>Json Pipe</h1>

<b>{{ jsonval | json }}</b>

<h1>Percent Pipe</h1>

<b>{{00.54565 | percent}}</b>

<h1>Slice Pipe</h1>

<b>{{months | slice:2:6}}</b>

// here 2 and 6 refers to the start and the end index

</div>

</div>

Routing basically means navigating between pages. You have seen many sites with links that direct you to a new page. This can be achieved using routing. Here the pages that we are referring to will be in the form of components. We have already seen how to create a component. Let us now create a component and see how to use routing with it.

In the main parent component **app.module.ts**, we have to now include the router module as shown below −

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

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

import { RouterModule} from '@angular/router';

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

import { NewCmpComponent } from './new-cmp/new-cmp.component';

import { ChangeTextDirective } from './change-text.directive';

import { SqrtPipe } from './app.sqrt';

@NgModule({

declarations: [

SqrtPipe,

AppComponent,

NewCmpComponent,

ChangeTextDirective

],

imports: [

BrowserModule,

RouterModule.forRoot([

{

path: 'new-cmp',

component: NewCmpComponent

}

])

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

import { RouterModule} from '@angular/router'

Here, the RouterModule is imported from angular/router. The module is included in the imports as shown below −

RouterModule.forRoot([

{

path: 'new-cmp',

component: NewCmpComponent

}

])

RouterModule refers to the **forRoot** which takes an input as an array, which in turn has the object of the path and the component. Path is the name of the router and component is the name of the class, i.e., the component created.

Let us now see the component created file −

New-cmp.component.ts

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

@Component({

selector: 'app-new-cmp',

templateUrl: './new-cmp.component.html',

styleUrls: ['./new-cmp.component.css']

})

export class NewCmpComponent implements OnInit {

newcomponent = "Entered in new component created";

constructor() {}

ngOnInit() { }

}

The highlighted class is mentioned in the imports of the main module.

New-cmp.component.html

<p>

{{newcomponent}}

</p>

<p>

new-cmp works!

</p>

Now, we need the above content from the html file to be displayed whenever required or clicked from the main module. For this, we need to add the router details in the **app.component.html**.

<h1>Custom Pipe</h1>

<b>Square root of 25 is: {{25 | sqrt}}</b><br/>

<b>Square root of 729 is: {{729 | sqrt}}</b>

<br />

<br />

<br />

<a routerLink = "new-cmp">New component</a>

<br />

<br/>

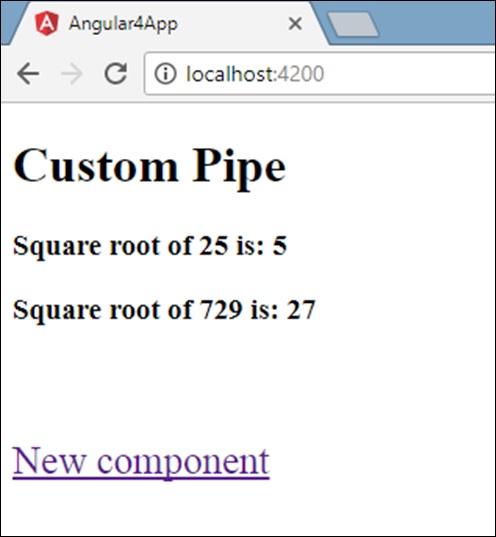
<router-outlet></router-outlet>

In the above code, we have created the anchor link tag and given routerLink as **“new-cmp”**. This is referred in **app.module.ts** as the path.

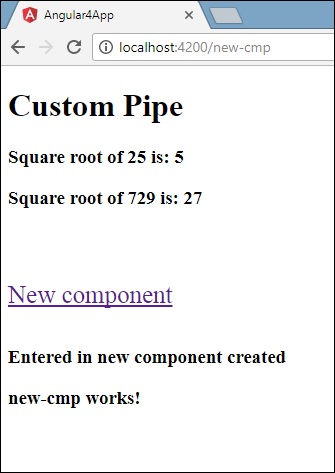
When a user clicks **new component**, the page should display the content. For this, we need the following tag - **<router-outlet> </router-outlet>**.

The above tag ensures that the content in the **new-cmp.component.html**will be displayed on the page when a user clicks **new component**.

Let us now see how the output is displayed on the browser.



When a user clicks New component, you will see the following in the browser.



The url contains **http://localhost:4200/new-cmp**. Here, the new-cmp gets appended to the original url, which is the path given in the **app.module.ts**and the router-link in the **app.component.html**.

When a user clicks New component, the page is not refreshed and the contents are shown to the user without any reloading. Only a particular piece of the site code will be reloaded when clicked. This feature helps when we have heavy content on the page and needs to be loaded based on the user interaction. The feature also gives a good user experience as the page is not reloaded.

In [Taking advantage of Observables Part one](https://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html) and [two](https://blog.thoughtram.io/angular/2016/01/07/taking-advantage-of-observables-in-angular2-pt2.html) we already highlighted the importance of Observables in Angular. We believe that mastering Observables can make a key difference in how we write our applications. Well, if you agree, here are some good news! This article is the first of a series of posts where we’ll explore operators of the Reactive Extensions for JavaScript (RxJS) and their practical applications.

The first operator we want to explore is the most commonly used one: map.

## [Understanding the map operator](https://blog.thoughtram.io/angular/2016/05/16/exploring-rx-operators-map.html#understanding-the-map-operator)

We’ve probably all used map before when we were working with arrays. The idea is that each item in the collection will potentially be projected into a different value.

Here is a very simple example where an array of numbers is transformed so that each number is multiplied by 10.

let values = [1, 2, 3];

let transformed = values.map(value => value \* 10);

//prints [10, 20, 30]

console.log(transformed);

By now we may be wondering what that has to do with Observables and the map operator that is part of RxJS.

Observables are very much like arrays in a way. Well, they are actually more like Iterators but let’s not get lost in the details. The key point to understand is that both represent a sequence of values. The key difference is that with Arrays/Iterators you pull values out as you want to work with them whereas with Observables you get values **pushed** to you as they arrive.

It’s this similarity that allows us to take advantage of pretty much all operators that we know from the pull-based world and apply them to the push-based world.

## [map and Observables](https://blog.thoughtram.io/angular/2016/05/16/exploring-rx-operators-map.html#map-and-observables)

Let’s start with a little demo. All we need is a simple <input> element to enter some text.

<input type="text" id="demo"/>

Then we create an Observable that emits every time that the value of our input changes.

let demoInput = document.querySelector('#demo')

let obs = Rx.Observable.fromEvent(demoInput, 'input');

// Activate the observable and log all 'pushed' events

obs.subscribe(event => console.log(event));

The payload of the Observable is the plain old Event object that is provided by the inputevent of the browser. But that may not match what we are most interested in. What if we are more interested in the current value of the input? The map operator lets us project the payload of the Observable into something else. All it takes to project the payload is this tiny change.

let obs = Rx.Observable.fromEvent(demoInput, 'input')

.map(e => e.target.value);

We can go on and chain map calls to project the data even further. For instance, it may be more convenient to work with a data structure that carries the value among with the length of the string.

let obs = Rx.Observable.fromEvent(demoInput, 'input')

.map(e => e.target.value)

.filter( value => value > 100 )

.map(v => {

return {

value: v,

length: v.length

};

});

Of course, we could have done the same in the first map call. But it’s sometimes more readable to break things into multiple steps. Notice that often we also use different operators in between of two map calls (e.g to filter something out).

If you like to play a bit with the operator yourself, here is a working demo.

At this point, you may think that Observables are really just a minor enhancement on the Observer or Promise patterns… better suited to handle a sequence of events rather than a single callback. And the .map() function certainly does not - at first glance - seem to offer any added-value. The power of Observables is revealed when you start using Rx operators to transform, combine, manipulate, and work with sequences of items emitted by Observables.

These operators allow you to compose asynchronous sequences together in a declarative manner with all the efficiency benefits of callbacks but without the drawbacks of nesting callback handlers that are typically associated with asynchronous systems.

We will see that in future articles. Watch out for the next article of this series where we’ll build upon this lesson with map() and take a look at the related flatMap operator.