**Data Binding in Angular:**

**Ng generate pipe mycoolpipe**

**Ng serve –open**

**Npm install @angular@forms**

**Data Binding:**

Data Binding is a process where data is passed from Angular Component to view (Template) and vice versa. The Data Binding is used to bind DOM Elements to Component properties. Binding can be used display component class property values to the user, change element styles, respond to a user event etc.

The Data binding is not new to Angular. It existed in AngularJs., but the syntax has changed

**There are four ways you can bind data in Angular**

1. Interpolation
2. Property Binding
3. Event Binding
4. Two Way Binding

**Template Expression:**

The content inside the double braces is called **Template Expression** in Angular.

The Angular first evaluates the Template Expression and converts it into a string. Then it replaces Template expression with this result in the HTML Markup.

**Template statement:**

Template statement is similar to Template expression but can change the state of the application.

Template statements are used in case of Event Bindings. It responds to the event raised by the user like clicking on a save button (Click event) or modifying the value of textbox (Change event) etc and invokes the method in the component class. Template statement is often is a method in the Component class

Angular evaluates the Template statement just like the Template expression. Angular updates the view after the Template statement is invoked

**Difference between Template Expression and Template statement:**

Template statement can change the Application state from the user input. The Template expression should not change the application state.

Both use different Parsers.

Template expression does not support assignment ( = only) operator. Template statement does.

Template Expression does not support chaining of expressions. Template statement does.

## One-way Data binding

## Interpolation

<h1> {{title}} </h1>

export class AppComponent {

title= 'Data Binding in Angular';

}

The title property, which is defined in the component class bound to the template using **double curly braces** in the template. This is called **Interpolation**

Interpolation provides the data-binding from **component to the View**. Interpolation is a one-way binding. It binds from Component Class to the Template.

The Template expression (double curly braces ) used for interpolation in Angular. The Angular evaluates the Template expression and replaces it with the result.

In the above example, we got the data from the class Property called title.

You can use interpolation to invoke a method in the component, Concatenate two string, perform some mathematical operations or change the property of the DOM element like colour etc.

**Bind to the Element Property**

In the following code, we bind to the colour property of the paragraph element. Using interpolation we can bind to any property of the DOM element.

**Notes on Interpolation**

* Interpolation is one way from component to View
* Binding source is a Template expression
* Interpolation Expression must result in a string. If we return an object it will not work

**Example:**

<p style.color={{color}}>This is red</p>

## Property Binding:

Property binding allows us to bind **Property of a view element** to the value of template expression

[Property]=”expression”

The Property Binding uses the **[] brackets**. The Binding Target is placed inside the square brackets. The Binding source is enclosed in quotes.

Property binding is one way from Component to the Target in the template.

You can add these codes to the app*.component.html* and see the result

<p [innerText]="title"></p>

<p [innerText]="getTitle()"></p>

<p [innerText]="'Hello & Welcome to '+ ' Angular Data binding '"></p>

<p [innerText]="100\*80"></p>

<p [style.color]="color">This is red</p>

#### **Notes on Property Binding:**

1. Property Binding is one way from component to View
2. Binding source is a Template expression
3. Non-string return values are allowed

## Property Binding Vs Interpolation:

Everything that can be done from Property binding can be done using the interpolation.  In fact, Angular automatically translates interpolations into the corresponding property bindings before rendering the view.

Interpolation is simple and readable.  For example, the h1 title in interpolation is intuitive

<h1>{{propVar}}</h1>

And you can achieve the same result using the Property binding using

<h1 [innerText] = "propVar"> </h1>

Interpolation requires expression to return a string. If you want to set an element property to a non-string data value, you must use property binding.

**Examples:**

Property Binding: to set an element property to a non-string data value, you must use property binding.

In the example below, we are disabling a button by binding to the Boolean property isDisabled.

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

@Component({

selector: 'my-app',

template: `<div>

<button [disabled]='isDisabled'>Try Me</button>

</div>`

})

export class AppComponent {

isDisabled: boolean = true;

}

If we decide to use interpolation instead of property binding, the button will always be disabled irrespective of isDisabled class property value is true of false.

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

@Component({

selector: 'my-app',

template: `<div>

<button disabled='{{isDisabled}}'>Try Me</button>

</div>`

})

export class AppComponent {

isDisabled: boolean = true;

}

One other thing I want to introduce is, instead of using pair of square bracket to depict property binding you can also use the canonical form which is an alternate syntax to square bracket.

<button bind-disabled='isDisabled'>Try Me</button>

What to be used either pair of square bracket or canonical form is based on preference.

## Event Binding:

The interpolation and Property binding are one-way bindings from component to view.

Event Binding is used to Perform an action in the component when the user clicks a button in the view.

<button (click)=’ClickPressed()’>

The Event Binding uses the **parenthesis**. The Name of the event is enclosed in Parentheses. It is then assigned to the **Template Statement**. Template statement is often the method in the component class

Add the following code to the app.component.html

<button (click)='buttonClicked()'>Click Me</button>

<p> Button Clicked {{count}} Times </p>

app.component.ts:

count: number=0;

buttonClicked() : void {

    this.count=this.count+1;

    console.log("Button Clicked")

}

In the above example, the component listens to the click event on the button. It then executes the ClickPressed() method and increases the count by one.

Event Binding is One-way binding.Unlike the Interpolation and Property Binding, the Data flows from Target to the Source. I.e from the View to the Component

***Notes on Event Binding:***

1. Event Binding is one way from View to Component
2. Binding source is a Template statement

## Two Way Binding:

Two-way binding means that changes made in the component data are propagated to the view and that any changes made in the view are immediately updated in the underlying component data.

Two-way binding is used mainly in data entry forms. Whenever user makes changes in the data, we would like to update our model in the component with the new data and if the model changes, we would like to update the view as well

The Angular uses the combination of Property binding (from component to view) and event binding (from view to component) to achieve the Two-way data binding. This is done so by using the ngModel directive

To Specify Two Way Binding, We need to use the ngModel directives as shown below

**<input [(ngModel)] =’name’></input>**

The ngModel directive placed inside the square & parentheses as shown above. This is assigned to the Template Statement. Template statement is usually the property in the component class

Note that both square & parentheses are used here. This is now known as **Banana in a box syntax**. The square indicates the Property binding & parentheses indicates the event binding.

The ngModel directive is not part of the Angular Core library. It is part of the FormsModule library. You need to import the FormsModule package into your Angular module.

Open the app.module.ts and make the following changes

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

Enter your Name : <input type='text' [(ngModel)]='name' />

You have entered {{name}}

***Notes on Two-way Binding:***

1. Two-way Binding is one way from View to Component and other way component to view.
2. You need to use ngModel directive, which is available in FormsModule and add it to the NgModule’s imports list
3. Uses **Banana in a box syntax**

### **Directives**

The Angular directive helps us to manipulate the DOM. You can change the appearance, behaviour or a layout of a DOM element using the Directives. They help you to extend HTML. The Angular directives are classified into three categories based on how they behave.  They are

* Component.
* Structural.
* Attribute Directives.

### **Structural & Attribute Directives:**

Rather than adding new DOM elements, both these types of directives modify the existing DOM and do not have templates.

Attribute: Modifies the appearance or behavior of an element.

Structural: Modifies the DOM layout.

**Attribute Directive**

Examples of built-in attribute directives that ship with Angular 2 are ngStyle and ngClass. As mentioned before, these directives modify the DOM and we are going to do the same in a simple custom attribute directive of our own. The naming convention for directives is: name.directive.ts

**Component:**

The Component is the main building block of an Angular Application.

The **View**looks and behaves. A view in Angular The component contains the data & user interaction logic that defines how refers to a **template** (HTML).

The Angular Components are plain javascript classes and defined using **@component Decorator**. This Decorator provides the component with the View to display & Metadata about the class

The Component is responsible to provide the data to the view. The Angular does this by using data binding to get the data from the Component to the View. This is done using the special HTML markup knows as the Angular Template Syntax. The Component can also get notified when the View Changes.

The Components consists of three main building blocks

* Template
* Class
* MetaData

**Template (View)**

The template defines the Layout of the View and defines what is rendered on the page.  Without the template,  there is nothing for Angular to render to the DOM.

The Templates are created with HTML. You can add Angular directives and bindings on the template.

There are two ways you can specify the Template in Angular.

1. Defining the Template Inline
2. Provide an external Template

**Class**

The class is the code associated with Template (View). The Class is created with the Typescript, but you can also use javascript directly in the class. Class Contains the Properties & Methods.

The Properties of a class can be bind to the view using Data Binding.

The Angular applications will have lots of components. Each component handles a small part of UI. These components work together to produce the complete user interface of the application

### **Metadata**

Metadata Provides additional information about the component to the Angular. Angular uses this information to process the class. The Metadata is defined with a **decorator**.

A decorator is a function that adds metadata to class, its methods & to its properties. The Components are defined with a @component class decorator. It is @component decorator, which defines the class as Component to the Angular

#### @Component decorator

A class becomes a Component when Component Decorator is used. A Decorator is always prefixed with @. The Decorator must be positioned immediately before the class definition. We can also build our own decorators. The decorators are Similar to attributes in C#

#### Important Component metadata properties.

##### **Selector**

Selector specifies the simple CSS selector, where our view representing the component is placed by the Angular.

##### **Providers**

The Providers are the services, that our component going to use. The Services provide service to the Components or to the other Services.

##### Directives

The directives that this component going to use are listed here.

##### Styles/styleUrls

The CSS Styles or style sheets, that this component needs. Here we can use either external stylesheet (using styleUrls) or inline styles (using Styles). The styles used here are specific to the component

##### **template/templateUrl**

The HTML template that defines our View. It tells angular how to render the Component’s view. The templates can be inline (using a template) or we can use an external template (using a templateUrl). The Component can have only one template. You can either use inline template or external template and not both

**Structural:** [ngFor](https://www.tektutorialshub.com/angular-2-ngfor-directive/), [ngSwitch Directive](https://www.tektutorialshub.com/angular-2-ngswitch-directive/), [ngIf Directive](https://www.tektutorialshub.com/angular-2-ngif-directive/)

**Attribute Directive:** [ngClass Directive](https://www.tektutorialshub.com/angular-2-ngclass-directive/), [ngStyle Directive](https://www.tektutorialshub.com/angular-2-ngstyle-directive/)

**ngFor:**

The ngFor is an Angular structural directive, which repeats a portion of HTML template once per each item from an iterable list (Collection). The ngSwitch allows us to Add/Remove DOM Element.It is similar to switch statement of C#.  The ngIf allows us to Add/Remove DOM Element, ngFor directive to tell angular to repeat that block of HTML elements for each item in the list.

Local Variables in ngFor

ngFor also provides several values to help us manipulate the collection. We can assign the values of these exported values to the local variable and use it in our template

The list of exported values provided by **ngFor directive**

index

This is a zero-based index and set to the current loop iteration for each template context.

first

This is a boolean value, set to true if the item is the first item in the iteration

last

This is a boolean value, set to true if the item is the last item in the iteration

even

This is a boolean value, set to true if the item is the even-numbered item in the iteration

odd

This is a boolean value, set to true if the item is the odd-numbered item in the iteration

<tr \*ngFor="let movie of movies; let i=index; let o= odd; let e=even;"

[ngClass]="{ odd: o, even: e }">

<td> {{i}} </td>

<td>{{movie.title}}</td>

<td>{{movie.director}}</td>

<td>{{movie.cast}}</td>

<td>{{movie.releaseDate}}</td>

</tr>

#### ngSwitch:

ngSwitch directive is applied to the container element with a switch expression.

*ngSwitchCase:*

The inner elements are placed inside the container element. The ngSwitchCase directive is applied to the inner elements with a match expression.Whenever the value of the match expression matches the value of the switch expression , the corresponding inner element is added to the DOM. All other inner elements are removed from the DOM

If there is more than one match, then all the matching elements are added to the DOM

#### ngSwitchDefault:

You can also apply the ngSwitchDefault directive. The element with ngSwitchDefault is displayed only if no match is found. The inner element with ngSwitchDefault can be placed anywhere inside the container element and not necessarily at the bottom. If you add more than one ngSwitchDefault directive, all of them are displayed.  
Any elements placed inside the container element, but outside the ngSwitchCase or ngSwitchDefault elements are displayed as it is.

Example:

<div>

Input string : <input type=*'text'* **[(ngModel)]** =*"num"*/>

<div **[ngSwitch]**=*num*>

<div **\*ngSwitchCase**=*"'1'"*>one dive</div>

<div **\*ngSwitchCase**=*"'2'"*>two dive</div>

<div **\*ngSwitchCase**=*"'3'"*>three dive</div>

<div **\*ngSwitchCase**=*"'4'"*>four dive</div>

<div **\*ngSwitchCase**=*"'one'"*>test case for string</div>

<div **\*ngSwitchDefault**>Default no</div>

</div>

</div>

**\*ngIf:**

<div class="row">

Show <input type="checkbox" [(ngModel)] ="showMe"/>

</div>

<div class='row'>

<div \*ngIf="showMe">

ShowMe is checked

</div>

<div \*ngIf="!showMe">

ShowMe is unchecked

</div>

</div>

Examples:

|  |
| --- |
| <div \*ngIf="title; else login"> The user is logged in.</div>  <ng-template #login>Please login to continue.</ng-template>  Title = ture;  <div \*ngIf="!title; else login"> |

## NgIf vs hidden

At first sight, the angular ngIf directive seems to just the same thing as binding to the HTML5 "hidden" attribute.   
So why do we need two ways to do the same thing?

<p [hidden]="!show">  
 Show this only if "show" is true  
</p>

Because they actually do two completely different things.  
  
While the hidden attribute is literally hiding the selected part of the DOM, just like the CSS "display: none" property, the element still sit on the DOM.   
They are just invisible.  
  
Angulars' ngIf directive, on the other hand, is completely removing the selected part from the DOM.  
The great advantage of that is, that this method is not interfering with any CSS-Style-sheets at all. It is simply removing anything.

**Working with "else"**

Just as we are used to from other programming languages, the angular ngFor directive does also allow us to declare an else-block.

This block is shown if the statement defined in the main block happens to be false.

<div \*ngIf="show; else notShow">

<p>

Show this only if "show" is true

</p>

</div>

<ng-template #notShow>

<p>

Show this only if "show" is not true

</p>

</ng-template>

**NgIf also has a "then":**

<ng-template

\*ngIf="show;then showBlock; else notShowBlock">

</ng-template>

<ng-template #showBlock>

<p>

Show this only if "show" is true

</p>

</ng-template>

<ng-template #notShowBlock>

<p>

Show this only if "show" is not true

</p>

</ng-template>

**Attribute Directive:**

**ngClass:**

The Angular ngClass Directive is an Angular Attribute Directive, which allows us to add or remove CSS classes to an HTML element. Using ngClass you can create dynamic styles in HTML pages.

Syntax:

<element [ngClass]="expression">... </element>

ngClass attribute is applied to an element of DOM. It is then bound to an expression. The ngClass attribute evaluates the expression and changes the class attribute of the element to which it is attached.

There are three different ways you can use the expression. The expression can be evaluated either to a string, array or an object

## Binding to a String:

You can use the String as expression and bind it to directly to the ngClass attribute. If you want to assign multiple classes, then separate each class with space as shown below.

### Syntax of ngClass<element [ngClass]="'cssClass1 cssClass2'">...</element>

Ex:

.red {

color: red;

}

.size20 {

font-size: 20px;

}

Html:

<div [ngClass]="'red size20'">

Red Text with Size 20px

</div>

You can also use the ngClass without a square bracket. In that case, the expression is not evaluated but assigned directly to the class attribute. We also need to remove the double quote around the expression as shown below.

<div class="row">

<div ngClass='red size20'>

Red Text with Size 20px : as string

</div>

</div>

## Binding to an array

You can achieve the same result by using an array instead of a string as shown below. The syntax for ngClass array syntax is as shown below

<element [ngClass]="['cssClass1', 'cssClass2']">...</element>

Ex:

<div [ngClass]="['red','size20']">

Red Text with Size 20px

</div>

## Binding to an object

You can also bind the ngClass to an object. Each property name of the object acts as a class name and is applied to the element if it is true. The syntax is as shown below

<element [ngClass]="{'cssClass1': true, 'cssClass2': true}">...</element>

<div class="row">

<div [ngClass]="{'red':true,'size20':true}">

Red Text with Size 20px : as object

</div>

</div>

In the above example, an object is bound to the ngClass. The object has two properties red and size20. The property name is assigned to the div element as a class name

### **Using Variable from the Component class**

Add the following code to your component class. You can dynamically change the CSS classes in your component to control the look of your template

cssStringVar: string= 'red size20';

<div class="row">

<div [ngClass]="cssStringVar">

Red Text with Size 20px : from component

</div>

</div>

### **Using object in Component class**

Create a class as follows and add it to your component

class CssClass {

red: boolean= true;

size20: boolean= true;

}

cssClass: CssClass = new CssClass();

<div class="row">

<div [ngClass]="cssClass">

Red Text with Size 20px : from component as object

</div>

</div>

**Other:**

obj :cssDemoClass;

clickme() : cssDemoClass {

alert("clicked")

this.obj = new cssDemoClass;

this.obj.red=true

this.obj.size12=true

return this.obj;

}

}

**ngStyle:**

The **ngStyle directive** allows you to modify the style of an HTML element using the expression. In this tutorial, we will look how to use ngStyle with few examples. Using ngStyle you can dynamically change the style of your HTML element. You also can bind these properties to components properties.

<element [ngStyle]="{'styleName': styleExp}">...</element>

we units (for example px, em) are prefixed to the styleName.

## Example of ngStyle

Initialise a variable color and add it to your component

color: string= 'red';

<input [(ngModel)]="color" />

<div [ngStyle]="{'color': color}">

Change my color

</div>

<input [(ngModel)]="size" />

<div [ngStyle]="{'font-size.px': size}">

Change my size

</div>

### Using object from Controller

class StyleClass {

color: string= 'blue';

'font-size.px': number= 20;

'font-weight': string= 'bold';

}

styleClass: StyleClass = new StyleClass();

<div [ngStyle]="styleClass">

Change my size & Color

</div>

### **Pipes:**

The Angular pipes are used to Transform the Data. For Example, the Date pipe formats the date according to locale rules. We can pass arguments to pipe and chain pipes. The Angular also allows us to create the Custom Pipe

1. [Angular Pipes](https://www.tektutorialshub.com/angular-2-pipes/)
2. [Angular Custom Pipes](https://www.tektutorialshub.com/angular-2-custom-pipes/)

Angular 2 Pipes helps us to format or transform data to display in our template. They are similar to Filters in AngularJS. In this Tutorial, we are going to look at what is pipes are & how to use them. We will see how to pass arguments to the pipe and how to chain pipes. We are also going to look at the few list of angular 2 built-in pipes like

1. currency pipe
2. date pipe
3. number pipe
4. percent pipe
5. decimal pipe
6. slice pipe
7. lowercase
8. uppercase

**Angular 2 Pipes:**

There are many circumstances where we may have to change the appearance of the data before presenting it the user. The most common examples are dates. That is where Angular 2 Pipes comes handy. They take the data as input and transforms that data to get the desired output.

**Date Pipe:**

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

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

@Component({

selector: 'app-root',

template:`<p>medium : {{toDate | date:'medium'}} </p>

<p>short : {{toDate | date:'short'}} </p>

<p>fullDate : {{toDate | date:'fullDate'}} </p>

<p>longDate : {{toDate | date:'longDate'}} </p>

<p>mediumDate : {{toDate | date:'mediumDate'}} </p>

<p>shortDate : {{toDate | date:'shortDate'}} </p>

<p>mediumTime : {{toDate | date:'mediumTime'}} </p>

<p>dd-MM-y : {{toDate | date:'dd-MM-y'}} </p>

<p>dd-MM-yy HH:mm : {{toDate | date:'dd-MM-yy HH:mm'}} </p>`

})

export class AppComponent

{

title: string = 'Angular 2 pipes Example' ;

toDate: Date = new Date();

}

template:`<p>Unformatted :{{msg}} </p>

<p>Uppercase :{{msg | uppercase}} </p>

<p>Lowercase :{{msg | lowercase}} </p>`

### **SlicePipe:**

Creates a new List or String containing a subset (slice) of the string or array. This Pipe uses the JavaScript API [Array.prototype.slice()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) and [String.prototype.slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice).

array\_or\_string\_expression | slice:start[:end]

**Where**

**array\_or\_string\_expression** is the string to slice

**slice** is the name of the pipe

**start** is the start position/index from where the slicing will start

**end**is the ending index/position in the array/string

The slice pipes take two arguments. The first argument*start* is the starting index of the string/array. The second argument **end** is the ending index of the string/array. If the start or end index is negative then the index is counted from end of the string/array

template:`<p>Complete String :{{msg}} </p>

<p>Example 1 :{{msg | slice:11:20}} </p>

<p>Example 2 :{{msg | slice:-9}} </p>`

mystr = "abcdefghijk"

myarr = [11,22,33,44,55,66,77,88]

<h1>

{{mystr | slice :3:7}}<br>

{{mystr | slice :3:-2}}<br>

{{mystr | slice :6}}<br>

{{mystr | slice :-6}}<br>

</h1>

<p>array slice</p>

<h1>

{{myarr | slice :3:7}}<br>

{{myarr | slice :3:-2}}<br>

{{myarr | slice :6}}<br>

{{myarr | slice :-6}}<br>

</h1>

**O/P:**

# defg defghi ghijk fghijk

array slice

# 44,55,66,77 44,55,66 77,88 33,44,55,66,77,88

**Json Pipe:**

objJson : Object = {foo: 'cubs',baz:'sox',nested:{xvz:3,numbers:[1,2,3,4,5]}};

<p>without pipe json</p>

<p>{{objJson}}</p>

<p>with pipe json</p>

<p>{{objJson | json}}</p>

<p>{{value1 | currency :'USD':true :'3.2-3'}}</p>

<p>{{value2 | currency :'INR':true : '3.2-3'}}</p>

without pipe json

[object Object]

with pipe json

{ "foo": "cubs", "baz": "sox", "nested": { "xvz": 3, "numbers": [ 1, 2, 3, 4, 5 ] } }

**DecimalPipe:**

DecimalPipe is an angular Pipe API and belongs to CommonModule. DecimalPipe is used to format a number as decimal number according to locale rules. It uses number keyword with pipe operator. Find the syntax.

number\_expression | number[:digitInfo]

Finally we get a decimal number as text. Find the description.

number\_expression: An angular expression that will give output a number.

number : A pipe keyword that is used with pipe operator.

digitInfo : It defines number format.

Now we will understand how to use digitInfo. The syntax for digitInfo is as follows.

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Find the description.

minIntegerDigits : Minimum number of integer digits. Default is 1.

minFractionDigits : Minimum number of fraction digits. Default is 0.

maxFractionDigits : Maximum number of fraction digits. Default is 3.

Now find some sample examples.

1. Using default format:

minIntegerDigits = 1

minFractionDigits = 0

maxFractionDigits = 3

Now find a number that will be formatted.

num1 = 12.638467846; Now use Decimal Pipe.

{{num1 | number}}

Find the output.

12.638 We will observe that fraction digit has been truncated to count 3, because maximum fraction digit is only 3.

2. Use format '3.2-5' :

minIntegerDigits = 3

minFractionDigits = 2

maxFractionDigits = 5

Now find the number that will be formatted.

num1 = 12.638467846; Now use Decimal Pipe.

{{num1 | number:'3.2-5'}}

Find the output.

012.63847 In our number, integer part is 12 that is of 2 digits, so adding 0 as prefix, it has been of 3 digits that is 012. This is because minimum integer digit required is 3.

3. Format '3.2-5'

minIntegerDigits = 3

minFractionDigits = 2

maxFractionDigits = 5

Now find the number that will be formatted.

num2 = 0.5; Now use Decimal Pipe.

{{num2 | number:'3.2-5'}}

Find the output.

000.50 Now find the component used in our example.

decimalpipe.component.ts

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

@Component({

selector: 'decimal-app',

template: `

<h3>Decimal Pipe</h3>

<div>

<p> {{num1 | number}} </p>

<p> {{num1 | number:'3.2-5'}} </p>

<p> {{num2 | number:'3.2-5'}} </p>

<p> {{num1 \* num2 | number:'1.3-6'}} </p>

</div>

`

})

export class DecimalPipeComponent {

num1: number = 12.638467846;

num2: number = 0.5;

}

Find the output.

Decimal Pipe

12.638

012.63847

000.50

6.319234

**PercentPipe:**

Angular PercentPipe is an angular Pipe API that formats a number as a percentage according to locale rules. It belongs to CommonModule. Find the syntax.

number\_expression | percent[:digitInfo]

Find the description.

number\_expression: An angular expression that will give output a number.

percent : A pipe keyword that is used with pipe operator and it converts number into percent.

digitInfo: It defines a percentage format. We have described the use of digitInfo in DecimalPipe section. It is used with following syntax.

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Now find some sample examples.

1. Using default format:

minIntegerDigits = 1

minFractionDigits = 0

maxFractionDigits = 3

Now find a number that will be changed into percentage.

num1 = 2.5; Now use Percent Pipe

{{num1 | percent}}

Find the output.

250%

2. Use format '2.2-5'

minIntegerDigits = 2

minFractionDigits = 2

maxFractionDigits = 5

Now find the number that will be changed into percentage.

num1 = 2.5; Now use Percent Pipe.

{{num1 | percent:'2.2-5'}}

Find the output.

250.00% We will observe that there is two digits in fraction part. This is because minimum fraction digits required is 2.

Now find the component used in our example.

percentpipe.component.ts

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

@Component({

selector: 'percent-app',

template: `

<h3>Percent Pipe</h3>

<div>

<p> {{num1 | percent}} </p>

<p> {{num1 | percent:'2.2-5'}} </p>

<p> {{num2 | percent:'1.2-5'}} </p>

<p> {{num1 \* num2 | percent:'1.2-3'}} </p>

</div>

`

})

export class PercentPipeComponent {

num1: number = 2.5;

num2: number = 0.5;

}

Find the output.

Percent Pipe

250%

250.00%

50.00%

125.00%

**CurrencyPipe:**

CurrencyPipe is an angular Pipe API that formats a number as currency using locale rules. It belongs to CommonModule. CurrencyPipe uses currency keyword with pipe operator to format a number into currency format. Find the syntax.

number\_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]

Find the description.

number\_expression : An angular expression that will give output a number.

currency : A pipe keyword that is used with pipe operator. It formats a number into currency format.

currencyCode : This is the currency code such as INR for Indian rupee, USD for US dollar. Default is USD.

symbolDisplay : Default is false. But if we assign true then it will display currency symbol such as $ for dollar.

digitInfo: It defines a currency format. We have described the use of digitInfo in DecimalPipe section. It is used with following syntax.

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Find some sample examples.

1. Using default format:

currencyCode = USD

symbolDisplay = false

minIntegerDigits = 1

minFractionDigits = 0

maxFractionDigits = 3

Now find a number that will be changed into currency.

cur1 = 0.25;

Now use Currency Pipe.

{{cur1 | currency}} Find the output.

USD0.25

2. Use format '2.2-4'

currencyCode = USD

symbolDisplay = true

minIntegerDigits = 2

minFractionDigits = 2

maxFractionDigits = 4

Now find a number that will be changed into currency.

cur2 = 10.263782; Now use Currency Pipe.

{{cur2 | currency:'USD':true:'2.2-4'}}

Find the output.

$10.2638 Now find the component used in our example.

currencypipe.component.ts

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

@Component({

selector: 'currency-app',

template: `

<h3> Currency Pipe</h3>

<div>

<p> {{cur1 | currency:'INR':false}} </p>

<p> {{cur2 | currency:'INR':false:'1.2-4'}} </p>

<p> {{cur1 | currency}} </p>

<p> {{cur2 | currency:'USD':true:'2.2-4'}} </p>

</div>

`

})

export class CurrencyPipeComponent {

cur1: number = 0.25;

cur2: number = 10.263782;

}

Find the output.

Currency Pipe

INR0.25

INR10.2638

USD0.25

$10.2638

**Pipes Custom:**

**Step 1:** We need to import the Pipe & PipeTransform libraries from Angular 2. These libraries are part of the Angular 2 Core

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

**Step 2:** Create class and implements **PipeTransform interface.**Every pipe we built must implement the **PipeTransform interface.**

The [PipeTransform interface](https://angular.io/docs/ts/latest/api/core/index/PipeTransform-interface.html) has only one method known as the transform. This interface takes the value being piped as the first argument. It takes the variable number of optional arguments of any type. It returns the final transformed data.

We have implemented transform method, which takes two arguments. The first is **Value** and the second is the **Unit**. The unit expects either C (Convert to Celsius) or F ( convert to Fahrenheit). It converts the value received to either to celsius or to Fahrenheit based on the Unit.

@pipe({

name: 'tempConverter'

})

We decorate the class with a @pipe decorator.[@pipe decorator](https://angular.io/docs/ts/latest/api/core/index/Pipe-interface.html) is what tells Angular 2 that the class is a Pipe. @Pipe decorator also provides the **metadata to pipe class**. In the above example, we have provided name metadata, which is the name of the pipe, which we will use in our template

**Step 3:** Custom pipes have to be registered in the "app.module.ts".

Example:

<p> {{sqrtNum | sqrtPipe}} </p>

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

@Pipe( {

name: 'sqrtPipe'

} )

export class PipesComponent implements PipeTransform {

transform(value : number ): number {

return Math.sqrt( value )

}

}

## Passing data to a child/nested component

In Angular, the Parent Component can communicate with the child component by setting its Property. To do that the Child component must expose its properties to the parent component. The Child Component does this by using the @Input decorator

In the Child Component

1. Import the @Input module from @angular/Core Library
2. Mark those property, which you need data from parent as input property using @Input decorator

**In the Parent Component**

1. Bind the Child component property in the Parent Component when instantiating the Child

## @Input Decorator

The @Input Decorator is used to configure the input properties of the component. This decorator as also supports change tracking.

When you mark a property as input property, then the Angular injects values into the component property using [Property Binding](https://www.tektutorialshub.com/data-binding-angular-2/#Property-Binding). The Property Binding uses the [] brackets. The Binding Target (Property of the child component) is placed inside the square brackets. The Binding source is enclosed in quotes. [Property binding](https://www.tektutorialshub.com/data-binding-angular-2/#Property-Binding) is one way from Component to the Target in the template

The parent component can pass data to the child component using @Input binding. This binding is defined in the child component to accept the value passed from the parent component.

|  |
| --- |
| import { Component} from '@angular/core';  @Component({  selector: 'app-root',  template: `  <h1>Welcome to {{title}}!</h1>  <button (click)="increment()">Increment</button>  <button (click)="decrement()">decrement</button>  <child-component [count]=Counter></child-component>` ,  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'Component Interaction';  Counter = 5;  increment() {  this.Counter++;  }  decrement() {  this.Counter--;  }  }  import { Component, Input } from '@angular/core';  @Component({  selector: 'child-component',  template: `<h2>Child Component</h2>  current count is {{ count }}  `  })  export class ChildComponent {  @Input() count: number;  } |

## Various ways to use @Input Decorator

We used input @Input annotation to mark the property in child component as input property. There are two ways you can do it Angular.

1. Using the @Input decorator to decorate the class property
2. Using the input array meta data of the component decorator

### Using the @Input decorator to decorate the class property

|  |
| --- |
| export class ChildComponent {  @Input() count: number;  } |

### Using the input array metadata of the component decorator

### The same result can be achieved by using Input array of the @Component decorator as shown below

|  |
| --- |
| @Component({  selector: 'child-component',  inputs: ['count'],  template: `<h2>Child Component</h2>  current count is {{ count }}  `  })  export class ChildComponent {} |

**Aliasing input Property**

You can Alias the input property and use the aliased name the parent component as shown below

|  |
| --- |
| export class ChildComponent {  @Input('MyCount') count: number;  } |

## Detecting the Input changes

We looked at how to pass the data from parent to the child using @Input annotation and property binding.

Passing the data to child component is not sufficient, the child Component needs to know when the input changes so that it can act upon it.

There are two ways of detecting when input changes in the child component in Angular

1. Using OnChanges LifeCycle Hook
2. Using Input Setter

Let us look at both the methods in detail

**Using OnChanges LifeCycle Hook:**

ngOnChanges is a lifecycle hook, which angular fires when it detects changes to data bound input property. This method receives a SimpeChanges object, which contains the current and previous property values. We can Intercept input property changes in the child component using this hook.

**How to use ngOnChanges for Change Detection**

1. Import the OnChanges interface, SimpleChanges, SimpleChange from @angule/core library.
2. Implement the ngOnChanges() method. The method receives the SimpleChanges object containing the changes each input property.

Let us update our Child component to use the OnChanges hook

|  |
| --- |
| import { Component, Input, OnChanges, SimpleChanges, SimpleChange } from '@angular/core';  @Component({  selector: 'child-component',  template: `<h2>Child Component</h2>  current count is {{ count }}  `  })  export class ChildComponent implements OnChanges {  @Input() count: number;  ngOnChanges(changes: SimpleChanges) {  for (let property in changes) {  if (property === 'count') {  console.log('Previous:', changes[property].previousValue);  console.log('Current:', changes[property].currentValue);  console.log('firstChange:', changes[property].firstChange);  }  }  }  }  This method receives the all the changes made to the input properties as SimpleChanges object. The SimpleChanges object whose keys are property names and values are instances of SimpleChange. |

| PROPERTY NAME | DESCRIPTION |
| --- | --- |
| previousValue:any | Previous value of the input property. |
| currentValue:any | New or current value of the input property. |
| FirstChange():boolean | Boolean value, which tells us whether it was the first time the change has taken place |

**Using Input Setter**

We can use the property getter and setter to detect the changes made to the input property as shown below

In the Child Component create a private property called \_count

private \_count = 0;

|  |
| --- |
| @Input()  set count(count: number) {  this.\_count = count;  console.log(count);  }  get count(): number { return this.\_count; } |

## Pass data to parent component:

There are three ways in which parent component can interact with the child component

1. Parent Listens to Child Event
2. Parent uses Local Variable to access the child
3. Parent uses a @ViewChild to get reference to the child component

|  |
| --- |
| import { Component, Input, Output, EventEmitter } from '@angular/core';  @Component({  selector: 'child-component',  template: `<h2>Child Component</h2>  <button (click)="increment()">Increment</button>  <button (click)="decrement()">decrement</button>  current count is {{ count }}  `  })  export class ChildComponent {  @Input() count: number;  @Output() countChanged: EventEmitter<number> = new EventEmitter();  increment() {  this.count++;  this.countChanged.emit(this.count);  }  decrement() {  this.count--;  this.countChanged.emit(this.count);  }  }  import { Component} from '@angular/core';  @Component({  selector: 'app-root',  template: `  <h1>Welcome to {{title}}!</h1>  <p> current count is {{ClickCounter}} </p>  <child-component [count]=Counter (countChanged)="countChangedHandler($event)"></child-component>` ,  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'Component Interaction';  Counter = 5;  countChangedHandler(count: number) {  this.Counter = count;  console.log(count);  }  } |

**Parent uses local variable to access the Child in Template:**

Parent Template can access the child component properties and methods by creating the template reference variable

|  |
| --- |
| import { Component} from '@angular/core';  @Component({  selector: 'child-component',  template: `<h2>Child Component</h2>  current count is {{ count }}  `  })  export class ChildComponent {  count = 0;  increment() {  this.count++;  }  decrement() {  this.count--;  }  }  import { Component} from '@angular/core';  @Component({  selector: 'app-root',  template: `  <h1>{{title}}!</h1>  <p> current count is {{child.count}} </p>  <button (click)="child.increment()">Increment</button>  <button (click)="child.decrement()">decrement</button>  <child-component #child></child-component>` ,  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'Parent interacts with child via local variable';  } |

The Template Reference variable is created, when you use #<varibaleName> and attach it to a DOM element. You can then, use the variable to reference the DOM element in your Template

**Parent uses a @ViewChild() to get reference to the Child Component:**

Injecting an instance of the child component into the parent as a @ViewChild is the another technique used by the parent to access the property and method of the child component

The @ViewChild decorator takes the name of the component/directive as its input. It is then used to decorate a property. The Angular then injects the reference of the component to the Property

|  |
| --- |
| import { Component, ViewChild } from '@angular/core';  import { ChildComponent } from './child.component';  @Component({  selector: 'app-root',  template: `  <h1>{{title}}</h1>  <p> current count is {{child.count}} </p>  <button (click)="increment()">Increment</button>  <button (click)="decrement()">decrement</button>  <child-component></child-component>` ,  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'Parent calls an @ViewChild()';  @ViewChild(ChildComponent) child: ChildComponent;  increment() {  this.child.increment();  }  decrement() {  this.child.decrement();  }  } |

**Content projection:**

Projection is a very important concept in Angular. It enables developers to build reusable components and make applications more scalable and flexible. To illustrate that, suppose we have a ChildComponent like:

Content projection allows you to insert a shadow DOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using the concept of content projection. In Angular, you achieve content projection using **< ng-content >< /ng-content >.**You can make reusable components and scalable applications by properly using content projection.

You can select a particular slot for projection using the <ng-content> selector. There are four types of selectors:

Project using tag selector.

Project using class selector.

Project using id selector.

Project using attribute selector.

You can use the tag selector for multi-slot projection as shown in the listing below:

Example:

|  |
| --- |
| Example 1:  <app-greet>  <h1>Content project elelments</h1>  <button>demo</button>  </app-greet>  <p>  greet works!  </p>  <div class = *"container"*>  <ng-content></ng-content>  </div>  Example 2:  <app-greet>  <h1 class=*"headerTxt"*> header text element </h1>  <button btnp>Click button</button>  </app-greet>  **import** { Component } **from** '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  **export** **class** AppComponent {  // title = 'app';  }  Great html:  <div class = "container">  <ng-content select=.headerTxt></ng-content>  {{demo}}  <ng-content select="[btnp]"></ng-content>  </div>  greatComponet:  **import** { Component, OnInit } **from** '@angular/core';  @Component({  selector: 'app-greet',  templateUrl: './greet.component.html',  styleUrls: ['./greet.component.css']  })  **export** **class** GreetComponent **implements** OnInit {  demo:**string** = "hello great message";  **constructor**() { }  ngOnInit() {  }  } |

<https://dzone.com/articles/simplifying-content-projection-in-angular>

**LifeCycle hook:**

**constructor**

This is invoked when Angular creates a component or directive by calling new on the class.

**ngOnChanges**

Invoked every time there is a change in one of th input properties of the component.

**ngOnInit**

Invoked when given component has been initialized.  
This hook is only called once after the first ngOnChanges

**ngDoCheck**

Invoked when the change detector of the given component is invoked. It allows us to implement our own change detection algorithm for the given component.

#### Important

ngDoCheck and ngOnChanges should not be implemented together on the same component.

**ngOnDestroy**

This method will be invoked just before Angular destroys the component.  
Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.

### [**Hooks for the components children**](https://codecraft.tv/courses/angular/components/lifecycle-hooks/#_hooks_for_the_components_children)

These hooks are only called for components and not directives.

**ngAfterContentInit**

Invoked after Angular performs any content projection into the components view (see the previous lecture on Content Projection for more info).

**ngAfterContentChecked**

Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.

**ngAfterViewInit**

Invoked when the component’s view has been fully initialized.

**ngAfterViewChecked**

Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.

The best place to initialise your components is in the ngOnInit lifecycle hook and not the constructor because only at this point have any input property bindings been processed.   
The reason we use ngOnInit and not ngOnChanges to initialise a component is that ngOnInit is only called once whereas ngOnChanges is called for every change to the input propertiess

## Angular 2 Forms Module:

Angular 2 forms module provides all the above services out of the box. It uses the [two-way data binding](https://www.tektutorialshub.com/data-binding-angular-2/) & [event binding](https://www.tektutorialshub.com/data-binding-angular-2/) bind the form field to the [Angular 2 component](https://www.tektutorialshub.com/angular2-components/) class. It tracks changes made to the form fields so that we can respond accordingly. The Angular forms provide the built-in validators to validate the inputs. You can create your own custom validator. It presents the validation errors to the user. Finally, it encapsulates all the input fields into an object structure when the user submits the form.

Angular 2 takes two approaches to building the forms. One is [**Template driven forms**](https://www.tektutorialshub.com/angular-template-driven-forms/) approach and another one is **model driven forms** approach

### **Template driven forms approach**

In [Template driven approach](https://www.tektutorialshub.com/angular-template-driven-forms/) is the easiest way to build the Angular 2 forms. The logic of the form is placed in the template. The approach here is similar to what we did in Angular 1

### **Model-driven forms approach**

In Model driven approach, the logic of the form is placed in the component. The Model driven approach has more benefits as it makes the testing of the component easier.

In this approach, the representation of the form is created in the component class. This form model is then bound to the HTML elements. This done using the special markups.

Template-driven forms in Angular 2 allows us to create sophisticated looking forms easily without writing any java script code. The model-driven forms are created in component class, where Form fields are created as properties of our component class.  This makes them easier to test.

**ngForm and ngModel:**

ngForm and ngModel are Angular directives that are essential to creating template-driven forms. Let's start with ngForm first. Here is an excerpt about ngForm from the Angular docs.

The NgForm directive supplements the form element with additional features. It holds the controls you created for the elements with an ngModel directive and name attribute, and monitors their properties, including their validity. It also has its own valid property which is true only if every contained control is valid.

First, update the form with the ngForm directive:

app/signup-form/signup-form.component.html

<form

class="form-horizontal"

#signupForm = "ngForm">

..

</form>

#signupForm is a template reference variable that refers to the ngForm directive which governs the entire form. The example below demonstrates the use of a ngForm reference object for validation.

app/signup-form/signup-form.component.html

<button

type="submit"

class="btn btn-success"

[disabled]="!signupForm.form.valid">

Submit

</button>

Here, signupForm.form.valid will return false unless all the form elements pass their respective validation checks. The submit button will be disabled until the form is valid.

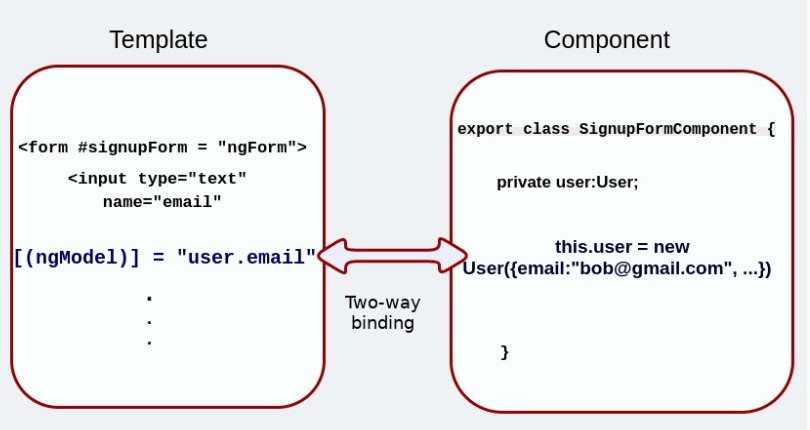
As for binding the template and the model, there are plenty of ways to do this, and ngModel has three different syntaxes to tackle this situation. They are:

* [(ngModel)]
* [ngModel]
* ngModel

Let's start with the first one.

**Two-Way Binding Using [(ngModel)]:**

[(ngModel)] performs two-way binding for reading and writing input control values. If a [(ngModel)]directive is used, the input field takes an initial value from the bound component class and updates it back whenever any change to the input control value is detected (on keystroke and button press). The image below describes the two-way binding process better.



Here is the code for the email input field:

<div class="form-group">

<label for="inputEmail">Email</label>

<input type="text"

[(ngModel)] = "user.email"

id="inputEmail"

name="email"

placeholder="Email">

</div>

[(ngModel)] = "user.email" binds the user's email property to the input value. I've also added a name attribute and set name="email". This is important, and you will get an error if you've not declared a name attribute while using ngModel.

The two-way binding [(ngModel)] syntax helps you build forms effortlessly. However, it has certain drawbacks; hence, there is an alternate approach that uses ngModel or [ngModel].

**Adding ngModel to the Mix:**

When ngModel is used, we are in fact responsible for updating the component property with the input control values and vice versa. The input data doesn't automatically flow into the component's user property.

So replace all instances of [(ngModel)] = " " with ngModel. We will keep the name attribute because all three versions of ngModel need the name attribute to work.

<div class="form-group">

<label for="inputEmail">Email</label>

<input type="text"

ngModel

id="inputEmail"

name="email"

placeholder="Email">

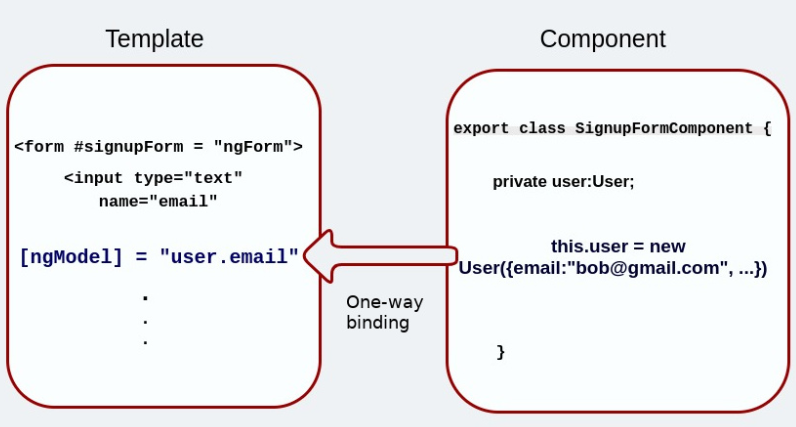
</div>

With ngModel, the value of the name attribute will become a key of the ngForm reference object signupForm that we created earlier. So, for example, signupForm.value.email will store the control value for the email id.

Replace {{user | json}} with {{signupForm.value | json }} because that's where all the state is stored right now.

**One-Way Binding Using [ngModel]:**

What if you need to set the initial state from the bound class component? That's what the [ngModel] does for you



Here the data flows from the model to the view. Make the following changes to the syntax to use one-way binding:

app/signup-form/signup-form.component.html

<div class="form-group">

<label for="inputEmail">Email</label>

<input type="text"

[ngModel] = "user.email"

id="inputEmail"

name="email"

placeholder="Email">

</div>

So which approach should you choose? If you're using [(ngModel)] and ngForm together, you will eventually have two states to maintain—user and signupForm.value—and that could be potentially confusing.

I will recommend using the one-way binding method instead. But that's something for you to decide.

**Track control state and validity with ngModel**

Using ngModel in a form gives you more than just two-way data binding. It also tells you if the user touched the control, if the value changed, or if the value became invalid.

The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. You can leverage those class names to change the appearance of the control.

|  |  |  |
| --- | --- | --- |
| State | Class if true | Class if false |
| The control has been visited. | ng-touched | ng-untouched |
| The control's value has changed. | ng-dirty | ng-pristine |
| The control's value is valid. | ng-valid | ng-invalid |

## Model Driven Forms:

## With reactive forms, you will be able to create and manipulate form control objects directly in the Component. Since the component class has access to the form control structure and the data model, you can push data model values into the form controls as well as pull values that have been changed by the user. The component is able to observe changes in the form control state and react to them. This is specially useful for showing a validation message.

### **Enabling Model Driven Forms**

We need to import **ReactiveFormsModule** instead of **FormModule** to make use of Model Driven Forms.We should also add the **ReactiveormsModule** to the *imports metadata property array*.

**Building the Model**

In the [Template driven approach](https://www.tektutorialshub.com/angular-template-driven-forms/), we used **ngModel** & **ngModelGroup** directive on the HTML elements, which created the FormGroup & FormControl instances behind the scene.

In the Model Driven approach, it is our responsibility to create the **FormGroup** and **FormControl**

**FormControl** encapsulates the state of a single form element in our form. It stores the value and state of the form element and helps us to interact with them using properties & methods

**FormGroup** represents a collection of **FormControls**. In fact, an angular form is a **FormGroup**.

In react forms we create instance of formControl and FormGroup classes.

Both FormControl and FormGroup classes inherit from AbstractControl bases class.

The AbstractControl classs has properties that helps us track both formControl and FormGroup

Value state.

Form Control tracks the value and state of an individual html element.

**Access elements from control and formGroup:**

* employeeForm.controls.fullname.value
* employeeForm.get(‘fullname’).value

**SetValue:**

Use setValue method to update all form controls

We are not update sub set of form controls, otherwise we will get exception.

**patchValue:**

Use patchValue method to update sub-set of form controls.

Also, update all the form controls.

**Example:**

|  |
| --- |
| empForm: FormGroup;  **constructor**(){}  ngOnInit() {  **this**.empForm = **new** FormGroup({  fullName: **new** FormControl(),  email: **new** FormControl(),  skills : **new** FormGroup({  skillName: **new** FormControl(),  experienceInYears: **new** FormControl(),  proficiency: **new** FormControl()    })    });  }  onSubmit() : **void** {  console.info("\*\*\*\*form value\*\*\*\*: ",**this**.empForm.value)  //console.info("\*\*\*\*skills \*\*\*\*: ",this.empForm.skills.value)  // console.info("\*\*\*\*skillName name\*\*\*\*: ",this.empForm.controls.skillName.value)  //console.info("\*\*\*\*skillName name\*\*\*\*: ",this.empForm.get('skillName').value)  }    onLoadDataClick() : **void** {  console.info("\*\*\*\*onLoadDataClick executed\*\*\*\*\*\*\*\*\*\*\*\*\*")  **this**.empForm.setValue({  fullName: 'Nagendra',  email: 'nagedra.kldm@gmail.com',  skills: {  skillName: 'java teche',  experienceInYears: '5',  proficiency: 'expert'  }  })  } |

FormBuilder reduces the amount of boilerplate code we have to write to build complex reactive forms.

**Difference:**

The major difference between these two approaches of creating forms? In reactive forms you do not use directives such as ngModel, required, ngForm etc. You create all controls and their validations in the component class. Reactive forms are easy to test and maintain, so in this post we will learn to create a basic Reactive Form, using FormControl, FormGroup, FormBuilder class, and adding validations.

There is an important difference between them and that is: Reactive forms are synchronous while Template-driven forms are asynchronous.

Examples:

|  |
| --- |
| **Html file:**  <div class="container">      <br />      <form (ngSubmit)='loginUser()' [formGroup]='loginForm' novalidate class="form">          <input formControlName='email' type="text" class="form-control" placeholder="Enter Email" />          <div class="alert  alert-danger" \*ngIf="loginForm.get('email').hasError('required') && loginForm.get('email').touched">              Email is required          </div>          <input formControlName='password' type="password" class="form-control" placeholder="Enter Password" />          <div class="alert  alert-danger" \*ngIf=" !loginForm.get('password').valid && loginForm.get('email').touched">              Password is required and should less than 10 characters          </div>          <button [disabled]='loginForm.invalid' class="btn btn-default">Login</button>      </form>  </div>  import { Component, OnInit } from '@angular/core';  import { FormGroup, FormControl, FormArray, Validators } from '@angular/forms';  @Component({      selector: 'app-root',      templateUrl: './app.component.html',      styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {        loginForm: FormGroup;        ngOnInit() {            this.loginForm = new FormGroup({              email: new FormControl(null, [Validators.required, Validators.minLength(4)]),              password: new FormControl(null, [Validators.required, Validators.maxLength(8)])          });      }      loginUser() {          console.log(this.loginForm.status);          console.log(this.loginForm.value);      }  } |

#### **Using FormBuilder :**

FormBuilder is used to simplify the syntax for FormGroup and FormControl. This is very useful when your form gets lengthy. Let;s refactor loginForm to use FormBuilder. To do so, first import FormBuilder from @angular/forms. Then inject it to the component as shown in the listing below

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { FormGroup, FormControl, FormArray, Validators, FormBuilder } from '@angular/forms';  @Component({      selector: 'app-root',      templateUrl: './app.component.html',      styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {        loginForm: FormGroup;        constructor(private fb: FormBuilder) {        }      ngOnInit() {            this.loginForm = this.fb.group({              email: [null, [Validators.required, Validators.minLength(4)]],              password: [null, [Validators.required, Validators.maxLength(8)]]            })      }        loginUser() {          console.log(this.loginForm.status);          console.log(this.loginForm.value);      }  } |

Custom Validators

|  |
| --- |
| <div class=*"alert alert-danger"* **\*ngIf**=*"contactform.get('age').dirty && contactform.get('age').errors && contactform.get('age').hasError('ageRange') "*>  Age should be in between 18 to 45 years</div>  **import** {Component, OnInit, Input, ViewChild} **from** '@angular/core';  **import** {FormGroup, FormControl, Validators, FormBuilder,AbstractControl} **from** '@angular/forms';  **import** {MyClass} **from** './myclass';  **import** {OtherChildComponent} **from** './other-child/other-child.component';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  **export** **class** AppComponent {  contactform :**any**    **constructor**(**private** formBuilder: FormBuilder) {  **this**.contactform = **this**.formBuilder.group({  firstname: ['', [ Validators.required,Validators.minLength(10)]],  lastname: ['', Validators.required],  // age: [null,[Validators.required], this.ageRangeValidator],  age: [**null**, [**this**.ageRangeValidator]],  address: **this**.formBuilder.group({  city :['', Validators.required],  street: ['', Validators.required],  pincode: ['', Validators.required]  })  });  }  onSubmit(){  console.info("fomr onsubmit method executed: ", **this**.contactform.value)  console.info("fomr onsubmit method executed: ", **this**.contactform.status)  }  ageRangeValidator(control: AbstractControl): { [key: **string**]: **boolean** } | **null** {    **if** (control.value !== **undefined** && (isNaN(control.value) || control.value < 18 || control.value > 45)) {  **return** { 'ageRange': **true** };  }  **return** **null**;  }  } |

## Using Angular HttpClient

The HttpClient is a separate model in Angular and is available under the @angular/common/http package. All you need to do is to import it and  inject it into our component/service. Then, Use HttpClient.Get method to send an HTTP Request and Subscribe to the response Asynchronously. And when the response arrives map it the desired object and displays the result.

### Call the HttpClient.Get method

We send the HttpClient.get request to our Service API endpoint. The HttpClient.Get method returns an Observable, which is subscribed. The Result arrives in JSON format which is displayed to the user.

### **HttpClient.get**

The HttpClient.get sends the HTTP Get Request to the API endpoint and parses the returned result to the desired type. The default response type is JSON. If you want any other type, then you need to specify explicitly using the responseType parameter.

#### Parameters of the HttpClient.get

##### headers

Allows you to add HTTP headers to the outgoing requests.

##### observe

The HttpClient.get method returns the body of the response parsed as JSON (or type specified by the responseType). Sometimes you may need to read the entire response along with the headers and status codes. To do this you can set the observe property to **response**.

The allowed options are

* response which returns the entire response
* body which returns only the body
* events which returns the response with events.

##### **params**

Allows us to Add the URL parameters to the Get Request

##### **reportProgress**

This is a boolean property. Set this to true, if you want to get notified of the progress of the Get Request. This is a pretty useful feature when you have a large amount of data to download (or upload) and you want the user to notify of the progress.

##### **responseType**

Json is the default response received. In case you want a different type of response, then you need to use this parameter. The Allowed Options are **arraybuffer**, **blob**, **json** and **text**.

##### **withCredentials**

It is of boolean type. If the value is true then HttpClient.get will request data with credentials

### Return from the httpClient.Get

Notice that when we use Angular HttpClient Services to get data, it actually returns an

**Observables.**

Observable help us to manage async data. You can think of Observables as an array of items, which arrive asynchronously over time.

The observables implement the observer design pattern, where observables maintains a list of dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods

Observer subscribes to an Observable. The observer reacts when the value of the Observable changes. An Observable can have multiple subscribers and all the subscribers are notified when the state of the Observable changes.

The Observables are used extensively in Angular. The new HTTP service and Event system are all Observable based.

The Observables are Proposed Feature for the next version of Javascript. The Angular 2 Uses Third party library called Reactive Extensions or RxJs to implement the Observables. You can learn about RxJs from these RxJx tutorials

### **Observables Operators**

Operators are methods that operate on an Observable and return an observable. Each Operator modifies the value it receives. These operators are applied one after the other in a chain.

The RxJs Provides several Operators, which allows you to filter, select, transform, combine and compose Observables. Examples of Operators are map, filter, take, mergeetc

**Angular cli commands:**

### Create New Project

ng new my-project

### Run Application

ng serve

### Create Component In A Module

ng g c employee/employee-detailes --spec=false –flate=true

create routing modile

ng g c employee/employee-detailes --spec=false –flate=true

https://grokonez.com/spring-framework/spring-data/spring-boot-angular-6-example-spring-data-jpa-rest-mysql-crud-example

**spring boot curd with angular 5:**

<https://grokonez.com/spring-framework/spring-data/spring-boot-angular-6-example-spring-data-jpa-rest-mysql-crud-example>

routing

https://www.techiediaries.com/angular-router/#Introduction\_to\_Angular\_6\_Router

Forms example:

<https://www.infragistics.com/community/blogs/b/infragistics/posts/how-to-create-your-first-angular-reactive-form>

<https://appdividend.com/2017/09/25/angular-form-validation-example-tutorial/>

good tutorial

<https://www.tektutorialshub.com/angular-2-tutorial/>

<https://coursetro.com/posts/code/52/Trying-out-the-New-Angular-4-If-Else-Conditionals>

<https://coryrylan.com/blog/angular-form-builder-and-validation-management>

<https://www.toptal.com/angular-js/angular-4-forms-validation>

<https://www.dotnetcurry.com/angularjs/1323/angular2-passing-data-master-detail-component>