*Angular 2*

Angular applications are made up of components. A component is the combination of an HTML template.

***Main Files in Angular 2 :***

***app/app.component.ts*** : It is the root component of what will become a tree of nested components as the application evolves.

***app/app.module.ts*** : Defines AppModule, the root module that tells Angular how to assemble the application.

Eg :

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

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

@NgModule({

imports: [ BrowserModule ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

* declarations - the view classes that belong to this module. Angular has three kinds of view classes: [components](https://angular.io/docs/ts/latest/guide/architecture.html#components), [directives](https://angular.io/docs/ts/latest/guide/architecture.html#directives), and [pipes](https://angular.io/docs/ts/latest/guide/pipes.html).
* exports - the subset of declarations that should be visible and usable in the component [templates](https://angular.io/docs/ts/latest/guide/architecture.html#templates) of other modules.
* imports - other modules whose exported classes are needed by component templates declared in this module.
* providers - creators of [services](https://angular.io/docs/ts/latest/guide/architecture.html#services) that this module contributes to the global collection of services; they become accessible in all parts of the app.
* bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Note : The export of AppComponent is just to show how to export; it isn't actually necessary in this example. A root module has no reason to export anything because other components don't need to import the root module.

***main.ts*** : Compiles the application with the JIT compiler and bootstraps the application's main module (AppModule) to run in the browser.

***JIT (Just in Time ) Compiler :***

A bootstrapping method of compiling components and modules in the browser and launching the application dynamically

Eg : // The browser platform with a compiler

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

// The app module

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

// Compile and launch the module

platformBrowserDynamic().bootstrapModule(AppModule);

***ARCHITECTURE OVERVIEW***

The architecture identifies the eight main building blocks of an Angular application:

1. **Modules :** Angular has its own modularity system called Angular modules or NgModules.

Every Angular app has at least one Angular module class, [the root module](https://angular.io/docs/ts/latest/guide/appmodule.html),  
conventionally named AppModule.

1. **Component :**

Every component begins with an @Component [decorator](https://angular.io/docs/ts/latest/glossary.html#decorator) function that takes a metadata object. The metadata object describes how the HTML template and component class work together.

**decorator** : A function that adds metadata to a class, its members (properties, methods) and function arguments

1. @Component({...})
2. export class AppComponent {
3. constructor(@Inject('SpecialFoo') public foo:Foo) {}
4. @Input() name:string;
5. }

@**component** decorator ( that identifies a class as an Angular component ) :

Component decorator allows you to mark a class as an Angular component and provide additional metadata that determines how the component should be processed, instantiated and used at runtime.

**Properties :**

* **selector** - css selector that identifies this component in a template. The selector property tells Angular to display the component inside a custom <my-app> tag in the index.html
* **styles** - inline-defined styles to be applied to this component's view
* **template** - inline-defined template for the view
* **templateUrl** - holds the external file path of HTML
* **styleUrls** - StyleUrl holds the external file path of CSS

**For more info** : <https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html>

@input decorator (applied to the name property of that component. )

Note : Always include parentheses () when applying a decorator.

1. **Templets :**

The component's view can be defined by using the template which tells Angular how to display the component.

Eg : Course.component.html

template :`<h2> courses </h2>

    {{title}}

    <ul>

    <li \*ngFor="let course of courses">

        {{ course }}

    </li>

    </ul>

    `,

1. **MataData :**

[Looking back at the code](https://angular.io/docs/ts/latest/guide/architecture.html#component-code) for CourseComponet , you can see that it's just a class. There is no evidence of a framework, no "Angular" in it at all.

To tell Angular that CourseComponet is a component, attach **metadata** to the class.

@Component({

selector: 'Course',

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

})

export class CourseComponet implements OnInit {

/\* . . . \*/

}

1. **Data Bininding :**

Data binding is a process of coordinating application data values by declaring bindings between sources and target HTML elements. It combines the template parts with components parts and template HTML is bound with markup to connect both sides. There are four types of data binding:

Eg:

<li>{{hero.name}}</li>

<hero-detail [hero]="selectedHero"></hero-detail>

<li (click)="selectHero(hero)"></li>

* The {{hero.name}} [interpolation](https://angular.io/docs/ts/latest/guide/displaying-data.html#interpolation) displays the component's hero.name property value within the <li> element.

Eg :

1. Sum of 20 + 30 is {{20 + 30}}
2. title}}

{{primeMinister.name.lname}}

1. 3<img src="{{imageUrl}}">

**More details:** <http://www.concretepage.com/angular-2/angular-2-interpolation-expression-html-example>

* The [hero] [**property binding**](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.

3 types of Property binding :

1.Element property binding :

**<a [href]="website.url" [textContent]="website.name"> </a>**

2. Directive property binding is performed as below.

**<p [ngClass]="'one two'"> Angular 2 Property Binding Example </p>**

1. Component property binding :

**<my-msg prefixMsg= "Website name is " [siteName] = "website.name"> </my-msg>**

**More details:** <http://www.concretepage.com/angular-2/angular-2-property-binding-example>

* The (click) [**event binding**](https://angular.io/docs/ts/latest/guide/user-input.html#click) calls the component's selectHero method when the user clicks a hero's name.

**1.Input Event Binding :**

**<input [value]="msg1" (input)="msg1=$event.target.value">**

We can also use :

**<input [value]="msg1" on-input="msg1=$event.target.value">**

Use input event to call the method :

**<input [value]="msg1" (input)="setMsg($event.target.value)">**

**setMsg(data:string) {**

**this.msg1 = data;**

**}**

**2. Click Event Binding**

**<button (click)="isValid=true">True</button>**

**<button (click)="isValid=false">False</button>**

**3. Change Event binding**

**<br/><br/><b>Change Event Binding </b><br/><br/>**

**<input on-change="changeText($event.target.value)">**

**<p [innerHTML] = "msg3"> </p>**

**changeText(mytext:string) {**

**this.msg3 = mytext;**

**}**

**Note : need press enter after typing in text field.**

**Two-way data binding** is an important fourth form that combines property and event binding in a single notation, using the ngModel directive.

**Eg:**

**<input [(ngModel)]="hero.nam e">**

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Component life cycle : watch video 28,29.

**6.Directives :**

Angular templates are dynamic. When Angular renders them, it transforms the

DOM according to the instructions given by **directives**.

Angular provides three types of directive: component directive, attribute directive and structural directive.

**Attribute directive :** Attribute directive changes the appearance or behavior of DOM element. Using custom attribute directive we can change appearance such as text color, background color and font size of body of an HTML element that can be called host element. To change appearance angular provides ElementRef class that can directly access DOM.

Link : <http://www.concretepage.com/angular-2/angular-2-custom-attribute-directive-example>

Imp : **select dropDown**

<select #selCombo>

 <option value='' selected> Select Color </option>

  <option \*ngFor = "let color of colors" [value]="color">

  {{color}}

  </option>

</select>

Radio Button :

<div>

  <input type="radio" name="colors" (click)="color='lightgreen'">Green

  <input type="radio" name="colors" (click)="color='yellow'">Yellow

  <input type="radio" name="colors" (click)="color='cyan'">Cyan

</div>

**Structural directive :** Structural directive is used to change the DOM layout by adding and removing DOM elements.

**NgIf :** The ngIf directive doesn't hide elements with CSS. It adds and removes them physically from the DOM.

<p \*ngIf="true"> <p \*ngIf="false">

Expression is true Expression is true

</p> </p>

Link : <http://www.concretepage.com/angular-2/angular-2-ngif-example>

How to us If in different ways :

<my-msg \*ngIf="emp1" [pname] = "emp1.name"> </my-msg>

<my-msg template="ngIf:emp1" [pname] = "emp1.name"> </my-msg>

<template [ngIf]="emp1">

<my-msg \*ngIf="emp1" [pname] = "emp1.name"> </my-msg>

</template>

### NgFor Local Variables

NgFor uses following local variables.   
  
**index**: Provides the index for current loop iteration. Index starts from 0.   
**first**: Provides Boolean value. It returns true if the element is first in the iteration otherwise false.   
**last**: Provides Boolean value. It returns true if the element is last in the iteration otherwise false.   
**even**: Provides Boolean value. For every index of elements in the iteration, if even then returns true otherwise false.   
**odd**: Provides Boolean value. For every index of elements in the iteration, if odd then returns true otherwise false.

Link : <http://www.concretepage.com/angular-2/angular-2-ngfor-example>

**7. Services :**

Angular services are injectable and injector can inject it in any component in our angular application. When we add our service in providers metadata of @NgModule in module file then the service becomes available globally in the application. To get instance of service in our component, we need to create a constructor with arguments of our service types in private scope.

### Steps to Create Angular Servicev :

**Step-1**: Create a class decorated with @Injectable().

When our class uses @Injectable() then angular injector finds this class available for instantiation. We should know that injector is also responsible to instantiate a component, pipe directive. This is possible because@Component, @Pipe and @Directive are subtype of @Injectable() decorator.

**Step-2**: Configure service name in providers metadata with @NgModule in module file.

**Step-3**: Inject service in Component.

export class StoreComponent {

constructor(private name : Service) { }

}

Link : <http://www.concretepage.com/angular-2/angular-2-services-example-using-injectable>

**Routing ans Navigation**

The Angular [Router](https://angular.io/api/router/Router) ("the router") borrows from this model. It can interpret a browser URL as an instruction to navigate to a client-generated view. It can pass optional parameters along to the supporting view component that help it decide what specific content to present.

### RouterModule and Routes :

1 . **Import RouterModule and Routes:**

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

2 . **Create Array of Routes**:

const routes: Routes = [

{ path: 'manage-book', component: ManageBookComponent },

{ path: 'update-book/:id', component: UpdateBookComponent },

{ path: '', redirectTo: '/manage-book ', pathMatch: 'full' },

{ path: '\*\*', component: PageNotFoundComponent }]

Routs explaination :

**a . Mapping a Route to a Component :**  when we access URL **/manage-book** then ManageBookComponent will be displayed.

**b .** **Configure Parameters :**  In the above path mapping we need to pass a path parameter, for example if we access the URL **update-book/100** then UpdateBookComponent will be displayed.

**c .  Redirect to a URL :** If we access a URL without specifying any component path such as **"/"** then it will be redirected to URL **/manage-book**path and hence by default ManageBookComponent will be displayed.

**d . Handling "Page Not Found" :** If we access path that has no mapping with any component, then to handle **404 Not Found** error, we use a path**(\*\*)** that is mapped with any component to show desired message.

**3. Using RouterModule.forRoot()**: Now we need to import RouterModule.forRoot(routes) using importsmetadata of @NgModule. Here argument routes is our constant that we have defined above as array of Routes to map path with component.

imports: [ RouterModule.forRoot(routes) ]

**2 .Router :** It is used to navigate from one component to another component. To use Router in any component, follow the steps.

**1.Import Router**: Import Router as follows.

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

**2.Router Instance**: Make Router service available in component using dependency injection with constructor.

constructor(private router: Router) {

}

**3.Using Router navigate()**: Call navigate() method of Router and pass path and parameter if any, to navigate from one component to another component. Find the code snippet.

this.router.navigate(['/update-book', id]);

Here URL **/update-book/:id** will be the path to navigate. When the navigate() method will be executed, the component mapped with URL **/update-book/:id** will be displayed.

**3. ActivatedRoute and Params :** ActivatedRoute is an angular service that contains route specific information such as route parameters, global query params etc. Params is an angular router API that contains the parameter value. To get the parameter value from Params we need to pass key. To use ActivatedRoute and Params in our component, find the steps.

**1. Import ActivatedRoute and Params**: Import ActivatedRoute and Params as given below.

import { ActivatedRoute, Params } from '@angular/router';

**2. ActivatedRoute Instance**: Make ActivatedRoute available in component using dependency injection with constructor.

constructor(private route: ActivatedRoute, private bookService: BookService) {}

In the above code BookService is our service which we will create in our example.

**3. Routing with Parameters**: Now suppose a URL **/update-book/100** is being accessed. To understand the fetching of parameter, find the mapping of component which we configure in module.

{ path: 'update-book/:id', component: UpdateBookComponent }

URL **/update-book/100** will invoke UpdateBookComponent. The path parameter will be accessed by **id** as given in path mapping with component. We will fetch URL as following.

this.route.params

.switchMap((params: Params) => this.bookService.getBook(+params['id']))

.subscribe(book => this.book = book);

We need to know following points.   
**a.** switchMap operator allows us to perform an action with the current value of the Observable and map it to new Observable. Import switchMap in component as follows.

import 'rxjs/add/operator/switchMap';

switchMap handles an Observable as well as a Promise to retrieve the value they emit. 

**b.** (+) converts string 'id' to a number. In +params['id'], id is the keyword used in URL mapping with component.+params['id'] will return **100** if we use URL **/update-book/100** .   
**c.** subscribe is used to detect the **id** changes to retrieve Book. Book is a class that will be created in our example.

**4. Location :**  Location is a service that is used to interact with browser URL for example navigating back and forward. Locationhas methods such as go(), forward() and back() etc. To use Location service, find the following points.   
**1. Import Location**: Import Location in component.

import { Location } from '@angular/common';

**2. Location Instance**: Make Location available in component using dependency injection with constructor.

constructor(private location: Location) { }

**3. Navigate Back**: If we want to go back, then we will call back() method as follows.

this.location.back();

**5.RouterLink and RouterLinkActive :** RouterLink is a directive that is used to bind a route with clickable HTML element. RouterLinkActive is a directive that is used to add or remove CSS classes. When the HTML element with RouterLink binding is clicked then the CSS classes bound with RouterLinkActive will be active. These directives are used are follows.

<a routerLink="/manage-book" routerLinkActive="active-link">Manage Book</a>

<a [routerLink]="['/view-detail', book.id]">View Detail</a>

In the first link routerLink is bound with a route and routerLinkActive is bound with a CSS class. When this linked will be clicked then the associated CSS class will be activated. In the second link we are binding only routerLinkwith a parameter.

**6. RouterOutlet :**

RouterOutlet is a directive that is used as <router-outlet>. The role of <router-outlet> is to mark where the router displays a view. Find the code snippet.

<nav [ngClass] = "'menu'">

<a routerLink="/home" routerLinkActive="active-link">Home</a> |

<a routerLink="/add-book" routerLinkActive="active-link">Add Book</a> |

</nav>

<router-outlet></router-outlet>

We have created menu items in the above code using RouterOutlet. They will be shown in every view where we navigate using the route binding with routerLink.

Link : <http://www.concretepage.com/angular-2/angular-2-routing-and-navigation-example>

**Forms**

There are 2 types of forms :

1. Template-driven forms.

2. Data – driven forms.

Tip: use FormsModule for template-driven, and ReactiveFormsModule for reactive forms.

1. **Template-driven forms :** With template-driven forms, we can essentially leave a component class empty until we need to read/write values (such as submit and setting initial or future data).

**@Component({**

**selector: 'signup-form',**

**template: `**

**<form novalidate>...</form>**

**`**

**})**

**export class SignupFormComponent {**

**constructor() {}**

**}**

#### ngForms and ngSubmit

**FormControl**: This class tracks the value and validation status of an individual form control.   
**FormGroup**: This class tracks the value and validity state of a group of FormControl.

**<form #form="ngForm">**

**...**

**</form>**

In this <form> we are exporting the ngForm value to a public #f variable, to which we can render out the value of the form.

Tip: #f is the exported form Object, so think of this as the generated output to your model’s input.

**<form #form="ngForm" (ngSubmit)="logForm(form.value)">**

**...**

**</form>**

We will use (ngSubmit) instead of (submit ) : Functionality wise both are same but ngSubmit ensures that the form doesn’t submit when the handler code throws (which is the default behaviour of submit) and causes an actual http post request.

#### [ngModel, [ngModel] and [(ngModel)]](https://toddmotto.com/angular-2-forms-template-driven#ngmodel-ngmodel-and-ngmodel)

* *ngModel* = if no binding or value is assigned, ngModel will look for a name attribute and assign that value as a new Object key to the global ngForm Object:

Tip: ngModel “talks to” the form, and binds the form value based on the nameattribute’s value

<p>Form value: <b> {{ userForm.value | json }}

Output : Form value: { "name": "", "city": "", "state": "" }

Form value: **{}**

* *[ngModel]* = one-way binding syntax, can set initial data from the bound component class.

*Data in component is set to form initially.*

<p>Form value: <b> {{ userForm.value | json }}

OutPut : Form value: **{}**

Note: The actual value of this.user.name is never updated upon form changes, this is one-way dataflow. Form changes from ngModel are exported onto the respectived f.value properties.

* *[(ngModel)]* = two-way binding syntax, can set initial data from the bound component class, but also update it:

<p>Form value: <b> {{ userForm.value | json }}

OutPut : Form value: **{}**

#### [ngModelGroup](https://toddmotto.com/angular-2-forms-template-driven#ngmodels-and-ngmodelgroup)

ngModelGroup enables us to semantically group our form controls. In addition to that, it also tracks validity state of the inner form controls. This comes in very handy if we want to check the validity state of just a sub set of the form.

<form novalidate #f="ngForm">

<label> Name</label>

<input type="text"

name="name"

ngModel>

<div ngModelGroup="account">

<label> Email</label>

<input type="email"

name="email"

ngModel>

<label> Confirm Email</label>

<input type="email"

name="confirm"

ngModel>

</div>

<button type="submit">Sign up</button>

</form>

This creates a nice structure based on the representation in the DOM that pseudo-looks like this:

ngForm -> '#f'

ngModel -> 'name'

ngModelGroup -> 'account'

-> ngModel -> 'email'

-> ngModel -> 'confirm'

#### [Template-driven error validation](https://toddmotto.com/angular-2-forms-template-driven#template-driven-error-validation)

*Disable on submit button :*

<button type="submit" [disabled]="f.invalid">Sign up</button>

*required attributes on each <input>:*

<lable>

Name: <input name="name" ngModel required minlength=”2” #userName="ngModel">

</lable**>**

<div \*ngIf="userName.errors?.required" class="error">

Name is required

</div>

<div \*ngIf="userName.errors?.minlength class="error">

Minimum of 2 characters

</div>

Link : <https://toddmotto.com/angular-2-forms-template-driven>.

***Data Driven Form(Reactive Form)***

**Reactive Form**: Creating form using FormControl, FormGroup and FormArray is said to be reactive form. They use ng module as ReactiveFormsModule.   
**Template-Driven Form**: Creating form using NgForm and NgModel, is said to be template-driven form. They use ng module as FormsModule.

### FormControl

FormControl is used to track values and validation of a form control. It can be used standalone as well as with the parent form. When we work with FormControl class, FormControlDirective and FormControlName directives are also used.

**FormControlDirective**: It syncs a standalone FormControl instance to form a control element.   
**FormControlName**: It is used with FormGroup with a <form> tag. FormControlName syncs a FormControl in an existing FormGroup to a form control by name.

### 1. Default Value

If we want to set a default value to our form control, we can pass a default value while instantiating a FormControlin our class.

city = new FormControl('Noida');

married = new FormControl(true);

In HTML template, we will use these properties with form control.

<input [formControl]="city">

<input type="checkbox" [checked]="married.value" (change)="changeValue()" />

### 2. Get and Set Value

To fetch the value of a form control, we have to use value property on the instance of FormControl in our class. In the same way we can fetch the value in HTML template.

city = new FormControl('Noida');

console.log(this.city.value);

If the user has entered new value in UI, our FormControl instance will be updated with new value. Now to set a value to a form control at run time, we need to call setValue() method on the instance of FormControl in our class.

setCityValue() {

this.city.setValue('Varanasi');

}

### 3. Disable FormControl

To disable a form control, we need to pass disabled property as true while instantiating FormControl in our class. Find the code snippet.

country = new FormControl({value: 'India', disabled: true});

Find the line of code in our HTML template.

<input [formControl]="country">

The above form control will be displayed as disabled in UI with a default value.

### 4. FormControl Validation

To validate a form control created by FormControl, we need to use Validators that belongs to @angular/formslibrary. Find the code snippet.

name = new FormControl('', [Validators.required, Validators.maxLength(15)]);

age = new FormControl(20, Validators.required);

In HTML template, we write code as below.

Name: <input [formControl]="name">

<label \*ngIf="name.invalid" [ngClass] = "'error'" > Name required with max 15 character. </label>

Age: <input [formControl]="age">

<label \*ngIf="age.invalid" [ngClass] = "'error'" > Age required. </label>

The above code is valid when we are using standalone FormControl and not with the HTML form tag.

### 5. Checkbox using FormControl

To create a checkbox using FormControl we can do as given below.

married = new FormControl(true);

this.userForm.patchValue({name: 'Mahesh'});

Find the HTML code.

<div>

<input type="checkbox" [checked]="married.value" (change)="changeValue()" />

Are you married?

</div>

To listen change event, find the method that needs be created in our class.

Chang eValue() {

console.log(this.married.value);

this.married = new FormControl(!this.married.value);

}

### FormControl with FormGroup using FormControlName

We will create form control with parent HTML <form> tag. All the form control which are the instances of FormControl class, will be grouped using FormGroup class. Find the code snippet

serForm = new FormGroup({

name: new FormControl('Mahesh', Validators.maxLength(10))

)}

**<form [formGroup]="userForm" (ngSubmit)="onFormSubmit()">**

**<div>**

**Name: <input formControlName="name" placeholder="Enter Name">**

**<lable \*ngIf= userForm.get('name').invalid > Exceeded the length </lable>**

**</div>**

**</form>**

### FormControl with FormArray using FormControlName

FormArray is used when we want to dynamically generate form controls such as **<input>** and **<select>**.

userForm = new FormGroup({

users: new FormArray([

new FormControl('Mahesh'),

new FormControl('Krishna'),

new FormControl()])

});

**In the above code we have created an array of FormControl instances. We will iterate it in our UI as follows**

<div \*ngFor="let user of users.controls; index as idx">

<input [formControlName]="idx" placeholder="Enter User Name">

</div>

**On form submit, we can fetch values as given below:**

for(let i = 0; i < this.users.length; i++) {

console.log(this.users.at(i).value);

}

**Now to add a form control at run time we need to use push() method of FormArray :**

addUserField() {

this.users.push(new FormControl());

}

**To remove at run time we need to use removeAt() method of FormArray.**

deleteUserField(index: number) {

this.users.removeAt(index);

}

Link : <file:///D:/Angular%202%20Deeksha/Angular%202/Angular%202%20FormControl%20Example.html>

<file:///D:/Angular%202%20Deeksha/Angular%202/Angular%202%20FormGroup%20Example.html>

### FormGroup :

**FormControlName**: It is a directive that syncs a FormControl in an existing FormGroup to a form control by name.   
**FormGroupDirective**: It is a directive that binds an existing FormGroup to a DOM element.   
**FormGroupName**: It is a directive that syncs a nested FormGroup to a DOM element.   
**FormArrayName**: It is a directive that syncs a nested FormArray to a DOM element.

### FormGroup setValue() and patchValue()

 setValue(), we have to set all form controls in our FormGroup

patchValue()  selected form control can be set.

userForm = new FormGroup({

name: new FormControl(),

age: new FormControl()

});

this.userForm.setValue({name: 'Mahesh', age: '20' });

this.userForm.patchValue({name: 'Mahesh'});

### FormGroup valueChanges() and statusChanges()

To track the value changes and status changes of a form control we need to listen them using subscribe() method. Find the code.

usrNameChanges: string;

usrNameStatus: string;

ngAfterViewInit(): void {

this.userForm.get('name').valueChanges.subscribe(data => this.usrNameChanges = data); this.userForm.get('name').statusChanges.subscribe(data => this.usrNameStatus = data);

}

<p \*ngIf="usrNameChanges">Name: <b>{{usrNameChanges}}</b> </p>

<p \*ngIf="usrNameStatus">Name Status: <b>{{usrNameStatus}}</b> </p>

### FormGroup with Radio Button

Select Gender: <input type="radio" formControlName="gender" value="male"> Male

<input type="radio" formControlName="gender" value="female" > Female

### FormGroup with Select Element

Here we will provide how to use select element with FormGroup. First of all we will create an array of elements that will contain key and value for select element. Find the array given below.

profiles = [

{name: 'Developer', shortName: 'dev'},

{name: 'Manager', shortName: 'man'},

{name: 'Director', shortName: 'dir'}

];

We will use the value of name in our array to display data in select element and the value of shortName will be assigned to value attribute of <option> element.

Now to use select element in a FormGroup, first we will create an instance of FormControl within a FormGroup.

profile: new FormControl(this.profiles[0].shortName);

We can pass the initial selected value while instantiating FormControl. In the above code first element of array profiles will be selected as default. Now we will create the form in HTML template as given below.

<div> Select Profile:

<select formControlName="profile">

<option \*ngFor="let pf of profiles" [ngValue]="pf.shortName">{{ pf.name }}</option>

</select>

</div>

We can access its value using get() method.

this.userForm.get('profile').value;

***Pipes***

 Pipes are the modernized version of filters in Angular 1.X.Here it isa called as Pipes.

### Usage of pipes

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

### General syntax

myValue | myPipe:param1:param2 | mySecondPipe:param1

### Angular 2 UpperCase Pipe and LowerCase Pipe

UpperCasePipe is a PIPE that transforms string to uppercase.

{{message | uppercase}}

LowerCasePipe is a PIPE that transforms string to lowercase.

{{message | lowercase}}

Link : <http://www.concretepage.com/angular-2/angular-2-uppercase-pipe-and-lowercase-pipe-example>

### Angular 2 Decimal, Percent and Currency Pipe

### 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 thedescription.   
**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

{{num1 | number}} //num1=12.25368

o/p : 12.254

**2. Use format '3.2-5' :**

{{num1 | number:'3.2-5'}} //num1 = 12.638472

o/p : 012.63847

**3. Format '3.2-5'**

{{num2 | number:'3.2-5'}} //num2 =0.5

o/p : 000.050

### 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]**

The syntax for **digitInfo** is as follows.

**{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}**

Now find some sample examples.   
**1. Using default format:**

{{num1 | percent}} //num1 = 2.5;

Find the output : 250%

1. **Use format '2.2-5' :**

{{num2 | percent:'2.2-5'}} //num1 = 2.5;

Find the output : 250.00%

1. **Use format '1.2-5' :**

{{num2 | percent:'1.2-5'}} //num1 = 0.5;

Find the output : 50.00%

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

{{cur1 | currency}} //cur1 = 0.25;

Find the output : USD0.25

**2. Use format '2.2-4'**

{{cur2 | currency:'USD':true:'2.2-4'}} //cur2 = 10.263782;

Find the output : $10.2638

Link : <http://www.concretepage.com/angular-2/angular-2-decimal-pipe-percent-pipe-and-currency-pipe-example>

Angular 2 Custom Pipe

To create custom pipe, angular provides Pipe and PipeTransforminterfaces. Every pipe is decorated with @Pipe where we define the name of our custom pipe. Every pipe will implement PipeTransform interface. This interface provides transform() method and we have to override it in our custom pipe class. transform() method will decide the input types, number of arguments and its types and output type of our custom pipe.

### @Pipe Decorator and PipeTransform Interface

To create custom pipe we need to understand angular Pipe and PipeTransform API.   
  
**@Pipe Decorator**  
  
@Pipe decorator is a Pipe interface. A typescript class is decorated by @Pipe to create an angular custom pipe. Find the Pipe interface from the angular [Pipe](https://angular.io/docs/ts/latest/api/core/index/Pipe-interface.html) API document.

interface Pipe {

name : string

pure : boolean

}

**name**: Assign custom pipe name.   
**pure**: Assign true or false. Default is true. If true then pipe will be a pure pipe otherwise impure pipe. So by default all pipes are pure pipe.

**PipeTransform Interface**

PipeTransform is an angular interface. It has been defined as follows in angular [PipeTransform](https://angular.io/docs/ts/latest/api/core/index/PipeTransform-interface.html) API document.

interface PipeTransform {

transform(value: any, ...args: any[]) : any

}

1. First parameter (value: any): This is the value in left side of our pipe operator (|).   
   2. Optional parameters: These are arguments used with pipe in right side of pipe operator (|).   
   3. Value returned by method: This is the value that is the output of our custom pipe.

**Simple Custom Pipe**

transform(value: string): string {

return srtring; }

{{person.name | welcome}}

**Pass Arguments in Custom Pipe**

**Example : 1**

string\_expression | strformat[:seperator]

string\_expression : This is the value which is present left side of the pipe.

Strformat : name of the pipe

Seperator : This may be any words or symbol.

export class StrFormatPipe implements PipeTransform {

transform(value: string, seperator: string): string {

return string;

}

UI : {{message | strformat:'+'}}

Message : hi chaithan

o/p : hi+chaithan

**Example : 2** Create **division** Custom Pipe

number\_expression | division[:num\_divisor]

**number\_expression** : An expression that will gives a number and that will be dividend.   
**num\_divisor**: This number is the argument of our pipe that will be used as divisor.

transform(dividend: number, divisor: number): number {

let num = dividend / divisor;

return num; }

UI : <p>

Division Result: {{dividend | division: divisor}}

</p>

Dividend = 10

Divisor = 2

o/p : 10/2=5

**Example : 3** Create **repeat** Custom Pipe

transform(word: string, frequency: number): string {

let cnt = 1;

let strResult= word;

while (cnt < frequency) {

strResult = strResult + ' ' + word;

cnt = cnt + 1;

}

return strResult;

}

UI : {{person | repeat:2}}

Person : chethu

o/p : chethu chethu

**Example 4.** Create **myjson** Custom Pipe   
  
Here we are creating **myjson** custom pipe. This pipe will convert an expression into JSON format. It will accept two arguments. Find the syntax.   
  
**expression | myjson[:prettyprint[:fields]]**  
  
Find the description.   
**expression**: Expression that will be converted into JSON format. It can be primitive data type or any object.   
**prettyprint**: If the value is 0, then no pretty print and if the value is 1 or greater than 1, then we will get pretty print JSON format.   
**fields**: If no fields are provided then all the fields of object will take part in JSON format, if specified then only those fields will take part in JSON format.   
  
If no argument is passed then **myjson** will convert the given object into JSON format with all fields and without pretty print.

transform(value: any, prettyprint: number, fields: string): string {

let array = (fields == null? null : fields.split(','));

let pp = (prettyprint == null ? 0 : prettyprint);

let result = JSON.stringify(value, array, pp);

return result;

}

UI : <pre>{{person | myjson}}</pre>

<pre>{{person | myjson:0:'id,age'}}</pre>

<pre>{{person | myjson:1:'id,name'}}</pre>

<http://www.concretepage.com/angular-2/angular-2-custom-pipe-example>

### Use Built-in Pipe in Custom Pipe

Custom pipe can also use angular built-in pipe such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe.

1. Extending UpperCasePipe built-in pipe class: UpperCasePipe is an angular built-in pipe. It implements PipeTransform interface and override transform() method as we do in our custom pipes.

export class MyUppercaseOnePipe extends UpperCasePipe{

transform(value: string): string {

let result = super.transform(value);

result = result.split(' ').join('-');

return result;

}}

UI : {{message | myuppercaseone}}

Using super keyword we call the parent class method. The value passed to transform() method of MyUppercaseOnePipe class is first processed by transform() method of UpperCasePipe class and then we perform our changes and then we return the final result

1. Using object of UpperCasePipe built-in pipe class:

Here we will create our custom pipe as usual by implementing PipeTransform and within transform() method we will use parameters and their types according to our requirements. To process the data by UpperCasePipe pipe, we will create the object of it and pass the data to its transform() method.

export class MyUppercaseTwoPipe implements PipeTransform{

transform(value: string, seperator: string): string {

let upipe = new UpperCasePipe();

let result = upipe.transform(value);

result = result.split(' ').join(seperator);

return result;

}

}

UI : {{message | myuppercasetwo:'+'}}

### Chaining of Custom Pipes

Chaining of custom pipes is using more than one pipe together. The output of first pipe will be the input for next pipe and so on.

<http://www.concretepage.com/angular-2/angular-2-custom-pipe-example>

### Pure and Impure Custom Pipe and Change Detection

**1. Pure Pipes**:

By default all pipes are pure pipe. Pure pipes are those pipes that have pure: true in @Pipe decorator while creating pipe class. If we have not used pure metadata then its default value will be **true** in @Pipe decorator. Pure pipes executes only for pure changes in its input values.   
Find the pure changes.   
**a.** Change to a primitive input values such as String, Number, Boolean.   
**b.** Change to object reference of Date, Array, Function, Object.

@Pipe({

name: 'companyone'

})

1. **Impure Pipes** :

Impure pipes will run for every component change detection cycle. So it is obvious that impure pipes will also run for pure changes. Impure pipe will run for every keystroke or mouse move. So the conclusion is that impure pipe will run a lot and hence we should take care while using impure pipe because it may reduce performance of the application and can destroy user experience.

@Pipe({

name: 'companytwo',

pure: false

})

*Angular 2 Http get()*

Angular HTTP library provides Http client for server communication. get() is the Http client method that uses HTTP GET method to communicate server using HTTP URL. We need to pass HTTP URL to Http.get() and it will return the instance of RxJS Observable.

### Http.get() : Htp.get() performs a request with HTTP GET method. The syntax is as given below.

get(url: string, options?: RequestOptionsArgs) : Observable<Response>

get() is the method of angular Http API that interacts with server using HTTP GET method. It accepts a HTTP URL and returns Observable instance. RequestOptionsArgs is optional.

**Step-1:**: First step is that we need to import HttpModule in @NgModule using imports metadata in our application module.

**Step-2:**: We should perform server communication in a service class and not in component. This approach is preferable by design. In a service class we will use dependency injection to get the instance of angular Http as given below.

constructor(private http:Http) { }

**Step-3:**: Pass URL to use http.get().

Observable<Response> ob = this.http.get(this.url);

http.get() returns instance of Observable that can be later subscribed to get result. We can also fetch Observabledirectly in HTML template using async pipe.

### Observable and Promise

**Observable**: This is a RxJS API. Observable is a representation of any set of values over any amount of time. All angular Http methods return Observable. Observable provides methods such as map(), catch() etc.

map() applies a function to each value emitted by source Observable and returns finally an instance of Observable.

catch() is called when an error is occurred. catch() also returns Observable.

**Promise**: This is a JavaScript class. Promise is used for asynchronous computation. A Promise represents value which may be available now, in the future or never. Promise is a proxy for a value which is not known when the promise is created. Promise has methods such as then(), catch() etc.

http.get() returns Observable and to convert it into Promise we need to call RxJS toPromise() on the instance of Observable.

The then() method of Promise returns a Promise and that can be chained later. The catch() method of Promise is executed when an error is occurred and it also returns Promise.

### Http.get with Observable

Http.get() returns instance of Observable<Response>. To change it into instance of Observable<Book[]>, we need to use RxJS map() operator.

getBooksWithObservable(): Observable<Book[]> {

return this.http.get(this.url).map(this.extractData).catch(this.handleErrorObservable);

}

Find the extractData() method. map() converts Response object of http.get into Book.

private extractData(res: Response) {

let body = res.json();

return body;

}

To handle error, http.get provides catch() method. Find the handleErrorObservable() method.

private handleErrorObservable (error: Response | any) {

console.error(error.message || error);

return Observable.throw(error.message || error);

}

### Http.get with Promise : Now find the code to use Http.get with Promise.

getBooksWithPromise(): Promise<Book[]> {

return this.http.get(this.url).toPromise().then(this.extractData)

.catch(this.handleErrorPromise);

}

toPromise() method that will convert it into Promise<Response>. After calling then() method on it, it returns Promise<Book[]>. If there is any error then the catch() method will execute. extractData() used in then(), is the same used in map() method with Observable.

private handleErrorPromise (error: Response | any) {

console.error(error.message || error);

return Promise.reject(error.message || error);

}

Link : <http://www.concretepage.com/angular-2/angular-2-http-get-example>

Angular 2 Http get() Parameters + Headers + URLSearchParams + RequestOptions

### Headers

Headers is the angular class that is used to configure request headers. Find the sample Headers instantiation.

let myHeaders = new Headers();

We can also pass headers as an argument while instantiating Headers class. Find the code snippet.

let myHeaders = new Headers({ 'Content-Type': 'application/json',

'Cache-Control': 'no-cache' });

To fetch, add and delete headers, Headers class has following methods.   
**append(name: string, value: string)**: Appends a header to existing list of header values for a given header name. We use append() as follows.

myHeaders.append('Accept', 'text/plain');

myHeaders.append('Accept', ' application/xhtml+xml ');

Now the Accept header will have the following values.

Accept: text/plain, application/xhtml+xml

**set(name: string, value: string|string[])**: Sets or overrides header value for given name. It is used as follows.

myHeaders.set('Accept', ' application/xml ');

Now the Accept header will have only the following value.

Accept: application/xml

**delete(name: string)**: Deletes all header values for the given name. We use it as follows.

myHeaders.delete('Accept');

**get(name: string) : string**: Returns first header that matches given name. Find the code snippet.

let acceptHeader = myHeaders.get('Accept');

**getAll(name: string) : string[]**: Returns list of header values for a given name.

let acceptHeaders = myHeaders.getAll ('Accept');

If we want to add multiple headers, we can achieve it by set() method as follows.

myHeaders.set('Content-Type', 'application/json');

myHeaders.set('Accept', 'text/plain');

If we want to add multiple headers by append() method, we can achieve it as follows.

myHeaders.append('Content-Type', 'application/json');

myHeaders.append('Accept', 'text/plain');

### URLSearchParams

URLSearchParams creates the query string in the URL. It is a map-like representation of URL search parameters. Find its constructor syntax.

constructor(rawParams?: string, queryEncoder?: QueryEncoder)

Both arguments in the constructor are optional. Angular queryEncoder parameter is used to pass any custom QueryEncoder to encode key and value of the query string. By default QueryEncoder encodes keys and values of parameter using JavaScript encodeURIComponent() method.   
Now we can instantiate URLSearchParams as given below.

let myParams = new URLSearchParams();

Now we can fetch, add and delete parameters using following methods.   
**append(param: string, val: string) : void**: Appends parameter value to existing list of parameter values for a given parameter name. It is used to add values in multi-value fields or arrays in query string. If we write the code as given below.

myParams.append('names', 'John');

myParams.append('names', 'David');

Then query parameter names will be an array. The query string will look like as given below.

?names[]=John&names[]=David

Server side code such as PHP will get names parameter value as an array.   
**set(param: string, val: string)**: Sets or overrides parameter value for given parameter name. We can use as follows.

myParams.set('names', 'Bob');

The query string will be as follows.

?names=Bob

**delete(param: string) : void**: Deletes all parameter values for the given parameter name. Find the code snippet.

myParams.delete('names');

**get(param: string) : string**: In case of multi-value fields, it returns the first value for given parameter name. Find the code snippet.

let nameParam = myParams.get('names');

**getAll(param: string) : string[]**: Returns list of values for a given parameter name. Find the code snippet.

let namesParam = myParams.getAll('names');

If we want to add multiple parameters, we can achieve it by set() method as follows.

myParams.set('category', catg);

myParams.set('writer', wtr);

If we want to add multiple parameters by append() method, we can achieve it as follows.

myParams.append('category', catg);

myParams.append('writer', wtr);

### RequestOptionsArgs and RequestOptions

RequestOptionsArgs is an interface that is used to construct a RequestOptions. The fields of RequestOptionsArgs are url, method, search, params, headers, body, withCredentials, responseType.   
RequestOptions is used to create request option. It is instantiated using RequestOptionsArgs. It contains all the fields of the RequestOptionsArgs interface. Now find the constructor of RequestOptions class.

constructor({method, headers, body, url, search, params,

withCredentials, responseType}?: RequestOptionsArgs)

In our example we will use following fields.   
**headers**: Sets headers for HTTP request. It is of Headers class type.   
**params**: Sets query parameters in the URL. It is of URLSearchParams class type. 

Then headers and params can be passed to RequestOptions as given below.

let options = new RequestOptions({ headers: myHeaders, params: myParams });

# Angular 2 Http post()

 It performs a request using HTTP POST method. In Http.post() method, we need to pass server URL, any object to post and request option that is optional. In request option we can set request headers such as content type and to handle this angular provides Headers and RequestOptions API. Http.post() method returns the instance of Observable of Response type. Using angular Response we can access response status, headers etc.

### Http.post()

Http.post() performs a request using HTTP POST method. Find the syntax.

post(url: string, body: any, options?: RequestOptionsArgs) : Observable<Response>

**url**: This is HTTP URL using which we post data to the server.   
**body**: This is the object which we need to post to the server.   
**options**: This is optional. Here we pass the instance of RequestOptionsArgs that uses headers.   
**Observable<Response>**: This is the return type of Http.post().

Link : <http://www.concretepage.com/angular-2/angular-2-http-post-example>