# *Angular*

## Angular Component:

Components are like the basic building block in an Angular Application. Components are defined using the @component decorator.A component has a 'selector','template','style' and other properties using which it specifies the metadate required to process the component.

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

@Component({
   selector: 'calc',
   templateUrl: 'calc.component.html',
   styleUrls: ['calc.component.css']
})
export class CalcComponent {

}
```

## Databinding:

The communication between the typescript file and the html file.

From typescript file to html file:
1.String interpolation
        {{ anything that returns or casts to string}}
2.Property Binding
        [Any property of DOM]

From html file to typescript file:
3.Event Binding
        (Any function that can be invoked)

Two-way binding: Combination of property binding and event binding
[(ngModel)] = "[property of your component]"

```
export class AppComponent {
   fullName: string = "Hello JavaTpoint";
}
```

```
<h2>Two-way Binding Example</h2>
     <input [(ngModel)]="fullName" /> <br/><br/>
<p> {{fullName}} </p>
```

## Directives:

It basically changes the appearance or behaviour of a DOM element.

Three kinds:
**1.Components** - directive witha a template.

**2.Structural directives**: change the DOM layout by adding and removing DOM element. '*' is used with this. Ex -ngIf,ngFor
**3.Attribute directives**: change the appearance or behaviour of a DOM element, component or other directive. Ex-ngClass,ngStyle

**2. ngIf ------**

```
<div *ngIf="condition; else elseBlock">Content to render when condition is true.</div>
<ng-template #elseBlock>Content to render when condition is false.</ng-template>
```

**ngFor-----**

```
<li *ngFor="let user of users; let i = index;>
```

Sharing data between child and parent directives and components:

```
<parent-component>
  <child-component></child-component>
</parent-component>
```

**@Input()** and **@Output** give a child component a way to communicate with its parent component.
**@Input()** allows a parent component to update data in the child component. -> The typescript file will get the input from somewhere -> Parent to Child
**@Output()** allows the child to send data to a parent component. -> It provides output to something -> Child to Parent

**@Input()----------**

*Configuring the child component:

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

export class ItemDetailComponent {
  @Input() item: string; // decorate the property with @Input()
}

<p>
  Today's item: {{item}}
</p>
```

*Configuring the parent component

```
<app-item-detail [item]="currentItem"></app-item-detail>

export class AppComponent {
  currentItem = 'Television';
}
```

**@Output----------**

Configuring the child component:

```
@Output() newItemEvent = new EventEmitter<string>();
```

@Output()—a decorator function marking the property as a way for data to go from the child to the parent
newItemEvent—the name of the @Output()
EventEmitter<string>—the @Output()'s type

new EventEmitter<string>()—tells Angular to create a new event emitter and that the data it emits is of type string.

```
export class ItemOutputComponent {

   @Output() newItemEvent = new EventEmitter<string>();

   addNewItem(value: string) {
      this.newItemEvent.emit(value);
   }
}
```

Configuring the child's template:

```
<label>Add an item: <input #newItem></label>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```

Configuring the parent component:

```
export class AppComponent {
   items = ['item1', 'item2', 'item3', 'item4'];

   addItem(newItem: string) {
      this.items.push(newItem);
   }
}
```

Configuring the parent's template:

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
<ul>
   <li *ngFor="let item of items">{{item}}</li>
</ul>
```

In the parent's template, bind the parent's method to the child's event.

Put the child selector, here <app-item-output>, within the parent component's template, app.component.html.
The event binding, (newItemEvent)='addItem($event)', connects the event in the child, newItemEvent, to the method in the parent, addItem().

The $event contains the data that the user types into the <input> in the child template UI.

To see the @Output() working, you can add the following to the parent's template:

## View Encapsulation:

In Angular, component CSS styles are encapsulated into the component's view and don't affect the rest of the application.

To control how this encapsulation happens on a per component basis, you can set the view encapsulation mode in the component metadata. Choose from the following modes:

**ShadowDom view encapsulation** uses the browser's native shadow DOM implementation (see Shadow DOM on the MDN site) to attach a shadow DOM to the component's host element, and then puts the component view inside that shadow DOM. The component's styles are included within the shadow DOM.

**Emulated view encapsulation (the default)** emulates the behavior of shadow DOM by preprocessing (and renaming) the CSS code to effectively scope the CSS to the component's view. For details, see Inspecting generated CSS below.

**None** means that Angular does no view encapsulation. Angular adds the CSS to the global styles. The scoping rules, isolations, and protections discussed earlier don't apply. This is essentially the same as pasting the component's styles into the HTML.

## Template Variables:

Template variables help you use data from one part of a template in another part of the template. With template variables, you can perform tasks such as respond to user input or finely tune your application's forms.

A template variable can refer to the following:

a DOM element within a template
a directive
an element
TemplateRef

<input #phone placeholder="phone number" />

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>

How Angular assigns values to template variables
Angular assigns a template variable a value based on where you declare the variable:

If you declare the variable on a component, the variable refers to the component instance.
If you declare the variable on a standard HTML tag, the variable refers to the element.
If you declare the variable on an <ng-template> element, the variable refers to a TemplateRef instance, which represents the template. For more information on <ng-template>, see the ng-template section of Structural directives.
If the variable specifies a name on the right-hand side, such as #var="ngModel", the variable refers to the directive or component on the element with a matching exportAs name.

**@ViewChild:**

ViewChild is used for component communication in Angular. Therefore, if a parent component wants access of child component then it uses ViewChild. Any component, directive, or element which is part of a template is ViewChild.It is used to get the first element or the directive matching the selector from the view DOM.
Local template reference also.

If you want to access following inside the Parent Component,
1.Child Component
2.Directive
3.DOM element

**<ng-content></ng-content>:**

It is basically used for content projection. It allows you to insert a shadowDOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you simply do that using the concept of content projection.
You can make reusabloe components and scalable applications by properly using content projection.
Using the @Input decorator, you can pass a simple string to the component but to pass
1.InnerHTML
2.HTML Elements
3.Styled HTML
4.Another Component

**Building a basic attribute directive:**

An attribute directive minimally requires building a controller class annotated with @Directive, which specifies the selector that identifies the attribute. The controller class implements the desired directive behavior.

Generate the directive from CLI: ng generate directive highlight

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
      constructor(el: ElementRef) {
          el.nativeElement.style.backgroundColor = 'yellow';
      }
}
```

Implementing it: <p appHighlight>Highlight me!</p>

## Renderer2 in Angular:

The Renderer2 class is an abstraction provided by Angular in the form of a service that allows to manipulate elements of your app without having to touch the DOM directly. This is the recommended approach because it then makes it easier to develop apps that can be rendered in environments that don't have DOM access, like on the server, in a web worker or on native mobile.

```
import { Directive, Renderer2, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[appGoWild]'
})
export class GoWildDirective implements OnInit {
constructor(private renderer: Renderer2, private el: ElementRef) {}

ngOnInit() {
  this.renderer.setStyle(
     this.el.nativeElement,
```

```
      'border-left',
      '2px dashed olive'
   );
}
```

## @HostListener():

In Angular, the @HostListener() function decorator allows you to handle events of the host element in the directive class.

Let's take the following requirement: when you hover you mouse over the host element, only the color of the host element should change. In addition, when the mouse is gone, the color of the host element should change to its default color. To do this, you need to handle events raised on the host element in the directive class. In Angular, you do this using @HostListener() .

```
import { Directive, Renderer2, ElementRef, OnInit } from '@angular/core';

@Directive({
   selector: '[appGoWild]'
})
export class GoWildDirective implements OnInit {
constructor(private renderer: Renderer2, private el: ElementRef) {}

@HostListener('mouseover') onMouseOver()
{
this.renderer.setElemenetsStyle(this.el.nativeElement,'color','red','false','false');
}
```

## @HostBinding():

It is used in place of renderer.

```
@Directive({
   selector: '[appGoWild]'
})
export class GoWildDirective implements OnInit {

@HostBinding('style.backgroundColor') backgroundColor:string = 'transparent';

@HostListener('mouseover') onMouseOver()
{
this.backgroundColor='red';
}
```

## ngSwitch:

The [ngSwitch] directive on a container specifies an expression to match against. The expressions to match are provided by ngSwitchCase directives on views within the container.

Every view that matches is rendered.
If there are no matches, a view with the ngSwitchDefault directive is rendered.

Elements within the [NgSwitch] statement but outside of any NgSwitchCase or ngSwitchDefault directive are preserved at the location.

```
<div></div>
<container-element [ngSwitch]="switch_expression">
    <some-element *ngSwitchCase="match_expression_1">...</some-element>
...
    <some-element *ngSwitchDefault>...</some-element>
</container-element>
```

## Services in Angular:

Service is a broad category encompassing any value, function, or feature that an app needs.Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.
Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).
Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service.

**Dependency Injection in Angular:**

DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.
use the @Injectable() decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) has a dependency.

The injector is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.

An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible.

A provider is an object that tells an injector how to obtain or create a dependency.

For any dependency that you need in your app, you must register a provider with the app's injector, so that the injector can use the provider to create new instances.
When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one using the registered provider, and adds it to the injector before returning the service to Angular.

**Providing services:**

You must register at least one provider of any service you are going to use. The provider can be part of the service's own metadata, making that service available everywhere, or you can register providers with specific modules or components. You register providers in the metadata of the service (in the @Injectable() decorator), or in the @NgModule() or @Component() metadata

By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the @Injectable() decorator.

```
@Injectable({
  providedIn: 'root',
})
```

Creating the services:

Using the Angular CLI, create a service called hero : ng generate service hero

Skeleton:

```
import { Injectable } from '@angular/core';
import { Hero } from './hero';
import { HEROES } from './mock-heroes';

@Injectable({
   providedIn: 'root',
})
export class HeroService {

   constructor() { }

getHeroes(): Hero[] {
   return HEROES;
}

}
```

@Injectable() services
Notice that the new service imports the Angular Injectable symbol and annotates the class with the @Injectable() decorator. This marks the class as one that participates in the dependency injection system.

Updating HeroesComponent:

```
import { HeroService } from '../hero.service';
heroes: Hero[];

constructor(private heroService: HeroService) {}

ngOnInit() {
   this.getHeroes();
}

getHeroes(): void {
   this.heroes = this.heroService.getHeroes();
}
```

call getHeroes() inside the ngOnInit lifecycle hook and let Angular call ngOnInit() at an appropriate time after constructing a HeroesComponent instance.

** A service in angular always follows hierarchical injector. (parent to all child and their child components)

## **Routing in Angular:**

To handle the navigation from one view to the next, you use the Angular Router. The Router enables navigation by interpreting a browser URL as an instruction to change the view.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule} from '@angular/router';

import { AppComponent } from './app.component';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
import { ChangeTextDirective } from './change-text.directive';
import { SqrtPipe } from './app.sqrt';
@NgModule({
    declarations: [
        SqrtPipe,
        AppComponent,
        NewCmpComponent,
        ChangeTextDirective
    ],
    imports: [
        BrowserModule,
        RouterModule.forRoot([
            {
                path: 'new-cmp',
                component: NewCmpComponent
            }
        ])
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

The following command uses the Angular CLI to generate a basic Angular app with an app routing module, called AppRoutingModule, which is an NgModule where you can configure your routes. The app name in the following example is routing-app.
Generating an app with routing enabled: ng new routing-app --routing

Adding components for Routing: ng generate component first  ng generate component second

Importing your components to AppRoutingModule:

```
import { FirstComponent } from './first/first.component';
import { SecondComponent } from './second/second.component';
```

Defining a basic route:
Import the AppRoutingModule into AppModule and add it to the imports array.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module'; // CLI imports AppRoutingModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
```

```
  imports: [
    BrowserModule,
    AppRoutingModule // CLI adds AppRoutingModule to the AppModule's imports array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

**There are three fundamental building blocks to creating a route.**
**1.Import RouterModule and Routes into your routing module**.

The Angular CLI performs this step automatically. The CLI also sets up a Routes array for your routes and configures the imports and exports arrays for @NgModule().

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router'; // CLI imports router

const routes: Routes = []; // sets up routes constant where you define your routes

// configures NgModule imports and exports
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

**2.Define your routes in your Routes array.**

Each route in this array is a JavaScript object that contains two properties. The first property, path, defines the URL path for the route. The second property, component, defines the component Angular should use for the corresponding path.

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
];
```

**3.Add your routes to your application**.. Assign the anchor tag that you want to add the route to the routerLink attribute. Set the value of the attribute to the component to show when a user clicks on each link. Next, update your component template to include <router-outlet>. This element informs Angular to update the application view with the component for the selected route.

```
<h1>Angular Router App</h1>
<!-- This nav gives you links to click, which tells the router which route to use (defined in the routes
constant in    AppRoutingModule) -->
<nav>
  <ul>
    <li><a routerLink="/first-component" routerLinkActive="active">First Component</a></li>
    <li><a routerLink="/second-component" routerLinkActive="active">Second Component</a></li>
  </ul>
</nav>
<!-- The routed views render in the <router-outlet>-->
<router-outlet></router-outlet>
```

**Route Order:**
The order of routes is important because the Router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes. List routes with a static path first,

followed by an empty path route, which matches the default route. The wildcard route comes last because it matches every URL and the Router selects it only if no other routes match first.

**Getting route information:**
Often, as a user navigates your application, you want to pass information from one component to another.
Path variable,query params
You can use a route to pass this type of information to your application components. To do so, you use the ActivatedRoute interface.
Import ActivatedRoute and ParamMap to your component.
import { Router, ActivatedRoute, ParamMap } from '@angular/router';

These import statements add several important elements that your component needs. To learn more about each, see the following API pages:

Router
ActivatedRoute
ParamMap

```
constructor(
   private route: ActivatedRoute,
) {}

ngOnInit() {
   this.route.queryParams.subscribe(params => {
      this.name = params['name'];
   });
}
```

**Setting up wildcard routes:**
```
{ path: '**', component: HomeComponent }

const routes: Routes = [
   { path: 'first-component', component: FirstComponent },
   { path: 'second-component', component: SecondComponent },
   { path: '**', component: PageNotFoundComponent },    // Wildcard route for a 404 page
];
```

**Setting up redirects:**
```
const routes: Routes = [
   { path: 'first-component', component: FirstComponent },
   { path: 'second-component', component: SecondComponent },
   { path: '',      redirectTo: '/first-component', pathMatch: 'full' }, // redirect to `first-component`
   { path: '**', component: PageNotFoundComponent },    // Wildcard route for a 404 page
];
```

**Nesting routes:**

```
<h2>First Component</h2>

<nav>
   <ul>
      <li><a routerLink="child-a">Child A</a></li>
      <li><a routerLink="child-b">Child B</a></li>
   </ul>
</nav>
```

```
<router-outlet></router-outlet>


const routes: Routes = [
  {
    path: 'first-component',
    component: FirstComponent, // this is the component with the <router-outlet> in the template
    children: [
      {
        path: 'child-a', // child route path
        component: ChildAComponent, // child route component that the router renders
      },
      {
        path: 'child-b',
        component: ChildBComponent, // another child route component that the router renders
      },
    ],
  },
];
```

**Relative paths:**

```
<h2>First Component</h2>

<nav>
  <ul>
    <li><a routerLink="../second-component">Relative Route to second component</a></li>
  </ul>
</nav>
<router-outlet></router-outlet>

goToItems() {
  this.router.navigate(['items'], { relativeTo: this.route });
}
```

**Accessing query parameters and fragments:**

```
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { switchMap } from 'rxjs/operators';

constructor(private route: ActivatedRoute) {}

heroes$: Observable;
selectedId: number;
heroes = HEROES;

ngOnInit() {
  this.heroes$ = this.route.paramMap.pipe(
    switchMap(params => {
      this.selectedId = Number(params.get('id'));
      return this.service.getHeroes();
    })
  );
}
```

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';
import { Observable } from 'rxjs';

hero$: Observable;

   constructor(
      private route: ActivatedRoute,
      private router: Router    ) {}

   ngOnInit() {
      const heroId = this.route.snapshot.paramMap.get('id');
      this.hero$ = this.service.getHero(heroId);
   }

   gotoItems(hero: Hero) {
      const heroId = hero ? hero.id : null;
      // Pass along the hero id if available
      // so that the HeroList component can select that item.
      this.router.navigate(['/heroes', { id: heroId }]);
   }
```

**Preventing unauthorized access:**

Use route guards to prevent users from navigating to parts of an app without authorization. The following route guards are available in Angular:

CanActivate
CanActivateChild
CanDeactivate

ng generate guard your-guard

```
export class YourGuard implements CanActivate {
   canActivate(
      next: ActivatedRouteSnapshot,
      state: RouterStateSnapshot): boolean {
         // your    logic goes here
   }
}

{
   path: '/your-path',
   component: YourComponent,
   canActivate: [YourGuard],
}
```

## Router-outlet

The RouterOutlet is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

```
<router-outlet></router-outlet>
<!-- Routed components go here -->
```

## Router-Links:

To navigate as a result of some user action such as the click of an anchor tag, use RouterLink.

```
<h1>Angular Router</h1>
<nav>
   <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
   <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

## Observables:

They are basically used for asynchronous programmming. For instance, suppose there is an application in which after some lines of codes, there are some lines which send request to the server. For that time, the execution of other lines will be stopped that we don't want. Because it may take a long time also for that to be executed by the server. So, to overcome this problem, an obervable is set
in between which is noting but a lazy collection of multiple values over time. So the execution of those lines are not hampered and whenever we require those data, we just subscribe to it. After subscribing only, the data is reflected.

A handler for receiving observable notifications implements the Observer interface. It is an object that defines callback methods to handle the three types of notifications that an observable can send:

NOTIFICATION TYPE DESCRIPTION
next Required. A handler for each delivered value. Called zero or more times after execution starts.
//data handlers
error Optional. A handler for an error notification. An error halts execution of the observable instance.
complete Optional. A handler for the execution-complete notification. Delayed values can continue to be delivered to the next handler after execution is complete.
An observer object can define any combination of these handlers.

An Observable instance begins publishing values only when someone subscribes to it. You subscribe by calling the subscribe() method of the instance, passing an observer object to receive the notifications.

```
// Create simple observable that emits three values
const myObservable = of(1, 2, 3);

// Create observer object
const myObserver = {
   next: x => console.log('Observer got a next value: ' + x),
   error: err => console.error('Observer got an error: ' + err),
   complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object
myObservable.subscribe(myObserver);

// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification
```

**Error handling:**

Because observables produce values asynchronously, try/catch will not effectively catch errors. Instead, you handle errors by specifying an error callback on the observer. Producing an error also causes the

observable to clean up subscriptions and stop producing values. An observable can either produce values (calling the next callback), or it can complete, calling either the complete or error callback.

```
myObservable.subscribe({
   next(num) { console.log('Next num: ' + num)},
   error(err) { console.log('Received an error: ' + err)}
});
```

While subscribing any observable, we can store that value as subscription and unsubscribe it in onDestroy method. Only those subscriptions we need to unsubscribe which we are creating manually. For angular observables, angular does it on its own.

**Difference between promise and observables:**

Both promise and observables provide us with abstractions that help us deal with the asynchronous nature of our applications but still there is a difference.

1.Promises work with asynchronous operations and they either return us a single value (i.e the promise resolves) or an error message (i.e the promise rejects.)
2.A request initiated from a promise is not cancellable.

Few problems with promises:
1. What if I want to cancel a request to the api
2. wht if we want to retry a failed call.
3. As application gets bigger, promises becomes hard to manage.

Observables:
It is an array or a sequence of events over time. It has at least two participants, The creator (the data source) and the subscriber (subscription -> where data is being consumed) and comparable to promise, an observable can be cancelled.
it provides operators like map, forEach,reduce similar to an array. There are also some powerful operations like retry() or replay(),retryWhen(),delay().

## Forms in Angular:

It actually maintains a form like a javascript object.

Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.

Angular provides two different approaches to handling user input through forms: reactive and template-driven. Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

Choosing an approach:

Reactive forms and template-driven forms process and manage form data differently. Each approach offers different advantages.

**Reactive forms** provide direct, explicit access to the underlying forms object model. Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

**Template-driven forms** rely on directives in the template to create and manipulate the underlying object model. They are useful for adding a simple form to an app, such as an email list signup form. They're easy to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.

**Key differences:**

The table below summarizes the key differences between reactive and template-driven forms.

| | REACTIVE | TEMPLATE-DRIVEN |
|---|---|---|
| Setup of form model | Explicit, created in component class | Implicit, created by directives |
| Data model | Structured and immutable | Unstructured and mutable |
| Predictability | Synchronous | Asynchronous |
| Form validation | Functions | Directives |

**Template Driven template:**

```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form (ngSubmit)="onSubmit()" #f="ngForm">
        <div
          id="user-data"
          ngModelGroup="userData"
          #userData="ngModelGroup">
          <div class="form-group">
            <label for="username">Username</label>
            <input
              type="text"
              id="username"
              class="form-control"
              ngModel
              name="username"
              required>
          </div>
          <button
            class="btn btn-default"
            type="button"
            (click)="suggestUserName()">Suggest an Username</button>
          <div class="form-group">
            <label for="email">Mail</label>
            <input
              type="email"
              id="email"
              class="form-control"
              ngModel
              name="email"
              required
              email
              #email="ngModel">
            <span class="help-block" *ngIf="!email.valid && email.touched">Please enter a valid
email!</span>
          </div>
        </div>
        <p *ngIf="!userData.valid && userData.touched">User Data is invalid!</p>
```

```
            <div class="form-group">
              <label for="secret">Secret Questions</label>
              <select
                id="secret"
                class="form-control"
                [ngModel]="defaultQuestion"
                name="secret">
                <option value="pet">Your first Pet?</option>
                <option value="teacher">Your first teacher?</option>
              </select>
            </div>
            <div class="form-group">
              <textarea
                name="questionAnswer"
                rows="3"
                class="form-control"
                [(ngModel)]="answer"></textarea>
            </div>
            <p>Your reply: {{ answer }}</p>
            <div class="radio" *ngFor="let gender of genders">
              <label>
                <input
                  type="radio"
                  name="gender"
                  ngModel
                  [value]="gender"
                  required>
                {{ gender }}
              </label>
            </div>
            <button
              class="btn btn-primary"
              type="submit"
              [disabled]="!f.valid">Submit</button>
          </form>
        </div>
      </div>
      <hr>
      <div class="row" *ngIf="submitted">
        <div class="col-xs-12">
          <h3>Your Data</h3>
          <p>Username: {{ user.username }}</p>
          <p>Mail: {{ user.email }}</p>
          <p>Secret Question: Your first {{ user.secretQuestion }}</p>
          <p>Answer: {{ user.answer }}</p>
          <p>Gender: {{ user.gender }}</p>
        </div>
      </div>
    </div>
```

Typescript file:

AppModule.ts:

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

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';


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

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

component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('f', { static: false }) signupForm: NgForm;
  defaultQuestion = 'teacher';
  answer = '';
  genders = ['male', 'female'];
  user = {
    username: '',
    email: '',
    secretQuestion: '',
    answer: '',
    gender: ''
  };
  submitted = false;

  suggestUserName() {
    const suggestedName = 'Superuser';
    // this.signupForm.setValue({
    //   userData: {
    //     username: suggestedName,
    //     email: ''
    //   },
    //   secret: 'pet',
    //   questionAnswer: '',
    //   gender: 'male'
    // });
    this.signupForm.form.patchValue({
```

```
      userData: {
        username: suggestedName
      }
    });
  }

  // onSubmit(form: NgForm) {
  //     console.log(form);
  // }

  onSubmit() {
    this.submitted = true;
    this.user.username = this.signupForm.value.userData.username;
    this.user.email = this.signupForm.value.userData.email;
    this.user.secretQuestion = this.signupForm.value.secret;
    this.user.answer = this.signupForm.value.questionAnswer;
    this.user.gender = this.signupForm.value.gender;

    this.signupForm.reset();
  }
}
```

**Reactive Approach:**

```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form [formGroup]="signupForm" (ngSubmit)="onSubmit()">
        <div formGroupName="userData">
          <div class="form-group">
            <label for="username">Username</label>
            <input
              type="text"
              id="username"
              formControlName="username"
              class="form-control">
            <span
              *ngIf="!signupForm.get('userData.username').valid &&
signupForm.get('userData.username').touched"
              class="help-block">
              <span *ngIf="signupForm.get('userData.username').errors['nameIsForbidden']">This
name is invalid!</span>
              <span *ngIf="signupForm.get('userData.username').errors['required']">This field is
required!</span>

            </span>
          </div>
          <div class="form-group">
            <label for="email">email</label>
            <input
              type="text"
              id="email"
              formControlName="email"
              class="form-control">
            <span
```

```
                    *ngIf="!signupForm.get('userData.email').valid &&
signupForm.get('userData.email').touched"
                    class="help-block">Please enter a valid email!</span>
                </div>
            </div>
            <div class="radio" *ngFor="let gender of genders">
                <label>
                    <input
                        type="radio"
                        formControlName="gender"
                        [value]="gender">{{ gender }}
                </label>
            </div>
            <div formArrayName="hobbies">
                <h4>Your Hobbies</h4>
                <button
                    class="btn btn-default"
                    type="button"
                    (click)="onAddHobby()">Add Hobby</button>
                <div
                    class="form-group"
                    *ngFor="let hobbyControl of signupForm.get('hobbies').controls; let i = index">
                    <input type="text" class="form-control" [formControlName]="i">
                </div>
            </div>
            <span
                *ngIf="!signupForm.valid && signupForm.touched"
                class="help-block">Please enter valid data!</span>
            <button class="btn btn-primary" type="submit">Submit</button>
        </form>
    </div>
  </div>
</div>
```

Typescript file:

```
import { Component, OnInit } from '@angular/core';
import { FormArray, FormControl, FormGroup, Validators } from '@angular/forms';
import { Observable } from 'rxjs/Observable';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm: FormGroup;
  forbiddenUsernames = ['Chris', 'Anna'];

  constructor() {}

  ngOnInit() {
    this.signupForm = new FormGroup({
      'userData': new FormGroup({
```

```
            'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),
            'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails)
        }),
        'gender': new FormControl('male'),
        'hobbies': new FormArray([])
    });
    // this.signupForm.valueChanges.subscribe(
    //     (value) => console.log(value)
    // );
    this.signupForm.statusChanges.subscribe(
        (status) => console.log(status)
    );
    this.signupForm.setValue({
        'userData': {
            'username': 'Max',
            'email': 'max@test.com'
        },
        'gender': 'male',
        'hobbies': []
    });
    this.signupForm.patchValue({
        'userData': {
            'username': 'Anna',
        }
    });
}

onSubmit() {
    console.log(this.signupForm);
    this.signupForm.reset();
}

onAddHobby() {
    const control = new FormControl(null, Validators.required);
    (<FormArray>this.signupForm.get('hobbies')).push(control);
}

forbiddenNames(control: FormControl): {[s: string]: boolean} {
    if (this.forbiddenUsernames.indexOf(control.value) !== -1) {
        return {'nameIsForbidden': true};
    }
    return null;
}

forbiddenEmails(control: FormControl): Promise<any> | Observable<any> {
    const promise = new Promise<any>((resolve, reject) => {
        setTimeout(() => {
            if (control.value === 'test@test.com') {
                resolve({'emailIsForbidden': true});
            } else {
                resolve(null);
            }
        }, 1500);
    });
    return promise;
}
```

```
}
```

App module.ts:

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

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

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

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

## Transforming data using pipes:

Use pipes to transform strings, currency amounts, dates, and other data for display. Pipes are simple functions you can use in template expressions to accept an input value and return a transformed value. Pipes are useful because you can use them throughout your application, while only declaring each pipe once.

Angular provides built-in pipes for typical data transformations, including transformations for internationalization (i18n), which use locale information to format data. The following are commonly used built-in pipes for data formatting:

DatePipe: Formats a date value according to locale rules.
UpperCasePipe: Transforms text to all upper case.
LowerCasePipe: Transforms text to all lower case.
CurrencyPipe: Transforms a number to a currency string, formatted according to locale rules.
DecimalPipe: Transforms a number into a string with a decimal point, formatted according to locale rules.
PercentPipe: Transforms a number to a percentage string, formatted according to locale rules

**Using pipes in a template:**

```
<p>The hero's birthday is {{ birthday | date }}</p>

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988 -- since month parameter is zero-based
```

```
}
```

**Chaining pipes:**

```
The chained hero's birthday is
{{ birthday | date | uppercase}}

The chained hero's birthday is
{{    birthday | date:'fullDate' | uppercase}}
```

**Creating a pipe:**

ts:

```
import { Pipe, PipeTransform } from '@angular/core';
/*
  * Raise the value exponentially
  * Takes an exponent argument that defaults to 1.
  * Usage:
  *      value | exponentialStrength:exponent
  * Example:
  *      {{ 2 | exponentialStrength:10 }}
  *      formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
   transform(value: number, exponent?: number): number {
      return Math.pow(value, isNaN(exponent) ? 1 : exponent);
   }
}
```

template:

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

@Component({
   selector: 'app-power-booster',
   template: `
      <h2>Power Booster</h2>
      <p>Super power boost: {{2 | exponentialStrength: 10}}</p>
   `
})
export class PowerBoosterComponent { }
```

**A pure pipe** is only called when Angular detects a change in the value or the parameters passed to a pipe.

**An impure pipe** is called for every change detection cycle no matter whether the value or parameter(s) changes.

This is relevant for changes that are not detected by Angular

when you pass an array or object that got the content changed (but is still the same instance)
when the pipe injects a service to get access to other values, Angular doesn't recognize if they have changed.
In these cases you probably still want the pipe to be executed.

You should be aware that impure pipes are prone to be inefficient. For example when an array is passed into the pipe to filter, sort, ... then this work might be done every time change detection runs (which is quite

often especially with the default ChangeDetectionStrategy setting) event though the array might not even have changed. Your pipe should try to recognize this and for example return cached results.

## Communicating with backend services using HTTP:

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services

template:

```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-md-6 col-md-offset-3">
      <form #postForm="ngForm" (ngSubmit)="onCreatePost(postForm.value)">
        <div class="form-group">
          <label for="title">Title</label>
          <input
            type="text"
            class="form-control"
            id="title"
            required
            ngModel
            name="title"
          />
        </div>
        <div class="form-group">
          <label for="content">Content</label>
          <textarea
            class="form-control"
            id="content"
            required
            ngModel
            name="content"
          ></textarea>
        </div>
        <button
          class="btn btn-primary"
          type="submit"
          [disabled]="!postForm.valid"
        >
          Send Post
        </button>
      </form>
    </div>
  </div>
  <hr />
  <div class="row">
    <div class="col-xs-12 col-md-6 col-md-offset-3">
      <button class="btn btn-primary" (click)="onFetchPosts()">
        Fetch Posts
      </button>
      |
      <button
        class="btn btn-danger"
        [disabled]="loadedPosts.length < 1"
```

```
            (click)="onClearPosts()"
          >
            Clear Posts
          </button>
        </div>
      </div>
      <div class="row">
        <div class="col-xs-12 col-md-6 col-md-offset-3">
          <p *ngIf="loadedPosts.length < 1 && !isFetching">No posts available!</p>
          <ul class="list-group" *ngIf="loadedPosts.length >= 1 && !isFetching">
            <li class="list-group-item" *ngFor="let post of loadedPosts">
              <h3>{{ post.title }}</h3>
              <p>{{ post.content }}</p>
            </li>
          </ul>
          <p *ngIf="isFetching && !error">Loading...</p>
          <div class="alert alert-danger" *ngIf="error">
            <h1>An Error Occurred!</h1>
            <p>{{ error }}</p>
            <button class="btn btn-danger" (click)="onHandleError()">Okay</button>
          </div>
        </div>
      </div>
    </div>
```

App component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';
import { Subscription } from 'rxjs';

import { Post } from './post.model';
import { PostsService } from './posts.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  loadedPosts: Post[] = [];
  isFetching = false;
  error = null;
  private errorSub: Subscription;

  constructor(private http: HttpClient, private postsService: PostsService) {}

  ngOnInit() {
    this.errorSub = this.postsService.error.subscribe(errorMessage => {
      this.error = errorMessage;
    });

    this.isFetching = true;
    this.postsService.fetchPosts().subscribe(
```

```
      posts => {
        this.isFetching = false;
        this.loadedPosts = posts;
      },
      error => {
        this.isFetching = false;
        this.error = error.message;
      }
    );
  }

  onCreatePost(postData: Post) {
    // Send Http request
    this.postsService.createAndStorePost(postData.title, postData.content);
  }

  onFetchPosts() {
    // Send Http request
    this.isFetching = true;
    this.postsService.fetchPosts().subscribe(
      posts => {
        this.isFetching = false;
        this.loadedPosts = posts;
      },
      error => {
        this.isFetching = false;
        this.error = error.message;
        console.log(error);
      }
    );
  }

  onClearPosts() {
    // Send Http request
    this.postsService.deletePosts().subscribe(() => {
      this.loadedPosts = [];
    });
  }

  onHandleError() {
    this.error = null;
  }

  ngOnDestroy() {
    this.errorSub.unsubscribe();
  }
}
```

App module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
```

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

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

Post model:

```
export interface Post {
  title: string;
  content: string;
  id?: string;
}
```

Client Service:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { map, catchError } from 'rxjs/operators';
import { Subject, throwError } from 'rxjs';

import { Post } from './post.model';

@Injectable({ providedIn: 'root' })
export class PostsService {
  error = new Subject<string>();

  constructor(private http: HttpClient) {}

  createAndStorePost(title: string, content: string) {
    const postData: Post = { title: title, content: content };
    this.http
      .post<{ name: string }>(
        'https://ng-complete-guide-c56d3.firebaseio.com/posts.json',
        postData
      )
      .subscribe(
        responseData => {
          console.log(responseData);
        },
        error => {
          this.error.next(error.message);
        }
      );
  }

  fetchPosts() {
    let searchParams = new HttpParams();
    searchParams = searchParams.append('print', 'pretty');
    searchParams = searchParams.append('custom', 'key');
    return this.http
      .get<{ [key: string]: Post }>(
```

```
            'https://ng-complete-guide-c56d3.firebaseio.com/posts.json',
            {
              headers: new HttpHeaders({ 'Custom-Header': 'Hello' }),
              params: searchParams
            }
          )
          .pipe(
            map(responseData => {
              const postsArray: Post[] = [];
              for (const key in responseData) {
                if (responseData.hasOwnProperty(key)) {
                  postsArray.push({ ...responseData[key], id: key });
                }
              }
              return postsArray;
            }),
            catchError(errorRes => {
              // Send to analytics server
              return throwError(errorRes);
            })
          );
    }

    deletePosts() {
      return this.http.delete(
        'https://ng-complete-guide-c56d3.firebaseio.com/posts.json'
      );
    }
}
```