Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications.

Angular is an application-design framework and development platform.

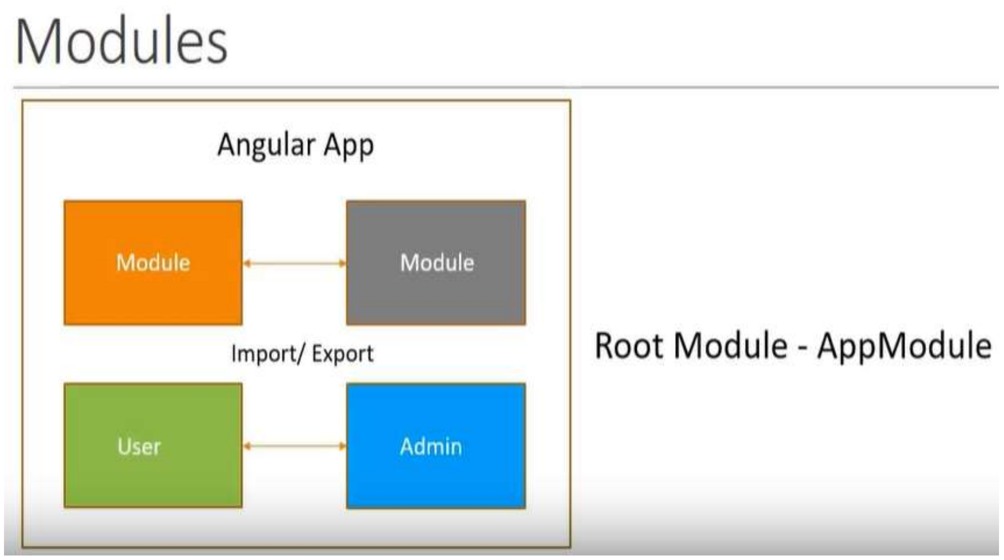
Angular is a development platform, built on TypeScript. As a platform, Angular includes:

- A component-based framework for building scalable web applications
- A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more
- A suite of developer tools to help you develop, build, test, and update your code

First building block is module.

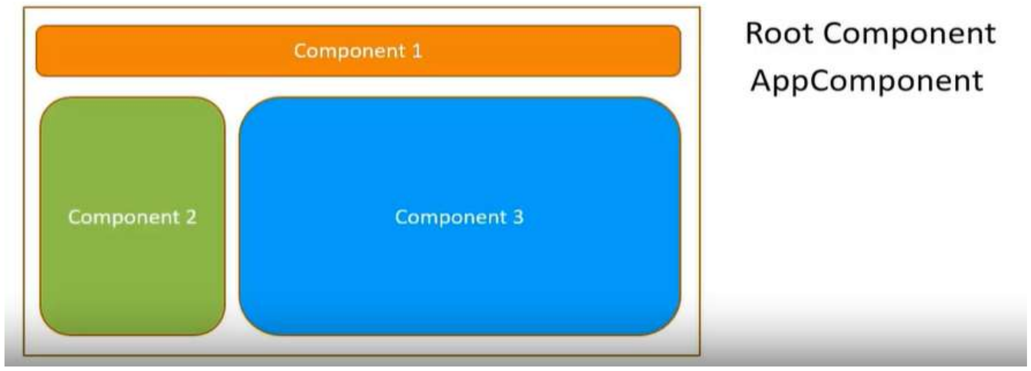
Angular application is a collection of modules

Every angular app contains at least one root module it's called app module



Each module intern made up of components and services.

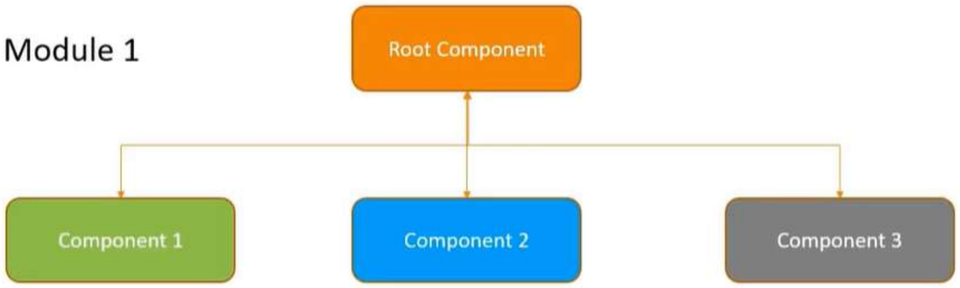
Components



Root Component
AppComponent

Components contd.

Module 1



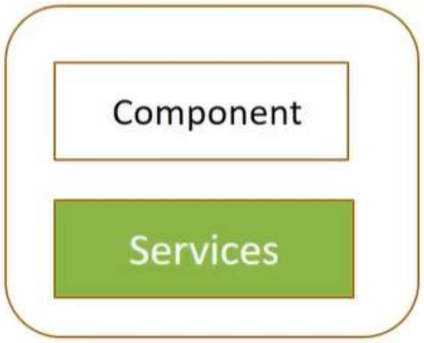
HTML Template

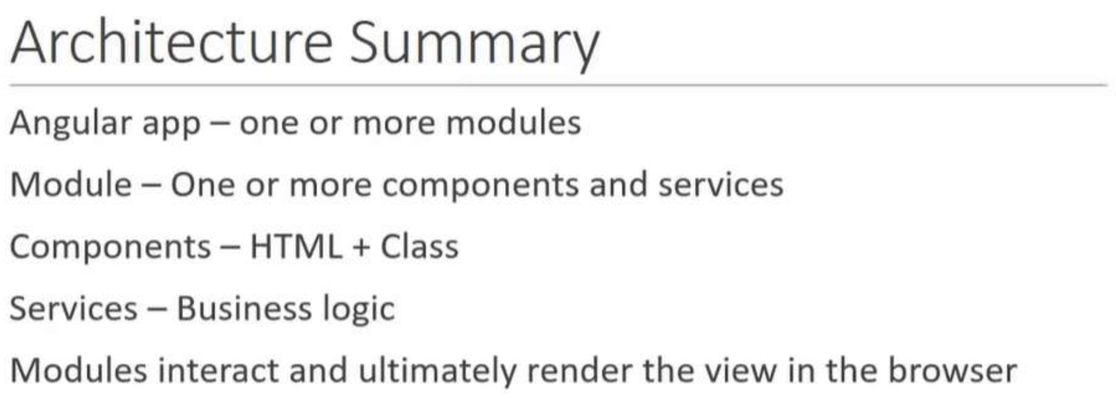
Class

Services

Business logic

Module





Architecture Summary

Angular app – one or more modules

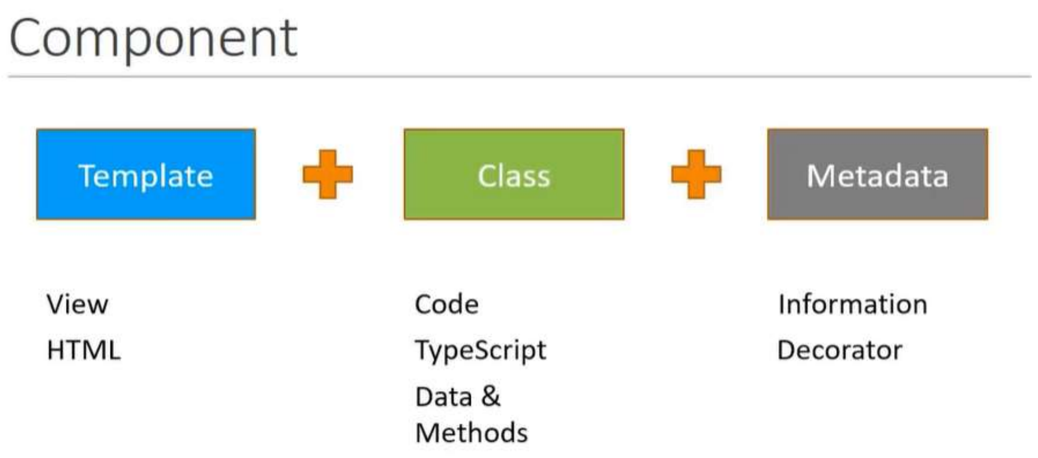
Module – One or more components and services

Components – HTML + Class

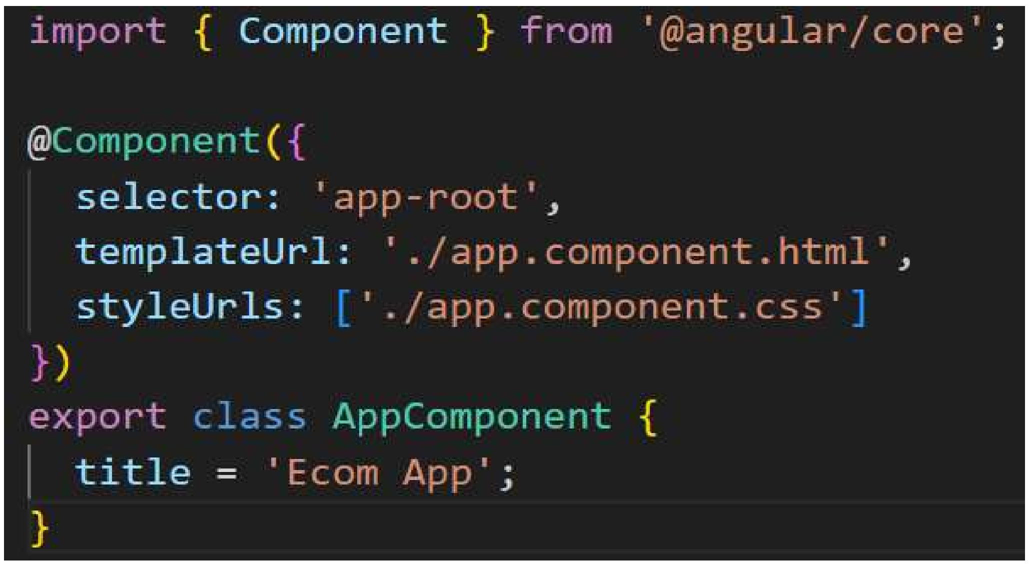
Services – Business logic

Modules interact and ultimately render the view in the browser

Components:



Component



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

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

How to create a new component?

- ➢ Ng g c test

The above component creates one folder under src.

3 files are created

Test-component.ts

Test-component.html

Test-component.css

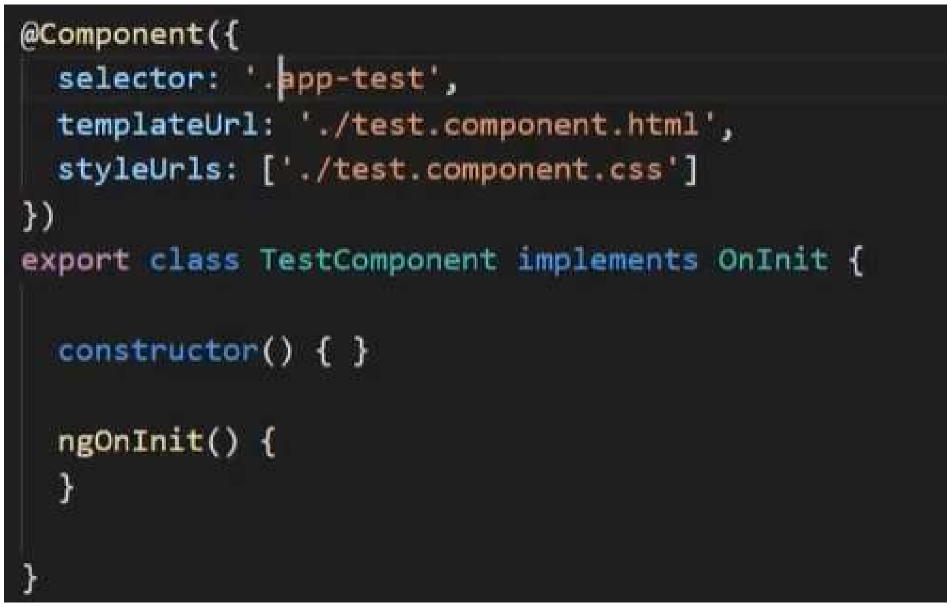
How to include test component in app component?

Two ways:

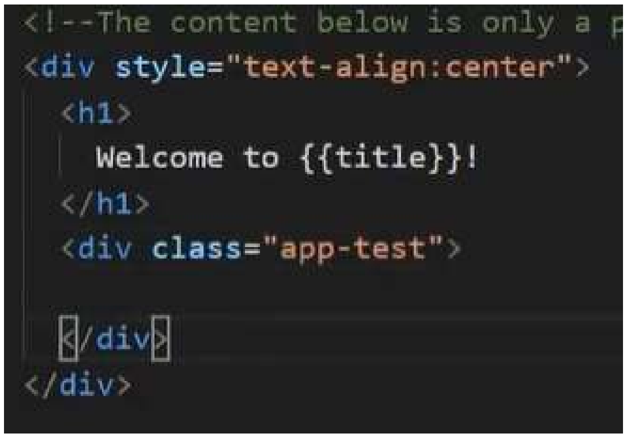
1. Using selector defined in component decorator

```
<app-test></app-test>
```

2. Using class name

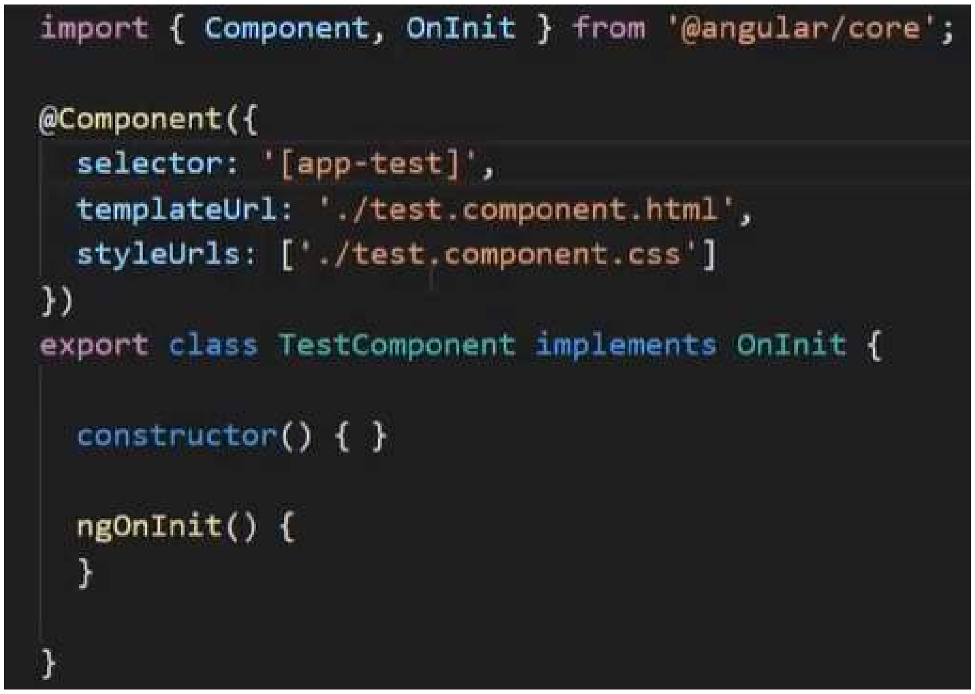


```
@Component({
  selector: '.app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

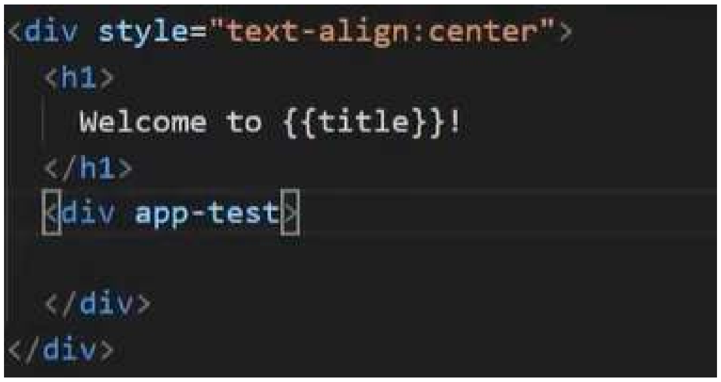


```
<!--The content below is only a p
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!
  </h1>
  <div class="app-test">

  </div>
</div>
```

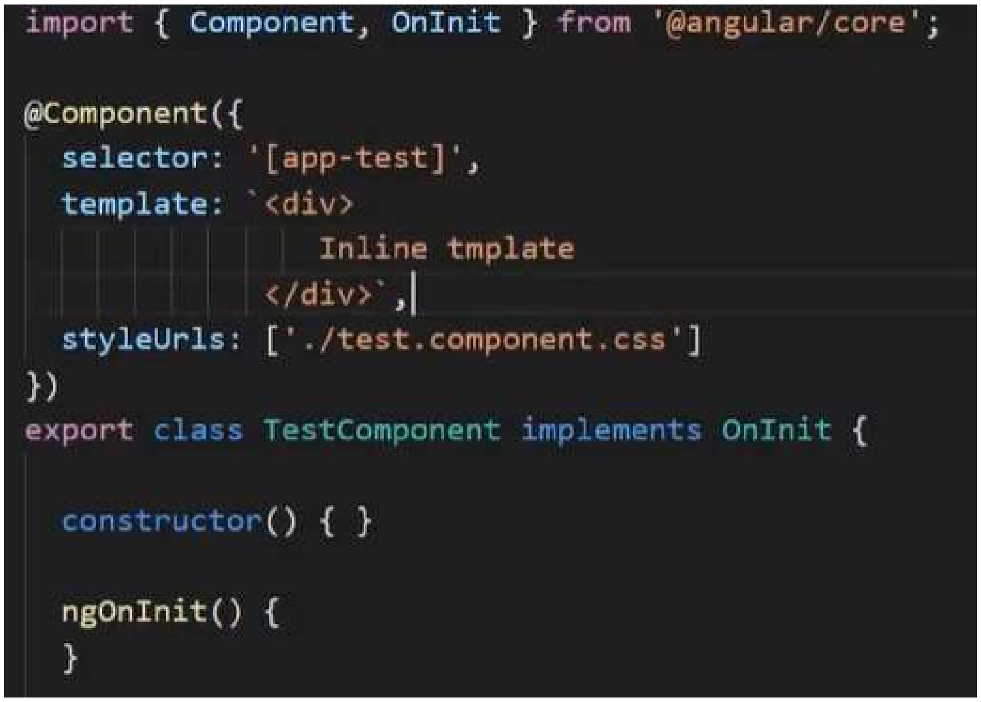
3. Approach

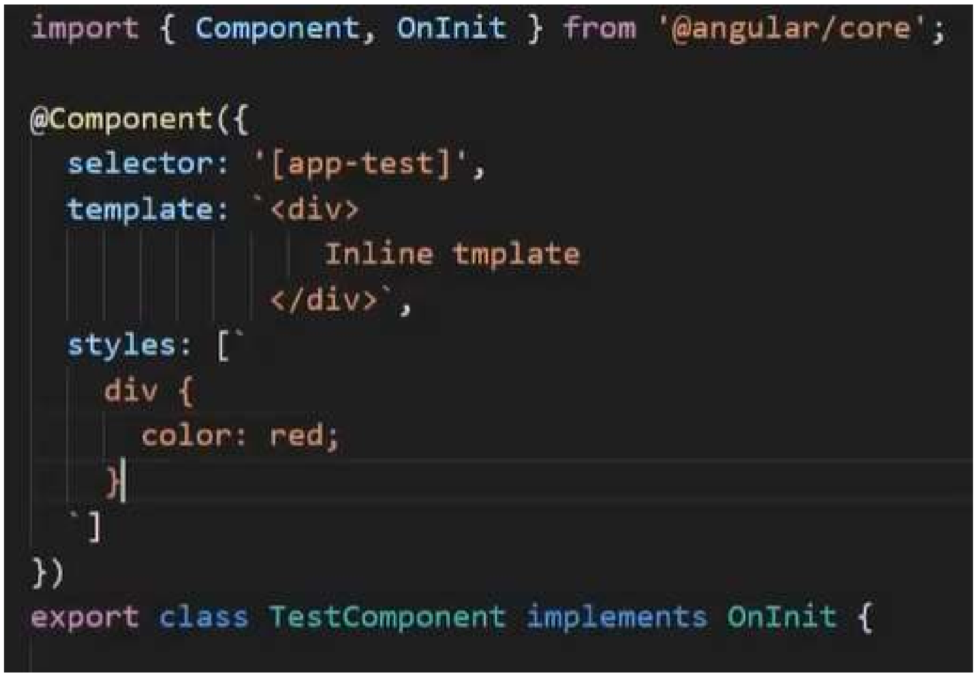


```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: '[app-test]',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```



```
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!
  </h1>
  <div app-test>

  </div>
</div>
```


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

@Component({
  selector: '[app-test]',
  template: `<div>
                Inline tmplate
              </div>`,
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
```


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

@Component({
  selector: '[app-test]',
  template: `<div>
                Inline tmplate
              </div>`,
  styles: [`
    div {
      color: red;
    }
  `]
})
export class TestComponent implements OnInit {
```

Data Binding:

Data binding is a technique, where the data stays in sync between the component and the view. Whenever the user updates the data in the view, Angular updates the component. When the component gets new data, the Angular updates the view.

There are many uses of data binding. You can show models to the user, dynamically Change element style, respond to user events, etc

The data binding in Angular can be broadly classified into two groups. *One way binding* or *two-way binding*

One way binding

In one-way binding data flows from one direction. Either from view to component or from component to view.

From Component to View

To bind data from component to view, we make use of Interpolation & Property Binding.

Interpolation:

The Angular uses the `{{ }}` (double curly braces) in the template to denote the interpolation.

Syntax:

```
{{ templateExpression }}
```

The content inside the double braces is called **Template Expression**

Example:

```

Welcome,  {{firstName}} {{lastName}}

```

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  firstName= 'Sachin';
  lastName="Tendulkar"
}

```

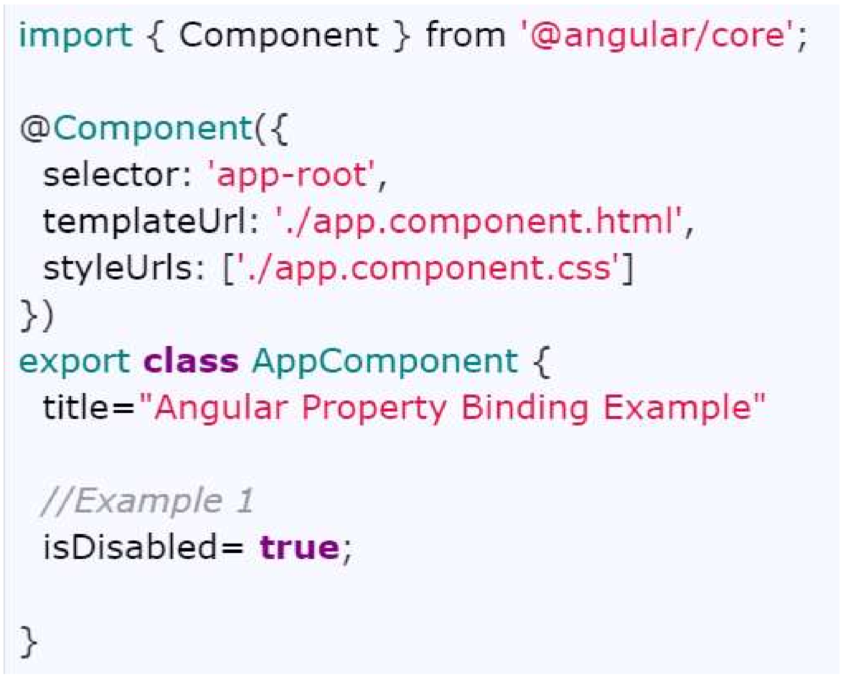
Property Binding:

Using property binding You can set the properties such as class, href, src, textContent, etc using property binding.

Syntax:

```
[binding-target]="binding-source"
```

```
<h1 [innerText]="title"></h1>
<h2>Example 1</h2>
<button [disabled]="isDisabled">I am disabled</button>
```



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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title="Angular Property Binding Example"

  //Example 1
  isDisabled= true;

}
```

Some more examples:

```
<h3>Binding to innerHTML with HTML tags:</h3>
<p [innerHTML]="text1"></p>
<div [innerHTML]="text2"></div>
```

```
<h3>Image:</h3>
<img [src]="itemImageUrl">
```

```
  isDisabled = true;
  title = "<b>Sample Heading</b>";

  text1="The <b>Angular</b> is printed in bold"
  text2="<p>This is first para</p><p>This is second para</p> "

  itemImageUrl="https://angular.io/assets/images/logos/angular/logo-
nav@2x.png";
```

Style binding in Angular

We can set the inline styles of a HTML element using the style binding in angular.

You can add styles conditionally to an element, hence creating a dynamically styled element.

Syntax

```
[style.style-property] = "style-value"
```

Example:

```
<p [style.color]="'red'">Give me red</p>
```

Some more examples:

```
<p>Setting the background color of a paragraph</p>
<p [style.background-color]="'grey'">some paragraph with grey background</p>

<p>Setting the border style of a button.</p>
<button [style.border]="'5px solid yellow'">Save</button>

<p>Setting the units</p>
<button [style.font-size.px]="'20'" >Big Button</button>
```

Inside component:

```
 status: string = 'error';

  color: string= 'red';

  getColor() {
    return 'yellow';
  }
```

```
<p>Conditionally setting the styles</p>
<button [style.color]="status=='error' ? 'red': 'blue'">Button 1</button>
```

```
<p>Setting Multiple styles</p>
<p [style.color]="getColor()"
   [style.font-size.px]="'20'"
   [style.background-color]="status=='error' ? 'red': 'blue'">
   paragraph with multiple styles
</p>
```

Angular ngStyle Directive

The Angular ngStyle directive allows us to set the many inline style of a HTML element using an expression. The expression can be evaluated at run time allowing us to dynamically change the style of our HTML element.

ngStyle Syntax

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

Example:

```
<some-element [ngStyle]="{'font-size': '20px'}">Set Font size to 20px</some-element>
```

```
<p>ngStyle multiple attributes</p>
<p [ngStyle]="{'color': 'purple',
               'font-size': '20px',
               'font-weight': 'bold'}">
     Multiple styles
</p>
```

```
<p>Using object from Controller</p>
<div [ngStyle]="styleClass">size & Color</div>
```

```
 export class StylebindingComponent implements OnInit {

  status: string = 'error';

  color: string= 'red';

  getColor() {
    return 'yellow';
  }

  styleClass: StyleClass = new StyleClass();
  constructor() { }

  ngOnInit() {
  }

}

class StyleClass {
  'color': string= 'blue';
  'font-size.px': number= 20;
  'font-weight': string= 'bold';
}
```

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

```
Syntax:
<element [ngStyle]="{'styleName.unit': widthExp}">...</element>

Example:
<some-element[ngStyle]="{'font-size.em': '3'}">...</some-element>
```

From View to Component

Event Binding

Event binding allows us to bind events such as keystroke, clicks, hover etc to a method in component.

Syntax:

```
(target-event)="TemplateStatement"
```

Example:

```
<button (click)="onSave()">Save</button>
```

Example:

```
<h1 [innerText]="title"></h1>

<h2>Example 1</h2>
<button (click)="clickMe()">Click Me</button>
<p>You have clicked {{clickCount}}</p>
```

```
clickCount=0
 clickMe() {
   this.clickCount++;
 }
```

Adding bootstrap:

```
npm install bootstrap -save
```

src/style.css

```
@import "~bootstrap/dist/css/bootstrap.css";
```

`Or`

```
"styles": [
   "node_modules/bootstrap/dist/css/bootstrap.min.css",
   "src/styles.css"
],
```

2nd approach:

```
npm install bootstrap -save

npm install jquery --save

npm install popper.js --save
```

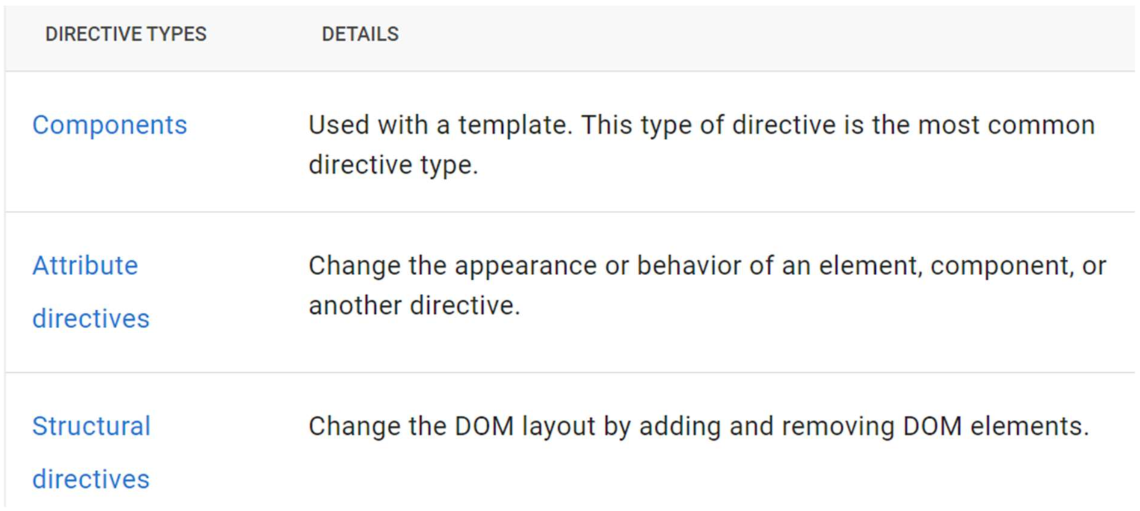
angular.json

```
....

      "styles": [

"node_modules/bootstrap/dist/css/bootstrap.min.css",

        "src/styles.css"

      ],

      "scripts": [

          "node_modules/jquery/dist/jquery.min.js",

"node_modules/bootstrap/dist/js/bootstrap.min.js"

      ]

.....
```

Directives:

Directives are classes that add additional behaviour to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

There are 3 types of Directives:

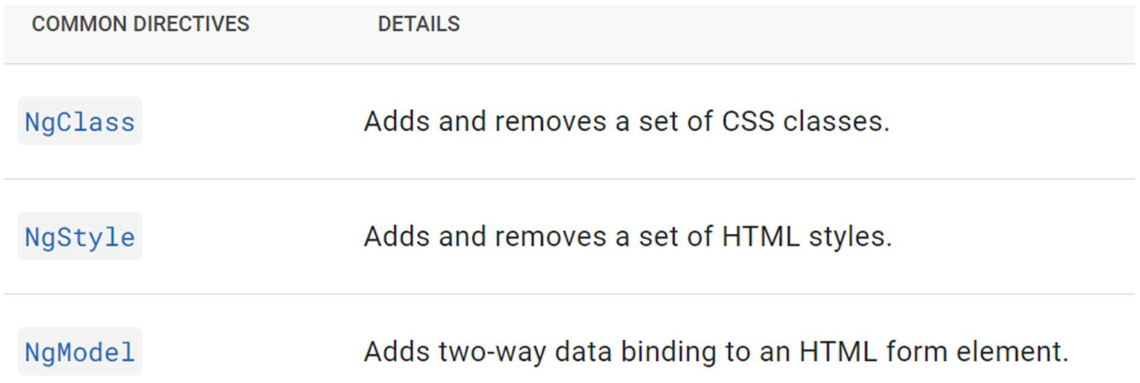


DIRECTIVE TYPES	DETAILS
Components	Used with a template. This type of directive is the most common directive type.
Attribute directives	Change the appearance or behavior of an element, component, or another directive.
Structural directives	Change the DOM layout by adding and removing DOM elements.

Built-in Attribute Directives:

Attribute directives listen to and modify the behaviour of other HTML elements, attributes, properties, and components.

Many attribute directives are defined through modules such as the `CommonModule`, `RouterModule` and `FormsModule`.



COMMON DIRECTIVES	DETAILS
`NgClass`	Adds and removes a set of CSS classes.
`NgStyle`	Adds and removes a set of HTML styles.
`NgModel`	Adds two-way data binding to an HTML form element.

Built-in Structural Directives:

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

COMMON BUILT-IN STRUCTURAL DIRECTIVES	DETAILS
`NgIf`	Conditionally creates or disposes of subviews from the template.
`NgFor`	Repeat a node for each item in a list.
`NgSwitch`	A set of directives that switch among alternative views.

ngIf Syntax :

```
<p *ngIf="condition">
   content to render when the condition is true
</p>
```

If the **ngIf** expression evaluates to false, then the Angular removes the entire element from the DOM. If true, it will insert the element into the DOM.

Example:

The following code uses ngIf to conditionally render a button based on a boolean variable `showButton`. The button is shown only when the showButton is true.

```
<button *ngIf="showButton">Click Me!</button>
```

ngIf else

The `ngIf` allows us to define optional `else` block using the `ng-template`

```
<div *ngIf="condition; else elseBlock">
   content to render, when the condition is true
</div>

<ng-template #elseBlock>
   content to render, when the condition is false
</ng-template>
```

ngIf then else

```
<div *ngIf="condition; then thenBlock else elseBlock">
   This content is not shown
</div>

<ng-template #thenBlock>
   content to render when the condition is true.
</ng-template>

<ng-template #elseBlock>
   content to render when condition is false.
</ng-template>
```

Example :

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = 'ngIf Example' ;
  showMe: boolean;
}
```

```
<h1>Simple example of ngIf </h1>


<div class="row">
 Show <input type="checkbox" [(ngModel)]="showMe" />
</div>
```

```
<h1>ngIf </h1>

<p *ngIf="showMe">
 ShowMe is checked
</p>
<p *ngIf="!showMe">
 ShowMe is unchecked
</p>
```

```
<h1>ngIf Else</h1>

<p *ngIf="showMe; else elseBlock1">
 ShowMe is checked
</p>

<ng-template #elseBlock1>
  <p>ShowMe is unchecked Using elseBlock</p>
</ng-template>
```

```
<h1>ngIf then else</h1>

<p *ngIf="showMe; then thenBlock2 else elseBlock2">
  This is not rendered
</p>

<ng-template #thenBlock2>
  <p>ShowMe is checked Using thenblock</p>
</ng-template>

<ng-template #elseBlock2>
  <p>ShowMe is unchecked Using elseBlock</p>
</ng-template>
```

ngSwitch:

Syntax:

```
<container_element [ngSwitch]="switch_expression">
   <inner_element *ngSwitchCase="match_expresson_1">…</inner_element>
   <inner_element *ngSwitchCase="match_expresson_2">…</inner_element>
   <inner_element *ngSwitchCase="match_expresson_3">…</inner_element>
   <inner_element *ngSwitchDefault>…</element>
</container_element>
```

Example 1:

```
num: number= 0;
```

```
<div class='card'>
 <div class='card-header'>
  ngSwitch Example
 </div>
 <div class="card-body">
  Input string : <input type='text' [(ngModel)]="num" />

   <div [ngSwitch]="num">
    <div *ngSwitchCase="'1'">One</div>
    <div *ngSwitchCase="'2'">Two</div>
    <div *ngSwitchCase="'3'">Three</div>
    <div *ngSwitchCase="'4'">Four</div>
    <div *ngSwitchCase="'5'">Five</div>
    <div *ngSwitchDefault>This is Default</div>
   </div>
 </div>
</div>
```

Example 2:

```
export class Switchdemo2Component{

  items: Item [] = [
    {name: 'One', val: 1},
    {name: 'Two', val: 2},
    {name: 'Three', val: 3},
    {name: 'Four', val: 3},
    {name: 'Five', val: 3}
  ];
  selectedValue1: string = 'One';
  selectedValue2: string = 'One';
  selectedValue3: string = 'One';
  selectedValue4: string = 'One';
}

class Item {
  name: string;
  val: number;
}
```

```
<div class='card'>
    <div class='card-header'>
      Multiple ngSwitchCase with same condition
    </div>

    <div class="card-body">
      <select [(ngModel)]="selectedValue1">
        <option *ngFor="let item of items;"
[value]="item.name">{{item.name}}</option>
      </select>

      <div [ngSwitch]="selectedValue1">
        <p *ngSwitchCase="'One'">One is Selected</p>
        <p *ngSwitchCase="'Two'">Two is Selected</p>
        <p *ngSwitchCase="'Two'">Two Again used in another element</p>
        <p *ngSwitchDefault>This is Default</p>
      </div>
    </div>

  </div>
```

---

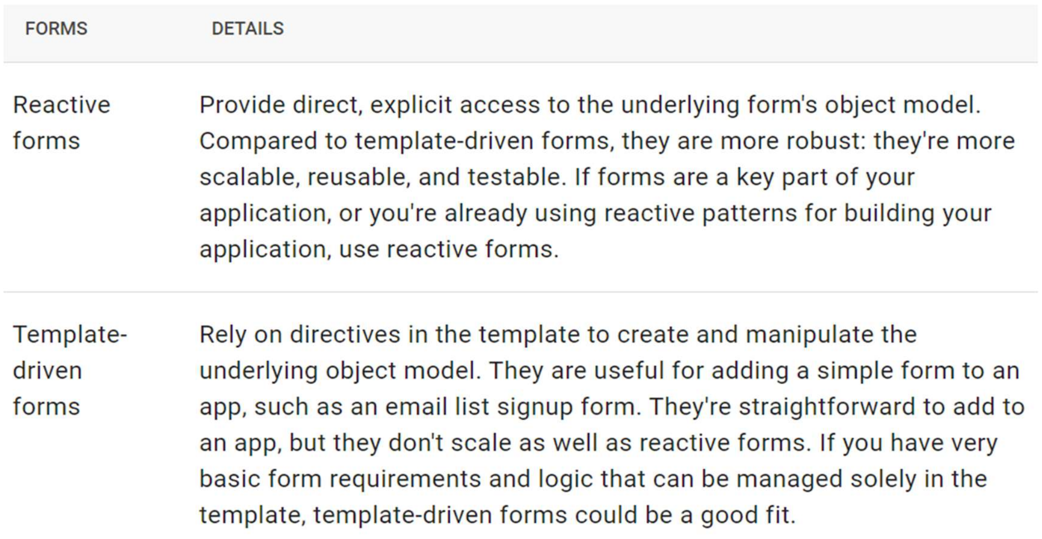
forms in Angular

Angular provides two different approaches to handling user input through forms:

1. reactive forms
2. template-driven forms

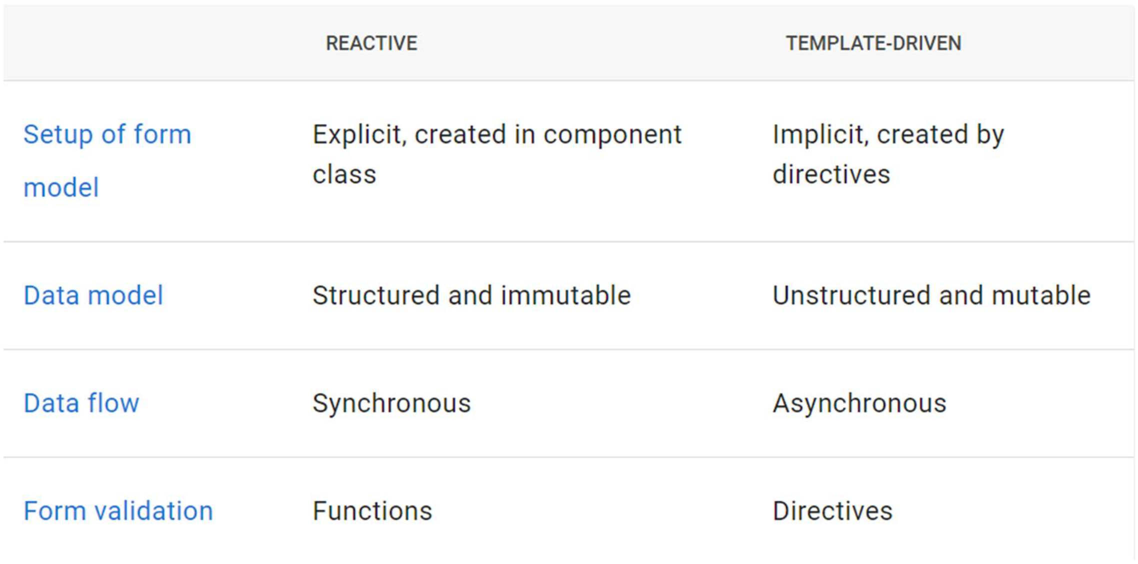
Choosing an approach:

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



FORMS	DETAILS
Reactive forms	Provide direct, explicit access to the underlying form's 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 straightforward 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



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

Template-driven forms

1. The form is set up using ngForm directive
2. controls are set up using the ngModel directive
3. ngModel also provides the two-way data binding
4. The Validations are configured in the template via directives

Template-driven forms are,

1. Contains little code in the component class
2. Easier to set up

While they are,

1. Difficult to add controls dynamically
2. Unit testing is a challenge

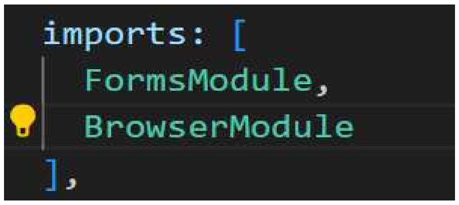
To work with Template-driven forms, we must import the FormsModule

Open the app.module.ts and add the import { FormsModule } from '@angular/forms'; to it.

And also add the FormsModule to the *imports metadata property array*

app.module.ts

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



```
imports: [
  FormsModule,
  BrowserModule
],
```

Example:

```
<form>

    <p>
      <label for="firstname">First Name</label>
      <input type="text" id="firstname" name="firstname">
    </p>

    <p>
      <label for="lastname">Last Name</label>
      <input type="text" id="lastname" name="lastname">
    </p>

    <p>
      <label for="email">Email </label>
      <input type="text" id="email" name="email">
    </p>

    <p>
```

```
      <label for="gender">Geneder</label>
      <input type="radio" value="male" id="gender" name="gender"> Male
      <input type="radio" value="female" id="gender" name="gender"> Female
    </p>

    <p>
      <button type="submit">Submit</button>
    </p>

  </form>
```

ngForm

Once, we have a form with few form elements, the angular automatically converts it into a Template-driven form. This is done by the `ngForm` directive.

The `ngForm` directive is what makes the Angular template-driven forms work. But we do not need to add it explicitly. Angular adds it automatically

When we include `FormsModule`, the Angular is going to look out for any `<form>` tag in our HTML template. Angular does this via `ngForm` [directive](). `ngForm` directive automatically detects the `<form>` tag and automatically binds to it.

We can export the `ngForm` instance into a local template variable using `ngForm` as the key (ex: `#contactForm="ngForm"`). This allows us to access the many properties and methods of `ngForm` using the template variable `contactForm`

Hence, update the `form` element as shown below.

```
<form #contactForm="ngForm">
```

FormControl

The `FormControl` is the basic building block of the [Angular Forms](). It represents a single input field in an [Angular form](). The [Angular Forms Module]() binds the input element to a `FormControl`. We use the `FormControl` instance to track the value, user interaction and validation status of an individual form element. Each individual `Form` element is a `FormControl`

We have four form elements in our HTML template.

We need to bind them to FormControl instance. We do this by using the ngModel directive. Add the ngModel directive to each control as shown below.

```
<input type="text" name="firstname" ngModel>
```

ngModel will use the name attribute to create the FormControl instance for each of the Form field it is attached.

Submit Form

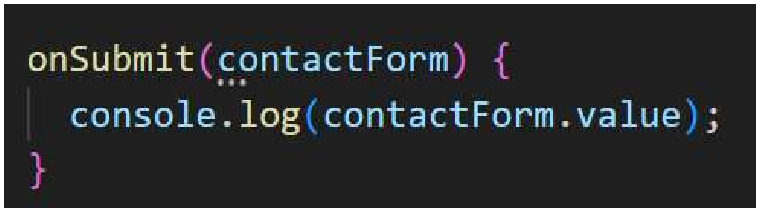
We use the ngSubmit event, to submit the form data to the component class.

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">
```

Final form :

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">
    <p>
      <label for="firstname">First Name</label>
      <input type="text" name="firstname" ngModel>
    </p>

    <p>
      <label for="lastname">Last Name</label>
      <input type="text" name="lastname" ngModel>
    </p>

    <p>
      <label for="email">Email </label>
      <input type="text" id="email" name="email" ngModel>
    </p>

    <p>
      <label for="gender">Geneder</label>
      <input type="radio" value="male" name="gender" ngModel> Male
      <input type="radio" value="female" name="gender" ngModel> Female
    </p>

    <p>
      <button type="submit">Submit</button>
    </p>

  </form>
```

Receive Form Data



```
onSubmit(contactForm) {
  console.log(contactForm.value);
}
```

Local Variable

We can assign the `ngForm`,`FormControl` or `FormGroup` instance to a template local variable. This allows us to check the status of the form like whether the form is `valid`, `submitted`, and `value` of the form elements, etc

ngForm

We have access to the `ngForm` instance via the local template variable `#contactForm`.

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">
```

Now, we can make use of some of the properties & methods to know the status of form. For Example

```
<p>
  <button type="submit">Submit</button>
</p>

<pre>Value : {{contactForm.value | json }} </pre>
<pre>Valid : {{contactForm.valid}} </pre>
<pre>Touched : {{contactForm.touched  }} </pre>
<pre>Submitted : {{contactForm.submitted  }} </pre>
```

`value` : The value property returns the object containing the value of every FormControl

`valid` : Returns true if the form is Valid else returns false.

`touched` : True if the user has entered a value in at least in one field.

`submitted` : Returns true if the form is submitted. else false.

FormControl

Similarly, we can also get access to the FormControl instance by assigning the ngModel to a local variable as shown below

```
<input type="text" name="firstname" #fname="ngModel" ngModel>
```

Now, the variable #fname holds the reference to the firstname FormControl. We can then access the properties of FormControl like value, valid, isvalid, tocuhed etc

```
<p>
  <label for="firstname">First Name </label>
  <input type="text" name="firstname" #fname="ngModel" ngModel>
</p>

<pre>Value    : {{fname.value}} </pre>
<pre>valid    : {{fname.valid}} </pre>
<pre>invalid  : {{fname.invalid}} </pre>
<pre>touched  : {{fname.touched}} </pre
```

value : Returns the current value of the control

valid : Returns true if the value is Valid else false

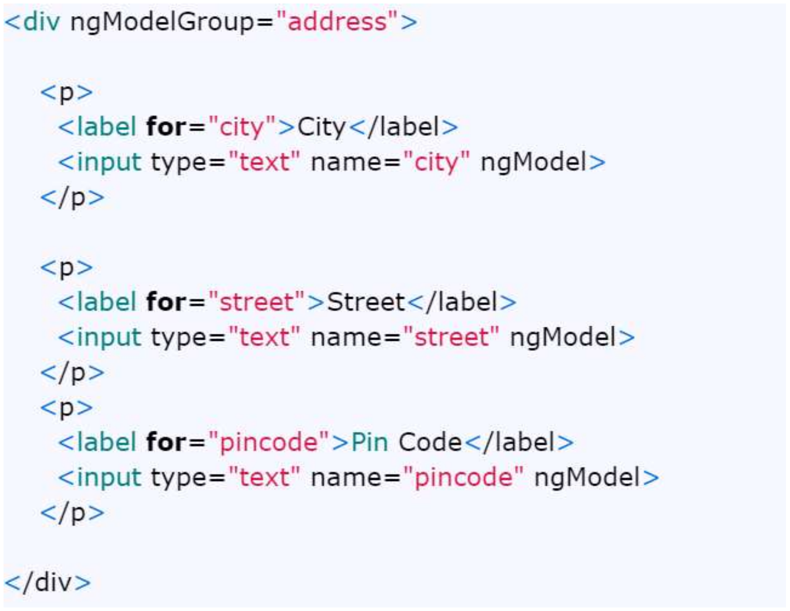
invalid : True if the value is invalid else false

touched : Returns true if the value is entered in the element

Nested FormGroup

The FormGroup is a collection of FormControl. It can also contain other FormGroup's.

We can add new FormGroup using the ngModelGroup directive.

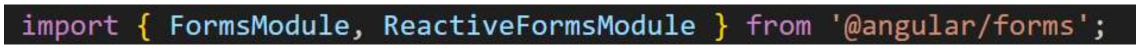


```html
<div ngModelGroup="address">

  <p>
    <label for="city">City</label>
    <input type="text" name="city" ngModel>
  </p>

  <p>
    <label for="street">Street</label>
    <input type="text" name="street" ngModel>
  </p>
  <p>
    <label for="pincode">Pin Code</label>
    <input type="text" name="pincode" ngModel>
  </p>

</div>
```

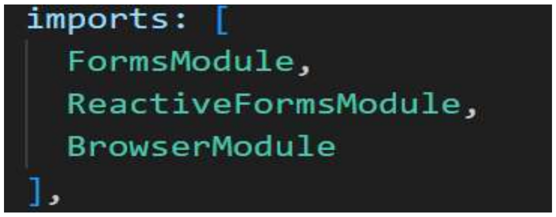
Reactive Forms

To build reactive forms, first, we need to import ReactiveFormsModule

In app.module.ts



```typescript
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
```



```typescript
imports: [
  FormsModule,
  ReactiveFormsModule,
  BrowserModule
],
```

In the template-driven approach, we used ngModel & ngModelGroup directive on the HTML elements. The FormsModule automatically creates the FormGroup & FormControl instances from the HTML template. This happens behind the scene.

In the Reactive Forms approach, It is our responsibility to build the Model using FormGroup, FormControl and FormArray.

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

FormGroup represents a collection of form controls. It can also contain other FormGroups and FormArrays. In fact, an angular form is a `FormGroup`.